USING OPENCL

Advances in Parallel Computing

This book series publishes research and development results on all aspects of parallel computing. Topics may include one or more of the following: high-speed computing architectures (Grids, clusters, Service Oriented Architectures, etc.), network technology, performance measurement, system software, middleware, algorithm design, development tools, software engineering, services and applications.

Series Editor:
Professor Dr. Gerhard R. Joubert

Volume 21

Recently published in this series

Vol. 20. I. Foster, W. Gentzsch, L. Grandinetti and G.R. Joubert (Eds.), High Performance Computing: From Grids and Clouds to Exascale
Vol. 19. B. Chapman, F. Desprez, G.R. Joubert, A. Lichnewsky, F. Peters and T. Priol (Eds.), Parallel Computing: From Multicores and GPU's to Petascale
Vol. 18. W. Gentzsch, L. Grandinetti and G. Joubert (Eds.), High Speed and Large Scale Scientific Computing
Vol. 17. F. Xhafa (Ed.), Parallel Programming, Models and Applications in Grid and P2P Systems
Vol. 16. L. Grandinetti (Ed.), High Performance Computing and Grids in Action
Vol. 15. C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr and F. Peters (Eds.), Parallel Computing: Architectures, Algorithms and Applications

Volumes 1–14 published by Elsevier Science.

ISSN 0927-5452 (print)
ISSN 1879-808X (online)

Using OpenCL

Programming Massively Parallel Computers

Janusz Kowalik

16477-107th PL NE, Bothell, WA 98011, USA

and

Tadeusz Puźniakowski

UG, MFI, Wit Stwosz Street 57, 80-952 Gdańsk, Poland

IOS Press

Amsterdam • Berlin • Tokyo • Washington, DC

ISBN 978-1-61499-029-1 (print)
ISBN 978-1-61499-030-7 (online)
Library of Congress Control Number: 2012932792
doi:10.3233/978-1-61499-030-7-i

Publisher
IOS Press BV
Nieuwe Hemweg 6B
1013 BG Amsterdam
Netherlands
fax: +31 20 687 0019
e-mail: order@iospress.nl

Distributor in the USA and Canada
IOS Press, Inc.
4502 Rachael Manor Drive
Fairfax, VA 22032
USA
fax: +1 703 323 3668
e-mail: iosbooks@iospress.com

PRINTED IN THE NETHERLANDS

This book is dedicated to Alex, Bogdan and Gabriela
with love and consideration.

Preface

This book contains the most important and essential information required for designing correct and efficient OpenCL programs. Some details have been omitted but can be found in the provided references. The authors assume that readers are familiar with basic concepts of parallel computation, have some programming experience with C or C++ and have a fundamental understanding of computer architecture. In the book, all terms, definitions and function signatures have been copied from official API documents available on the page of the OpenCL standards creators.

The book was written in 2011, when OpenCL was in transition from its infancy to maturity as a practical programming tool for solving real-life problems in science and engineering. Earlier, the Khronos Group successfully defined OpenCL specifications, and several companies developed stable OpenCL implementations ready for learning and testing. A significant contribution to programming heterogeneous computers was made by NVIDIA which created one of the first working systems for programming massively parallel computers – CUDA. OpenCL has borrowed from CUDA several key concepts. At this time (fall 2011), one can install OpenCL on a heterogeneous computer and perform meaningful computing experiments. Since OpenCL is relatively new, there are not many experienced users or sources of practical information. One can find on the Web some helpful publications about OpenCL, but there is still a shortage of complete descriptions of the system suitable for students and potential users from the scientific and engineering application communities.

Chapter 1 provides short but realistic examples of codes using MPI and OpenMP in order for readers to compare these two mature and very successful systems with the fledgling OpenCL. MPI used for programming clusters and OpenMP for shared memory computers, have achieved remarkable worldwide success for several reasons. Both have been designed by groups of parallel computing specialists that perfectly understood scientific and engineering applications and software development tools. Both MPI and OpenMP are very compact and easy to learn. Our experience indicates that it is possible to teach scientists or students whose disciplines are other than computer science how to use MPI and OpenMP in a several hours time. We hope that OpenCL will benefit from this experience and achieve, in the near future, a similar success.

Paraphrasing the wisdom of Albert Einstein, we need to simplify OpenCL as much as possible but not more. The reader should keep in mind that OpenCL will be evolving and that pioneer users always have to pay an additional price in terms of initially longer program development time and suboptimal performance before they gain experience. The goal of achieving simplicity for OpenCL programming requires an additional comment. OpenCL supporting heterogeneous computing offers us opportunities to select diverse parallel processing devices manufactured by different vendors in order to achieve near-optimal or optimal performance. We can select multi-core CPUs, GPUs, FPGAs and other parallel processing devices to fit the problem we want to solve. This flexibility is welcomed by many users of HPC technology, but it has a price.

Programming heterogeneous computers is somewhat more complicated than writing programs in conventional MPI and OpenMP. We hope this gap will disappear as OpenCL matures and is universally used for solving large scientific and engineering problems.

Acknowledgements

It is our pleasure to acknowledge assistance and contributions made by several persons who helped us in writing and publishing the book.

First of all, we express our deep gratitude to Prof. Gerhard Joubert who has accepted the book as a volume in the book series he is editing, Advances in Parallel Computing. We are proud to have our book in his very prestigious book series.

Two members of the Khronos organization, Elizabeth Riegel and Neil Trevett, helped us with evaluating the initial draft of Chapter 2 Fundamentals and provided valuable feedback. We thank them for the feedback and for their offer of promoting the book among the Khronos Group member companies.

Our thanks are due to NVIDIA for two hardware grants that enabled our computing work related to the book.

Our thanks are due to Piotr Arłukowicz, who contributed two sections to the book and helped us with editorial issues related to using LaTeX and the Blender^{3D} modeling open-source program.

We thank two persons who helped us improve the book structure and the language. They are Dominic Eschweiler from FIAS, Germany and Roberta Scholz from Redmond, USA.

We also thank several friends and family members who helped us indirectly by supporting in various ways our book writing effort.

Janusz Kowalik
Tadeusz Puźniakowski

How to read this book

The text and the source code presented in this book are written using different text fonts. Here are some examples of diffrent typography styles collected.

variable – for example:
... the variable *platform* represents an object of class ...

`type` or `class name` – for example:
... the value is always of type `cl_ulong`...
... is an object of class `cl::Platform`...

`constant` or `macro` – for example:
... the value `CL_PLATFORM_EXTENSIONS` means that...

function, **method** or **constructor** – for example:
... the host program has to execute **clGetPlatformIDs**...
... can be retrieved using **cl::Platform::getInfo** method...
... the context is created by **cl::Context** construcor...

`file name` – for example:
... the `cl.h` header file contains...

keyword – for example:
... identified by the **__kernel** qualifier...

Contents

1 Introduction **1**

1.1 Existing Standard Parallel Programming Systems 1
1.1.1 MPI 1
1.1.2 OpenMP 4
1.2 Two Parallelization Strategies: Data Parallelism and Task Parallelism . 9
1.2.1 Data Parallelism 9
1.2.2 Task Parallelism 9
1.2.3 Example 10
1.3 History and Goals of OpenCL 12
1.3.1 Origins of Using GPU in General Purpose Computing 12
1.3.2 Short History of OpenCL 13
1.4 Heterogeneous Computer Memories and Data Transfer 14
1.4.1 Heterogeneous Computer Memories 14
1.4.2 Data Transfer 15
1.4.3 The Fourth Generation CUDA 15
1.5 Host Code 16
1.5.1 Phase a. Initialization and Creating Context 17
1.5.2 Phase b. Kernel Creation, Compilation and Preparations 17
1.5.3 Phase c. Creating Command Queues and Kernel Execution . . . 17
1.5.4 Finalization and Releasing Resource 18
1.6 Applications of Heterogeneous Computing 18
1.6.1 Accelerating Scientific/Engineering Applications 19
1.6.2 Conjugate Gradient Method 19
1.6.3 Jacobi Method 21
1.6.4 Power Method 22
1.6.5 Monte Carlo Methods 22
1.6.6 Conclusions 23
1.7 Benchmarking CGM 24
1.7.1 Introduction 24
1.7.2 Additional CGM Description 24
1.7.3 Heterogeneous Machine 24
1.7.4 Algorithm Implementation and Timing Results 24
1.7.5 Conclusions 25

2 OpenCL Fundamentals **27**
2.1 OpenCL Overview . . . 27
2.1.1 What is OpenCL . . . 27
2.1.2 CPU + Accelerators . . . 27
2.1.3 Massive Parallelism Idea . . . 27
2.1.4 Work Items and Workgroups . . . 29
2.1.5 OpenCL Execution Model . . . 29
2.1.6 OpenCL Memory Structure . . . 30
2.1.7 OpenCL C Language for Programming Kernels . . . 30
2.1.8 Queues, Events and Context . . . 30
2.1.9 Host Program and Kernel . . . 31
2.1.10 Data Parallelism in OpenCL . . . 31
2.1.11 Task Parallelism in OpenCL . . . 32
2.2 How to Start Using OpenCL . . . 32
2.2.1 Header Files . . . 33
2.2.2 Libraries . . . 33
2.2.3 Compilation . . . 34
2.3 Platforms and Devices . . . 34
2.3.1 OpenCL Platform Properties . . . 36
2.3.2 Devices Provided by Platform . . . 37
2.4 OpenCL Platforms – C++ . . . 40
2.5 OpenCL Context to Manage Devices . . . 41
2.5.1 Different Types of Devices . . . 43
2.5.2 CPU Device Type . . . 43
2.5.3 GPU Device Type . . . 44
2.5.4 Accelerator . . . 44
2.5.5 Different Device Types – Summary . . . 44
2.5.6 Context Initialization – by Device Type . . . 45
2.5.7 Context Initialization – Selecting Particular Device . . . 46
2.5.8 Getting Information about Context . . . 47
2.6 OpenCL Context to Manage Devices – C++ . . . 48
2.7 Error Handling . . . 50
2.7.1 Checking Error Codes . . . 50
2.7.2 Using Exceptions – Available in C++ . . . 53
2.7.3 Using Custom Error Messages . . . 54
2.8 Command Queues . . . 55
2.8.1 In-order Command Queue . . . 55
2.8.2 Out-of-order Command Queue . . . 57
2.8.3 Command Queue Control . . . 60
2.8.4 Profiling Basics . . . 61
2.8.5 Profiling Using Events – C example . . . 61
2.8.6 Profiling Using Events – C++ example . . . 63
2.9 Work-Items and Work-Groups . . . 65
2.9.1 Information About Index Space from a Kernel . . . 66
2.9.2 NDRange Kernel Execution . . . 67
2.9.3 Task Execution . . . 70
2.9.4 Using Work Offset . . . 70

2.10 OpenCL Memory . . . 71
2.10.1 Different Memory Regions – the Kernel Perspective . . . 71
2.10.2 Relaxed Memory Consistency . . . 73
2.10.3 Global and Constant Memory Allocation – Host Code . . . 75
2.10.4 Memory Transfers – the Host Code . . . 78
2.11 Programming and Calling Kernel . . . 79
2.11.1 Loading and Compilation of an OpenCL Program . . . 81
2.11.2 Kernel Invocation and Arguments . . . 88
2.11.3 Kernel Declaration . . . 90
2.11.4 Supported Scalar Data Types . . . 90
2.11.5 Vector Data Types and Common Functions . . . 92
2.11.6 Synchronization Functions . . . 94
2.11.7 Counting Parallel Sum . . . 96
2.11.8 Parallel Sum – Kernel . . . 97
2.11.9 Parallel Sum – Host Program . . . 100
2.12 Structure of the OpenCL Host Program . . . 103
2.12.1 Initialization . . . 104
2.12.2 Preparation of OpenCL Programs . . . 106
2.12.3 Using Binary OpenCL Programs . . . 107
2.12.4 Computation . . . 109
2.12.5 Release of Resources . . . 113
2.13 Structure of OpenCL host Programs in C++ . . . 114
2.13.1 Initialization . . . 115
2.13.2 Preparation of OpenCL Programs . . . 115
2.13.3 Using Binary OpenCL Programs . . . 116
2.13.4 Computation . . . 120
2.13.5 Release of Resources . . . 121
2.14 The SAXPY Example . . . 122
2.14.1 Kernel . . . 123
2.14.2 The Example SAXPY Application – C Language . . . 123
2.14.3 The example SAXPY application – C++ language . . . 128
2.15 Step by Step Conversion of an Ordinary C Program to OpenCL . . . 131
2.15.1 Sequential Version . . . 132
2.15.2 OpenCL Initialization . . . 132
2.15.3 Data Allocation on the Device . . . 134
2.15.4 Sequential Function to OpenCL Kernel . . . 135
2.15.5 Loading and Executing a Kernel . . . 136
2.15.6 Gathering Results . . . 139
2.16 Matrix by Vector Multiplication Example . . . 139
2.16.1 The Program Calculating $matrix \times vector$ . . . 140
2.16.2 Performance . . . 142
2.16.3 Experiment . . . 142
2.16.4 Conclusions . . . 144

3 Advanced OpenCL **147**
3.1 OpenCL Extensions . . . 147
3.1.1 Different Classes of Extensions . . . 147

3.1.2 Detecting Available Extensions from API . . . 148
3.1.3 Using Runtime Extension Functions . . . 149
3.1.4 Using Extensions from OpenCL Program . . . 153
3.2 Debugging OpenCL codes . . . 155
3.2.1 Printf . . . 155
3.2.2 Using GDB . . . 157
3.3 Performance and Double Precision . . . 162
3.3.1 Floating Point Arithmetics . . . 162
3.3.2 Arithmetics Precision – Practical Approach . . . 165
3.3.3 Profiling OpenCL Application . . . 172
3.3.4 Using the Internal Profiler . . . 173
3.3.5 Using External Profiler . . . 180
3.3.6 Effective Use of Memories – Memory Access Patterns . . . 183
3.3.7 Matrix Multiplication – Optimization Issues . . . 189
3.4 OpenCL and OpenGL . . . 194
3.4.1 Extensions Used . . . 195
3.4.2 Libraries . . . 196
3.4.3 Header Files . . . 196
3.4.4 Common Actions . . . 197
3.4.5 OpenGL Initialization . . . 198
3.4.6 OpenCL Initialization . . . 201
3.4.7 Creating Buffer for OpenGL and OpenCL . . . 203
3.4.8 Kernel . . . 209
3.4.9 Generating Effect . . . 213
3.4.10 Running Kernel that Operates on Shared Buffer . . . 215
3.4.11 Results Display . . . 216
3.4.12 Message Handling . . . 218
3.4.13 Cleanup . . . 219
3.4.14 Notes and Further Reading . . . 221
3.5 Case Study – Genetic Algorithm . . . 221
3.5.1 Historical Notes . . . 221
3.5.2 Terminology . . . 221
3.5.3 Genetic Algorithm . . . 222
3.5.4 Example Problem Definition . . . 225
3.5.5 Genetic Algorithm Implementation Overview . . . 225
3.5.6 OpenCL Program . . . 226
3.5.7 Most Important Elements of Host Code . . . 234
3.5.8 Summary . . . 241
3.5.9 Experiment Results . . . 241

A Comparing CUDA with OpenCL 245
A.1 Introduction to CUDA . . . 245
A.1.1 Short CUDA Overview . . . 245
A.1.2 CUDA 4.0 Release and Compatibility . . . 245
A.1.3 CUDA Versions and Device Capability . . . 247
A.2 CUDA Runtime API Example . . . 249
A.2.1 CUDA Program Explained . . . 251

A.2.2 Blocks and Threads Indexing Formulas 257
A.2.3 Runtime Error Handling 260
A.2.4 CUDA Driver API Example 262

B Theoretical Foundations of Heterogeneous Computing 269
B.1 Parallel Computer Architectures 269
B.1.1 Clusters and SMP 269
B.1.2 DSM and *cc*NUMA 270
B.1.3 Parallel Chip Computer 270
B.1.4 Performance of OpenCL Programs 270
B.2 Combining MPI with OpenCL 277

C Matrix Multiplication – Algorithm and Implementation 279
C.1 Matrix Multiplication 279
C.2 Implementation 279
C.2.1 OpenCL Kernel 279
C.2.2 Initialization and Setup 280
C.2.3 Kernel Arguments 282
C.2.4 Executing Kernel 282

D Using Examples Attached to the Book 285
D.1 Compilation and Setup 286
D.1.1 Linux 286
D.1.2 Windows 286

Bibliography and References 289

Chapter 1

Introduction

1.1. Existing Standard Parallel Programming Systems

The last decade of the 20^{th} century and the first decade of the 21^{st} century can be called the Era of Parallel Computing. In this period of time, not only were extremely powerful supercomputers designed and built, but two de facto standard parallel programming systems for scientific and engineering applications were successfully introduced worldwide. They are **MPI** (Message Passing Interface) for clusters of computers and **OpenMP** for shared memory multi-processors. Both systems are predecessors of the subject of this book on OpenCL. They deserve short technical description and discussion. This will help to see differences between the older MPI/OpenMP and the newest OpenCL parallel programming systems.

1.1.1. MPI

MPI is a programming system but not a programming language. It is a library of functions for C and subroutines for FORTRAN that are used for message communication between parallel processes created by MPI. The message-passing computing model (Fig. 1.1) is a collection of interconnected processes that use their local memories exclusively. Each process has an individual address identification number called the rank. Ranks are used for sending and receiving messages and for workload distribution. There are two kinds of messages: point-to-point messages and collective messages. Point-to-point message C functions contain several parameters: the address of the sender, the address of the receiver, the message size and type and some additional information. In general, message parameters describe the nature of the transmitted data and the delivery envelope description.

The collection of processes that can exchange messages is called the communicator. In the simplest case there is only one communicator and each process is assigned to one processor. In more general settings, there are several communicators and single processors serve several processes. A processor rank is usually an integer number running from 0 to p-1 where p is the total number of processes. It is also possible to number processes in a more general way than by consecutive integer numbers. For example, their address ID can be a double number such as a point (x, y) in Carte-

sian space. This method of identifying processes may be very helpful in handling matrix operations or partial differential equations (PDEs) in two-dimensional Cartesian space.

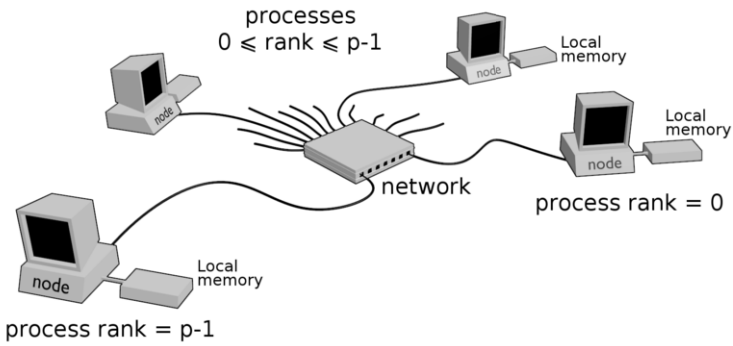

Figure 1.1: The message passing model.

An example of a collective message is the broadcast message that sends data from a single process to all other processes. Other collective messages such as `Gather` and `Scatter` are very helpful and are often used in computational mathematics algorithms.

For the purpose of illustrating MPI, consider a parallel computation of the SAXPY operation. SAXPY is a linear algebra operation $\mathbf{z} = a\mathbf{x}+\mathbf{y}$ where **a** is a constant scalar and **x**, **y**, **z** are vectors of the same size. The name SAXPY comes from **S**um of **a**x plus **y**. An example of a SAXPY operation is presented in section 2.14.

The following assumptions and notations are made:

1. The number of processes is p.
2. The vector size n is divisible by p.
3. Only a part of the code for parallel calculation of the vector **z** will be written.
4. All required MPI initialization instructions have been done in the front part of the code.
5. The vectors x and y have been initialized.
6. The process ID is called `my_rank` from 0 to $p-1$. The number of vector element pairs that must be computed by each process is: $N = n/p$. The communicator is the entire system.
7. $n = 10000$ and $p = 10$ so $N = 1000$.

The C loop below will be executed by all processes from the process from `my_rank` equal to 0 to `my_rank` equal to $p-1$.

```
{
  int i;
  for (i = my_rank*N; i < (my_rank+1)*N; i++)
    z[i]=a*x[i]+y[i];
}
```

For example, the process with my_rank=1 will add one thousand elements a*x[i] and y[i] from i=N=1000 to 2N–1=1999. The process with the my_rank=p–1=9 will add elements from i=n–N=9000 to n–1=9999.

One can now appreciate the usefulness of assigning a rank ID to each process. This simple concept makes it possible to send and receive messages and share the workloads as illustrated above. Of course, before each process can execute its code for computing a part of the vector z it must receive the needed data. This can be done by one process initializing the vectors x and y and sending appropriate groups of data to all remaining processes. The computation of SAXPY is an example of data parallelism. Each process executes the same code on different data.

To implement function parallelism where several processes execute different programs process ranks can be used. In the SAXPY example the block of code containing the loop will be executed by all p processes without exception. But if one process, for example, the process rank 0, has to do something different than all others this could be accomplished by specifying the task as follows:

```
1 if (my_rank == 0)
2   {execute specified here task}
```

This block of code will be executed only by the process with my_rank==0. In the absence of "if (my_rank==something)" instructions, all processes will execute the block of code {execute this task}. This technique can be used for a case where processes have to perform several different computations required in the task-parallel algorithm. MPI is universal. It can express every kind of algorithmic parallelism.

An important concern in parallel computing is the efficiency issue. MPI often can be computationally efficient because all processes use only local memories. On the other hand, processes require network communication to place data in proper process memories in proper times. This need for moving data is called the *algorithmic synchronization*, and it creates the communication overhead, impacting negatively the parallel program performance. It might significantly damage performance if the program sends many short messages. The communication overhead can be reduced by grouping data for communication. By grouping data for communication created are larger user defined data types for larger messages to be sent less frequently. The benefit is avoiding the latency times.

On the negative side of the MPI evaluation score card is its inability for incremental (part by part) conversion from a serial code to an MPI parallel code that is algorithmically equivalent. This problem is attributed to the relatively high level of MPI program design. To design an MPI program, one has to modifiy algorithms. This work is done at an earlier stage of the parallel computing process than developing parallel codes. Algorithmic level modification usually can't be done piecewise. It has to be done all at once. In contrast, in OpenMP a programmer makes changes to sequential codes written in C or FORTRAN.

Fig. 1.2 shows the difference. The code level modification makes possible incremental conversions. In a commonly practiced conversion technique, a serial code is converted incrementally from the most compute-intensive program parts to the least compute intensive parts until the parallelized code runs sufficiently fast. For further study of MPI, the book [1] is highly recommended.

1. Mathematical model.
2. Computational model.
3. Numerical algorithm and parallel conversion: MPI
4. Serial code and parallel modification: OpenMP
5. Computer runs.

Figure 1.2: The computer solution process. OpenMP will be discussed in the next section.

1.1.2. OpenMP

OpenMP is a shared address space computer application programming interface (API). It contains a number of compiler directives that instruct C/C++ or FORTRAN "OpenMP aware" compilers to execute certain instructions in parallel and distribute the workload among multiple parallel threads. Shared address space computers fall into two major groups: centralized memory multiprocessors, also called *Symmetric Multi-Processors* (SMP) as shown in Fig. 1.3, and *Distributed Shared Memory* (DSM) multiprocessors whose common representative is the cache coherent Non Uniform Memory Access architecture (ccNUMA) shown in Fig. 1.4.

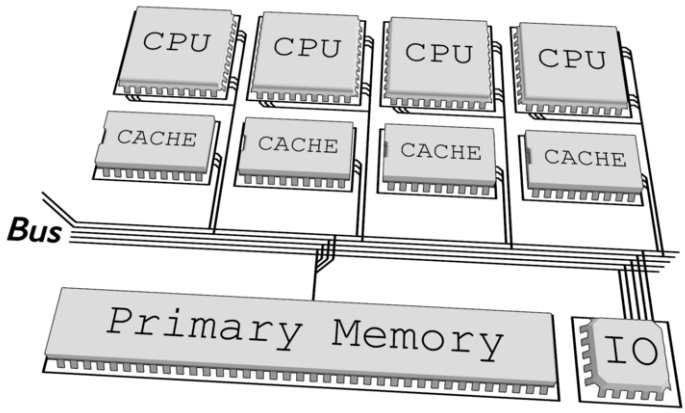

Figure 1.3: The bus based SMP multiprocessor.

SMP computers are also called Uniform Memory Access (UMA). They were the first commercially successful shared memory computers and they still remain popular. Their main advantage is uniform memory access. Unfortunately for large numbers of processors, the bus becomes overloaded and performance deteriorates. For this reason, the bus-based SMP architectures are limited to about 32 or 64 processors. Beyond these sizes, single address memory has to be physically distributed. Every processor has a chunk of the single address space.

Both architectures have single address space and can be programmed by using OpenMP. OpenMP parallel programs are executed by multiple independent threads

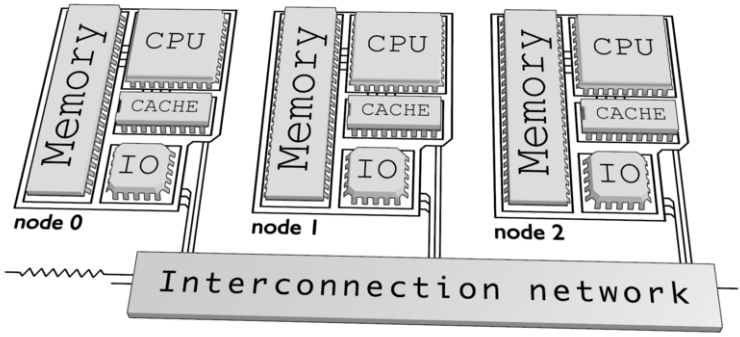

Figure 1.4: ccNUMA with cache coherent interconnect.

that are streams of instructions having access to shared and private data, as shown in Fig. 1.5.

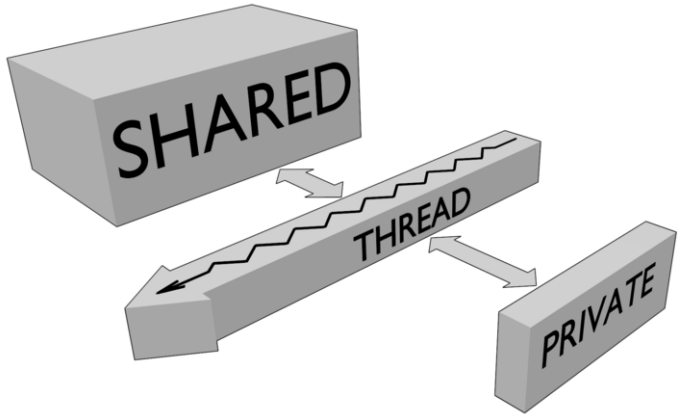

Figure 1.5: The threads' access to data.

The programmer can explicitly define data that are shared and data that are private. Private data can be accessed only by the thread owning these data. The flow of OpenMP computation is shown in Fig. 1.6. One thread, the master thread, runs continuously from the beginning to the end of the program execution. The worker threads are created for the duration of parallel regions and then are terminated.

When a programmer inserts a proper compiler directive, the system creates a team of worker threads and distributes workload among them. This operation point is called the fork. After forking, the program enters a parallel region where parallel processing is done. After all threads finish their work, the program worker threads cease to exist or are redeployed while the master thread continues its work. The event of returning to the master thread is called the join. The **fork and join** technique may look like a simple and easy way for parallelizing sequential codes, but the

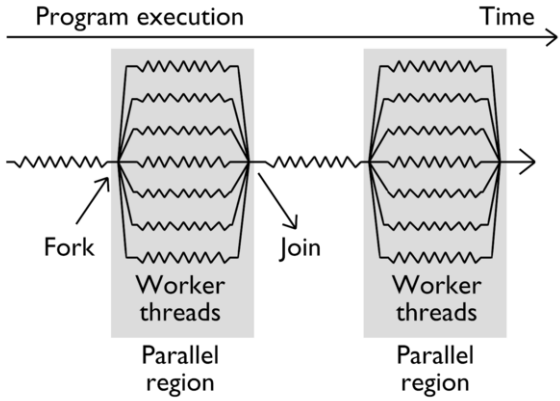

Figure 1.6: The fork and join OpenMP code flow.

reader should not be deceived. There are several difficulties that will be described and discussed shortly. First of all, how to find whether several tasks can be run in parallel?

In order for any group of computational tasks to be correctly executed in parallel, they have to be independent. That means that the result of the computation does not depend on the order of task execution. Formally, two programs or parts of programs are independent if they satisfy the Bernstein conditions shown in 1.1.

$$\begin{aligned} I_j \cap O_i &= 0 \\ I_i \cap O_j &= 0 \\ O_i \cap O_j &= 0 \end{aligned} \tag{1.1}$$

Bernstein's conditions for task independence.

Letters I and O signify input and output. The $\cap$ symbol means the intersection of two sets that belong to task i or j. In practice, determining if some group of tasks can be executed in parallel has to be done by the programmer and may not be very easy. To discuss other shared memory computing issues, consider a very simple programming task. Suppose the *dot product* of two large vectors $\mathbf{x}$ and $\mathbf{y}$ whose number of components is n is calculated. The sequential computation would be accomplished by a simple C loop

```
1 double dp = 0;
2 for(int i=0; i<n; i++)
3   dp += x[i]*y[i];
```

Inserting, in front of the above for-loop, the OpenMP directive for parallelizing the loop and declaring shared and private variables leads to:

```
double dp = 0;
#pragma omp parallel for shared(x,y,dp) (private i)
for(int i=0; i<n; i++)
  dp += x[i]*y[i];
```

Unfortunately, this solution would encounter a serious difficulty in computing dp by this code. The difficulty arises because the computation of dp+=x[i]*y[i]; is not atomic. This means that more threads than one may attempt to update the value of dp simultaneously. If this happens, the value of dp will depend on the timing of individual threads' operations. This situation is called the data race. The difficulty can be removed in two ways.

One way is by using the OpenMP critical construct #pragma omp critical in front of the statement dp+=x[i]*y[i]; The directive **critical** forces threads to update dp one at a time. In this way, the updating relation dp+=x[i]*y[i] becomes atomic and is executed correctly. That means every thread executes this operation alone and completely without interference of other threads. With this addition, the code becomes:

```
double dp = 0;
#pragma omp parallel for shared(x,y,dp) private(i)
for(int i=0; i<n; i++){
  #pragma omp critical
  dp += x[i]*y[i];
}
```

In general, the block of code following the critical construct is computed by one thread at a time. In our case, the critical block is just one line of code dp+=x[i]*y[i];. The second way for fixing the problem is by using the clause reduction that ensures adding correctly all partial results of the dot product. Below is a correct fragment of code for computing dp with the reduction clause.

```
double dp = 0;
#pragma omp parallel for reduction(+:dp) shared(x,y) private(i)
for (int i=0;i<n ;i++)
  dp += x[i]*y[i];
```

Use of the critical construct amounts to serializing the computation of the critical region, the block of code following the critical construct. For this reason, large critical regions degrade program performance and ought to be avoided. Using the reduction clause is more efficient and preferable. The reader may have noticed that the loop counter variable i is declared private, so each thread updates its loop counter independently without interference. In addition to the parallelizing construct **parallel for** that applies to C for-loops, there is a more general **section construct** that parallelizes independent sections of code. The section construct makes it possible to apply OpenMP to task parallelism where several threads can compute different sections of a code. For computing two parallel sections, the code structure is:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    /* some program segment computation */
    #pragma omp section
    /* another program segment computation */
  }
  /* end of sections block */
}
/* end of parallel region */
```

OpenMP has tools that can be used by programmers for improving performance. One of them is the clause `nowait`. Consider the program fragment:

```
#pragma omp parallel shared(a,b,c,d,e,f) private(i,j)
{
  #pragma omp for nowait
  for(int i=0; i<n; i++)
    c[i] = a[i]+b[i];
  #pragma omp for
  for(int j=0; j<m; j++)
    d[j] = e[j]*f[j];
  #pragma omp barrier
  g = func(d);
}
/* end of the parallel region; implied barrier */
```

In the first parallelized loop, there is the clause `nowait`. Since the second loop variables do not depend on the results of the first loop, it is possible to use the clause `nowait` – telling the compiler that as soon as any first loop thread finishes its work, it can start doing the second loop work without waiting for other threads. This speeds up computation by reducing the potential waiting time for the threads that finish work at different times.

On the other hand, the construct `#pragma omp barrier` is inserted after the second loop to make sure that the second loop is fully completed before the calculation of `g` being a function of `d` is performed after the second loop. At the barrier, all threads computing the second loop must wait for the last thread to finish before they proceed. In addition to the explicit barrier construct `#pragma omp barrier` used by the programmer, there are also implicit barriers used by OpenMP automatically at the end of every parallel region for the purpose of thread synchronization. To sum up, barriers should be used sparingly and only if necessary. `nowait` clauses should be used as frequently as possible, provided that their use is safe.

Finally, we have to point out that the major performance issue in numerical computation is the use of cache memories. For example, in computing the matrix/vector product $\mathbf{c} = \mathbf{Ab}$, two mathematically equivalent methods could be used. In the first method, elements of the vector $\mathbf{c}$ are computed by multiplying each row of $\mathbf{A}$ by the vector $\mathbf{b}$, i.e., computing dot products. An inferior performance will be obtained if $\mathbf{c}$ is computed as the sum of the columns of $\mathbf{A}$ multiplied by the elements of $\mathbf{b}$. In this

case, the program would access the columns of **A** not in the way the matrix data are stored in the main memory and transferred to caches.

For further in-depth study of OpenMP and its performance, reading [2] is highly recommended. Of special value for parallel computing practitioners are Chapter 5 “How to get Good Performance by Using OpenMP” and Chapter 6 “Using OpenMP in the Real World”. Chapter 6 offers advice for and against using combined MPI and OpenMP. A chapter on combining MPI and OpenMP can also be found in [3]. Like MPI and OpenMP, the OpenCL system is standardized. It has been designed to run regardless of processor types, operating systems and memories. This makes OpenCL programs highly portable, but the method of developing OpenCL codes is more complicated. The OpenCL programmer has to deal with several low-level programming issues, including memory management.

1.2. Two Parallelization Strategies: Data Parallelism and Task Parallelism

There are two strategies for designing parallel algorithms and related codes: data parallelism and task parallelism. This Section describes both concepts.

1.2.1. Data Parallelism

Data parallelism, also called Single Program Multiple Data (SPMD) is very common in computational linear algebra. In a data parallel code, data structures such as matrices are divided into blocks, sets of rows or columns, and a single program performs identical operations on these partitions that contain different data. An example of data parallelism is the matrix/vector multiplication $\mathbf{a} = \mathbf{Ab}$ where every element of **a** can be computed by performing the dot product of one row of **A** and the vector **b**. Fig. 1.7 shows this data parallel concept.

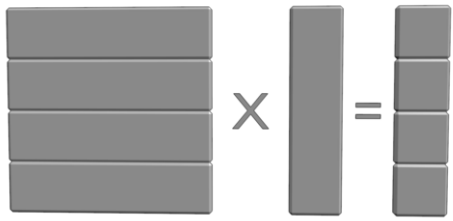

Figure 1.7: Computing matrix/vector product.

1.2.2. Task Parallelism

The task parallel approach is more general. It is assumed that there are multiple different independent tasks that can be computed in parallel. The tasks operate on

their own data sets. Task parallelism can be called Multiple Programs Multiple Data (MPMD). A small-size example of a task parallel problem is shown in Fig. 1.8. The directed graph indicates the task execution precedence. Two tasks can execute in parallel if they are not dependent.

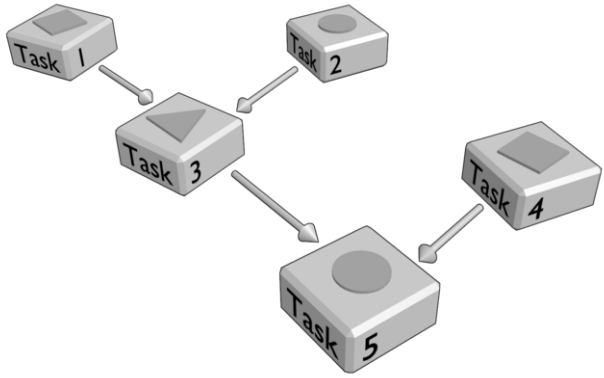

Figure 1.8: Task dependency graph.

In the case shown in Fig. 1.8, there are two options for executing the entire set of tasks. Option 1. Execute tasks T1, T2 and T4 in parallel, followed by Task T3 and finally T5. Option 2. Execute tasks T1 and T2 in parallel, then T3 and T4 in parallel and finally T5. In both cases, the total computing work is equal but the time to solution may not be.

1.2.3. Example

Consider a problem that can be computed in both ways – via data parallelism and task parallelism. The problem is to calculate $\mathbf{C} = \mathbf{A} \times \mathbf{B} - (\mathbf{D} + \mathbf{E})$ where **A**, **B**, **D** and **E** are all square matrices of size $n \times n$. An obvious task parallel version would be to compute in parallel two tasks $\mathbf{A} \times \mathbf{B}$ and $\mathbf{D} + \mathbf{E}$ and then subtract the sum from the product. Of course, the task of computing $\mathbf{A} \times \mathbf{B}$ and the task of computing the sum $\mathbf{D} + \mathbf{E}$ can be calculated in a data parallel fashion. Here, there are two levels of parallelism: the higher task level and the lower data parallel level. Similar multilevel parallelism is common in real-world applications. Not surprisingly, there is also for this problem a direct data parallel method based on the observation that every element of **C** can be computed directly and independently from the coefficients of **A**, **B**, **D** and **E**. This computation is shown in equation 1.2.

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} - d_{ij} - e_{ij} \tag{1.2}$$

The direct computation of **C**.

The equation (1.2) means that it is possible to calculate all n^2 elements of **C** in parallel using the same formula. This is good news for OpenCL devices that can handle only one kernel and related data parallel computation. Those devices will be discussed in the chapters that follow. The matrix C can be computed using three standard programming systems: MPI, OpenMP and OpenCL.

Using MPI, the matrix **C** could be partitioned into sub-matrix components and assigned the subcomponents to processes. Every process would compute a set of elements of **C**, using the expression 1.2. The main concern here is not computation itself but the ease of assembling results and minimizing the cost of communication while distributing data and assembling the results. A reasonable partitioning would be dividing **C** by blocks of rows that can be scattered and gathered as user-defined data types. Each process would get a set of the rows of **A**, **D** and **E** and the entire matrix **B**. If matrix **C** size is n and the number of processes is p, then each process would get n/p rows of **C** to compute. If a new data type is defined as n/p rows, the data can easily be distributed by strips of rows to processes and then results can be gathered. The suggested algorithm is a data parallel method. Data partitioning is shown in Fig. 1.9.

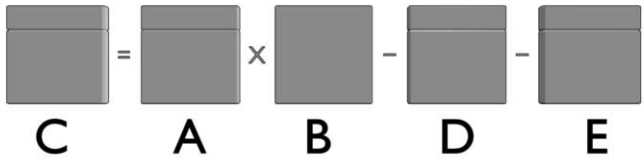

Figure 1.9: Strips of data needed by a process to compute the topmost strip of **C**.

The data needed for each process include one strip of **A**, **D** and **E** and the entire matrix **B**. Each process of rank `0<=rank<p` computes one strip of **C** rows. After finishing computation, the matrix **C** can be assembled by the collective MPI communication function `Gather`. An alternative approach would be partitioning `Gather` into blocks and assigning to each process computing one block of **C**. However, assembling the results would be harder than in the strip partitioning case.

OpenMP would take advantage of the task parallel approach. First, the subtasks $\mathbf{A} \times \mathbf{B}$ and $\mathbf{D} + \mathbf{E}$ would be parallelized and computed separately, and then $\mathbf{C} = \mathbf{A} \times \mathbf{B} - (\mathbf{D} + \mathbf{E})$ would be computed, as shown in Fig. 1.10. One weakness of task parallel programs is the efficiency loss if all parallel tasks do not represent equal workloads. For example, in the matrix **C** computation, the task of computing $\mathbf{A} \times \mathbf{B}$ and the task of computing $\mathbf{D} + \mathbf{E}$ are not load-equal. Unequal workloads could cause some threads to idle unless tasks are selected dynamically for execution by the scheduler. In general, data parallel implementations are well load balanced and tend to be more efficient. In the case of OpenCL implementation of the data parallel option for computing **C**, a single kernel function would compute elements of **C** by the equation 1.2 where the sum represents dot products of $\mathbf{A} \times \mathbf{B}$ and the remaining terms represent subtracting **D** and **E**. If the matrix size is $n = 1024$, a compute device could execute over one million work items in parallel. Additional performance gain can be achieved by using the tiling technique [4].

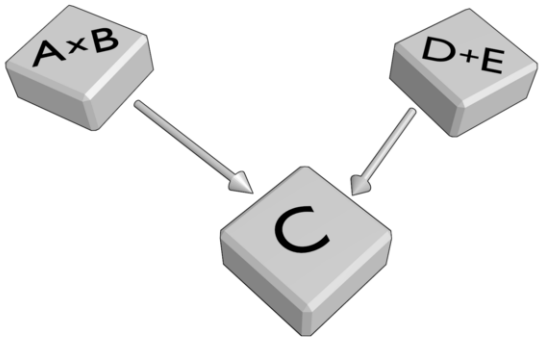

Figure 1.10: The task parallel approach for OpenMP.

In the tiling technique applied to the matrix/matrix product operation, matrices are partitioned into tiles small enough so that the data for dot products can be placed in local (in the NVIDIA terminology, shared) memories. This way, slow access to global memory is avoided and smaller, but faster, local memories are used.

The tiling technique is one of the most effective ways for enhancing performance of heterogeneous computing. In the matrix/matrix multiplication problem, the NVIDIA GPU Tesla C1060 processor was used as a massively parallel acceleration device.

In general: the most important heterogeneous computing strategy for achieving efficient applications is optimizing memory use. This includes avoiding CPU-GPU data transfers and using fast local memory. Finally, a serious limitation of many current (2011) generation GPUs has to be mentioned. Some of them are not capable of running in parallel multiple kernels. They run only one kernel at a time in data parallel fashion. This eliminates the possibility of a task parallel program structure requiring multiple kernels. Using such GPUs, the only possibility of running multiple kernels would be to have multiple devices, each one executing a single kernel, but the host would be a single processor. The context for managing two OpenCL devices is shown in Fig. 2.3 in section 2.1.

1.3. History and Goals of OpenCL

1.3.1. Origins of Using GPU in General Purpose Computing

Initially, massively parallel processors called GPUs (Graphics Processor Units) were built for applications in graphics and gaming.

Soon the scientific and engineering communities realized that data parallel computation is so common in the majority of scientific/engineering numerical algorithms that GPUs can also be used for compute intensive parts of large numerical algorithms solving scientific and engineering problems. This idea contributed to creating a new computing discipline under the name General Purpose GPU or **GPGPU**. Several nu-

merical algorithms containing data parallel computations are discussed in Section 1.6 of this Chapter.

As a parallel programming system, OpenCL is preceded by MPI, OpenMP and CUDA. Parallel regions in OpenMP are comparable to parallel kernel executions in OpenCL. A crucial difference is OpenMP's limited scalability due to heavy overhead in creating and managing threads. Threads used on heterogeneous systems are lightweight, hence suitable for massive parallelism. The most similar to OpenCL is NVIDIA's CUDA (Compute Unified Driver Architecture) because OpenCL heavily borrowed from CUDA its fundamental features. Readers who know CUDA will find it relatively easy to learn and use OpenCL after becoming familiar with the mapping between CUDA and OpenCL technical terms and minor programming differences.

The technical terminology differences are shown in Tab.1.1.

Table 1.1: Comparision of terminology of CUDA and OpenCL

CUDA	OpenCL
Thread	Work item
Block	Work group
Grid	Index space

Similar one-to-one mapping exists for the CUDA and OpenCL API calls.

1.3.2. Short History of OpenCL

OpenCL (**Open C**omputing **L**anguage) was initiated by Apple, Inc., which holds its trademark rights. Currently, OpenCL is managed by the Khronos Group, which includes representatives from several major computer vendors. Technical specification details were finalized in November 2008 and released for public use in December of that year. In 2010, IBM and Intel released their implementations of OpenCL. In the middle of November 2010, Wolfram Research released Mathematica 8 with an OpenCL link. Implementations of OpenCL have been released also by other companies, including NVIDIA and AMD. The date for a stable release of OpenCL 1.2 standard is November 2011.

One of the prime goals of OpenCL designers and specification authors has been portability of the OpenCL codes. Using OpenCL is not limited to any specific vendor hardware, operating systems or type of memory. This is the most unique feature of the emerging standard programming system called OpenCL. Current (2011) OpenCL specification management is in the hands of the international group Khronos, which includes IBM, SONY, Apple, NVIDIA, Texas Instruments, AMD and Intel.

Khronos manages specifications of OpenCL C language and OpenCL runtime APIs. One of the leaders in developing and spreading OpenCL is the Fixstars Corporation, whose main technology focus has been programming multi-core systems and optimizing their application performance. This company published the first commercially available book on OpenCL [5].

The book was written by five Japanese software specialists and has been translated into English. It is currently (2011) available from Amazon.com in paperback or the electronic book form. In the latter form, the book can be read using Kindle.

1.4. Heterogeneous Computer Memories and Data Transfer

1.4.1. Heterogeneous Computer Memories

A device's main memory is the *global memory* (Fig. 1.11). All work items have access to this memory. It is the largest but also the slowest memory on a heterogeneous device. The global memory can be dynamically allocated by the host program and can be read and written by the host and the device.

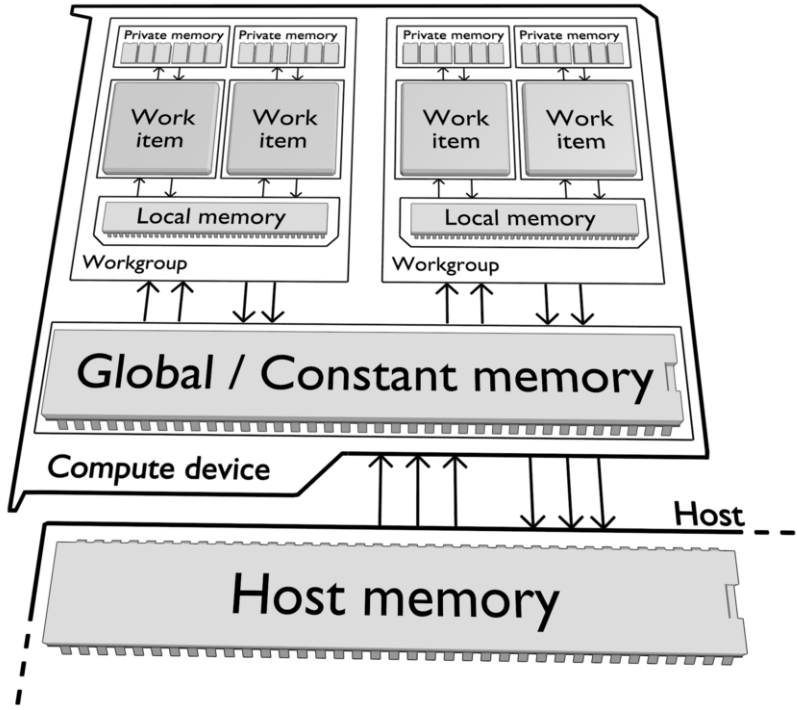

Figure 1.11: Heterogeneous computer memories. Host memory is accesible only to processes working on the host. Global memory is a GPU memory accessible both in read and write for a kernel run on the GPU device. Constant memory is accessible only for read operations.

The *constant memory* can also be dynamically allocated by the host. It can be used for read and write operations by the host, but it is read only by the device. All work items can read from it. This memory can be fast if the system has a supporting cache.

Local memories are shared by work items within a work group. For example, two work items can synchronize their operation using this memory if they belong to the same work group. Local memories can't be accessed by the host.

Private memories can be accessed only by each individual work item. They are registers. A kernel can use only these four device memories.

In contrast, the host system can use the host memory and the global/constant memories on the device. If a device is a multicore CPU, then all device memories are portions of the RAM.

1.4.2. Data Transfer

It has to be pointed out that all data movements required by a particular algorithm have to be programmed explicitly by the programmer, using special commands for data transfer between different kinds of memories. To illustrate different circumstances of transferring data, consider two algorithms: algorithm 1 for matrix–matrix multiplication and algorithm 2 for solving sets of linear equations $\mathbf{Ax} = \mathbf{b}$ with a positive definite matrix **A**. Algorithm 2 is an iterative procedure called The Conjugate Gradient Method (CGM).

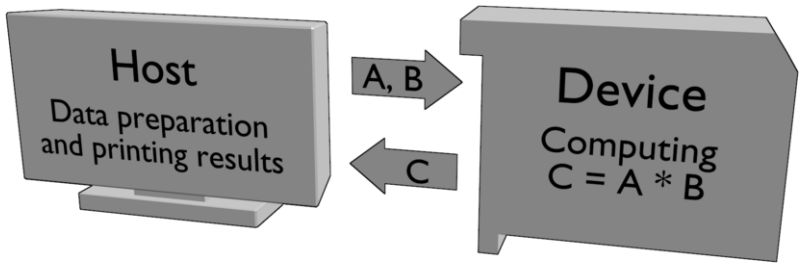

Figure 1.12: Data transfer in matrix/matrix multiplication.

Executing a matrix/matrix multiplication algorithm requires only two data transfer operations. They are shown in Fig. 1.12 by thick arrows. The first transfer is needed to load matrices **A** and **B** into the device global memory. The second transfer moves the resulting matrix **C** to the host RAM to enable printing the results by the host.

The second algorithm is the Conjugate Gradient Method (CGM) discussed in section 1.6. It is assumed that every entire iteration is computed on the GPU device. Fig. 1.13 shows the algorithm flow.

CGM is executed on a heterogeneous computer with a CPU host and a GPU device. The thick arrow indicates data transfer from the CPU memory to the device global memory. Except for the initial transfer of the matrix A and the vector p and one bit after every iteration, there is no need for other data transfers for the CGM algorithm.

1.4.3. The Fourth Generation CUDA

The most recent (March 2011) version of NVIDIAs's Compute Unified Device Architecture (CUDA) brings a new and important memory technology called Unified Virtual Addressing (UVA). UVA provides a single address space for merged CPU memory

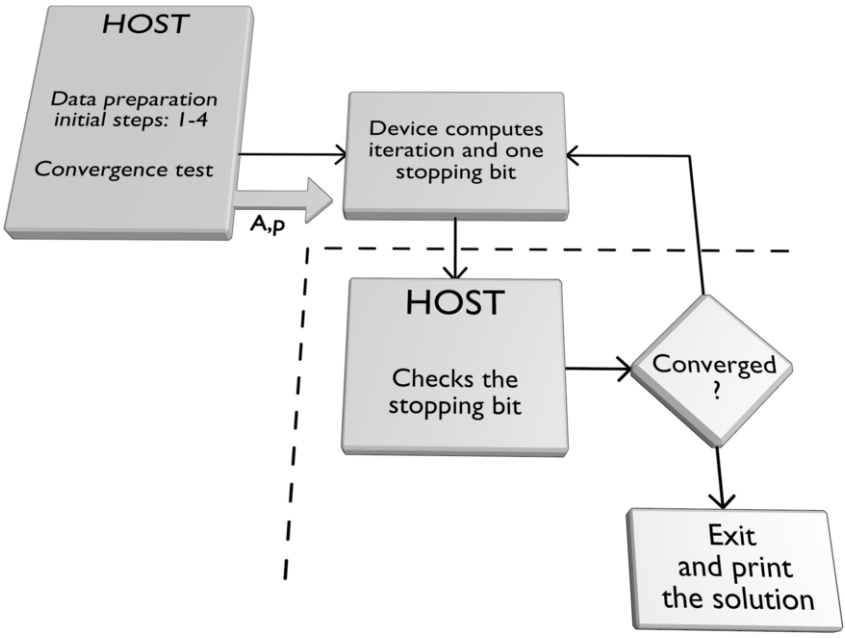

Figure 1.13: The CGM algorithm iterations. The only data transfer takes place at the very beginning of the iterative process.

and global GPU memory. This change will simplify programming systems that combine the CPU and GPU and will eliminate currently necessary data transfers between the host and the device.

1.5. Host Code

Every OpenCL program contains two major components: a host code and at least one kernel, as shown in Fig. 1.14.

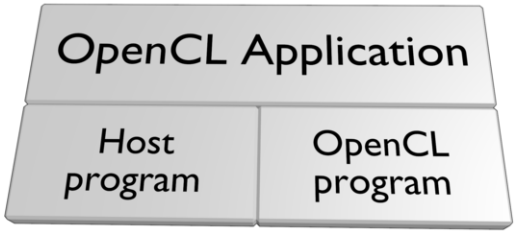

Figure 1.14: OpenCL program components.

Kernels are written in the OpenCL C language and executed on highly parallel devices. They provide performance improvements. An example of a kernel code is shown in section 2.1.

In this section, the structure and the functionality of host programs is described. The main purpose of the host code is to manage device(s). More specifically, host codes arrange and submit kernels for execution on device(s). In general, a host code has four phases:

a) Initialization and creating context,
b) kernel creation and preparation for kernel execution,
c) creating command queues and kernels execution,
d) finalization and release of resources.

Section 2.8 in Chapter 2 provides details of the four phases and an example of the host code structure. This section is an introduction to section 2.11 in Chapter 2. This introduction may be useful for readers who start learning OpenCL since the concept of the host code has not been used in other parallel programming systems.

1.5.1. Phase a. Initialization and Creating Context

The very first step in initialization is getting available OpenCL platforms. This is followed by device selection. The programmer can query available devices and choose those that could help achieve the required level of performance for the application at hand.

The second step is creating the context.

In OpenCL, devices are managed through contexts. Please see Fig. 2.3, illustrating the context for managing devices. To determine the types and the numbers of devices in the system, a special API function is used. All other steps are also performed by invoking runtime API functions. Hence the host code is written in the conventional C or C++ language plus the API Library runtime function calls.

1.5.2. Phase b. Kernel Creation, Compilation and Preparations for Kernel Execution

This phase includes kernel creation, compilation and loading. The kernel has to be loaded to the global memory for execution on the devices. It can be loaded in binary form or as source code.

1.5.3. Phase c. Creating Command Queues and Kernel Execution

In this phase, a Command Queue is created. Kernels must be placed in command queues for execution. When a device is ready, the kernel at the head of the command queue will be executed. After kernel execution, the results can be transferred from the device global memory to the host memory for display or printing. Kernels can be executed out of order or in order. In the out-of-order case, the kernels are independent and can be executed in any sequence. In the second case, the kernel must be executed in a certain fixed order.

1.5.4. Finalization and Releasing Resource

After finishing computation, each code should perform a cleanup operation that includes releasing of resources in preparation for the next application. The entire host code structure is shown in Fig. 1.15. In principle, the host code can also perform some algorithmic computation that is not executed on the device – for example, the initial data preparation.

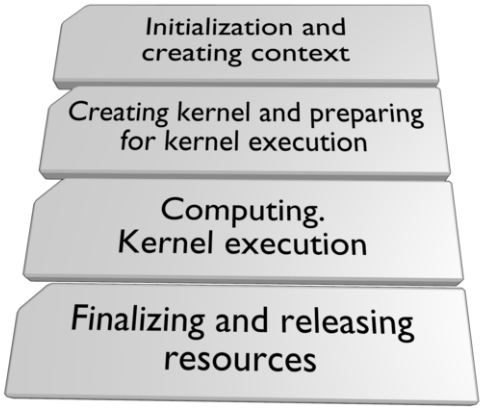

Figure 1.15: Host code structure and functionality.

1.6. Applications of Heterogeneous Computing

Heterogeneous computation is already present in many fields. It is an extension of general-purpose computing on graphics processing units in terms of combining traditional computation on multi-core CPUs and computation on any other computing device. OpenCL can be used for real-time post-processing of a rendered image, or for accelerating a ray-tracing engine. This standard is also used in real-time enhancement for full-motion video, like image stabilization or improving the image quality of one frame using consecutive surrounding frames. Nonlinear video-editing software can accelerate standard operations by using OpenCL; for example, most video filters work per-pixel, so they can be naturally parallelized. Libraries that accelerate basic linear algebra operations are another example of OpenCL applications. There is a wide range of simulation issues that can be targeted by OpenCL – like rigid-body dynamics, fluid and gas simulation and virtually any imaginable physical simulation involving many objects or complicated matrix computation. For example, a game physics engine can use OpenCL for accelerating rigid- and soft-body dynamics in real time to improve graphics and interactivity of the product. Another field where massively parallel algorithms can be used is stock trade analysis.

This section describes several algorithms for engineering and scientific applications that could be accelerated by OpenCL. This is by no means an exhaustive list of all the algorithms that can be accelerated by OpenCL.

1.6.1. Accelerating Scientific/Engineering Applications

Computational linear algebra has been regarded as the workhorse for applied mathematics in nearly all scientific and engineering applications. Software systems such as Linpack have been used heavily by supercomputer users solving large-scale problems.

Algebraic linear equations solvers have been used for many years as the performance benchmark for testing and ranking the fastest 500 computers in the world. The latest (2010) TOP500 champion, is a Chinese computer, the Tianhe-1A.

The Tianhe supercomputer is a heterogeneous machine. It has 14336 Intel Xeon CPUs and 7168 NVIDIA Tesla M2050 GPUs.

The champion's Linpack benchmark performance record is about 2.5 petaflops (2.5×10 to the power 15 floating point operations per second). Such impressive speed has been achieved to a great extent by using massively parallel GPU accelerators. An important question arises: do commonly used scientific and engineering solution algorithms contain components that can be accelerated by massively parallel devices? To partially answer this question, this section of the book considers several numerical algorithms frequently used by scientists and engineers.

1.6.2. Conjugate Gradient Method

The first algorithm, a popular method for solving algebraic linear equations, is called the conjugate gradient method (CG or CGM) [6].

($x^{(0)} \in \Re^n$ given)
1. $x := x^{(0)}$
2. $r := b - Ax$
3. $p := r$
4. $\alpha := \|r\|^2$
5. while $\alpha > tol^2$:
6. $\quad \lambda := \alpha/(p^T Ap)$
7. $\quad x := \lambda p$
8. $\quad r := r - \lambda Ap$
9. $\quad p := r + (\|r\|^2/\alpha)p$
10. $\quad \alpha := \|r\|^2$
11. end

The CGM algorithm solves the system of linear equations $\mathbf{Ax} = \mathbf{b}$ where the matrix $\mathbf{A}$ is positive definite. Every iteration of the CGM algorithm requires one matrix-vector multiplication $\mathbf{Ap}$, two vector dot products:

$$dp = x^T y = \sum_{i=0}^{n-1} x_i y_i \tag{1.3}$$

and three SAXPY operations $\mathbf{z} = \mathbf{ax} + \mathbf{y}$, where $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are n-component vectors and α is a scalar number. The total number of flops (floating point operations) per

iteration is $N = n^2 + 10n$. The dominant first term is contributed by the matrix-vector multiplication $\mathbf{w} = \mathbf{Ap}$.

For large problems, the first term will be several orders of magnitude greater than the second linear term. For this reason, we may be tempted to compute in parallel on the device only the operation $\mathbf{w} = \mathbf{Ap}$.

The matrix-vector multiplication can be done in parallel by computing the elements of **w** as dot products of the rows of A and the vector **p**. The parallel compute time will now be proportional to $2n$ instead of $2n^2$ where n is the vector size. This would mean faster computing and superior scalability. Unfortunately, sharing computation of each iteration by the CPU and the GPU requires data transfers between the CPU and the device, as depicted in Fig. 1.16.

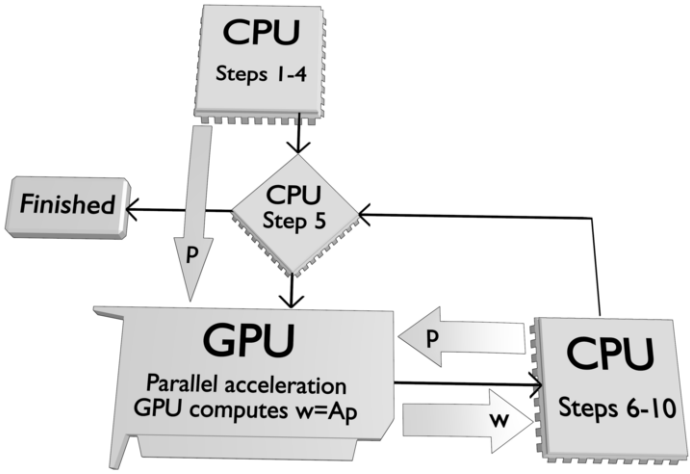

Figure 1.16: The CPU+GPU shared execution of the CGM with data transfer between the CPU and the GPU.

In the CGM algorithm, the values of **p** and **w** change in every iteration and need to be transferred if **q** is computed by the GPU and **d** is computed by the CPU. The matrix **A** remains constant and need not be moved. To avoid data transfers that would seriously degrade performance, it has been decided to compute the entire iteration on the device. It is common practice to regard the CGM as an iterative method, although for a problem of size n the method converges in n iterations if exact arithmetic is used. The speed of the convergence depends upon the condition number for the matrix **A**. For a positive definite matrix, the condition number is the ratio of the largest to the smallest eigenvalues of **A**. The speed of convergence increases as the condition number approaches 1.0. There are methods for improving the conditioning of **A**. The CGM method with improved conditioning of **A** is called the preconditioned CGM and is often used in practical applications. Readers interested in preconditioning techniques and other mathematical aspects of the CGM may find useful information in [6] and [7].

1.6.3. Jacobi Method

Another linear equations solver often used for equations derived from discretizing linear partial differential equations is the stationary iterative Jacobi method. In this method, the matrix **A** is split as follows: $\mathbf{A} = \mathbf{L}+\mathbf{D}+\mathbf{U}$ where **L** and **U** are strictly lower and upper triangular and **D** is diagonal. Starting from some initial approximation x^0, the subsequent approximations are computed from

$$Dx^{k+1} = b - (\mathbf{L} + \mathbf{U})x^k \tag{1.4}$$

This iteration is perfectly parallel. Each element of x can be computed in parallel. Putting it differently, the Jacobi iteration requires a matrix-vector multiplication followed by a SAXPY, both parallizable. Fig. 1.17 shows the Jacobi algorithm computation on a heterogeneous computer.

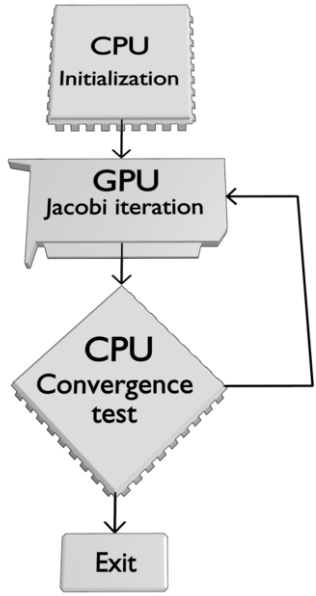

Figure 1.17: The Jacobi algorithm computation on a heterogeneous machine.

Implementing the Jacobi iterative algorithm on a heterogeneous machine can result in a significant acceleration over the sequential execution time because most of the work is done by the massively parallel GPU. The CPU processor must initialize computation before iterations start, and it keeps checking if the solution converged. Closely related to the Jacobi iterative method is the well-known method of Gauss-Seidel, but it is not as easily parallelizable as the Jacobi algorithm. If both methods converge, the Gauss-Seidel converges twice as fast as the Jacobi. Here, there is an interesting dilemma. Which of the two algorithms is preferred – the slower-converging Jacobi parallel algorithm or the faster-converging Gauss-Seidel method, that may

not be as easy to parallelize and run on a heterogeneous machine. The traditional algorithm evaluation criteria may require some modification by adding parallelism as an important algorithm property.

1.6.4. Power Method

The third algorithm considered here is used for computing the leading eigenvalue and the corresponding eigenvector of a large matrix. The algorithm is called the power method and contains only two operations, as shown below.

Initialize (x^0)
1. for $k = 1, 2, ...$
2. $\quad \mathbf{x}^k = \mathbf{A}\mathbf{k}^{k-1}$
3. $\quad \mathbf{x}^k = \mathbf{x}^k / \max \mathbf{x}^k$
4. $\quad$ test convergence
5. end

The power method algorithm.

The power method contains two operations; matrix-vector multiplication and vector normalization. In the normalization step, all elements of the vector **x** are divided by its largest absolute value component. Both computing steps can be executed in parallel by the GPU accelerator. The CPU computation includes only the initialization step plus stopping and printing the converged computations. These computations are minor compared with the parallel operations executed by the GPU. Except one initial transfer of data from the CPU to the GPU no other significant transfers are needed. Assuming that GPU computes the convergence test, it can notify the CPU by sending one bit if convergence is achieved.

1.6.5. Monte Carlo Methods

The last example comes from a large group of Monte Carlo methods. Monte Carlo methods have been applied to problems of physics, financial markets, analysis of queuing systems and engineering. Monte Carlo methods are favored in many high performance computing applications because they are computationally intensive but do not require substantial amounts of communication. They are also perfectly parallelizable. For the purpose of this chapter, the Monte Carlo method for evaluating integrals in multidimensional Euclidean space is considered. The integral is shown in Formula 1.5.

$$\theta = \int \int \int \int \ldots \int Df(x_1, x_2, \ldots, x_d)\, \mathrm{d}x_1\, \mathrm{d}x_2 \ldots \mathrm{d}x_d \tag{1.5}$$

Integral of a multidimensional function.

The problem is finding the integral of the function **f** defined in the d-dimensional Euclidean space. For a multidimensional case, there are no formulas similar to the trapezoidal rule or the Simpson's method used for functions of one variable. The Monte Carlo method for evaluating such an integral is as follows:

a) enclose the region **D** in the parallelepiped whose volume is **V**,
b) generate n random points inside the d-dimensional parallelepiped,
c) count points that fall inside **D**. They are called hits.

Suppose that the total number of hits is **H**. Now the integral of the function f can be estimated from equation 1.5.

$$\Theta = V\frac{H}{n} \tag{1.6}$$

Randomized estimate of the function f integral in equation 1.5.

The described Monte Carlo algorithm is perfectly parallel because the location of each random point can be evaluated independently for all points in parallel. If the algorithm uses n trials and T threads (work items), each thread tests n/T points. After finishing all trials, the threads add their number of individual hits. This is a classic example of the SPMD (Single Program Multiple Data) parallel program.

Monte Carlo methods have been used for optimization problems such as finding optimizers of functions, simulated annealing and genetic algorithms. An introduction to these applications can be found in the book [8].

1.6.6. Conclusions

An important historical note: In the 1970s Richard Hanson of Washington State University, with his colleagues C. Lawson, F. Krogh and D. Kincaid, developed and published standard optimized Basic Linear Algebra Subprograms (BLAS). BLAS have become building blocks for many software systems used in scientific and engineering applications [7]. Examples of BLAS operations are listed in Table 1.2.

Table 1.2: Examples of BLAS Level 1, 2 and 3.

BLAS Level 1
dotprod $\rightarrow x^T y$
SAXPY $\rightarrow \mathbf{ax}+\mathbf{y}$
norm2 $\rightarrow \|x\|^2$

BLAS Level 2
$\mathbf{y}=\mathbf{Ax}$
$\mathbf{y}=\alpha\mathbf{Ax}+\beta\mathbf{y}$
$\mathbf{A}=\alpha\mathbf{xy}^T+\mathbf{B}$

BLAS Level 3
$\mathbf{C}=\mathbf{AB}$
$\mathbf{C}=\mathbf{A}^T\mathbf{B}$
$\mathbf{C}=\mathbf{BT}$
$\mathbf{B}=\alpha\mathbf{T}^{-1}\mathbf{A}$

A complete list of BLAS and a description of Linear Algebra Package LAPACK, ScaLAPAC and Basic Linear Algebra Communication Subprograms can be found in [7]. The portable Parallel Linear Algebra library LAPACK is described in [9].

To sum up, it is evident that general purpose GPU (GPGPU) processing is applicable to a large majority of scientific and engineering numerical algorithms. Most of the numerical algorithms for scientific and engineering problems contain linear algebra building blocks, such as BLAS, whose computing is highly parallelizable. This creates confidence that heterogeneous computation can and will find wide application in scientific and engineering computation, where increasing the flops rates and shortening times to solution are important. A significant benefit is also the CPU+GPU computing superior scalability essential for solving large scale problems.

1.7. Benchmarking CGM

1.7.1. Introduction

Solving sets of linear algebraic equations $\mathbf{Ax} = \mathbf{b}$ is a very common task in applied mathematics. For large general systems, the method of choice is often the classic Gaussian $\mathbf{A} = \mathbf{LU}$ decomposition but with added sparse matrix techniques for storing data and pivoting strategies to minimize matrix fill-ins. In some scientific/engineering applications, the task of using the sparse matrix technology is significantly reduced because the solution algorithms are using the original matrix **A**. This happens if the equations matrix **A** is positive definite. For a system with a positive definite matrix **A**, the solution methods of choice are members of the Krylov algorithms such as the Conjugate Gradient Method (CGM). Please see section 1.6. CGM is an iterative algorithm but with a finite number of iterations n where n is the number of equations if the precise arithmetic could be used. In reality, it is expected that adequate approximations are achieved after a number of iterations much smaller than n.

1.7.2. Additional CGM Description

CGM has two distinct phases. The initial phase includes steps 1-5, and the iterative phase includes steps 6-10. In the initial phase computed are the initial residual vector **r**, the initial search direction vector **p** and a convergence test is applied in step 5. Steps 6 to 9 compute the new solution approximation **x**, the new residual vector **r**, and the new search direction **p**. Each iteration requires computing the matrix-vector product $\mathbf{w} = \mathbf{Ap}$ whose computational complexity is $O(n^2)$. All remaining computations have lower computational complexity $O(n)$. They include two dot products and three SAXPY operations $a \times \mathbf{x} + \mathbf{y}$ where **a** is a scalar and **x**, **y** are vectors. CGM method convergence depends on the matrix **A** condition number $k_2(A) = \lambda_{max}/\lambda_{min}$ where lambdas are the eigenvalues of **A**. Among the many applications of CGM, one is used for magnetic resonance imaging that supports medical diagnose [4]. In one version of the MRI process, the CGM method is used for solving very large sets of sparse linear algebraic equations [4]. In this application, the speed of providing required images is essential. Using heterogeneous computing may be very helpful or even necessary.

1.7.3. Heterogeneous Machine

The heterogeneous computer used to solve CGM contained CPU and devices as shown in Table 1.3.

1.7.4. Algorithm Implementation and Timing Results

The entire CGM iteration is computed in parallel on selected devices. This is a multi-kernel computation, since the CGM algorithm iteration contains several parallel linear algebra operations. Among them are the matrix/vector product, dot products and SAXPYs. Fig. 1.18 shows three execution times. Time is in seconds. N is the number of equations.

Table 1.3: Heterogeneous computer components.

CPU	
Name	Intel Core i7 920
Number of cores	4
Number of threads per core	2
Speed	2.67 GHz
Cache size	8,192 kB

OpenCL view:	
Global memory	3,221 GB
Constant buffer size	65,535 B
Local memory size	32,768 B

Devices:		
Device name	Tesla C2070	Tesla C1060
Device vendor	NVIDIA Corporation	NVIDIA Corporation
Device type	GPU	GPU
Device system	OpenCL 1.0 CUDA	OpenCL 1.0 CUDA
Max compute units	14	30
Max work item dimension	3	3
Max work item sizes	1024/1024/64	512/512/64
Max work group size	1024	512
Device max clock frequency	1147 MHz	1296 MHz
Device address bits	32	32
Max MemAlloc size	1343 MB	1823 MB
Global memory size	5375 MB	4095 MB
Local memory size	48 kB	16 kB
Max constant buffer size	64 kB	64 kB

1. The sequential computation on the CPU is the longest.
2. In the second case, the 2-core CPU (4 logical cores) serves as the OpenCL device.
3. The fastest run was obtained by using the Tesla C2070 GPU as the OpenCL device.

1.7.5. Conclusions

1. The speedup measured by the ratio of sequential to parallel execution times increases with the problem size (the number of equations). Large sets of equations benefit more than smaller ones from the highly parallel heterogeneous computation.
2. Computing the entire CGM iteration on the device turns out to be a good strategy. Relatively slow inter-memories data transfers between iterations were avoided. Only initial data transfer was necessary.

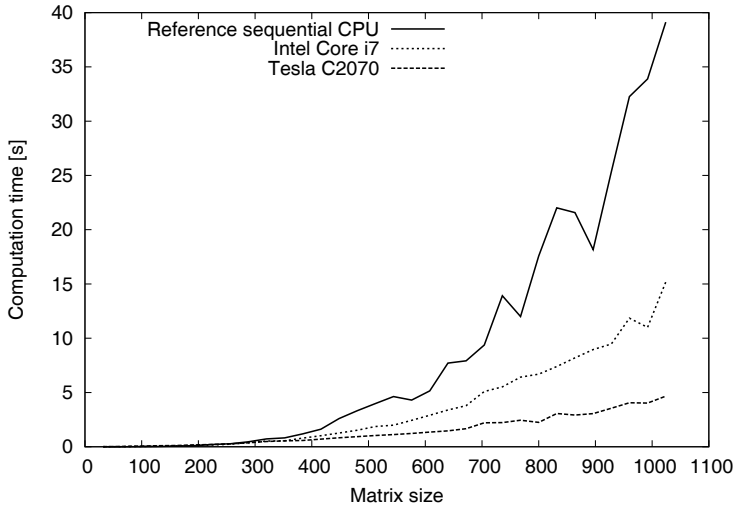

Figure 1.18: Timing of three different cases of CGM computation.

3. An inferior alternative strategy was to compute on the device only the computationally most complex operation, the matrix/vector multiplication **Ap** in step 6.

Our observations in this section may apply to other similar numerical algorithms.

Chapter 2

OpenCL Fundamentals

2.1. OpenCL Overview

2.1.1. What is OpenCL

OpenCL is a software system that lets programmers write a portable program capable of using all resources available in some *heterogeneous* platform. A heterogeneous platform may include multi-core CPUs, one or more GPUs and other compute devices.

In principle, OpenCL can be used also for programming homogeneous systems such as Intel multi-core processors, where one core can be a host and the others provide improved performance by massive parallel processing.

2.1.2. CPU + Accelerators

The majority of current parallel computing systems can be called hardware homogeneous. In homogeneous systems, similar computers or processors are interconnected into a single computer capable of parallel processing. The types of parallel systems include shared memory multiprocessors and clusters of whole computers. In this book, we consider the idea of parallel computation based on heterogeneity in which the system has two or more distinct types of processors: where the ordinary CPU, called the *host*, is responsible for managing functions of one or more highly parallel processors called the *devices* (for example GPU devices controlled by a CPU host system). Devices are responsible for highly parallel processing that accelerates computation. These systems are called heterogeneous because they consist of very different types of processors.

The fundamental idea behind OpenCL is its independence from the vendor hardware and operating systems. This universality and portability has a price. The price is more programming effort and longer learning time.

2.1.3. Massive Parallelism Idea

In the platform definition there is one *host* plus one or more *compute devices* (Fig. 2.1). Usually the host is a CPU. Every compute device is composed of one or

several *Compute Units* (CU) that in turn are divided into *Processing Elements* (PE). This definition is represented in Fig. 2.1.

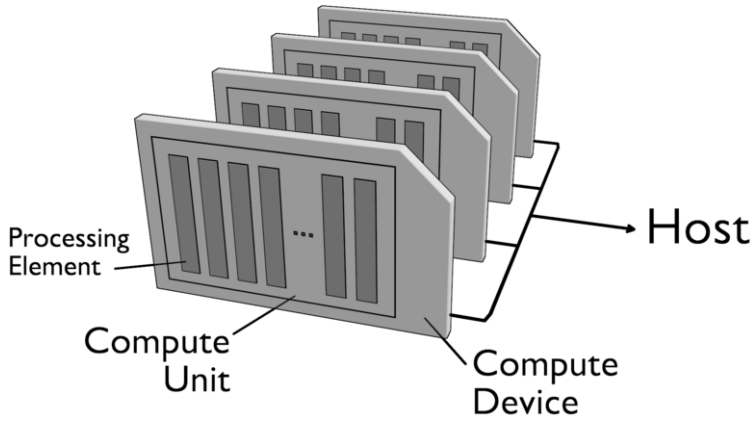

Figure 2.1: The OpenCL platform model as defined by the Khronos Group.

The idea of massive parallelism in OpenCL computing is rooted in the concept of the *kernel* C function. Two programs, a conventional C program and the equivalent OpenCL kernel program, are presented below for comparison. The kernel program is executed on a device.

A conventional C code:

```
void Add(int n, float *a, float *b, float *c)
{
  int i;
  for(i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
```

An equivalent OpenCL kernel program:

```
kernel void
  Add(
    global const float* a,
    global const float* b,
    global float *c
  )
  {
    int id = get_global_id(0);
    c[id] = a[id] + b[id];
  }
/* execute over n work items. */
```

The name work item, in more traditional terminology, means a thread. This kernel code is executed at each point in a data parallel problem domain. For example,

for computing the sum of two matrices of size 1024 by 1024, there will be 1,048,576 kernel executions. In this case the problem domain of work-items is 2 dimensional. Please note that the concept of the C-loop has been replaced by the index space id. For the above vector addition problem, only one-dimensional work items index space is needed.

2.1.4. Work Items and Workgroups

A two-dimensional domain of work-items is shown in Fig. 2.2. Each small square is a workgroup that may have, for example, 128×128 work items. The significance of the workgroup is that it executes together. Also, work items within a group can communicate for coordination. On the other hand, work items in different workgroups can't communicate. For example, work items in work group (0,0) and (0,1) cannot communicate.

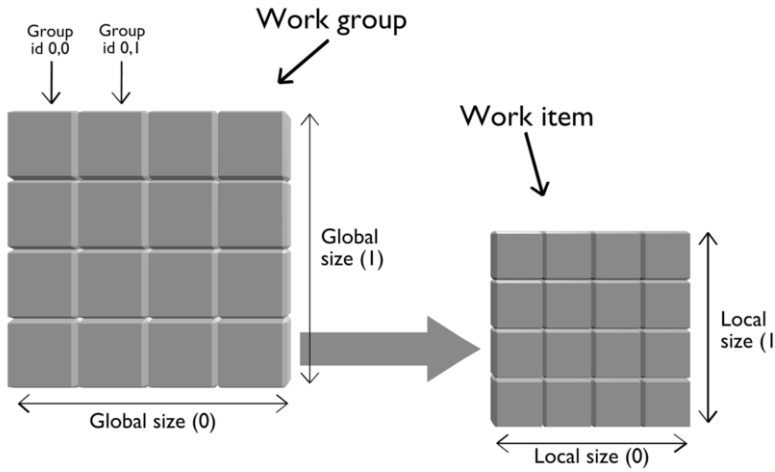

Figure 2.2: OpenCL 2-D work items domain.

2.1.5. OpenCL Execution Model

An OpenCL program runs on a host CPU that submits parallel work to compute devices. In a data parallel case, all work items execute one kernel (a C function). In more general cases the OpenCL host program is a collection of kernels and other functions from the run time APL library. The environment in which work items execute is called the *context*. It includes devices, memories and command queues. An OpenCL parallel execution model and context for managing compute devices is shown in Fig. 2.3.

2.1.6. OpenCL Memory Structure

OpenCL memories are managed explicitly by the programmer, who has to move data from the host to device global memory, or vice versa.

Global/Constant memories are visible to all work-items. They are the largest and slowest memories used by devices. Constant memory is read only for the device but is read/write by the host.

Local memory is sharable within a workgroup. It is smaller but faster than global memory.

Private memory is available to individual work-items.

In addition, there is host memory on the CPU. It is important to remember that memory management is explicit and totally in the hands of the programmer, who is responsible for all data movements. Some of them, like CPU/device transfers, are very expensive. This problem can't be overemphasized. The issue of minimizing data transfers is discussed further in sections 1.4 and B.1.4.

2.1.7. OpenCL C Language for Programming Kernels

The OpenCL C language for programming kernels is a subset of ISO C99 but with some exclusions. These exclusions are: standard C99 headers, function pointers, recursion, variable length arrays and bit fields. On the other hand, there are several additions for work items and workgroups, vector types, synchronization and address space qualifiers.

OpenCL C includes also a number of built-in functions for work-item manipulation and specialized mathematical routines. Data types that can be used for programming kernels include scalars: `char`, `short`, `int`, `long`, `float`, `bool`, etc.; vector data types; different lengths and types of components; vector operations; and built-in functions.

2.1.8. Queues, Events and Context

To submit a kernel for execution by a device, the host program has to create a command queue. This is done by calling an appropriate function in the OpenCL API run time library.

Once the command queue is created, the host can insert a kernel into the queue. When the device is available, it removes the kernel from the head of the queue for execution.

Fig. 2.3 shows the context, including two devices and corresponding queues. If the two kernels at the head of their respective queues are independent, they can execute in parallel on two compute devices. But if it is required that kernel A finish before kernel B can start executing, then this synchronization can be accomplished by using *events*. In this case, the event would be the completion of work by kernel A. The respective algorithm structure would decide if there is a need for the kernels' execution synchronization.

When a computer contains several compute devices, it may be possible to implement task parallelism with double level parallelism. More than one kernel can execute in parallel, and each kernel executes on multiple work-items.

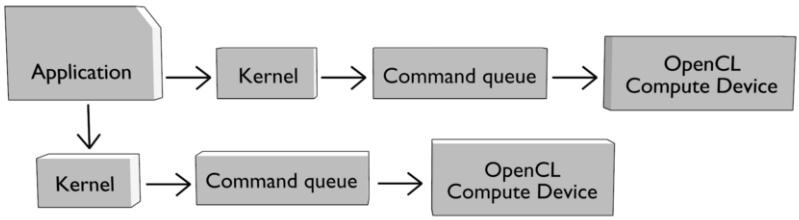

Figure 2.3: Context for managing devices.

2.1.9. Host Program and Kernel

The host is programmed using C/C++ and OpenCL Runtime APIs. The devices run kernels written in the OpenCL C. Fig. 2.4 shows languages for programming an OpenCL platform-compliant heterogeneous computer.

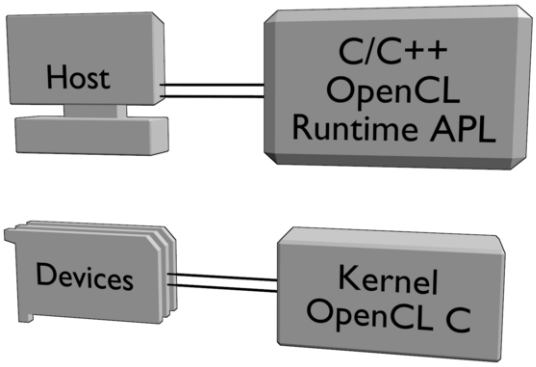

Figure 2.4: Host and device programming languages.

2.1.10. Data Parallelism in OpenCL

After examining the most frequently used scientific and engineering computing algorithms, it becomes quite clear that many of them contain data parallel components. This observation has justified and fueled efforts for developing general purpose applications of GPU processors (GPGPU). Section 1.6 presents several examples.

In data parallel computation, one kernel (a C function) is run on multiple data sets. In this case, OpenCL creates an ID for each work group running on a compute unit (CU) and also an ID for each work item running on a processing element (PE). One or more compute devices and the host constitute together OpenCL Platform Architecture. Heterogeneous computer memory structure is discussed in more detail in section 1.4.

2.1.11. Task Parallelism in OpenCL

In cases where an algorithm contains parallel tasks, different kernels execute on different compute units or processing elements. In this case, the number of work groups and the number of work items are one. Using task parallelism creates the potential for loss of efficiency due to load imbalance. This problem does not exist for data parallel processing, where all tasks are computationally identical but use different data sets. The task parallel processing can be accomplished by creating multiple kernels, but programmers must be aware of the workload balance problem.

2.2. How to Start Using OpenCL

There is need for clarification of terminology. The OpenCL standard provides definitions of multiple terms that are used in the book. The most important are [10]:

Program

An OpenCL program consists of a set of kernels. Programs may also contain auxiliary functions called by the **`kernel`** functions and constant data.

Application

The combination of the program running on the host and the OpenCL devices.

Buffer Object

A memory object that stores a linear collection of bytes. Buffer objects are accessible using a pointer in a kernel executing on a device. Buffer objects can be manipulated by the host, using OpenCL API calls.

OpenCL allows for computation in heterogeneous environments, so it is natural to define multiple levels of the standard – the platform, runtime and compiler. The OpenCL standard defines these three components as [10]:

OpenCL Platform layer

The platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts.

OpenCL Runtime

Runtime allows the host program to manipulate contexts once they have been created.

OpenCL Compiler

The OpenCL compiler creates program executables that contain OpenCL kernels. OpenCL C programming language, implemented by the compiler, supports a subset of the ISO C99 language with extensions for parallelism.

OpenCL libraries are usually included in hardware drivers. OpenCL application does not need any special compiler. The preferred language for writing OpenCL applications is C or C++, but it is also possible to use this technology with any language that supports linking to binary libraries. There are a number of wrapper libraries for many other languages. The most significant ones are:

- JOCL - the library enabling usage of OpenCL in Java applications [11].

- PyOpenCL - OpenCL binding for the Python programming language [12].
- Ruby-OpenCL - binding for Ruby [13].

OpenCL programs are usually compiled by a compiler included in drivers during application runtime. There is also the possibility to store a compiled program, but it will work only on the hardware it was compiled for. This functionality is intended for caching programs to speed up application startup time and for embedded solutions.

2.2.1. Header Files

In most cases, only one header file has to be included – `cl.h` or `cl.hpp`. For convenience, the `opencl.h` header file is available; it includes all of the OpenCL header files.

The standard file that contains platform-specific definitions and macros is called `cl_platform.h`. This file also includes type definitions.

The `cl.h` header file contains all basic API calls definitions. It also includes `cl_platform.h`, so the platform-specific definitions are also included. There are also definitions of error codes, types, and definition of constant values used in API calls. Many applications use only this header. Using `cl.h` header file is a perfect solution if application uses only core OpenCL API without extensions.

The next header is `cl_ext.h`. This file contains the extensions that do not need to be implemented. These extensions are vendor-dependent. Currently there are functions specifically for hardware from AMD, Apple and NVIDIA.

If the application requires DirectX connectivity, then the header `cl_d3d10.h` should be included. This header contains standard extension functions definitions. The most important functions from this header file are dedicated for sharing OpenCL memory buffers with DirectX, so there is no need for the application to transfer memory via PCIex bus.

The header files `cl_gl.h` and `cl_gl_ext.h` contain OpenCL extensions for OpenGL. The first one contains standard definitions, the second contains vendor-specific extensions which are usually available on some particular implementations. These header files define functions similar to the ones from `cl_d3d10.h`. Most of them are used for exchanging texture data between OpenCL and OpenGL.

The file `cl.hpp` contains macros, function and class definitions for applications written in C++. This file is actually a wrapper. There are class definitions containing inline methods that invoke standard C OpenCL API calls. This header file uses C++ language properties extensively, so almost every programmer familiar with this programming language will save time that would otherwise be spent for programming such wrapper. This is very convenient for C++ programmers, because it provides official, consistent and well documented API.

The usual way of including core OpenCL header files is shown in listing 2.1 for C and in listing 2.2 for C++. Most of examples in this book use this header file.

2.2.2. Libraries

OpenCL allows for the presence and usage of many implementations on one system. There is only one dynamically linked library file with the name OpenCL. This library contains functions that allow applications to get the list of available OpenCL

```
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif
```

Listing 2.1: Common include section of OpenCL application – C code.

```
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif
```

Listing 2.2: Common include section of OpenCL application – C++ code.

platforms and select one. Different OpenCL implementations can use different library names, but an application needs to be linked to `libOpenCL` or `OpenCL.dll` or `OpenCL.framework` – or whatever library format is supported.

2.2.3. Compilation

OpenCL is platform-independent and does not need any additional compilers. The only thing a programmer must perform in order to compile his application successfully is to link it against the OpenCL library. The common library name is OpenCL, with prefixes and suffixes adequate to a given operating system. The example compilations using free compilers on different operating systems would be:

1. The example compilation on Linux is:

   ```
   gcc SomeExample.c -o SomeExample -lOpenCL
   ```

2. The compilation on Windows MinGW would look like:

   ```
   gcc SomeExample.c -o SomeExample \
     -lOpenCL -static-libstdc++ -static-libgcc
   ```

3. And the compilation on Mac would be:

   ```
   gcc SomeExample.c -o SomeExample -framework OpenCL
   ```

2.3. Platforms and Devices

To describe the OpenCL context and many other elements of this standard, some terms have to be introduced. The OpenCL standard defines a platform as:

Platform

The host plus a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform [10].

In practice, the platform is an implementation of the OpenCL standard from some vendor. OpenCL allows for coexistence of multiple platforms on one host. The overview schematics of multiple platforms on one host are depicted in figure 2.5.

To get the list of available platforms, the host program has to execute **clGetPlatformIDs**. An example can be seen in listing 2.3. This example gets the list of available platforms and displays the number of them. Note that the function **clGetPlatformIDs** is executed twice. This is because the first one gets the number of platforms, and the second one just gets the actual list of platform IDs. In some circumstances, it is easier to just assume that there are no more than X platforms and use statically allocated memory for a list of them.

clGetPlatformIDs

```
cl_int clGetPlatformIDs ( cl_uint num_entries, cl_platform_id *platforms,
    cl_uint *num_platforms);
```

Obtain the list of available platforms.

clGetPlatformInfo

```
cl_int clGetPlatformInfo ( cl_platform_id platform, cl_platform_info
    param_name, size_t param_value_size, void *param_value, size_t
    *param_value_size_ret);
```

Get specific information about the OpenCL platform.

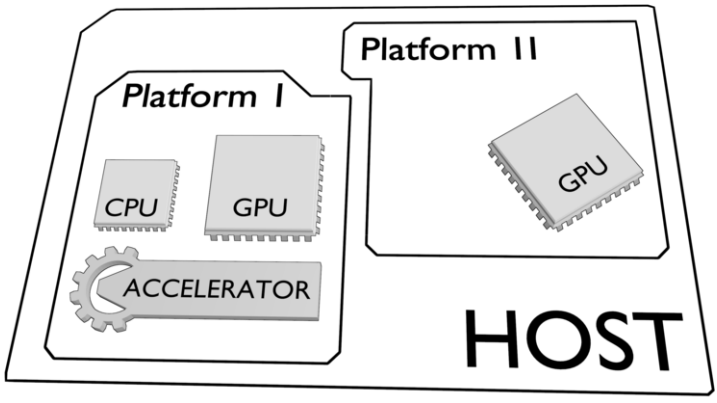

Figure 2.5: Different platforms on one host.

```
cl_platform_id *platforms;
cl_uint platforms_n;
clGetPlatformIDs(0, NULL, &platforms_n);
platforms = (cl_platform_id *)malloc(sizeof(cl_platform_id) * platforms_n);
clGetPlatformIDs(platforms_n, platforms, &platforms_n);
printf("There are %d platforms.\n\n", platforms_n);
```

Listing 2.3: Getting the list of available platforms – C code.

```
char ret [1024];
clGetPlatformInfo(platforms [i], CL_PLATFORM_VENDOR, 1024, ret, NULL);
printf("CL_PLATFORM_VENDOR: %s\n", ret);
clGetPlatformInfo(platforms [i], CL_PLATFORM_NAME, 1024, ret, NULL);
printf("CL_PLATFORM_NAME: %s\n", ret);
clGetPlatformInfo(platforms [i], CL_PLATFORM_VERSION, 1024, ret, NULL);
printf("CL_PLATFORM_VERSION: %s\n", ret);
clGetPlatformInfo(platforms [i], CL_PLATFORM_PROFILE, 1024, ret, NULL);
printf("CL_PLATFORM_PROFILE: %s\n", ret);
clGetPlatformInfo(platforms [i], CL_PLATFORM_EXTENSIONS, 1024, ret, NULL);
printf("CL_PLATFORM_EXTENSIONS: %s\n", ret);
```

Listing 2.4: Getting information about particular OpenCL platform.

2.3.1. OpenCL Platform Properties

OpenCL API allows the user to obtain information about each platform by the **clGetPlatformInfo** API call. An example can be seen in listing 2.4.

In the example, the static buffer has been used, but for actual applications it should be allocated dynamically. The function **clGetPlatformInfo** fills the buffer (in the example, it is called *ret*) with value corresponding to a specified platform parameter. All the OpenCL 1.2 parameters available are used in the example. The most commonly used constants are CL_PLATFORM_VERSION and CL_PLATFORM_EXTENSIONS. The first one provides a version string identifying the given platform. The format of this string must be in the following format:

OpenCL<space><major_version>.<minor_version><space><platform-specific information>

For the platforms implementing the OpenCL 1.2 standard, the <major_version>.<minor_version> must be 1.2. This platform parameter is very important when the application uses some OpenCL standard features not available in previous versions. This method allows for checking if system requirements are met for the given application.

The second platform parameter CL_PLATFORM_EXTENSIONS allows for checking what extensions are available. The usual case is to check if platform supports double-precision computation. The extension list is returned as a null terminated string. Implementation specific extensions can also be listed.

The CL_PLATFORM_PROFILE allows for checking the implementation profile. The profile can be FULL_PROFILE or EMBEDDED_PROFILE. The first one informs that the given platform implements the full OpenCL standard. This does not provide informa-

tion about any extensions supported, but assures that every standard feature is available. The second profile type is used in small implementations available on hand-held devices or embedded systems. It must implement a specific subset of the OpenCL standard.

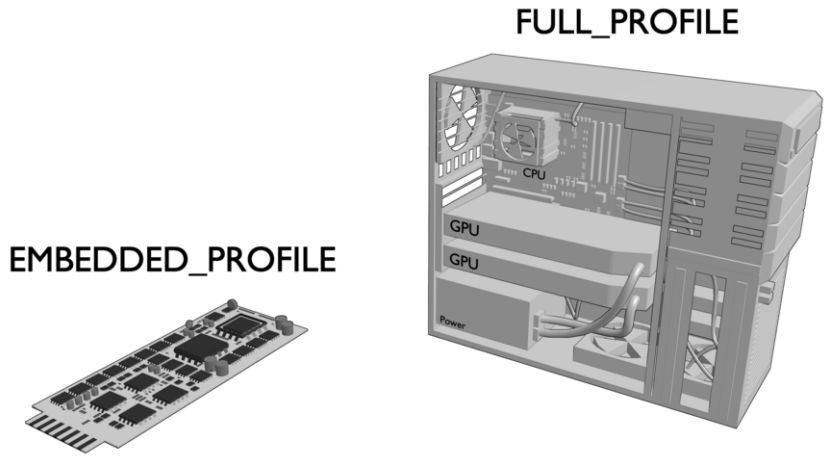

Figure 2.6: Graphical explanation of the difference between `FULL_PROFILE` and `EMBEDDED_PROFILE`.

The most significant difference is that the embedded profile does not need to contain the OpenCL compiler. The book focuses on the full profile, because it covers the widest range of applications.

2.3.2. Devices Provided by Platform

clGetDeviceIDs

```
cl_int clGetDeviceIDs ( cl_platform_id platform, cl_device_type device_type,
    cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices );
```

Obtain the list of devices available on a platform.

Each platform contains devices. A device is a representation of actual hardware present in a computer. Getting the list of available devices, along with information about each, is shown in listing 2.5. The function **`clGetDeviceIDs`** is used for getting the list of devices in platform. The platform ID is passed in parameter *p*. The value *CL_DEVICE_TYPE_ALL* orders this function to list all available devices. This parameter is a bitfield of type `cl_device_type`. It also accepts one of the following values or a combination thereof:

- *CL_DEVICE_TYPE_CPU* – lists only processors which are located in the host. For this kind of device every memory buffer resides in the host memory. This kind of device is best suited for algorithms that use small numbers of work-items with high amounts of data to be transferred between the host and the device.

```
int i;
cl_device_id *devices;
cl_uint devices_n;
clGetDeviceIDs(p, CL_DEVICE_TYPE_ALL, 0, NULL, &devices_n);
cl_uint ret_st;
cl_ulong ret_ul;
devices = (cl_device_id *)malloc(sizeof(cl_device_id) * devices_n);
clGetDeviceIDs(p, CL_DEVICE_TYPE_ALL, devices_n, devices, &devices_n);
for (i = 0; i < devices_n; i++) {
  char ret [1024];
  printf("Platform %d , device %d\n", index, i);
  clGetDeviceInfo(devices [i], CL_DEVICE_NAME, 1024, ret, NULL);
  printf("CL_DEVICE_NAME: %s\n", ret);
  clGetDeviceInfo(devices [i], CL_DEVICE_VENDOR, 1024, ret, NULL);
  printf("CL_DEVICE_VENDOR: %s\n", ret);
  clGetDeviceInfo(devices [i], CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(ret_st),
    &ret_st,
    NULL);
  printf("CL_DEVICE_MAX_COMPUTE_UNITS: %d\n", ret_st);
  clGetDeviceInfo(devices [i], CL_DEVICE_MAX_CLOCK_FREQUENCY, sizeof(ret_st),
    &ret_st,
    NULL);
  printf("CL_DEVICE_MAX_CLOCK_FREQUENCY: %d\n", ret_st);
  clGetDeviceInfo(devices [i], CL_DEVICE_LOCAL_MEM_SIZE, sizeof(ret_ul), &ret_ul,
    NULL);
  printf("CL_DEVICE_LOCAL_MEM_SIZE: %lu\n", (unsigned long int)ret_ul);
  clGetDeviceInfo(devices [i], CL_DEVICE_GLOBAL_MEM_SIZE, sizeof(ret_ul), &ret_ul,
    NULL);
  printf("CL_DEVICE_GLOBAL_MEM_SIZE: %lu\n", (unsigned long int)ret_ul);
}
free(devices);
```

Listing 2.5: Getting the list of available devices for a given platform and information about each of them – C code.

- *CL_DEVICE_TYPE_GPU* – lists only graphic cards. This device is most commonly used for OpenCL computation. These devices are well suited for massively parallel tasks. They must also be able to accelerate 3D APIs such as OpenGL or DirectX. A GPU can be used for graphical filters and generation of some special effects, because OpenGL or DirectX can share buffers with OpenCL.
- *CL_DEVICE_TYPE_ACCELERATOR* – lists available accelerators. An accelerator is a device dedicated to OpenCL computations. It can be much more powerful than a GPU and is best suited for massively parallel tasks. An accelerator does not provide monitor output and cannot cooperate with graphic libraries.
- *CL_DEVICE_TYPE_CUSTOM* – Dedicated accelerators that do not support programs written in OpenCL C. This type of device was introduced in OpenCL version 1.2.
- *CL_DEVICE_TYPE_DEFAULT* – this allows the OpenCL platform to choose a default device. It can be of any type. This is a good starting point for prototyp-

ing software when OpenCL program correctness, and not performance, need testing.

This bitfield is also used during context initialization; an example and a description for this process is described in section 2.5. There is also more detailed description of different device types.

clGetDeviceInfo

```
cl_int clGetDeviceInfo ( cl_device_id device, cl_device_info param_name,
    size_t param_value_size, void *param_value, size_t *param_value_size_ret );
```

Get information about an OpenCL device.

Each device is accompanied by a set of properties. This information can be obtained using **clGetDeviceInfo**. The example in listing 2.5 shows usage of this function. This information is intended for the OpenCL application to choose the best device for its purposes. This information can also be presented to the end user, allowing him to select manually the device he wants to use for computation.

The full example of querying platforms and devices is on the attached disk. Sample output for this program can be seen in listing 2.6. This example shows that the coexistence of two OpenCL platforms from different companies is possible. Listed is a CPU form AMD and a GPU from NVIDIA.

```
1  There are 2 platforms.
2
3  Platform 0
4  CL_PLATFORM_VENDOR: Advanced Micro Devices, Inc.
5  CL_PLATFORM_NAME: ATI Stream
6  CL_PLATFORM_VERSION: OpenCL 1.1 ATI-Stream-v2.3 (451)
7  CL_PLATFORM_PROFILE: FULL_PROFILE
8  CL_PLATFORM_EXTENSIONS: cl_khr_icd cl_amd_event_callback cl_amd_offline_devices
9  Platform 0, device 0
10 CL_DEVICE_NAME: AMD Phenom(tm) II X2 550 Processor
11 CL_DEVICE_VENDOR: AuthenticAMD
12 CL_DEVICE_MAX_COMPUTE_UNITS: 2
13 CL_DEVICE_MAX_CLOCK_FREQUENCY: 3100
14 CL_DEVICE_LOCAL_MEM_SIZE: 32768
15 CL_DEVICE_GLOBAL_MEM_SIZE: 1073741824
16
17 Platform 1
18 CL_PLATFORM_VENDOR: NVIDIA Corporation
19 CL_PLATFORM_NAME: NVIDIA CUDA
20 CL_PLATFORM_VERSION: OpenCL 1.0 CUDA 4.0.1
21 CL_PLATFORM_PROFILE: FULL_PROFILE
22 CL_PLATFORM_EXTENSIONS: cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
       cl_nv_compiler_options cl_nv_device_attribute_query cl_nv_pragma_unroll
23 Platform 1, device 0
24 CL_DEVICE_NAME: GeForce GTS 250
25 CL_DEVICE_VENDOR: NVIDIA Corporation
26 CL_DEVICE_MAX_COMPUTE_UNITS: 16
27 CL_DEVICE_MAX_CLOCK_FREQUENCY: 1836
28 CL_DEVICE_LOCAL_MEM_SIZE: 16384
29 CL_DEVICE_GLOBAL_MEM_SIZE: 536150016
```

Listing 2.6: Sample output for device query

2.4. OpenCL Platforms – C++

The examples in this section are analogous to the ones presented in section 2.3. The details related to the platform definition and some constant values are in section 2.3, so it will not be thoroughly described here.

In the OpenCL C++ API, the list of platforms can be obtained using the method **cl::Platform::get**. This method sets the vector of `cl::Platform` to the list of available platforms. In listing 2.7, there is an example of the host code that displays the platforms count.

cl::Platform::get

```
static cl_int cl::Platform::get ( VECTOR_CLASS<Platform> *platforms );
```

Retrieves the list of available platforms.

The information about each platform can be retrieved using the **cl::Platform::getInfo** method. This method is a template and returns information according to the template type name. The code in listing 2.8 prints platform details into standard output. Note that, contrary to the C version of this example (listing 2.4), the C++ code does not involve any helper variables – making the code even more clear.

cl::Platform::getInfo

```
cl_int cl::Platform::getInfo ( cl_platform_info name, STRING_CLASS *param );
```

Gets specific information about the OpenCL platform.

The list of available devices for a platform can be obtained using the method **cl::Platform::getDevices**. This method selects the devices matching selected device class and puts them into the vector of `cl::Device` values. The vector is provided as a pointer. The example in that can be seen in 2.9 shows how to print the list of devices available in the platform with a short description of each. The variable *p* is an object of the class `cl::Platform`. This code selects devices of any type because of `CL_DEVICE_TYPE_ALL`. Then it displays the information returned by the method

```
1 std::vector < cl::Platform > platforms;
2 cl::Platform::get(&platforms);
3 std::cout << "There are " << platforms.size() << " platforms." << std::endl <<
4 std::endl;
```

Listing 2.7: Getting the list of available platforms – C++ code.

```
std::cout << "CL_PLATFORM_VENDOR: " <<
platforms [j].getInfo <CL_PLATFORM_VENDOR> () << std::endl;
std::cout << "CL_PLATFORM_NAME: " <<
platforms [j].getInfo < CL_PLATFORM_NAME > () << std::endl;
std::cout << "CL_PLATFORM_VERSION: " <<
platforms [j].getInfo < CL_PLATFORM_VERSION > () << std::endl;
std::cout << "CL_PLATFORM_PROFILE: " <<
platforms [j].getInfo < CL_PLATFORM_PROFILE > () << std::endl;
std::cout << "CL_PLATFORM_EXTENSIONS: " <<
platforms [j].getInfo < CL_PLATFORM_EXTENSIONS > () << std::endl;
```

Listing 2.8: Getting the information about platform – C++ example

`cl::Device::getInfo`. The template type name determines what information is obtained.

`cl::Platform::getDevices`

```
cl_int cl::Platform::getDevices ( cl_device_type type, VECTOR_CLASS<Device>
    *devices );
```

Retrieves the list of available devices on the OpenCL platform.

`cl::Device::getInfo`

```
template <typename T>cl_int cl::Device::getInfo ( cl_device_info name,
    T *param );
```

Gets specific information about an OpenCL device.

2.5. OpenCL Context to Manage Devices

This section focuses on creation and management of OpenCL contexts. The OpenCL context provides consistent way of performing computations. The context can be seen as an abstraction for computation in heterogeneous environments. Every imaginable object necessary for computation is available. The OpenCL standard defines context as:

Context

The environment within which the kernels execute and the domain in which synchronization and memory management are defined. The context includes a set of devices, the memory accessible to those devices, the corresponding memory properties and one or more command-queues used to schedule execution of a kernel(s) or operations on memory objects. [10]

Every device within a context can communicate with other devices in the same context via memory buffers. The buffers can be shared. There is no direct method for memory transfer between different contexts. The same kernel code can be compiled for any or all devices within a context. This construction allows for better-fitting

```
std::vector < cl::Device > devices;
p.getDevices(CL_DEVICE_TYPE_ALL, &devices);
if (devices.size() > 0) {
  for (size_t j = 0; j < devices.size(); j++) {
    std::cout << "Platform " << index << " , device " << j << std::endl;
    std::cout << "CL_DEVICE_NAME: " << devices [j].getInfo <CL_DEVICE_NAME> () <<
    std::endl;
    std::cout << "CL_DEVICE_VENDOR: " <<
    devices [j].getInfo < CL_DEVICE_VENDOR > () << std::endl;
    std::cout << "CL_DEVICE_MAX_COMPUTE_UNITS: " <<
    devices [j].getInfo < CL_DEVICE_MAX_COMPUTE_UNITS > () << std::endl;
    std::cout << "CL_DEVICE_MAX_CLOCK_FREQUENCY: " <<
    devices [j].getInfo < CL_DEVICE_MAX_CLOCK_FREQUENCY > () << std::endl;
    std::cout << "CL_DEVICE_LOCAL_MEM_SIZE: " <<
    devices [j].getInfo < CL_DEVICE_LOCAL_MEM_SIZE > () << std::endl;
    std::cout << "CL_DEVICE_GLOBAL_MEM_SIZE: " <<
    devices [j].getInfo < CL_DEVICE_GLOBAL_MEM_SIZE > () << std::endl;
  }
}
```

Listing 2.9: Getting the list of available devices for each platform along with some basic description – C++ code.

computation into existing hardware – for example some algorithms execute faster on the CPU and others on the GPU. Figure 2.7 shows a graphical explanation of different OpenCL objects in context.

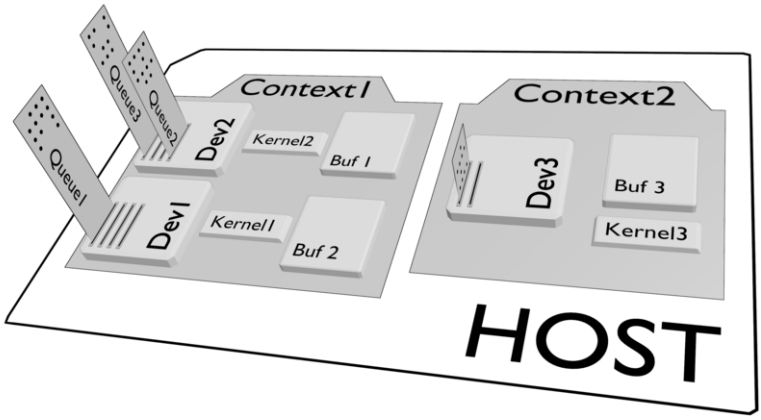

Figure 2.7: Multiple contexts defined on one host.

Even though buffers can be shared among different kinds of devices, the programmer has to be aware that there is the possibility that actual data will be transferred via PCI-E bus or system bus in order to deliver it to different devices. The

OpenCL standard provides only an abstraction for this construct, and different implementations can provide different solutions for it.

Note that in the picture 2.7 there are multiple contexts on one host. This construction is possible because the OpenCL standard does not limit the number of contexts created by the host program. This is most useful when there are multiple platforms available for different hardware.

2.5.1. Different Types of Devices

OpenCL defines enumerated type `cl_device_type`. This type is used for device classification and consists of the following values: `CL_DEVICE_TYPE_CPU`, `CL_DEVICE_TYPE_GPU`, `CL_DEVICE_TYPE_ACCELERATOR`, `CL_DEVICE_TYPE_DEFAULT`, `CL_DEVICE_TYPE_CUSTOM` and `CL_DEVICE_TYPE_ALL`.

Every kind of device matches the `CL_DEVICE_TYPE_ALL`. This is often used when there is need for selection of all devices. The `CL_DEVICE_TYPE_DEFAULT` is for selecting the default device in the system. It is used mostly for prototyping software. The OpenCL implementation decides which device is the default and, as such, should not be used in quality software.

2.5.2. CPU Device Type

The `CL_DEVICE_TYPE_CPU` corresponds to the central processing unit located in the host. The standard defines it as:

> **CL_DEVICE_TYPE_CPU**
>
> An OpenCL device that is the host processor. The host processor runs the OpenCL implementations and is a single or multi-core CPU. [10]

This is the same device as the one executing the host code. It is able to execute the same kernels as any other OpenCL device, but it also provides a very convenient and efficient way for sharing memory buffers.

This device has direct access to the host memory. OpenCL buffers can use the host memory as storage that can be directly accessed from both the host and the device code.

The CPU device can even be used for ordinary sequential algorithms. In some circumstances, the code executed via the OpenCL CPU device can run faster than the same algorithm implemented as the host code. That is because the OpenCL compiler can tune the OpenCL program for given hardware. OpenCL implementation can be created by the same company that provides the CPU, so it can incorporate knowledge about some features unavailable in other CPUs for optimization. If no optimization is turned on in the host C compiler, then it is almost certain that the compiler embedded in the OpenCL implementation will produce faster code.

2.5.3. GPU Device Type

OpenCL specification defines GPU as:

> **CL_DEVICE_TYPE_GPU**
> An OpenCL device that is a GPU. By this we mean that the device can also be used to accelerate a 3D API such as OpenGL or DirectX. [10]

The devices matching `CL_DEVICE_TYPE_GPU` `cl_device_type` are usually graphic cards. These devices are not as fast as CPUs in sequential algorithm execution but can be very efficient in massively parallel tasks. The GPU by definition can also accelerate 3D API such as OpenGL or Direct3D.

This kind of device allows for highly parallel kernel execution. For the hardware available during the time this book was written, GPUs were capable of executing more than 3000 work-items in parallel. This allows for standard algebra algorithms to run multiple times faster than on the CPU. For example, multiplication of matrices can be done in $O(n)$ time for matrices smaller than 54x54, or a dot product in $O(1)$ for vectors of size less than 3000.

Another feature of the GPU device is that it often allows for cooperation between OpenCL and graphic standards such as OpenGL and DirectX. The textures and other data can occupy regions in the GPU memory that can be shared between OpenCL programs and graphic APIs. This feature allows not only for generic computations but also for fast image processing with results seen in real time as well.

There is also a class of devices that are `CL_DEVICE_TYPE_GPU`s but do not allow for video output to a monitor. This is contrary to intuition, but not to the definition. Some Tesla accelerators do not output graphics to a screen but do support OpenGL and DirectX, so by definition they are GPUs.

2.5.4. Accelerator

Accelerators are dedicated OpenCL processors. By definition, an accelerator is:

> **CL_DEVICE_TYPE_ACCELERATOR**
> A dedicated OpenCL accelerator (for example, the IBM CELL Blade). These devices communicate with the host processor using a peripheral interconnect such as PCIe. [10]

These are devices that do not support graphic acceleration and are not processors on the host. This device type is only for computation. This distinction from the other two device types is necessary, because the accelerator does not use the same address space as the CPU on the host and cannot cooperate with graphic libraries. It is just for computation and nothing else.

2.5.5. Different Device Types – Summary

The programmer should consider the algorithm construction before choosing one particular device type. The CPU is best for traditional algorithms that run on OpenMP or threads. Massively parallel computations such as operations on big matrices, or particle calculation, can run fast on a GPU or ACCELERATOR. If the application uses OpenCL for one of the graphic generation stages, then the GPU would be the best choice.

The reader should note that the device type does not provide any information about performance of a given device. This information can be obtained using, for example, a benchmark. Sometimes it is good practice to let the end-user decide which available device to use. The device type should be used as information for the application about capabilities and architecture of hardware. The most common situation in whichj the device type is important is when OpenCL is used for graphic filters – a GPU would be best because it allows for cooperation with graphic APIs.

2.5.6. Context Initialization – by Device Type

The simplest way of creating an OpenCL context is by using the function **clCreateContextFromType**. An example function for creating context in this way can be seen in listing 2.10. This method is commonly used in cases where any device from the selected class is capable of efficient algorithm computation or prototyping applications.

clCreateContextFromType

```
cl_context clCreateContextFromType ( const cl_context_properties *properties,
    cl_device_type device_type, void (CL_CALLBACK *pfn_notify) (const char
    *errinfo, const void *private_info, size_t cb, void *user_data),
    void *user_data, cl_int *errcode_ret);
```

Create an OpenCL context from a device type that identifies the specific device(s) to use.

This function allows for choosing a class of devices and including it into a newly created context. The device type (*devtype* in the example) can be one or more device types joined with the "OR" operator. For example, parameter *devtype* can be CL_DEVICE_TYPE_GPU | CL_DEVICE_TYPE_ACCELERATOR or CL_DEVICE_TYPE_ALL. Different device types were described in section 2.5.1.

This function also requires an array of cl_context_properties for context setup. This array specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0 [10]. If this parameter is NULL, then the behavior of this function is implementation-specific. Some OpenCL implementations do not allow for a NULL value here. In the example, the list contains only the platform identifier. OpenCL 1.2 accepts multiple other possible properties that allow for better cooperation between OpenCL and other libraries like OpenGL or DirectX. Some of these properties will be described in section 3.4.

The platform ID is obtained using the function **clGetPlatformIDs**. This function is used for getting the list of available platforms. In the example, it writes the platform list into the array *platforms*. The size is limited by $i+1$, because the platform is selected by its index in the platform list.

Note that the platform provides an environment for contexts. It is impossible to include devices from different platforms within one context.

The last three parameters for the function **clCreateContextFromType** are responsible for selecting the callback function and for the error code. The error code is not checked in the example for reasons of simplicity. Error handling is described in section 2.7.

```
cl_context createContextFromType(cl_device_type devtype, int i) {
  cl_context context;
  cl_platform_id *platforms;
  cl_uint platforms_n;
  cl_context_properties cps [3] = { 0, 0, 0 };
  platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id) * (i + 1));
  clGetPlatformIDs(i + 1, platforms, &platforms_n);
  cps [0] = CL_CONTEXT_PLATFORM;
  cps [1] = (cl_context_properties)(platforms [i]);
  context = clCreateContextFromType(cps, devtype, NULL, NULL, NULL);
  free(platforms);
  return context;
}
```

Listing 2.10: OpenCL context initialization using **clCreateContextFromType**

```
cl_context createContextFromIndex(int pidx, int didx) {
  cl_context context;
  cl_platform_id *platforms;
  cl_device_id *devices;
  cl_context_properties cps [3] = { 0, 0, 0 };
  devices = (cl_device_id *)malloc(sizeof(cl_device_id) * (didx + 1));
  platforms = (cl_platform_id *)malloc(sizeof(cl_platform_id) * (pidx + 1));
  clGetPlatformIDs(pidx + 1, platforms, NULL);
  clGetDeviceIDs(platforms [pidx], CL_DEVICE_TYPE_ALL, didx + 1, devices, NULL);
  cps [0] = CL_CONTEXT_PLATFORM;
  cps [1] = (cl_context_properties)(platforms [pidx]);
  context = clCreateContext(cps, 1, &devices [didx], NULL, NULL, NULL);
  free(devices);
  free(platforms);
  return context;
}
```

Listing 2.11: OpenCL context initialization using **clCreateContext**

2.5.7. Context Initialization – Selecting Particular Device

Another way of creating an OpenCL context is by using the function **clCreateContext**. This function takes the list of devices and the selected platform to produce a configured OpenCL context.

clCreateContext

```
cl_context clCreateContext ( const cl_context_properties *properties,
    cl_uint num_devices, const cl_device_id *devices, (void CL_CALLBACK
    *pfn_notify) (const char *errinfo, const void *private_info,
    size_t cb, void *user_data), void *user_data, cl_int *errcode_ret );
```

Creates an OpenCL context.

The example helper function **createContextFromIndex** that is creating a context with one device can be seen in listing 2.11. In this example, parameter *pidx* is

the index of the platform on the platform list and parameter *didx* is the index of the device within the platform. The OpenCL standard does not specify if the order of device IDs will remain the same between querying. In the example, this assumption is made for simplicity's sake. In most OpenCL implementations, it works this way. For production software, the best way is to keep the name of the selected device in the configuration file and compare it with the *devname* from consecutive calls:

```
1 clGetDeviceInfo(devID[i],CL_DEVICE_NAME, 1024, devname ,NULL);
```

In listing 2.11, the list of available platform IDs is obtained using **clGetPlatformIDs**. It stores this list in the array *platforms*, which is of size *pidx*+1.

The array *devices* is filled with device IDs for the selected platform, using function **clGetDeviceIDs**. This list is of size *didx*+1.

So the selected IDs of the platform and the device are *platforms[pidx]* and *devices[didx]*. These values are used as parameters to the function **clCreateContext**. The ID *platforms[pidx]* is passed indirectly using the array *cps* of type `cl_context_properties`. The first parameter of the **clCreateContext** function is the same as in the function **clCreateContextFromType** – the context properties. The second parameter is the selected device ID. The last three parameters can be used for error handling but in the example are set to NULL. Error handling is described in section 2.7.

After the context is created, the selected devices in the context remain the same. There is no way to add a device to a context or to remove any device from it.

2.5.8. Getting Information about Context

The function **clGetContextInfo** is used for getting information about an OpenCL context. The example for displaying the list of devices in a given context can be seen in listing 2.12. The function **clGetContextInfo** takes the enumeration constant value to determine which parameter to retrieve. In the example, there are two of them – `CL_CONTEXT_NUM_DEVICES` and `CL_CONTEXT_DEVICES`. The first one means that the number of devices included in the context should be retrieved, and the second is for querying the list of device IDs.

clGetContextInfo

```
cl_int clGetContextInfo ( cl_context context, cl_context_info param_name,
    size_t param_value_size, void *param_value, size_t *param_value_size_ret);
```

Query information about a context.

The number of devices in a context is used for setting the size of array for storing the list of device IDs. The loop is used for displaying some basic information about each device using **clGetDeviceInfo**.

```
void listDevicesInContext(cl_context context) {
  int i;
  cl_device_id *devices;
  cl_uint devices_n = 0;
  clGetContextInfo(context, CL_CONTEXT_NUM_DEVICES, sizeof(devices_n), &devices_n,
    NULL);
  if (devices_n > 0) {
    char ret [1024];
    devices = (cl_device_id *)malloc(sizeof(cl_device_id) * (devices_n + 1));
    clGetContextInfo(context, CL_CONTEXT_DEVICES, devices_n * sizeof(cl_device_id),
      devices,
      NULL);
    for (i = 0; i < devices_n; i++) {
      printf("Device %d\n", i);
      clGetDeviceInfo(devices [i], CL_DEVICE_NAME, 1024, ret, NULL);
      printf("CL_DEVICE_NAME: %s\n", ret);
      clGetDeviceInfo(devices [i], CL_DEVICE_VENDOR, 1024, ret, NULL);
      printf("CL_DEVICE_VENDOR: %s\n", ret);
    }
    free(devices);
  }
}
```

Listing 2.12: Listing devices in an OpenCL context

2.6. OpenCL Context to Manage Devices – C++

The examples in this section are analogous to the ones in section 2.5. The class that represents an OpenCL context in C++ is `cl::Context`.

The first context creation example can be seen in listing 2.13. The listing contains a function **`createContextFromType`** that takes two parameters. The first one is a device type that can be any of the device types supported by OpenCL:

- `CL_DEVICE_TYPE_GPU`
- `CL_DEVICE_TYPE_CPU`
- `CL_DEVICE_TYPE_ACCELERATOR`
- `CL_DEVICE_TYPE_ALL`
- `CL_DEVICE_TYPE_DEFAULT`
- `CL_DEVICE_TYPE_CUSTOM`

The second parameter is the index of the OpenCL platform.

The example function creates a context containing devices chosen by type from the selected platform. The method **`cl::Platform::get`** inserts the list of available platforms into the vector of `cl::Platform` values – in the example, this vector is called *`platforms`*. Then, using the constructor **`cl::Context`**, the context is created. The array *`cps`* holds the same type of values as in the example in listing 2.10 – the zero terminated list of `cl_context_properties` values. The `CL_CONTEXT_PLATFORM` means that the next value is the selected platform.

```
cl::Context createContextFromType(cl_device_type devtype, int i) {
  std::vector < cl::Platform > platforms;
  cl::Platform::get(&platforms);
  cl_context_properties cps [3] =
  { CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms [i])(), 0 };
  return cl::Context(devtype, cps);
}
```

Listing 2.13: OpenCL Context initialization using **clCreateContextFromType**

```
cl::Context createContextFromIndex(int pidx, int didx) {
  std::vector < cl::Device > devices;
  std::vector < cl::Device > device;
  std::vector < cl::Platform > platforms;
  cl::Platform::get(&platforms);
  platforms [pidx].getDevices(CL_DEVICE_TYPE_ALL, &devices);
  cl_context_properties cps [3] =
  { CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms [pidx])(), 0 };
  device.push_back(devices [didx]);
  return cl::Context(device, cps, NULL, NULL);
}
```

Listing 2.14: OpenCL context initialization using **clCreateContext**

The second example of context creation can be seen in listing 2.14. It shows the function **createContextFromIndex** that creates a context on the platform with the index *pidx* on the platforms list. The context will contain only one device selected by the device index *didx* on the devices list from the given platform. The method **cl::Platform::getDevices** fills the vector of `cl::Device` values. It is the list of all devices present on the regarded platform. In the next step, the selected device is added to the vector *device*, so this vector will contain only one device. Then the context is created, using the constructor **cl::Context**. Remember that an OpenCL context can contain more than one device. If the vector *device* from this example contained more devices, the context would be created with them.

cl::Context::Context

```
cl::Context::Context ( VECTOR_CLASS<Device>& devices, cl_context_properties
    *properties = NULL, void (CL_CALLBACK * pfn_notify) (const char *errorinfo,
    const void *private_info_size, ::size_t cb, void *user_data) = NULL,
    void *user_data = NULL, cl_int *err = NULL );
```

Creates an OpenCL context.

Consider the example function displaying the list of devices present in the context from listing 2.15. Getting the list of devices present in an OpenCL context can be performed using the template method **cl::Context::getInfo** with the type name `CL_CONTEXT_DEVICES`. The returned vector of `cl::Device` elements is the same list as the one passed to the constructor **cl::Context**. The information about each de-

```
void listDevicesInContext(cl::Context & context){
  std::vector < cl::Device > devices = context.getInfo<CL_CONTEXT_DEVICES > ();
  if (devices.size() > 0) {
    for (size_t j = 0; j < devices.size(); j++) {
      std::cout << "Device " << j << std::endl;
      std::cout << "CL_DEVICE_NAME: " <<
      devices [j].getInfo <CL_DEVICE_NAME> () << std::endl;
      std::cout << "CL_DEVICE_VENDOR: " <<
      devices [j].getInfo < CL_DEVICE_VENDOR > () << std::endl;
    }
  }
}
```

Listing 2.15: Listing devices in OpenCL Context

vice is displayed in the **for** loop. Basic information about each device is obtained using the template method **cl::Device::getInfo** with the appropriate property name. Note that the **cl::Context::getInfo** method can also be used to get any other information about an OpenCL context.

cl::Context::getInfo

```
template <typename T>cl_int cl::Context::getInfo ( cl_context_info name,
    T *param );
```

Gets specific information about the OpenCL context.

2.7. Error Handling

Every well-written application must deal with errors. OpenCL provides ways of receiving the exit status of functions. OpenCL C++ API provides more sophisticated ways of error handling, but both C and C++ provide sufficient information for the application.

2.7.1. Checking Error Codes

The most traditional and straightforward way of handling error is by checking the value that is returned by the function being checked. This method is present in both C and C++ OpenCL API. Almost every function or method present in OpenCL can return an error code. The code in listings 2.16 and 2.17 presents this method of error handling. This frequently used fragment of code creates the OpenCL program object. The error code is stored into the value *err*. `CL_SUCCESS` indicates that everything went well. The other error codes are also visible in the example – `CL_INVALID_CONTEXT`, `CL_OUT_OF_RESOURCES` and `CL_OUT_OF_HOST_MEMORY`. Error codes have self-explanatory names. For example `CL_OUT_OF_HOST_MEMORY` means that the host does not have enough free memory to perform the requested operation. Error codes are gathered in the header file `cl.h`. In the C++ API, error

```
program = clCreateProgramWithSource(context, 1, ( const char ** )&appsource, &size,
  &err);
if (err != CL_SUCCESS) {
  printf("ERROR: ");
  switch (err) {
  case CL_INVALID_CONTEXT:
    printf("Context is not a valid context.\n");
    break;
  case CL_OUT_OF_RESOURCES:
    printf(
      "There is a failure to allocate resources required by the OpenCL
           implementation on the device.\n");
    break;
  case CL_OUT_OF_HOST_MEMORY:
    printf(
      "There is a failure to allocate resources required by the OpenCL
           implementation on the host.\n");
  }
  return NULL;
}
```

Listing 2.16: Error handling using error codes – the C version

handling in this way is the default, but it is possible to turn this off and use exceptions.

```
program = cl::Program(context, sources, &err);
if (err != CL_SUCCESS) {
  std::cout << "ERROR: ";
  switch (err) {
  case CL_INVALID_CONTEXT:
    std::cout << "Context is not a valid context." << std::endl;
    break;
  case CL_OUT_OF_RESOURCES:
    std::cout <<
    "There is a failure to allocate resources required by the OpenCL implementation
         on the device."
            << std::endl;
    break;
  case CL_OUT_OF_HOST_MEMORY:
    std::cout <<
    "There is a failure to allocate resources required by the OpenCL implementation
         on the host."
            << std::endl;
  }
  exit(EXIT_FAILURE);
}
```

Listing 2.17: Error handling using error codes – the C++ version

```
#define __CL_ENABLE_EXCEPTIONS

#include <cstdlib>
#include <iostream>

#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif

int main(int argc, char **argv) {
  cl::Context context;
  std::vector < cl::Platform > platforms;
  cl::Platform::get(&platforms);
  cl_context_properties cps [3] =
  { CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms [0])(), 0 };
  try {
    context = cl::Context(CL_DEVICE_TYPE_ACCELERATOR, cps);
  } catch (cl::Error &e) {
    std::cout << "Error in function " << e.what() << ": " << e.err() << std::endl;
  }
  return 0;
}
```

Listing 2.18: The full host program that uses exceptions available in C++ OpenCL API

2.7.2. Using Exceptions – Available in C++

The OpenCL C++ wrapper API provides a way of error handling by using exceptions. This method is used in most of the C++ examples in this book. It is a very convenient way of error handling. It allows for skipping implicit error checking and displays information about errors at the top-level function. Implicit error handling can even be completely omitted allowing the default exception handler to display errors. In order to use this feature, the program must define `__CL_ENABLE_EXCEPTIONS`. This preprocessor macro must be defined before inclusion of `cl.hpp` in order to work. An example of an application that catches the exception generated during context creation can be seen in listing 2.18.

On most configurations this code will generate an error, because it tries to create context with an accelerator device. Most consumer computers do not have such a device, so the error will be equal to (-1), that means `CL_DEVICE_NOT_FOUND`.

Note that `__CL_ENABLE_EXCEPTIONS` is defined in the very beginning of the application code. It is important to remember this, because it is a common mistake to define this macro after inclusion of the OpenCL header files. In that case, the `cl::Error` will be undefined and it will not turn on exceptions.

```
#define __CL_ENABLE_EXCEPTIONS
#define __CL_USER_OVERRIDE_ERROR_STRINGS

#define __GET_DEVICE_INFO_ERR "Get device info failed"
#define __CREATE_CONTEXT_FROM_TYPE_ERR "create context from type failed"
// ...
```

Listing 2.19: The macros that turn on the custom error strings

```
// ...
#define __CREATE_SUB_DEVICES "Create subdevices failed"
#define __CREATE_CONTEXT_ERR "Create context failed"

#include <cstdlib>
#include <iostream>
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif

int main(int argc, char **argv) {
  cl::Context context;
  std::vector < cl::Platform > platforms;
  cl::Platform::get(&platforms);
  cl_context_properties cps [3] =
  { CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms [0])(), 0 };
  try {
    context = cl::Context(CL_DEVICE_TYPE_ACCELERATOR, cps);
  } catch (cl::Error &e) {
    std::cout << "Error: " << e.what() << ": " << e.err() << std::endl;
  }
  return 0;
}
```

Listing 2.20: The main function for the example of custom error strings usage

2.7.3. Using Custom Error Messages

The default behavior of OpenCL C++ API is to store the function name that caused an error in the exception object of `cl::Error` class. The API allows for altering this behavior. This can be done by defining `__CL_USER_OVERRIDE_ERROR_STRINGS` and string constants for all of the possible functions. It is not a commonly used feature, but it allows more descriptive exceptions. In listings 2.19 and 2.20, there are examples of this approach. The code is split into two parts because of the number of needed macros.

All of the example source codes are available on the disk attached to the book.

```
cl_command_queue queue;
cl_device_id devices [1];
clGetContextInfo(context, CL_CONTEXT_DEVICES, sizeof(cl_device_id), devices, NULL);
queue = clCreateCommandQueue(context, devices [0], 0, NULL);
```

Listing 2.21: In-order command queue initialization – C programming language

2.8. Command Queues

OpenCL executes commands via a command queue. It is possible to attach multiple command queues to one device, but not one command queue to multiple devices. The standard defines:

Command

OpenCL operation that is submitted to a command-queue for execution. For example, OpenCL commands issue kernels for execution on a compute device, manipulate memory objects, etc. [10]

Command-Queue

An object that holds commands that will be executed on a specific device. The command-queue is created on a specific device in a context. Commands to a command queue are queued in order but may be executed in order or out of order. [10]

2.8.1. In-order Command Queue

The command queue gives a very convenient abstraction for code execution. The in–order command queues are best for executing code that must sequentially execute kernels. This kind of queue is also less complicated in implementation and understanding. Most examples in this part use this kind of queue.

Algorithm 1 Calculation example

$\lambda \leftarrow \frac{\alpha}{p^T A p}$
$x \leftarrow x + \lambda p$
$r \leftarrow r - \lambda A p$
$p \leftarrow r + x$
$\alpha \leftarrow ||r||^2$

Consider the algorithm 1. This algorithm operates on vectors (small letters), matrices (capital letters) and scalars (Greek letters). Each algorithm step can be implemented as a kernel. As stated before, kernels execute work–items in parallel.

The schematic of in–order execution for this algorithm is depicted in figure 2.8. The commands on the queue are executed from first to last. The execution goes along the arrows in the queue. The last command put into the queue is executed last. These commands are executed in the exact same order as in the algorithm 1.

An example of command queue initialization can be seen in listing 2.21 for C and in listing 2.22 for C++. For the C programming language, the function

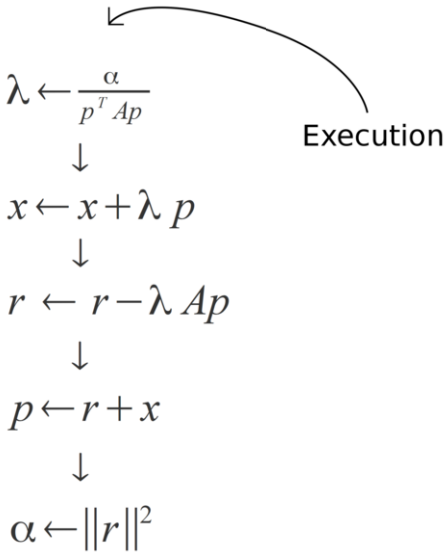

Figure 2.8: In-order queue execution

```
1 cl::CommandQueue queue;
2 queue = cl::CommandQueue(context, context.getInfo<CL_CONTEXT_DEVICES > () [0]);
```

Listing 2.22: In-order command queue initialization – C++ programming language

clCreateCommandQueue is used. The C++ code is actually even less complicated than the C version because it does not even need the temporary variable for storing the device identifier. The constructor **cl::CommandQueue** is used in this language for queue creation. Both codes get the first device from the context and create the command queue on it. The device selection is performed via **getInfo** call. This is a convenient way for obtaining list of devices. The host code does not need to remember the list of devices in the context.

clCreateCommandQueue

```
cl_command_queue clCreateCommandQueue ( cl_context context, cl_device_id
    device, cl_command_queue_properties properties, cl_int *errcode_ret );
```

Create a command-queue on a specific device.

cl::CommandQueue::CommandQueue

```
cl::CommandQueue::CommandQueue ( const Context& context, const Device& device,
    cl_command_queue_properties properties = 0, cl_int *err = NULL );
```

Creates command queue in given context on specified devices.

```
clEnqueueTask(queue, kernels [0], 0, NULL, NULL);
clEnqueueTask(queue, kernels [1], 0, NULL, NULL);
// ...
```

Listing 2.23: In-order command enqueue – C programming language

```
queue.enqueueTask(kernels [0]);
queue.enqueueTask(kernels [1]);
// ...
```

Listing 2.24: In-order command enqueue – C+ programming language

The simple example that enqueues commands in the same way as in figure 2.8 can be seen in listing 2.23 for C and in listing 2.24 for C++. The indexing is from zero in order to comply with C programming convention.

In the C example, the command **clEnqueueTask** is used to enqueue the task. The task is a kernel that is executed in only one instance.

clEnqueueTask

```
cl_int clEnqueueTask ( cl_command_queue command_queue, cl_kernel kernel,
    cl_uint num_events_in_wait_list, const cl_event *event_wait_list,
    cl_event *event );
```

Enqueues a command to execute a kernel on a device.

The **cl::CommandQueue::enqueueTask** method is used to enqueue task in C++ OpenCL wrapper.

cl::CommandQueue::enqueueTask

```
cl_int cl::CommandQueue::enqueueTask ( const Kernel& kernel,
    const VECTOR_CLASS<Event> *events = NULL, Event *event = NULL );
```

Enqueues OpenCL task on command queue.

2.8.2. Out-of-order Command Queue

On the other hand, it is possible to parallelize execution of commands. OpenCL allows for a higher level of abstraction by allowing out-of-order command execution on a command queue. The programmer can define the execution order in such a queue, using events and event wait lists.

Algorithm 1 is an example of an algorithm that can be parallelized on a higher level. Note that steps 2 and 3 are independent of each other but use value generated in step 1, as with steps 4 and 5. Analyzing this algorithm leads to the flow graph.

The event object identifies a particular kernel execution instance. It is created in the moment of putting a command into the command queue. Event objects are

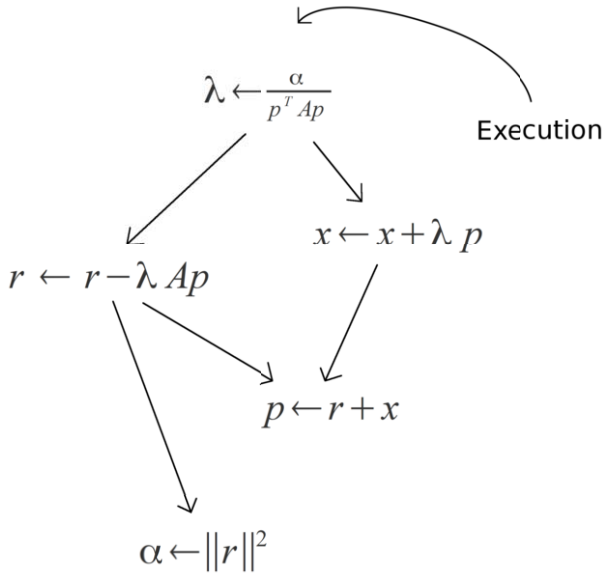

Figure 2.9: Out-of-order queue execution

unique and can be used to identify a particular kernel execution instance. The event wait list contains a list of event objects. A command can execute only if the commands identified by events in this list are finished.

In figure 2.9, there is an example of this queue execution model. This is the same algorithm, but with events defined. The command number 1 does not have any dependencies (empty event wait list), so it can execute immediately. Commands 2 and 3 can be executed concurrently just after command 1. Command 4 must wait for both commands 2 and 3. Command 5 can be executed immediately after command 2.

Note that commands 3 and 4 can be executed concurrently, but it is also possible that OpenCL implementation will run them sequentially. In this example, the commands could be put into queue in any order. The implementation can execute some commands (like 2 and 3) in parallel. Remember that commands are usually highly parallel algorithms themselves. This queue construction allows for a task and data parallel computing mixture.

The example for creating an out-of-order command queue can be seen in listing 2.25 for C and in listing 2.26 for C++. The out-of-order type of command queue is driven by the bitfield value `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`.

Simple command queue usage following the example algorithm in figure 2.9 can be seen in listing 2.27 for C and in listing 2.28 for C++. In this code, the indexing is from zero instead of one in order to follow C and C++ convention.

The function **`clEnqueueTask`** enqueues a simple task – the kernel that is executed in only one instance. The last three parameters are for setting up events. The third parameter determines the size of the list of events to wait for. The fourth parameter points to the list of events that have to finish before this kernel can execute.

```
cl_command_queue queue;
cl_device_id devices [1];
clGetContextInfo(context, CL_CONTEXT_DEVICES, sizeof(cl_device_id), devices, NULL);
queue = clCreateCommandQueue(context, devices [0],
  CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, NULL);
```

Listing 2.25: Out-of-order command queue initialization – C programming language

```
cl::CommandQueue queue;
queue = cl::CommandQueue(context, context.getInfo<CL_CONTEXT_DEVICES > () [0],
  CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);
```

Listing 2.26: Out-of-order command queue initialization – C++ programming language

```
cl_event event [5];
cl_event eventsWait [5] [2];

clEnqueueTask(queue, kernel [0], 0, NULL, &event [0]);

eventsWait [0] [0] = event [0];
clEnqueueTask(queue, kernel [1], 1, eventsWait [0], &event [1]);
clEnqueueTask(queue, kernel [2], 1, eventsWait [0], &event [2]);

eventsWait [1] [0] = event [1];
eventsWait [1] [1] = event [2];
clEnqueueTask(queue, kernel [3], 2, eventsWait [1], &event [3]);

eventsWait [2] [0] = event [2];
clEnqueueTask(queue, kernel [4], 1, eventsWait [2], &event [4]);
```

Listing 2.27: Out-of-order command queue usage – C programming language

The last parameter is a pointer to the event object that will store any event associated with this kernel execution (command).

In the C++ version, the queue object provides methods to enqueue commands on it. The method **cl::CommandQueue::enqueueTask** enqueues a task. The second parameter contains the list of events that have to finish before the kernel being enqueued can start. The last parameter points to an event that will be set to the event object associated with this kernel command.

There is no guarantee for the execution order in an out-of-order command queue, except for explicitly set using events. Note that OpenCL implementation can also execute these commands sequentially.

Detailed examples for executing kernels are in section 2.11.

```
cl::Event event [5];
std::vector<cl::Event> eventsWait [5];

queue.enqueueTask(kernel [0], NULL, &event [0]);

eventsWait [0].push_back(event [0]);
queue.enqueueTask(kernel [1], &eventsWait [0], &event [1]);
queue.enqueueTask(kernel [2], &eventsWait [0], &event [2]);

eventsWait [1].push_back(event [1]);
eventsWait [1].push_back(event [2]);
queue.enqueueTask(kernel [3], &eventsWait [0], &event [3]);

eventsWait [2].push_back(event [2]);
queue.enqueueTask(kernel [4], &eventsWait [0], &event [4]);
```

Listing 2.28: Out-of-order command queue usage – C++ programming language

2.8.3. Command Queue Control

The commands enqueued into the command queue do not need to be executed instantly. OpenCL provides the function to order command queue to start execution. The function allowing this is **clFlush** for C and **cl::CommandQueue::flush** for C++. The flush operation issues all previously queued OpenCL commands in a command queue to the device associated with it. The commands on the command queue do not need to finish before the flush call returns.

clFlush

```
cl_int clFlush ( cl_command_queue command_queue );
```

Issues all previously queued OpenCL commands in a command-queue to the device associated with the command-queue.

cl::CommandQueue::flush

```
cl_int cl::CommandQueue::flush ( void );
```

Issues all previously queued OpenCL commands in a command-queue to the device associated with the command-queue.

It is sometimes a good practice to issue a flush after a block of commands. This forces OpenCL implementation to start execution "now". Not using a flush is also possible, but in some implementations it can provide inferior performance.

The functions **clFinish** for C and **cl::CommandQueue::finish** for C++ allow waiting for all commands on the command queue to finish. This can be used as a synchronization point.

Note that using the **clFinish** or **cl::CommandQueue::finish** command can be harmful to performance. The advised way of synchronization is by using events

clFinish

```
cl_int clFinish ( cl_command_queue command_queue );
```

Blocks until all previously queued OpenCL commands in a command-queue are issued to the associated device and have completed.

cl::CommandQueue::finish

```
cl_int cl::CommandQueue::finish ( void );
```

Blocks until all previously queued OpenCL commands in a command-queue are issued to the associated device and have completed.

or the blocking version of buffer operations. A good place to use the finish command is during application shutdown, when resources are released.

2.8.4. Profiling Basics

OpenCL allows for measurement of times for command execution stages. There are four stages of command execution:

1. Putting a command into a command queue.
2. Submission of a command to the device for execution. This stage is managed by the command queue.
3. Execution start.
4. Execution finish.

OpenCL stores the start times of each stage. The times are stored in nanoseconds. Time is expressed in device time. Each stage of command execution has an associated constant with it of type `cl_profiling_info`. The constants are:

1. `CL_PROFILING_COMMAND_QUEUED` – the moment of putting the command into the command queue.
2. `CL_PROFILING_COMMAND_SUBMIT` – the time of submission of the command to be executed in the device.
3. `CL_PROFILING_COMMAND_START` – the execution start.
4. `CL_PROFILING_COMMAND_END` – the execution finish time.

Profiling is a very powerful tool for finding hot spots in the OpenCL program. This allows for better application optimization without any additional tools. There are of course external profiling tools, but profiling in command queues is given "for free" with the OpenCL standard.

2.8.5. Profiling Using Events – C example

An example of command queue initialization that allows for profiling can be seen in listing 2.29. This is also an out-of-order command queue.

An example of one command profiling is shown in listing 2.30. The command in the example is also a task. This could be of course any available command, like ordinary kernel execution or memory transfer.

```
cl_command_queue queue;
cl_device_id devices [1];
clGetContextInfo(context, CL_CONTEXT_DEVICES, sizeof(cl_device_id), devices, NULL);
queue = clCreateCommandQueue(context, devices [0],
  CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE | CL_QUEUE_PROFILING_ENABLE, NULL);
```

Listing 2.29: The C code of command queue initialization with profiling enabled.

```
cl_event event;
cl_ulong queued;
cl_ulong submit;
cl_ulong start;
cl_ulong end;

clEnqueueTask(queue, kernel, 0, NULL, &event);
clFlush(queue);
clWaitForEvents(1, &event);
// get the moment in time of enqueueing command
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_QUEUED,
  sizeof(cl_ulong), &queued, NULL);
// get the submission time - from queue to execution
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_SUBMIT,
  sizeof(cl_ulong), &submit, NULL);
// get the start time of execution
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
  sizeof(cl_ulong), &start, NULL);
// get the time of execution finish
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
  sizeof(cl_ulong), &end, NULL);
printf("%u dT=0\n", (unsigned)queued);
printf("%u dT=%d\n", (unsigned)submit, (int)(submit - queued));
printf("%u dT=%d\n", (unsigned)start, (int)(start - submit));
printf("%u dT=%d\n", (unsigned)end, (int)(end - start));
```

Listing 2.30: The profiling example – C code.

clGetEventProfilingInfo

```
cl_int clGetEventProfilingInfo ( cl_event event, cl_profiling_info param_name,
    size_t param_value_size, void *param_value, size_t *param_value_size_ret );
```

Returns profiling information for the command associated with event if profiling is enabled.

In order to enable profiling, the command queue must be created with the flag `CL_QUEUE_PROFILING_ENABLE` set.

The event object associated with a command is necessary for profiling. The profiling information can be retrieved using **`clGetEventProfilingInfo`**. This function gets the `cl_profiling_info` for selecting which time to retrieve. The result is stored in a value pointed by the fourth parameter. For the times that are listed in this section, the value is always of type `cl_ulong`. The computation of time between differ-

clWaitForEvents

```
cl_int clWaitForEvents ( cl_uint num_events, const cl_event *event_list );
```

Waits on the host thread for commands identified by event objects to complete.

```
1 cl::CommandQueue queue;
2 queue = cl::CommandQueue(context, context.getInfo<CL_CONTEXT_DEVICES > () [0],
3   CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE | CL_QUEUE_PROFILING_ENABLE);
```

Listing 2.31: The C++ code of command queue initialization with profiling enabled.

ent stages is straightforward. The example fragment prints the profiling information into the console.

2.8.6. Profiling Using Events – C++ example

Profiling in C++ is done in a very similar way as for C. First, the command queue has to be created with profiling enabled. An example of creating an appropriate command queue can be seen in listing 2.31.

The method **cl::Event::getProfilingInfo** provides a way of obtaining profiling information. This method takes the same constant values as the C function **clGetEventProfilingInfo**. The last parameter provides a pointer to the `cl_ulong` value for storing time. The example code that uses profiling features of OpenCL in C++ can be seen in listing 2.32. This example prints profiling information into the screen along with calculated times of execution stages.

cl::Event::getProfilingInfo

```
template <cl_int name> typename detail::param_traits
   <detail::cl_profiling_info, name>::param_type cl::Event::getProfilingInfo (
   void );
```

Returns profiling information for the command associated with event.

cl::Event::wait

```
cl_int cl::Event::wait ( void );
```

Waits on the host thread for the command associated with the particular event to complete.

```
cl::Event event;
cl_ulong queued;
cl_ulong submit;
cl_ulong start;
cl_ulong end;
queue.enqueueTask(kernel, NULL, &event);
queue.flush();
event.wait();
// get the moment in time of enqueueing command
event.getProfilingInfo<cl_ulong>(CL_PROFILING_COMMAND_QUEUED, &queued);
// get the submission time - from queue to execution
event.getProfilingInfo<cl_ulong>(CL_PROFILING_COMMAND_SUBMIT, &submit);
// get the start time of execution
event.getProfilingInfo<cl_ulong>(CL_PROFILING_COMMAND_START, &start);
// get the time of execution finish
event.getProfilingInfo<cl_ulong>(CL_PROFILING_COMMAND_END, &end);
std::cout << queued << " dT=0" << std::endl;
std::cout << submit << " dT=" << submit - queued << std::endl;
std::cout << start << " dT=" << start - submit << std::endl;
std::cout << end << " dT=" << end - start << std::endl;
```

Listing 2.32: Profiling example – C++ code.

2.9. Work-Items and Work-Groups

OpenCL uses non-traditional terminology for parallel computation. The logical elements performing computation are called **work-items** instead of threads. The work-item is a single processing unit that executes one kernel instance.

Kernel is the function that is parallelized and executed in work-items. The same function executes on different work-items, and the only difference between instances is its ID. This ID allows for different execution paths for each work-item.

Global ID

A global ID is used to identify uniquely a work-item and is derived from the number of global work-items specified when executing a kernel. The global ID is an N-dimensional value that starts at $(0, 0, \ldots 0)$.

Local ID

A local ID specifies a unique work-item ID within a given work-group that is executing a kernel. The local ID is an N-dimensional value that starts at $(0, 0, \ldots 0)$.

OpenCL defines an index space for kernel execution. The coordinate in an index space allows kernel instances to perform different computations. The ID can be compared to rank in MPI. OpenCL defines its index space as NDRange:

NDRange

The index space supported in OpenCL. An NDRange is an N-dimensional index space, where N is one, two or three. An NDRange is defined by an integer array of length N specifying the extent of the index space in each dimension starting at an offset index F (zero by default). Each work-item's global ID and local ID are N-dimensional tuples. The global ID components are values in the range from F, to F plus the number of elements in that dimension minus one.

This method of indexing (local and global ID) has its origins in GPU construction. The GPU, among other elements, contains multiple streaming multiprocessors. Each of them computes multiple threads in parallel. For example, some modern GPUs (year 2011) can contain 16 streaming multiprocessors, each with 32 cores [14]. OpenCL 1.2 allows for `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS` dimensions. The value of this property can be acquired by using **`clGetDeviceInfo`**. This value must be greater or equal to 3.

Each instance of an executing kernel can obtain information about its coordinates in an NDRange. The functions **`get_global_id`** and **`get_local_id`** allow this. This information is then used to perform computation on different data or in different ways. A more detailed description of OpenCL C is described in section 2.11.

The NDRange allows for a data parallel programming model. OpenCL modifies the classical definition:

Data Parallel Programming Model

Traditionally, this term refers to a programming model where concurrency is expressed as instructions from a single program applied to multiple elements within a set of data structures. The term has been generalized in OpenCL to refer to a model wherein a set of instructions from a single program is applied concurrently to each point within an abstract domain of indices.

2.9.1. Information About Index Space from a Kernel

The OpenCL programming language provides functions that allow kernel to identify itself within an index space. **get_global_id** and **get_local_id** will be described in more detail in more detail in the examples. For reference, here are the functions available for a kernel that refers to an NDrange:

get_global_id

```
size_t get_global_id ( uint dimindx );
```

Returns the unique global work-item ID value for a dimension identified by *dimindx*.

get_global_size

```
size_t get_global_size ( uint dimindx );
```

Returns the number of global work-items specified for dimension identified by dimindx.

get_global_offset

```
size_t get_global_offset ( uint dimindx );
```

Returns the offset values specified by clEnqueueNDRangeKernel.

get_group_id

```
size_t get_group_id ( uint dimindx );
```

Returns the work-group ID for given dimension.

get_local_id

```
size_t get_local_id ( uint dimindx );
```

Returns the unique local work-item ID.

get_local_size

```
size_t get_local_size ( uint dimindx );
```

Returns the number of local work-items.

get_num_groups

```
size_t get_num_groups ( uint dimindx );
```

Returns the number of work-groups that will execute a kernel.

For more details about each function, please refer to OpenCL API documentation [10] and section 2.11 of this book.

get_work_dim

```
uint get_work_dim ( );
```

Returns the number of dimensions in use.

```
1 size_t global_work_size[] = { 4 };
2 clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, NULL, 0, NULL,
3   NULL);
```

Listing 2.33: Enqueue of a kernel in the NDRange of size 4. The work group size is selected automatically. – C programming language

2.9.2. NDRange Kernel Execution

The most commonly used function that allows for kernel execution is **clEnqueueNDRangeKernel** (C programming language) or **cl::CommandQueue::enqueueNDRangeKernel** (for C++). These functions enqueue a given kernel so it will be eventually executed on the device. There is no way to execute a kernel directly (without using command queue).

clEnqueueNDRangeKernel

```
cl_int clEnqueueNDRangeKernel ( cl_command_queue command_queue,
    cl_kernel kernel, cl_uint work_dim, const size_t *global_work_offset,
    const size_t *global_work_size, const size_t *local_work_size,
    cl_uint num_events_in_wait_list, const cl_event *event_wait_list,
    cl_event *event );
```

Enqueues a command to execute a kernel on a device.

cl::CommandQueue::enqueueNDRangeKernel

```
cl_int cl::CommandQueue::enqueueNDRangeKernel ( const Kernel& kernel,
    const NDRange& offset, const NDRange& global, const NDRange& local,
    const VECTOR_CLASS<Event> * events = NULL, Event * event = NULL );
```

This method enqueues a command to execute a kernel on a device.

Running a kernel in a 1D index space

The example of running a kernel in the 1D NDRange of size 4 can be seen in listings 2.33 and 2.34. Both examples use NULL values as *global_work_offset* and *local_work_size* - this means that there will be no offset, and work-groups will be created using a default size.

Consider the code that can be seen in listing 2.33. When using the C programming language, one has to define appropriate arrays storing the sizes for the NDRange. Only *global_work_size* must be set. The other values can be NULL; if so, the implementation would choose the best values for local work size.

In C++ OpenCL implementation, the class cl::NDRange is defined. This is a wrapper class for storing information about an NDRange. The constructor can take 1,

```
queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(4), cl::NullRange);
```

Listing 2.34: Enqueue of a kernel in the NDRange of size 4. The work group size is selected automatically. – C++ programming language

```
kernel void sampleKernel() {
  size_t gid[] = { get_global_id(0), get_global_id(1) };
  size_t lid[] = { get_local_id(0), get_local_id(1) };
  // ... some computation goes here ...
}
```

Listing 2.35: Example kernel.

2 or 3 parameters depending on the number of dimensions. There is also a predefined constant `cl::NullRange`, which is used to mark unspecified values.

Assuming that the system selects 2 as the size of a work-group, the work-items in the NDRange would have the IDs and sizes as shown in figure 2.10. There will be two work-groups with two elements each. All of these values are available from running a kernel. Each kernel instance will execute in one work-item and will have access to information about its coordinates.

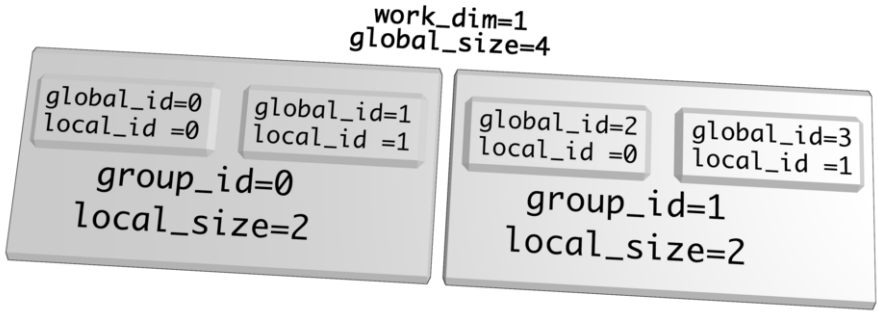

Figure 2.10: One-dimensional NDRange of size 4 with work-group of size 2.

Consider the example kernel code in listing 2.35. This code part demonstrates how to get local and global coordinates. The function **`get_global_id`** returns the global ID on the given dimension. The **`get_local_id`** returns the local ID on the given dimension. The dimensions checked here are 0 and 1, but when executed by the code that can be seen in 2.33 or 2.34, only the axis of index 0 will contain coordinates different than 0. This is because for dimensions higher than the declared dimensions, these values are always 0.

```
size_t global_work_offset[] = { 0, 0 };
size_t global_work_size[] = { 4, 4 };
size_t local_work_size[] = { 2, 2 };
clEnqueueNDRangeKernel(queue, kernel, 2, global_work_offset, global_work_size,
  local_work_size, 0, NULL,
  NULL);
```

Listing 2.36: Enqueue of kernel in the NDRange of size 4×4 with work-groups of size 2×2, so there are 4 work groups – C programming language

```
queue.enqueueNDRangeKernel(kernel, cl::NullRange,
  cl::NDRange(4, 4), cl::NDRange(2, 2));
```

Listing 2.37: Enqueue of kernel in the NDRange of size 4×4 with work-groups of size 2×2, so there are 4 work groups – C++ programming language

Running kernel in 2D index space

Kernel execution in two-dimensional NDRange for C and C++ can be seen in listings 2.36 and 2.37, respectively. Also defined is *local_work_size*, which means that the kernel must be executed using work-groups of size 2×2. The graphical explanation of an NDRange created in this way is depicted in figure 2.11.

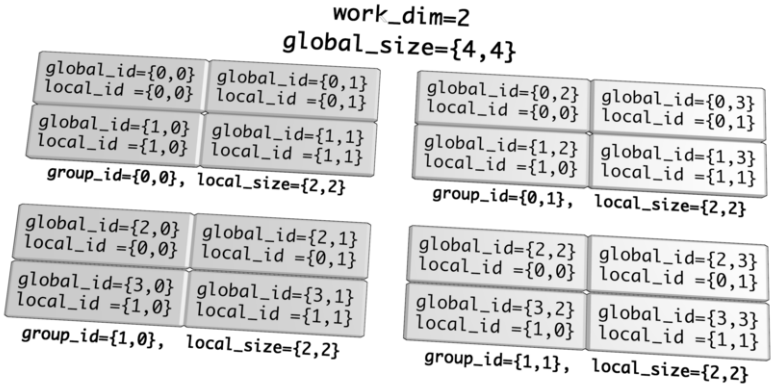

Figure 2.11: Example of two dimensional index space.

This execution model is very convenient for working with linear algebra algorithms and simulations, because coordinates in an NDRange can map to coordinates in resulting matrices or simulation cells.

```
clEnqueueTask(queue, kernel, 0, NULL, NULL);
```

Listing 2.38: Task-parallel kernel execution example – C programming language

```
queue.enqueueTask(kernel, 0);
```

Listing 2.39: Task-parallel kernel execution example – C++ programming language

2.9.3. Task Execution

OpenCL is very flexible in terms of task and data parallelism. This standard also allows for kernel execution as a single task. In such cases, the index space is limited to 1 and the calls to **get_global_id** and **get_local_id** will always return 0. This method allows for a traditional task-parallel approach, where each thread (work-item) performs different algorithm. OpenCL defines task-parallel programming model in the following way:

Task Parallel Programming Model

A programming model in which computations are expressed in terms of multiple concurrent tasks, where a task is a kernel executing in a single work-group of size one. The concurrent tasks can be running different kernels.

An example of task execution can be seen in listings 2.38 and 2.39. C implementation uses the function **clEnqueueTask**. In the example, this function enqueues a kernel object into a command queue. For C++, the command queue object provides the method **cl::Queue::enqueueTask** that takes a kernel object as a parameter.

2.9.4. Using Work Offset

Sometimes the computation is necessary only for some part of the data. The most common case is when there are not enough resources to fit the whole computation at once. Some OpenCL platform implementations allow kernels to execute in a limited time frame, so the long-running kernels are terminated by the execution environment.

Another example is the case when the application should provide the user with progress information. Most consumer applications are not supposed to freeze for more than one second. In this case, it is good to monitor the computation process by dividing the problem into smaller parts, updating the UI after each step.

The offset value is added to the index of the work-item. In this case, the function **get_global_id** returns the value shifted by the offset value.

An example kernel enqueue using this feature is in listings 2.40 and 2.41 for C and C++, respectively. The resulting NDRange is depicted in figure 2.12.

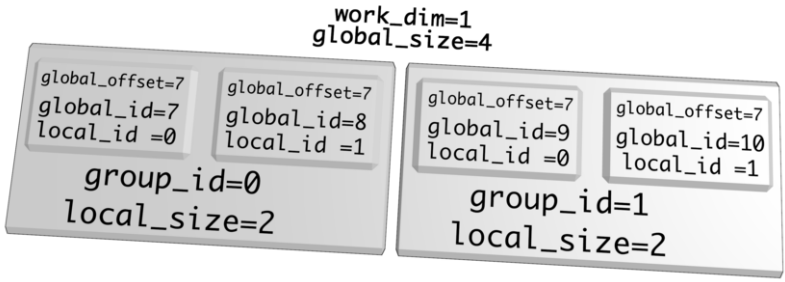

Figure 2.12: Example of NDRange for kernel execution with offset.

```
size_t global_work_offset[] = { 7 };
size_t global_work_size[] = { 4 };
size_t local_work_size[] = { 2 };
clEnqueueNDRangeKernel(queue, kernel, 1, global_work_offset, global_work_size,
  local_work_size, 0, NULL,
  NULL);
```

Listing 2.40: Enqueue kernel in an NDRange with offset – the C code.

```
queue.enqueueNDRangeKernel(kernel, cl::NDRange(7), cl::NDRange(4), cl::NDRange(2));
```

Listing 2.41: Enqueue kernel in an NDRange with offset – the C++ code.

2.10. OpenCL Memory

OpenCL provides APIs that are very close to hardware. One of the skills that the programmer has to learn is memory organization viewed from the kernel perspective and from the host program. These are slightly different.

The host recognizes only the memory located in the host, defined and used by the host program and the memory buffers or textures located in contexts. It allocates and frees memory in both regions. Note that memory in context can also be located in the same physical memory as the host or in memory located on the device. The host program can also order memory transfers between buffers located in the context or in the host memory. It can also order OpenCL to allocate local memory accessible by the OpenCL program, but the host program cannot access it.

On the other hand, the OpenCL program sees four different memory pools – constant, global, local and private. Each of them has different properties and performance. Data can be explicitly copied between these memory regions.

2.10.1. Different Memory Regions – the Kernel Perspective

In OpenCL, memory is not uniform. It is divided into different regions (pools). This is because it is better related to the hardware. The memory pools are defined in the

following way:

Memory Region (or Pool)

A distinct address space in OpenCL. Memory regions may overlap in physical memory, though OpenCL will treat them as logically distinct. Memory regions are denoted as private, local, constant, and global.[10]

This distinction of different kinds of memory maps almost directly into the hardware memory organization on the GPU.

A graphics card usually has a big amount of RAM that is soldered into the board (called device memory). This is the biggest memory available to the GPU, but it is relatively slow because it must depend on a memory bus between the GPU and the graphics RAM.

The next level of memory is called shared memory or global data share memory. This memory is located inside the graphics processor. This memory is several orders of magnitude smaller than global memory, but it can communicate with the processor at a much higher speed.

The fastest memory is located directly beside the stream processors, similar to a cache in the CPU, and is the fastest memory available. This memory is called private memory.

There is also some amount of constant memory that is usually in the same region as global memory, but it can be cached and therefore also be accessed very fast.

On the other hand, these memory pools overlap when used on a CPU. Global and local memory are usually located in the system RAM. In the current implementations, private memory is also located in the system RAM, but theoretically it can be located in the CPU cache for faster access.

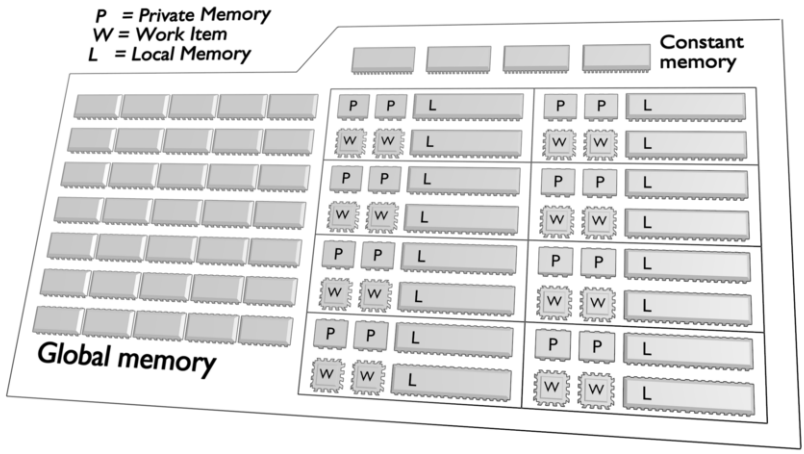

Figure 2.13: Overview of different memory pools visible from OpenCL program.

Figure 2.13 shows the overview of OpenCL memory regions with work-items and work-groups. Note that on some hardware the memory regions can overlap, but for OpenCL program there are always distinct memory pools. Private memory

is available only for the work-item. Local memory is shared between every work-item in work group, but it is not accessible from other work-groups. Global and constant memories are available globally. Each work-item can read and write to global memory and read from constant memory.

Remember that it is sometimes faster to copy some part of global memory into local memory and then perform a computation on it, than to do it directly in global memory. This model of operation is usually faster when dealing with a discrete GPU, but it can be slower for GPUs located on a motherboard or for CPUs.

Memory regions are an abstraction that can help programmers and hardware vendors to achieve the common sense of terminology. OpenCL programmers can assume that data put into local or private memory will be accessed faster than data in global memory, but global memory can store more information. Hardware manufacturers can optimize their drivers to utilize the most computational power from their hardware and leave the algorithmic issues of memory optimization to OpenCL programmers.

2.10.2. Relaxed Memory Consistency

The different memory pools provide different memory consistency models. Private memory is available only from a current kernel instance, so there is no need for synchronization and it will not be discussed. Local and global memory pools provide a relaxed consistency model. OpenCL defines it in the following way:

Relaxed Consistency

A memory consistency model in which the contents of memory visible to different work-items or commands may be different except at a barrier or other explicit synchronization point.

This is driven by the need for leverage between computation efficiency and memory sharing. The programmer's duty is to provide explicit synchronization points. To imagine this model, consider the parallel algorithm for summation of an array with four elements. The result is written into every element of an array. The array is located in local memory. The correct answer for the array:

```
1 array[] = {1, 2, 3, 4}
```

is

```
1 array[] = {10, 10, 10, 10}
```

It is easy to see that with two processors it is possible to count this sum in two steps and distribute the result also in two steps, so the whole algorithm would take $O(4)$ time. The first kernel instance would count (pseudocode):

```
Work-item 0:
// summation
array[0] = array[0] + array[1]
array[0] = array[0] + array[2]
// result distribution
array[1] = array[0]
```

and the second would count:

```
Work-item 1:
// summation
array[2] = array[2] + array[3]
// result distribution
array[2] = array[0]
array[3] = array[0]
```

After that it seems the results would be correct, but this is not the case. The result is not uniquely determined; it can be either 10 (the correct answer) or 6. Moreover, the array elements 2 and 3 can contain 1. The reason for that is the fact that work-items here do not use any kind of synchronization. In figure 2.14, there is an example of how this algorithm could execute. It is very important to remember that memory can be cached, so different work-items can see different memory values and therefore perform incorrect computations. The correct approach to this problem is

```
Work-item 0:
// summation
array[0] = array[0] + array[1]
barrier
array[0] = array[0] + array[2]
// result distribution
barrier
array[1] = array[0]
```

and

```
Work-item 1:
// summation
array[2] = array[2] + array[3]
barrier
// result distribution
barrier
array[2] = array[0]
array[3] = array[0]
```

Note that there are explicit synchronization points that assure memory consistency in those points. Correct memory synchronization using OpenCL C is shown in example 2.67.

To fix this problem, synchronization is needed. After modifying memory that could be read by other work-items, a barrier has to be made to assure consistency of memory. The execution of the fixed version of this algorithm is shown in figure 2.15. The usage of a barrier can be seen in listing 2.67.

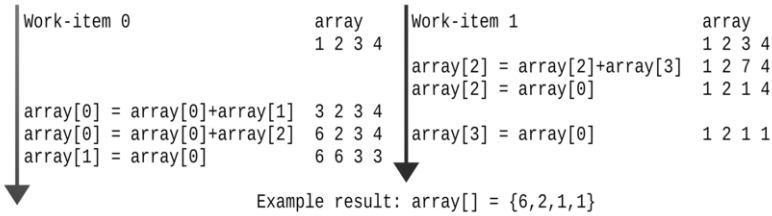

Figure 2.14: Incorrectly written parallel sum execution.

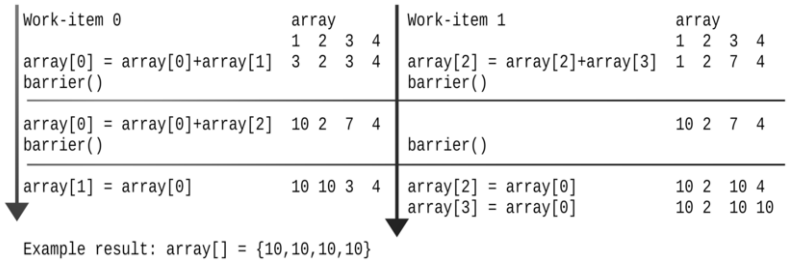

Figure 2.15: Correct parallel sum execution.

It is important to be aware that **synchronization is possible only per work-group**. It is impossible to synchronize different work-groups. In the case of global memory usage, the work-groups should write to different regions of it. One solution for the situation in which different work-groups use the same global memory region is to execute a kernel multiple times, so memory would be synchronized every time on kernel exit.

2.10.3. Global and Constant Memory Allocation – Host Code

OpenCL allows for a high level of control in terms of memory allocation and management. This allows for better utilization of hardware resources.

There are two types of buffers allocable from the host program: The memory buffer and the texture buffer. The first one is used in most cases; the other is used mostly for cooperation with OpenGL or Direct3D and will be discussed in chapter 3.4.

OpenCL uses `cl_mem` type for identifying memory buffers. The C++ wrapper uses `cl::Buffer` for this purpose. The examples of allocation of different memory buffers are in listings 2.42 and 2.43.

clCreateBuffer

```
cl_mem clCreateBuffer ( cl_context context, cl_mem_flags flags,
    size_t size, void *host_ptr, cl_int *errcode_ret);
```

Creates a buffer object.

The function **`clCreateBuffer`** creates a memory buffer. It allocates memory within a context that can be accessed by kernels. The C++ equivalent is a constructor

```
// (1)
cl_float example_array[] = { 1, 2, 3, 4 };
cl_float example_ret_arr[] = { 0, 0, 0, 0 };
// (2)
cl_mem new_buffer_dev;
cl_mem copied_buffer_dev;
cl_mem in_host_buffer_dev;
// (3)
copied_buffer_dev = clCreateBuffer(context,
  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_float) * 4, example_array,
  &ret);
// (4)
new_buffer_dev = clCreateBuffer(context, 0, sizeof(cl_float) * 4, NULL, &ret);
// (5)
in_host_buffer_dev = clCreateBuffer(context,
  CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR, sizeof(cl_float) * 4, example_array,
  &ret);
```

Listing 2.42: Memory buffers allocation – the C code.

```
// (1)
cl_float example_array[] = { 1, 2, 3, 4 };
cl_float example_ret_arr[] = { 0, 0, 0, 0 };
// (2)
cl::Buffer new_buffer_dev;
cl::Buffer copied_buffer_dev;
cl::Buffer in_host_buffer_dev;
// (3)
copied_buffer_dev = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  sizeof(cl_float) * 4, example_array);
// (4)
new_buffer_dev = cl::Buffer(context, 0, sizeof(float) * 4);
// (5)
in_host_buffer_dev = cl::Buffer(context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
  sizeof(cl_float) * 4, example_array);
```

Listing 2.43: Memory buffers allocation – the C++ code.

cl::Buffer::Buffer

```
cl::Buffer::Buffer ( const Context& context, cl_mem_flags flags,
    ::size_t size, void * host_ptr = NULL, cl_int * err = NULL );
```

The constructor that creates an OpenCL buffer object.

cl::Buffer. Both methods allocate memory accessible by kernels. These memory regions are created within context. Kernels cannot access memory that has been allocated in different contexts. The actual location of the memory region allocated in this way is driven by a bitfield called `cl_mem_flags` that is the second parameter of both functions. If none of the flags is set, then the default is selected: read-write memory allocated in the device memory. The available bitfield values are:

- CL_MEM_READ_WRITE: This is the default. It means that an allocated memory buffer is readable and writable by kernels.
- CL_MEM_WRITE_ONLY : This buffer will be writable only for a kernel. The host code can both read and write to it.
- CL_MEM_READ_ONLY: The memory buffer will be read only from kernels. It is advised to use this type of memory buffer to store constant values that can be passed as constant memory to the kernels. It gives a hint about its purpose to the OpenCL platform.
- CL_MEM_USE_HOST_PTR: This type of memory buffer will use the given host memory as storage space. OpenCL implementation can, however, cache the data in this buffer, so this buffer in the host memory can be incoherent with the data visible by kernels. To assure the coherency of this memory, the programmer can use for example **clEnqueueReadBuffer** and **clEnqueueWriteBuffer**. This memory is very useful when dealing with algorithms run by the CPU. The buffers are used directly by the kernels, so there would be no memory transfers between the host memory and the device memory.
- CL_MEM_ALLOC_HOST_PTR: This value orders OpenCL to allocate a memory buffer in the host-accessible memory. This is also very useful for cooperation with the CPU. CL_MEM_ALLOC_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.
- CL_MEM_COPY_HOST_PTR: This value is also very commonly used. It means that OpenCL must copy the data pointed by the fourth value (*ptr*) into the newly created memory buffer. This value is commonly used in examples because it helps to write more compact codes.
- CL_MEM_COPY_HOST_WRITE_ONLY: This flag specifies that the host will only write to the memory object (using OpenCL APIs that enqueue a write or a map for a write). This can be used to optimize write access from the host (e.g., enable write-combined allocations for memory objects for devices that communicate with the host over a system bus such as PCIe).
- CL_MEM_COPY_HOST_READ_ONLY: This flag specifies that the host will only read the memory object (using OpenCL APIs that enqueue a read or a map for read). CL_MEM_HOST_WRITE_ONLY and CL_MEM_HOST_READ_ONLY are mutually exclusive.
- CL_MEM_COPY_HOST_NO_ACCESS: This flag specifies that the host will not read or write the memory object. Note that this flag is mutually exclusive with CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_READ_ONLY.

Note that the distinction between constant and global memory applies only to the way a kernel sees them. From the host program, these are just memory buffers that can be accessed. On the other hand, the kernel does not have knowledge about physical location of memory buffers; it does not know if it is created, for example, with the CL_MEM_USE_HOST_PTR switch.

There is another way the host code allocates memory for a kernel – during setting parameters. Consider the kernel that takes a local memory pointer as a parameter. The host code sets that parameter for the kernel by passing the size of the buffer and NULL pointer. OpenCL implementation allocates the appropriate amount of local memory before the moment of kernel execution. This is described in detail in section 2.11.2.

2.10.4. Memory Transfers – the Host Code

OpenCL provides multiple functions for transferring data. The first method is by copying an array of the host memory during OpenCL buffer initialization. This method is shown in examples 2.42 and 2.43 on the lines marked (3). This is actually a very convenient method because it joins memory initialization and transfer into one command. This operation is blocking, and the source array after this operation can be reused.

clEnqueueReadBuffer

```
cl_int clEnqueueReadBuffer ( cl_command_queue command_queue,
    cl_mem buffer, cl_bool blocking_read, size_t offset, size_t size,
    void *ptr, cl_uint num_events_in_wait_list, const cl_event
    *event_wait_list, cl_event *event);
```

Enqueue commands to read from a buffer object to the host memory.

clEnqueueWriteBuffer

```
cl_int clEnqueueWriteBuffer ( cl_command_queue command_queue,
    cl_mem buffer, cl_bool blocking_write, size_t offset, size_t size,
    const void *ptr, cl_uint num_events_in_wait_list, const cl_event
    *event_wait_list, cl_event *event);
```

Enqueue commands to write to a buffer object from the host memory.

clEnqueueCopyBuffer

```
cl_int clEnqueueCopyBuffer ( cl_command_queue command_queue,
    cl_mem src_buffer, cl_mem dst_buffer, size_t src_offset,
    size_t dst_offset, size_t cb, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event );
```

Enqueues a command to copy from one buffer object to another.

cl::CommandQueue::enqueueReadBuffer

```
cl_int cl::CommandQueue::enqueueReadBuffer ( const Buffer& buffer,
    cl_bool blocking_read, ::size_t offset, ::size_t size, const void * ptr,
    const VECTOR_CLASS<Event> * events = NULL, Event * event = NULL );
```

This method enqueues a command to read from a buffer object to the host memory.

cl::CommandQueue::enqueueWriteBuffer

```
cl_int cl::CommandQueue::enqueueWriteBuffer ( const Buffer& buffer,
    cl_bool blocking_write, ::size_t offset, ::size_t size, const void * ptr,
    const VECTOR_CLASS<Event> * events = NULL, Event * event = NULL );
```

This method enqueues a command to write to a buffer object from the host memory.

cl::CommandQueue::enqueueCopyBuffer

```
cl_int cl::CommandQueue::enqueueCopyBuffer ( const Buffer & src,
    const Buffer & dst, ::size_t src_offset, ::size_t dst_offset,
    ::size_t size, const VECTOR_CLASS<Event> * events = NULL,
    Event * event = NULL );
```

This method enqueues a command to copy a buffer object identified by *src* to another buffer object identified by *dst*. The OpenCL context associated with command-queue, *src* and *dst* must be the same.

```
1 // (1)
2 clEnqueueWriteBuffer(queue, in_host_buffer_dev, CL_TRUE, 0,
3   sizeof(cl_float) * 4, example_array, 0, NULL, NULL);
4 // (2)
5 clEnqueueCopyBuffer(queue, in_host_buffer_dev, new_buffer_dev,
6   sizeof(cl_float), 0, sizeof(cl_float) * 3, 0, NULL, NULL);
7 // (3)
8 clEnqueueReadBuffer(queue, new_buffer_dev, CL_TRUE, 0,
9   sizeof(cl_float) * 4, example_ret_arr, 0, NULL, NULL);
```

Listing 2.44: Memory transfer. Basic operations – C code.

Another way of transferring data is by using the functions **clEnqueueWriteBuffer**, **clEnqueueReadBuffer**, **clEnqueueCopyBuffer** for the C programming language. The methods provided by the C++ wrapper are **cl::CommandQueue::enqueueReadBuffer** and **cl::CommandQueue::enqueueWriteBuffer** for copying data between the host and the device memory, and **cl::CommandQueue::enqueueCopyBuffer** for copying data between different buffers. Example usage is shown in listings 2.44 and 2.45. The operations are performed on the memory buffers allocated by the code from listings 2.42 and 2.43. The execution of this fragment is depicted in figure 2.16. Note that operation (2) copies only part of the memory buffer – three `float` values from the beginning of *in_host_buffer_dev* to the last 3 `float` values of *new_buffer_dev*.

The `CL_TRUE` parameter means that these functions will block until the data is transferred. It is possible to use them asynchronously, but for this example it is not necessary. Synchronous memory transfers can also be used as synchronization points, so the flush or finish functions can be omitted.

2.11. Programming and Calling Kernel

The OpenCL programming language is based on the C99 standard (see [15]). It contains most of the standard features of the original language, except for the elements that are specific to heterogeneous computational environments. The usage of pointer types is limited. The function address cannot be obtained, and some types are slightly different; for example, there are vector types, and the `double` type is optional. OpenCL-specific features will be described in the text when necessary.

```
// (1)
queue.enqueueWriteBuffer(in_host_buffer_dev, CL_TRUE, 0,
  sizeof(cl_float) * 4, example_array);
// (2)
queue.enqueueCopyBuffer(in_host_buffer_dev, new_buffer_dev,
  sizeof(cl_float), 0, sizeof(cl_float) * 3);
// (3)
queue.enqueueReadBuffer(new_buffer_dev, CL_TRUE, 0,
  sizeof(cl_float) * 4, example_ret_arr);
```

Listing 2.45: Memory transfer. Basic operations – C++ code.

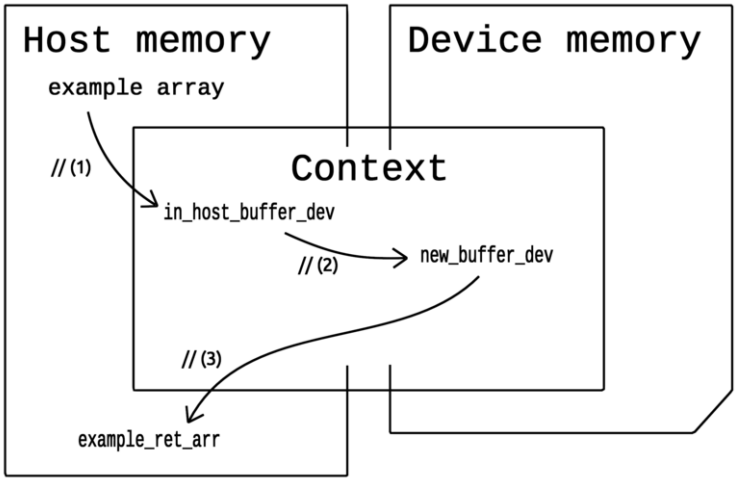

Figure 2.16: Graphical description of memory transfers executed in listings 2.44 and 2.45.

The OpenCL program consists of multiple kernel and helper functions. The kernels can be invoked from the host code. The helper functions are runnable only from kernels. The helper functions allow for better code management, and they make programming in OpenCL almost the same as in the C programming language.

The term **kernel** means the special kind of function that executes on different work-items (processing elements) in parallel. Each instance of a kernel has its own coordinates in an NDRange. It can be imagined as multiple copies of the same program executing in parallel. An overview is depicted in figure 2.17. The kernel is run in four work-items. The invocation of **get_global_id** is replaced with the actual global ID. The example function just fills the array *a[4]* with the values from 0 to 3.

Kernels in OpenCL extend the classical data parallel execution model. Traditional data parallel execution allowed execution of the same instructions on different parts of data, but it did not support branches. Every instance of a program had to follow the same execution path. In OpenCL, this is no longer the case. Each kernel instance can execute in a different way by using **if** and other conditional statements.

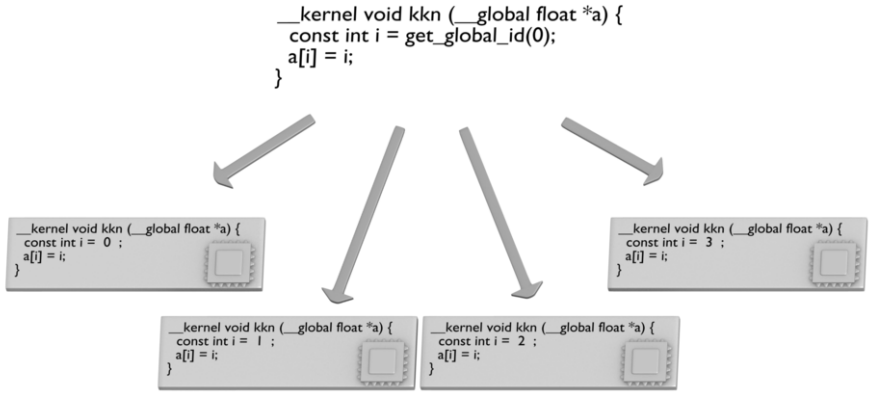

Figure 2.17: Kernel execution overview.

The kernel and kernel objects are defined in the following way:

Kernel

A kernel is a function declared in a program and executed on an OpenCL device. A kernel is identified by the **`__kernel`** or **`kernel`** qualifier applied to any function defined in a program.[10]

Kernel Object

A kernel object encapsulates a specific kernel function declared in a program and the argument values to be used when executing this kernel function.[10]

2.11.1. Loading and Compilation of an OpenCL Program

The full-profile OpenCL library must contain a compiler. This compiler is called by the host program. Before the kernels can be executed, three actions that must be taken:

1. Load the program binary or source into memory
2. Build the program
3. Create kernel objects

The application can decide for which devices the program should be built. The source format is a text string that is passed as a parameter to OpenCL API. The binary representation is implementation-dependent. Some vendors use executable program representation, some others use Assembler for a binary program. The program has to be compiled first, before any kernels can be used. The OpenCL standard defines program and program object as:

Program

An OpenCL program consists of a set of kernels. Programs may also contain auxiliary functions called by kernel functions and constant data [10].

Program Object

A program object encapsulates the following information:

- A reference to an associated context.
- A program source or binary.
- The latest successfully built program executable, the list of devices for which the program executable is built, the build options used and a build log.
- The number of kernel objects currently attached [10].

Load program into memory

An example of loading an OpenCL program from a source can be seen in listings 2.46 and 2.47. In both versions, the source is loaded from a file named *fname*. The whole file is loaded into memory. This code even allows for very big files, because it dynamically allocates the memory. The C++ version is more compact because of the language features.

clCreateProgramWithSource

```
cl_program clCreateProgramWithSource ( cl_context context, cl_uint count,
    const char **strings, const size_t *lengths, cl_int *errcode_ret );
```

Creates a program object for a context and loads the source code specified by the text strings in the strings array into the program object.

cl::Program::Program

```
cl::Program::Program ( const Context& context, const Sources& sources,
    cl_int * err = NULL );
```

This constructor creates an OpenCL program object for a context and loads the source code specified by the text strings in each element of the vector sources into the program object.

In the C version example that can be seen in listing 2.46, the program object is created using the function **clCreateProgramWithSource**. The program is created in a given context. The OpenCL program source can be in multiple memory regions. This is allowed by the third parameter (here it is *appsource*), that is, an array of pointers to string buffers. The second parameter (here, it is 1) passes the number of program fragments. In the example, there is just a pointer to the array of characters because this code loads only one string array. The fourth parameter is an array of `size_t` values that determines the size of the corresponding string buffers. The last parameter is an error code.

C++ implementation (listing 2.47) also reads the whole file into a memory buffer, but this time the sizes and the strings are stored into a vector and inserted into a sources object; the OpenCL data type for sources is `cl::Program::Sources`. The program is created by the constructor **cl::Program** that takes the context and sources. It also produces the program object, which does not yet contain any kernels that can be executed. For the loading of binary program representation, please refer to section 2.12.3.

```
cl_program createProgram(cl_context context, char *fname) {
  cl_int r;
  size_t size = 0, partsize = 128;
  char buffer [4096];
  cl_program program;
  char *appsource = ( char * )malloc(128);

  FILE *f = fopen(fname, "r");
  if (f == NULL) {
    exit(EXIT_FAILURE);
  } else {
    while (partsize == 128) {
      partsize = fread(appsource + size, 1, 128, f);
      size += partsize;
      appsource = ( char* )realloc(appsource, 128 + size);
    }
    fclose(f);
  }
  program = clCreateProgramWithSource(context, 1,
    ( const char ** )&appsource, &size, &r);

  r = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  if (r != CL_SUCCESS) {
    cl_device_id device_id [16];
    clGetContextInfo(context,
      CL_CONTEXT_DEVICES, sizeof(device_id), &device_id, NULL);
    clGetProgramBuildInfo(program, device_id [0],
      CL_PROGRAM_BUILD_LOG, sizeof(buffer) - 1, buffer, &size);
    printf("Error: Could not build OpenCL program (%d)\n%s\n", r, buffer);
    exit(EXIT_FAILURE);
  }
  free(appsource);
  return program;
}
```

Listing 2.46: OpenCL program compilation – C code

```
cl::Program createProgram(cl::Context &context, std::string fname,
                          const std::string params = "") {
  cl::Program::Sources sources;
  cl::Program program;
  std::vector <cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES > ();

  std::ifstream source_file(fname.c_str());
  std::string source_code(
    std::istreambuf_iterator <char> (source_file),
    (std::istreambuf_iterator <char> ()));
  sources.insert(sources.end(),
    std::make_pair(source_code.c_str(), source_code.length()));
  program = cl::Program(context, sources);

  try {
    program.build(devices, params.c_str());
  } catch (cl::Error e) {
    std::cout << "Compilation build error log:" << std::endl <<
    program.getBuildInfo <CL_PROGRAM_BUILD_LOG > (devices [0]) << std::endl;
    throw e;
  }

  return program;
}
```

Listing 2.47: OpenCL program compilation – C++ code

OpenCL Program Build

clBuildProgram

```
cl_int clBuildProgram ( cl_program program, cl_uint num_devices,
    const cl_device_id *device_list, const char *options, void (CL_CALLBACK
    *pfn_notify)(cl_program program, void *user_data), void *user_data);
```

Builds (compiles and links) a program executable from the program source or binary.

Consider the source code that can be seen in listing 2.46 for C or 2.47 for C++. The program is built by the function **clBuildProgram** for C and **cl::Program::build** for C++. These functions order OpenCL to run a compiler to prepare a binary program representation. Error handling checks if the kernel has been built successfully, and if not, it gets the build log for the program and displays it. The error log is for the user; the standard does not provide any hints about the content of this log. The example build logs for erroneous compilation for the same program are in listings 2.48 and 2.49.

cl::Program::build

```
cl_int cl::Program::build ( const VECTOR_CLASS<Device> devices,
    const char *options = NULL, (CL_CALLBACK *pfn_notify) (cl_program,
    void *user_data) = NULL, void *data = NULL );
```

This method builds (compiles and links) a program executable from the program source or binary for all devices or (a) specific device(s) in the OpenCL context associated with the program.

```
ptxas application ptx input, line 26;
error : Label expected for argument 0 of instruction 'call'
ptxas application ptx input, line 26; error : Call target not recognized
ptxas application ptx input, line 26;
error : Function 'get_local_sizes' not declared in this scope
ptxas application ptx input, line 26; error : Call target not recognized
ptxas application ptx input, line 27; error : Unknown symbol 'get_local_sizes'
ptxas application ptx input, line 27;
error : Label expected for forward reference of 'get_local_sizes'
ptxas fatal : Ptx assembly aborted due to errors
```

Listing 2.48: The build log generated by an NVIDIA compiler

```
/ tmp / OCLoQsHmP.cl(21) : error : function "get_local_sizes" declared implicitly
  const size_t local_size = get_local_sizes(0);
^

1 error detected in the compilation of "/tmp/OCLoQsHmP.cl" .
```

Listing 2.49: The build log generated by an AMD compiler

```
cl_kernel kernel_x;
kernel_x = clCreateKernel(program, "kernel_x", NULL);
```

Listing 2.50: Usage of **clCreateKernel** to create **kernel** object – C code

```
cl::Kernel kernel_x;
kernel_x = cl::Kernel(program, "kernel_x");
```

Listing 2.51: Usage of **cl::Kernel** to create **kernel** object – C++ code

```
int i;
cl_uint num_kernels = 32;
cl_uint num_kernels_ret = 0;
cl_kernel kernels [32];
char value_s [128];
cl_uint value_i;
clCreateKernelsInProgram(program, num_kernels, kernels, &num_kernels_ret);
printf("The program contains %d kernels:\n", num_kernels_ret);
for (i = 0; i < num_kernels_ret; i++) {
  clGetKernelInfo(kernels [i], CL_KERNEL_FUNCTION_NAME, 128, value_s, NULL);
  clGetKernelInfo(kernels [i], CL_KERNEL_NUM_ARGS,
    sizeof(cl_uint), &value_i, NULL);
  printf(" - %s, %d args\n", value_s, value_i);
}
```

Listing 2.52: Usage of **clCreateKernelsInProgram** and **clGetKernelInfo** for listing available kernels – C code

Create Kernel Object

In listings 2.50 and 2.51 there are examples of kernel object creation. This is the most common way of performing this task – by giving the name of the kernel located in program. The program has to be already compiled.

The other method of kernel object creation is by using the function **clCreateKernelsInProgram** or **cl::Program::createKernels**. These functions allow for getting multiple kernels at once. The kernels are stored into an array or vector. This is very useful for programs that need automatic means of kernel creation. The sample code fragment that lists kernels available for a given program can be seen in listings 2.52 and 2.53. This code first creates multiple kernel objects at once, and then it checks the kernel names and parameters. The kernel information is obtained using the functions **clGetKernelInfo** and **cl::Kernel::getInfo** for C and C++ respectively.

```
std::vector<cl::Kernel> kernels;
ret = program.createKernels(&kernels);
std::cout << "The program contains " << kernels.size() << " kernels:" << std::endl;
for (unsigned i = 0; i < kernels.size(); i++) {
  std::cout << " - " <<
  kernels [i].getInfo<CL_KERNEL_FUNCTION_NAME>() <<
  ", " << kernels [i].getInfo<CL_KERNEL_NUM_ARGS>() <<
  " args" << std::endl;
}
```

Listing 2.53: Usage of **cl::Program::createKernels** and **cl::Kernel::getInfo** for listing available kernels – C++ code

cl::Kernel::Kernel

```
cl::Kernel::Kernel ( const Program& program, const char *name,
   cl_int *err = NULL );
```

This constructor will create a kernel object.

clCreateKernel

```
cl_kernel clCreateKernel ( cl_program program, const char *kernel_name,
   cl_int *errcode_ret );
```

Creates a kernel object.

cl::Program::createKernels

```
cl_int cl::Program::createKernels ( const VECTOR_CLASS<Kernel> *kernels );
```

This method creates kernel objects (objects of type `cl::Kernel`) for all kernels in the program.

clCreateKernelsInProgram

```
cl_int clCreateKernelsInProgram ( cl_program program, cl_uint num_kernels,
   cl_kernel *kernels, cl_uint *num_kernels_ret );
```

Creates kernel objects for all kernel functions in a program object.

cl::Kernel::getInfo

```
template <cl_int name> typename detail::param_traits<detail::cl_kernel_info,
   name>::param_type cl::Kernel::getInfo ( void );
```

The method gets specific information about the OpenCL kernel.

clGetKernelInfo

```
cl_int clGetKernelInfo ( cl_kernel kernel, cl_kernel_info param_name,
   size_t param_value_size, void *param_value, size_t *param_value_size_ret );
```

Returns information about the kernel object.

```
kernel void kernelparams(constant int *c, global float *data, local float *cache,
                         int value) {
  // ...
}
```

Listing 2.54: Kernel with parameters from different memory pools.

2.11.2. Kernel Invocation and Arguments

In OpenCL, kernel parameters must be passed using the appropriate API calls **clSetKernelArg** and **cl::Kernel::setArg** for C and C++, respectively. After a kernel parameter has been set, it is always present, no matter how many times the kernel is called. The host program does not have to set parameters each time the kernel is enqueued, but only when some parameters have changed.

The parameter is always accompanied with its size, except for C++ where the size of the parameter is passed implicitly based on the function **cl::Kernel::setArg** prototype. The values accepted in kernel parameters are memory buffers, the simple types and undefined values with given size. The last ones are used for allocating local memory arrays that can be used in the kernel.

clSetKernelArg

```
cl_int clSetKernelArg ( cl_kernel kernel, cl_uint arg_index,
    size_t arg_size, const void *arg_value );
```

Used to set the argument value for a specific argument of a kernel.

cl::Kernel::setArg

```
template <typename T> cl_int cl::Kernel::setArg ( cl_uint index,
    T value );
```

This method is used to set the argument value for a specific argument of a kernel.

Examples of setting parameters for the kernel from listing 2.54 are in listings 2.55 and 2.56. Every parameter is set by **clSetKernelArg** and **cl::Kernel::setArg**. For every parameter, one of these functions has to be invoked. OpenCL does not allow for setting all parameters using one call (contrary to, for example, CUDA). The C++ wrapper API provides helper class `cl::KernelFunctor` that allows for function-like kernel invocation, but it has some limitations and therefore it will be described later, in section 3.3.2.

The first kernel parameter (line marked (1)) is the constant memory buffer. This buffer has to be allocated by the host code. The data can be physically stored in the host memory or the device memory, but from the kernel perspective, it is just the constant memory buffer.

The second kernel parameter (line marked (2)) is the global memory buffer. Note that it is the same as in the previous parameter. This is correct, because memory buffers in OpenCL are allocated in the same way for constant and for global memory buffers.

```
cl_int int_value = 4;
ret = clSetKernelArg(kernelp, 0, sizeof(buffer_dev), &buffer_dev); // (1)
ret = clSetKernelArg(kernelp, 1, sizeof(buffer_dev), &buffer_dev); // (2)
ret = clSetKernelArg(kernelp, 2, sizeof(cl_float) * 4, NULL); // (3)
ret = clSetKernelArg(kernelp, 3, sizeof(cl_int), &int_value); // (4)
```

Listing 2.55: Setting kernel parameters – The C code.

```
cl_int int_value = 4;
ret = kernelp.setArg(0, buffer_dev); // (1)
ret = kernelp.setArg(1, buffer_dev); // (2)
ret = kernelp.setArg(2, sizeof(cl_float) * 4, NULL); // (3)
ret = kernelp.setArg(3, int_value); // (4)
```

Listing 2.56: Setting kernel parameters – The C++ code.

```
size_t global_size[] = { 4 };
size_t local_size[] = { 4 };
ret = clEnqueueNDRangeKernel(queue, kernelp,
  1, NULL, global_size, local_size, 0, NULL, NULL); // (5)
```

Listing 2.57: Invoking the kernel – the C programming language version

```
queue.enqueueNDRangeKernel(kernelp, cl::NullRange, cl::NDRange(4), cl::NDRange(4));
```

Listing 2.58: Invoking the kernel – the C++ programming language version

The next kernel parameter is the array located in the local memory pool. This memory cannot be accessed by the host program, nor can it be allocated in the same way as ordinary memory buffers or textures. It is allocated indirectly by setting this parameter (line marked (3) in listings 2.55 and 2.56). In this case, the size of the parameter is used to determine the size for the local memory buffer. The appropriate memory will be allocated in the local memory and passed to the kernel.

The third kernel parameter (line (4)) is a basic type variable. In the example, it is an `integer`, and it is copied directly as a kernel parameter. After this command exit, the value of the variable *`int_value`* can be altered by the host; it does not affect the value that is visible by the kernel.

After setting the parameters, the kernel can be enqueued. The parameters are not reset after kernel enqueue, so it is possible to execute the same kernel again with the same set of parameters, or change just one of them.

```
kernel void vector_sum(global float *v1, global float *v2, int size) {
  // ...
}

__kernel void vector_mul(__global float *v1, __global float *v2, int size) {
  // ...
}

__kernel void vector_algorithm(__global float *v1, __global float *v2, int size) {
  vector_sum(v1, v2, size);
  // ...
}
```

Listing 2.59: Three kernel declarations.

2.11.3. Kernel Declaration

The kernel function declaration looks like the ordinary C function prefixed with the keyword **`__kernel`** or **`kernel`** and with the return type `void`. These functions are intended to be run on OpenCL devices. The kernel is a function that is run on multiple work-items (cores). Note that both **`kernel`** and **`__kernel`** are correct, so for shorter code, the **`kernel`** version is advised.

Example kernels are declared in listing 2.59. A kernel can call another kernel; in that case, the called kernel is treated as an ordinary function. The kernel **`vector_algorithm`** uses **`vector_sum`** in this manner. Remember that using the local memory in a kernel function called by another kernel is implementation-defined; therefore, it is not recommended to use this feature in production software. It is also worth mentioning, that recursion is not included in the standard.

2.11.4. Supported Scalar Data Types

Most data types used in OpenCL programs are derived from the original C99 language. There are some additional types that are specific to OpenCL C, and some of the types defined by C99 are not present in OpenCL. The scalar data types available in OpenCL are listed in table 2.1.

Most types are the same as in C99. `size_t` does not have a counterpart in API because it has variable size, depending on the implementation. The special types `ptrdiff_t`, `intptr_t` and `uintptr_t` are used in addressing and therefore do not map directly into API types.

The types with equivalents in API can be used directly as kernel parameters. Consider the kernel shown in listing 2.60: it takes one parameter of the type `float`. This is an OpenCL C type. Setting a parameter for this kernel can be done in the way shown in listing 2.61. Note that there is no conversion of types. This is a one-to-one copy of data.

Remember that it is impossible to pass pointers to the kernel. This is because the OpenCL program uses different address space than the host program.

OpenCL C	API type	size	notes
bool	-	-	
char	cl_char	8	
unsigned char, uchar	cl_uchar	8	
short	cl_short	16	
unsigned short, ushort	cl_ushort	16	
int	cl_int	32	
unsigned int, uint	cl_uint	32	
long	cl_long	64	
unsigned long, ulong	cl_ulong	64	
float	cl_float	32	IEEE 754
half	cl_half	16	IEEE 754-2008
double	cl_double	64	optional, IEEE-754
size_t	-	32/64	
ptrdiff_t	-	32/64	
intptr_t	-	32/64	
uintptr_t	-	32/64	
void	void	0	

Table 2.1: List of scalar data types available in OpenCL C and its counterparts in C/C++ API.

```
cl_float num = 123.4;
// ...
clSetKernelArg(sample_kernel, 0, sizeof(cl_float), &num);
// ...
clEnqueueTask(queue, sample_kernel, NULL, NULL, NULL);
```

Listing 2.60: Host program setting one parameter to kernel.

```
kernel void sample(float number) {
  // ...
}
```

Listing 2.61: The kernel declaration with one parameter.

OpenCL arithmetic is strictly defined, in contrast to the ordinary C host program. In some cases, there can be some differences while computing the same algorithm on the host and using OpenCL. These issues are discussed in section 3.3.

Standard OpenCL 1.2 does not support `double` by default. The application must check the availability of this type by querying **`clGetDeviceInfo`** for the value of `CL_DEVICE_DOUBLE_FP_CONFIG`. This property tells if the `double` is supported by this device. In previous versions of OpenCL, the support for `double` was provided as an extension `cl_khr_fp64`. When an application wants to use this type, it should check if it is available. The usage of double type is described in section 3.3 and

OpenCL C	API type
char*n*	cl_char*n*
uchar*n*	cl_uchar*n*
short*n*	cl_short*n*
ushort*n*	cl_ushort*n*
int*n*	cl_int*n*
uint*n*	cl_uint*n*
long*n*	cl_long*n*
ulong*n*	cl_ulong*n*
float*n*	cl_float*n*

Table 2.2: The list of vector data types available in OpenCL C and its counterparts in C/C++ API

partially in section 3.1.

2.11.5. Vector Data Types and Common Functions

OpenCL defines multiple vector data types. Vector data type allows for natural representation of data used in various computational problems. The presence of these types also allows hardware vendors to create devices that accelerate operations on vectors. Every basic data type that has a counterpart in API has its accompanying vector type.

Vector data types are listed in table 2.2. The *n* can be 2, 3, 4, 8, 16 and means the size of the vector. Note that vectors are aligned to the power of two bytes, so the three-dimensional vectors are in fact four-dimensional vectors, and names like `cl_int3` actually refer to `cl_int4`. In order to properly access packed vectors from a kernel, one can use the functions **`vload3`** and **`vstore3`**.

The example usage of **`vload`***n* and **`vstore`***n* is in the sample kernel function **`normalization`** that can be seen in listing 2.62. This kernel performs normalization of every vector in a given array. The input and output array of vectors is in a packed format, so the data is not aligned. It is just stored in memory as $v_1.x, v_1.y, v_1.z, v_2.x, \ldots v_n.x, v_n.y, v_n.z$. The function **`vload3`** converts an unaligned vector into an aligned one, so it can be copied into `float3`. The next step uses the built-in function **`normalize`**, which normalizes the given vector. The last operation stores the result back into the packed vector array using **`vstore3`**.

vloadn

```
gentypen vloadn ( size_t offset, const mempool gentype *p );
```

Read vectors from a pointer to memory. The mempool can be **`constant`**, **`local`**, **`global`** or **`private`**.

Vector data types allow for very convenient vector manipulations inside an OpenCL program. All of the basic vector operations are implemented. The example kernel using vector data type can be seen in listing 2.63. Note that this kernel implements a fragment of many simulation programs – adding velocity multiplied by some time difference to the position.

vstoren

`void vstoren ( gentype*n* data, size_t offset, mempool gentype *p );`

Write **sizeof**(gentype*n*) bytes to memory. The mempool can be **constant**, **local**, **global** or **private**.

```
kernel void normalization(global float *v) {
  const size_t i = get_global_id(0);
  float3 vp = vload3(i, v);
  vp = normalize(vp);
  vstore3(vp, i, v);
}
```

Listing 2.62: The example kernel with usage of built in vector function with vload and vstore

normalize

`float*n* normalize ( floatn p );`

Normal vector length 1.

```
kernel void simulation(global float2 *positions, global float2 *velocities,
                    float dt) {
  const size_t i = get_global_id(0);
  positions [i] = positions [i] + velocities [i] * dt;
}
```

Listing 2.63: Kernel using `float2`

```
cl_float2 positions [4];
cl_float2 velocities [4];
size_t array_size = sizeof(cl_float2) * 4;
cl_float dt = 0.1;
```

Listing 2.64: Definition of array of vectors

The host can declare arrays of vectors in the way shown in listing 2.64. This is both for C and C++. The vector elements can be accessed using the name of the coordinate, like in listing 2.65, or by using the index of the coordinate, like in listing 2.66.

The OpenCL standard defines a set of built-in functions. Most of these functions are the same as in a standard math library, so there is no need for describing them here. The functions that expose information about the NDRange to the kernel has already been introduced in section 2.9.1. One big difference with C is that OpenCL extends standard C API for vector functions. The functions operating on vectors are listed and shortly described in table 2.3.

Each of these functions has its fast_ prefixed version. The fast functions can be faster than the standard ones, but they do not guarantee the same level of precision.

```
for (i = 0; i < 4; i++) {
  positions [i].x = 0.5 * i;
  positions [i].y = 0.5 * i;
  velocities [i].x = 0.5 * std::sin(i);
  velocities [i].y = 0.5 * std::cos(i);
}
```

Listing 2.65: Usage of vector types by coordinate name – the host code

```
for (i = 0; i < 4; i++) {
  positions [i].s [0] = 0.5 * i;
  positions [i].s [1] = 0.5 * i;
  velocities [i].s [0] = 0.5 * sin(i);
  velocities [i].s [1] = 0.5 * cos(i);
}
```

Listing 2.66: Usage of vector types by element index – the host code

OpenCL C function	Short description
cross	calculates cross product of two 3 or 4 dimensional vectors.
dot	calculates dot product of fectors
distance	returns the euclidean distance between two points in n-dimensional space
length	returns the length of a vector
normalize	Returns the normalized vector, that is the vector of the same direction as the original but with the length of 1.

Table 2.3: List of some OpenCL C functions for vectors.

OpenCL C function	Short description
isnan	checks if parameter is NaN
isinf	checks if parameter is infinity – positive or negative
isfinite	checks if parameter is finite value
vload*n*	loads aligned *n* component vector from packed buffer
vstore*n*	stores aligned *n* component vector into packed buffer
degrees	converts radians to degrees
radians	converts degrees to radians

Table 2.4: List of other OpenCL C functions operationg on float

Other useful functions are listed in table 2.4.

2.11.6. Synchronization Functions

OpenCL allows kernel instances (work-items) to synchronize within a work group. There is only one synchronization function: **`barrier`**. This function must be executed

on all work-items within a group. It blocks operation until every work-item reaches it. It also ensures memory consistency on the global or local memory. This function takes flags as a parameter for selecting what memory must be flushed. This parameter can be one or many of the values: CLK_LOCAL_MEM_FENCE, CLK_GLOBAL_MEM_FENCE.

barrier

```
void barrier ( cl_mem_fence_flags flags );
```

All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier.

mem_fence

```
void mem_fence ( cl_mem_fence_flags flags );
```

Orders loads and stores of a work-item executing a kernel.

read_mem_fence

```
void read_mem_fence ( cl_mem_fence_flags flags );
```

Read memory barrier that orders only loads.

write_mem_fence

```
void write_mem_fence ( cl_mem_fence_flags flags );
```

Write memory barrier that orders only stores.

The "barrier" function is commonly present in other parallel standards, so its usage is intuitive for experienced programmers. The example usage of this function is shown in section 2.11.7, so no further examples will be shown here.

OpenCL also allows for explicit memory fence functions in programs. These functions allow for ordering of memory operations at run time. The difference between the barrier and the fence is that the barrier synchronizes every work-item in a work-group and the memory fence synchronizes only on a specified memory operation. There are three memory fence operations that synchronize on local (CLK_LOCAL_MEM_FENCE) or global (CLK_GLOBAL_MEM_FENCE) memory – **mem_fence**, **read_mem_fence** and **write_mem_fence**.

The function **mem_fence** waits for all memory operations to finish – that is, read and write operations into a specified memory pool. The other two just wait for the write or read operations to finish.

The presence of barrier and fence functions allows for efficient work-item synchronization. The barrier works in every situation, but in many cases the fence functions can provide better efficiency.

2.11.7. Counting Parallel Sum

An example that uses different memory pools can be seen in listing 2.67. It presents a program containing one kernel called **`parallel_sum`**. This implements an algorithm called prefix sum.

Algorithm Description

This algorithm allows for summation of the values from an array. The presented version of the algorithm runs in $O(\log(n))$ time for an array of size n and on $O(n)$ processors. This algorithm uses the local memory to speed up computation and to allow cooperation of work-items in work-groups. The visual representation of the prefix sum algorithm is depicted in figure 2.18.

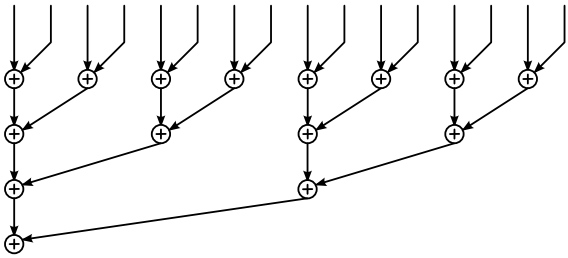

Figure 2.18: Graphical explanation of prefix sum.

The prefix sum is calculated by summing pairs of numbers. This allows for parallelization of this algorithm. For big arrays this algorithm is very efficient. Obviously, there would be no performance gain in cases where data would be copied into OpenCL device, only to calculate the sum. This is because of memory transfers – first the data must be copied into memory located in the device, then it would be calculated; eventually, the data would be copied back into the host. It is very likely that copying data would take more time than the time consumed by the standard sequential algorithm executed on a CPU. This algorithm is most useful as a part of some larger application.

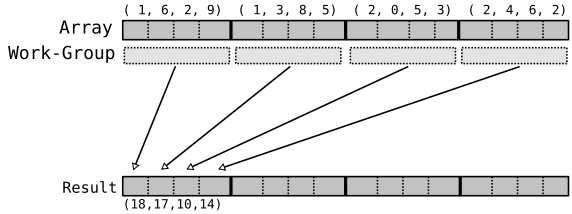

Figure 2.19: Execution of a prefix sum with four work groups and the (standard_results = 0). Note that the result can be directly processed by another prefix sum.

The presented version of this algorithm can behave in two different ways depending on the **`constant int`** *`standard_results`*. The first way

(standard_results = 0) is to save only the result from each work group into consecutive indexes of a ***results*** array. The second (standard_results = 1) copies the whole local memory array into a results array.

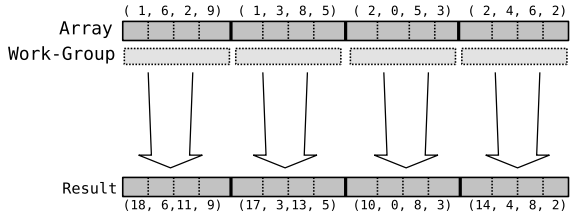

Figure 2.20: Execution of a prefix sum with four work groups and the (standard_results = 1). The result is a partial result of the full classical prefix sum.

2.11.8. Parallel Sum – Kernel

This version of a prefix sum can utilize the OpenCL execution model, so a kernel can be executed in many work-groups. Obviously, if the execution were performed in only one work-group, the global result would be ready to use just after the kernel exit. A kernel for calculating a prefix sum can be seen in listing 2.67. The kernel execution and determination of which parts of an input array would be processed by which work-group is shown in figures 2.19 and 2.20. This kernel includes simple debugging capabilities using **`printf`**. This function is available starting with the 1.2 OpenCL API version and is very useful in this example; it also justifies usage of the variable ***debug***. If debugging is enabled, information about each kernel instance statistics will be displayed.

```
// (1)
constant int debug = 0;
// (2)
kernel void parallel_sum(global float *array, global float *results,
                     local float *local_array,
                     constant int *standard_results) {
  // (3)
  int i;
  const size_t global_id = get_global_id(0), local_id = get_local_id(0);
  const size_t local_size = get_local_size(0), group_id = get_group_id(0);
  // (3b)
  local int additions [1];
  additions [0] = 0;

  // (4)
  local_array [local_id] = array [global_id];
  barrier(CLK_LOCAL_MEM_FENCE);

  // (5)
  for (i = 1; i < local_size; i = i * 2) {
    if ((local_id % (i * 2)) == 0) {
      local_array [local_id] = local_array [local_id] + local_array [local_id + i];
      if (debug == 1) atomic_inc(additions);
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  // (6)
  if (standard_results [0] == 1)
    results [global_id] = local_array [local_id];
  else
  if (local_id == 0)
    results [group_id] = local_array [local_id];
  // (7)
  if (debug) {
    #ifdef printf
    printf("%d: %f\n", global_id, local_array [local_id]);
    if (local_id == 0) printf("group %d performed %d add operations\n", group_id,
        additions [0]);
    #endif
  }
}
```

Listing 2.67: OpenCL program for counting parallel sum – slightly modified prefix sum or scan.

There are four different memory pools that are disjoint with each other: **__constant**, **__global**, **__local** and **__private** – corresponding to the **global**, **local**, **constant** and **private** memory regions. Note that these keywords can also be used without leading underscore characters ("**__**"). This is shown in the listing in kernel declaration – after the line marked by (2).

This example kernel uses all of the available memory regions. Here is a short description of each of them.

The Constant Memory Pool

The constant qualifier is used in example 2.67 in two places – the parameter in (2) and the constant value marked by (1). These are two ways of using this memory pool. The constant values are intended to be very fast, thanks to the cache mechanism, but this also prevents them from being writable. Note that variables in program scope can be only in the constant memory region. In the example, this construction allows for algorithm parametrization.

The Global Memory Pool

Global memory is one of the most common ways of passing arguments to a kernel. Consider the code that can be seen in listing 2.67. In the example in (2), there are two arguments using global memory – the *array* that is the input array for summation, and the *results* for the result. These are pointers to the global memory region prepared by the host program.

The Local Memory Pool

This memory pool is the only memory pool that allows for communication between work-items. Remember that local memory is shared only among work-items from the same work-group. There is no means of global synchronization and communication.

In the example, this pool is used both – for calculation and for work-items data exchange mechanism. The third argument in a kernel declaration points to the local memory region allocated for a kernel. This construction has another benefit – the algorithm should run more efficiently, because it reads global memory only once (fragment marked by (4)) and then operates on the local memory (fragment (5)) until the result is calculated. Then it writes it into the global memory (in (6)).

Local memory is also used for algorithm statistics. The variable *additions* is declared inside a kernel. The construction marked by (3*b*) is another way of allocating the local memory buffer. This variable is then increased by every work-item, using **atomic_inc** in (5). This is an atomic operation and allows for calculation of the number of addition operations performed by this work-group.

atomic_inc

```
int atomic_inc ( volatile __global int *p );
```

Read the 32-bit value (referred to as old) stored at the location pointed by p. Compute $(old + 1)$ and store result at the location pointed by p. The function returns old.

Note that a local memory region passed as a parameter for a kernel allows for dynamic allocation of this memory. Inside the kernel body, the local memory buffer declarations must be static.

The Private Memory Pool

The variables in the local memory pool are declared inside the kernel function. These are located in fragment (3) of the example that can be seen in listing 2.67. In fact, there is one private variable – `i`. There are also **`const`** values, but they are not in constant memory; these are constants that work exactly like their equivalents in C99. These variables are attached to a work-item, and it cannot be used for data exchange.

2.11.9. Parallel Sum – Host Program

An application that uses OpenCL must contain two elements – the OpenCL program and the host program. The host code function that calculates a parallel sum can be seen in listings 2.68 and 2.69 for C and C++, respectively.

Both codes define the memory buffers located in the OpenCL context in (2). This must be done, because it is not possible to pass memory pointers directly into a kernel. Next, the kernel parameters are set in (3). Note that not all parameters are set here. The parameters set in (3) are only the ones that are constant for every kernel invocation. The main loop of the algorithm is in (4). This part sets the remaining parameters according to the current step of calculation. Remember that each parameter must be set before the kernel can execute. The kernel is enqueued to run in an of size i with work-groups of size wgs. This loop allows for iterated execution of the algorithm described in the beginning of section 2.11.7. In each step, the result array and the input array are switched. The results of one parallel sum kernel execution go to the other kernel execution as input value. This loop finishes when the resultant array is of size 1 and that is the result gathered and returned in (5). Note that the order of steps in both – C and C++ – versions is the same; only the code conventions differ. This is because it executes the same algorithm.

```
cl_float parallelSum(cl_context context, cl_command_queue queue,
                     cl_program program, int n, cl_float *array) {
  // (1)
  int i;
  cl_float sum = 0;
  int wgs = 4;
  size_t array_size = sizeof(cl_float) * n;
  cl_int ret;
  cl_int options[] = { 0 };
  cl_kernel parallel_sum;
  parallel_sum = clCreateKernel(program, "parallel_sum", NULL);
  // (2)
  cl_mem array_dev = clCreateBuffer(context,
    CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, array_size, array, &ret);
  cl_mem results_dev = clCreateBuffer(context,
    CL_MEM_READ_WRITE, array_size, NULL, &ret);
  cl_mem options_dev = clCreateBuffer(context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_int), options, &ret);
  size_t local_size [1], global_size [1];
  // (3)
  ret = clSetKernelArg(parallel_sum, 2, array_size, NULL);
  ret = clSetKernelArg(parallel_sum, 3, sizeof(options_dev), &options_dev);
  // (4)
  i = n;
  while (i > 0) {
    clFinish(queue);
    ret = clSetKernelArg(parallel_sum, 0, sizeof(array_dev), &array_dev);
    ret = clSetKernelArg(parallel_sum, 1, sizeof(results_dev), &results_dev);
    local_size [0] = (i > wgs) ? wgs : i;
    global_size [0] = i;
    ret = clEnqueueNDRangeKernel(queue, parallel_sum, 1,
      NULL, global_size, local_size, 0, NULL, NULL);
    i /= wgs;
    switchValues(&results_dev, &array_dev);
  }
  // (5)
  ret = clEnqueueReadBuffer(queue, array_dev, CL_TRUE, 0, sizeof(cl_float), &sum,
    0, NULL, NULL);
  return sum;
}
```

Listing 2.68: Host code for parallel sum algorithm. The C programming language.

```
cl_float parallelSum(cl::Context context, cl::CommandQueue queue,
                     cl::Program program, int n, cl_float *array) {
  // (1)
  int i;
  cl_float sum = 0;
  int wgs = 4;
  size_t array_size = sizeof(cl_float) * n;
  cl_int options[] = { 0 };
  cl::Kernel parallel_sum;
  parallel_sum = cl::Kernel(program, "parallel_sum");
  // (2)
  cl::Buffer array_dev(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                       array_size, array);
  cl::Buffer results_dev(context, CL_MEM_READ_WRITE, array_size);
  cl::Buffer options_dev(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                         sizeof(cl_int), options);
  // (3)
  parallel_sum.setArg(2, array_size, NULL);
  parallel_sum.setArg(3, options_dev);
  // (4)
  i = n;
  while (i > 0) {
    parallel_sum.setArg(0, array_dev);
    parallel_sum.setArg(1, results_dev);
    queue.enqueueNDRangeKernel(parallel_sum, cl::NullRange, cl::NDRange(i),
      cl::NDRange((i > wgs) ? wgs : i));
    i /= wgs;
    switchValues(&results_dev, &array_dev);
  }
  // (5)
  queue.enqueueReadBuffer(array_dev, CL_TRUE, 0, sizeof(cl_float), &sum);
  return sum;
}
```

Listing 2.69: Host code for parallel sum algorithm. The C++ programming language.

```
cl_int r; // this variable is for storing error codes.
cl_context context; // computation context
cl_command_queue queue; // command queue
cl_program program; // this variable will hold our program
cl_kernel vectSumKernel; // this represents specified kernel in program

cl_device_id context_devices [1]; // list of devices included in context
```

Listing 2.70: Variables used in a vector sum example.

2.12. Structure of the OpenCL Host Program

An OpenCL application consists of two kinds of code – one executed on the host and the second executed on the compute device. The host code is a supervisor of the program. Its responsibility is to load the OpenCL program into the memory, compile it and select which kernels to use. It also checks for errors, manages memory buffers and provides user interaction.

This section will discuss the host code, which is common for a wide range of applications. The device code here is very small. OpenCL standard has been created in the spirit of giving the programmer as much control as possible. The price for this is additional code for using the device and preparing execution of kernels.

The execution of most OpenCL applications can be generalized using the sequence of steps it performs. The usual OpenCL host code has four distinct steps:

- Initialization - creating context
- Preparation of OpenCL programs - creating kernels
- Computation - creating command queues, executing kernels and managing memory buffers
- Release of resources (context, queues, kernels)

There is the possibility that these steps do not always apply, but they are used most of the time. OpenCL programs are loaded and compiled at the very beginning of an application. There are, of course, situations where this schema will not apply – for example, applications that need to run dynamically generated code.

Note that in real life, these steps should be performed in dedicated functions or objects.

The following example program will perform a floating point vector sum. This is a very simplified example and does not load dynamically allocated vectors from the user. It also does not perform sufficient error checking, which would be required in production implementation.

These limitations are reasonable because the code here should be as clear as possible. The variables used later are defined in listing 2.70. These variables hold global information about the most important OpenCL objects. Variable called *context* holds the OpenCL context, *queue* represents the command queue that will receive and execute commandsm, and *program* stores the OpenCL program.

```
cl_platform_id platforms [4]; // list of available platforms
cl_uint num_platforms; // number of OpenCL platforms
cl_context_properties context_properties [3];
// First we have to obtain available platforms
r = clGetPlatformIDs((cl_uint)2, platforms,
  &num_platforms);
// next we have to select one platform, we can choose first one available.
context_properties [0] = (cl_context_properties)CL_CONTEXT_PLATFORM;
context_properties [1] = (cl_context_properties)platforms [0];
context_properties [2] = (cl_context_properties)0;

// now we can initialize our computational context
context = clCreateContextFromType(context_properties,
  CL_DEVICE_TYPE_ALL, NULL, NULL, &r);
if (r != CL_SUCCESS) {
  printf("err:%d\n", r); exit(-1);
}
// we have to obtain devices which are included in context
r = clGetContextInfo(context, CL_CONTEXT_DEVICES,
  1 * sizeof(cl_device_id), context_devices, NULL);
```

Listing 2.71: Simple initialization of OpenCL.

2.12.1. Initialization

In order to create a computation context, OpenCL requires the application to choose the OpenCL implementation and the devices. This usually cannot be done automatically by selecting a default implementation and a device. In this phase, the application can also query which devices are available for different implementations.

In listing 2.71, there is a simple way of initializing an OpenCL context. It invokes **clGetPlatformIDs** in order to get the list of available platforms. Here, it retrieves only one. The next step is to create a context. OpenCL provides the function **clCreateContextFromType** for creating a computation context. This function must be invoked with at least the first two parameters set – *properties* and *device_type*. Note that in version 1.2 of OpenCL standard, the properties can contain only one CL_CONTEXT_PLATFORM with the corresponding *cl_platform_id*. There is no possibility to create a context containing multiple platforms. If the properties are set to NULL, then the behavior of this function is implementation-specific, so it will not be discussed further. The device type is a bit field which identifies what kind of device or devices should be included in the newly created context. In this example, the context is created with all available devices.

It is also possible to check the device frequency and number of processors, so the application can estimate performance and select the fastest device available. This can be done using the API call **clGetDeviceInfo** with the parameter *param_name* set to MAX_COMPUTE_UNITS and MAX_CLOCK_FREQUENCY.

The initialization code that can be seen in listing 2.72 shows the second and more sophisticated approach. Flops are estimated using the simple formula shown in equation 2.1, where *mcu* stands for Max Compute Units and *mcf* stands for Max Clock Frequency.

```
int i, j; // counters
cl_platform_id platforms [32]; // list of available platforms
cl_uint num_platforms; // number of OpenCL platforms
cl_context_properties context_properties [3];
cl_platform_id fastestPlatform = 0;
cl_device_id fastestDevice = 0;
cl_uint flopsMax = 0;
// First obtain available platforms
r = clGetPlatformIDs((cl_uint)32, platforms, &num_platforms);
for (i = 0; i < num_platforms; i++) {
  cl_device_id *devices;
  cl_uint devsCount, freq, ncu;
  clGetDeviceIDs(platforms [i], CL_DEVICE_TYPE_ALL, 0, NULL, &devsCount);
  printf("There are %d devices on platform %d\n", devsCount, i);
  devices = (cl_device_id *)malloc(sizeof(cl_device_id) * devsCount);
  clGetDeviceIDs(platforms [i], CL_DEVICE_TYPE_ALL, devsCount, devices, NULL);
  for (j = 0; j < devsCount; j++) {
    clGetDeviceInfo(devices [j], CL_DEVICE_MAX_CLOCK_FREQUENCY,
      sizeof(cl_uint), &freq, NULL);
    clGetDeviceInfo(devices [j], CL_DEVICE_MAX_COMPUTE_UNITS,
      sizeof(cl_uint), &ncu, NULL);
    printf("Flops (estimated): %dMFlops (%d Mhz * %d units)\n",
      freq * ncu, freq, ncu);
    if (flopsMax < (freq * ncu)) {
      fastestPlatform = platforms [i];
      fastestDevice = devices [j];
      flopsMax = freq * ncu;
    }
  }
  free(devices); devices = NULL;
}
printf("Selected device with estimated : %d MFlops\n", flopsMax);
// select platform
context_properties [0] = (cl_context_properties)CL_CONTEXT_PLATFORM;
context_properties [1] = (cl_context_properties)fastestPlatform;
context_properties [2] = (cl_context_properties)0;
// initialize computational context
context = clCreateContext(context_properties, 1, &fastestDevice, NULL, NULL, &r);
if (r != CL_SUCCESS) printf("Error: %d\n", r); exit(-1);
context_devices [0] = fastestDevice;
```

Listing 2.72: Initialization of an OpenCL context with the fastest (estimated) available device.

$$\text{flops} = \text{mcu} \times \text{mcf} \tag{2.1}$$

Note that the result is very imprecise. This is because the formula assumes that every floating point operation is performed using only one device tick. The assumed behavior is not always realized. This assumption is justified by its simplicity in imple-

mentation and it gets the fastest device when there are sufficiently great performance differences between them.

The application gets the list of available platforms using **clGetPlatformIDs**. Here, it is assumed that there are no more than 32 available. The application iterates through all platforms. In every iteration, it gets the device count in the current platform, using **clGetDeviceIDs**. This function sets the last parameter to the count of all devices in the platform when the fourth parameter is set to `NULL`. Next, the program allocates a buffer for the list of available device IDs and reads them using **clGetDeviceIDs** again, this with the buffer and its size set. The information about each OpenCL device is checked using **clGetDeviceInfo**. It checks the device frequency and the device computing units. Comparing estimated flops allows it to select the platform and device with the highest performance.

The last step is to create a context using the **clCreateContext** function. Contrary to **clCreateContextFromType**, this function allows it to specify the particular devices and platform to use. The last instruction saves the device ID for use in the later stages of the example application, especially while checking build log.

2.12.2. Preparation of OpenCL Programs

In the example, the program code is stored in a variable called *vectSumSrc*. The kernel performing the vector sum is shown in listing 2.73. The keyword **kernel** tells the OpenCL compiler that this function can be used as a kernel. The kernel is named *vectSum* and takes four parameters:

- *v1* – pointer to global memory, which stores first vector as an array of float values.
- *v2* – pointer to global memory, which stores second vector as an array of float values.
- *r* – result vector. This also points to global memory.
- *n* – size of vector. This parameter is passed directly by the function **clSetKernelArg**.

Each kernel instance counts its own vector element. The unique ID of a work-item is retrieved using the OpenCL C language function **get_global_id**. A more comprehensive description of kernel programming can be found in section 2.11.

Loading and compilation of kernels is usually done just after creation of a context. The application needs to create a compiled program object. The program can be loaded as a binary using **clCreateProgramWithBinary** or as a source code using **clCreateProgramWithSource**. The most common way is to use a source, because it makes application more portable and it allows for use of dynamically generated code. Loading a program from binary is often used as a way to speed up application initialization. The program in vendor-specific form can be obtained using **clGetProgramInfo**. Then it can be stored in a cache for the next usage. The example caching of a binary OpenCL program is presented in section 2.12.3

The next step is to build an OpenCL program using **clBuildProgram**. This function prepares a program for execution on the device. It generates a binary representation executable on the device. This function must always be executed, even if the function **clCreateProgramWithBinary** has been used to create the program.

```
char *vectSumSrc =
  "kernel void vectSum ( "
  " global float *v1, "
  " global float *v2, "
  " global float *r, "
  " int n) { "
  " const int i = get_global_id(0); "
  " r[i] = v1[i] + v2[i]; "
  "} ";
```

Listing 2.73: OpenCL program containing one kernel. The kernel computes a vector sum.

clCreateProgramWithBinary

```
cl_program clCreateProgramWithBinary ( cl_context context, cl_uint num_devices,
    const cl_device_id *device_list, const size_t *lengths, const unsigned char
    **binaries, cl_int *binary_status, cl_int *errcode_ret );
```

Creates a program object for a context, and loads the binary bits specified by binary into the program object.

This is because a binary form can be in implementation-specific intermediate representation (for example, assembler) or device-specific executable bits or even both. If the program has been created using source code, this function compiles it into binary form, which can be executed. The compilation log can be checked using **clGetProgramBuildInfo**.

clGetProgramBuildInfo

```
cl_int clGetProgramBuildInfo ( cl_program program, cl_device_id device,
    cl_program_build_info param_name, size_t param_value_size,
    void *param_value, size_t *param_value_size_ret );
```

Returns build information for each device in the program object.

In listing 2.74, there is an example of loading a program into memory, building it and creating a kernel object. Error checking is included. OpenCL allows for getting the build log. This log is always available after compilation, but using it is most useful in the case of failure. It contains human-readable information about the process of compilation and error locations. To retrieve it, the function **clGetProgramBuildInfo** is used. The required parameters are the program and device to query. The build log is stored in a buffer as a null-terminated string. The output string format is implementation-dependent, so in most cases it is not possible to parse it automatically, but it proves to be very useful in the development stage of an application.

2.12.3. Using Binary OpenCL Programs

OpenCL programs are compiled for a given hardware brand and model. The executables are not portable, so in most cases the source code representation of a program

```
// load program into context
program = clCreateProgramWithSource(context,
  1, (const char **)(&vectSumSrc),
  NULL, &r);
// compile program
r = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (r != CL_SUCCESS) {
  char buffer [4096];
  printf("Error building program.\n");
  clGetProgramBuildInfo(program, context_devices [0], CL_PROGRAM_BUILD_LOG,
    sizeof(buffer), buffer, NULL);
  printf("Compilation log:\n%s\n", buffer);
  exit(0);
}
// create kernel object
vectSumKernel = clCreateKernel(program, "vectSum", &r);
```

Listing 2.74: Loading the OpenCL program into a context and creating a kernel object.

is a good choice. Binary representation, however, can be obtained and reused. This is very useful for big projects, where compilation can consume significant amounts of time. This approach is also one of the ways to keep algorithmic solutions hidden from users. The reader should be aware that *this method is not advised for code obfuscation*, because the binary program representation will likely fail on a different version of hardware or even on the same hardware with a different driver version. The intended use of this feature is to optimize application loading times.

An example of compiled code usage is shown in listings 2.75 and 2.76. This example implements the usage of binary representation as a cache for an OpenCL program. The assumption is that the program has been compiled for only one device.

In this example, the file containing binary program representation is expressed in the following format:

- First **sizeof**(*binary_program_size*) bytes are storing the size of binary data.
- Next is *binary_program_size* bytes of binary device program.

The first step, to generate the file name, is shown in listing 2.75. This is important because there should be some way to determine for which device the code has been compiled. This code sample retrieves the device name and vendor name of the device utilized in this example. The function **clGetDeviceInfo** is used for getting information about the device. Here, the buffers are of fixed size for simplicity. The file name is generated as a series of hex digits. The resultant file name is stored in a variable called *file_name*.

The code in listing 2.76 checks the existence of a file with compiled binaries. If a binary file exists, then this host code loads the compiled program into the memory buffer *binary_program* of size *binary_program_size*. The function **clCreateProgramWithBinary** creates an OpenCL program object using a given binary buffer. Note that the format of this buffer is implementation-dependent. In most cases, it is not portable between different devices.

```
size_t str_size;
char dev_name [500], vendor_name [500], file_name [1024];
FILE *file;
unsigned char *binary_program;
size_t binary_program_size;
// check the current device name and vendor and generate apropriate file name
r = clGetDeviceInfo(context_devices [0], CL_DEVICE_NAME, 500, dev_name, &str_size);
r = clGetDeviceInfo(context_devices [0], CL_DEVICE_VENDOR, 500, vendor_name,
  &str_size);
// create buffer for file name
sprintf(file_name, "%s_%s", vendor_name, dev_name);
str_size = strlen(file_name);
// the file name will be modified in order to work with any modern operating system
for (i = str_size - 1; i >= 0; i--) {
  file_name [i * 2] = (file_name [str_size - 1 - i] & 0xf) + 'a';
  file_name [i * 2 + 1] = (file_name [str_size - 1 - i] >> 4) + 'a';
}
file_name [str_size * 2] = 0;
strcat(file_name, ".bin");
```

Listing 2.75: Generating a file name for the given architecture.

The function **clBuildProgram** must be used, even though the binary representation has been loaded. This stage allows OpenCL implementation to process the program in binary form. Some vendors use this stage for linking or processing the assembler format.

If the binary file does not exist, then compilation from the sources is needed. This part is done in the same way as in listing 2.74. After compilation, the program size can be read using **clGetProgramInfo** with switch CL_PROGRAM_BINARY_SIZES. There is a plural form because it is possible to have a program which is compiled for multiple devices. This size is saved into the variable *binary_program_size*. Next, the memory buffer is allocated to store *binary_program_size* bytes of the compiled program. Function **clGetProgramInfo** with the CL_PROGRAM_BINARIES switch is used for reading compiled program data into the memory buffer. After this operation, the binary data is saved into a file for future use.

The last step here is creating the kernel object. This is done in the usual way – using **clCreateKernel**. In both cases, the binary program data can be freed.

An alternative way of getting binary representation for every device that the program is compiled to is shown in listing 2.77. The steps are almost the same, except for using *program_dev_n* binaries instead of one.

2.12.4. Computation

At this stage, the actual computations are performed. Here, the buffers on the device are allocated or freed and data transfers are performed. In listing 2.78, there is a very short example of computing the vector sum. The kernel has been initialized in section 2.12.2, and it is time to use it in real computation. The example alone will not show any improvement in performance compared to the sequential version,

```
file = fopen(file_name, "rb");
if (file != NULL) {
  printf("Loading binary file '%s'\n", file_name);
  fread(&binary_program_size, sizeof(binary_program_size), 1, file);
  binary_program = (unsigned char *)malloc(binary_program_size);
  fread(binary_program, binary_program_size, 1, file);
  program = clCreateProgramWithBinary(context, 1, context_devices,
    &binary_program_size, (const unsigned char **)&binary_program, NULL, &r);
  if (r != CL_SUCCESS) exit(-1);
  fclose(file);
  if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS)
    exit(-1);
} else {
  printf("Saving binary file '%s'\n", file_name);
  // load program into context
  program = clCreateProgramWithSource(context, 1,
    (const char **)(&vectSumSrc), NULL, &r);
  // compile program
  if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS) exit(-1);
  // save binary data
  clGetProgramInfo(program, CL_PROGRAM_BINARY_SIZES, sizeof(size_t),
    &binary_program_size, NULL);
  binary_program = (unsigned char *)malloc(binary_program_size);
  clGetProgramInfo(program, CL_PROGRAM_BINARIES, sizeof(unsigned char **),
    &binary_program, &str_size);
  file = fopen(file_name, "wb");
  fwrite(&binary_program_size, 1, sizeof(size_t), file);
  fwrite(binary_program, 1, binary_program_size, file);
  fclose(file);
}
// create kernel object
vectSumKernel = clCreateKernel(program, "vectSum", &r);
if (r != CL_SUCCESS) exit(-1);
free(binary_program);
```

Listing 2.76: Selecting source or binary form of an OpenCL program. Example shows approach to cache problem.

because memory transfers will damage performance of the computation. But as part of a larger project, this method can be very useful.

The first lines in listing 2.78 prepare host buffers storing vectors and the buffer for the result vector. In actual application, these vectors would be much longer and initialized in some more sophisticated way, for example from a file. The variable *vectorSize* is for storing the size of input vectors. Next, there are three variables of the type `cl_mem`, which will be initialized later. These are pointers to the memory buffers in the OpenCL device. This is the way to reference memory objects from the host program.

The API call **clCreateCommandQueue** creates a command queue object. Every computation and data transfer is performed by putting commands into a queue. The third parameter – *properties* – is set to 0, meaning that this queue is created using default options. Commands will be executed in order, and profiling is disabled.

```
cl_uint program_dev_n;
clGetProgramInfo( program, CL_PROGRAM_NUM_DEVICES, sizeof(cl_uint), &program_dev_n,
     NULL );
size_t binaries_sizes[program_dev_n];
clGetProgramInfo(program,CL_PROGRAM_BINARY_SIZES, program_dev_n*sizeof(size_t),
    binaries_sizes, NULL );
char **binaries = malloc(sizeof(char **)*program_dev_n);
for (size_t i = 0; i < program_dev_n; i++)
  binaries[i] = malloc( sizeof(char)*(binaries_sizes[i]+1) );
clGetProgramInfo(program, CL_PROGRAM_BINARIES, program_dev_n*sizeof(size_t),
    binaries, NULL);
```

Listing 2.77: Getting the binary representation for all devices.

```
cl_float vector1 [3] = { 1, 2, 3.1 }; // sample input data
cl_float vector2 [3] = { 1.5, 0.1, -1 };
cl_float result [3];
cl_int vectorSize = 3; // size of input data
cl_mem vector1_dev, vector2_dev, result_dev; // memory on the device
printf("vector1 = (%.1f,%.1f,%.1f)\n", vector1 [0], vector1 [1], vector1 [2]);
printf("vector2 = (%.1f,%.1f,%.1f)\n", vector2 [0], vector2 [1], vector2 [2]);
// next we have to prepare command queue
queue = clCreateCommandQueue(context, context_devices [0], 0, &r);
// we have to prepare memory buffer on the device
vector1_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  vectorSize * sizeof(cl_float), vector1, &r);
vector2_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  vectorSize * sizeof(cl_float), vector2, &r);
result_dev = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
  vectorSize * sizeof(cl_float), NULL, &r);
// set parameters for kernel
clSetKernelArg(vectSumKernel, 0, sizeof(cl_mem), &vector1_dev);
clSetKernelArg(vectSumKernel, 1, sizeof(cl_mem), &vector2_dev);
clSetKernelArg(vectSumKernel, 2, sizeof(cl_mem), &result_dev);
clSetKernelArg(vectSumKernel, 3, sizeof(cl_int), &vectorSize);
// add execution of kernel to command queue
size_t dev_global_work_size [1] = { vectorSize };
clEnqueueNDRangeKernel(queue, vectSumKernel,
  1, NULL, dev_global_work_size, NULL, 0, NULL, NULL);
// now we have to read results and copy it to hello_txt
clEnqueueReadBuffer(queue, result_dev, CL_TRUE, 0,
  vectorSize * sizeof(cl_float), result, 0, NULL, NULL);
printf("result = (%.1f,%.1f,%.1f)\n", result [0], result [1], result [2]);
// free memory buffer
clReleaseMemObject(vector1_dev);
clReleaseMemObject(vector2_dev);
clReleaseMemObject(result_dev);
```

Listing 2.78: Computation and memory management.

Now it is time to create buffers in the device memory. This is done using **clCreateBuffer**. In the example, the device buffers for holding operands are created and initialized for values copied from *vector1* and *vector2*. This behavior is driven by the setting CL_MEM_COPY_HOST_PTR in *flags*. The flag CL_MEM_READ_ONLY marks the buffer as a read-only for kernels. This can allow for better optimization of memory access by the drivers and less error-prone source code. The last one only creates the buffer and does not initialize it. The buffer is marked as write-only by the flag CL_MEM_WRITE_ONLY.

Kernel parameters are set using function **clSetKernelArg**. This function gets the kernel object, parameter number, size of argument and argument as parameters. The kernel parameters are numbered from 0. The last parameter, *vectorSize*, is just one element of a simple type, so it can be passed directly. In production implementation, the errors should be checked, but in the example it is assumed that if the context and kernel have been created successfully, then the rest of the application will work correctly.

Note that the parameters, once set, hold their values. The values passed to the kernel are copied. For example, changing the *vectorSize* value after calling **clSetKernelArg** does not affect the value visible from kernel.

Kernel execution is done via a command queue using **clEnqueueNDRangeKernel** or **clEnqueueTask**. The second one enqueues execution of a single kernel instance and is not suitable for kernel summing vectors. In the example, the kernel will be enqueued to be executed in a one-dimensional work space of size *vectorSize*. This is achieved using the parameter *dev_global_work_size*. Note that the kernel in listing 2.73 uses only one **get_global_id** with 0 as a parameter. This is because of the one-dimensional computation space created by the function **clEnqueueNDRangeKernel**, that can be seen in line 24 of listing 2.78.

Now the results should be retrieved, using **clEnqueueReadBuffer**. This function puts into a queue the command to copy device memory buffer content into the host memory buffer. This is a blocking function because of the third parameter – *blocking_read* – set to 1. If it were to 0, then the command would immediately return, not waiting for the command to finish.

Note that in the moment when the **clEnqueueReadBuffer** function is invoked, the kernel can still be in execution. The queue is processed as a parallel task, and objects put into it do not always execute immediately. The solution presented in listing 2.78 shows the way to synchronize the host and device programs by using this function. The other way to synchronize them is by using **clFinish**.

The last step is to free the memory buffers. This is done by using the function **clReleaseMemObject**. This function decrements the memory object reference count; when it reaches 0, the memory is released.

clReleaseMemObject

```
cl_int clReleaseMemObject ( cl_mem memobj );
```

Decrements the memory object reference count. After the *memobj* reference count becomes zero and commands that are queued for execution on a command-queue(s) that use *memobj* have finished, the memory object is deleted.

```
1 // free kernel and program
2 clReleaseKernel(vectSumKernel);
3 clReleaseProgram(program);
4
5 // free resources
6 clReleaseCommandQueue(queue);
7 r = clReleaseContext(context);
```

Listing 2.79: Releasing OpenCL objects.

2.12.5. Release of Resources

clReleaseKernel

```
cl_int clReleaseKernel ( cl_kernel kernel );
```

Decrements the kernel reference count. The kernel object is deleted once the number of instances that are retained to the kernel become zero and the kernel object is no longer needed by any enqueued commands that use the kernel.

clReleaseCommandQueue

```
cl_int clReleaseCommandQueue ( cl_command_queue command_queue );
```

Decrements the *command_queue* reference count. After the *command_queue* reference count becomes zero and all commands queued to *command_queue* have finished, the command-queue is deleted.

clReleaseProgram

```
cl_int clReleaseProgram ( cl_program program );
```

Decrements the *program* reference count. The program object is deleted after all kernel objects associated with *program* have been deleted and the *program* reference count becomes zero.

clReleaseContext

```
cl_int clReleaseContext ( cl_context context );
```

Decrements the context reference count.

Every well-written application should perform a cleanup of resources. The buffers were managed in the previous phase – computation. In the current stage, the application should release the computation context, command queue, kernels and program. The code performing these operations is shown in listing 2.79. Note that according to OpenCL 1.2 standard specification, the context is released only when there are no more references to it and all the objects attached to it are also freed.

```
#include <cstdlib>
#include <vector>
#include <iostream>

#define __CL_ENABLE_EXCEPTIONS
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif
```

Listing 2.80: Macro definition and necessary header files

```
cl::Context context; // computation context
cl::CommandQueue queue; // command queue
cl::Program program; // this variable will hold our program
cl::Kernel vectSumKernel; // this represents specified kernel in program
std::vector < cl::Device > contextDevices; // list of devices included in context
```

Listing 2.81: Global variables for vector sum example

2.13. Structure of OpenCL host Programs in C++

Consider again the vector summation algorithm. The C++ host code structure is basically the same as in C. There are also the same steps:

- Initialization – creating context and command queues
- Preparation of OpenCL programs – creating kernels
- Computation – executing kernels and managing memory buffers
- Release of resources (context, queues, kernels)

The header files and definition of macro `__CL_ENABLE_EXCEPTIONS` can be seen in listing 2.80. This definition allows for usage of exceptions. This is for many people a more convenient way of dealing with errors and is also compatible with object-oriented programming paradigms.

Note that there is a conditional block – one for Apple and one for other platforms – because in MacOS X, the header files are stored in different locations specific to this system. On other systems, the header files are located in the `CL` directory that is intuitive for users of, for example, OpenGL (directory `GL`).

Variables that are used in the whole example are in listing 2.81. These are objects with empty constructors.

The variable *`context`* stores the object that represents the OpenCL context. Every calculation is performed within the context so it is the most important object for the OpenCL application.

The command queue object is stored in the variable *`queue`*. In the example, there is only one command queue. Note that OpenCL allows for multiple command queues.

```
std::vector < cl::Platform > platforms; // list of available platforms
cl_context_properties context_properties [3];
// First we have to obtain available platforms
cl::Platform::get(&platforms);
// next we have to select one platform, we can choose first one available.
context_properties [0] = (cl_context_properties)CL_CONTEXT_PLATFORM;
context_properties [1] = (cl_context_properties)(platforms [0])();
context_properties [2] = (cl_context_properties)0;
// now we can initialize our computational context
context = cl::Context(CL_DEVICE_TYPE_ALL, context_properties);
// we have to obtain devices which are included in context
contextDevices = context.getInfo < CL_CONTEXT_DEVICES > ();
// next we have to prepare command queue
queue = cl::CommandQueue(context, contextDevices [0]);
```

Listing 2.82: Initialization of necessary objects

The program is represented by the *program* variable. It will store the compiled program – in this example, it is only one kernel. This kernel is represented by the variable *vectSumKernel*.

2.13.1. Initialization

Initialization steps common for many OpenCL applications are in listing 2.82. A few temporary variables are used in initialization – *platforms* and *context_properties*. The first one is used for storing a list of available platforms and is set by the **cl::Platform::get** method. The second variable stores the properties that will be used to create a new context. Properties is a list of `cl_context_properties` values. The end of this list is marked by the 0 value. In the example, the only information stored in this list is the selected platform. It is the first platform available. The context is created by the constructor **cl::Context**. It takes the type of devices that will be included in the context and the list of properties.

Next, the command queue is created. The command queue is the element that actually executes commands on the devices. The host code cannot directly call kernels; it has to put them in a queue. It is also possible to define multiple command queues.

2.13.2. Preparation of OpenCL Programs

The kernel is the same as in the C example; it can be seen in listing 2.83. The kernel source is stored in a variable called *vectSumSrc*. This approach has the benefit of being very simple, but after any change to the kernel code, the whole project must be recompiled.

The compilation process is shown in listing 2.84. It consists of four steps; load the sources into a list of sources, create a program, compile it and create a kernel object from the program. `cl::Program::Sources` is actually a vector of pairs: string and its length. The second step is done by the constructor **cl::Program**. It takes the context – the program will exist within this context – and the list of sources. This

```
char vectSumSrc[] =
  "kernel void vectSum ( "
  " global float *v1, "
  " global float *v2, "
  " global float *r, "
  " int n) { "
  " const int i = get_global_id(0); "
  " r[i] = v1[i] + v2[i]; "
  "} ";
```

Listing 2.83: The variable storing the source code of a **kernel** that sums two vectors of given size

```
cl::Program::Sources sources;
std::string source_code(vectSumSrc);
sources.push_back(std::make_pair(source_code.c_str(), source_code.length()));
program = cl::Program(context, sources);
try {
  program.build(contextDevices);
} catch (cl::Error &e) {
  std::cout << "Compilation build error log:" << std::endl <<
  program.getBuildInfo <CL_PROGRAM_BUILD_LOG > (contextDevices [0]) << std::endl;
  throw e;
}
vectSumKernel = cl::Kernel(program, "vectSum");
```

Listing 2.84: Compilation of OpenCL program

list in this example contains only one source. The last step is to build the program using method **cl::Program::build**. The program is in source form, so the build operation will compile it. Error checking is performed using exceptions. The only exception that should be intercepted is `cl::Error`. In case of error, the build log will be displayed and the exception will be thrown further to the top of the execution stack.

2.13.3. Using Binary OpenCL Programs

OpenCL standard allows for getting the compiled binary form of the OpenCL program and using binary program representation instead of source code. This allows for caching the program. This has the benefit of speeding up application loading time and also allows for the binary distribution of the application (but it limits portability). The part of this example that loads the binary program could also be used for applications that use embedded OpenCL implementations where the compiler is not present.

The example of loading an OpenCL program with usage of a cache is split among listings 2.85, 2.86, 2.87 and 2.88. The files saved by the code from this example are in the following format: the first 4 or 8 bytes store `size_t` number that informs about the size of the binary program data and the data itself.

```
cl::Program createProgram(cl::Context &context, std::string source_code,
                      const std::string params = "") {
// (1)
  cl_int r;
  cl::Program::Binaries binaries;
  cl::Program::Sources sources;
  cl::Program program;
  std::vector <cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES > ();
// (2)
  try {
// (3)
    binaries = loadBinaries(context, params);
// (4)
    program = cl::Program(context, devices, binaries);
    try {
      program.build(devices, NULL);
    } catch (cl::Error e) {
      std::cout << "Compilation build error (\"" << params << "\") log:" <<
      std::endl << program.getBuildInfo <CL_PROGRAM_BUILD_LOG > (devices [0]) <<
      std::endl;
      throw e;
    }
// (5)
  } catch (cl::Error &e) {
// (6)
    sources.insert(sources.end(),
      std::make_pair(source_code.c_str(), source_code.length()));
// (7)
    program = cl::Program(context, sources, &r);
    try {
      program.build(devices, params.c_str());
    } catch (cl::Error e) {
      std::cout << "Compilation build error (\"" << params << "\") log:" <<
      std::endl << program.getBuildInfo <CL_PROGRAM_BUILD_LOG > (devices [0]) <<
      std::endl;
      throw e;
    }
// (8)
    saveBinaries(context, program, params);
  }
  return program;
}
```

Listing 2.85: Program compilation with cache in C++

The main part of this example can be seen in listing 2.85. The example uses an exceptions mechanism instead of the traditional **if**-**then**-**else**. First, it tries to load a binary program from a disk using function **loadBinaries** (marked by (3)). If it goes well, the object *program* of type `cl::Program` is created and built using the loaded binaries (marked by (4)). After these steps, the binary OpenCL program is ready to use.

If there are errors, for example, if cached binary files cannot be found, then the

```
cl::Program::Binaries loadBinaries(cl::Context &context, std::string params =
                                   "") {
  cl::Program::Binaries binaries;
  std::vector <cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES > ();
  std::string bin_file_name;
  std::vector<cl::Device>::iterator d;
  char *bin_data = NULL;

  for (d = devices.begin(); d != devices.end(); ++d) {
    bin_file_name = generateFileName(*d, params);
    std::ifstream bin_file(bin_file_name.c_str(), std::ios::in | std::ios::binary);
    if (bin_file.is_open()) {
      size_t bin_size;
      char new_line;
      bin_file.read((char*)&bin_size, sizeof(bin_size));
      bin_file.read((char*)&new_line, sizeof(new_line));
      bin_data = new char [bin_size];
      bin_file.read(bin_data, bin_size);
      binaries.insert(binaries.begin(), std::make_pair(bin_data, bin_size));
      bin_file.close();
    } else {
      throw cl::Error(-999, "binariesNotAvailable");
      break;
    }
  }
  return binaries;
}
```

Listing 2.86: Loading of binary program representation

exception is generated. This exception is caught in the block marked by (5). The program is then loaded from sources (the line marked by (6)) given in the string *source_code*. Then it is compiled and built using method **cl::Program::build**. If compilation is successful, then the binaries are written to disk using the function **saveBinaries** (line marked by (8)). The functions **loadBinaries** and **saveBinaries** are in listings 2.86 and 2.87.

Note that even though the binary program can be loaded, it still has to be built. This allows for OpenCL implementation to perform some preprocessing – for example, dynamic linkage to some libraries.

The loading of a binary program can be seen in listing 2.86. It is wrapped into the function named **loadBinaries**. It first gets the list of the devices in a context. This is done using method **cl::Context::getInfo** parametrized by CL_CONTEXT_DEVICES. This is necessary because OpenCL allows for compilation to specified devices. In the example, the compilation is performed on all available devices, but it could be compiled for only one device as well. Next, the function **loadBinaries** iterates through all devices, and for each, it tries to open the file with the name generated by the function **generateFileName**. This function can be seen in listing 2.88 and it just generates a file name based on the name of the device. After the file name is generated, the application checks for the existence of this file. If it exists, then it is loaded and put into the program binaries object. OpenCL provides the

```
1  void saveBinaries(cl::Context &context, cl::Program &program,
2                    std::string params = "") {
3  // (1)
4    std::string bin_file_name;
5    std::vector <cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES > ();
6    std::vector<size_t> bin_sizes;
7    std::vector<char *> bin_binaries;
8    std::vector<cl_device_id> bin_devices;
9  // (2)
10   bin_devices = program.getInfo <CL_PROGRAM_DEVICES > ();
11   bin_sizes = program.getInfo <CL_PROGRAM_BINARY_SIZES > ();
12 // (3)
13   for (::size_t i = 0; i < bin_sizes.size(); i++) {
14     bin_binaries.insert(bin_binaries.end(), new char [bin_sizes [i]]);
15   }
16 // (4)
17   program.getInfo(CL_PROGRAM_BINARIES, &bin_binaries);
18 // (5)
19   for (::size_t i = 0; i < devices.size(); i++) {
20     bin_file_name = generateFileName(devices [i], params);
21     std::ofstream bin_file(bin_file_name.c_str(), std::ios::out | std::ios::binary);
22 // (6)
23     if (bin_file.is_open()) {
24       char new_line = '\n';
25       bin_file.write((char*)&bin_sizes [i], sizeof(size_t));
26       bin_file.write((char*)&new_line, sizeof(new_line));
27       bin_file.write(bin_binaries [i], bin_sizes [i]);
28       bin_file.close();
29     } else {
30       std::cout << "Error writing binary program !" << std::endl;
31       exit(-2);
32     }
33   }
34 // (7)
35   for (::size_t i = 0; i < bin_sizes.size(); i++) {
36     delete bin_binaries [i];
37   }
38 }
```

Listing 2.87: Saving binary program representation

`cl::Program::Binaries` class that represents a binary program. If all needed data are available, it returns a prepared `cl::Program::Binaries` object. If the files do not exist or any other problem occurs, it throws an exception of the type `cl::Error`.

The function for saving a binary program representation can be seen in listing 2.87. It presents the function **saveBinaries** that gets the binary program representation from the context and saves it into the files – one for each device the program was compiled for.

First, the function **saveBinaries** from listing 2.87 takes the list of devices the program was compiled for (lines marked by (2)). This is done using method **cl::Program::getInfo** typed with CL_PROGRAM_DEVICES. It also has to retrieve

```
std::string generateFileName(cl::Device &device, std::string params) {
  std::string bin_file_name;
  std::string vendor_name, device_name, app_ver = __TIME__ " " __DATE__;
  device_name = device.getInfo<CL_DEVICE_NAME > ();
  vendor_name = device.getInfo<CL_DEVICE_VENDOR > ();
  bin_file_name = "compiled_" + app_ver + " " + vendor_name + " " + device_name +
                  params + ".bin";
  for (::size_t j = 0; j < bin_file_name.length(); j++) {
    if (!(((bin_file_name [j] >= 'a') && (bin_file_name [j] <= 'z')) ||
          ((bin_file_name [j] >= 'A') && (bin_file_name [j] <= 'Z')) ||
          ((bin_file_name [j] >= '0') && (bin_file_name [j] <= '9')) ||
          (bin_file_name [j] == '.'))) {
      bin_file_name [j] = '_';
    }
  }
  return bin_file_name;
}
```

Listing 2.88: Function generating file name for binary program representation

the sizes of the binary data. It is done using the same method but typed with CL_PROGRAM_BINARY_SIZES. The next step is to allocate memory for every binary program data in the loop marked by (3). The binaries are obtained using method **cl::Program::getInfo** with the parameter CL_PROGRAM_BINARIES. This method fills the vector of the char array with binary program representations. Note that OpenCL implementation can return this data in any format, so it can be, for example, Assembler code, x86 executable binary or some other data. The only information provided by OpenCL is the size of the data and the data itself.

The actual saving of the data is performed in the **for** loop marked by (5). It saves binary files for each device. The file name is generated using the function **generateFileName** shown in listing 2.88 – the same function that is used in **loadBinaries**.

The full source code of this example is available on the attached disk. Please note that the intended use of this method is to optimize program loading times, so using it as a means of code obfuscation is very risky. The different versions of hardware, or even different versions of drivers on the same hardware, can use different program representations.

2.13.4. Computation

The computation is the part of the program where the work is done. Most applications follow the schema: input, compute, output. The example in listing 2.89 is no different. The input is simulated by usage of statically defined vectors, and its values are called *vector1* and *vector2*. The result is stored in the vector (array) *result*. The vector length is really small, only 3 elements, but it is sufficient for this example. First, the input data is displayed.

The input vectors should be put into context. There also should be a place in the context to store results. Constructor **cl::Buffer** creates a buffer in a given context.

The buffers *vector1_dev* and *vector2_dev* are initialized with values copied from host memory vectors *vector1* and *vector2*. The buffer *result_dev* is allocated only in the device memory. It will store the results of computation.

The kernel **vectSumKernel** takes four parameters. They are set using method **cl::Kernel::setArg**.

The next step is to enqueue the kernel **vectSumKernel**. It will be executed using the default local size and the global size set to the size of the vector, so every work-item will be busy. Note that this is not exactly the same as executing a function on the device. The command queue will decide the time of the actual kernel execution. In almost every case, it will be executed as soon as enqueued. The method **cl::CommandQueue::flush** can be used to notify the command queue that it should start execution now. In most cases, this is not necessary.

The last step is to gather and display results. The method **cl::CommandQueue::enqueueReadBuffer** orders the command queue to get the data from the buffer (that is in context) and write it to the given host array (buffer). In the example, it is also a synchronization point because of CL_TRUE set as a second parameter for **enqueueReadBuffer**. The result is displayed using standard output.

2.13.5. Release of Resources

In C++ the release of resources is done on the function exit. This is because destructor methods in OpenCL objects call the API functions to release resources. The release of resources would be necessary in the case of using dynamically allocated memory and pointers to OpenCL objects.

```
cl_float vector1 [3] = { 1, 2, 3.1 }; // sample input data
cl_float vector2 [3] = { 1.5, 0.1, -1 };
cl_float result [3];
cl_int vectorSize = 3; // size of input data
cl::Buffer vector1_dev, vector2_dev, result_dev; // memory on the device

std::cout << "vector1 = (" << vector1 [0] << "," << vector1 [1] << "," <<
vector1 [2] << ")" << std::endl;
std::cout << "vector2 = (" << vector2 [0] << "," << vector2 [1] << "," <<
vector2 [2] << ")" << std::endl;

// we have to prepare memory buffer on the device
vector1_dev = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  vectorSize * sizeof(cl_float), vector1);
vector2_dev = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  vectorSize * sizeof(cl_float), vector2);
result_dev = cl::Buffer(context, CL_MEM_WRITE_ONLY, vectorSize * sizeof(cl_float),
  NULL);

// set parameters for kernel
vectSumKernel.setArg(0, vector1_dev);
vectSumKernel.setArg(1, vector2_dev);
vectSumKernel.setArg(2, result_dev);
vectSumKernel.setArg(3, vectorSize);

// add execution of kernel to command queue
queue.enqueueNDRangeKernel(vectSumKernel, cl::NullRange,
  cl::NDRange(vectorSize), cl::NullRange);
// now we have to read results and copy it to hello_txt
queue.enqueueReadBuffer(result_dev, CL_TRUE, 0, vectorSize * sizeof(cl_float),
  result);
std::cout << "result = (" << result [0] << "," << result [1] << "," <<
result [2] << ")" << std::endl;
```

Listing 2.89: Computation

2.14. The SAXPY Example

In this section a complete OpenCL host program and associated kernel for a simple linear algebra operation SAXPY is provided. SAXPY is a very common operation in numerical linear algebra algorithms, explained in section 1.1.1. For example, there are three SAXPY operations in every iteration of the CGM algorithm described in section 1.6. The main purpose of this section is to give the reader an idea of an OpenCL host program and kernel but without all detailed explanations. The relevant explanations about the structure of OpenCL host programs and programming kernels are provided in sections 2.11, 2.12 and 2.13 of Chapter 2.

The provided example contains minimal error handling and simplified selection of computational devices. The host code may seem to be too big for SAXPY, but every OpenCL-capable application must create necessary objects in order to perform computations on the device. This code does not grow as the application is extended

```
kernel void saxpy(global float *c, global float *a, constant float *x,
                  global float *b) {
  const size_t i = get_global_id(0);
  c [i] = a [i] * x [0] + b [i];
}
```

Listing 2.90: Kernel for calculating SAXPY function

to contain more kernels and features. This is something that has to be done once. It is very similar to OpenGL – there are many commands to set up the environment, but once it is done, the other elements are rather small. OpenCL is a very flexible and universal standard. This approach has many benefits. The most important is that once an application is written, it can be executed on any system that supports OpenCL. In the case of the consumer market (for example, A-class computer games or web browsers) it is a very big advantage – the target audience owns a very large variety of hardware/software platforms. The drawback is that the initialization is rather complicated.

The example contains only two files: `saxpy.cl` and `saxpy.c` or `saxpy.cpp`. The C and C++ versions of the example execute in the same way, and the output is also exactly the same. This example source code is also provided on the disk attached to this book.

2.14.1. Kernel

In order to perform some computation on an OpenCL device (for example, a GPU) there must be an OpenCL program to execute. The programmer's task is to prepare such a program. The usual way of storing the program that is executed on the device is by using a separate file with a source code written in the OpenCL C programming language. OpenCL API usually provides a compiler, so there is no need for additional preprocessing of it. The example program can be seen in listing 2.90. It contains only one kernel – a special function that is executed on multiple work-items (cores). The kernel is called `saxpy` and implements SAXPY for the vector of `float` values. The OpenCL standard does not specify file extension for OpenCL C source files, so in the book it will be `.cl`. This program is saved in a file called `saxpy.cl`. The details about the device code are described in section 2.11.

Note that this kernel does not contain any loop. This is because it is executed in as many instances as the size of the array (vector), so it needs only to know its coordinate.

2.14.2. The Example SAXPY Application – C Language

This section shortly presents and describes the host code – that is, the part of the application that manages computation and provides instructions to OpenCL devices.

This is an ordinary C code. It can be compiled using any popular compiler. The only thing that has to be done is to link the resultant binary against the OpenCL library.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif
```

Listing 2.91: The most common includes and definitions

```
cl_context context;
cl_program program;
cl_command_queue queue;
cl_kernel saxpy_k;
```

Listing 2.92: Global variables used in SAXPY example

The host code is split into listings 2.91, 2.92, 2.93, 2.94, 2.95, 2.96, 2.97, 2.98 and 2.99.

The first element is the inclusion of header files. This can be seen in listing 2.91. Note that for Apple, the header files are located in a different directory, so for fully portable code, the conditional directives have to be used.

The next important element is definition of global variables. The example application supports only one device, one context, one command queue and one kernel object. This fragment can be seen in listing 2.92.

As mentioned before – the device program is stored in the source form, so before any computation can be executed it has to be compiled. The loading and compilation process can be seen in listing 2.93. This function loads the whole file into an array and creates a program object that represents the compiled program in the OpenCL device. More details on this stage are described in section 2.11.

The computation is meaningless without any output. The sample application prints vectors to a standard output, using the function shown in listing 2.94.

The example application must prepare vectors for computation. The stage of data preparation is present in almost any application. In the example, it is very simplified. This is only to mark the place where the actual data input should be present. The function that initializes vectors can be seen in listing 2.95.

An application using OpenCL must create context, program, command queue and kernels in order to perform any computation. In order not to favor any hardware or software vendor, the selection of OpenCL implementation and device must be performed by hand. The context object identifies the computation environment. Every action takes place within the context. The context encapsulates devices, memory objects and command queues that are essential for computation. The host code that initializes the necessary elements can be seen in listing 2.96. The initialization of these elements is described in sections 2.8, 2.5 and 2.3.

```
cl_program createProgram(cl_context context, char *fname) {
  cl_int r;
  size_t size = 0;
  size_t partsize = 128;
  cl_program program;
  char *appsource = ( char * )malloc(128);

  FILE *f = fopen(fname, "r");
  if (f == NULL) {
    exit(EXIT_FAILURE);
  }
  while (partsize == 128) {
    partsize = fread(appsource + size, 1, 128, f);
    size += partsize;
    appsource = ( char* )realloc(appsource, 128 + size);
  }
  fclose(f);

  program = clCreateProgramWithSource(context, 1, ( const char ** )&appsource,
    &size, &r);
  if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS) {
    printf("Error: Could not build OpenCL program (%d)\n", r);
    exit(EXIT_FAILURE);
  }
  free(appsource);
  return program;
}
```

Listing 2.93: OpenCL program compilation

```
void printVector(cl_float *a, int size) {
  int i;
  printf("[ ");
  for (i = 0; i < size; i++) printf("%.0f ", a [i]);
  printf("]\n");
}
```

Listing 2.94: The function that prints a vector to standard output

```
cl_float *loadVector(int size) {
  int i;
  cl_float *a;
  a = (cl_float *)malloc(sizeof(cl_float) * size);;
  for (i = 0; i < size; i++) a [i] = (8 - i) / 4 + 1;
  return a;
}
```

Listing 2.95: The function for vector initialization

```
void initialize() {
  cl_device_id devices [1];
  cl_platform_id platforms [1];
  cl_context_properties cps [3] = { 0, 0, 0 };
  if (clGetPlatformIDs(1, platforms, NULL) != CL_SUCCESS) {
    printf("Error: Could not get platform IDs\n");
    exit(EXIT_FAILURE);
  }
  cps [0] = CL_CONTEXT_PLATFORM;
  cps [1] = (cl_context_properties)(platforms [0]);
  context = clCreateContextFromType(cps, CL_DEVICE_TYPE_ALL, NULL, NULL, NULL);
  program = createProgram(context, "saxpy.cl");
  saxpy_k = clCreateKernel(program, "saxpy", NULL);

  clGetContextInfo(context, CL_CONTEXT_DEVICES, sizeof(cl_device_id), devices,
    NULL);
  queue = clCreateCommandQueue(context, devices [0], 0, NULL);
}
```

Listing 2.96: The function that encapsulates all initialization steps

```
void release() {
  clReleaseKernel(saxpy_k);
  clReleaseProgram(program);
  clReleaseCommandQueue(queue);
  clReleaseContext(context);
}
```

Listing 2.97: Release of resources

```
void saxpy(cl_mem c, const cl_mem a, const cl_mem x, const cl_mem b, int size) {
  size_t global_size[] = { size };
  clSetKernelArg(saxpy_k, 0, sizeof(c), &c);
  clSetKernelArg(saxpy_k, 1, sizeof(a), &a);
  clSetKernelArg(saxpy_k, 2, sizeof(x), &x);
  clSetKernelArg(saxpy_k, 3, sizeof(b), &b);
  clEnqueueNDRangeKernel(queue, saxpy_k, 1, NULL, global_size, NULL, 0, NULL, NULL);
}
```

Listing 2.98: Enqueue (run) the kernel

When using C implementation of OpenCL standard, the resources need to be freed as in any C program. The freeing resources can be seen in listing 2.97.

The kernel in OpenCL is run through a command queue. The host program must put the kernel on the queue for execution. In this example, the command queue executes consecutive commands (not only kernels). The execution of kernels is described in more detail in sections 2.8 and 2.9.

The main function of the SAXPY example can be seen in listing 2.99. The first

```
1  int main(int argc, char **argv) {
2    setbuf(stdout, NULL);
3    initialize();
4
5    size_t size = 32;
6    cl_float *a, *b, *c, *x;
7
8    c = (cl_float *)malloc(sizeof(cl_float) * size);
9    a = loadVector(size);
10   x = loadVector(1);
11   b = loadVector(size);
12
13   printVector(a, size); printVector(x, 1); printVector(b, size);
14
15   cl_mem a_dev, x_dev, b_dev, c_dev;
16   a_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
17     sizeof(cl_float) * size, a, NULL);
18   b_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
19     sizeof(cl_float) * size, b, NULL);
20   c_dev = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
21     sizeof(cl_float) * size, c, NULL);
22   x_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
23     sizeof(cl_float), x, NULL);
24   saxpy(c_dev, a_dev, x_dev, b_dev, size);
25
26   clEnqueueReadBuffer(queue, c_dev, CL_TRUE, 0, sizeof(cl_float) * size,
27     c, 0, NULL, NULL);
28   printVector(c, size);
29
30   clReleaseMemObject(a_dev); clReleaseMemObject(b_dev);
31   clReleaseMemObject(c_dev); clReleaseMemObject(x_dev);
32   free(a); free(b); free(c); free(x);
33   release();
34   return EXIT_SUCCESS;
35 }
```

Listing 2.99: The main function

steps prepare the execution environment (**initialize**); after that, the global variables are set. Next it allocates and loads vectors. Then the input is displayed. The next lines are specific to OpenCL applications – the memory buffers. The variables *x_dev*, *a_dev*, *b_dev* and *c_dev* of type `cl_mem` hold information about memory buffers allocated in a context. These are kernel-accessible memory regions. **clCreateBuffer** creates these buffers and initializes it, using values stored in arrays (vectors) *a*, *b*, *c* and *x*. More information concerning memory can be seen in section 2.10. Next step is to execute the **saxpy** defined in listing 2.98 – this is where the actual computation is enqueued. The function **clEnqueueReadBuffer** is used to download computation results into the array located in the host address space (variable called *c*). The result is printed using **printVector** defined in listing 2.94. The last step that can be seen in lines 30 . . . 34 of listing 2.99 is to free the memory and release resources.

```
#include <iostream>
#include <cstdlib>
#include <fstream>

#define __CL_ENABLE_EXCEPTIONS 1
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif
```

Listing 2.100: The most common includes and definitions

```
cl::Context context;
cl::Program program;
cl::CommandQueue queue;
cl::Kernel saxpy_k;
```

Listing 2.101: Global variables used in SAXPY example

2.14.3. The example SAXPY application – C++ language

Analogically to the C version, the C++ code also needs to include header files. As in the C version, the directory names for these files are different for Apple and for other platforms. This problem is also solved using conditional compilation using preprocessor directives. The includes are in listing 2.100. In this listing, there is also one very useful declaration – `__CL_ENABLE_EXCEPTIONS`. This makes OpenCL C++ API generate a specific object which handles the exception in the event of error. Thanks to this and despite the fact that there is no explicit error handling in the code, the user will be notified if anything goes wrong.

The global variables used by this example are of a different type than that found in the C version. These are wrapper objects for OpenCL elements like context, command queue and so on. These wrappers provide convenient methods for manipulating these objects. The declaration of global objects necessary to run sample computations on the device can be seen in listing 2.101. The details about classes in this fragment are in sections 2.11, 2.8, 2.6 and 2.5.

Loading and compilation of an OpenCL program can be seen in listing 2.102. The OpenCL device code is loaded from a file named `saxpy.cl`, using very convenient C++ API that allows for a load of the whole file into a string using only two lines of code. Next, this string is passed to the constructor of the type `cl::Source`, and then the source is passed to the constructor variable *`cl::Program`*. The program is then built. The resultant object representing the compiled OpenCL program is returned. More information about program creation can be seen in section 2.11.

The function for printing a vector to a console can be seen in listing 2.103. It does the same thing as the analogous function in the C version of this program.

```
cl::Program createProgram(cl::Context context, std::string fname) {
  cl::Program::Sources sources;
  cl::Program program;
  std::vector <cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES > ();
  std::ifstream source_file(fname.c_str());
  std::string source_code(std::istreambuf_iterator <char> (
                        source_file), (std::istreambuf_iterator <char> ()));
  sources.push_back(std::make_pair(source_code.c_str(), source_code.length()));
  program = cl::Program(context, sources);
  program.build(devices, NULL);
  return program;
}
```

Listing 2.102: OpenCL program compilation

```
void printVector(cl_float *a, int size) {
  int i;
  std::cout << "[ ";
  for (i = 0; i < size; i++) std::cout << a [i] << " ";
  std::cout << "]" << std::endl;
}
```

Listing 2.103: The function that prints a vector to standard output

```
cl_float *loadVector(int size) {
  int i;
  cl_float *a;
  a = new cl_float [size];
  for (i = 0; i < size; i++) a [i] = (8 - i) / 4 + 1;
  return a;
}
```

Listing 2.104: The function for vector initialization

The C++ version also initializes the vectors using an arithmetic sequence of values. The initialization function can be seen in listing 2.104. The vectors are also represented by an array of `float` values allocated using the **`new`** operator.

Initialization of OpenCL can be seen in listing 2.105. It gets the list of available platforms. The first one available is selected, and then the context is created on it. The context uses all of the available devices. This is a very simple way of initializing and does not guarantee efficiency (it can select a CPU, for example). Context creation is described in more detail in section 2.5.

The program is created using the function shown in listing 2.102. Then the kernel object is extracted from the program – this object is used for executing the kernel. The last step is to create a command queue. The command queue is essential for executing any commands on an OpenCL device. Every action (like execution of kernels,

```
void initialize() {
  std::vector < cl::Platform > platforms;
  cl::Platform::get(&platforms);
  cl_context_properties cps [3] =
  { CL_CONTEXT_PLATFORM, ( cl_context_properties )(platforms [0])(), 0 };
  context = cl::Context(CL_DEVICE_TYPE_ALL, cps);
  program = createProgram(context, "saxpy.cl");
  saxpy_k = cl::Kernel(program, "saxpy");
  queue = cl::CommandQueue(context, context.getInfo<CL_CONTEXT_DEVICES > () [0]);
}
```

Listing 2.105: The function that encapsulates all initialization steps

```
void release() {
}
```

Listing 2.106: Release of resources – in C++ it is empty

```
void saxpy(cl::Buffer c, const cl::Buffer a, const cl::Buffer x,
          const cl::Buffer b,
          int size) {
  saxpy_k.setArg(0, c);
  saxpy_k.setArg(1, a);
  saxpy_k.setArg(2, x);
  saxpy_k.setArg(3, b);
  queue.enqueueNDRangeKernel(saxpy_k, cl::NullRange, cl::NDRange(
      size), cl::NullRange);
}
```

Listing 2.107: Enqueue (run) the kernel

memory transfers and so on) is executed by the device via the command queue. The contexts and devices are more widely described in section 2.3 and 2.4.

In C++, the release of an OpenCL object is done on the exit of the last function that uses it, so the function in listing 2.106 is empty. It is left here only for comparison with the C version of the SAXPY application.

Execution of a kernel is shown in listing 2.107. This function sets parameters for the kernel using the **`cl::Kernel::setArg`** method. The kernel is enqueued using the method **`cl::CommandQueue::enqueueNDRangeKernel`**. Execution of the kernel is described in sections 2.8, 2.9 and 2.11.

The main function can be seen in listing 2.108. This function first initializes OpenCL objects using the function **`initialize`** shown in listing 2.105. Next, it allocates and initializes vectors using the function **`loadVector`** shown in listing 2.104. The buffers – *`x_dev`*, *`a_dev`*, *`b_dev`* and *`c_dev`* – are allocated using the constructor `cl::Buffer`. The class `cl::Buffer` encapsulates `cl_mem`. The values stored in the

```
int main(int argc, char **argv) {
  initialize();
  size_t size = 32;
  cl_float *a, *b, *c, *x;
  c = new cl_float [size];
  a = loadVector(size);
  x = loadVector(1);
  b = loadVector(size);
  printVector(a, size); printVector(x, 1); printVector(b, size);

  cl::Buffer a_dev, x_dev, b_dev, c_dev;
  a_dev = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeof(cl_float) * size, a);
  b_dev = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeof(cl_float) * size, b);
  c_dev = cl::Buffer(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeof(cl_float) * size, c);
  x_dev = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeof(cl_float), x);
  saxpy(c_dev, a_dev, x_dev, b_dev, size);

  queue.enqueueReadBuffer(c_dev, CL_TRUE, 0, sizeof(cl_float) * size, c);
  printVector(c, size);

  delete a; delete b; delete c; delete x;
  release();
  return EXIT_SUCCESS;
}
```

Listing 2.108: The main function

OpenCL buffers are initialized using data from arrays (vectors) *a*, *b*, *c* and *x*. For details about OpenCL memory, please refer to section 2.10.

The actual computation is executed using the function **saxpy** from listing 2.107. Data is then obtained using the command queue method **cl::CommandQueue::enqueueReadBuffer** that downloads data from the buffer into the host memory. Then the results are printed, using the function **printVector**. The last step is to free the memory and release resources.

2.15. Step by Step Conversion of an Ordinary C Program to OpenCL

This section will discuss how to convert a simple sequential program for using OpenCL technology. The example is based on standard SAXPY, because this is one of the most widely used operations and it is complex enough for demonstration. In this section, only the C programming language will be used.

```
#include <stdio.h>
#include <stdlib.h>

void saxpy(float *z, float a, float *x, float *y, int n) {
  int i;
  for (i = 0; i < n; i++)
    z [i] = a * x [i] + y [i];
}
void initializeData(float *a, float *x, float *y, int n) {
  // some initialization code goes here
}
void printResults(float *x, int n) {
  // output results
}

int main(){
  float *x, *y, *z; // vectors
  float a; // scalar
  int n; // vector size
  n = 512;
  x = (float*)malloc(sizeof(float) * n);
  y = (float*)malloc(sizeof(float) * n);
  z = (float*)malloc(sizeof(float) * n);
  initializeData(&a, x, y, n);

  saxpy(z, a, x, y, n);

  printResults(z, n);
  return EXIT_SUCCESS;
}
```

Listing 2.109: Simple sequential SAXPY program.

2.15.1. Sequential Version

The sequential version of a SAXPY example can be seen in listing 2.109. This example initializes three vectors of size 512 and a scalar. It uses some data initialization function, which is not essential for purposes of the example (it can, for example, load data from file). All of the computation is done in the function **saxpy**.

There is a loop which iterates the same number of times as the size of the vectors. Every iteration is independent of the previous ones; this gives a great opportunity to parallelize this fragment of code. The appropriate kernel function must be created, plus all of the initialization code.

2.15.2. OpenCL Initialization

First of all, the header files for OpenCL have to be included. This part of the host code can be seen in listing 2.110.

The variable types should also be changed to OpenCL types. This should be done because on some systems, the format of data stored in the host memory can be different than the format used in OpenCL. On x86 machines, the OpenCL types

```
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif
```

Listing 2.110: OpenCL header files.

```
cl_float *x, *y, *z; // vectors
cl_float a; // scalar
cl_int n; // vector size
```

Listing 2.111: Variable types.

```
cl_int ret; // this variable is for storing error codes
cl_context context; // computation context
cl_command_queue queue; // command queue
```

Listing 2.112: Global variables.

are usually the same as on the host CPU. The result of this operation can be seen in listing 2.111.

The next important thing is to create global variables holding the most important OpenCL objects. The code performing this operation can be seen in listing 2.112.

The first one, *ret*, is usually underestimated. It will be used to check the status codes of invoked API calls. It is a good practice to check it always – or at least every time something unexpected can happen. This will allow finding bugs in every stage of development and excluding situations which are triggered by hardware without enough resources for some tasks. In the examples, the exit codes are sometimes omitted, but this is in order to keep things clear and easy to understand.

The variable *context* is used for holding the computation context. This element of OpenCL is described in more detail in section 2.5.

The next one, *queue*, holds the command queue. Only one command queue will be created in this computation context. This object is also described in detail in section 2.8. The code in listing 2.113 shows initialization of the context and the command queue.

In development versions of OpenCL standard, these steps could be omitted; but the final version does not allow for the implementation to guess which platform to choose. In the current version, it is necessary to get the list of available OpenCL platforms. The function **clGetPlatformIDs** gets the list of them. This example needs only one.

The next step is to prepare *context_properties*. This array specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value, and the list is terminated with 0. This is needed to select the platform.

```
cl_platform_id platforms [1]; // list of available platforms
cl_uint num_platforms; // number of OpenCL platforms
cl_context_properties context_properties [3] = { 0, 0, 0 };
cl_device_id context_devices [32]; // list of devices included in context

// First we have to obtain available platforms
ret = clGetPlatformIDs((cl_uint)1, platforms, &num_platforms);
if (ret != CL_SUCCESS) {
  printf("Error!"); exit(-1);
}
// next we have to select one platform, we can choose first one available.
// Note that context_properties is a zero-ended list.
context_properties [0] = (cl_context_properties)CL_CONTEXT_PLATFORM;
context_properties [1] = (cl_context_properties)platforms [0];
context_properties [2] = (cl_context_properties)0;

// now we can initialize our computational context
context = clCreateContextFromType(
  context_properties, CL_DEVICE_TYPE_ALL,
  NULL, NULL, &ret);

// we have to obtain devices which are included in context
ret = clGetContextInfo(context,
  CL_CONTEXT_DEVICES, 32 * sizeof(cl_device_id),
  context_devices, NULL);

// next we have to prepare command queue
queue = clCreateCommandQueue(context,
  context_devices [0], 0, &ret);
```

Listing 2.113: Initialization of OpenCL.

The function **clCreateContextFromType** creates the computation context. The example application lets OpenCL decide which particular GPU device to use. This behavior is caused by the CL_DEVICE_TYPE_GPU flag. The platform is selected via *context_properties*.

Command queues are associated with devices within a context. In order to create a queue, the related devices have to be selected. The function **clGetContextInfo** gets various context parameters. In listing 2.113, it is invoked to get the list of devices in the computation context. When it is done, the command queue can be created using the function **clCreateCommandQueue**, which takes the devices list and the context.

After these steps, the computation context and command queue should be ready. These objects will be referred to very often in later stages of the example saxpy program.

2.15.3. Data Allocation on the Device

Up until now, all the data in the example application were stored in the system memory associated with the CPU. In order to perform some computation on an OpenCL

```
cl_mem x_dev;
cl_mem y_dev;
cl_mem z_dev;

x_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  n * sizeof(cl_float), x, &ret);
if (ret != CL_SUCCESS) {
  printf("Error allocating OpenCL memory buffer!\n"); exit(-1);
}
y_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  n * sizeof(cl_float), y, &ret);
// .. error checking!
z_dev = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
  n * sizeof(cl_float), x, &ret);
```

Listing 2.114: Creating memory buffers on the device.

device, the memory buffers which are accessible by it have to be created and data should be transferred into it if needed. OpenCL memory is described in section 2.10, so it will not be explained here in detail. One thing that must be mentioned here is that the host code can access only global and constant memory.

The function **clCreateBuffer** with flags set to CL_MEM_READ_ONLY and CL_MEM_COPY_HOST_PTR orders OpenCL to create a memory buffer in the device memory which is read-only for the kernel, and the host memory is copied to the newly created buffer. If the second flag were omitted, it would be necessary to copy data into the device using the function **clEnqueueWriteBuffer**. Note that this instruction is also a synchronization point between the host and device programs. The last **clCreateBuffer** function invocation with the flag CL_MEM_WRITE_ONLY creates a memory buffer in the device which will be write-only for OpenCL. In most cases, using CL_MEM_WRITE_ONLY and CL_MEM_READ_ONLY does not affect application performance, but it is advised to use it when possible. Some implementations may optimize memory access adequately.

During data allocation and transfer, it is important to check for errors. One of the very common problems is when a program tries to allocate a memory buffer bigger than the memory available on the device. This is very easy to encounter because usually the system memory is multiple times greater than the memory on the graphic card, and buffers which fit perfectly into the host memory are too big for the device memory.

2.15.4. Sequential Function to OpenCL Kernel

The usual way of changing a sequential program into kernel is by removing the outermost loop or loops. Depending on the problem dimensions, it can be one, two or three levels of loops[1]. The programmer must be aware of relations between each iteration. If they are independent (like in saxpy) the loop can be replaced by $i =$

[1] OpenCL supports computation space dimensions of a size up to 3.

```
1 kernel void saxpy(global float *z, float a, global float *x, global float *y,
2                int n) {
3   int i = get_global_id(0);
4   z [i] = a * x [i] + y [i];
5 }
```

Listing 2.115: SAXPY kernel.

```
1 FILE *f = fopen("saxpy/saxpyKernel.cl", "r");
2 if (f == NULL) exit(-1);
3 char kernelSource [1024];
4 size_t sizeOfKernelSrc;
5 sizeOfKernelSrc = fread(kernelSource, 1, 1024, f);
6 kernelSource [sizeOfKernelSrc] = 0; // !!
7 fclose(f);
```

Listing 2.116: Loading OpenCL program form file to host memory.

$get\_global\_id(0)$ (if it was iterated by i). The following listing shows the result of that operation.

The prefix kernel denotes that this function can be used as a kernel – the special type of function that is executed in parallel on the OpenCL device.

The OpenCL C variable types map to the host variable types with cl_ prefix (from OpenCL header files). These types are usually the same in the kernel and in the host code, but it is safer to use prefixed ones on the host.

The keyword **global** means that the following parameter is located in the OpenCL program's global memory. This is the usual way of passing the memory buffer to the kernel. These buffers are used for input and output.

The function **get_global_id**(0) returns the current work-item ID in the global work space along the first axis (first coordinate). Note that in OpenCL, the work-item term is very similar to the word thread used in traditional terminology. During kernel execution, the variable i is stored in the kernel's private memory. By default, every local variable in the kernel is stored that way. The actual computation is done in line 5 and is an exact copy of the formula from the sequential program.

Note that not all functions can be converted that way. In some functions, the iteration is dependent on previous iterations. One of the examples of this kind of function is the function calculating an n-th Fibonacci number. In this case, conversion is impossible, or the whole algorithm must be changed in order to fit into a massively parallel paradigm.

2.15.5. Loading and Executing a Kernel

As stated before, OpenCL programs have to be loaded from some storage device or memory. Here, OpenCL program source code is loaded from a file stored in the application directory into the system memory. Note that the loaded string is null-terminated. It will be described later.

```
1 char *kernelSource =
2   "\
3 __kernel void saxpy(__global float *z, float a, __global \
4   float *x, __global float *y, int n) {\
5   int i = get_global_id(0); \
6   z[i] = a*x[i]+y[i];\
7 }" ;
```

Listing 2.117: OpenCL program in host variable.

For SAXPY, it would also be reasonable to store an OpenCL program in some variable. The way it can be done is shown in listing 2.117.

Both approaches are correct for small programs, but for more complicated cases it is advised to load OpenCL program from a file or even compile it and store it in some application cache. Use of the binary program representation is described in section 2.13.3.

The next step is to load the program into the OpenCL memory. This is done using the function **clCreateProgramWithSource**. This is shown in listing 2.118. This function copies the program source into the device memory. The parameter *lengths* is not set, because a null-terminated string is passed as the program source code; in this case, it is not necessary to notify OpenCL about the size of it.

The usual way of loading a small OpenCL program is to compile it every time the application is started. API call **clBuildProgram** compiles the program stored in the OpenCL device into executable form[2]. The function **clCreateKernel** creates a kernel. It selects one of the available kernel functions in our program and returns the kernel object which represents the kernel in the computation device's memory. Remember that an OpenCL program can contain multiple kernels and helper functions.

From now on, the kernel code can be executed on the OpenCL device.

The kernel parameters must be set using **clSetKernelArg**. Simple data types like float and integer can be passed as arguments without any additional work (see lines 3 and 6 in listing 2.119). The memory buffers must be from the device accessible memory; this is why memory buffers had to be created in the previous steps.

The function **clEnqueueNDRangeKernel** adds kernel execution to the command queue. This function needs information about the number of dimensions and size of problem along each axis. For SAXPY,only one dimension is needed, and the size of the work is equal to the size of the vectors. This function will enqueue parallel execution of n instances of our kernel. Note that the moment when the kernel will actually be executed is chosen by the OpenCL implementation. It is, of course, possible to use synchronization mechanisms, but it is generally better to synchronize only while obtaining the final results.

[2]Note that this executable is for specific hardware – in this case probably the GPU – so it will not run on a CPU or even on a GPU from other vendor.

```
cl_program program;
cl_kernel kernel;

program = clCreateProgramWithSource(context,
  1, (const char **)(&kernelSource),
  NULL, &ret);
if (ret != CL_SUCCESS) {
  printf("Error loading source!!\n"); exit(-1);
}
ret = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (ret != CL_SUCCESS) {
  printf("Error: Failed to build program executable\n");
}
kernel = clCreateKernel(program, "saxpy", &ret);
if (ret != CL_SUCCESS) {
  printf("Error creatng kernel!!\n"); exit(-1);
}
```

Listing 2.118: Building OpenCL program.

```
// set parameters for kernel
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &z_dev);
ret = clSetKernelArg(kernel, 1, sizeof(cl_float), &a);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), &x_dev);
ret = clSetKernelArg(kernel, 3, sizeof(cl_mem), &y_dev);
ret = clSetKernelArg(kernel, 4, sizeof(int), &n);

// here we get the sizes of work groups and other such stuff
size_t global_work_size [1] = { 0 };
global_work_size [0] = n;

ret = clEnqueueNDRangeKernel(queue,
  kernel, 1, NULL,
  global_work_size, NULL, 0, NULL, NULL);
clFinish(queue);
```

Listing 2.119: SAXPY kernel execution.

```
1 ret = clEnqueueReadBuffer(queue,
2   z_dev, CL_TRUE, 0,
3   n * sizeof(cl_float), z, 0, NULL, NULL);
4 clReleaseMemObject(x_dev);
5 clReleaseMemObject(y_dev);
6 clReleaseMemObject(z_dev);
7 clReleaseKernel(kernel);
8 clReleaseProgram(program);
```

Listing 2.120: Gathering results and cleanup.

2.15.6. Gathering Results

The last step is to gather the results and release any used resources. Consider the code that can be seen in listing 2.120. The function **clEnqueueReadBuffer** commands OpenCL to add a buffer read command into a queue. This function waits for the queue execution to be finished, because the parameter *blocking_read* is set to CL_TRUE. Note that up to this function invocation, there was no knowledge about which commands had finished. The function **clFinish** could be used for performing synchronization, but for performance reasons[3] it is advised to synchronize as rarely as possible.

The functions in lines 4–6 of the code that can be seen in listing 2.120 release memory objects using the function **clReleaseMemObject**. Next, there is release of the kernel using the **clReleaseKernel** function. The last step is to release the program using the function **clReleaseProgram**. After that, the results are stored in the memory buffer *z*.

Note that this example is too simple to show any speedup compared to sequential code. This is because transferring data to an OpenCL device takes more time than computation on a CPU. If this program were just a part of some larger application, where almost all the data were stored and processed on a computation device, then the speedup would be significant.

The SAXPY function is used in the CGM example that is discussed in section 1.6, and performance issues are discussed in section 3.3. It is important to check how the kernels access the memory. Sometimes just rewriting the functions is not enough. It is most often seen in the case of execution on a GPU, where memory access, depending on the method, can be multiple times faster or multiple times slower than on the host CPU. Note that the method presented here for the C programming language is also correct for the C++ version. Performance considerations are focused in the chapter 3 of this book.

The whole parallel program is available on the CD-ROM attached to this book.

2.16. Matrix by Vector Multiplication Example

The example presented here performs the $matrix \times vector$ operation. This is one of the basic algebra operations that are present in solutions for many computational

[3] This is valid in any multithreading environment.

problems faced by engineers and scientists. Assume the vector b and square matrix A. These are defined in equations 2.2 and 2.3.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \tag{2.2}$$

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2.3}$$

The result vector is defined as in equation 2.4.

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \tag{2.4}$$

Each element of the result vector is computed in the way shown in equation 2.5.

$$\mathbf{c_i} = a_{i,1}b_1 + a_{i,2}b_2 + \cdots + a_{i,n}b_n = \sum_{k=1}^{n} a_{i,k}b_k \tag{2.5}$$

2.16.1. The Program Calculating $matrix \times vector$

The sequential algorithm calculating this equation can be implemented in the way shown in listing 2.121. Note that the matrix is stored in one continuous memory region by rows, so the matrix element with the index (i, j) is at index $i \cdot n + j$ in the memory.

This code can be easily translated into a kernel by stripping it of the outermost **`for`** loop. Every work-item is intended to compute one resultant vector element.

The example host code is a mixture of the samples presented in this part, so it is not necessary to show the whole code here. The whole experiment is available on the attached disk. The most important parts are the kernels. In order to show the impact of using different memory pools on performance, there are two kernels. The one using global memory can be seen in listing 2.122, and the one that uses the constant memory pool wherever possible can be seen in listing 2.123.

A simple approach is by removing the outermost loop and putting the **`get_global_id`** instead. This is the most straightforward implementation that can

```
void referenceMatrixVector(size_t n, cl_float *A, cl_float *b, cl_float *c) {
  size_t j, i;
  for (i = 0; i < n; i++) { // wiersze macierzy
    c [i] = 0;
    for (j = 0; j < n; j++) {
      c [i] += A [i * n + j] * b [j];
    }
  }
}
```

Listing 2.121: Sequential algorithm calculating $matrix \times vector$ product

```
kernel void matrix_vector_mul_a(global float *A, global float *b,
                                global float *c) {
  int j, i, n;
  n = get_global_size(0);
  i = get_global_id(0);

  c [i] = 0;
  for (j = 0; j < n; j++) {
    c [i] += A [i * n + j] * b [j];
  }
}
```

Listing 2.122: The simple **kernel** version that performs matrix vector multiplication using global memory

```
kernel void matrix_vector_mul_e(constant float *A, constant float *b,
                                global float *c) {
  int j;
  const int n = get_global_size(0);
  const int i = get_global_id(0);

  float t = 0;
  for (j = 0; j < n; j++) {
    t += A [i * n + j] * b [j];
  }
  c [i] = t;
}
```

Listing 2.123: Kernel code that uses constant memory pool for input parameters for matrix vector multiplication

be created almost instantly. It does not take advantage of any additional features available in OpenCL, so it is also the slowest solution.

The optimization in listing 2.123 lies in the fact that the constant memory should provide caching features. The kernel parameters that store the input matrix and the vector are constant; that allows the OpenCL implementation to properly handle reads

from these memory regions. The global memory pool is used as a results vector. Another optimization can be seen in the line 9 inside the **for** loop. Previously (see listing 2.122) it was saving intermediate results in the global results vector; now it uses the temporary private variable `t` that is copied into the results only once, at the very end of the kernel.

2.16.2. Performance

Consider the matrix × vector product operation $c = Ab$, where A is a square matrix of size n. The sequential execution time for this operation is proportional to the square of n. We have to compute n elements of c, and each of them is a dot product of a row of A and the vector b. In heterogeneous computation, elements of c are computed in parallel. If it is possible to increase the number of computationally active cores as we increase the matrix A size n, each core will be responsible for computing one element of c. This means that the parallel computation time will be linear, proportional to the computational complexity of the dot product that is n. In such a case, massive parallelism not only reduces computation time but also improves the sequential computing complexity. This is excellent news for users who need to solve large problems containing multiple matrix × vector operations – for example, CGM discussed in Chapter 1. In the above scenario, both the problem size (the amount of work) and the machine size expand.

2.16.3. Experiment

Analyzing the matrix × vector algorithm, one can notice that copying data can be a bottleneck to the whole approach. The matrix×vector sequential algorithm computes in $O(n^2)$ time, and the parallel algorithm using OpenCL computes in $O(n)$ time, but it has to be fed with data using a very slow PCI-Express bus. Moreover, the data size is $O(n^2)$ because of the square matrix, so the situation is as follows:

For a sequential algorithm: the data transfer time is 0 because it is available locally. The computation time is $O(n^2)$. For a parallel OpenCL algorithm run on a discrete GPU: the data transfer time is $O(n^2)$ because of the matrix, and the execution time is $O(n)$.

So both algorithms will execute in $O(n^2)$ with some multiplicative constant difference. This may seem like a really big drawback, but even in such an unlucky scenario the computation on an OpenCL device can be executed much faster than sequential code. Note that the data transfer problem does not affect applications that store data and results on an OpenCL device and read the results relatively rarely.

The experiment was intended to compare different kinds of devices with the sequential code. It also should prove the assumptions presented in this section. The sample application has been executed on the following devices:

- CPU – AMD Phenom II 3.1GHz
- GPU – the Tesla C2070 processor
- Sequential CPU – run on Phenom as a reference

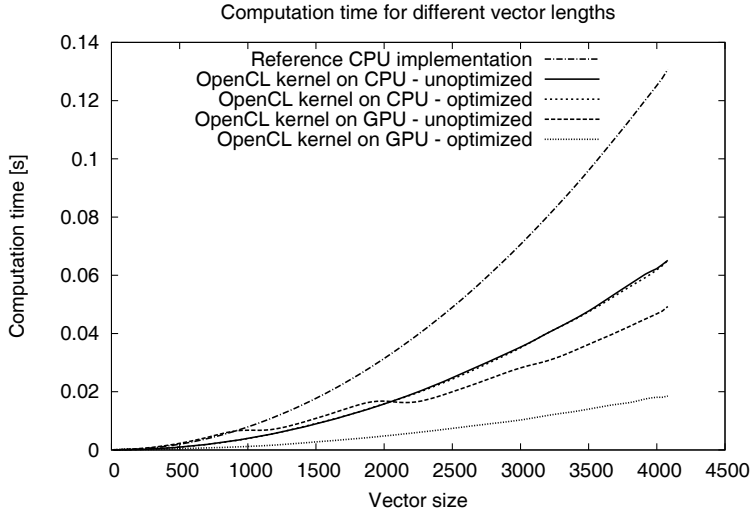

Figure 2.21: Performance comparison of different algorithms and hardware computing matrix × vector

The overall chart is depicted in figure 2.21. The results are exactly as predicted. The experiment has been performed multiple times, and the minimal times were saved in order to maximally exclude OS overhead.

The optimized version of the algorithm run on the GPU was almost 10 times faster than the sequential implementation, even though data had to be transferred to the device. The OpenCL CPU version was about two times faster than the sequential version; this is because OpenCL uses both cores of the dual-core AMD Phenom CPU.

Consider only the impact of usage of the constant memory on computation performance. The figure 2.22 shows the computation time difference between the same host code using an optimized and an unoptimized kernel. The difference is significant – about double the performance gain.

These experiments were performed using OpenCL profiling capabilities that will be described in more detail in section 3.3.4. The information from the profiler allows for deeper understanding of the program execution. There are three phases of this simple experiment – writing data to the device, executing the kernel and obtaining results from the device. Statistics for every stage have been saved, so it is possible to show which phases take the most time. The results are in figure 2.23.

In order to check if the profiler gives reliable information, the experiment also measures the time obtained from system. The differences are shown in figure 2.24. The small constant difference is probably because of the time consumed by calling the API functions.

Note that the most time was consumed by sending the matrix to the device. The execution of the kernel and getting data were so quick that in the chart, it seems to be one line.

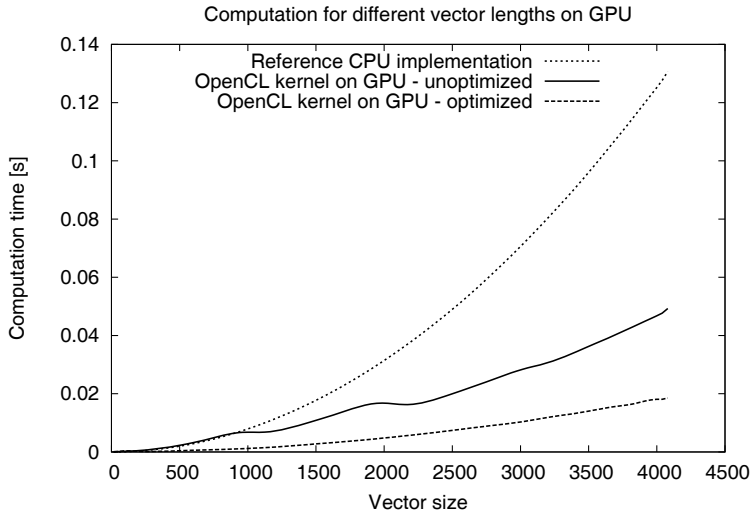

Figure 2.22: The difference between optimized and unoptimized code on GPU

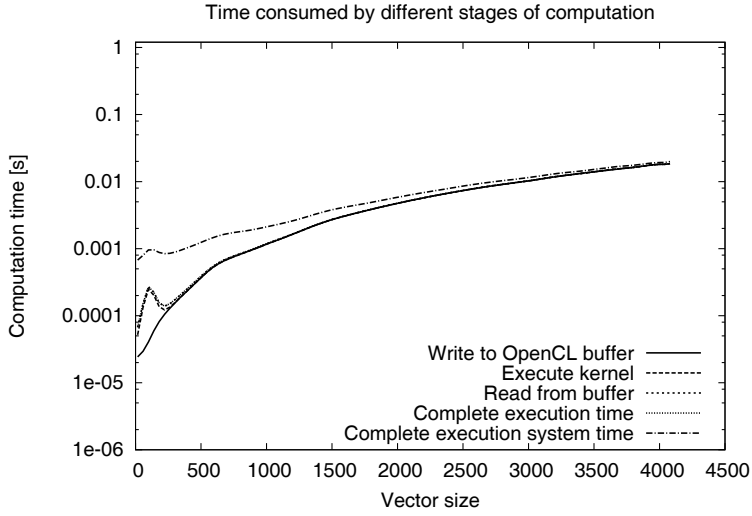

Figure 2.23: The different stages of computation – write data to buffer, execute kernel, read data from buffer

2.16.4. Conclusions

The profiler proves to be a very efficient tool. It is reliable enough to find bottlenecks in an application. The current hardware is powerful enough to provide a sig-

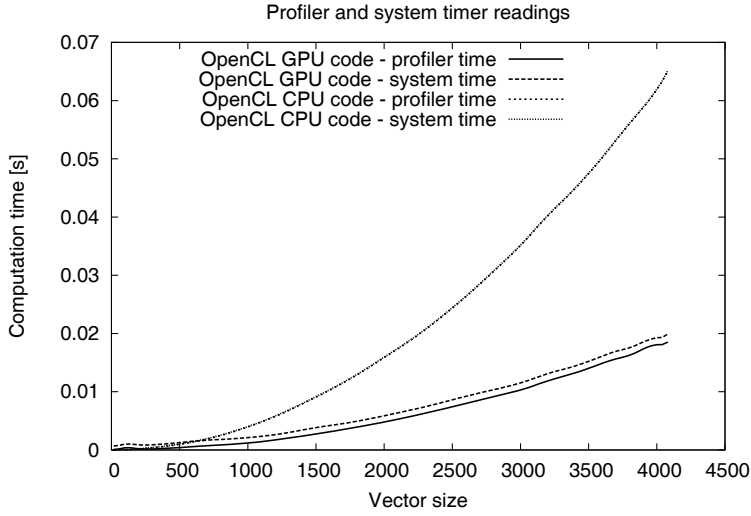

Figure 2.24: The difference between time obtained from profiler and time from system

nificant performance gain for algebra operations. A good programmer should always remember to use the proper data types and memory pools, because otherwise, the performance of the resultant code can be very poor.

Chapter 3

Advanced OpenCL

3.1. OpenCL Extensions

Pure OpenCL is very ascetic. It is possible to create very useful applications, but this standard allows for more. The extensions mechanism is the feature that unleashes the full power of OpenCL standard. The extension mechanism will be used in some parts of this chapter.

3.1.1. Different Classes of Extensions

Base OpenCL standard is intended to work on almost any computing device, so it assures the programmer that if he uses the basic functionalities, his code will work on any OpenCL implementation. The extension mechanism reveals some platform specific functionalities that are not included in standard. It is a very powerful mechanism that lets the programmer use all of the hardware features. This potentially allows faster code or more debugging options. Each extension has its name. The naming convention is that every extension contains only letters, numbers and underscore characters. The extension names are prefixed according to the origin of the extension. There are three types of extension prefixes:

1. extensions officially approved by the OpenCL Working Group have the name prefixed by `cl_khr_`, so for example `cl_khr_fp64` belongs to this class of extensions. This kind of extension can be promoted to be included in future OpenCL versions.
2. extensions provided by multiple vendors are prefixed by `cl_ext_`. For example, `cl_ext_device_fission`.
3. vendor-specific extensions. These extensions are prefixed with the shortened name of the company – `cl_(companyName)_`. The example of such an extension is `cl_nv_compiler_options`.

Extensions can provide additional functions and constants for runtime and the OpenCL program. The OpenCL platform can provide additional API functions that are not included in the original standard. Some extensions affect the OpenCL program and can be turned on in the program source code.

```
clGetPlatformInfo(platform, CL_PLATFORM_EXTENSIONS, 0, NULL, &extensions_size);
extensions = (char *)malloc(extensions_size);
clGetPlatformInfo(platform, CL_PLATFORM_EXTENSIONS, extensions_size, extensions,
  NULL);
printf("CL_PLATFORM_EXTENSIONS: %s\n", extensions);
free(extensions);
```

Listing 3.3: Getting the list of available extensions for a given platform. The C code.

3.1.2. Detecting Available Extensions from API

The OpenCL platform can provide some sets of extensions. These extensions are available for every device within this platform.

The extensions list can be obtained using API functions called **clGetPlatformInfo** and **cl::Platform::getInfo** for C and C++, respectively. The property name that has to be passed to these functions is CL_PLATFORM_EXTENSIONS. The returned list of extensions is in the form of a character string with property names written as text and separated by a space. This list can look like the one seen in listing 3.1. The application can parse it to check if some functionalities are available and adapt its internal procedures to the environment.

```
cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
```

Listing 3.1: Sample extensions list

It is possible that different devices provide different sets of additional extensions. For example some devices in the platform can support double precision arithmetics and others may not. To check what extensions are supported by a particular device, the function **clGetDeviceInfo** or **cl::Device::getInfo** can be used. The example extensions list can be seen in listing 3.2. Note that every device in the platform must provide at least the extensions listed on the platform layer.

```
cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing cl_khr_fp64
```

Listing 3.2: Sample extensions list

The example C source code for getting the list of available extensions for platform can be seen in listing 3.3. The function **clGetPlatformInfo** with the parameter CL_PLATFORM_EXTENSIONS returns the null-terminated string containing the extensions list. In the example code, the memory is dynamically allocated because the extensions list can be very long on some platforms. This code sample prints the extensions list to the console, but in most cases this list would be parsed by the application.

The example C++ source code for getting the list of available extensions for a given platform can be seen in listing 3.4. The method **cl::Platform::getInfo** with

```
1 std::cout << "CL_PLATFORM_EXTENSIONS: " <<
2 platform.getInfo < CL_PLATFORM_EXTENSIONS > () << std::endl;
```

Listing 3.4: Getting the list of available extensions for a given platform. The C++ code.

```
1 clGetDeviceInfo(device, CL_DEVICE_EXTENSIONS, 0, NULL, &extensions_size);
2 extensions = (char*)malloc(extensions_size);
3 clGetDeviceInfo(device, CL_DEVICE_EXTENSIONS, extensions_size, extensions, NULL);
4 printf("CL_DEVICE_EXTENSIONS: %s\n", extensions);
```

Listing 3.5: Getting the list of available extensions for a given device. The C code.

```
1 std::cout << "CL_DEVICE_EXTENSIONS: " <<
2 device.getInfo < CL_DEVICE_EXTENSIONS > () << std::endl;
```

Listing 3.6: Getting the list of available extensions for a given device. The C++ code.

CL_PLATFORM_EXTENSIONS returns the string that stores the list of extensions. Note that, contrary to the C version, memory is quietly allocated by the `std::string`.

Getting the list of available extensions for a particular device can be seen in listing 3.5. The list of available extensions is returned by the function **clGetDeviceInfo** with the CL_DEVICE_EXTENSIONS parameter. Memory for the output string is also allocated dynamically. This string will be at least as long as the extensions list on the platform level, because it contains every extension listed in the platform, and probably some device specific.

The C++ version of getting the list of available extensions for device can be seen in listing 3.6. This code is similar to the one getting the list of platform extensions. It uses the method **cl::Device::getInfo** template function parametrized with CL_PLATFORM_EXTENSIONS.

Getting the Extensions List as Vector – C++

The code that gets the list of available extensions can be seen in listing 3.7. It presents function that returns the `vector` of available extensions for the device *d*. Line 5 gets the string representing the list of available extensions. The variable *iss* that is declared in line 3 and set up in line 6 is used to extract individual words from the string *extensions*. Then, using the **while** loop, the individual extension names are saved back into the results `vector` *ret*.

3.1.3. Using Runtime Extension Functions

API extensions can provide additional functions and constant definitions. It can also change the behavior of some functions, but only by extending its mode of opera-

```
std::vector<std::string> getExtList(cl::Device d) {
  std::string extensions;
  std::istringstream iss;
  std::vector<std::string> ret;
  extensions = d.getInfo < CL_DEVICE_EXTENSIONS > ();
  iss.str(extensions);
  while (iss) {
    std::string extname;
    iss >> extname;
    ret.push_back(std::string(extname));
  }
  return ret;
}
```

Listing 3.7: Getting the extension list as a `std::vector`.

tion. The official extensions are listed on the Khronos website, along with detailed descriptions.

The usual way of using API extensions is as follows:

- declaration in header file of the functions and constants that are included in the extension
- checking if the extension is supported on a given platform or device
- if the extension provides some additional functions, getting the function addresses is done using the OpenCL API function **clGetExtensionFunctionAddress** for OpenCL version 1.1 or **clGetExtensionFunctionAddressForPlatform** for later versions. Note that the provided example uses deprecated API, because there are no 1.2-compliant implementations available yet (November 2011).

clGetExtensionFunctionAddress

```
void* clGetExtensionFunctionAddress ( const char *funcname );
```

Returns the address of the extension function named by funcname. This function is deprecated since OpenCL 1.2.

clGetExtensionFunctionAddressForPlatform

```
void* clGetExtensionFunctionAddressForPlatform ( cl_platform_id platform,
   const char *funcname );
```

Returns the address of the extension function named by funcname for a given platform. This function was introduced in the 1.2 API version.

Some extensions are already included in official OpenCL header files. One such extension is `cl_khr_gl_sharing`. This extension allows for cooperation between OpenGL and OpenCL. Thanks to it OpenGL can operate on the same memory regions

```
#ifndef cl_khr_gl_sharing
#define cl_khr_gl_sharing 1

typedef cl_uint cl_gl_context_info;

#define CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR -1000
#define CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR 0x2006
#define CL_DEVICES_FOR_GL_CONTEXT_KHR 0x2007
#define CL_GL_CONTEXT_KHR 0x2008
#define CL_EGL_DISPLAY_KHR 0x2009
#define CL_GLX_DISPLAY_KHR 0x200A
#define CL_WGL_HDC_KHR 0x200B
#define CL_CGL_SHAREGROUP_KHR 0x200C

typedef CL_API_ENTRY cl_int (CL_API_CALL * clGetGLContextInfoKHR_fn)(
  const cl_context_properties * properties,
  cl_gl_context_info param_name,
  size_t param_value_size,
  void * param_value,
  size_t * param_value_size_ret);
#endif
```

Listing 3.8: The declarations for `cl_khr_gl_sharing` taken from <`cl_gl.h`> (see also [10])

as OpenCL. This allows for omitting memory transfers when the results of OpenCL computations are meant to be displayed.

This extension is not included in the standard version, because some devices do not support OpenGL. For example, some devices provide only computation capabilities and therefore do not support OpenGL.

Consider the following example as a short explanation of API extensions usage. The example tries to get the function **`clGetGLContextInfoKHR`** that is part of the `cl_khr_gl_sharing` extension.

The first step is to declare functions and constants in header file. Because the example uses one of the standard extensions (the extension from `cl_khr` family), the declarations are already available in <`cl_gl.h`>. The declarations code can be seen in listing 3.8.

The example usage in the application can be seen in listings 3.9 and 3.10. This code tries to obtain the **`clGetGLContextInfoKHR`** function address in order to enable its usage later on.

The C code example can be seen in listing 3.9. The function variable is defined in this listing on the line marked by (1). The first action that has to be done is checking if the extension is available. In order to do so, the application gets the string representing the list of available extensions – it is done in the lines marked by (2). The actual checking if the `cl_khr_gl_sharing` is in the extensions list can be seen in the fragment marked by (3). If the extension is found, then the function address can be obtained. This is done in (5). Note that the function **`clGetExtensionFunctionAddress`** returns `(void *)` that is a data pointer. This is analogous to the POSIX standard function **`dlsym`**. This can raise a warning on some

```
// (1) using function definition from cl_gl.h
clGetGLContextInfoKHR_fn clGetGLContextInfo = NULL;
// (...)
// (2) check if the extension is available for given device
// get the extensions list
size_t extensions_size = 0;
cl_device_id devices [1];
clGetContextInfo(context, CL_CONTEXT_DEVICES, sizeof(cl_device_id), devices, NULL);
clGetDeviceInfo(devices [0], CL_DEVICE_EXTENSIONS, 0, NULL, &extensions_size);
char *extensions = (char*)malloc(extensions_size);
clGetDeviceInfo(devices [0], CL_DEVICE_EXTENSIONS, extensions_size, extensions,
  NULL);

// (3) check if the extension cl_khr_gl_sharing is available
char *cl_khr_gl_sharing_str = "cl_khr_gl_sharing";
int i = 0, j = 0;
do {
  if (j >= 0) {
    if (cl_khr_gl_sharing_str [j] == 0) break; // found
    if (cl_khr_gl_sharing_str [j] == extensions [i]) j++;
    else j = -1;
  }
  if (extensions [i] == ' ') j = 0;
} while (extensions [++i] != 0);
// (4) if it reached the end of the cl_khr_gl_sharing_str string
if (cl_khr_gl_sharing_str [(j >= 0) ? j : 0] == 0) {
  // (5) get the function address
  clGetGLContextInfo = (clGetGLContextInfoKHR_fn)clGetExtensionFunctionAddress(
    "clGetGLContextInfoKHR");
  printf("Extension cl_khr_gl_sharing loaded successfully\n");
} else {
  printf("Extension cl_khr_gl_sharing is not supported!\n");
}
free(extensions);
```

Listing 3.9: Obtaining extension function address – the C code

compilers, because casting the object address to the function address is not included in the standards.

The C++ code in listing 3.10 performs the same operation of getting the **clGetGLContextInfoKHR** function address. The code also declares the function pointer in the fragment marked by (1). Checking of extensions is more compact, thanks to the C++ wrapper API. The list of extensions can be obtained using only one line of code in (3). Checking if `cl_khr_gl_sharing` is present in the extensions list is done using the class `std::istringstream` that provides facilities to read the string as stream. If the extension name is found, then the function address is obtained. This operation can be seen in the lines marked by (4).

Note that in both cases, the warning can be raised by the compiler because of the ISO C or C++ incompatibility. The conversion from `(void *)` to `(void (*)())` is implementation-defined. This is correct because on the systems where OpenCL works, the pointers to data and to executable codes are the same.

```
// (1) using function definition from cl_gl.h
clGetGLContextInfoKHR_fn clGetGLContextInfo = NULL;
// (...)
// (2) check if the extension is available for given device
std::string extensions =
  context.getInfo<CL_CONTEXT_DEVICES > () [0].getInfo < CL_DEVICE_EXTENSIONS > ();
std::istringstream iss;
iss.str(extensions);
while (iss) {
  std::string extname;
  iss >> extname;
  // (3) if there is cl_khr_gl_sharing string in the list of extensions
  if (extname.compare("cl_khr_gl_sharing") == 0) {
    // (4) getting the function address
    clGetGLContextInfo = (clGetGLContextInfoKHR_fn)clGetExtensionFunctionAddress(
      "clGetGLContextInfoKHR");
    std::cout << "Extension cl_khr_gl_sharing loaded successfully" << std::endl;
    break;
  }
}
if (clGetGLContextInfo == NULL) {
  std::cout << "Extension cl_khr_gl_sharing is not supported!" << std::endl;
}
```

Listing 3.10: Obtaining an extension function address – the C++ code

3.1.4. Using Extensions from OpenCL Program

By default, all OpenCL compiler extensions are disabled. In order to use an extension, the program must enable it by the `#pragma OPENCL EXTENSION <extname> : enable` directive. If any extension is available, then it is also defined in the OpenCL program. The usual way to turn on an extension is by using the conditional compilation block #ifdef or #ifndef. The example of this technique can be seen in listing 3.11. The OpenCL program source listed here tries to enable the `cl_khr_fp64` extension. First it checks if the `cl_khr_fp64` is defined; if so, it enables this extension using the pragma directive: `#pragma OPENCL EXTENSION cl_khr_fp64 : enable`.
This code can be used in many programs that prefer to use floating point types with higher precision if possible. Note that in OpenCL version 1.2, `cl_khr_fp64` is provided only for backward compatibility if double floating-point precision is supported. The example that can be seen in listing 3.11 still uses this deprecated API, because type `double` is a very important feature and there are no OpenCL 1.2 implementations available yet.

Half and double precision arithmetic as well as a detailed description of the floating point standard used in OpenCL, is described in section 3.3.

```
#ifdef cl_khr_fp64
  #pragma OPENCL EXTENSION cl_khr_fp64 : enable
  #define FLOATTYPE double
#else
  #warning "The cl_khr_fp64 extension is not available, using float"
  #define FLOATTYPE float
#endif
```

Listing 3.11: Checking for an available extension from an OpenCL program source code

3.2. Debugging OpenCL codes

Debugging OpenCL programs is complicated in terms of non-portability of debugging software and the massive parallelism during execution of OpenCL programs. There are multiple solutions available. Some of them will be presented here.

3.2.1. Printf

printf

```
int printf ( constant char * restrict format, ... );
```

The printf built-in function writes output to an implementation-defined stream such as stdout under control of the string pointed to by format that specifies how subsequent arguments are converted for output.

The function **printf** became one of the standard OpenCL C functions along with introduction of OpenCL version 1.2. Even though it is now standard, there are still many implementations that are not compatible with the newest OpenCL version. This is the reason to treat the function **printf** as an extension in this section. When OpenCL 1.2 is implemented on a wide variety of hardware, this section will still be valid.

One of the useful extensions is defined on some implementations prior to OpenCL 1.2 is the function **printf**. This extension allows for printing text to the console directly from OpenCL kernels. This extension is available as `cl_intel_printf` or `cl_amd_printf` on Intel and AMD CPUs. Its usage is very similar to the debugging using the ordinary **printf** C function. This is the simplest approach for solving simple problems. This extension, however, is not currently available for GPUs because of the place of execution of the kernel. The introduction of OpenCL 1.2 standard will encourage platform vendors to implement the **printf** function in such a way that it will be possible to compile OpenCL programs that contain this function even if it will not produce any output. Probably it will even be possible to retrieve some output from future GPUs.

The **printf** is very convenient, and many programmers prefer this style of debugging. Using **printf** in an OpenCL program, however, brings some pitfalls that the programmer must be aware of. On older OpenCL implementations, before using the function **printf**, there is need of enabling the appropriate extension. The program code that performs this can be seen in listing 3.12. This code also takes into account the situation when there is no **printf** extension available; then, it just defines an empty macro that allows for flawless compilation of code, but the output does not display anything on the console.

Please note that it is really easy to generate enormous amounts of output using this extension. Imagine executing a kernel in the NDRange of size 32×32. Assume that the kernel contains one **printf** instruction. Every work-item in this situation will produce just one line of output, but for one kernel run it will be 1024 lines. It is really complicated to obtain valuable information from this amount of data. The best solution for this is to execute the kernels with very limited NDRange sizes or using conditional blocks to print debugging information only from interesting work-

```
#ifdef cl_amd_printf
#pragma OPENCL EXTENSION cl_amd_printf : enable
#endif
#ifdef cl_intel_printf
#pragma OPENCL EXTENSION cl_intel_printf : enable
#endif

#ifndef printf
#define printf(...) {}
#endif
```

Listing 3.12: The OpenCL program fragment that turns on the printf extension for AMD and Intel platforms.

```
kernel void dbg_add_v1(global float *B, global float *A) {
  uint i = get_global_id(0);
  uint j = get_global_id(1);
  uint w = get_global_size(0);
  uint h = get_global_size(1);
  printf("[%d, %d] ", i, j);
  printf("%2.1f+%2.1f=", B [i * w + j], A [i * w + j]);
  B [i * w + j] = B [i * w + j] + A [i * w + j];
  printf("%2.1f\n", B [i * w + j]);
}
```

Listing 3.13: Kernel using printf to output some information about it's internal operation. The incorrect approach.

items. It is also possible to use some automatic approach by parsing the output and presenting it in a graphical way.

Another obstacle is parallel execution of work-items. It is impossible to determine the exact order of **printf** invocation. The execution model is also the reason that using many consecutive **printf** calls to produce only one line of output is useless. Consider the situation presented in listing 3.13. If it were a sequential program, then the output would be readable, in just one line per kernel instance and informative enough. Execution of this code on a parallel machine, however, gives the output shown in listing 3.14 for a work of size 4×4. This is because of the race between different work-items.

There are methods to cope with this problem. One is by printing debugging information in one **printf** call per output line. This is the simpliest solution and works in most cases. The result can be seen in listing 3.16 for work size 4×4. The kernel that produces this output is shown in listing 3.15. Note that the order of execution of the **printf** command is nondeterministic, so the order of output can vary.

The last approach that allows for output that is always in the same order is by using a loop and synchronization. This method simulates serialized execution of the code. Note that this works only for kernels run in one workgroup, because there is

```
(...)
[2, 3] adding 2.000000 + -1.000000 = 1.000000
[3, 3] adding 4.000000 + -2.000000 = adding -4.000000 + 2.000000 = -2.000000
[2, 0] 2.000000
[0, 0] adding 4.000000 + -2.000000 = 2.000000
adding -2.000000 + 1.000000 = -1.000000
[3, 0] adding -0.000000 + 0.000000 = 0.000000
(...)
```

Listing 3.14: The fragment of output for the kernel shown in listing 3.13.

```
kernel void dbg_add_v2(global float *B, global float *A) {
  uint i = get_global_id(0);
  uint j = get_global_id(1);
  uint w = get_global_size(0);
  uint h = get_global_size(1);
  printf("[%d, %d] %2.1f+%2.1f=%2.1f\n", i, j,
    B [i * w + j],
    A [i * w + j],
    B [i * w + j] + A [i * w + j]
    );
  B [i * w + j] += A [i * w + j];
}
```

Listing 3.15: Kernel using printf to output some information about its internal operation.

```
(...)
[3, 2] 0.0+2.0=2.0
[3, 0] 2.0+0.0=2.0
[0, 2] 1.0+2.0=3.0
(...)
```

Listing 3.16: The fragment of output for the kernel shown in listing 3.15.

no officially supported method to synchronize different work-groups. This kernel is shown in listing 3.17, and the example output can be seen in listing 3.18.

3.2.2. Using GDB

Some CPU implementations of OpenCL allow for usage of the GNU debugger for debugging a program. The method described here works for AMD OpenCL implementation. It works for the Linux and Windows versions of GDB along with code generated by GCC or minGW (as of third quarter of 2011).

The first step to be done is to append debugging options for the OpenCL compiler. These options must be appended to the options list passed to the function **clBuildProgram**. It can be done by editing the source code of the application or by

```
kernel void dbg_add_v3(global float *B, global float *A) {
  uint i = get_global_id(0);
  uint j = get_global_id(1);
  uint w = get_global_size(0);
  uint h = get_global_size(1);
  uint k, l;
  for (l = 0; l < h; l++) {
    for (k = 0; k < w; k++) {
      barrier(CLK_GLOBAL_MEM_FENCE);
      if ((i == l) && (j == k)) {
        printf("[%d, %d] %2.1f+%2.1f=%2.1f ", i, j,
          B [i * w + j], A [i * w + j],
          B [i * w + j] + A [i * w + j]
          );
        if (k == w - 1) {
          printf("\n", i, j,
            B [i * w + j], A [i * w + j],
            B [i * w + j] + A [i * w + j]
            );
        }
      }
    }
  }
  B [i * w + j] += A [i * w + j];
}
```

Listing 3.17: Kernel using printf to output some information about its internal operation.

```
[0, 0] 0.0+0.0=0.0 [0, 1] 2.0+1.0=3.0 [0, 2] 1.0+2.0=3.0 [0, 3] 0.0+0.0=0.0
[1, 0] 2.0+1.0=3.0 [1, 1] 2.0+2.0=4.0 [1, 2] 1.0+0.0=1.0 [1, 3] 0.0+1.0=1.0
[2, 0] 2.0+2.0=4.0 [2, 1] 1.0+0.0=1.0 [2, 2] 1.0+1.0=2.0 [2, 3] 0.0+2.0=2.0
[3, 0] 2.0+0.0=2.0 [3, 1] 1.0+1.0=2.0 [3, 2] 0.0+2.0=2.0 [3, 3] 0.0+0.0=0.0
```

Listing 3.18: The output for the kernel shown in listing 3.17.

setting the environment variable CPU_COMPILER_OPTIONS in the way shown in listing 3.19. The “-g” activates inclusion of debugging symbols into the binary program representation, and the “-O0” disables optimization of code.

```
export AMD_OCL_BUILD_OPTIONS_APPEND = "-g -O0"
```

Listing 3.19: Setting the parameters to be appended to the options list for compilation of OpenCL program using command line.

The next step is to order the OpenCL implementation to use only one thread. This is also set using an environmental variable, CPU_MAX_COMPUTE_UNITS, and setting it to 1. The command to achieve this is shown in listing 3.20.

```
export CPU_MAX_COMPUTE_UNITS = 1
```

Listing 3.20: Setting the number of threads used by AMD OpenCL implementation.

After these preparations, the application that uses OpenCL can be launched in GDB in the same way as any other application. One important difference is that kernels are not visible for GDB before they are compiled. This problem is easy to solve by setting a breakpoint to some instruction that is executed after the kernels are compiled. A good choice is the function **clEnqueueNDRangeKernel**. To set a breakpoint on this function, one can use command **b** or **break** from GDB. Then, when the application is launched, the GDB will break execution on the **clEnqueueNDRangeKernel** function, and the real OpenCL code debugging can be done.

The kernel function names are constructed in the following way: if the name of the kernel is **dbg_add**, then the function for this kernel is named **__OpenCL_dbg_add**. Then, setting the breakpoint inside the kernel function is done using command **b**. This command takes the function name and sets the breakpoint on it. An example command invoked from GDB that sets the breakpoint in this way can be seen in listing 3.21.

```
b __OpenCL_dbg_add
```

Listing 3.21: Command that sets the breakpoint on kernel dbg_add.

There is a useful GDB command that shows the list of functions that match some regular expression. It is **info functions**. To show the list of available kernels, one can use the GDB command shown in listing 3.22.

```
info functions __OpenCL
```

Listing 3.22: Getting the list of functions matching a given regular expression.

Having this amount of knowledge, it is possible to use GDB to debug OpenCL applications. The example session is shown in listings 3.23, 3.24, 3.23 and 3.23. The first two lines are for setting up the environment variables that are needed for OpenCL implementation to cooperate properly with GDB.

```
 $ export CPU_MAX_COMPUTE_UNITS=1
 $ export AMD_OCL_BUILD_OPTIONS_APPEND="-g -O0"
 $ gdb ./dbg-std
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
           (...)
```

Listing 3.23: Sample session with GDB and OpenCL application – setting the environment and loading the application.

The second part of the example session with a debugger is shown in listing 3.24. It shows how to set the breakpoint to the function that enqueues the kernel execution.

```
Reading symbols from /(..)/gdb/dbg-std...done.
(gdb) info functions __OpenCL
All functions matching regular expression "__OpenCL":
(gdb) b clEnqueueNDRangeKernel
Breakpoint 1 at 0x8049cb0
```

Listing 3.24: Sample session with GDB and OpenCL application – setting first breakpoint.

Listing 3.25 shows how to run an application in a debugger environment and how to obtain the kernel names. Note that there are two invocations of the debugger command to show kernels – the first one fails to find any symbols, while the second succeeds. This is because the second command was issued after the breakpoint on **clEnqueueNDRangeKernel**. Note that in the moment of queuing, the kernel must be already compiled and available for the debugger.

```
(gdb) run
Starting program: /(...)/gdb/dbg-std
[Thread debugging using libthread_db enabled]
Device 0
CL_DEVICE_NAME: AMD Phenom(tm) II X2 550 Processor
CL_DEVICE_VENDOR: AuthenticAMD
[New Thread 0xb41aeb70 (LWP 15563)]
[New Thread 0xb416db70 (LWP 15564)]

Breakpoint 1, 0xb7fc59c0 in clEnqueueNDRangeKernel () from /(...)/libOpenCL.so.1
(gdb) info functions __OpenCL
All functions matching regular expression "__OpenCL":

File OCLe5RoDo.cl:
void __OpenCL_dbg_add_kernel(float *, float *);
Non-debugging symbols:
0xb7d1135c __OpenCL_dbg_add_kernel@plt
0xb7d11430 __OpenCL_dbg_add_stub
(gdb) b __OpenCL_dbg_add_kernel
Breakpoint 2 at 0xb7d113a0: file OCLe5RoDo.cl, line 4.
```

Listing 3.25: Sample session with gdb and OpenCL application – running the application and listing kernels.

The last part of the example session can be seen in listing 3.26 and it shows breaks on a kernel. The first break is on the kernel instance with index $(0, 0)$. The next break is on the work-item with index $(0, 1)$. It is good to check the coordinates of currently analyzed work-item, especially after **barrier**.

```
(gdb) c
Continuing.
[Switching to Thread 0xb416db70 (LWP 15564)]

Breakpoint 2, __OpenCL_dbg_add_kernel (B=0x805b000, A=0x8058000) at OCLe5RoDo.cl:4
4 uint i = get_global_id(0);
(gdb) p get_global_id(0)
$1 = 0
(gdb) p get_global_id(1)
$2 = 0
(gdb) c
Continuing.

Breakpoint 2, __OpenCL_dbg_add_kernel (B=0x805b000, A=0x8058000) at OCLe5RoDo.cl:4
4 uint i = get_global_id(0);
(gdb) p get_global_id(0)
$1 = 1
(gdb) p get_global_id(1)
$2 = 0
(gdb)
```

Listing 3.26: Sample session with GDB and OpenCL application – break on first and second kernel instance.

If the breakpoint is needed in some particular work-item, then it can be set using the command shown in listing 3.27, where the first parameter can be a line number, or a function name including kernel function, and the *N* is a dimension number and can be 0, 1 or 2. Note that it is also possible to put some more sophisticated conditional expression in the place presented one.

```
b function if (get_global_id(N) == ID)
```

Listing 3.27: Setting the breakpoint on a particular work-item ID.

The example usage of this method is shown in listing 3.28. Note that there is only one particular kernel selected from the two dimensional NDRange. Then the data from the matrix is displayed.

```
(gdb) run
Starting program: /(...)/gdb/dbg-std
[Thread debugging using libthread_db enabled]
Device 0
CL_DEVICE_NAME: AMD Phenom(tm) II X2 550 Processor
CL_DEVICE_VENDOR: AuthenticAMD
[New Thread 0xb41aeb70 (LWP 5542)]
[New Thread 0xb416db70 (LWP 5543)]

Breakpoint 1, 0xb7fc59c0 in clEnqueueNDRangeKernel () from /(...)/libOpenCL.so.1
(gdb) b __OpenCL_dbg_add_kernel if ((get_global_id(0)==3) && (get_global_id(1)==2))
Breakpoint 2 at 0xb7d113a0: file OCLpuYQII.cl, line 4.
(gdb) c
Continuing.
```

```
Breakpoint 2, __OpenCL_dbg_add_kernel (B=0x8477000, A=0x827a000) at OCLpuYQII.cl:4
4 uint i = get_global_id(0);
(gdb) step
5 uint j = get_global_id(1);
(gdb)
6 uint w = get_global_size(0);
(gdb)
7 uint h = get_global_size(1);
(gdb)
8 B[i*w+j] = B[i*w+j] + A[i*w+j];
(gdb) printf "[%d,%d] %f = %f + %f\n", i, j, B[i*w+j], B[i*w+j], A[i*w+j]
[3,2] 0.000000 = 0.000000 + 2.000000
```

Listing 3.28: Using a break on a particular kernel instance and displaying the private items and data from the global memory.

3.3. Performance and Double Precision

The OpenCL standard was developed in order to allow programmers to use an unified method of accessing highly parallel and diverse hardware. This should allow applications to compute complex problems in shorter time. This section presents some issues concerning performance of OpenCL applications.

3.3.1. Floating Point Arithmetics

The OpenCL standard supports `float`, a 32-bit floating point type compliant with the IEEE 754 standard, and `half`, a 16-bit floating point type that conforms to the IEEE 754-2008 standard. `double` is not supported in a standard, but as an extension. If it is enabled, then this type conforms to the IEEE-754 double precision storage format. For more information on this standard please refer to [16] and [17].

Note that OpenCL standard always supports a `float` type. This is a single precision floating point. The early generations of hardware that supported OpenCL were not capable of computing floating point numbers of higher precision, so the base standard also does not demand floating point types of higher resolution. The type `double` can be used by enabling one of the official extensions – `cl_khr_fp64`. Most currently available hardware supports this extension.

The arithmetic operations performed on the OpenCL device do not produce exceptions. The host program's responsibility is to check if the returned values are correct and inform about errors. For example, a divide by zero with integer types does not cause an exception but will result in an unspecified value, the same operation applied for floating-point types will result in infinity of NaN as prescribed by the IEEE-754 standard. The results of floating point operations that would raise exception must match the results defined in the IEEE standard. The absence of the exceptions mechanism in OpenCL is due to the lack of an efficient way for current hardware to implement this functionality. There is, of course, the possibility that some OpenCL platforms can support arithmetic exceptions using an extensions mechanism, but it

```
kernel void roundingmode(global char *result, global float *a) {
  int i = get_global_id(0);
  switch ((i % 5)) {
  case 0: result [i] = (char)a [i]; break;
  case 1: result [i] = convert_char_sat(a [i]); break;
  case 2: result [i] = convert_char_sat_rtp(a [i]); break;
  case 3: result [i] = convert_char_rtn(a [i]); break;
  case 4: result [i] = convert_char_rtz(a [i]); break;
  }
}
```

Listing 3.29: Explicit rounding during conversion of types.

must be initially disabled. If the application would like to use such an extension, then it must be explicitly turned on.

Rounding Modes

Internally, the computation on an OpenCL device can be done using higher precision and then the result can be rounded to fit into a desired type, for example `float`. The rounding mode that must be implemented on all OpenCL platforms is to round to nearest. There is the possibility of statically changing the rounding mode. There are four types of rounding modes:

_rte Round to nearest even.

_rtp Round toward positive infinity.

_rtn Round toward negative infinity.

_rtz Round to zero.

The explicit conversion can be done using the set of functions **convert_destType<_sat><_roundingMode>** where **destType** is the name of the destination type, **sat** tells that the value should be saturated and **roundingMode** is the shorthand of the rounding mode that should be used. The example usage of this functionality can be seen in listing 3.29.

The result of the example run on a CPU of this kernel for given values can be seen in listing 3.30. The example run of the same kernel, but on the GeForce GTS 250, gives the results shown in listing 3.31.

```
1 = (char)1.7;
127 = convert_char_sat(199.7);
2 = convert_char_sat_rtp(1.7);
1 = convert_char_rtn(1.7);
57 = convert_char_rtz(-199.7);
```

Listing 3.30: The example run of a conversion example on CPU OpenCL implementation.

```
1 = (char)1.7;
127 = convert_char_sat(199.7);
2 = convert_char_sat_rtp(1.7);
1 = convert_char_rtn(1.7);
-128 = convert_char_rtz(-199.7);
```

Listing 3.31: The example run of a conversion example on the GeForce GTS 250.

Note that they are slightly different. This is correct, because OpenCL standard defines that only the default – not saturated and rounded to nearest even type – conversion must be implemented on the platform. The other methods can work, but it is not obligatory.

Calculation Precision, Double and Float

The OpenCL standard allows for some degree of errors in calculation. These errors are small enough to be acceptable in numeric calculations. Every arithmetic operation and function provided by OpenCL has its accompanying maximal error value in the ULP. The ULP means the units in the last place. The definition from OpenCL API documentation is as follows:

ULP

If x is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then $ulp(x) = |b - a|$, otherwise $ulp(x)$ is the distance between the two non-equal finite floating-point numbers nearest x. Moreover, $ulp(NaN)$ is NaN. [10]

In other words, the ulp means how much the result value differs from the precise value on the last bits of the result. If the ulp is 0, it means that the result is perfectly correct. Note that any operation that involves rounding must have $ulp > 0$. The acceptable maximal values of ulp for most common functions defined in OpenCL 1.2 for single precision operation are:

0	every math function that does not involve rounding, **copysign**, **fabs**, **fmaxfmod**, **frexp**, **ilogb**, **logb**, **fmin**, **rint**, **round**, **trunc**
0.5	$x + y$, $x - y$, $x \cdot y$, **ceil**, **fdim**, **floor**, **fma**, **fract**, **ldexp**, **maxmag**, **minmag**, **modf**, **nan**, **nextafter**, **remainder**, **remquo**
2	**cbrt**, **log1p**, **rsqrt**
2.5	$\frac{1}{x}$, $\frac{x}{y}$
3	**exp**, **exp2**, **exp10**, **expm1**, **log**, **log2**, **log10**, **sqrt**
4	**acos**, **asin**, **acosh**, **asinh**, **cos**, **cosh**, **cospi**, **hypot**, **sin**, **sincos**, **sinh**, **sinpi**
5	**acospi**, **atanh**, **asinpi**, **atan**, **atanpi**, **tan**, **tanh**
6	**atan2**, **atan2pi**, **tanpi**
16	**erfc**, **erf**, **pow**, **pown**, **powr**, **tgamma**, **rootn**
8192	almost every **half_** function
inf	**mad**

For the ulp values for every defined function for both `float` and `double`, please refer to OpenCL API documentation [10].

```
void initOpenCL(cl_device_type devtype) {
  std::string buildparams("");
  context = contextFromType(devtype);

  if (context.getInfo<CL_CONTEXT_DEVICES>() [0]
      .getInfo <CL_DEVICE_DOUBLE_FP_CONFIG> () != 0) {
    available_double = 1;
    buildparams = "-DDOUBLE";
  }
  program = createProgramBin(context, "cmpfloat.cl", buildparams);
  queue = cl::CommandQueue(context, context.getInfo<CL_CONTEXT_DEVICES>() [0]);
  powerofFunct = cl::KernelFunctor(cl::Kernel(program, "powerof"), queue,
    cl::NullRange, cl::NDRange(EXPERIMENT_SIZE), cl::NullRange);
  if (available_double == 1) {
    powerofdblFunct = cl::KernelFunctor(
      cl::Kernel(program, "powerofdbl"), queue,
      cl::NullRange, cl::NDRange(EXPERIMENT_SIZE), cl::NullRange
      );
  }
}
```

Listing 3.32: Initializing OpenCL along with checking if `double` is supported.

```
kernel void powerof(global float *result, global float *a, global float *b) {
  int i = get_global_id(0);
  result [i] = pow(a [i], b [i]);
}
```

Listing 3.33: The kernel that computes the function **pow** on every element from two arrays.

3.3.2. Arithmetics Precision – Practical Approach

This section describes an example experiment that shows how big floating precision errors are in practice. This experiment tests the `float` and `double` versions of the function **pow** and prints the percentage of the results that were different on the host CPU and on OpenCL implementation. This section is intended for people who use OpenCL for solving real-life problems; in such cases, the information about calculation precision is crucial for the usability of results. The code is written in a fashion that allows for running it even on older implementations, but there is also information about the current state of the OpenCL standard.

In older versions of OpenCL – 1.0 and 1.1 – the `double` type was available via the `cl_khr_fp64` extension. Most modern implementations and hardware support this precision. The application must check if this feature is available. Previously, it was done by checking the presence of the extension name in the extensions list string before using it. It is still valid, because OpenCL exports the list of extensions that were included in the standard.

```
#ifdef DOUBLE
// this pragma is not needed in OpenCL 1.2
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
kernel void powerofdbl(global double *result, global double *a,
                       global double *b) {
  int i = get_global_id(0);
  result [i] = pow(a [i], b [i]);
}
#endif
```

Listing 3.34: The kernel that computes the function **pow** on every element from two arrays. This kernel operates on double precision variables.

The function presented in listing 3.32 is an example how to set up an OpenCL context and compilation, depending on availability of the `double` type. The variable *buildparams* declared in line 2 holds the build parameters and initially is set to empty. The OpenCL context is created using the device type passed as a parameter to the function **initOpenCL**. It uses the shared function that creates a context of a given type. This function is available on the storage attached to the book and uses the knowledge presented in section 2.5. The lines 5...9 check the availability of the `double` type. If it is available, then *buildparams* is set to define a "DOUBLE" macro, and the variable *available_double* is set to one. The condition that can be seen in line 5 of the discussed code gets the first device from the OpenCL context and using the **cl::Device::getInfo** method, checks the value of `CL_DEVICE_DOUBLE_FP_CONFIG`. This field of device configuration informs about availability of double-precision arithmetic.

Line 10 creates and compiles program from the file `cmpfloat.cl` with build parameters set to *buildparams*. How to write the function **createProgramBin** was described in section 2.12.3. The command queue for the device within the context is created in line 11. It creates an ordinary in-order command queue. Next, the object of the class `cl::KernelFunctor` is created in line 12. This is the class that provides a function-like interface for kernels. It is built on `cl::Kernel` with the set `cl::NDRange`. The functor for the kernel **powerofdbl** is created only if the `double` type is available. Checking for this condition is performed in line 14.

`cl::KernelFunctor::KernelFunctor`

```
cl::KernelFunctor::KernelFunctor ( const Kernel &kernel, const CommandQueue
   &queue, const NDRange &offset, const NDRange &global, const NDRange &local
   );
```

Constructor for class `cl::KernelFunctor` that wraps `cl::Kernel`. `cl::KernelFunctor` allows for enqueueing kernel in function-like fashion.

The initialization code creates the objects of `cl::KernelFunctor` that encapsulate the `cl::Kernel` and allow enqueuing it in function-like fashion. The `cl::KernelFunctor`s are for the kernel **powerof**, shown in listing 3.33, and the kernel **powerofdbl**, shown in listing 3.34. Note that the **powerofdbl** is available only

```
1  void experiment() {
2    cl_float resultDevice [EXPERIMENT_SIZE], resultHost [EXPERIMENT_SIZE],
3           a [EXPERIMENT_SIZE], b [EXPERIMENT_SIZE];
4    size_t errors = 0;
5
6    for (size_t i = 0; i < EXPERIMENT_SIZE; i++) {
7      a [i] = ((float)rand() - (float)rand()) / (float)rand();
8      b [i] = ((float)rand() - (float)rand()) / (float)rand();
9    }
10
11   cl::Buffer resultDevice_dev(context, 0, EXPERIMENT_SIZE * sizeof(cl_float));
12   cl::Buffer a_dev(context, CL_MEM_COPY_HOST_PTR,
13                   EXPERIMENT_SIZE * sizeof(cl_float), a);
14   cl::Buffer b_dev(context, CL_MEM_COPY_HOST_PTR,
15                   EXPERIMENT_SIZE * sizeof(cl_float), b);
16
17   powerofHost(EXPERIMENT_SIZE, resultHost, a, b);
18   powerofFunct(resultDevice_dev, a_dev, b_dev);
19   queue.enqueueReadBuffer(resultDevice_dev, CL_TRUE, 0,
20     EXPERIMENT_SIZE * sizeof(cl_float), resultDevice);
21
22   for (size_t i = 0; i < EXPERIMENT_SIZE; i++) {
23     if (resultDevice [i] != resultHost [i]) {
24       if ((!std::isnan(resultDevice [i])) || (!std::isnan(resultHost [i]))) {
25         std::cout << "float: \t";
26         std::cout << resultDevice [i] << " != " << resultHost [i];
27         std::cout << "\terr_order=";
28         std::cout <<
29         log10((fabs(resultDevice [i] - resultHost [i])) / fabs(resultHost [i]));
30         std::cout << std::endl;
31         errors++;
32       }
33     }
34   }
35   std::cout << "float: error percentage is " <<
36   (float)((float)errors / (float)EXPERIMENT_SIZE) << std::endl;
37 }
```

Listing 3.35: The function that performs the experiment of calculation of multiple random **pow** functions on host and device.

on devices that support double precision arithmetic. Also note that for OpenCL 1.2, there is no need of using #**pragma** to enable `double`. This fragment is left here only for compatibility with older OpenCL implementations.

The verification code can be seen in listing 3.35. There is also a comparison for the version of the example that uses `double`, but it is almost the same as the version for the `float` type. This code generates two arrays of random `float` numbers and executes calculation on them for host and device. Random number generation can be seen in lines 6...9. Note the method of enqueuing the kernel using `cl::KernelFunctor` in line 18. This method mimics the ordinary function execution. The verification function **powerofHost** is also called to calculate the results for

the CPU. The results are compared in lines 22 . . . 34, and the results that were different are displayed. The function also displays the overall fraction of results that were different on host and device.

```
cl_khr_fp64 available
float:  3.7644e-07 != 3.7644e-07 err_order=-7.12205
(...)
float:  0.727084 != 0.727084 err_order=-7.0863
float: error percentage is 0.178467
double:  6.78296 != 6.78296 err_order=-15.8829
(...)
double:  0.00354319 != 0.00354319 err_order=-15.9122
double: error percentage is 0.0837402
```

Listing 3.36: The output of the arithmetic precision experiment for the Tesla C2070 device.

```
cl_khr_fp64 available
float:  5.65044e+10 != 5.65044e+10 err_order=-7.13972
(...)
float:  0.485375 != 0.485375 err_order=-7.21183
float: error percentage is 0.00195312
double:  0.0034507 != 0.0034507 err_order=-15.9007
(...)
double:  1.15412 != 1.15412 err_order=-15.7158
double: error percentage is 0.0012207
```

Listing 3.37: The output of the arithmetic precision experiment for an ordinary CPU (Core i5) device.

In the CPU implementation, the fragment of results can be seen in listing 3.37; the same code; run on the Tesla C2070, can be seen in listing 3.36. To compare with some older hardware, the same application was run on the GeForce GTS 250; the results fragment can be seen in listing 3.38. Note that in every case, the errors were about 7 orders of magnitude smaller than the results for the `float` version of the experiment and more than 14 orders of magnitude for `double`. The results for the old GPU, however, had the biggest differences of all. The programmer must be aware of these inaccuracies. The information about errors of different functions and arithmetic operations is important for the engineering calculations. OpenCL provides precise information about acceptable errors, so it is possible to estimate the overall error for every calculation that uses this standard. In scientific and engineering computations, even small numerical errors may seriously impact the quality of results or slow down iterative computations.

```
cl_khr_fp64 unavailable
float:  1.38105 != 1.38105 err_order=-7.0639
(...)
float:  1.55582 != 1.55582 err_order=-7.11565
float: error percentage is 0.171875
```

Listing 3.38: The output of the arithmetic precision experiment for an old GPU (GTS 250) device.

Performance of Float and Double

The usage of a double-precision floating point type can influence the performance of the application. Tasks that are performed by graphics hardware do not need the `double` type, because they operate on images. The human eye cannot distinguish some subtle errors. In engineering, however, small errors on some stages of computation can lead to big bias in the results or even result in algorithm divergence.

In order to show this phenomenon in a practical way, some experiments have been performed using the CGM algorithm. These experiments show the relation between floating point precision and algorithm convergence, as well as the overall performance of the `double` and `float` types.

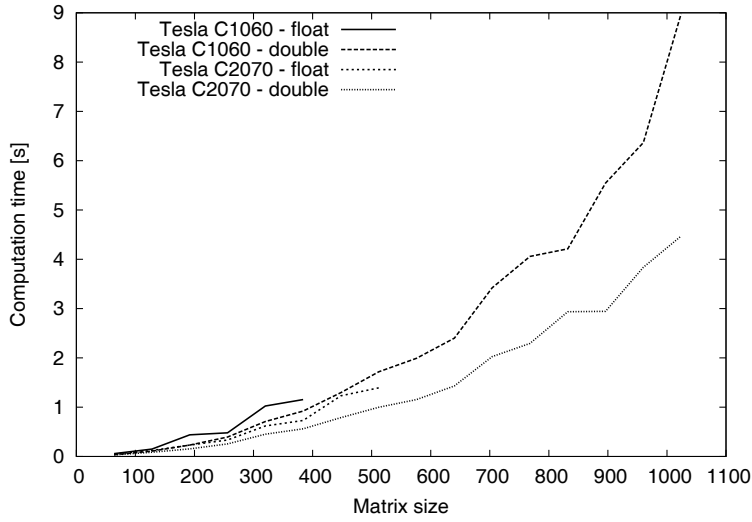

Figure 3.1: Performance comparison between two different data types – double and float – for two generations of Tesla GPUs.

The CGM algorithm was presented in section 1.6, and the application that implements this algorithm is on the attached disk. This application can be compiled to

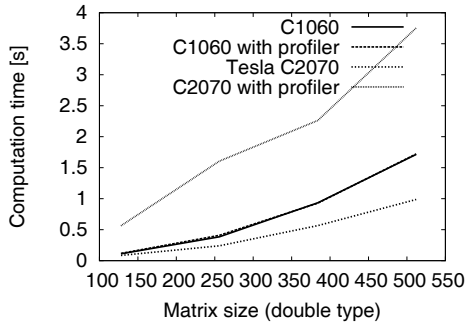

Figure 3.2: The results of the same CGM application with and without OpenCL profiling capabilities.

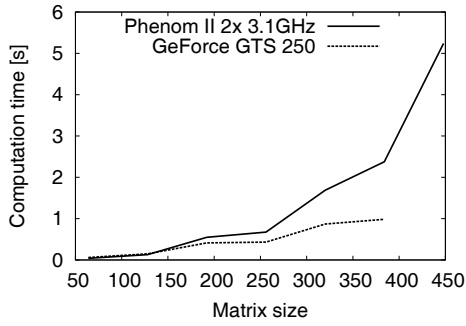

Figure 3.3: Execution times of CGM for desktop computer.

```
1 while ((alpha.get(0, 0) > 0.000001) && (i < (result.h * 10))) {
```

Listing 3.39: The CGM algorithm termination condition.

use `double` or `float`. The algorithm termination condition is shown in listing 3.39. Note that the iteration count is limited. If the method won't converge, then it will be terminated. The function *alpha*.**get(0,0)** returns the current value of α from the CGM algorithm. It enqueues reading of the hardware buffer that holds the α value. If there were only the checking for the iteration count, then there would be no synchronization points during the algorithm run at all. That would lead to even faster computation but would prevent terminating the algorithm when the desired precision is reached.

In theory, this algorithm is always convergent, but only for accurate values. Unfortunately, computers natively use approximated values for floating point operations. This is also the case for OpenCL. The standard allows, however, for use of

`float` and `double`. The section about performance is a good place to compare the convergence and times of execution of the CGM method for `float` and `double`.

Every experiment was performed on a computer with Tesla C2070 and Tesla C1060 GPUs. The CPU was Intel(R) Core(TM) i7 920 @ 2.67GHz. The installed drivers were the NVIDIA CUDA 4.0.8 on Linux Ubuntu 11.04, with every patch installed as of 2011-09-26.

Every experiment was repeated three times. On the chart the smallest values of computation time were recorded. The times were taken from the system timer, because the profiling capabilities of OpenCL were slowing down computation of C2070 to an unacceptable level – see the chart on figure 3.2.

Just for comparative purposes the same experiment was performed on a common office computer equipped with the GeForce GTS 250 and the AMD Phenom II 3.1GHz. The results are shown in figure 3.3.

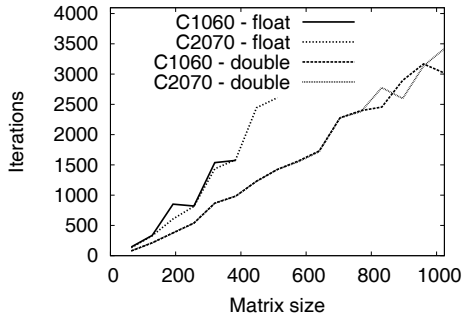

Figure 3.4: Number of iterations that had to be done in order to achieve acceptable results for different data types and for different Tesla machines.

The most interesting part of the experiment was to show how the usage of `double` impacts computation time and convergence. In figure 3.1, there is a comparative chart of the computation times for two generations of high-end GPUs and two different floating point computation precisions. Note that even though every operation on the `double` type executes slower than the same operation performed using `float`, the overall computation time was better for `double`. Thanks to better calculation precision, the algorithm executed on the `double` type was converging much faster than the same algorithm performed on `float`. Note that charts for the `float` type are shorter than the ones for `double`. This is because for that size of matrices, the `float` version of algorithm was divergent. The iteration count for different sizes of problems are shown in figure 3.4. Note that the convergence of the `double` version of the CGM algorithm highly outperforms the single precision version.

Interesting is the fact that, even though both C1060 and C2070 were using the same starting point for calculation, the execution on C2070 was convergent for bigger matrices. This is because internal computation on the C2070 is performed with higher precision.

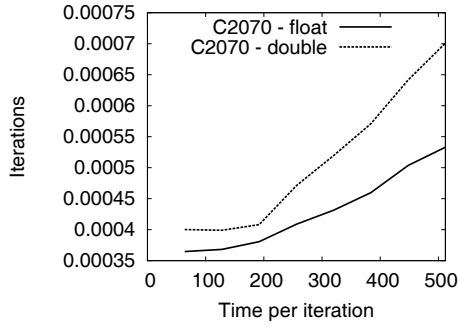

Figure 3.5: The time for one iteration for different matrix sizes.

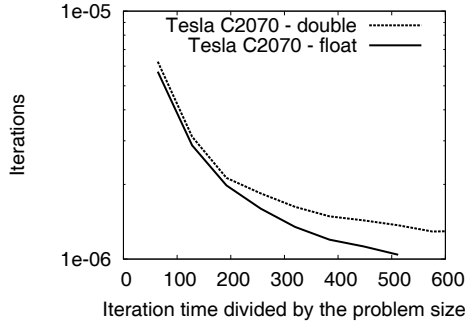

Figure 3.6: The time for one iteration divided by the matrix size.

To check how the floating point precision impacts on computation time of single step the chart 3.5 was prepared. It is clearly apparent, that computation using single precision is faster. The question is also, how does the step execution time depend on the size of the matrix? The answer is depicted in figure 3.6. Note that the execution time per matrix element decreases for bigger problems. It is a very important issue of computation in a heterogeneous environment – the bigger the problem, the better the performance gain.

3.3.3. Profiling OpenCL Application

There are two methods that allow for profiling OpenCL code – one that is done using internal profiling functionalities available in standard, and the other which is done by using external dedicated profiling application that can usually be obtained from the vendors of a given OpenCL platform implementation.

There are pros and cons for every option. The external profiler in most cases is easy to use and it allows for quite fast localization of troublesome kernels. It is the first choice for many problems that involve finding hot-spots in an application. Usage of an internal profiler is more flexible, but more difficult. The application that

is being tested must contain the profiling code. This method, however, provides very precise information on any part of the application. It is easier to skip some execution times that are not crucial for the analysis.

3.3.4. Using the Internal Profiler

The OpenCL standard defines functions and constants for profiling memory and execution operations for a given queue. This issue was initially described in section 2.8.4. Here it is described how to use this functionality efficiently.

The simplest way of using profiling is to check the execution times after command queue synchronization. This was already described and does not need any additional explanation. The other approach is to use callback functions. This method is more difficult at first, but it proves to be very efficient.

Let us consider the application that calculates CGM that was already introduced in section 1.6.2. This application is available on attached disk. The algorithm itself demands many different kernels and memory objects. This situation provides many challenges, the one that will be described and solved here is the identification of the kernel that takes the most computing time. This is the first step in optimizing any program. It is reasonable to reduce the execution time of the most inefficient functions than to optimize some minor functions that will not give any significant overall speedup.

The OpenCL program consists of the following kernels:

- matrix_mul – implements simple matrix multiplication algorithm.
- matrix_scalar_mul – the matrix $\times$ scalar multiplication algorithm.
- matrix_add – addition of two matrices.
- matrix_sub – substitution of two matrices.
- saxpy_kernel – the saxpy operation. This is shorthand for $s = ax + y$ vector operation.
- matrix_negate – this function negates the matrix, that is it multiplies every matrix element by -1.
- matrix_transpose – performs the matrix transposition.
- normal_square – calculates the square of the length of the vector.
- scalar_div – this is to divide two scalars represented as matrices. This is to avoid memory transfer from and to the OpenCL device.

The kernels are in separate file. The OpenCL program is loaded in the usual way with usage of cache for binary program representation.

The task is to find which kernel is the one that calculates most of the time during the CGM algorithm. The CGM was described in section 1.6.2. The first element that has to be done is to enable the profiling on the command queue. This is done during initialization. The code that enables it can be seen in listing 3.40 as a reminder.

Starting with OpenCL version 1.1 there is the possibility to attach a callback function to the event. The callback will be called when the event changes its state to given. The current 1.2 version of the standard, 1.2, supports only the state `CL_COMPLETE`. This is sufficient, even for very complex situations.

This is an opportunity to create a flexible event handler that will be executed only after the function that generated the event finishes. This will not block execu-

```
queue = cl::CommandQueue(context,
  context.getInfo<CL_CONTEXT_DEVICES > () [0], CL_QUEUE_PROFILING_ENABLE);
```

Listing 3.40: Initialization of command queue with profiling enabled.

```
void (CL_CALLBACK * pfn_notify)(cl_event, cl_int, void *)
```

Listing 3.41: The event handler function type.

```
class Profiler {
std::deque<ProfilerItem> profilerItems;
int lock;

static void callbackFunction (cl_event event, cl_int event_command_exec_status,
                              void *user_data);

public:
std::string name;

Profiler(std::string n = "generic");
~Profiler();
void handleEvent (cl::Event event);
double getSummaryTime ();
void reset ();
};
```

Listing 3.42: The simple class that utilizes profiling capabilities of OpenCL 1.2.

tion, so it should not slow down computation because of the many synchronization points. The idea is to collect the execution times of multiple kernels and then to calculate overall usage statistics of the device. This will provide comprehensive enough information about algorithm execution.

The event handler is a function that must comply with the type shown in listing 3.41.

This function takes the event object (the C API), the status, that the event has changed to and some user data. Note that even though current OpenCL standard provides C++ bindings, it does not define the C++ version of this callback function. This brings some difficulties that will be described and solved soon. The programmer should be also aware that multiple events can finish in the same time, so it is very likely that the callback function sometimes will operate in parallel. The standard is very clear about this: It is the application's responsibility to ensure that the callback function is thread-safe.

An example class interface that provides functionality to conveniently operate on events and set up callback function can be seen in listing 3.42.

The callback function is defined as **callbackFunction**. Note that this must be static. It would not work for ordinary instance methods. This function will be

```
1  class ProfilerItem {
2  public:
3  static const int QUEUED = 0;
4  static const int SUBMIT = 1;
5  static const int START = 2;
6  static const int END = 3;
7  cl_ulong times [4];
8
9  ProfilerItem(cl_event event);
10 cl_ulong getInterval (int first, int second);
11 };
```

Listing 3.43: The object that stores the time intervals.

called by the OpenCL implementation when the given event changes its status to CL_COMPLETE.

The object `std::deque<ProfilerItem>` *profilerItems* stores the list of event times. Every `ProfilerItem` object stores information about event times; the class that defines this object can be seen in listing 3.43. This list collects many time measures. The variable *lock* is needed because callback function can be called in parallel. The method **handleEvent** sets the callback function to be called after given the event finishes. This class also allows for setting the name of the object. This can be set to the name of the analyzed kernel.

The callback function is set by the method **handleEvent** shown in listing 3.44. The OpenCL C++ wrapper provides the method **cl::Event::setCallback** for registering callback function. The callback function will be called when `cl::Event` reaches the CL_COMPLETE state. The method **cl::Event::setCallback** can also register event callback functions for the CL_SUBMITTED, CL_RUNNING or CL_COMPLETE states. The user data passed to the callback function is the pointer to the current `Profiler` object.

cl::Event::setCallback

```
cl_int cl::Event::setCallback ( cl_int type, void (CL_CALLBACK *pfn_notify)
    (cl_event event, cl_int command_exec_status, void *user_data),
    void *user_data = NULL );
```

This method registers a user callback function for a specific command execution status. The registered callback function will be called when the execution status of command associated with event changes to the execution status specifiedby *command_exec_status*.

The method **callbackFunction** is implemented as shown in listing 3.45. This function creates `ProfilerItem`, which holds the times of the event stages. The lock mechanism on this method is as portable as possible. It just waits in an infinite loop until the variable *lock* is 0 and then it sets it to 1; perform operation and then set it back to 0. It could be done using some system specific implementation of a critical section, but it would just blur the code. Remember that when using event callbacks, it is important to ensure that the callback function is thread-safe.

```
void Profiler::handleEvent(cl::Event event) {
#ifndef DISABLEPROFILING
  void *objpointer = this;
  clRetainEvent(event()); // this is needed to work on broken implementations.
  event.setCallback(CL_COMPLETE, (void (CL_CALLBACK *)(cl_event, cl_int, void *))
    Profiler::callbackFunction, objpointer);
#endif
}
```

Listing 3.44: The method that sets up the callback function.

```
void Profiler::callbackFunction(cl_event event,
                            cl_int event_command_exec_status,
                            void *user_data) {
  if (event_command_exec_status == CL_COMPLETE) {
    Profiler *p = (Profiler*)user_data;
    while (p->lock != 0) usleep(1);
    p->lock = 1;
    p->profilerItems.push_back(ProfilerItem(event));
    p->lock = 0;
    clReleaseEvent(event);
  }
}
```

Listing 3.45: The callback function that handles the event finish.

The usage of the functions **clRetainEvent** and **clReleaseEvent** is because it prevents broken implementations from releasing a `cl_event` object before execution of a callback function. The function **clRetainEvent** increases the references number of the `cl_event` object so it will not be released too early. Then the function **clReleaseEvent** decreases the reference count. If the reference count of this object is zero, then the OpenCL implementation releases the resources occupied by this object.

clRetainEvent

```
cl_int clRetainEvent ( cl_event event );
```

Increments the event reference count.

clReleaseEvent

```
cl_int clReleaseEvent ( cl_event event );
```

Decrements the event reference count.

The constructor used in the function from listing 3.45 is presented in listing 3.46. This constructor obtains profiling information and stores it into the `ProfilerItem` object fields.

```
ProfilerItem::ProfilerItem(cl_event event) {
  clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_QUEUED,
    sizeof(cl_ulong), &times [0], NULL);
  clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_SUBMIT,
    sizeof(cl_ulong), &times [1], NULL);
  clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
    sizeof(cl_ulong), &times [2], NULL);
  clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong), &times [3], NULL);
}
```

Listing 3.46: The constructor of ProfilerItem class.

```
MatrixOCL & MatrixOCL::operator=(const MatrixOCL &orig) {
  if (this != &orig) {
    setSize(orig.h, orig.w);
    queue.enqueueCopyBuffer(orig.data_dev, data_dev, 0, 0, h * w *
      sizeof(FLOAT_TYPE), NULL, &event);
    profilers [copy_prof].handleEvent(event);
  }
  return *this;
}
```

Listing 3.47: The overloaded '=' operator for MatrixOCL that is in CGM example.

The CGM example is in the attached disk, so the whole source code will not be presented here. However, some basic information about this application is needed in order to explain the profiling method used in it. The CGM program uses a simple matrix class for calculation. There are two implementations of the matrix – `MatrixCPU` that implements basic matrix operations on host CPU, and `MatrixOCL`, which implements exactly the same functionalities but operates using OpenCL. The example is in fact a small benchmark that runs both implementations and displays the execution times. The `MatrixOCL`, however, contains a more sophisticated method of calculating execution times – the profiling functionalities included in OpenCL API.

In the OpenCL implementation in the CGM example, there are multiple `Profiler` objects, one for each kernel. This construction allows for calculation of the execution times of every kernel. There are also some for memory operations like read, write and copy.

The usage of the `Profiler` object can be seen in listing 3.47. This is the implementation of an overloaded '=' operator. It copies content of one `MatrixOCL` object into another. It uses a buffer copy operation to transfer data between matrices. Note that the method **`cl::CommandQueue::enqueueCopyBuffer`** generates a `cl::Event` object. This is the object that represents this copy operation. Then the selected profiler is ordered to set the event callback using the method shown in listing 3.44. The array *`profilers`* is defined in 3.48. The constant value *`copy_prof`* stores an index on this array. This is for convenience.

```
Profiler MatrixOCL::profilers [MatrixOCL::PROFILER_COUNT];
```

Listing 3.48: Declaration of array of Profiler objects.

```
void MatrixOCL::profilersPrint() {
  queue.finish();
  double summaryTime = 0;
  for (int i = 0; i < PROFILER_COUNT; i++) {
    std::cout << "prof: " << profilers [i].name << ": " <<
    profilers [i].getSummaryTime() << std::endl;
    summaryTime += profilers [i].getSummaryTime();
  }
  std::cout << "prof: SUMMARY_TIME: " << summaryTime << std::endl;
}
```

Listing 3.49: The method that prints the summary profiling report.

The usage of an array instead of separate `Profiler` objects is that it is possible to automatically display a summary report and operate on the whole collection of these objects at once. The method that does this is shown in listing 3.49. It just iterates through every `Profiler` object and shows the summary computation time.

The Results Explained

This CGM example was executed on a CPU and a GPU to compare how the kernel execution times differ. The exact specification of the hardware is not very important, because the results will be similar for these classes of hardware. The results for the Phenom II CPU can be seen in listing 3.51 and for the GeForce GTS 250 GPU in listing 3.50. This is the same application, but run on different hardware. The results are shown in seconds. The results from the profiler are on the lines starting with the word “prof:”.

```
   Time : 44.9375
prof: matrix_mul: 40.7348
prof: matrix_scalar_mul: 0.0286804
prof: matrix_add: 0
prof: matrix_sub: 0.0198583
prof: saxpy_kernel: 0.074825
prof: matrix_negate: 0
prof: matrix_transpose: 0.0420781
prof: normal_square: 0.0559362
prof: scalar_div: 0.037704
prof: write: 5.792e-06
prof: read: 0
prof: copy: 2.24947
prof: SUMMARY_TIME: 43.2433
```

Listing 3.50: The computation times for the CGM experiment run on GPU.

```
   Time : 45.9646
prof: matrix_mul: 15.6315
prof: matrix_scalar_mul: 0.110661
prof: matrix_add: 0
prof: matrix_sub: 0.111795
prof: saxpy_kernel: 0.274134
prof: matrix_negate: 0
prof: matrix_transpose: 0.220076
prof: normal_square: 2.31258
prof: scalar_div: 0.0038257
prof: write: 0
prof: read: 0
prof: copy: 26.3449
prof: SUMMARY_TIME: 45.0095
```

Listing 3.51: The computation times for the CGM experiment run on CPU.

In the example configuration where the GPU was used, the kernel that executed most of the time was **matrix_mul**, the kernel that performs matrix multiplication. This would be the point to start with optimization. However if the application was executed on the CPU device, the multiply operation took less time than the copy operation. This is because GPUs usually contain more efficient memory bus than CPU.

Note that the time measured by the system timer differs from the one returned from OpenCL profiling functions. This is because OpenCL registers only the time of execution of different OpenCL kernels and memory operations, but not the time that is consumed by the host code that enqueues these operations. It is also impossible to measure the time of allocation and release of OpenCL memory objects by directly using profiling methods.

To sum up: The profiler included in OpenCL can provide valuable information about the execution of an application. This information can then be used to locate the kernels that need to be optimized. In the example, the kernel that really needs some optimization is the operation of matrix multiplication. In fact, this kernel was executed for the matrix $\times$ vector because of the CGM algorithm construction. In a real-life case, the next step would be to modify the multiplication algorithm. This is shown in section 3.3.6.

Note that, depending on the hardware and drivers, it is possible that the summary execution time calculated as the sum of execution times obtained from profiling functions can be higher than the time measured by the host, as the difference between computation start and finish with synchronization at the beginning and the end. This is because a properly written OpenCL application uses the OpenCL **finish** command very rarely, so the OpenCL implementation can determine which parts of the computation can be done in parallel and then execute it in this way to utilize the maximum of resources. In this case, the host time will tell how long the execution lasts. The time from the profiler will tell how long the calculation would take if consecutive kernel executions and memory transfers were executed sequentially.

3.3.5. Using External Profiler

The external profiler is the tool that runs the OpenCL application and analyzes the execution of kernels and other elements of OpenCL that are used by the application. This is in most cases very easy to use. In this subsection, some profilers shipped with the most popular OpenCL SDKs will be described.

AMD APP sprofile

The **sprofile** is an application that is attached to the AMD APP SDK. This application allows for a command line trace of OpenCL API calls along with times of execution. This application can be very useful in automated build and check systems because it works from the command line and produces output to files that can be automatically processed. This profiler allows also for finding memory leaks, because it prints every API call that has been made by the application, in particular memory allocation and releasing.

The default compilation of the example CGM algorithm already uses the profiling capabilities of OpenCL. In order to efficiently use the external profiler, it is good to compile this example without profiling. This can be done by issuing **make** in the way shown in listing 3.52

```
$ CPPFLAGS_LOCAL="-DDISABLEPROFILING" make
```

Listing 3.52: The compilation of the CGM example without profiling API calls.

If the profiling commands were issued from inside the application being profiled, then every such call would also be listed in the external profiling program output. That would make the results hard to read because it would also contain every API call to the internal OpenCL profiler. For the CGM example, it would double the size of the output.

The **sprofile** accepts multiple command arguments. The most usable arguments are (from the documentation):

- `-o arg` Path to OutputFile (the default is `/Session1.csv` when collecting performance counters; the default is `/cltrace.atp` when performing an API trace).
- `-s arg` Character used to separate fields in the OutputFile. Ignored when performing an API trace.
- `-t` Trace OpenCL application and generate CPU and GPU time stamps and detailed API call traces.
- `-T` Generate summary page from an input .atp file.
- `-w` arg Set the working directory (the default is the app binary's path).

Note that the **sprofile** is located in the directory `tools/AMDAPPProfiler` in the AMD APP SDK directory. In this section, it is assumed that the path to the **sprofile** is configured.

The example run of the the CGM example with **sprofile** is shown in listing 3.53.

```
$ sprofile -o cgmethod-profiling.csv ./cgmethod
```

Listing 3.53: The simple execution of a CGM experiment with external profiler – sprofile.

This will produce the file in a comma separated values format that will contain the profiling information of kernel execution and memory transfers. This provides basically the same information as presented in subsection 3.3.4. This file is intended to be used in automatic processing.

Another way of running this profiler is to trace API calls. This will generate a file that will contain the list of executed API calls, along with its parameters and return values. This can provide much information about the application that is being checked. The run of **`sprofile`** that will generate this API trace is shown in listing 3.54. This file is in a format that can be easily processed by batch scripts. It provides not only the information about API calls, but timestamps as well, so it is possible to check execution times of OpenCL functions that otherwise would not be checked by the internal profiler.

```
$ sprofile -t -o cgmethod-profiling.atp ./cgmethod
```

Listing 3.54: Generating API trace with sprofile.

The produced file can later be processed to produce the report in HTML format, or this report can be generated just after the tested application exits. To achieve this, the command shown in listing 3.55 can be used. This will generate the HTML files in the same directory as the `.atp` file. The reports will contain much valuable information – for example, about the times of different kernels and the times of API calls execution. In automated build systems, these pages can even be automatically published on the web page.

```
$ sprofile -t -T -o cgmethod-profiling.atp ./cgmethod
```

Listing 3.55: Generating API trace with sprofile along with HTML reports.

The example report that shows the usage of API functions is depicted in figure 3.7. This report is for the execution of a CGM algorithm for the matrix of size 512.

NVIDIA Compute Visual Profiler

This simple tool is available in the CUDA Toolkit available for NVIDIA graphics cards. It is located in the directory `computeprof` under the CUDA installation directory. This tool provides analysis of the kernel and memory operation times. The typical usage of this tool is to find the hot spots in an application. The first step is to select the application to check. Visual Profiler greets the user with the appropriate window. This window is shown in figure 3.8 Select the button Profile application to select the

API Name	Cumulative Time(ms)	%of Total Time	# of Calls	Avg Time(ms)	Max Time(ms)	Min Time(ms)
clFinish	2402.86380	76.44115	6	400.47730	1362.86820	0.00783
clEnqueueReadBuffer	420.95924	13.39177	4	105.23981	420.12454	0.01559
clSetKernelArg	70.82908	2.25325	45098	0.00157	15.66298	0.00020
clEnqueueCopyBuffer	62.33087	1.98290	10264	0.00607	13.63144	0.00117
clEnqueueNDRangeKernel	53.76527	1.71041	10248	0.00525	7.78095	0.00142
clCreateBuffer	31.90851	1.01509	3114	0.01025	4.17198	0.00076
clRetainMemObject	20.56505	0.65423	30782	0.00067	5.59524	0.00019
clEnqueueTask	19.66797	0.62569	2048	0.00960	2.71515	0.00261
clReleaseMemObject	18.38966	0.58502	33896	0.00054	3.63961	0.00019
clReleaseEvent	16.97473	0.54001	23635	0.00072	5.65292	0.00019
clRetainEvent	11.39204	0.36241	20526	0.00055	4.77998	0.00019
[illegible]	76877	0.7	[illegible]	[illegible]	[illegible]	0.016

Figure 3.7: Example report for CGM algorithm run for matrix of size 512.

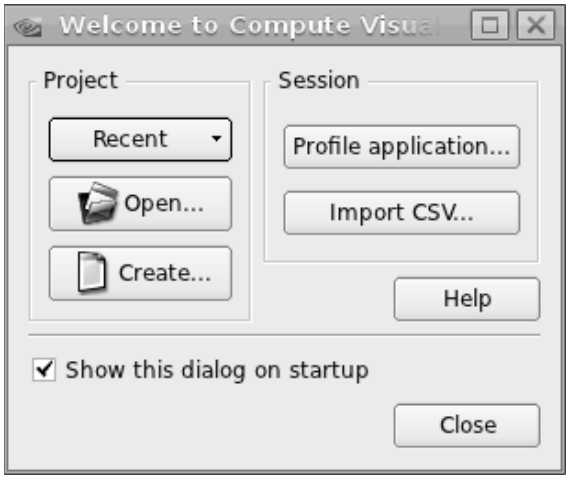

Figure 3.8: The greeting screen of the NVIDIA Compute Visual Profiler.

application that will be tested. The window shown in figure 3.9 will appear. There, it is possible to select the application to launch by selecting it in the field labeled “Launch”.

The field “Arguments” is for setting the application arguments. If the CUDA API trace is selected, then it is possible to also see the times of the API execution times, not only the times of the functions executed on the device. After clicking the “OK” button, the project is set and ready. The tested application can then be launched by clicking the appropriate button on the toolbar of Visual Profiler.

The results are presented as a table or as charts. The table view – Summary Table – allows for obtaining additional information about the kernel. The user can just double-click the kernel name to obtain additional information about it. Figure 3.10 shows the example output of the profiler and detailed information about selected kernel.

The information about throughput shown in figure 3.10 informs how close the code is to the hardware limit. Comparison of the effective bandwidth with the profiler

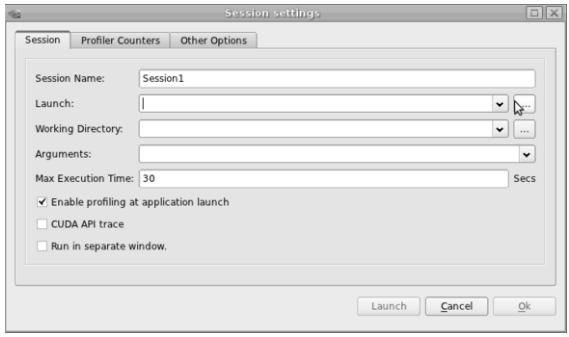

Figure 3.9: Selection of the application for profiling.

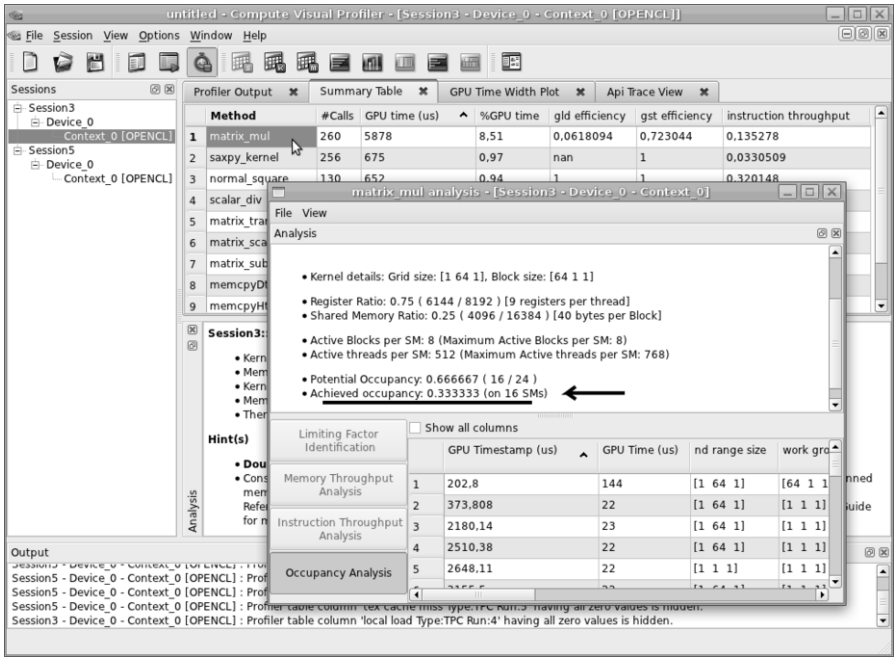

Figure 3.10: The example NVIDIA profiler output.

number presents a good estimate of how much bandwidth is wasted by suboptimal coalescing of memory accesses. In the presented figure, the theoretical bandwidth is not reached, so the kernel **`matrix_mul`** can be suspected to be ineffectively written.

This tool is very simple, but it can be the first choice for finding some efficiency problems. This is the reason for mentioning it here.

3.3.6. Effective Use of Memories – Memory Access Patterns

OpenCL portability is very convenient, but to achieve maximal performance, one has to analyze the hardware on which the application will be executed. For a CPU device, it is relatively simple. In most cases, the OpenCL program that will run on a CPU can be written by removing the outer loops – every kernel that was presented

in the Fundamentals chapter works in this way. The CPU case will not be discussed here in details because for this kind of device, the memory access patterns are not as important. In many cases, no memory access optimization is needed for CPU.

For a GPU device, the situation is not that simple anymore. OpenCL memory pools are very similar to the physical organization of the memory on the graphics card. The local memory is fast but small, and the global memory can be really slow for some access patterns.

Theoretical Memory Bandwidth for GPU

A very important limiting factor of the performance of an OpenCL program run on a GPU is memory access. The theoretical bandwidth for the GPU can be calculated using the equation 3.1.

$$d_0 = C \cdot I \cdot \frac{1}{8} \left[\frac{B}{s} \right] \tag{3.1}$$

Where d_0 is the theoretical bandwidth, C is the graphics RAM effective frequency, I is the width of the memory interface in bits. This bandwidth is expressed in $\left[\frac{B}{s}\right]$. The data for this equation are usually available in the device documentation. For example, having a graphics card that has 256-bit memory wide interface, and the memory is a DDR RAM operating at the frequency of 1100MHz, then the theoretical bandwidth would be calculated in the way shown in equation 3.2.

$$d_0 = 2 \cdot 1100 \cdot 10^6 Hz \cdot 256 \cdot \frac{1}{8} = \tag{3.2}$$

$$141.6 \cdot 10^9 \frac{B}{s} = 70,4 GBps \approx 65,6 GiBps \tag{3.3}$$

Note that the frequency is multiplied by 2. This is because DDR memory reads are performed two times per clock tick, so the effective frequency is two times higher than the memory frequency.

Effective Memory Bandwidth for GPU

The effective memory bandwidth is calculated by timing the OpenCL program. The effective bandwidth can say how close the program performance is to the theoretical performance. The perfect, but unreachable, situation is when the OpenCL program's effective and theoretical memory bandwidths are the same. Equation 3.4 shows the method of calculating effective memory bandwidth.

$$d_1 = \frac{D_r + D_w}{T} \left[\frac{B}{s} \right] \tag{3.4}$$

```
kernel void copy_coalesced(global float *B, global float *A) {
  uint i = get_global_id(0);
  B [i] = A [i];
}
```

Listing 3.56: Coalesced memory copy.

Host and Device Memory

It is important to remember that in most cases every memory transfer between host memory and device memory is very expensive. The bandwidth of PCI bus is on order of magnitude lower than the internal memory bandwidth on a GPU. This means that it is better to put data into the device once, perform the computation to the end and then retrieve the result, even if not every kernel used in computation gives a performance gain.

Coalesced Memory Access

On most of the OpenCL-capable GPUs, the global memory is aligned in segments. The segment size is dependent on hardware, but it is almost always some power of 2 and usually more than 32. Every global memory read and write operation involves reading or writing one segment. That means that even when only one work-item requests for only one byte, the transaction will involve transferring the whole segment. The hardware, however, tries to group read and write operations by segments. When many work-item requests for consecutive memory addresses, then these requests are grouped into one memory transaction. The situation when every work item accesses memory in this way is called a *coalesced memory access pattern*.

The kernel shown in listing 3.56 performs the coalesced read and write. It copies array `A` into array `B`. Every instance of this kernel will copy one memory element from `A` to `B`. Every work-item accesses data only at indexes that are the same as its global ID. In this case the hardware will be able to group memory reads into bigger chunks. This will lead to efficient memory read and write.

This is the most efficient memory access pattern on all hardware. The actual count of memory transactions is equal to $n = 2 \cdot \frac{\text{data\_size}}{\text{seg\_size}}$ where n is the number of memory transactions, `data_size` is the size of each buffer, and `seg_size` is the size of the segment. The quotient is multiplied by two because there is one read and one write operation. In this example, the effective bandwidth should be close to the theoretical memory bandwidth. This memory access pattern is depicted in figure 3.11. The small rectangles represent work-items that access the memory, shown as a grid of squares. Every square represents a word of data, and every row represents a memory segment that is read by the hardware in one transaction. The kernels access the memory along the arrows. The i-th work-item accesses the i-th consecutive memory word.

Note that even if some kernels did not read the memory in this case, it would still be a very efficient memory access pattern. Historically, this was the only access pattern where allowed for good performance; any other led to the situation that

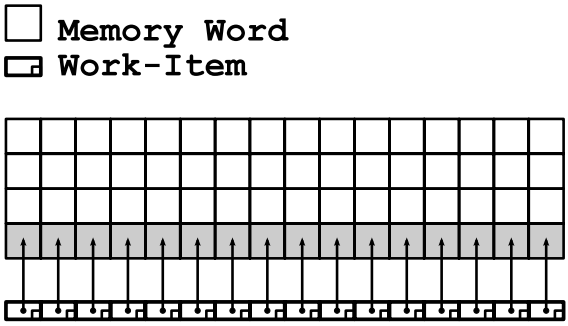

Figure 3.11: Coalesced memory access pattern.

every work-item was issuing a separate memory transaction. So, for example, 16 work-items could issue 16 transactions, even though every one of them used only one data word from a transaction.

The next access pattern that on most implementations would provide good efficiency is the one where work-items still access one memory segment, but in permuted order. This access pattern is also effective, because the hardware groups memory transfer commands. When requests are made from work-items to fetch data that is located in one segment, it will be retrieved in one transaction. An image that presents this access pattern is depicted in figure 3.12.

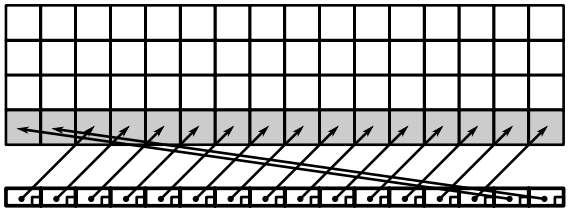

Figure 3.12: The misaligned, sequential memory access that fits into a 128 byte segment.

The kernel that executes a permuted memory access pattern is shown in listing 3.57. This time, even though this kernel will also copy array *A* into array *B* without any changes in data and order, the access pattern is different. Every work-item reads and writes memory at indexes shifted by two. There is an assumption that every memory segment is 16-words long. The operation shown in line 3 gets the segment based on the global ID. This segment is already multiplied by 16. The offset calculation is performed by the fourth line of the discussed listing. This offset is shifted by two, so it is not coalesced memory access anymore. This kernel will work efficiently on modern hardware, but on some older graphics cards it will work about 16 times slower than the kernel shown in listing 3.56, even though both are performing the same operation.

The type of access pattern explained next is also sequential and misaligned, but it touches two consecutive memory segments. This access pattern is depicted in figure 3.13. It involves two memory transactions on almost every modern GPU and 16 transactions for older hardware.

```
1 kernel void copy_permuted(global float *B, global float *A) {
2   const uint id = get_global_id(0);
3   uint i = id & 0x0fffffff0;
4   uint j = (id + 2) & 0x0f;
5   B [i + j] = A [i + j];
6 }
```

Listing 3.57: Kernel that executes misaligned sequential memory access pattern to one 128 byte segment.

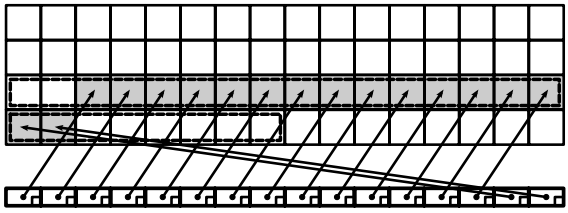

Figure 3.13: The sequential access that involves two memory transactions.

```
1 kernel void copy_misaligned(global float *B, global float *A) {
2   const uint id = get_global_id(0);
3   uint i = (id + 2) % get_global_size(0);
4   B [i] = A [i];
5 }
```

Listing 3.58: Kernel that executes misaligned memory access pattern.

The dotted rectangles mark the two memory transactions. Note that there are 8 blocks wasted in this operation. The kernel that performs this type of operation can be seen in listing 3.58. This kernel also copies array *A* into array *B*. The index of the copied item is calculated in such a way, that every work item copies elements on the index shifted by two. The formula for calculating this index can be seen in this listing at line 3.

Strided Memory Access

The last type of memory access described here is called the *strided memory access pattern*. This type of access is typical to the situation of matrix multiplication, where one matrix is accessed by rows and the other by columns. This is the situation when every work-item accesses memory cells that are separated from each other by some step. The kernel that generates the strided copy situation can be seen in listing 3.59. It copies array *A* into array *B*. This kernel calculates the size of the grid of height 256 and the proper width to fill the whole work size. This operation can be seen in lines 5...6. Global work size must be a multiplication of 256 in order for this kernel to work properly. The next operation is to calculate the x and y coordinates in this grid based on global ID. This operation can be seen in lines 7...8. Then, in line 9, the

```
1 kernel void copy_strided(global float *B, global float *A) {
2   const uint id = get_global_id(0);
3   const uint size = get_global_size(0);
4   uint i, x, y;
5   int w = size / 256;
6   int h = size / w;
7   x = id / w;
8   y = id % w;
9   i = y * h + x;
10  B [i] = A [i];
11 }
```

Listing 3.59: The kernel that executes a strided memory access pattern.

index is calculated, based on swapped x and y. The last operation is to copy values. It is done in line 10.

Figure 3.14 shows this access pattern in a graphical way. The picture shows this access for a stride of size 8. The dotted rectangles show the memory transactions. There are 16 work-items, each accessing one memory word. This operation generates 16 memory transactions of size $16 \cdot 8 = 128$ words, even though only 16 words are used by the kernel instances. This access pattern is very inefficient because every work-item issues one separate memory transaction.

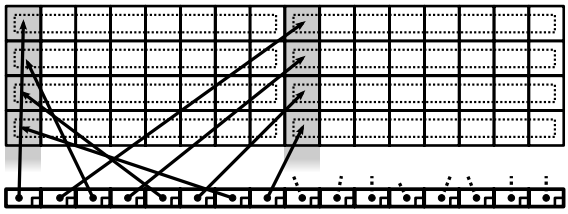

Figure 3.14: The strided copy operation.

Note that this pattern is very common – there are many algorithms that access memory in this way. One of the solutions that allow for overcoming this difficulty is to use local memory as a cache. The solution will be presented shortly.

For a relatively old graphics card – GeForce GTS 250 – the times of copying the 16MiB using the presented kernels for one test run are:

- **copy_coalesced**: 2.4696 ms
- **copy_permuted**: 20.8035 ms
- **copy_misaligned**: 20.5666 ms
- **copy_strided**: 105.7796 ms

These are the times of kernel execution measured by internal OpenCL profiling functions. Note that the coalesced copy was the fastest. The strided copy of the same amount of data took about 50 times more time. The first copy operation utilized $\frac{1024*1024*16*4*2}{0.0024696} \approx 54.3 GBps$, which is close enough to the theoretical band-

```
// (1)
kernel void matrix_mul(global FLOAT_TYPE *C,
                    const global FLOAT_TYPE *A, const global FLOAT_TYPE *B,
                    int2 sizeA, int2 sizeB
                    ) {
  // (2)
  const int x = get_global_id(0); // column
  const int y = get_global_id(1); // row
  int k;
  FLOAT_TYPE c = 0;
  // (3)
  for (k = 0; k < sizeA.x; k++) {
    // (4)
    c += A [y * sizeA.x + k] * B [k * sizeB.x + x];
  }
  C [y * sizeB.x + x] = c;
}
```

Listing 3.60: The initial version of a matrix multiplication kernel – without optimization.

width of about $70.4GBps$. However the strided copy performs at approx $1.3Gbps$. This shows how important memory access pattern optimization is.

Notes

The whole application that compares the performance of the kernels presented in this subsection is available in attached disk. On different hardware, the results obtained from this example can differ significantly, because memory access is a topic that can be solved in many different ways.

3.3.7. Matrix Multiplication – Optimization Issues

One of the common situations where strided memory access occurs is matrix multiplication. This situation is due to the fact that every work-item accesses one operand by rows and one operand by columns. The simple matrix multiplication $C = A \times B$ kernel is shown in listing 3.60. It is here as a reminder, because matrix multiplication has already been described.

In this simple approach, the work-items read the data in the way shown in figure 3.15. Assume that accessing the memory is performed in 4 word chunks. The figure shows hypothetical kernel execution for square matrices of sizes 16×16 and the work-group sizes of 4. The equation is $A \times B$. The bold rectangle marks the location of the work-group with coordinates $get\_group\_id(0) = 1$ and $get\_group\_id(1) = 2$. This is the work-group that will be discussed. The arrows show the order of work-item access to the memory in the matrix. The small numbers near arrows are the consecutive values of counter k from the kernel. Both matrices are saved by rows in the memory. The continuous memory ranges are marked by dotted rectangles. The thirteenth read operation is marked. The gray rectangles labeled by m and n mark

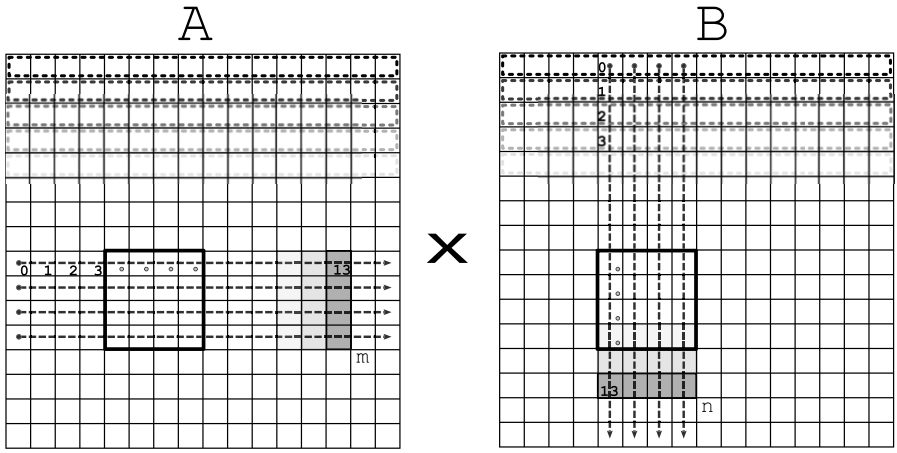

Figure 3.15: The unoptimized matrix multiplication kernel execution.

the data that is requested from the matrices when every work-item of the work-group reaches $k = 13$. Note that accessing data from matrix B realizes the coalesced memory access pattern. However, access to matrix A is strided.

Note that the memory area marked by the gray rectangles will be accessed by every work-item from the work-group. There would be every combination of words from these two regions. For example, a work-item with local id $(0,0)$ will take the first elements from both regions (the ones with the number 13), and the work-item with $(2,3)$ will take the fourth word from m rectangle and the third word from n rectangle. The dots mark the work-items that will access the cells marked by the number 13 on them from m and n.

Note that the following description is for hardware available in the year 2011. In the future, there is the possibility that this kind of operation will be optimized on the hardware level. However, it will probably be slower than the version optimized on the algorithm level, but not to the extent seen today.

In the k-th step, the kernel will perform one addition and one multiplication per work-item. Hardware will most likely execute 1 memory transaction per work-item for accessing the A matrix elements, and 1 memory transaction for every 4 work-items in a row from B. So the overall memory transaction count would be $4 \cdot 4 + 4 \cdot 4 \cdot \frac{1}{4} = 20$ memory transactions, even though there are only 8 words needed for this step of calculation. For the marked work-group, it would be $16 \cdot 20 = 320$ words. The actual number of words that need to be obtained from the global memory is only 8 for this step. Note that for some hardware, multiple accesses to the one memory block will be done in one transaction, so it will only take 5 transactions – that is, 20 words – but it is still about 3 times too many.

The solution to this problem is the usage of local memory as the cache to hold the fragment of the matrix used in the current calculation phase.

Note that the simplest approach would be to copy both matrices to local memory and then perform calculation. However, it is usually impossible, because the local memory is too small. This approach works only when the matrices are not bigger than the size of the work-group.

```
cl::size_t<3> wgs =
  matrix_mul.getWorkGroupInfo<CL_KERNEL_COMPILE_WORK_GROUP_SIZE>
    (queue.getInfo<CL_QUEUE_DEVICE > ());
cl::NDRange workgroup;
if (wgs [0] == 0) {
  workgroup = cl::NullRange;
} else {
  workgroup = cl::NDRange(wgs [0], wgs [1]);
}
```

Listing 3.61: Getting the size of a work-group size for a given kernel. The C++ example.

First Step Towards Optimized Matrix Multiplication

To start, notice that work-groups in this example form squares. It would be good to use a local memory matrix of the size of work-group. There would be $N = \frac{G}{W}$ phases where N is the number of phases, G is the global size – the size of the matrix – and W is local size.

The kernel that performs the matrix multiplication in this way can be seen in listing 3.62.

The line marked by (1) is responsible for setting attributes of the kernel. The **__attribute__**, parametrized with ((**reqd_work_group_size**(BLOCK_SIZE, BLOCK_SIZE, 1))), means that this kernel must be run in work-groups of the given size; here, it is BLOCK_SIZE×BLOCK_SIZE. In listing 3.61, there is the C++ code that allows for obtaining the local work-group size for the given kernel.

In the **matrix_mul_lx** kernel there is a local memory buffer declaration located inside the kernel body. It is on the line marked by (2) and will be used to store the temporary local matrix of elements taken from the global A matrix. Note that the line marked by (4) from listing 3.60 expands to lines (4), (5), (6) and (7) and the counter k now steps by the BLOCK_SIZE in the kernel shown in listing 3.62. The BLOCK_SIZE is the size of the work-group and the size of the local memory matrix.

Consider again the work-group with coordinates $get\_group\_id(0) = 1$ and $get\_group\_id(1) = 2$. This situation is depicted in figure 3.16. It shows the moment when $k = 12$.

It would be desirable to make 4 memory reads from global memory to fill the local matrix and then operate on the local memory. In order to achieve this, the work-items from the work-group should cooperate.

The gray triangles mark the places where the counter k changes (see the loop marked by (3) in the **matrix_mul_lx** kernel code). Note that this divides the matrix multiplication into multiple phases. The variable k now marks the beginning of the block.

The copy from global memory to local memory is performed on the line marked by (5). Note that every work-item in a work-group performs only one copy operation. The index is selected in the way that data is accessed by rows – ly and lx are both used to access the memory in the coalesced manner.

```
kernel // (1)
__attribute__((reqd_work_group_size(BLOCK_SIZE, BLOCK_SIZE, 1)))
void matrix_mul_lx(
  global FLOAT_TYPE *C,
  const global FLOAT_TYPE *A,
  const global FLOAT_TYPE *B,
  int2 a, int2 b
  ) {
  // (2)
  local FLOAT_TYPE lA [BLOCK_SIZE] [BLOCK_SIZE];
  const int lx = get_local_id(0);
  const int gx = get_group_id(0);
  const int x = get_global_id(0); // column
  const int ly = get_local_id(1);
  const int gy = get_group_id(1);
  const int y = get_global_id(1); // row
  int k, d;
  FLOAT_TYPE sum = 0;

  // (3)
  for (k = 0; k < a.x; k += BLOCK_SIZE) {
    // (4)
    barrier(CLK_LOCAL_MEM_FENCE); // crucial for correctness
    // (5)
    lA [lx] [ly] = A [a.x * (gy * BLOCK_SIZE + ly) + k + lx];
    // (6)
    barrier(CLK_LOCAL_MEM_FENCE); // crucial for correctness
    // (7)
    for (d = 0; d < BLOCK_SIZE; d++) {
      sum += lA [d] [ly] * B [b.x * (k + d) + x]; // using local memory!
    }
  }
  C [b.x * y + x] = sum;
}
```

Listing 3.62: Matrix multiplication with local memory used for caching the *A* matrix elements.

In figure 3.16, this phase for one row is marked by (5). The local memory buffer in the figure is shown as a square below the *A* matrix.

A very important element that appears in this version of matrix multiplication is the **barrier**. Memory synchronization must be used in order to ensure memory coherency. When any of the barriers is omitted, then the results of the multiplication are undefined. The first barrier ensures that all local memory reads are finished, so a work-item will not overwrite memory that is read by another work-item. The second barrier guarantees that reading from the local memory will occur only after the writes are done.

When the block fragment of *A* matrix is copied to the local memory, then the ordinary multiplication operation is performed – but this time using the local memory buffer instead of the global memory. This step is also marked on figure 3.16 by the

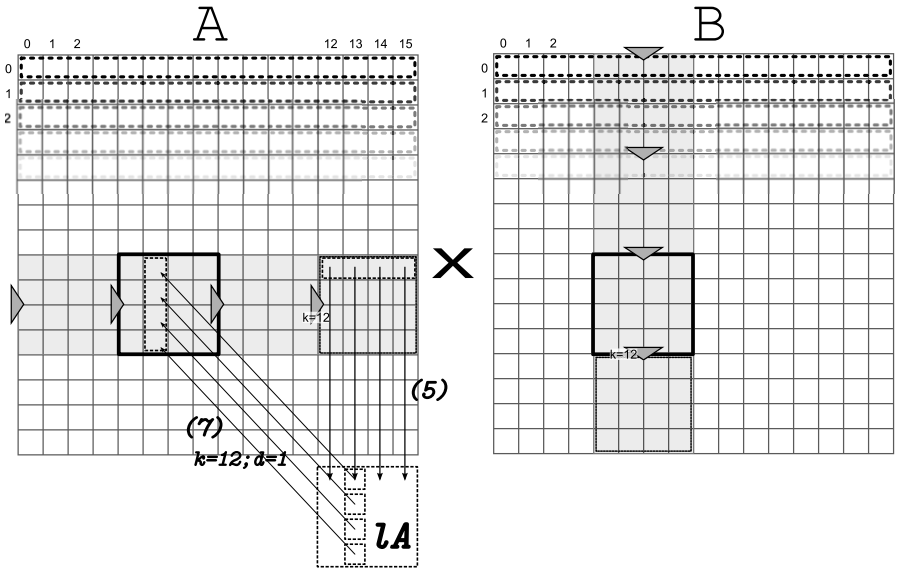

Figure 3.16: Illustration of the first step to optimizing a matrix multiplication kernel.

(7) label. In the figure, it is for $k = 12$ and $d = 1$. The local memory is efficient for both coalesced and random access.

After this optimization, exactly the same bytes will be transferred from the global *A* matrix as is needed for computation. The only thing left is to optimize the reads from the *B* matrix, because it is still possible that multiple accesses to the global memory will slow down computation. This is very similar to the method presented before, so the figurative description is not needed. The kernel that uses local memory to optimize access to both *A* and *B* matrices is shown in listing 3.63.

Result of Matrix Multiplication Optimization

The source code and the experiment are available on the attached disk. To show the possible performance gain thanks to this optimization the simple experiment was performed. The kernels that were presented in this subsection were executed for matrices of size 512 and the times for one run on idle computer with the GeForce GTS 250 – desktop graphic card – are as follows:

- **`matrix_mul`**: 0.028148 s
- **`matrix_mul_lx`**: 0.001601 s
- **`matrix_mul_ll`**: 0.000643 s

The same kernels run on the host CPU present the following results:

- **`matrix_mul`**: 0.023549 s
- **`matrix_mul_lx`**: 0.106567 s
- **`matrix_mul_ll`**: 0.108546 s

```
kernel
__attribute__((reqd_work_group_size(BLOCK_SIZE, BLOCK_SIZE, 1)))
void matrix_mul_ll(
  global FLOAT_TYPE *C,
  const global FLOAT_TYPE *A,
  const global FLOAT_TYPE *B,
  int2 a, int2 b
  ) {
  local FLOAT_TYPE lB [BLOCK_SIZE] [BLOCK_SIZE];
  local FLOAT_TYPE lA [BLOCK_SIZE] [BLOCK_SIZE];
  const int lx = get_local_id(0);
  const int gx = get_group_id(0);
  const int x = get_global_id(0); // column
  const int ly = get_local_id(1);
  const int gy = get_group_id(1);
  const int y = get_global_id(1); // row
  int t, dt;
  FLOAT_TYPE sum = 0;

  for (t = 0; t < a.x; t += BLOCK_SIZE) {
    lB [ly] [lx] = B [b.x * (t + ly) + gx * BLOCK_SIZE + lx];
    lA [lx] [ly] = A [a.x * (gy * BLOCK_SIZE + ly) + t + lx];
    barrier(CLK_LOCAL_MEM_FENCE); // crucial for correctness
    for (dt = 0; dt < BLOCK_SIZE; dt++) {
      sum += lA [dt] [ly] * lB [dt] [lx]; // using local memory!
    }
    barrier(CLK_LOCAL_MEM_FENCE); // crucial for correctness
  }
  C [b.x * y + x] = sum;
}
```

Listing 3.63: Optimized matrix multiplication kernel that uses local memory to cache both A and B matrix access.

The times were measured using OpenCL profiling functions. Remember that memory optimizations that are good for a GPU are not always good for a CPU. It is of course possible to run the same code on the GPU and on the CPU, but the performance may be different.

3.4. OpenCL and OpenGL

OpenGL is an open standard for generating computer graphics. It provides a state machine that allows for convenient graphical manipulations. For details about OpenGL, please refer to [18]. Implementations of this standard are available on most modern platforms. It is easy to notice that both OpenGL and OpenCL can operate on the same hardware. Imagine the situation when the results of OpenCL computations must be displayed on the screen. The first approach is to create separate OpenCL and OpenGL buffers and then copy the results via the host memory. This is obviously inefficient, due to the usage of the system bus to copy data.

OpenCL provides buffer sharing with OpenGL via the extensions mechanism described in section 3.1. The same memory region located on the graphics card can be used by an application that utilizes both OpenGL and OpenCL.

The example presented here displays a simple lens effect in real time. It requires an OpenGL 2.1 and OpenCL 1.1 capable device. Most modern graphics cards are capable of running the example. The screenshot of the running example is depicted in figure 3.17. Note that to calculate this effect, the whole picture is processed in every frame.

Figure 3.17: The example program output

The example uses two buffers; the usual OpenCL buffer is for storing picture data, and the OpenGL pixel buffer is for destination texture. The second is the one that allows for cooperation of these two technologies.

The example does not provide full error checking, especially in the C version, because of simplicity requirements. The reader can easily add such functionality in a way similar to that described in section 2.7.

The example has been tested on Windows and Linux, but it should also be possible to run it on MacOS X with some minor changes.

3.4.1. Extensions Used

In order to use OpenCL–OpenGL memory buffer sharing, the drivers should support the following extensions and functions:

- OpenCL must support the `cl_khr_gl_sharing` extension. This is one of the standard extensions, so it is not necessary to use **`clGetExtensionFunctionAddress`**. The application's responsibility is to check if the extension is available for a given device

```
#if defined(__MACOSX__)
#include <OpenGL/gl.h>
#include <OpenGL/glu.h>
#else
#include <GL/gl.h>
#include <GL/glu.h>
#endif
```

Listing 3.64: OpenGL header files. This code is valid for both the C and the C++ example

- OpenGL must be in version 2.1 or higher. This is due to the usage of the functions **glGenBuffers**, **glBindBuffer**, **glBufferData** and **glDeleteBuffers** – defined in OpenGL standard version 1.5 – and the macro definitions that are from version 2.1

Note that almost every modern personal computer supports the following technologies, so the functionality presented in this section can be widely used in the consumer market – for example, in games and in the field of human-computer interaction.

3.4.2. Libraries

In order to make the code as portable as possible, the example uses a third-party library – Simple DirectMedia Layer. This library is freely available for many platforms like Windows, Linux and MacOS X. Its usage in the field is driven by the need for simplifying initialization code so the reader can focus on the issue of OpenCL–OpenGL cooperation. The SDL library is available per the terms of the GNU LGPL license and is mature enough that it is used in many commercial applications [19].

In the example, the SDL library is used for initialization of the OpenGL context and obtaining OpenGL 1.4 function addresses. Note that only core OpenGL 1.1 functions can always be linked during compilation.

3.4.3. Header Files

The application must include SDL, OpenGL and OpenCL header files. Listing 3.64 shows the usual way of including OpenGL header files.

The next important set of headers is for SDL. Listing 3.65 shows one of the ways to do it. Note that it is also possible to create an OpenGL context and window by using the system API, but it would make the example too complicated. `SDL.h` contains the core SDL API, and `SDL_opengl.h` basically allows SDL to use OpenGL. The SDL library is not the topic of this book, so it will not be described in detail. For more information about it, please see [19].

In order to properly create an OpenCL context, the application must obtain some system-specific information about OpenGL. In order to do so, `glx.h` or `wingdi.h` must be included. The first defines API specific to Linux, and the second is for Windows API. In order to use the example on MacOS X, one must include `CGLCurrent.h`

```
#include <SDL/SDL.h>
#include <SDL/SDL_opengl.h>
```

Listing 3.65: The SDL header files

```
#ifndef _WIN32
#include <GL/glx.h>
#else
#include <Wingdi.h>
#endif
```

Listing 3.66: The system specific headers allowing OpenGL context management

```
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/opencl.h>
#else
#include <CL/opencl.h>
#endif
```

Listing 3.67: OpenCL header files for C programming language

```
#define __CL_ENABLE_EXCEPTIONS 1
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif
```

Listing 3.68: OpenCL header files for C++ programming language

and `CGLTypes.h`. For further reading, please refer to the CGL documentation [20]. This include block can be seen in listing 3.66.

Last but not least, OpenCL header files. It has already been described in earlier chapters, but it is important enough that it can be presented here also. Listings 3.67 and 3.68 show the inclusion of API definitions of OpenCL for C and C++, respectively.

The C++ example uses the official C++ OpenCL API wrapper. This allows for very convenient usage of OpenCL. Note the preprocessor definition `__CL_ENABLE_EXCEPTIONS`. This macro defines that the exceptions can be thrown instead of traditional C-style error handling.

3.4.4. Common Actions

The sample application, as well as many real world examples using the technique presented here, will follow the following steps:

```
int initSDL(int width, int height) {
  SDL_Init(SDL_INIT_VIDEO);
  SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
  screen = SDL_SetVideoMode(width, height, 0, SDL_OPENGL);
  glGenBuffers = (PFNGLGENBUFFERSPROC)SDL_GL_GetProcAddress("glGenBuffers");
  glBindBuffer = (PFNGLBINDBUFFERPROC)SDL_GL_GetProcAddress("glBindBuffer");
  glBufferData = (PFNGLBUFFERDATAPROC)SDL_GL_GetProcAddress("glBufferData");
  glDeleteBuffers = (PFNGLDELETEBUFFERSPROC)SDL_GL_GetProcAddress(
    "glDeleteBuffers");
  return 0;
}
```

Listing 3.69: Initialization of OpenGL window and library functions using SDL. The code for C and C++ programming language

- Initialization of OpenGL window and context. In the example, this is done using SDL.
- Obtaining OpenGL functions for OpenGL versions above 1.1. This can be done using additional libraries or system-specific API calls, but in the example it is done using SDL.
- Initialization of OpenGL variables that drives the way of displaying objects on the screen.
- Initialization of OpenCL. This must be done **after** OpenGL context is created.
- Creation of OpenGL textures.
- Creation of OpenGL pixel buffer objects that can be shared with OpenCL.
- Computation.
- Cleanup.

The order of these steps is important, especially the initialization of the OpenCL context. It must be done after the OpenGL context is created. The inverse order is impossible because of the way the OpenCL context is created.

3.4.5. OpenGL Initialization

SDL_Init

```
int SDL_Init ( Uint32 flags );
```

The **SDL_Init** function initializes the Simple Directmedia Library and the subsystems specified by *flags*.

SDL_GL_SetAttribute

```
int SDL_GL_SetAttribute ( SDL_GLattr attr, int value );
```

Sets a special SDL/OpenGL attribute.

SDL_SetVideoMode

```
SDL_Surface* SDL_SetVideoMode ( int width, int height, int bitsperpixel,
   Uint32 flags );
```

Set up a video mode with the specified width, height and bits-per-pixel.

SDL_GL_GetProcAddress

```
void * SDL_GL_GetProcAddress ( const char* proc );
```

This SDL function returns the address of the GL function proc, or NULL if the function is not found.

```
1 PFNGLGENBUFFERSPROC glGenBuffers;
2 PFNGLBINDBUFFERPROC glBindBuffer;
3 PFNGLBUFFERDATAPROC glBufferData;
4 PFNGLDELETEBUFFERSPROC glDeleteBuffers;
```

Listing 3.70: Declaration of dynamically loaded OpenGL library functions

The window in the example is created using the SDL library. This library allows for simple creation of windows that contain the OpenGL context. The code for the C and C++ programming languages can be seen in listing 3.69. The first step is to initialize SDL using the function **SDL_Init**. Then the library parameter SDL_GL_DOUBLEBUFFER is set using **SDL_GL_SetAttribute** to instruct SDL that OpenGL will use double buffering – this is in order to make animation smoother. The actual window creation is done by the function **SDL_SetVideoMode**. This function creates a window with the OpenGL context and sets its parameters. The parameter SDL_OPENGL tells SDL that the created window is requested to also contain OpenGL. After **SDL_SetVideoMode**, OpenGL is ready to use. Everything that is system-dependent is done.

The next step is to get function addresses for OpenGL library functions that were added to the standard after version 1.1. This can be omitted on some systems (like Linux). The SDL function that allows for getting the OpenGL function address is called **SDL_GL_GetProcAddress** and can be seen in listing 3.69. These OpenGL functions declarations can be seen in listing 3.70. This code is for C and C++ and should be located at the beginning of the application after the include block. Note that these operations can be performed using GDI, GLX or CGL in Windows, Linux or MacOS X, respectively.

glGenBuffers

```
void glGenBuffers ( GLsizei n, GLuint *buffers );
```

Generate buffer object names

glBindBuffer

```
void glBindBuffer ( GLenum target, GLuint buffer);
```

Binds a named buffer object to *target*.

glBufferData

```
void glBufferData ( GLenum target, GLsizeiptr size, const GLvoid *data,
   GLenum usage );
```

This function creates and initializes a buffer object's data store.

glDeleteBuffers

```
void glDeleteBuffers ( GLsizei n, const GLuint *buffers );
```

This function deletes named buffer objects.

```
1  int initOpenGL(int width, int height) {
2    glEnable(GL_TEXTURE_2D);
3    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
4    glViewport(0, 0, width, height);
5    glMatrixMode(GL_PROJECTION);
6    glLoadIdentity();
7    glOrtho(0.0f, width, height, 0.0f, -1.0f, 1.0f);
8    glMatrixMode(GL_MODELVIEW);
9    glLoadIdentity();
10   return 0;
11 }
```

Listing 3.71: Configuration of OpenGL window scale and perspective. This source code is in both C and C++ versions of the example

OpenGL Display Configuration

Another common phase present in applications that use OpenGL is configuration of OpenGL features – initial state. An example initializes OpenGL in the way shown in listing 3.71. This listing is the same for C and C++ versions of application. This is because OpenGL is basically the C API.

The first command enables texturing. The default OpenGL behavior is to skip the texturing phase, so in this application it has to be turned on. The function **glEnable** is used to enable selected features of OpenGL. The next command – **glClearColor** – sets the color that will be used to clean the OpenGL window. The function **glViewport** sets the bounds of the frame used by OpenGL to draw. This frame is within a window.

OpenGL supports perspective and orthogonal views. For the example, the orthogonal view is selected. This is done by using the function **glOrtho** and setting a projection matrix and a modelview matrix to an identity. The current transformation matrix is selected by the function **glMatrixMode**, and then the function **glLoadIdentity** sets this matrix to identity.

glClearColor

```
void glClearColor ( GLclampf red, GLclampf green, GLclampf blue,
   GLclampf alpha );
```

This function specifies clear values for the color buffers.

glViewport

```
void glViewport ( GLint x, GLint y, GLsizei width, GLsizei height );
```

This function sets the viewport.

glOrtho

```
void glOrtho ( GLdouble left, GLdouble right, GLdouble bottom,
    GLdouble top, GLdouble nearVal, GLdouble farVal );
```

This function multiplies the current matrix with an orthographic matrix.

glMatrixMode

```
void glMatrixMode ( GLenum mode );
```

This function specifies which matrix is the current matrix.

glLoadIdentity

```
void glLoadIdentity ( void );
```

This function replaces the current matrix with the identity matrix.

3.4.6. OpenCL Initialization

Most of OpenCL initialization is done in the same fashion as presented in previous chapters and sections. The major difference lies in context creation.

OpenCL Context Creation

In order to enable cooperation of OpenCL and OpenGL, the OpenCL context must be initialized in a special way. This is because OpenCL must "know" which OpenGL context to cooperate with.

The function for creating an OpenCL context can be seen in listings 3.72 and 3.73 for C and C++, respectively. The major difference between this method of context creation and the one presented in section 2.5 is in the way `cl_context_properties` is set. The usual way is to include only information about the platform. Here are also handles to the OpenGL context and display. Note that this is done in a conditional compilation block. This is because OpenGL contexts are managed in a system-dependent way. On Linux, GLX is used to set up the OpenGL context, and on Windows it is done by GDI functions.

In the C++ version, there is also checking for the presence of `cl_khr_gl_sharing` on the device. This function is presented in listing 3.74.

Both functions search for the device that is of the type given in the parameter. These functions check every platform; if the device of a selected type is available, then it is selected to be put into the context. Note that it will work properly only for OpenGL-capable devices, so it will probably not work on the `CPU` or `ACCELERATOR` device. The only reasonable device type in this case is `CL_DEVICE_TYPE_GPU`. It can probably work for `CL_DEVICE_TYPE_ALL` and `CL_DEVICE_TYPE_DEFAULT` as well.

OpenCL Objects Initialization

The example function that sets up the context, program, command queue and kernel is shown in listings 3.75 and 3.76 for C and C++.

```
cl_context contextFromTypeGL(cl_device_type devtype) {
  int i;
  cl_context context = NULL;
  cl_platform_id *platforms;
  cl_device_id devices [1];
  cl_context_properties cps [7];
  cl_uint devices_n, platforms_n;
  clGetPlatformIDs(0, NULL, &platforms_n);
  platforms =
    ( cl_platform_id * )malloc(sizeof(cl_platform_id) * (platforms_n + 1));
  clGetPlatformIDs(platforms_n, platforms, &platforms_n);
  for (i = 0; i < platforms_n; i++) {
    clGetDeviceIDs(platforms [i], devtype, 0, NULL, &devices_n);
    if (devices_n > 0) {
      clGetDeviceIDs(platforms [i], devtype, 1, devices, NULL);
    #ifndef _WIN32 // Linux
      cps [0] = CL_GL_CONTEXT_KHR;
      cps [1] = (cl_context_properties)glXGetCurrentContext();
      cps [2] = CL_GLX_DISPLAY_KHR;
      cps [3] = (cl_context_properties)glXGetCurrentDisplay();
      cps [4] = CL_CONTEXT_PLATFORM;
      cps [5] = (cl_context_properties)platforms [i];
    #else // Win32
      cps [0] = CL_GL_CONTEXT_KHR;
      cps [1] = (cl_context_properties)wglGetCurrentContext();
      cps [2] = CL_WGL_HDC_KHR;
      cps [3] = (cl_context_properties)wglGetCurrentDC();
      cps [4] = CL_CONTEXT_PLATFORM;
      cps [5] = (cl_context_properties)platforms [i];
    #endif
      cps [6] = 0;
      context = clCreateContext(cps, 1, &devices [0], NULL, NULL, NULL);
      break;
    }
  }
  free(platforms);
  return context;
}
```

Listing 3.72: Initialization of OpenCL context that can share data with OpenGL context. Example in C programming language

Both codes perform the same operations. First, the function **contextFromTypeGL** is called. This function was described in a previous subsection. Next, it initializes the OpenCL program using the function **createProgramBin**. This function is the one presented in previous sections. The methods for loading the program were described in section 2.12.3. The command queue is also initialized in the usual way. It is a standard in-order command queue. The kernel that generates the lens effect is stored in the kernel object called *lensEffectKernel*.

The reader may notice that the differences between the normal OpenCL setup and the initialization presented here are very small.

```
cl::Context contextFromTypeGL(cl_device_type devtype) {
  std::vector < cl::Device > devices;
  std::vector < cl::Device > device;
  std::vector < cl::Platform > platforms;
  cl::Platform::get(&platforms);

  for (size_t i = 0; i < platforms.size(); i++) {
    try {
      platforms [i].getDevices(devtype, &devices);
      if (devices.size() > 0) {
        for (size_t j = 0; j < devices.size(); j++) {
          if (checkExtension(devices [j], "cl_khr_gl_sharing")) {
#if defined(linux)
            cl_context_properties cps[] = {
              CL_GL_CONTEXT_KHR, (cl_context_properties)glXGetCurrentContext(),
              CL_GLX_DISPLAY_KHR, (cl_context_properties)glXGetCurrentDisplay(),
              CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms [i])(),
              0
            };
#else // Win32
            cl_context_properties cps[] = {
              CL_GL_CONTEXT_KHR, (cl_context_properties)wglGetCurrentContext(),
              CL_WGL_HDC_KHR, (cl_context_properties)wglGetCurrentDC(),
              CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms [i])(),
              0
            };
#endif
            device.push_back(devices [j]);
            return cl::Context(device, cps, NULL, NULL);
          }
        }
      }
    } catch (cl::Error e) { // this is thrown in case of 0 devices of specified type
      // nothing to be done here
    }
  }
  throw cl::Error(-1, "No such device");
}
```

Listing 3.73: Initialization of OpenCL context that can share data with OpenGL context. The example in C++ programming language

3.4.7. Creating Buffer for OpenGL and OpenCL

OpenGL contains an object type called the pixel buffer object (PBO). This kind of object is one of the types of buffers available for OpenGL. It can store information about points – for example, the image. These objects are very similar to the vertex buffer objects for storing vertex information.

Functions for handling PBOs were introduced in OpenGL 1.4, and the functions and definitions used in the example are from version 2.1, which is supported on most modern graphics hardware. A PBO can easily be used as a texture source.

If an OpenCL context is created that is bound to an OpenGL context, then the

```
int checkExtension(cl::Device d, const char *ext) {
  std::string extensions = d.getInfo < CL_DEVICE_EXTENSIONS > ();
  std::istringstream iss;
  iss.str(extensions);
  while (iss) {
    std::string extname;
    iss >> extname;
    if (extname.compare(ext) == 0) {
      return 1;
    }
  }
  return 0;
}
```

Listing 3.74: Simple checking for the presence of an extension on given device. The C++ code

```
int initOpenCL() {
  cl_int r;
  context = contextFromTypeGL(CL_DEVICE_TYPE_GPU);
  if (context == NULL)
    errorCheck(CL_INVALID_ARG_VALUE, "Could not create context");
  listDevicesInContext(context);
  program = createProgramBin(context, "kernel/oclogl.cl", NULL);
  if (program == NULL)
    errorCheck(CL_INVALID_ARG_VALUE, "Could not create program");
  cl_device_id devices [1];
  clGetContextInfo(context, CL_CONTEXT_DEVICES,
    sizeof(cl_device_id), devices, NULL);
  queue = clCreateCommandQueue(context, devices [0], 0, &r);
  errorCheck(r, "Could not create command queue");
  lensEffectKernel = clCreateKernel(program, "lensEffect", &r);
  errorCheck(r, "Could not create kernel");
  return 0;
}
```

Listing 3.75: Initialization of OpenCL objects – the C example

```
void initOpenCL() {
  context = contextFromTypeGL(CL_DEVICE_TYPE_GPU);
  listDevicesInContext(context);
  program = createProgramBin(context, "kernel/oclogl.cl");
  queue = cl::CommandQueue(context, context.getInfo<CL_CONTEXT_DEVICES > () [0], 0);
  lensEffectKernel = cl::Kernel(program, "lensEffect");
}
```

Listing 3.76: Initialization of OpenCL objects – the C++ example

```
typedef struct TextureBufferShared {
  int width, height;
  GLuint gl_texture_id;
  GLuint gl_buffer_id;
  cl_mem cl_tex_buffer;
} TextureBufferShared;

cl_mem inputPicture;
TextureBufferShared myTexture;
```

Listing 3.77: Global variables and types for storing textures and buffers. The C source code

```
typedef struct TextureBufferShared {
  int width, height;
  GLuint gl_texture_id;
  GLuint gl_buffer_id;
  cl::Memory cl_tex_buffer;
} TextureBufferShared;

cl::Buffer inputPicture;
TextureBufferShared myTexture;
```

Listing 3.78: Global variables and types for storing textures and buffers. The C++ source code

PBOs can share the same memory region as the OpenCL global memory. This allows for the usage of PBOs by the OpenCL program.

In order to display a PBO as a texture, the texture of the same size must be created. Before displaying such a texture, OpenGL must be instructed to bind the PBO as a pixel source for the texture.

To sum up: The objects needed for full cooperation between OpenGL and OpenCL are:

- Pixel buffer object - the buffer that will be visible for OpenGL and will store image data.
- OpenGL texture - this object is directly used by OpenGL and its source can be bound to the PBO.
- OpenCL buffer – created in such a special way that it will use the same memory region as the PBO.

These three objects can be grouped for convenience, and it is done in this way in the example. The global variables and the type for grouping these three objects together are presented in listings 3.77 and 3.78 for C and C++.

These example parts define the `TextureBufferShared` type that groups texture, PBO and OpenCL memory objects. The objects defined in *`myTexture`* will be used as an output of the lens effect filter. In these listings, there is also a defined

```
1 void createPixelBufferObject(GLuint* pbo, int width, int height) {
2   glGenBuffers(1, pbo);
3   glBindBuffer(GL_ARRAY_BUFFER, *pbo);
4   glBufferData(GL_ARRAY_BUFFER, sizeof(GLubyte) * width * height * 4, NULL,
5     GL_DYNAMIC_DRAW);
6   glBindBuffer(GL_ARRAY_BUFFER, 0);
7 }
```

Listing 3.79: Creation of pixel buffer object

OpenCL memory buffer called *inputPicture*. This buffer is for storing the original, unchanged picture that will be used for the lens effect.

Initialization of Pixel Buffer Object

PBO initialization is presented in listing 3.79. This code is valid for both C and C++ versions of the example.

PBO creation is similar to the creation of an OpenGL texture object. The procedure is as follows:

- Generate one buffer. The function **glGenBuffers** can also produce multiple buffers, but for this demonstration it is not necessary.
- Select the newly created buffer. This is done using the function **glBindBuffer** with the parameter GL_ARRAY_BUFFER that binds the buffer to the OpenGL array buffer so it can be used and configured.
- Create and initialize a selected buffer object's data store using function **glBufferData**.
- Unbind the buffer. This is done by the function **glBindBuffer** with the buffer number equal to 0.

Note that this is a sequence of commands. It is because OpenGL is a standard that describes a state machine, so some operations must be done in this way.

Initialization of OpenGL Texture

Another important element is an OpenGL texture object or, briefly, texture. This is an object that stores image data that can be mapped to the 3D and 2D figures. In the example, this texture is mapped to the square that fills the whole application window. The effect results are displayed via this texture.

The function that generates the texture can be seen in listing 3.80. This is valid for both C and C++ versions of the example. This function first creates one texture object for OpenGL. This is done by using the API function **glGenTextures**. The next step is similar to the one from the creation of PBO – bind the texture to the current 2D texture object using the function **glBindTexture**. The next step is to set the texture alignment using the function **glPixelStorei**. This is the value that will be applied to the rows. The default is 4, meaning that every row will be of a length divisible by 4. In the example, the alignment is set to 1, so the size of the texture will always be

```
1 void createTextureObject(GLuint *texture, int w, int h) {
2   glGenTextures(1, texture);
3   glBindTexture(GL_TEXTURE_2D, *texture);
4   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
5   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
6   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, w, h, 0, GL_RGBA, GL_UNSIGNED_BYTE,
7     NULL);
8 }
```

Listing 3.80: Creation of an OpenGL texture object along with its setup

the same as the number of pixels. In fact, for this example, the default alignment will also work well because the image size is 512×512 – that is, divisible by 4.

glGenTextures

```
void glGenTextures ( GLsizei n, GLuint *textures );
```

This function generates texture names.

glBindTexture

```
void glBindTexture(GLenum target, GLuint texture );
```

This function binds a named texture to a texturing target.

glPixelStorei

```
void glPixelStorei ( GLenum pname, GLint param );
```

This function sets pixel storage modes.

glTexParameteri

```
void glTexParameteri ( GLenum target, GLenum pname, GLint param );
```

This function sets texture parameters.

The function **glTexParameteri** configures the texture options that are not related to the physical store of the data. The texture filter must be set in order to display the texture properly. This is because of OpenGL specification [21]. This filter is responsible for interpolation of pixels from the texture. `GL_LINEAR` means that the filter will interpolate linearly. The other option is `GL_NEAREST`, which would select just the nearest pixel value. There are also options for mipmapped textures, but this is not the topic of the current discussion.

The last step is to create the storage space for the texture. This is done using the function **glTexParameteri**. This function allocates memory for the texture and sets the storage method. `GL_RGBA8` means that the texture will be stored as red, green,

```
TextureBufferShared prepareSharedTexture(int w, int h) {
  cl_int r;
  TextureBufferShared t;
  t.width = w;
  t.height = h;
  // generate texture
  createPixelBufferObject(&t.gl_buffer_id, w, h);
  // create texture
  createTextureObject(&t.gl_texture_id, w, h);
  // create OpenCL buffer that shares memory with OpenGL PBO
  t.cl_tex_buffer = clCreateFromGLBuffer(context,
    CL_MEM_READ_WRITE, t.gl_buffer_id, &r);
  errorCheck(r, "Could not create memory object from GL object");
  return t;
}
```

Listing 3.81: Preparation of PBO, OpenGL texture and OpenCL buffer – the C code

```
TextureBufferShared prepareSharedTexture(int w, int h) {
  TextureBufferShared t;
  t.width = w;
  t.height = h;
  // create shared buffer
  createPixelBufferObject(&t.gl_buffer_id, w, h);
  // create texture
  createTextureObject(&t.gl_texture_id, w, h);
  // create OpenCL buffer that shares data with PBO
  t.cl_tex_buffer = cl::BufferGL(context, CL_MEM_READ_WRITE, t.gl_buffer_id);

  return t;
}
```

Listing 3.82: Preparation of PBO, OpenGL texture and OpenCL buffer – the C++ code

blue and alpha values. Each color value will be of size 8 bits, so one pixel will be 4 bytes. `GL_UNSIGNED_BYTE` makes the texture data to be stored as unsigned bytes.

Note that the type and size of the texture is compatible with the PBO created by the function **`createPixelBufferObject`**.

Binding Buffers Together

The function that creates the PBO, the texture and the OpenCL buffer can be seen in listings 3.81 and 3.82 for C and C++.

This function uses the previously described functions to create the PBO and the texture.

The most important commands used in this fragment are **`clCreateFromGLBuffer`** and **`cl::BufferGL`** for C and C++, respectively. **`cl::BufferGL`** is, in fact, the constructor of the OpenCL buffer object that uses

clCreateFromGLBuffer

```
cl_mem clCreateFromGLBuffer ( cl_context context, cl_mem_flags flags,
   GLuint bufobj, cl_int * errcode_ret );
```

Creates an OpenCL buffer object from an OpenGL buffer object.

cl::BufferGL::BufferGL

```
cl::BufferGL::BufferGL ( const Context & context, cl_mem_flags flags,
   GLuint bufobj, cl_int *err = NULL );
```

Creates a buffer object that can be shared between OpenCL and OpenGL.

the OpenGL PBO. This constructor takes the OpenCL context and the PBO object identifier to produce the `cl::BufferGL` object that uses the memory from OpenGL. The same functionality is provided by the function **`clCreateFromGLBuffer`**. This function also takes the OpenCL context and the PBO to produce the `cl_mem` buffer that uses memory from the PBO. Note that after creation of the OpenCL buffer object that shares memory with the PBO, the memory is owned by the OpenGL system, so OpenCL cannot use it yet. The solution will be presented in the following subsections.

The Main Initialization Function

The full initialization function used in the example can be seen in listings 3.83 and 3.84 for C and C++. This function gathers the fragment initialization functions shown in previous subsections and also loads the picture from disk to OpenCL buffer.

Please note the order of initialization:

- Initialization of the SDL window that will hold the OpenGL context as well as the initialization of the context itself.
- Next – the initialization of OpenGL features. This is possible because the OpenGL window and context are ready.
- After these steps, it is time to initialize OpenCL. Actually, this can be done immediately after OpenGL context creation.
- Initialization of OpenGL and OpenCL objects. This is done last, because it depends on all of the previous steps.

In these codes, the image is loaded from a file called `picture.raw` that is located, along with the example, on the attached disk.

3.4.8. Kernel

The kernel in this example is used to process the image in order to produce a lens effect. In every frame, the whole image is processed. The algorithm for the lens effect is very simple. It could be probably calculated in real time by an ordinary CPU, but this is just an example.

The kernel used in the example is shown in listing 3.87. This kernel uses additional functions shown in listings 3.85 and 3.86.

Helper functions are used in order to simplify and clarify code. **`getpixel`** returns the pixel value at given coordinates. The function **`setpixel`** sets the pixel value at

```
void init() {
  cl_int r;
  // the initialization order is important
  initSDL(WIDTH, HEIGHT);
  initOpenGL(WIDTH, HEIGHT);
  initOpenCL();

  myTexture = prepareSharedTexture(WIDTH, HEIGHT);
  size_t size = WIDTH * HEIGHT * 4;
  cl_uchar *bin_data = (cl_uchar *)malloc(size);
  FILE *f = fopen("picture.raw", "rb");
  if (f != NULL) {
    r = fread(bin_data, 1, size, f);
    inputPicture = clCreateBuffer(context,
      CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, size, bin_data, &r);
    errorCheck(r, "Could not create OpenCL buffer");
    free(bin_data);
  } else {
    printf("Could not open picture.raw\n");
    exit(-1);
  }
}
```

Listing 3.83: The function that gathers the whole initialization code in one place – the C programming language

```
void init() {
  // the initialization order is important
  initSDL(WIDTH, HEIGHT);
  initOpenGL(WIDTH, HEIGHT);
  initOpenCL();

  myTexture = prepareSharedTexture(WIDTH, HEIGHT);
  std::ifstream imageFile("picture.raw", std::ios::in | std::ios::binary);
  if (imageFile.is_open()) {
    size_t size = WIDTH * HEIGHT * 4;
    cl_uchar *bin_data = new cl_uchar [size];
    imageFile.read((char *)bin_data, size);
    imageFile.close();
    inputPicture = cl::Buffer(context,
      CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, size, bin_data);
    delete bin_data;
  } else {
    throw cl::Error(-999, "Raw picture file not available");
  }
}
```

Listing 3.84: The function that gathers the whole initialization code in one place – the C++ programming language

```
uchar4 getpixel(global uchar4 *data, int w, int h, int x, int y) {
  if (x >= w) x = w - 1;
  if (y >= h) y = h - 1;
  if (x < 0) x = 0;
  if (y < 0) y = 0;
  return data [y * w + x];
}
```

Listing 3.85: OpenCL program function getpixel

```
void setpixel(global uchar4 *data, int w, int h, int x, int y, uchar4 color) {
  if ((x >= w) || (y >= h) || (x < 0) || (y < 0)) return;
  data [y * w + x] = color;
}
```

Listing 3.86: OpenCL program function setpixel

given coordinates. Both functions check if the requested pixel is within the size of the image. If not, then **getpixel** returns the closest one that fits in the image, and **setpixel** just returns.

The kernel shown in listing 3.87 takes the following parameters:

- *inputbuffer* – the image source for the lens effect.
- *buffer* – the output buffer. The results will be written to this buffer. In the example, this points to the texture data, which will be displayed on the screen.
- *width* – the width of the images.
- *height* – the height of the images.
- *mouse_x* – the x coordinate of the effect center; it is equal to the mouse x coordinate.
- *mouse_y* – the y coordinate of the effect center, it is equal to the mouse y coordinate.

The kernel uses the vector types available in OpenCL C. Every kernel instance calculates only one pixel of the output. Note that the kernel parameters do not differ from the ones that were presented in previous chapters. From the kernel point of view, the global memory is just global memory. It does not distinguish between video memory or any other kind of memory. This allows for seamlessly adapting existing kernels to be used in graphics.

Algorithm

The idea behind the lens effect is to “push back” pixels that are closer to the center of the effect. The further from the center, the more push is done. Figure 3.18 shows this idea.

In figure 3.18, l is the distance between the center of the effect – the mouse position – and the current pixel. Please recall the kernel from listing 3.87. The current

```
kernel void lensEffect(global uchar4 *inputbuffer, global uchar4 *buffer,
                       int width, int height, int mouse_x,
                       int mouse_y) {
  float l, w;
  float2 delta, src_p, dst_p, mouse_p;
  uchar4 color;
  int2 global_id = { get_global_id(0), get_global_id(1) };
  // conversion int -> float
  dst_p.x = global_id.x;
  dst_p.y = global_id.y;
  mouse_p.x = mouse_x;
  mouse_p.y = mouse_y;

  delta = mouse_p - dst_p;
  l = length(delta);
  if (l < EFFECTSIZE) {
    w = EFFECTSIZE - l;
    src_p = mouse_p - delta * 0.3 * pow(EFFECTSIZE / w, 0.1);
  } else {
    src_p = dst_p;
  }
  // getting the appropriate pixel from input and copy it's value to output
  color = getpixel(inputbuffer, width, height, (int)src_p.x, (int)src_p.y);
  setpixel(buffer, width, height, global_id.x, global_id.y, color);
}
```

Listing 3.87: OpenCL kernel producing lens effect.

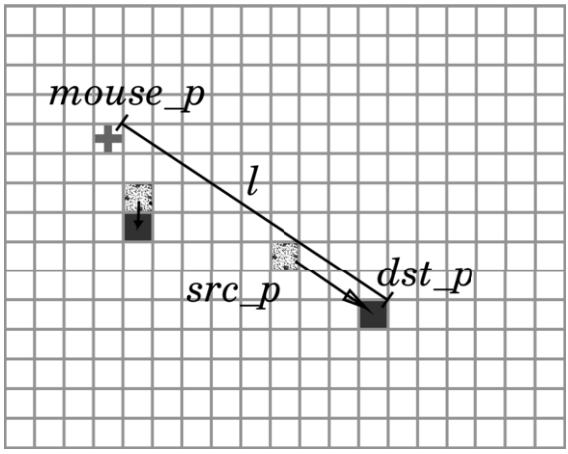

Figure 3.18: Graphical description of the method used by kernel to generate lens effect.

pixel is at the coordinates equal to the `global ID`, so every kernel instance calculates the pixel at its own coordinates. The center of the lens effect is called *mouse_p* and the destination pixel coordinates are called *dst_p*. The value that must be calculated is *src_p*. Note that at some distance from center, the effect is not present.

In order to properly calculate *src_p*, the kernel must first get the distance between the center of the effect and the self coordinates (*dst_p*). The vector used in the example that points from *mouse_p* (center) to the *dst_p* is called *delta* and it is calculated using OpenCL vector substitution. The length is calculated using the built-in OpenCL C function **length** that operates on vectors. The length of *delta* is stored in l.

The equation 3.8 is used to calculate the source pixel position. Let R be equal to the lens radius; in the kernel it is called EFFECTSIZE.

$$C_1 = 0.3 \tag{3.5}$$

$$C_2 = 10 \tag{3.6}$$

$$w = R - l \tag{3.7}$$

$$\text{src\_p} = \text{mouse\_p} - C_1 \cdot \text{delta} \cdot \sqrt[C_2]{\frac{R}{w}} \tag{3.8}$$

Note that this equation is calculated onlywithin the radius (R) of the effect. The main element of the equation is implemented by substitution $\text{mouse\_p} - C_1 \cdot \text{delta}$. This allows to magnification of part of the image. The magnification factor is C_1. This factor is below 1 here, so it magnifies the image; if it were greater than 1, the effect would be the opposite – a diverging lens. $\sqrt[C_2]{\frac{R}{w}}$ drives the "flatness" of the effect. If it were omitted, then the lens would just magnify the image, with the same magnitude at every point. The smaller the degree of the root (C_2) the more the effect resemples looking through a crystal ball. The greater it is, the more the effect is "flat".

3.4.9. Generating Effect

clEnqueueAcquireGLObjects

```
cl_int clEnqueueAcquireGLObjects ( cl_command_queue command_queue,
    cl_uint num_objects, const cl_mem *mem_objects, cl_uint
    num_events_in_wait_list, const cl_event *event_wait_list,
    cl_event *event );
```

Acquire OpenCL memory objects that have been created from OpenGL objects.

cl::CommandQueue::enqueueAcquireGLObjects

```
cl_int cl::CommandQueue::enqueueAcquireGLObjects ( const VECTOR_CLASS< Memory >
    *mem_objects=NULL, const VECTOR_CLASS< Event > *events=NULL,
    Event *event=NULL );
```

Acquire OpenCL memory objects that have been created from OpenGL objects.

OpenCL–OpenGL shared buffers are operated in a special way. If OpenGL needs to access this buffer, then the OpenCL code should not operate on it. The opposite also holds true – in order to perform some calculations on this kind of buffer, the OpenGL should not use it. Failing to obey these rules can lead to undefined behavior of the application; in particular, it can crash the application.

clEnqueueReleaseGLObjects

```
cl_int clEnqueueReleaseGLObjects ( cl_command_queue command_queue,
    cl_uint num_objects, const cl_mem *mem_objects, cl_uint
    num_events_in_wait_list, const cl_event *event_wait_list,
    cl_event *event );
```

Release OpenCL memory objects that have been created from OpenGL objects.

cl::CommandQueue::enqueueReleaseGLObjects

```
cl_int cl::CommandQueue::enqueueReleaseGLObjects ( const VECTOR_CLASS< Memory >
    *mem_objects=NULL, const VECTOR_CLASS< Event > *events=NULL,
    Event *event=NULL );
```

Release OpenCL memory objects that have been created from OpenGL objects.

```
1 void acquireFromGLtoCL(TextureBufferShared t) {
2   glFlush();
3   clEnqueueAcquireGLObjects(queue, 1, &t.cl_tex_buffer, 0, NULL, NULL);
4 }
```

Listing 3.88: Acquiring OpenGL objects for OpenCL. The C example

```
1 void acquireFromGLtoCL(TextureBufferShared tex) {
2   std::vector<cl::Memory> textureBuffers;
3   textureBuffers.push_back(tex.cl_tex_buffer);
4   glFlush();
5   queue.enqueueAcquireGLObjects(&textureBuffers);
6 }
```

Listing 3.89: Acquiring OpenGL objects for OpenCL. The C++ example

Ownership of OpenCL/OpenGL Objects

The operation of changing the current system that owns the object is done using these functions:

- The C function **clEnqueueAcquireGLObjects** and the C++ API method **cl::CommandQueue::enqueueAcquireGLObjects** transfer the ownership of object from OpenGL to OpenCL.
- The C API function **clEnqueueReleaseGLObjects** and the C++ API method **cl::CommandQueue::enqueueReleaseGLObjects** transfer the ownership of the object back to OpenGL.

The sample code in listings 3.88 and 3.89 show how to acquire OpenGL objects so they can be used safely by OpenCL. Note that there must be an implicit synchronization point that ensures that every OpenGL operation is finished. The function **glFlush** is responsible for synchronization with OpenGL. This is necessary because the behavior of the program, in the case of omitting this step is undefined. Acquisition of the OpenGL object can be seen in the third line of listing 3.88 and on the fifth line of listing 3.89. The C++ version needs to use the `vector` type that can be seen

```
1 void acquireFromCLtoGL(TextureBufferShared tex) {
2   std::vector<cl::Memory> textureBuffers;
3   textureBuffers.push_back(tex.cl_tex_buffer);
4   queue.enqueueReleaseGLObjects(&textureBuffers);
5   queue.finish();
6 }
```

Listing 3.91: The example function that returns the ownership of the object back to OpenGL – the C example.

in line 2 to pass the list of buffers. The functions **clEnqueueAcquireGLObjects** and **cl::CommandQueue::enqueueAcquireGLObjects** operate on lists of objects, so it is also possible to acquire more than one object using one function call. In the situation when there are multiple such objects, it is advised to acquire them all at once. Failing to do so may lead to low performance of the application.

```
1 void acquireFromCLtoGL(TextureBufferShared t) {
2   clEnqueueReleaseGLObjects(queue, 1, &t.cl_tex_buffer, 0, NULL, NULL);
3   clFinish(queue);
4 }
```

Listing 3.90: The example function that returns the ownership of the object back to OpenGL – the C example.

The functions that return ownership of the buffer back to OpenGL are shown in listings 3.90 and 3.91 for C and C++. Note that there are also implicit synchronization points. OpenCL operations must finish before the objects can be returned to OpenGL. Both functions operate on lists of buffers, so it is possible to release multiple buffers using one method/function call.

3.4.10. Running Kernel that Operates on Shared Buffer

Execution of the OpenCL kernel that uses OpenGL buffer is shown in listings 3.92 and 3.93 for C and C++ respectively. It is very important to acquire OpenGL buffers for OpenCL and release them when the kernel finishes its operation.

In the listings, the step marked by (1) is responsible for the acquiring buffers. It is a call to the function shown in listings 3.88 and 3.88 for C and C++, respectively. This step cannot be omitted, because failing to do so will result in undefined behavior. It is an important issue, because it can happen that an application will work as expected on a developer computer but will fail on the customer's hardware.

The lines marked by (2) are responsible for running a kernel. Note that it does not differ from the usual kernel execution. It sets the parameters first and then executes the kernel for an NDRange of `WIDTH` × `HEIGHT` and an automatically selected work group size.

The last, but also very important, step is to release the buffer so it can be used by OpenGL. This is done in step (3). It uses the function presented in listings 3.90 and 3.91 for C and C++.

```
void lens(cl_int x, cl_int y) {
  cl_int w = WIDTH, h = HEIGHT;
  // (1)
  acquireFromGLtoCL(myTexture);
  // (2)
  size_t global_size[] = { w, h };
  clSetKernelArg(lensEffectKernel, 0, sizeof(cl_mem), &inputPicture);
  clSetKernelArg(lensEffectKernel, 1, sizeof(cl_mem), &myTexture.cl_tex_buffer);
  clSetKernelArg(lensEffectKernel, 2, sizeof(w), &w);
  clSetKernelArg(lensEffectKernel, 3, sizeof(h), &h);
  clSetKernelArg(lensEffectKernel, 4, sizeof(x), &x);
  clSetKernelArg(lensEffectKernel, 5, sizeof(y), &y);
  clEnqueueNDRangeKernel(queue, lensEffectKernel,
    2, NULL, global_size, NULL, 0, NULL, NULL);
  // (3)
  acquireFromCLtoGL(myTexture);
}
```

Listing 3.92: Execution of lens effect – the C example

```
void lens(cl_int x, cl_int y) {
  // (1)
  acquireFromGLtoCL(myTexture);
  // (2)
  lensEffectKernel.setArg(0, inputPicture);
  lensEffectKernel.setArg(1, myTexture.cl_tex_buffer);
  lensEffectKernel.setArg(2, WIDTH);
  lensEffectKernel.setArg(3, HEIGHT);
  lensEffectKernel.setArg(4, x);
  lensEffectKernel.setArg(5, y);
  queue.enqueueNDRangeKernel(lensEffectKernel,
    cl::NullRange, cl::NDRange(WIDTH, HEIGHT), cl::NullRange);
  // (3)
  acquireFromCLtoGL(myTexture);
}
```

Listing 3.93: Execution of lens effect – the C+ example

3.4.11. Results Display

One of the OpenGL state elements is the current texture. This texture is then mapped to 3D or 2D objects. The usual way of binding texture as current texture is shown in listing 3.94. The function that binds a PBO to the texture is shown in listing 3.95. This function is valid for both the C and C++ source codes, because it uses only the OpenGL API. This function replaces the usual way of binding a texture.

The first operation performed in function from listing 3.95 is to bind the pixel buffer object as the current buffer. The second step is to bind the current texture object. This is done using **glBindTexture**. Note that these two steps can be executed in a different order. The important thing is to select both – the pixel buffer object and the texture. When both objects are bound, then the function **glTexSubImage2D**

```
glBindTexture(GL_TEXTURE_2D, gl_texture_id);
```

Listing 3.94: Binding an OpenGL texture as a current texture.

```
void bindAsCurrentTexture(TextureBufferShared t) {
  // Now the texture must be selected
  glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, t.gl_buffer_id);
  glBindTexture(GL_TEXTURE_2D, t.gl_texture_id);
  // set selectd buffer as texture subimage
  glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0,
    t.width, t.height, GL_RGBA, GL_UNSIGNED_BYTE, 0);
}
```

Listing 3.95: Using a PBO as a pixel source for the texture.

```
void draw() {
  bindAsCurrentTexture(myTexture);

  glBegin(GL_QUADS);
  glTexCoord2i(0, 0); glVertex3f(0.f, 0.f, 0.0f);
  glTexCoord2i(1, 0); glVertex3f(WIDTH, 0, 0);
  glTexCoord2i(1, 1); glVertex3f(WIDTH, HEIGHT, 0);
  glTexCoord2i(0, 1); glVertex3f(0, HEIGHT, 0);
  glEnd();
}
```

Listing 3.96: The function for drawing a square with texture generated by the OpenCL kernel.

can operate in a different way than usual. Now **glTexSubImage2D** selects the pixel source for the texture to be from the PBO. The last parameter of this function is no longer the pointer to the client-side memory from which the pixels are to be unpacked; now, it is the offset in the PBO. In the example, it is 0.

glTexSubImage2D

```
void glTexSubImage2D ( GLenum target, GLint level, GLint xoffset,
    GLint yoffset, GLsizei width, GLsizei height, GLenum format,
    GLenum type, const GLvoid *data );
```

This function specifies a two-dimensional texture subimage.

The function that displays the image is shown in listing 3.96 for both C and C++ versions of the example. This function uses the function from listing 3.95 to select the current texture.

After binding the texture, this function begins drawing a square. OpenGL operates sequentially, so the state changes are valid until they are altered. Here, it first initializes drawing quads; then it sets the consecutive coordinate of the vertexes and

```
void mainLoop() {
  int quit = 0;
  SDL_Event event;
  int x = 0, y = 0;

  while (!quit) {
    while (SDL_PollEvent(&event)) {
      switch (event.type) {
      case SDL_KEYDOWN:
        if (event.key.keysym.sym == 27) quit = 1;
        break;
      case SDL_MOUSEMOTION:
        x = event.motion.x;
        y = event.motion.y;
        break;
      case SDL_QUIT:
        quit = 1;
        break;
      }
    }
    lens(x, y);
    draw();
    SDL_GL_SwapBuffers();
    SDL_Delay(10);
    glFinish();
  }
}
```

Listing 3.97: The main loop. Message handling and results display

the corresponding coordinates on the texture. Note that coordinates on the texture are from 0 to 1. It is a real value, so fractions are also acceptable.

Note that usage of the shared buffers adds only a small amount of additional code. Texture binding adds only two more function calls, so this technique can be very useful and cheap in implementation.

3.4.12. Message Handling

The last element is the main loop. This is the place where the commands from the user are processed and consecutive calls to display an image are done. This function is shown in listing 3.97 for C and C++.

The *x* and *y* variables store the current mouse cursor coordinates. This function's loop does the following operations:

First, it checks the user events. It is possible that there is more than one event that appeared since the last check, so this is done in a loop. The events are received using the SDL function **SDL_PollEvent**. If the mouse is moved, then the current mouse coordinates are updated so the effect can calculate correctly.

The next step is to execute the kernel that calculates the lens effect at the coordinates equal to the mouse coordinates. This calls the function described in listing 3.92.

SDL_PollEvent

```
int SDL_PollEvent ( SDL_Event *event );
```

Polls for currently pending events.

SDL_GL_SwapBuffers

```
void SDL_GL_SwapBuffers ( void );
```

Swap OpenGL framebuffers/Update Display.

```
1 void deletePixelBufferObject(GLuint* pbo) {
2   glBindBuffer(GL_ARRAY_BUFFER, *pbo);
3   glDeleteBuffers(1, pbo);
4   *pbo = 0;
5 }
```

Listing 3.98: Releasing OpenGL pixel object buffer.

Then the square is drawn, using the calculation results. The function **draw** can be seen in listing 3.96.

The command **SDL_GL_SwapBuffers** causes OpenGL to swap buffers. This means that the back buffer – the one that was the target of the image rendering – now becomes the front buffer – the buffer that is currently displayed. Basically, it means that the results of the drawing appear on the screen.

The function **SDL_Delay** sleeps the process for the given amount of miliseconds, in order to limit the usage of resources. It is possible to loop without delay, but the difference wouldn't be visible.

The last command – **glFinish** – is used to synchronize the current process with OpenGL. It ensures that at this point, all OpenGL operations are finished.

glFinish

```
void glFinish ( void );
```

This function blocks until all GL execution is complete

3.4.13. Cleanup

The cleanup code for an OpenGL pixel buffer object can be seen in listing 3.98. This code is valid for both the C and C++ versions of the application. The first operation selects the current buffer using **glBindBuffer**. The next operation – **glDeleteBuffers** – releases the buffer pointed by the variable *pbo*. It is the same as the currently selected buffer, so the result is that it releases the buffer and also reverts to client memory usage for texture loading.

The function that frees the resources used by the shared texture is shown in listings 3.99 and 3.100 for C and C++, respectively.

The first instruction releases the OpenCL memory object. For the C version, it is done by calling the **clReleaseMemObject** function. The C++ version quietly calls

```
void releaseTexture(TextureBufferShared t) {
  clReleaseMemObject(t.cl_tex_buffer);
  deletePixelBufferObject(&t.gl_buffer_id);
  glBindTexture(GL_TEXTURE_2D, 0);
  glDeleteTextures(1, &t.gl_texture_id);
}
```

Listing 3.99: Release of OpenGL and OpenCL objects – the C example

```
void releaseTexture(TextureBufferShared &t) {
  t.cl_tex_buffer = cl::Memory();
  deletePixelBufferObject(&t.gl_buffer_id);
  glBindTexture(GL_TEXTURE_2D, 0);
  glDeleteTextures(1, &t.gl_texture_id);
}
```

Listing 3.100: Release of OpenGL and OpenCL objects – the C++ example

```
void releaseSDL() {
  // release SDL
  SDL_FreeSurface(screen);
  SDL_Quit();
}
```

Listing 3.101: Release of SDL window and library.

this function when the object has no references, so setting the CL buffer to empty causes the release of the memory object.

The next step is to delete the PBO, using the previously described function.

Then the current texture is set to none by the function **glBindTexture**. The texture with ID equal to zero is the default empty texture.

The last step is to free the texture names (the IDs). The function **glDeleteTextures** deletes textures and frees the resources that were used by them.

glDeleteTextures

```
void glDeleteTextures ( GLsizei n, const GLuint * textures );
```

This function deletes named textures.

The cleanup code for OpenCL is basically very similar to that of every other OpenCL application presented in this book. It consists of releasing memory buffers, kernels, command queues and context.

The cleanup code for SDL and release of the OpenGL context can be seen in listing 3.101 for C and C++. The function **SDL_FreeSurface** destroys the window and the OpenGL context. The next command – **SDL_Quit** – releases the resources used by SDL.

SDL_FreeSurface

```
void SDL_FreeSurface ( SDL_Surface* surface );
```

Frees the resources used by a previously created SDL_Surface.

3.4.14. Notes and Further Reading

The whole example for Windows and Linux is available on the attached disk. Note that this method can be used not only to dynamically generate textures. There are very similar methods to generate particle effects, full-screen postprocessing and many other things. It was not presented, but inverse data flow is also possible; for example, texture can be used as a source of OpenCL computations, and the result can be stored in an OpenCL memory buffer.

3.5. Case Study – Genetic Algorithm

Genetic algorithms are a widely used class of search heuristic algorithms that allow for computation of near-optimal solutions for optimization problems. Genetic algorithms belong to the family of evolutionary algorithms.

The example presented here will show how to parallelize a classic genetic algorithm for OpenCL. It will solve the function One-Max, commonly used as an example of an optimization problem. The presented genetic algorithm uses the fine-grained parallel genetic algorithm-style breeding similar to the one presented in book [22].

3.5.1. Historical Notes

Genetic algorithms were developed by J. H. Holland and his colleagues and students at the University of Michigan [23]. Later studies were intensively undertaken by D. E. Goldberg [24]. Genetic algorithms are evolutionary algorithms that are used for finding suboptimal solutions for optimization problems. Genetic algorithms are inspired by natural evolution. Originally they used binary strings as representations of chromosomes; but the modified technique, where chromosome representation is closer to the solution, is also allowed. This more generic approach to this algorithm was proposed by Z. Michalewicz, who called its solution an evolution program [25].

Because of their simplicity and efficiency, these algorithms are widely used. Probably the most popular application of genetic algorithms is in finding the shortest path in real time. This is applied in computer games and robotics. There are also uses in teaching neural networks, optimizing functions and many others.

3.5.2. Terminology

In a genetic algorithm, the element that encodes the candidate solution is called **chromosome** or **genotype**. Traditionally, it was always a binary string. Currently, it is also acceptable to use more complex chromosome representations or even to identify a genotype with a chromosome. The chromosome is not the same as the solution. As in biology, chromosomes encode genetic information but are not the actual organisms. The solution is built on the chromosome. In the classic genetic algorithm, the candidate solution is obtained by calling the **decode** function on the chromosome. In

the literature, the candidate solution is usually called the **phenotype**. For example, the chromosome can be a binary string, but the phenotype based on this chromosome is a path containing a series of $[x, y]$ values.

The **population** refers to the collection of individuals or chromosomes that are evolving. This is also inspired by the evolution. The difference between biological evolution and the genetic algorithm is that in real life, populations can interact with each other. In the algorithm, this is impossible, but it can be mimicked by the appropriate fitness function.

The mechanism that drives the direction of evolution is called the **fitness function** or **objective function**. The fitness function determines the quality of the phenotype (individual based on the chromosome). This function simulates the environment. This is the function that will be optimized – maximized or minimized, depending on the problem.

The **selection** function is a mechanism incorporated into genetic algorithm, that allows reproduction of better – based on fitness – individuals with some probability. Note that it is the mechanism that not always choose the best solutions. With some probability, it can choose weak individuals. This is similar to natural evolution, where the weak specimen can survive and have offspring if it is lucky enough. Selection algorithms are the subject of many studies, but it is too complex a problem to describe here. The matter is described, among others, in [23] or [25].

Two operations are performed on the chromosomes in order to achieve evolution progress – **crossover** and **mutation**. These operations mimic the natural phenomena that happen during animal reproduction and are the main cause of evolution. Crossover allows for joining two chromosomes into a new one that contains some features from both parents. Mutation is the operation that prevents stagnation in evolution by introducing new chromosomes that potentially were not yet present, so it allows the population to maintain diversity.

The **individual** or **specimen** in genetic algorithm is a structure that collects a chromosome, its fitness and possibly other information about the solution candidate. In the classic version presented by Goldberg [24], the individual consisted of chromosome, phenotype, fitness function value, both parents and the point of crossover.

3.5.3. Genetic Algorithm

The genetic algorithm consists of multiple procedures. Every element introduced in the previous subsection has its algorithmic representation.

The global variables for the algorithm are:

- *oldPop* – the current population.
- *newPop* – the offspring of the current population.
- *sumfitness* – the sum of all fitnesses of the current population.
- *popsize* – the number of specimens in each generation.
- *pcross* – probability of the crossover of two specimens.
- *pmutation* – probability of mutation of every gene from the chromosome.

In the original algorithm, there were also some statistical variables. The variables not necessary for the algorithm to work have been omitted.

```
1 struct specimen {
2   chromosome chrom;
3   phenotype x;
4   float fitness;
5 };
```

Listing 3.102: The classical data structure that holds the specimen.

The structure that holds the specimen can be seen in listing 3.102. The *chromosome* holds the list of bits. In the original algorithm, there was also information about the parents and the point of crossover, but it is not necessary for the algorithm to work.

The main part of the algorithm is shown in algorithm 2. Note that the base part of the genetic algorithm is very simple. Line 2 prepares the initial population. It is the list of specimens that are the starting point of optimization. This can be generated randomly or by using some set of predefined chromosomes. This function also calculates the initial statistics for the population. The algorithm iterates until the termination condition is fulfilled. In the example, the termination condition can be seen in line 3 and is a fixed number of iterations. Note that the termination condition can be based on other criteria; one of the most popular is reaching some fitness value or the moment when average fitness would stop growing. The actual work of the algorithm is done in lines 5 and 6. **newGeneration** performs the genetic operations – crossover and mutation. The function **countStatistics** calculates the sum of fitnesses of all specimens – the minimum, maximum and average fitness. The next iteration can use some of these these values during the **newGeneration** function.

Algorithm 2 Genetic Algorithm

```
1: i ← 0
2: oldPop ← initialize()
3: while i < maxgen do
4:    i ← i + 1
5:    newPop ← newGeneration(oldPop)
6:    countStatistics(newPop
7:    oldPop ← newPop
8: end while
```

Algorithm 3 presents the function **newGeneration**. *oldPop* represents the current generation, and *nextPop* is the next generation. This function generates the new generation using crossover and mutation methods. Originally, the mutation was called from inside the crossover function. The version presented here performs the same operation, but the crossover and mutation are clearly separated. This modification is here because it gives a better clue as to how to adapt it to the parallel computational environment.

This function iterates through the whole population and in every iteration selects two specimens to breed. These specimens are then crossed over (see the line 5). There are multiple crossover methods, but the classic approach is to select the point

Algorithm 3 New Generation

```
1: i ← 0
2: while i < popsize do
3:    parent_1 ← select(oldPop, sumfitness)
4:    parent_2 ← select(oldPop, sumfitness)
5:    crossover(oldPop[mate_1], oldPop[mate_2], newPop[i], newPop[i + 1])
6:    newPop[i] ← mutate(newPop[i])
7:    newPop[i].x ← decode(newPop[i].chrom)
8:    newPop[i].fitness ← objfunc(newPop[i].x)
9:    newPop[i + 1] ← mutate(newPop[i + 1])
10:   newPop[i + 1].x ← decode(newPop[i + 1].chrom)
11:   newPop[i + 1].fitness ← objfunc(newPop[i + 1].x)
12:   i ← i + 2
13: end while
```

where the chromosomes will be split and then join them in such a way that both children get chromosome parts from both parents. This method is shown in figure 3.19. This method of crossover is called one point crossover.

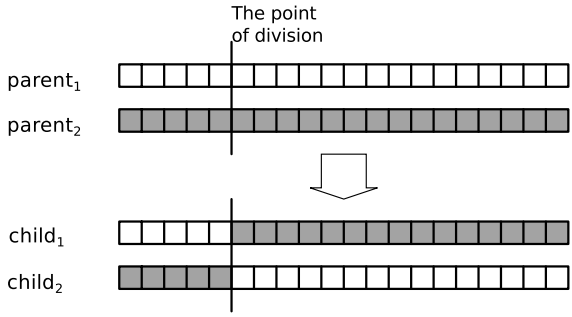

Figure 3.19: One Point Crossover.

The classical mutation flips every bit of the chromosome with some small probability. Because probability is usually very small, the mutation will most likely flip only one bit.

One of the very important elements is the selection function. Convergence of the optimization highly depends on this function. The two most classic approaches to this function are **`roulette wheel`** selection and **`tournament`** selection.

The first one is traditionally the first selection type described. It is based on the idea of a roulette wheel, where the pockets are of the size proportional to the fitness of the specimens. Each pocket is for one specimen. This gives a higher chance of selection for better specimens and a lower probability for the weaker solutions.

The second selection algorithm – tournament – works in the following way. It selects randomly two or more specimens, and then it checks which one of them has a higher fitness value. The better one wins. Note that this type of selection can be performed without knowledge of the global fitness distribution. This allows for straightforward implementation on the parallel system. The pseudocode for this selection

method is presented in algorithm 4. It is the tournament selection with a tournament of size 2.

Algorithm 4 Tournament Selection

```
1: i ← random()  mod popsize
2: j ← random()  mod popsize
3: if oldPop[i].fitness > oldPop[j].fitness then
4:    selected ← oldPop[i]
5: else
6:    selected ← oldPop[j]
7: end if
```

3.5.4. Example Problem Definition

The function that is optimized in this example is very simple. It is the **OneMax** function, which counts the number of 1's in an array of bits. This function is one of the most common examples in prototyping genetic algorithms.

Formally, given a bit array a of length n, the **OneMax** function is defined as follows.

$$\text{OneMax}(a,n) = \sum_{i=0}^{n} a_i \tag{3.9}$$

The problem is to find the array where the sum is maximal. The fitness, or objective, function f for a specimen a of length n is defined as follows.

$$f(a) = \frac{\text{OneMax}(a,n)}{n} \tag{3.10}$$

This function is normalized to be within the range $< 0,1 >$ where 1 is the best possible value.

In the sample application, the chromosomes were constructed as shown in listing 3.103. The bits were implemented as an array of `char`s containing values 1 or 0. The representation of the chromosome is analogous to the one in the classic genetic algorithm. There also was the an array of elements instead of one unsigned integer number.

Note that this representation does not need the **decode** function, because the chromosome is identical to its phenotype.

3.5.5. Genetic Algorithm Implementation Overview

The **genetic algorithm** from now on will be sometimes referred to as **GA** for short. The example program is divided into several files:

```
#define specimenbits 63
typedef struct specimen {
  float fitness;
  char c [specimenbits]; // chromosome
} specimen;
```

Listing 3.103: The definition of the specimen used in the example.

- `oclga.cpp` – The main program. This source code file is responsible for running and configuring the genetic algorithm.
- `GeneticAlgorithm.cpp` and `GeneticAlgorithm.h` – The source code for the host part of the genetic algorithm. These files contain the class `GeneticAlgorithm`.
- `ga.cl` – This is the OpenCL program file that contains the main part of the genetic algorithm. This is also the place that gathers the other three OpenCL program files using the **`#include`** directive.
- `cfg.cl` – The configuration file for the genetic algorithm device code. This file declares where the problem formulation is located – here it is `onemax.cl` – and the configuration of the genetic algorithm – the probability of mutation and crossover.
- `declarations.cl` – This is the OpenCL program that declares global functions. For example, it defines the function **`rand_int`** that is used as a pseudo random number generator.
- `onemax.cl` – This is the problem formulation. It contains fitness function, declaration of data type to store specimen and the crossover and mutation procedures.

Note that construction of the device code should be flexible enough that it can easily implement other problems to solve. The application also uses the common files that contain functions described in the OpenCL Fundamentals chapter.

The host code and the device code fit in 500 lines of code, so the working genetic algorithm can be really compact. Note that this code can be incorporated into some bigger project.

3.5.6. OpenCL Program

The OpenCL program implements all the elements needed for the fully functional genetic algorithm. The GA is executed on the OpenCL device, and the host code is only for steering the execution order of the kernels.

The genetic algorithm elements described in the subsection introducing classical genetic algorithms were parallelized. The chosen selection algorithm is the tournament, because it does not need any global fitness information. This method can easily be parallelized. Almost every element of the GA can be parallelized, except for the main loop, which must be executed sequentially.

```
int rand_int(int *seed) {
  unsigned int ret = 0;
  unsigned int xi = *(unsigned int *)seed;
  unsigned int m = 65537 * 67777;

  xi = (xi * xi) % m;
  *seed = *(unsigned int *)&xi;
  return xi % RAND_MAX_GA;
}
```

Listing 3.104: The function producing pseudo random numbers. The OpenCL C code.

Pseudo Random Number Generator

The first obstacle that the programmer must face is the lack of a pseudo random number generator in the OpenCL C language. It is easy to understand the reason for this fact. The traditional random number generators need some type of counter that will hold the current random seed. This counter is almost always in one instance and is globally available. The random generator gets this counter (the random seed), calculates the pseudo random number and then updates the counter in the way appropriate for the algorithm.

Note that accessing one counter from thousands of threads (work-items) would be bottleneck for the OpenCL program. The accesses to the random seed would have to be serialized, so basically the parallel algorithm would become a sequential algorithm because of the random number generation algorithm.

The solution for this problem is writing some random number generator and using different seeds for different work-items.

The random number generator used in the example is a simple implementation of the Blum Blum Shub pseudo random number generator that was proposed in 1986 by Lenore Blum, Manuel Blum and Michael Shub [26]. Note that any random number generator algorithm can be used in this application. The one used here also uses the pseudo random value generated during the calculation as the next random seed. The source code for the helper function from the OpenCL program can be seen in listing 3.104.

When the kernel is called, the random seeds and values for work-items are generated in the following way: Take the fixed random seed, call it s_0. Assume the kernel is executed. Let the work-item ID be i. The initial random seed for work item i is $s_i^0 = s_0 + i$. Note that the consecutive random seeds for work-item i would be like $s_i^{j+1} = \text{nrs}(s_i^j)$ where $\text{nrs}(x)$ is some value generated in the function from listing 3.104 – the random number. This value is likely different from the returned pseudo random number. The initial random seed for the next kernel execution would be incremented by the size of the NDRange. This mechanism will be clearly visible in kernels that use the pseudo random number generator and functions that enqueue them.

```
void generateSpecimen(global specimen *s, int *random_seed, size_t index) {
  int i;
  for (i = 0; i < specimenbits; i++)
    s [0].c [i] = (int)rand_int(random_seed) % 2;
}
```

Listing 3.105: OpenCL program function that generates one random specimen.

```
kernel void initPopulation(global specimen* newpop, const int size,
                           const int random_seed) {
  const int i = get_global_id(0);
  int seed = random_seed + i;
  generateSpecimen(&newpop [i], &seed, i);
}
```

Listing 3.106: Kernel generating initial population.

Initial Population

The genetic algorithm always starts from some initial population. This first generation of specimens can be generated in various ways, starting with completely random specimens as in the current example or using some intelligent algorithms. The OpenCL program function that generates a random specimen can be seen in listing 3.105. This function is called by the kernel from listing 3.106. The function that generates the specimen is constructed in a way that allows for easy extension for intelligent specimen generation. This is because it also takes the index of the newly generated specimen, so it can use, for example, some predefined constant array of specimens.

The **initPopulation** kernel that fills the population with newly created specimens can be seen in listing 3.106. Note the usage of *random_seed*. Every kernel instance has its own, individual *seed* that is used for generating pseudo random numbers.

Selection

The selection mechanism used in the example is the classical tournament selection algorithm with tournament of size 2. The listing 3.107 shows the program function that is called by the kernel to choose one specimen. Note that tournament selection gives less evolutionary pressure than roulette rule selection, the most widely used in literature. This is a virtue or a vice, depending on the problem type. In the simple example computing the best value of a OneMax function, this selection method performs well. The returned value is the index of the specimen in the population.

Note that it is really unlikely for this selection to select the best specimen in one shot. This function just selects the better of the two randomly chosen specimens. In the algorithm, where this function is very often called, it is possible to eventually reach the best solution using this selection method. This method performs very well

```
int selectSpecimen(global specimen *pop, int size, int *random_seed) {
  int i, j;
  i = rand_int(random_seed) % size;
  j = (rand_int(random_seed) % (size - 1) + i + 1) % size;
  if (pop [i].fitness > pop [j].fitness) return i;
  return j;
}
```

Listing 3.107: Tournament selection algorithm

```
void crossover(specimen *parent, specimen *offspring, int *random_seed) {
  int i;
  int cpoint = rand_int(random_seed) % specimenbits;
  for (i = 0; i < specimenbits; i++) {
    int part = (i < cpoint) ? 1 : 0;
    offspring [0].c [i] = parent [part].c [i];
    offspring [1].c [i] = parent [1 - part].c [i];
  }
  offspring [0].fitness = 0;
  offspring [1].fitness = 0;
}
```

Listing 3.108: The crossover function used in the example.

for functions that contain multiple local optima. The GA using this selection does not stop when it finds the place where the derivative is zero.

Crossover

There are many types of crossover operations. The classical approach is to divide both parents in some randomly chosen locus and then join them interchangeably. It has been described in the introduction to genetic algorithms in previous subsections. This is the one used in the example. The OpenCL program function performing this operation can be seen in listing 3.108.

Recall figure 3.19. The crossover point is selected randomly and is called *cpoint*. Note that if the crossover point is 0, then the new specimens are just copies of the parents.

Mutation

The mutation function is shown in listing 3.109. This algorithm is the same as the one used in the original genetic algorithm. This function iterates through all genes of the chromosome and flips each of them with some fixed probability. This probability should be small enough that the mutation won't destroy the solution candidate.

```
void mutate(specimen *parent, int *random_seed) {
  int i;
  for (i = 0; i < specimenbits; i++) {
    if (rand_float(random_seed) < pmutation) {
      parent->c [i] = 1 - parent->c [i];
    }
  }
}
```

Listing 3.109: The mutation algorithm used in the example.

```
kernel void newGeneration(global specimen* pop, global specimen* newpop,
                      const int size,
                      const int random_seed) {
  const int i = get_global_id(0) * 2;
  specimen parent [2], offspring [2];
  int seed = random_seed + i;

  parent [0] = pop [selectSpecimen(pop, size, &seed)];
  parent [1] = pop [selectSpecimen(pop, size, &seed)];

  if (rand_float(&seed) < pcross) {
    crossover(parent, offspring, &seed);
  } else {
    offspring [0] = parent [0];
    offspring [1] = parent [1];
  }

  mutate(&offspring [0], &seed);
  mutate(&offspring [1], &seed);
  newpop [i] = offspring [0];
  newpop [i + 1] = offspring [1];
}
```

Listing 3.110: The kernel function performing the *new generation* phase of the Genetic Algorithm.

newGeneration Kernel

The crossover, mutation and selection functions are now described, so it is time to join them together to form the function that creates a new generation. This function can be seen in listing 3.110. The kernel function **newGeneration** performs the same algorithm as the one from the classical GA, except it does not count fitness values. This has been excluded from this function and put into a separate kernel described later. The reason for that is that it allows for calculating fitness in a convenient way, even outside the GA loop – for example, during initialization.

The **newGeneration** kernel is constructed in such a way that it works only for even sizes of population. This is because every kernel instance selects two individuals for genetic operations and also produces two new specimens. This simplification is

```
1 float fitness(const specimen *sp) {
2   int s = 0, i = 0;
3   for (i = 0; i < specimenbits; i++) {
4     s += (int)(sp->c [i]);
5   }
6   float r = ((float)s) / (float)specimenbits;
7   return r;
8 }
```

Listing 3.111: The fitness function implementation

made in order to optimize kernel execution times.

The first action that this kernel does is to calculate a proper random seed. This is done in line 6. The *random_seed* passed by the parameter from the host program is increased by the global ID of the work-item. This operation makes an individual random seed for every thread. The different random seed for every work-item is important, because it allows for different heuristic paths of work-item computations. If this element were skipped, then every work-item would follow the same path and the power of the Genetic Algorithm would be lost.

The next step, almost identical to the one from the original genetic algorithm, is the selection. This is done in lines 8 and 9 by calling **selectSpecimen**. The program selects two parents. Note that it takes the *seed* parameter that is the individual random seed for the work-item.

The fragment in lines 11 . . . 16 is responsible for the first genetic operation – the crossover. This operation is performed with some probability. The helper function **rand_float** generates a pseudo random number from range $[0, 1)$. If the drawn value is less than the probability of crossover, then the two parents are crossed over in order to produce two children (offspring). In the other case, the offspring is just a copy of the parents. The offspring is then mutated, using the **mutate** helper function. This can be seen in lines 18 and 19. The last step – lines 20 and 21 – is to insert the new specimens into the new population.

The difference between the original genetic algorithm and the version implemented here is that in the procedure **newGeneration**, fitness is not calculated. The overall algorithm does logically the same operations as the original, but it is divided somewhat differently into functions.

Note that most of the steps uses the random seed intensively. Thanks to local seeds, every work-item can calculate independently, so there is no synchronization between work-items. This parallelizes very well.

Fitness Function

Recall the OneMax problem description from subsection 3.5.4. The fitness function was defined as $f(a) = \frac{\text{OneMax}(a,n)}{n}$. The implementation of this function is shown in listing 3.111. This function takes the chromosome that is basically the same as the phenotype and calculates the sum of all 1's in the binary string. The value is then normalized to be between 0 and 1. Note that this function is trivial and it is here only

```
1 kernel void countFitness(global specimen* pop,
2                          const int size) {
3   const int i = get_global_id(0);
4   const specimen sp = pop [i]; // kopiujemy do lokalnej!
5   pop [i].fitness = fitness(&sp);
6 }
```

Listing 3.112: The kernel calculating fitness function for every specimen in the given population.

for explanatory reasons. The real uses of genetic algorithms are more complex, and the fitness calculation can take most of the algorithm time.

The fitness function is called from the kernel that calculates fitnesses for every specimen. This is easily parallelizable because the fitness function for one specimen is independent of that for other specimens. The source code for the kernel that calculates fitness for all of the specimens can be seen in listing 3.112.

Summary Fitness

At the end of the genetic algorithm, the population consists of specimens that represent suboptimal solutions or even optimal solutions. The application needs to get the best specimen from the current population and present it as a result. Another important piece of information about the population is the summary fitness of the specimens. The kernel that finds the best specimen and calculates the summary fitness can be seen in listing 3.113.

This kernel is designed to work in only one work-group. It can work on populations of any size, but it is most efficient for the ones that have the same size as the work-group.

This kernel is divided into two different stages. In the first stage (lines 10...21), every work-item calculates the partial result for the $\frac{n}{g}$ or $\frac{n}{g}-1$ elements, where n is the number of specimens in a population and the g is the size of the workgroup. Every work-item combines values from population elements at indexes $(l, l+1\cdot g, l+2\cdot g, \dots)$ where l is the local ID of the work-item. This stage allows for calculation on bigger populations than the maximal size of the work-group using only one kernel invocation. An image that explains this part is depicted in figure 3.20. The result of this part is the local best and the summary fitness for the work-item stored in *bestLocal*[*rank*] and *sumfitnessLocal*[*rank*] (see lines 22 and 23). There is a synchronization point at the end of this stage, implemented using the **barrier** function in line 24. The synchronization ensures that every operation on the local memory is committed.

The second part is based on the scan algorithm, which was described in section 2.11.7. The adaptation of this algorithm for the purpose of gathering statistics from a population can be seen in lines 26...35 of listing 3.113. It calculates summary fitness and finds the best specimen in parallel. This version of the scan algorithm operates on indexes instead of the `specimen` objects, because this solution involves fewer memory operations. The **for** loop visible in line 26 sets the consecutive steps

```
1  kernel void countSummaryFitnessP(global specimen* pop, global specimen* best,
2                                   global float *sumfitness,
3                                   const int size,
4                                   local float *sumfitnessLocal,
5                                   local int *bestLocal) {
6    const int wgsize = get_local_size(0), rank = get_global_id(0);
7    int j, bestIndex = 0, currStep;
8    float bestFitness, sumFitnessPrv = 0, f;
9    // gathering values that are located at indexes beyond work-group size
10   if (rank < size) {
11     bestFitness = sumFitnessPrv = pop [rank].fitness;
12     bestIndex = rank;
13     for (j = rank + wgsize; j < size; j += wgsize) {
14       f = pop [j].fitness;
15       sumFitnessPrv += f;
16       if (bestFitness < f) {
17         bestIndex = j;
18         bestFitness = f;
19       }
20     }
21   }
22   bestLocal [rank] = bestIndex;
23   sumfitnessLocal [rank] = sumFitnessPrv;
24   barrier(CLK_LOCAL_MEM_FENCE);
25   // the scan algorithm
26   for (currStep = 1; currStep < wgsize; currStep <<= 1) {
27     if ((rank % (currStep << 1) == 0) && ((rank + currStep) < wgsize)) {
28       sumfitnessLocal [rank] += sumfitnessLocal [rank + currStep];
29       if (pop [bestLocal [rank]].fitness <
30           pop [bestLocal [rank + currStep]].fitness) {
31         bestLocal [rank] = bestLocal [rank + currStep];
32       }
33     }
34     barrier(CLK_LOCAL_MEM_FENCE);
35   }
36   if (rank == 0) {
37     sumfitness [0] = sumfitnessLocal [0];
38     best [0] = pop [bestLocal [0]];
39   }
40 }
```

Listing 3.113: The function that calculates the sum of fitnesses in parallel.

for summation that are powers of 2. This is because work-items involved in every step always perform summation and comparison of two elements – the element at the index *rank* and the element at the index *rank+currStep*.

Note the usage of barriers in lines 25 and 34 of listing 3.113. Especially the second one is worth mentioning, because it is inside the loop. The OpenCL programmer must be very careful when using barriers inside a loop, because every work-item in a work-group must execute the **barrier** function. The loop in lines 26...35 is an example of such a loop. It's number of iterations is independent of the work-item ID, so every work-item will execute the **barrier** function exactly the same number of

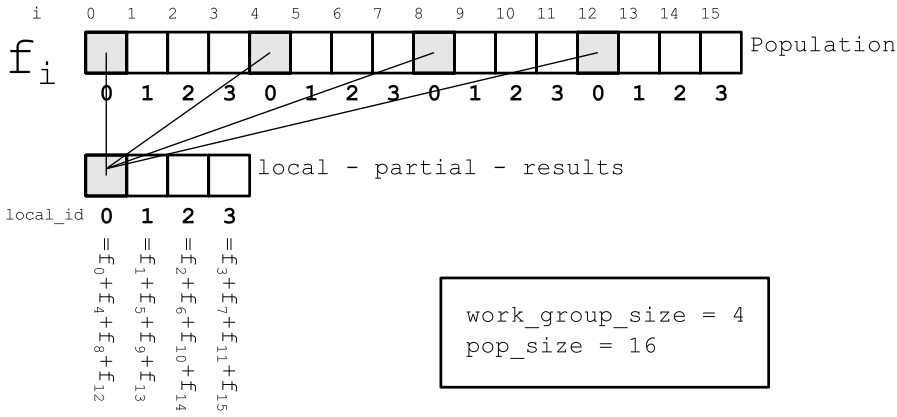

Figure 3.20: The graphical explanation of the first part of the kernel calculating summary fitness and finding the best specimen.

times (line 34). The operations that are different for different work-items are inside the **for** loop. Some work-items will eventually perform empty loops, just to fulfill the requirement of execution of **barrier**, depending on the condition in line 27.

3.5.7. Most Important Elements of Host Code

The example genetic algorithm displays the result of the calculation and some basic performance statistics. This example calculates the solution for a OneMax problem.

The main part of the code is organized as shown in algorithm 5. The actual source code also contains functions for counting execution time from the host perspective. These functions are not necessary for understanding the code.

Algorithm 5 The Main Function

```
1: parseCommandLine()
2: ga ← newGeneticAlgorithmobject
3: ga.initializePopulation()
4: generation ← 0
5: while generation < iterations do
6:    ga.iterate()
7:    generation ← generation + 1
8: end while
9: ga.getResult()
10: printresults
11: printtimes
```

The function from algorithm 5, line 1, checks the command line options. This function is very simple, so its detailed description can be omitted. The sample application on the attached disk will provide information about command line options when called with *-h* or *-help*.

The important part is the initialization of the GA object. This object encapsulates procedures for genetic algorithm initialization and also initializes OpenCL objects.

```
GeneticAlgorithm ga = GeneticAlgorithm(config.popSize, config.vendorNum,
  config.deviceNum);
ga.setRandomSeed(19881202);
```

Listing 3.114: Calling the constructor of the genetic algorithm.

```
ga.initializePopulation();
```

Listing 3.115: Setting up the initial population.

```
for (generation = 0; generation < config.iterations; generation++) {
  ga.iterate();
}
```

Listing 3.116: The main loop of the Genetic Algorithm.

The actual initialization is called as shown in listing 3.114. Note also the setting of the random seed for some fixed value. The important problem of the random number generator is discussed in subsection 3.5.6. The constructor of the `GeneticAlgorithm` object takes the index of the the platform and the index of the device on the platform. The fact that it can be run on different platforms allows for comparing them. *popSize* defines the size of the population that will evolve during the GA run time. The details about initialization will be described in more detail in the later subsections.

The calling of the creation of the initial population is shown in listing 3.115. Note that fragments from listings 3.114 and 3.115 correspond to the first step of the genetic algorithm, as shown in the algorithm in subsection 3.5.3.

The main loop of the GA is shown in listing 3.116. The termination condition is the fixed number of iterations. The method **`iterate`** encapsulates the creation of new population and counting statistics.

The elements that are left do not demand any additional description because of its simplicity. The results are obtained by calling the kernel that prepares the result; then it is copied to the host memory and displayed on the console.

Genetic Algorithm Class

The example application is written in C++. This allows for usage of classes. The declaration of the genetic algorithm class can be seen in listing 3.117. The full declaration listing is provided here, because it is necessary to introduce variables that will be used in later subsections.

The variable *random_seed* holds the current seed for the random number generator. Note that the genetic algorithm needs some pseudo random number generator, because it is an heuristic algorithm and OpenCL does not provide any such function.

context, *program* and *queue* are the variables that the reader should already be familiar with.

pop_size stores the count of the specimens for every population. *pop_buf_size* and *specimen_size* determine the sizes used in initialization and usage of buffers.

The buffer names are probably self-explanatory. The variable that deserves more attention is *sumfitness_dev*. Summary fitness is the value calculated as the sum of every individual fitness. It is important information that gives a hint about overall population quality. If roulette wheel selection would be used, this information would also be necessary in every algorithm iteration.

Note the introduction of the new class *cl::KernelFunctor*. This is specific to C++ OpenCL API. This is a wrapper that allows for queuing a kernel like the ordinary C or C++ function. This class allows for OpenCL operation in almost the same way as the CUDA. Usage and initialization of this class objects will be described later on.

The private methods are called from the **GeneticAlgorithm** constructor and are responsible for stages of algorithm initialization. Most of this class interface reflects GA stages and procedures.

Genetic Algorithm Initialization

The **GeneticAlgorithm** constructor initializes the OpenCL objects *context*, *queue* and *program*. It also prepares kernels configures GA options and buffers.

Initialization of OpenCL objects is basically the same as in chapter 2. It creates an in-order command queue and an ordinary context. The main difference is that it uses a different the compilation parameter than `NULL`. The build command executed during initialization can be seen in listing 3.118

Note the third parameter – "-I./ga". This parameter is passed to the function **cl::Program::build** as compilation options. The "-I" switch selects the directory from which the source files will be loaded. This allows for usage of the **#include** preprocessor directive from inside of the OpenCL program. In genetic algorithm implementation, the only file that is loaded directly from the host code is `ga/ga.cl` the other program files are loaded by the OpenCL compiler.

OpenCL allows for running kernels in a similar fashion to that done in CUDA. The object that allows this behavior is called `cl::KernelFunctor`. The initialization of functors and kernels can be seen in listing 3.119. Note that the **cl::KernelFunctor** constructor takes the kernel, command queue and full information about the **NDRange** space of execution. This allows enqueuing of kernels like the C or C++ functions. This feature of OpenCL C++ API was not introduced before because it is present only in the C++ version, not in base API. It also has some limitations. For example, it is impossible to properly allocate local memory buffers using this method[1]. The usages of `cl::KernelFunctor`s will be shown later in this section.

The genetic algorithm can solve a wide range of different computational problems. This implies that data structures may have different memory requirements. This example is constructed in such a way that it would not be necessary to have two copies of the definition of specimen structure – one for the host code and one for the device code. `specimen` is defined in the OpenCL program, and there is also a kernel

[1] Th OpenCL version 1.1 `cl::KernelFunctor` objects do not support local memory buffers.

```
class GeneticAlgorithm {
protected:
int random_seed;

cl::Context context;
cl::Program program;
cl::CommandQueue queue;

cl_int pop_size;
size_t pop_buf_size;
size_t specimen_size;

cl::Buffer population_dev;
cl::Buffer population_new_dev;
cl::Buffer current_best_dev;
cl::Buffer sumfitness_dev;

cl::KernelFunctor initPopulationFunct;
cl::KernelFunctor countFitnessFunct;
cl::Kernel countSummaryFitnessP;
cl::KernelFunctor newGenerationFunct;
cl::KernelFunctor specimenToStringConvFunct;

cl_float sumfitness;

void initializeOpenCL (int splatform, int deviceNum);
void initializeKernels ();
void initializeOptions ();
void initializeBuffers ();

void summaryFitnessCalculation (cl::Buffer &pop);

public:

GeneticAlgorithm(int populationSize, int splatform, int deviceNum);
virtual ~GeneticAlgorithm();
void initializePopulation ();
void iterate ();
void getResult (char *result, int resultSize, float *fitness, int index = -1);
float getSummaryFitness ();
void synchronize ();
void setRandomSeed (int rs);
};
```

Listing 3.117: The declaration of the GeneticAlgirithm class.

```
program.build(context.getInfo<CL_CONTEXT_DEVICES > (), "-I./ga");
```

Listing 3.118: The building of an OpenCL program for the Genetic Algorithm.

```
void GeneticAlgorithm::initializeKernels() {
  initPopulationFunct = cl::KernelFunctor(cl::Kernel(program,
      "initPopulation"), queue, cl::NullRange, cl::NDRange(
      pop_size), cl::NullRange);
  countFitnessFunct = cl::KernelFunctor(cl::Kernel(program,
      "countFitness"), queue, cl::NullRange, cl::NDRange(pop_size), cl::NullRange);
  countSummaryFitnessP = cl::Kernel(program, "countSummaryFitnessP");
  newGenerationFunct = cl::KernelFunctor(cl::Kernel(program,
      "newGeneration"), queue, cl::NullRange, cl::NDRange(
      pop_size / 2), cl::NullRange);
  specimenToStringConvFunct =
    cl::KernelFunctor(cl::Kernel(program,
        "specimenToStringConv"), queue, cl::NullRange, cl::NDRange(
        1), cl::NullRange);
}
```

Listing 3.119: The initialization of kernel functors and kernel objects.

```
kernel void getBufSizes(global int *sizes, int specimens) {
  sizes [0] = getPopulationBufSize(specimens);
}
```

Listing 3.120: Kernel that returns buffer sizes. It can easily be extended to return other problem-specific features. The OpenCL program code.

```
int getPopulationBufSize(int specimensN) {
  return sizeof(specimen) * specimensN;
}
```

Listing 3.121: The OpenCL program function that returns the memory size for the population. The OpenCL program code.

that returns the size of this structure back to the host program. This simple trick improves application flexibility. The kernel and the helper function that returns the memory requirements are in listings 3.121 and 3.120, and the initialization of the genetic algorithm, along with getting the sizes of data structures from the kernel, can be seen in listing 3.122. Note the usage of `cl::KernelFunctor` **`getBufSizesFunct`**. It is used as if it were an ordinary C function. It is a very useful feature.

OpenCL buffers are initialized in the usual way, using the data obtained from OpenCL implementation of the GA. The listing of function that is responsible for allocating these buffers can be seen in listing 3.123.

The second stage of GA initialization is generation of the initial population. This is in a separate method, because it could be possible to run one genetic algorithm multiple times, starting with different initial populations. The initial population is created using the **`initPopulation`** kernel represented in the host code as

```
1 // genetic algorithm initialization and configuration
2 void GeneticAlgorithm::initializeOptions() {
3   cl::KernelFunctor getBufSizesFunct;
4   getBufSizesFunct = cl::KernelFunctor(cl::Kernel(program,
5       "getBufSizes"), queue, cl::NullRange, cl::NDRange(1), cl::NullRange);
6   cl_int buf_sizes [2];
7   cl::Buffer buf_sizes_dev(context, CL_MEM_READ_WRITE, sizeof(cl_int));
8   getBufSizesFunct(buf_sizes_dev, pop_size);
9   queue.enqueueReadBuffer(buf_sizes_dev, CL_TRUE, 0, sizeof(cl_int), buf_sizes);
10  pop_buf_size = buf_sizes [0];
11  specimen_size = pop_buf_size / pop_size;
12 }
```

Listing 3.122: Initialization of Genetic Algorithm options. The host code.

```
1 void GeneticAlgorithm::initializeBuffers() {
2   population_dev = cl::Buffer(context, CL_MEM_READ_WRITE, pop_buf_size);
3   population_new_dev = cl::Buffer(context, CL_MEM_READ_WRITE, pop_buf_size);
4   current_best_dev = cl::Buffer(context, CL_MEM_READ_WRITE, specimen_size);
5   sumfitness_dev = cl::Buffer(context, CL_MEM_READ_WRITE, sizeof(cl_float));
6 }
```

Listing 3.123: The initialization of OpenCL buffers that are used by the genetic algorithm

```
1 void GeneticAlgorithm::initializePopulation() {
2   initPopulationFunct(population_dev, pop_size, random_seed);
3   random_seed += pop_size;
4   countFitnessFunct(population_dev, pop_size);
5   sumfitness = -1;
6 }
```

Listing 3.124: Method for creating initial population.

countFitnessFunct `cl::KernelFunctor`. The function that executes this kernel can be seen in listing 3.124

New Generation Function

The method **GeneticAlgorithm::iterate** that performs the new generation function can be seen in listing 3.125. This function first enqueues the **newGeneration** kernel that was described before. This operation can be seen in line 3 of discussed the listing. `cl::KernelFunctor` overloads "(" and ")" so the kernel can be enqueued just like an ordinary function. The function-like kernel queuing can be seen in lines 3 and 5 of this listing. Also please note the way *random_seed* is updated in line 4. It is increased by the total number of work-items in the global work space after kernel enqueue operation.

```
1  void GeneticAlgorithm::iterate() {
2    cl::Buffer tmp;
3    newGenerationFunct(population_dev, population_new_dev, pop_size, random_seed);
4    random_seed = random_seed + pop_size;
5    countFitnessFunct(population_new_dev, pop_size);
6    sumfitness = -1;
7    // we reuse buffers, so we do not need any unnecessary memory transfers
8    tmp = population_new_dev;
9    population_new_dev = population_dev;
10   population_dev = tmp;
11 }
```

Listing 3.125: The function that generates a new population and executes everything inside the GA loop. The host code.

The next step is to enqueue the **countFitness** kernel. This is also done using the kernel functor, so it takes only one line instead of three. The functors are not present in C API, so they are introduced here as specific to OpenCL C++ API. Remember that the functor must be initialized with the proper NDRanges. The kernel **countFitness** calculates the fitness of every specimen in the new population. Note that in this stage, the variable *sumfitness* is set to -1, which means the joint fitness of all the specimens from the population is yet to be computed.

The last step is to swap the old population (previous generation) with the new population. This is done in lines marked by $8 \ldots 10$. Note that there are no memory transfers involved. The only thing that happens here is switching the handlers to memory objects. This is very efficient, even for enormously big populations.

Summation of Fitnesses

The summary fitness and finding the best specimen kernel are invoked from the method **summaryFitnessCalculation** which can be seen in listing 3.126. This function checks if it is necessary to calculate the summary fitness again before invoking the kernel (see line 3 of discussed listing). Only when the fitness must be updated, it enqueues the **countSummaryFitnessP** kernel, which calculates the sum of fitnesses for all individuals and also finds the best specimen (line 11). Note that this kernel is not enqueued using a functor. It is enqueued in the usual way – by setting arguments and using the **cl::CommandQueue::enqueueNDRangeKernel**. This is because the kernel uses local memory. The functors do not support setting the parameters for thekernel that determine the size of the local memory buffers. Setting arguments that cannot be passed to the kernel functor can be seen in lines 9 and 10 of this listing.

After thecalculation is finished, the **summaryFitnessCalculation** method reads the results buffer and saves it into host memory: this operation can be seen in line 14 of the discussed listing. This method is blocking, so it is also one of the very few synchronization points in this application. The method that obtains the summary fitness can be seen in lines $18 \ldots 21$ of listing 3.126. This method is called **getSummaryFitness**. The rest of the application that uses the GeneticAlgorithm

```
void GeneticAlgorithm::summaryFitnessCalculation(cl::Buffer &pop) {
  size_t wgs;
  if (sumfitness < 0) {
    wgs = countSummaryFitnessP.getWorkGroupInfo<CL_KERNEL_WORK_GROUP_SIZE > (
      queue.getInfo<CL_QUEUE_DEVICE > ());
    countSummaryFitnessP.setArg(0, pop);
    countSummaryFitnessP.setArg(1, current_best_dev);
    countSummaryFitnessP.setArg(2, sumfitness_dev);
    countSummaryFitnessP.setArg(3, pop_size);
    countSummaryFitnessP.setArg(4, sizeof(cl_float) * wgs, NULL); // this is the
        reason of not using functor
    countSummaryFitnessP.setArg(5, sizeof(cl_int) * wgs, NULL);
    queue.enqueueNDRangeKernel(countSummaryFitnessP, cl::NullRange,
      cl::NDRange(wgs), cl::NDRange(wgs));
    queue.enqueueReadBuffer(sumfitness_dev, CL_TRUE, 0, sizeof(cl_float),
      &sumfitness);
  }
}
float GeneticAlgorithm::getSummaryFitness() {
  summaryFitnessCalculation(population_dev);
  return sumfitness;
}
```

Listing 3.126: Finding the best specimen and calculating the summary fitness for the current population. The host code.

class treats this function as an interface for obtaining statistical data about population.

3.5.8. Summary

The example presented shows some of the OpenCL C++ API features that are not present in the C version of the specification, especially `cl::KernelFunctionr`, which allows for a convenient kernel enqueue. Also presented was usage of the build parameters for an OpenCL program. The example can easily be extended or incorporated into some bigger application.

Note that the synchronization functions are completely avoided. This is considered a good practice in this kind of application.

The parallel genetic algorithm turns out to be as compact as the classical Goldberg's version.

The fully functional version of this example is available on the attached disk.

3.5.9. Experiment Results

In order to check the performance of the parallel genetic algorithm implemented in OpenCL, some experiments have been performed.

The first one was to check if it really finds the best or close-to-best solution. The algorithm was executed for a different size of the population and for a constant iteration count. The iteration count was fixed at the value of 60. The size of the

population was increasing from 2 to 256 with step of 2. The results are shown in figure 3.21. This figure shows the fitness value of the best specimen in function of population size.

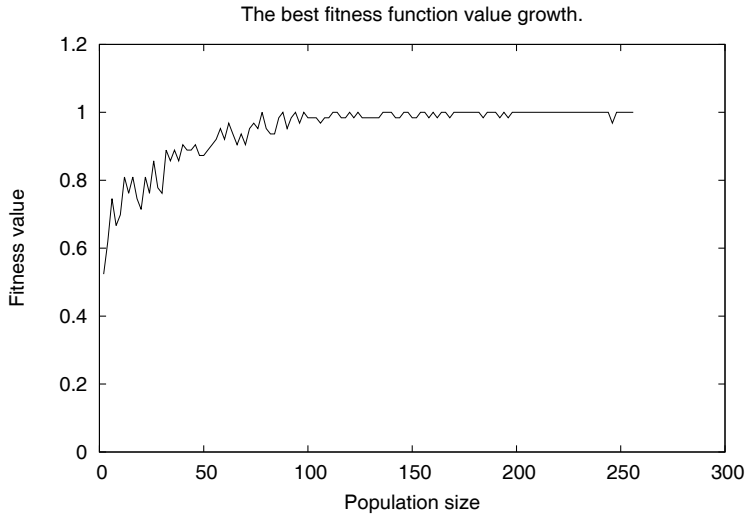

Figure 3.21: The fitness values of the best specimen for different population sizes.

The value of 1 is the maximum that the fitness function can reach. The fitness value grows fast when increasing the population size and after reaching a population size 70, it slows down to reach almost constant optimal value at the population size of 200.

This shows that when increasing the number of specimens, the solution quality is also growing. Note that for a OneMax problem, it is sufficient to use a population of size 200 and 60 iterations to find the best possible solution. It is a very simple function, so it can reach optimum. In real-world applications, a situation where the GA finds an optimal solution is not always the case.

The next step was to compare the performance of different hardware and software configurations. The first objective was to check if the size of the NDRange impacts the computation time linearly or not. The experiment was performed for big populations where the computation time was significant. It was also an experiment in which population size was increasing with constant iteration count. The population size increased from 8 to 4096 with a step of 8. The experiment was performed on a GeForce GTS 250 GPU. It appeared that the computation time was taking various values, depending on the size of the problem, but it wasn't linear. The interesting part of this experiment is depicted in figure 3.22.

Note that the computation times were oscillating between about 0.04 and 0.18 when the population size was increasing. Another interesting fact was that the peaks of good performance were in constant distances. It was visible for the

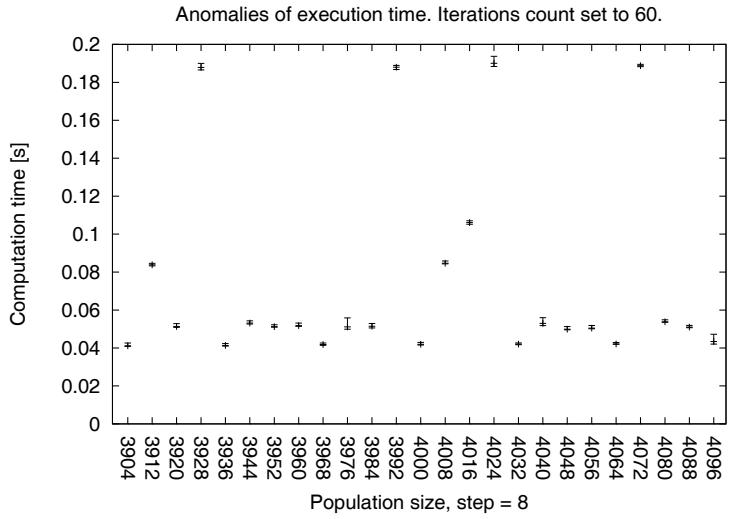

Figure 3.22: Anomalies concerning the computation time for different problem sizes.

whole results set; in the figure, which shows only part of the experiment results, it can also be observed. The peak performances are for population sizes of $3904, 3936, 3968, 4000, \ldots$. It can easily be calculated that it is always 32. This is because of the internal construction of the GPU. The work-items are grouped together, and using only some part of the group is computationally very expensive.

The last experiment was to actually compare different hardware and platforms. The experiment took into account that the size of the population must be a multiple of 32. The experiment was performed 5 times for every population size, and the minimal execution times are shown in figure 3.23.

Note that OpenCL scales very well – on every CPU implementation, execution time grew linearly. On the GPU, the computation time contains some constant factor (because of the queue latencies). The algorithm running on the GPU eventually reaches and outperforms that of the CPU. Note that on the Tesla GPU, the computation times were periodically increasing and decreasing. This is because this processor is intended to be used in computationally very intensive calculations. The high and low performance is due to the fact that different processing units are turned on and some of them are not fully populated by work-items. This case is similar to the one shown in figure 3.22, but on a larger scale.

The experiment can be extended by the reader to use a profiler from OpenCL. The system timer was used here intentionally – it does not add any code to the genetic algorithm implementation, so it is as simple as possible. The differences between the system timer and the profiler are so small that it is not necessary to use it in this example. Please refer to section 2.8.4 for more details.

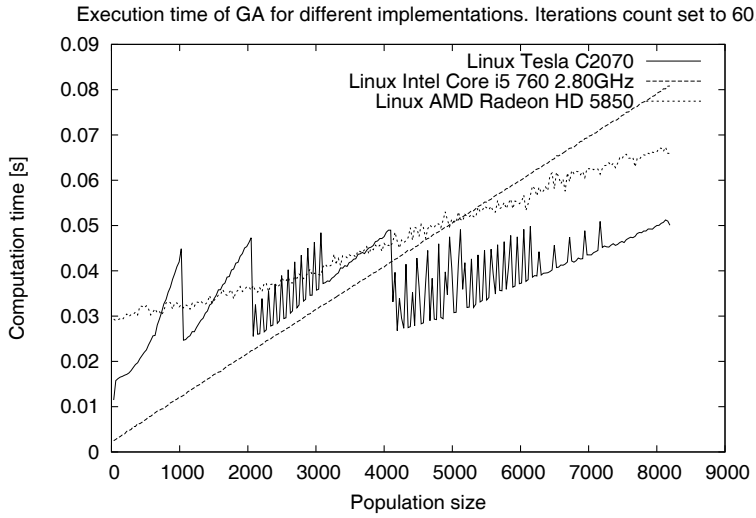

Figure 3.23: Comparison of execution time of genetic algorithm on different hardware.

Experiment Configuration

The experiment was performed on two different machines. The configuration of the first machine was:

- AMD Radeon HD 5850
- Intel Core i5 760 x4 2.9GHz – family 6, model 30, stepping 5
- Linux Mint 10 32bit with updates for date 2011-09-16
- AMD APP SDK 2.4

The configuration of the second machine was:

- Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz
- NVIDIA Tesla C2070
- Linux Ubuntu 11.04 with updates for date 2011-09-16
- OpenCL 1.1 CUDA 4.0.1

Appendix A

Comparing CUDA with OpenCL

A.1. Introduction to CUDA

A.1.1. Short CUDA Overview

CUDA is an OpenCL competitor and predecessor. It was developed by NVIDIA (`http://nvidia.com`, CEO: Jen-Hsun Huang). The first version of CUDA was unveiled in 2006, along with a new GeForce 8800 graphics card. The fourth stable version was released in May 2011. CUDA is a parallel computing architecture designed as a set of tools and programming APIs. It makes GPU programming possible with the limited set of language extensions. The CPU and GPU are treated as separate devices and can therefore be programmed independently. This allows incremental programming and gradual parallelization of any old sequential code. The typical CUDA installation contains NVIDIA GPU Computing SDK, a driver appropriate for the GPU used, and a standard set of compilers and development tools from third parties. CUDA is designed with portability in mind and can be applied to NVIDIA devices working on Windows, Linux, Mac or some mobile platforms. The learning curve, documentation and developer support are very good. However, in comparison to OpenCL, CUDA is limited to the devices from one company only.

A.1.2. CUDA 4.0 Release and Compatibility

The current stable version of CUDA 4.0 works naturally only on machines equipped with NVIDIA GPU accelerators. The oldest and first CUDA-capable is the GeForce 8800, released in 2006. Nowadays, there are many CUDA-ready devices. The 4.0 release has several exciting features and improvements. Details can be found in the NVIDIA documentation attached to the 4.0 release. Some of the new possibilities include sharing GPUs across multiple threads, no-copy pinning of system memory and unified virtual adressing. There are improved developer tools, like the possiblity of C++ debugging, a new version of the `cuda-gdb` debugger for Mac users, the NVIDIA Visual Profiler for C/C++ and a binary disassembler, to name a few. The most important files of internal content of the CUDA release are shown in the list below [27].

- nvcc: The command line compiler, which wraps compilation to different agents according to language type and destination platform. It will be called simply the "CUDA compiler" in the book. More detailed information about nvcc can be found in CUDA documentation, especially in nvccCompilerInfo.pdf, released by NVIDIA. nvcc accepts some standard compiler options and can be used in some cases as a drop-in-replacement for the whole build process, but it still requires regular system tools to work. For example, for MS Visual Studio compiler, it can translate its options into "cl" commands. The source code is split into host and GPU parts and compiled independently, using system compilers for host code and proprietary compilers/assemblers for GPU code. The whole process is done automatically, but developers can perform some of the stages on their own, using a number of nvcc options.

- include/* Essential files of the whole API and developers' utilities:
  - cuda.h CUDA driver, the main API handler,
  - cudaGL.h, cudaVDPAU.h, cuda_gl_interop.h, cuda_vdpau_interop.h are for Linux developers and contain a toolkit for OpenGL, as well as both driver and toolkit for VDPAU,
  - cudaD3D9.h, cudaD3D10.h, cudaD3D11.h contain CUDA DirectX 9-11 headers for Windows,
  - cufft.h, cublas.h, cusparse.h, curand.h, curand_kernel.h, thrust/*, npp.h contain headers for APIs of computational libraries: CUFFT (Fast Fourier Transform), CUBLAS (Linear Algebra), CUSPARSE (Sparse Algebra), CURAND (Random Number Generators, API and device), Thrust (C++ library) and NPP (NVIDIA Performance Primitives),
  - nvcuvid.h, cuviddec.h, NV*.h, IN*.h video decoding headers for Windows and/or Linux (C-libraries for encoders and decoders for DirectDraw and DirectShow).
- Windows library files: cuda.lib, cudart.lib, cublas.lib, cusparse.lib, curand.lib, npp.lib, nvcuvenc.lib, nvcuvid.lib,
- Linux library files: libcuda.so, libcudart.so, libcublas.so, libcufft.so, libcusparse.so, libcurand.so, libnpp.so
- Mac OS X library files: libcuda, libcudart, libcublas, libcufft, libcusparse, libcurand, libnpp, all with the extension .dylib.

The support for the hardware and the software is quite good. Table A.1 shows the most important operating systems and compilers which can be used for CUDA programming. The columns with the numbers 32 and 64 mean the type of system architecture: 32 or 64 bit.

Unfortunately, right now (2011) there is no support for the gcc-4.5 compiler, although it is able to compile nearly all example SDK programs. In order to use gcc-4.5, one must manually edit host_config.h file, which usually resides at the /usr/local/cuda/include path. There is a section in this file which forces gcc to exit if its minor version is not below 5:

```
1 #if __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ > 4)
```

Table A.1: Operating systems and compilers supported by CUDA.

Operating system	32	64	Kernel	Compiler	GLIBC
Windows 7/Vista/XP, Windows Server: 2008 R2, 2008, 2003	x	x		MSVC8 (14.00), MSVC9 (15.00), MSVC2010 (16.00)	
SLES11-sp1	x	x	2.6.32.12-0.7-pae	gcc 4.3-62.198	2.11.1-0.17.4
RHEL-6.0		x	2.6.32-71.el6	gcc 4.4.4	2.12
Ubuntu-10.10	x	x	2.6.35-23-generic	gcc 4.4.5	2.12.1
OpenSUSE-11.2	x	x	2.6.31.5-0.1	gcc 4.4.1	2.10.1
Fedora13	x	x	2.6.33.3-85	gcc 4.4.4	2.12
RHEL-4.8		x	2.6.9-89.ELsmpl	gcc 3.4.6	2.3.4
RHEL-5.5	x	x	2.6.18-194.el5	gcc 4.1.2	2.5
Mac OS X 10.6	x	x	10.0.0	4.2.1 (build 5646)	
Mac OS X 10.5.2+	x	x			discontinued
Mac OS X 10.5.7	x	x	9.7.0	4.0.1 (build 5490)	discontinued

Changing this line to:

```
1 #if __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ > 5)
```

allows `gcc` to compile even if its version is exactly 4.5. Notice the change from 4 to 5. This simple solution is rather a hack and should be avoided in production environments. However, it works in typical Linux installations, with newer `gcc` and `glibc`. It is obvious that NVIDIA will change this to a fully working version in the future.

The supported hardware list is quite long and can be found at the NVIDIA web page: *http://www.nvidia.com/object/cuda_gpus.html*.

A.1.3. CUDA Versions and Device Capability

To distinguish between different CUDA versions working on diffrent graphic processors, two numbers are introduced. The first is the *compute capability*, the second is the *version number* of CUDA Runtime and CUDA Driver APIs. Such numbers can be checked by running the `deviceQuery` program from the CUDA SDK package. An example of the information gained by this program for the NVIDIA GeForce GTX 590 is given below. These numbers and even more information can also be gained from a call to the **`cudaGetDeviceProperties`** function and by accessing elements from the structure it returns.

```
Device 1: "GeForce GTX 590"
  CUDA Driver Version / Runtime Version 4.0 / 4.0
  CUDA Capability Major/Minor version number: 2.0
  Total amount of global memory: 1536 MBytes (1610153984 bytes)
  (16) Multiprocessors x (32) CUDA Cores/MP: 512 CUDA Cores
  GPU Clock Speed: 1.22 GHz
  Memory Clock rate: 1707.00 Mhz
  Memory Bus Width: 384-bit
  L2 Cache Size: 786432 bytes
  Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory: 65536 bytes
  Total amount of shared memory per block: 49152 bytes
  Total number of registers available per block: 32768
  Warp size: 32
```

```
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and execution: Yes with 1 copy engine
Run time limit on kernels: Yes
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Concurrent kernel execution: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support enabled: No
Device is using TCC driver mode: No
Device supports Unified Addressing (UVA): Yes
Device PCI Bus ID / PCI location ID: 4 / 0
Compute Mode:
   < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

In this case, the version for a driver and CUDA Runtime API is 4.0, and CUDA capability is 2.0. The compute capability and driver versions are very important due to the possible abilities of the hardware which can be used by the program. For example, only a GPU with a compute capability of 1.3 and later can use double precision natively (defined by `double` type in C). Earlier versions convert `double` precision types to their `float` replacements, degrading precision and sometimes numerical stability. More details and a complete overview of the compute capabilities of GPUs supported by CUDA can be found in Appendixes A and F of the *Cuda Programming Guide* [28].

The host part of the program compiled for a CUDA environment is doing a number of things not related to the computations on the GPU. It is responsible for management of the device, context, memory, code module and texture as well as execution control and interoperability with OpenGL and Direct3D (on a limited number of platforms). The device part compiled for the GPU is responsible for handling execution and calculations only.

Host part has two programming interfaces to CUDA: the low-level *CUDA Driver API* and the higher-level *CUDA Runtime API* built on top of the first one. Both of them can be used in one code and are interchangeable. Programming in CUDA Driver API is more elaborate and requires more code; programming in CUDA Runtime API is easier and provides implicit initialization, context management, and device code module management. These two APIs are distinguishable by the use of specific notation. The CUDA Driver API uses the "`cu`" prefix and is delivered by the `nvcuda/libcuda` dynamic library; the CUDA Runtime API has all functions prefixed with the "`cuda`" prefix and is delivered by the `cudart` dynamic library. These differences are important only for the host code. On a GPU there is no difference, regardless of the API used.

The CUDA Driver API is backward-compatible but forward-incompatible. This means that one can use the latest driver for CUDA applications, and even programs compiled for compute capability 1.0 will work. However, programs compiled against compute capability 2.0 will be unable to run on older drivers. The exact dependencies between architectures and driver versions are shown in Fig. A.1.

It is clear that, the higher the compute capability, the more possibilities and the more advanced functions are available. When the target hardware is uncertain, the best choice is to compile for compute capability 1.0. Although this gives a very limited set of functions, it is guaranteed that it will work everywhere. For compilations on targeted platforms, where hardware is known and features are predictable, a direct

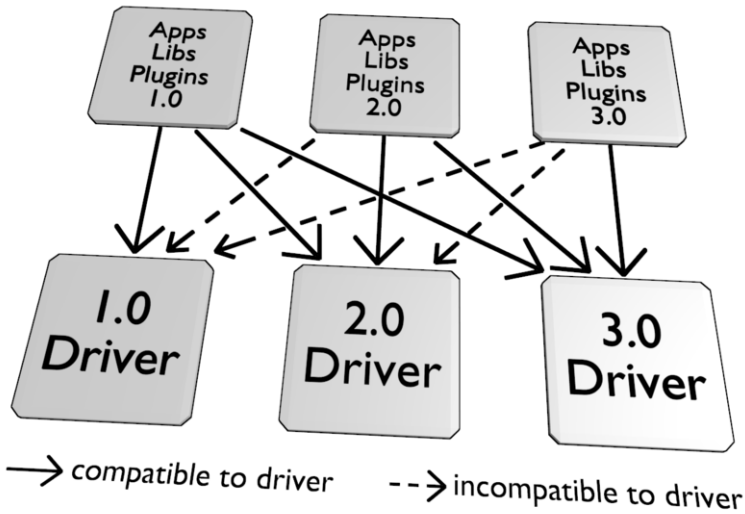

Figure A.1: CUDA Driver versions and software compatibility.

desired compute capability should be used. The options for specifying target compute capability and NVIDIA drivers and hardware are `-arch`, `-code` and `-gencode`. Double-precision calculations are possible only for compute capability higher than or equal to 1.3, so the option `-arch=sm_13` should be specified at least. The details are described in the `nvcc` User Manual [29].

A.2. CUDA Runtime API Example

Each CUDA-oriented program has to be built in four generic stages:

1. Data preparation on host;
2. Data copying to GPU device from host;
3. Computation;
4. Data copying from GPU device back to host.

Despite the common steps noted above, CUDA source in its most basic form looks simpler than the OpenCL version and requires much less typing effort.

The SAXPY operation was presented in section 2.14 and is explained in detail using C/C++ examples. The CUDA implementation of SAXPY can be seen in listing A.1. This version is utilitizing the simplicity of CUDA Runtime API mentioned in the previous section.

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <cuda.h>
5
```

```
// fill vectors with some usable values
void initializeHostData(float *a, float *x, float *y, int n) {
  *a = 1.0f; // for example:
  for (int i = 0; i < n; i++) x [i] = y [i] = (float)i; // 0.0, 1.0, 2.0, 3.0,...
}

// print results to stdout
void printResults(float *x, int n){
  for (int i = 0; i < n; i++) printf("%8.3f", x [i]);
  printf("\n");
}

// cuda kernel
__global__ void saxpy(float *z, float a, float *x, float *y, int n){
  int i = threadIdx.x;
  z [i] = a * x [i] + y [i];
}

int main(int argc, char **argv){
  float *hostX, *hostY, *hostZ; // host-allocated vectors
  float *devX, *devY, *devZ; // vectors allocated on GPU
  float a; // scalar variable on host
  int n; // vector size
  int mem; // memory size in bytes

  n = 8; // 8 elements in each vector
  mem = sizeof(float) * n; // each could take 32 or 64 bytes
                                  // ...depending of architecture

  // allocate memory on host, all data are CPU-visible
  hostX = (float*)malloc(mem);
  hostY = (float*)malloc(mem);
  hostZ = (float*)malloc(mem);
  initializeHostData(&a, hostX, hostY, n);

  // allocate memory on GPU for vectors,
  // only pointers to data are CPU-visible, not data itself
  cudaMalloc((void**)&devX, mem);
  cudaMalloc((void**)&devY, mem);
  cudaMalloc((void**)&devZ, mem);

  // copy data from CPU-vectors to GPU memory
  cudaMemcpy(devX, hostX, mem, cudaMemcpyHostToDevice);
  cudaMemcpy(devY, hostY, mem, cudaMemcpyHostToDevice);

  // launch saxpy calculation by invoking CUDA kernel on GPU device
  saxpy << < 1, n >> > (devZ, a, devX, devY, n);

  // copy calculated vector devZ back to CPU accesible vector hostZ
  cudaMemcpy(hostZ, devZ, mem, cudaMemcpyDeviceToHost);

  // print results
  printResults(hostZ, n);

  // release allocated memory
  free(hostX); free(hostY); free(hostZ);
```

```
62   cudaFree(devX); cudaFree(devY); cudaFree(devZ);
63
64   return cudaSuccess;
65 }
```

Listing A.1: The simplest SAXPY CUDA example.

For clarity and simplicity, it does not include any error checking or data management. The program is complete and can be run. Compilation on the Linux platform can be performed with the command:

```
1 nvcc saxpycudasimple.cu -o saxpycudasimple
```

The result of the above program is a simple vector listed below:

```
0.000   2.000   4.000   6.000   8.000  10.000  12.000  14.000
```

The values printed are elements of the vector *devZ*, calculated using SAXPY on a GPU from two other vectors, *devX* and *devY*, and from scalar value *a* using the formula $devZ = a\ devX + devY$. Both vectors contain increasing numbers (0.0, 1.0, 2.0, and so on), so the results have to look like presented above. In this example, vectors have only 8 elements.

A.2.1. CUDA Program Explained

The program presented on listing A.1 is very straightforward. It is built upon the steps mentioned in the previous section. Now it's time to look closer at the internals of the source code.

The headers section is simple and has to contain at least one crucial include: `cuda.h`:

```
1 #include <cuda.h>
```

This inclusion is necessary for all CUDA-related functions to make the compilation possible. In bigger programs, usually other header files and libraries are added. In the examples from SDK, there is a simple but handy multiplatform `cutil.h` library (not supported by NVIDIA); the others are mentioned in section A.1.2.

All data used for SAXPY calculations are just simple, one-dimensional vectors defined as:

```
1 float *hostX, *hostY, hostZ;
2 float *devX, *devY, devZ;
```

There are three vectors, but they are represented by six variables. At this point, it should be obvious that one set of variables is designated to handle data only on the host, with CPU code (their names are `host`-prefixed), while the other one is dedicated to be used only on a GPU (names are `dev`-prefixed). Only the variables with the `host` prefix are accessible from the CPU. The set of variables which is `dev`

prefixed can make the program crash with a SIGSEGV signal (Segment Violation), if they are accessed by the CPU code using, for example, the *devX notation. The CPU is able to pass and access *pointers* only to data stored on the GPU, not to manage or access data pointed by them. Common sense assumes that GPU data are inaccessible by the CPU and vice versa. Therefore, it is a good and recommended practice to define variables for both CPU and GPU, ignoring the fact that they may hold the same data.

The next important step, after variable declarations, is to allocate memory for data. This is accomplished in two ways. On the host side, in CPU code, allocation is done by calling one of the memory allocation functions usually **malloc**, **calloc** or several invocations of **new** in C++. This allocation is language-dependent and can be done in several ways. Second, memory has to be allocated also on the GPU device, for pointers prefixed with dev.

```
1 hostX = (float*) malloc(mem);
2 hostY = (float*) malloc(mem);
3 hostZ = (float*) malloc(mem);
```

The above listing presents allocation with the **malloc** function, the standard C memory allocator. It requires an argument of the size_t type (integer value, compatible with int), which specifies the number of bytes for allocation. It returns a void pointer to allocated memory. This pointer has to be cast to the float* type in order to keep type conformity. The *mem* variable is just an integer value of the memory amount (in bytes) which should be allocated. This value must be calculated using the expression sizeof(float)*n, where n is the number of floats allocated. If the allocation fails, then NULL is returned. Allocation on a GPU device cannot be done by **malloc**, but there is another API function.

```
1 cudaMalloc((void**) &devX, mem);
2 cudaMalloc((void**) &devY, mem);
3 cudaMalloc((void**) &devZ, mem);
```

In the above listing, the allocation is done for the same memory amount, expressed by the *mem* variable. The **cudaMalloc** function is used, and the pointer is given by its address. This function requires two arguments: the far pointer of void type and the amount of memory to allocate. The address operator (&) makes the far pointer from the dev-prefixed pointers, and therefore there is a need for double casting (void** notation). If memory allocation fails, this function returns cudaErrorMemoryAllocation. If memory allocation is successful, cudaSuccess is returned.

Data allocated on the CPU should be initialized. It is done by the helper function **initializeHostData**. In the SAXPY example shown on listing A.1, the vectors are initialized with increasing float numbers, starting from 0.0.

The initialized vectors, and in general any data, have to be copied into a device. This is necessary at least once, before calculations are done. Each invocation of the **cudaMemcpy** function copies one vector from the host to the GPU memory. The process or the data transfer between CPU memory and GPU memory can be slow due to

the limitations of architecture. Nowadays (2011), it does not surpass the 16 gigabits per second limit.

```
cudaMemcpy(devX, hostX, mem, cudaMemcpyHostToDevice);
cudaMemcpy(devY, hostY, mem, cudaMemcpyHostToDevice);
```

The **cudaMemcpy** function copies *mem* bytes from the memory area pointed by `host`-prefixed variables to the memory area pointed by `dev`-prefixed pointers. The last argument of invocation, the `cudaMemcpyHostToDevice`, is one of the constants shown in table A.2 and specifies the direction of data copying. The ranges of copied memory

Table A.2: CUDA memory copy directions constants.

`cudaMemcpyHostToHost`	Data are copied from CPU pointer to CPU pointer.
`cudaMemcpyHostToDevice`	Data are copied from CPU to device pointer. This is very often used.
`cudaMemcpyDeviceToHost`	Data are copied from device to host. This is usually used after calculations.
`cudaMemcpyDeviceToDevice`	Data are copied from device, to another device pointer. This version of copy is incredibly fast.

cannot overlap. If the source is exchanged with the destination, the result of the copy is undefined.

After data are copied to the GPU, it is time to run the calculations. In CUDA terminology, the function run on the GPU is called a *kernel*. Kernels are just work items. They run together as threads on the GPU device. Such a kernel should be defined as a `void` C function, with the prefix of `__global__`; this means that the function can be called on the CPU but is executed on the GPU.

The kernel launch is done by one single line of C code with some non-standard addititions. Here are the special characters used: <<< and >>> to configure the run.

```
saxpy<<< 1, n >>>(devZ, a, devX, devY, n);
```

The line contains the kernel name (`saxpy`), opening brackets (<<<), value of 1, variable *n*, closing brackets (>>>), and arguments list in round brackets (`...`). The values of 1 and *n* have significant meaning here and are explained below. The argument list is a standard argument list as in normal C.

The configuration parameters enclosed in triangle brackets are composed of two numbers. The first of them is just equal to 1, and the second is the vectors data size *n*. Both numbers are of integer type. This configuration forces the GPU to run *n* threads, organized linearly within one block. There will be 8 kernels run simultaneously if *n* is equal to 8. This number is limited to the physical bounds of the device. For example, the Tesla C1060 is able to run 512 threads in linear fashion, in a single block. The Tesla C2070 can run up to 1024 threads. It depends on GPU capability.

The first number in triangle braces is the number of work groups. In CUDA vocabulary, this means the number of blocks. The block is a thread organization unit, similar to a work group in OpenCL. The number of blocks which can be run at once also has some limits, but it is much higher than the maximal number of threads in a single block. Usually, common GPUs are able to run up to 65535 blocks. The block set scheduled for a run is called a grid.

The simple example discussed here is far too small to fully occupy the whole GPU. It uses only 8 threads organized linearly in one block only. Bigger problems require a more complicated organization of threads and blocks, and their configuration does not have to be expressed by integer numbers. Blocks and threads can be organized in multidimensional areas by the usage of the `dim3` type (which is a C structure with three fields: `x`, `y`, and `z`, denoting three dimensions in 3D space). Blocks in the grid can be run either as a linear or a two-dimensional area of blocks. Threads can be run either as one-, two- or three-dimensional blocks. Let's see some examples.

The run where there are two blocks of threads (maximum value of `n` is 1024 for the Tesla C1060 or 2048 for the Tesla C2070):

```
1 saxpy <<< 2, n/2 >>>(devZ, a, devX, devY, n);
```

Because of the two blocks, the number of threads in one block has to be divided by two. In this configuration, the same number of threads will be run as for <<<1,n>>>, but in two smaller blocks. It is even possible to run one single thread per block, with *n* blocks:

```
1 saxpy <<< n, 1 >>>(devZ, a, devX, devY, n);
```

This setup allows computation of SAXPY for vectors of 65535 elements. Two-dimensional grid configuration requires the usage of special variables of the type `dim3`.

```
1 dim3 grid(2,2);
2 dim3 block(2);
3 saxpy <<< grid, block >>>(devZ, a, devX, devY, n);
```

In this case, the grid is defined as a two-dimensional area of size 2×2 and the variable *grid* has field `x` equal to 2, `y` equal to 2 and `z` equal to 1. The whole grid contains 4 blocks. The block itself is defined by the *block* variable, which has the same structure, so the `x` dimension has the value of 2 and the remaining fields are equal to 1. Because the `dim3` type is a 3d structure internally, the whole shape for the grid has dimensions of `{2,2,1}` and each block in the grid has dimensions of `{2,1,1}`. Such configuration of a launch also computes 8 elements of the vector. In real applications, the grid and block sizes have to be computed. For 2048 elements in the vector, this computation can look like this:

```
1 dim3 grid( n/32, 1, 1);
2 dim3 block( 32, 1, 1);
```

or, with some often used constants and variables:

```
dim3 grid(size_x/TILE_DIM, size_y/TILE_DIM);
dim3 block(TILE_DIM,BLOCK_ROWS);
```

The configuration of the kernel run can be extended by adding two more parameters (shared memory size and stream execution number), but they will not be explained here. Suggested reading is in [4] and [30]. An example of the kernel lanuch with 1024 bytes of shared memory (shared in CUDA terminology; in OpenCL, it is called local memory), is shown below:

```
dim3 gridsize( n/TILE, 1, 1);
dim3 blocksize( TILE, 1, 1);
saxpy <<< gridsize, blocksize, 1024 >>>(...);
```

Using this calling method, an additional shared memory is allocated and accessible as an independent memory associated with each block. Threads can communicate with each other using this very memory.

The similar architecture and different names can lead to confusion. A comparison of nomenclature in OpenCL and CUDA is collected in table A.3 [4].

Table A.3: Comparison between CUDA and OpenCL nomenclature.

CUDA name	OpenCL name
global memory	global memory
constant memory	constant memory
shared memory	local memory
local memory	private memory
kernel	kernel
host program	host program
grid	NDRange (index space)
block	work group
thread	work item

The CUDA kernel has to be properly defined. First, it can not return any type, thus it is a **`void`** function. The argument list can contain only data passed by the value or pointer to GPU allocated memory. Kernel function can launch only functions defined with the `__device__` prefix. It can not make recursive or far function calls. This makes debugging a bit harder, because no function from the standard C library can be called.

The shape of grids and blocks has significant influence on further calculation. They not only shapes the task to solve, but also may require additional calculations. For example, if there is a large vector to calculate and it has to be processed in a single-row manner, then the number of threads possible to run inside one block will not be sufficient. The usage of multiple blocks will require special addressing instructions, as shown in the full example kernel in the next sections.

In the example shown in listing A.1, the kernel function is simple (look also for the version in listing A.2, which has range control added):

Table A.4: CUDA predefined variables visible in the kernel scope.

Variable	Description
threadIdx	3-dimensional thread index inside the block.
blockDim	3-dimensional size of the block
blockIdx	3-dimensional block index inside the grid
gridDim	3-dimensional grid size

```
1 __global__ void saxpy(float *devZ, float *a, float *devX, float *devY){
2   int i = threadIdx.x;
3   devZ[i] = a * devX[i] + devY[i];
4 }
```

The most notable C extension is `__global__`, mentioned earlier in this section. The other one is the *threadIdx* variable, which in this scope is predefined. It contains the index number of each particular thread. In the case of vector size 8, the first thread ID has number 0 and the last one has number 7. Indexing of threads can be done using the `dim3` type, so the whole index is 3-dimensional, with the fields of `{x,y,z}`. Each of those fields denotes the thread index in one dimension of the block. The maximal dimensions of the block used for calculation are stored in *blockDim* (also of type `dim3`).

Besides the *threadIdx* variable, there are a few other predefined variables, which exists in the scope of kernel function. They are collected in table A.4. Although the grid is described by a 3-dimensional variable, the device limits cause the size of the grid to be at most two-dimensional. The last field, `z`, has just the value of 1.

The data index calculation is simple, as in line 20 of the listing A.1. Each vector element is processed in its own thread, and the thread index number is the same as the corresponding index of the vector element. For bigger problems, this is not sufficient. The threads inside the different blocks must also use other predefined variables, because the numbering with *threadIdx* is the same for each block and always starts from 0. Therefore, the proper data index is calculated form the general formula:

$$\text{dataindex} = \text{threadIdx} + \text{blockIdx} \times \text{blockDim} \tag{A.1}$$

In the case of one-dimensional data laid over the x-axis (which is often default), the x-field should be used and the formula becomes: *threadIdx.x* + *blockIdx.x***blockDim.x*. This gives the exact index number for each element of the long vector after a split into several blocks. For other runtime configurations, other formulas should be used; see section A.2.2.

The kernel function for SAXPY calculation has one serious flaw. It is far too simple to be operational without errors. Simplicity here is a bad thing, because there is no control of the data limits. In some circumstances, this function can write data beyond the allowed range by using too big an `index` value. Therefore, a valid function has to check if the `index` value does not exceed the allowed range stored and passed in by *n*:

```
// cuda kernel with index limited to n
__global__ void saxpy(float *z, float a, float *x, float *y, int n) {
  int i = threadIdx.x;
  if(i < n) // check for allowed index range
    z[i] = a * x[i] + y[i];
}
```

Listing A.2: Repaired SAXPY kernel with range checking

The limit checking added to the kernel shown in listing A.2 is very important because without it, this kernel can cause data errors that are very difficult to find or even random runtime errors. For more complicated cases, tests can be launched from a dedicated device function defined elsewhere.

A.2.2. Blocks and Threads Indexing Formulas

Depending on the different shape of the grid and blocks, there are a number of formulas to calculate the real, unique index of data.

There is a limited number of combinations of one- or two-dimensional grids with one-, two- or three-dimensional blocks of threads. In the formulas below, additional variables are introduced. *Bindex* is the unique block index, *Tindex* is a real, unique data index. Both variables are compatible with integers. Also, the 1D, 2D or 3D symbols mean the shape of the space (1D - vector, 2D - matrix, 3D - cube matrix). If the value of 1 is given, that means there is an exact number of 1 element used. It is assumed that in the case of 1D shape, the x-axis is used. So, the formulas are the following:

1. One block in the grid, with one thread only (1:1). This is the simplest possible case and there is no parallelism at all.

   ```
   Bindex = 1;
   Tindex = 1;
   ```

2. One block in the grid, threads organized in a one-dimensional array (1:1D). This case works only if the number of threads does not exceed the GPU limits.

   ```
   Bindex = 1;
   Tindex = threadIdx.x;
   ```

3. One block in the grid, two-dimensional array of threads (1:2D). The GPU limit causes the threads to be limited to the rectangle with an area not bigger than the maximal thread count possible. For square blocks, this number is `sqrt(GPU limit)`, usually `(int)sqrt(512)` or `(int)sqrt(1024)`, which gives rather small numbers of 22 and 32, respectively.

   ```
   Bindex = 1;
   Tindex = blockDim.x * threadIdx.y + threadIdx.x;
   ```

4. One block in the grid, three-dimensional set of threads (1:3D). This case is even more limited by the total number of threads possible to run in a single block (for cubic shapes, it is 8 for the Tesla C1060 or 10 for the Tesla C2070).

```
Bindex = 1;
Tindex = blockDim.x * blockDim.y * threadIdx.z +
         blockDim.x * threadIdx.y + threadIdx.x;
```

5. One-dimensional array of blocks, one thread per block (1D:1). This configurations usually allows running usually up to 65535 threads and so is the maximum data size handled.

```
Bindex = blockIdx.x;
Tindex = Bindex;
```

6. One-dimensional array of blocks, one-dimensional array of threads (1D:1D). This setup is very often used. It is capable of handling 65535 blocks $\times$ 512 (or 1024) threads, which gives 33553920 (or 67107840) simple data elements.

```
Bindex = blockIdx.x;
Tindex = Bindex * blockDim.x + threadIdx.x;
```

7. One-dimensional array of blocks, two-dimensional array of threads (1D:2D).

```
Bindex = blockIdx.x;
Tindex = Bindex * threadIdx.y + threadIdx.x;
```

8. One-dimensional array of blocks, three-dimensional array of threads (1D:3D).

```
Bindex = blockIdx.x;
Tindex = blockIdx.x * blockDim.x * blockDim.y * blockDim.z +
         threadIdx.z * blockDim.y * blockDim.x +
         threadIdx.y * blockDim.x + threadIdx.x;
```

9. Two-dimensional array of blocks and one thread per block (2D:1).

```
Bindex = gridDim.x * blockIdx.y + blockIdx.x;
Tindex = Bindex;
```

10. Two-dimensional array of blocks and one-dimensional array of threads (2D:1D).

```
Bindex = blockIdx.y * gridDim.x + blockIdx.x;
Tindex = Bindex * blockDim.x + threadIdx.x;
```

11. Two-dimensional array of blocks and two-dimensional array of threads (2D:2D).

```
Bindex = blockIdx.y * gridDim.x + blockIdx.x;
Tindex = Bindex * blockDim.y * blockDim.x +
         threadIdx.y * blockDim.x + threadIdx.x;
```

12. Two-dimensional array of blocks and three-dimensional array of threads (2D:3D).

```
Bindex = blockIdx.y * gridDim.x + blockIdx.x;
Tindex = Bindex * blockDim.z * blockDim.y * blockDim.x +
         threadIdx.z * blockDim.y * blockDim.z +
         threadIdx.y * blockDim.x + threadIdx.x;
```

After the kernel is launched on the GPU, the host code continues its execution on the CPU. The return from the kernel function is immediate. In order to decide which function should work where, a few special prefixes are introduced. The functions which are launched on the CPU and executed on the CPU have the prefix **__host__**. This is the default prefix for all C functions and does not have to be specified. Some special functions have to be invoked on the CPU but are executed only on the GPU. Such functions are kernels, and they have to be prefixed with the **__global__** keyword. The kernel functions are launched on the host using a special notation with triangle brackets. All functions called from the device and executed on the device have to be prefixed with the **__device__** keyword. No functions are launched on the GPU and executed on the CPU host. There is the possibility to make functions which can be executed both on the GPU and on the CPU. They have to have the double prefix: **__host__ __device__**. Although such functions share the same source code, `nvcc` generates two versions of executable code: one for the CPU host and the other one for the GPU device. This is possible because during the compilation process, such functions are compiled separately for each target platform. The keywords are collected in table A.5. Not only functions can have such prefixes, but variables

Table A.5: CUDA prefixes.

__host__	Allows run on CPU host, this is default setting
__device__	Allows run only on GPU device, cannot be launched from host
__global__	Allows launch of kernel function

and constants too. By default **__host__** is appended, but specyfying any other prefix changes the default.

There is one additional variable declared and defined inside the kernel function in listing A.1. This is the *i* variable, defined for indexing the data. This kind of variable is called *local*, because it is declared and defined inside the kernel (or device) function. In simple cases, such variables are treated by the compiler like register variables, but there are some circumstances where they become local and are placed

in the local memory (in the CUDA terms). This is explained in detail in NVIDIA documentation and in [4].

The typical SAXPY operation is just an iteration over the vector elements. However, in the `saxpy` kernel there is no loop. This can be surprising, but the loop functionality is taken by parallelism here. Notice the difference in the listings:

```
for(i=0;i<N;i++)
  Z[i] = a*X[i] + Y[i];
...

__device__ void saxpy(float* Z, float a, float X, float Y){
  Z[i] = a*X[i] + Y[i];
}
saxpy<<<N>>>(Z,a,X,Y);
```

The **for** loop is replaced by calling the kernel exactly *N* times. The loop index from the **for** loop is replaced by the data index calculated from *threadIdx* and other variables. Because of that organization, there are situations where O(n) complexity is reduced to O(1).

Calculations carried out by the kernel usually give some results. All necessary data have to be downloaded back to the host device. In order to copy data, the same function is used as before: **cudaMalloc**, but the direction is reversed. The constant used to copy is `cudaMemcpyDeviceToHost`. The whole line of program is the following:

```
cudaMemcpy(hostZ,devZ,cudaMemcpyDeviceToHost);
```

After copying, the results can be used by the CPU code on the host.

A.2.3. Runtime Error Handling

The simple example of SAXPY given in listing A.1 has no error control. In the case of error or some strange situation there will be no tracking of the problem and no feedback. Fortunately, CUDA has a few functions which can cast a light on some unexpected things.

All CUDA functions (except the kernel) return error codes, which can be used for investigation. If the error occurs, the next invocation of CUDA functions is blocked until the error-state is cleared. Consider the following example:

```
int variable = 1;
cudaError_t error_id = cudaFree(&variable);
printf("error code: %i\n",(int)error_id);
```

The code tries to free statically allocated data from the address generated by the address operator. This is faulty code, and the error returned is not equal to 0. This way, it can be recognized and displayed.

The error codes alone are not very informative. There is the **cudaGetErrorString** function, which takes the error code as an argument. The code can be replenished with an additional line of code:

```
int variable = 1;
cudaError_t error_id = cudaFree(&variable);
printf("error code: %i\n",(int)error_id);
printf("error info: %s\n",cudaGetErrorString(error_id));
```

In this way, information about the error can be displayed for the user (the error string is "`invalid device pointer`"). Unfortunately, some of the errors can not be caught using this approach. The concern affects the kernel invocation because by design, it does not return anything. There are also other unmaskable errors, for example those caused by deadly signals or hardware failures.

To check for errors caused by kernel invocation or the kernel itself, one has to use the **cudaGetLastError** function. This function returns the last error code from any CUDA function and resets the error state. The recommended way of checking kernel runtime errors is the following:

```
cudaError_t error_id = cudaSuccess; # at the start
...
kernel <<< gridsize, blocksize, sharedmem, streamnr >>> ( arguments );
error_id = cudaGetLastError(); # get and reset
if(error_id != cudaSuccess){
  ... # handle error
}
```

An easy, simple and handy way to deal with errors is to wrap all CUDA functions with immediate error control by using the of wrapper function, as follows:

```
void SafeCall( cudaError_t error_id ) {
  if( error_id != cudaSuccess ){
    fprintf(stderr,"Error: code %i, %s\n",error_id, cudaGetErrorString(error_id));
    # do whatever is needed
  }
}
```

This function tests the error code passed by the value of *error_id* and, if the error really occurs, prints the error and possibly does other things – for example, free GPU memory, etc. Immediate exiting from the program is generally not a good idea here, but sometimes there is no other way. The CUDA functions can then be called with the wrapper usage:

```
SafeCall(cudaMalloc((void**)&data,sizeof(data)));
```

This allows basic runtime error handling and can be easily extended. This way of error handling is used by SDK examples. **cutilSafeCall** uses a very similar code.

In the 4.0 release of CUDA, there is one more function that can be helpful. This is the **cudaPeekAtLastError** function. It acts exactly like **cudaGetLastError**, but *does not reset* the error state. Consequently, all calls to CUDA-related functions will still be blocked until the error flag is not reset.

A.2.4. CUDA Driver API Example

In this section, the other approach to CUDA programming is presented. It is still the same SAXPY example, but with the usage of more low-level programming required by CUDA Driver API. It has more similarities to OpenCL than CUDA Runtime API and is also longer in comparison to CUDA Runtime API. The version presented here is in its simplest possible form, and therefore the similarities to the OpenCL version are clearly visible. After adding error checking and other natural enhancements, the code quickly becomes more elaborate. It can be compiled by a native C compiler, without the need of `nvcc`, exactly like OpenCL code. For the sake of simplicity, no error control is provided and the code is reduced to necessary invocations only. The conventions are the same as in listing A.1.

Because of the lower-level code, the program has to be split into two parts. One of them is the kernel alone. It has to be compiled by `nvcc` or `cudacc` programs. The result file should be a special *cubin* object file loaded into the GPU directly. The remaining part of the program has to be compiled with the use of a native system compiler. In this example, `gcc` is used and compilation invocation is compatible with the Linux Operating System.

The kernel function used in this SAXPY example is exactly the same as in the previous example. In fact, it is only in a separate file with the "`.cu`" extension. This allows compiling it to `cubin` file with `nvcc`.

```
1 // cuda kernel for cubin compilation
2 __global__ void saxpy(float *z, float a, float *x, float *y, int n) {
3   int i = threadIdx.x;
4   if( i < n) // check for allowed index range
5     z[i] = a * x[i] + y[i];
6 }
```

Listing A.3: The SAXPY kernel for cubin compilation, the file `saxpyKernel.cu`.

The compilation to object file is performed by `nvcc` with a special argument in the command line:

```
nvcc -cubin saxpy_kernel.cu -o saxpy.cubin
```

This invocation is the simplest one possible. Without the `-o` argument, the default object name is the same as the source file, with the `.cubin` extension added (so in this example, it would be `saxpyKernel.cubin`). Depending on the needs, a number of additional options can be given:

```
nvcc --machine 64 -arch sm_20 --compiler-options="-Wall" \
  saxpyKernel.cu -o saxpy.cubin
```

Where `-machine 64` forces the compiler to generate a binary code for 64-bit architectures, `-arch sm_20` enables CUDA Compute Capability 2.0, and the longest option here – `-compiler-options` specifies additional options which are directly passed to `gcc` (or other used compilers; in this case, this option enables warnings generated during the compilation process).

The main program can be seen in listing A.4. It is made to be similar to the CUDA Runtime API program with a SAXPY example presented in listing A.1. The corresponding sections are easy to compare.

```
#include <stdio.h>
#include <string.h>
#include <cuda.h>

// fill vectors with some usable values
void initializeHostData(float *a, float *x, float *y, int n) {
  *a = 1.0f; // for example:
  for (int i = 0; i < n; i++) x [i] = y [i] = (float)i; // 0.0, 1.0, 2.0, 3.0,...
}

// print results to stdout
void printResults(float *x, int n){
  for (int i = 0; i < n; i++) printf("%8.3f", x [i]);
  printf("\n");
}

// Host code
int main(int argc, char** argv){
  float *hostX, *hostY, *hostZ; // host-allocated vectors
  float a; // scalar variable on host
  int n; // vector size
  int mem; // memory size in bytes

  CUdevice cuDevice; // device to use
  CUcontext cuContext; // context for run
  CUmodule cuModule; // module to load
  CUfunction saxpy; // our kernel function

  CUdeviceptr devX; // vectors allocated on GPU
  CUdeviceptr devY; // in simpler RuntimeAPI
  CUdeviceptr devZ; // they were just floats

  n = 8; // 8 elements in each vector
  mem = sizeof(float) * n; // each could take 32 or 64 bytes
                                  // ...depending of architecture
  // initialize driver API
  cuInit(0);

  // pick up device (default 0)
  cuDeviceGet(&cuDevice, 0);

  // Create context
  cuCtxCreate(&cuContext, 0, cuDevice);
  cuModuleLoad(&cuModule, "saxpy.cubin");

  // Get function handle from module
  cuModuleGetFunction(&saxpy, cuModule, "saxpy");

  // Allocate input vectors hostX and hostY in host memory
  hostX = (float*)malloc(mem);
  hostY = (float*)malloc(mem);
  hostZ = (float*)malloc(mem);
```

```
53  initializeHostData(&a, hostX, hostY, n);
54
55  // Allocate vectors in device memory
56  cuMemAlloc(&devX, mem);
57  cuMemAlloc(&devY, mem);
58  cuMemAlloc(&devZ, mem);
59
60  // Copy vectors from host memory to device memory
61  cuMemcpyHtoD(devX, hostX, mem);
62  cuMemcpyHtoD(devY, hostY, mem);
63
64  // Launch the CUDA kernel
65  void *args[] = { &devZ, &a, &devX, &devY, &n };
66  cuLaunchKernel(saxpy, 2, 1, 1, 8, 1, 1, 0, NULL, args, NULL);
67
68  // copy results back to host from device and print
69  cuMemcpyDtoH(hostZ, devZ, mem);
70  printResults(hostZ, n);
71
72  //release allocated memory
73  cuMemFree(devX); cuMemFree(devY); cuMemFree(devZ);
74  free(hostX); free(hostY); free(hostZ);
75
76  // release device context
77  cuCtxDetach(cuContext);
78
79  return 0;
80 }
```

Listing A.4: The SAXPY CUDA Driver API Example.

CUDA Driver API is in general very similar to the corresponding OpenCL API, only the functions, variables and constant names are a bit different. Nevertheless, the similiarity goes from the underlying hardware, and this is the point where those two APIs show their close relationship. The program given in listing A.4 is just the bare, simplest version of CUDA Driver API, and it looks nearly like the OpenCL program presented in section 2.14.2. Some important parts of this program are explained below.

```
1 CUdevice cuDevice
2 CUcontext cuContext;
3 CUmodule cuModule;
4 CUfunction saxpy;
5 CUdeviceptr devX;
```

The device number *cuDevice* is compatible with the `int` type. The default value is set to 0. The latter variables *cuContext* and *cuModule* in lines 2 and 3 are used for storing a device context and for a module loaded from a CUBIN object. The SAXPY function (**saxpy**) from line 4 is the kernel. The data stored on the device are accesible by the `CUdeviceptr` type.

```
1 cuInit(0);
2 cuDeviceGet(&cuDevice, 0);
3 cuCtxCreate(&cuContext, 0, cuDevice);
4 cuModuleLoad(&cuModule, "saxpy.cubin");
5 cuModuleGetFunction(&saxpy, cuModule, "saxpy");
```

The start of the program begins with some initialization routines. First, Driver API is initialized (line 1). The **cuInit** function has to be called before any other API function is called, otherwise the error CUDA_ERROR_NOT_INITIALIZED will be returned. In version 4.0, flags have to be set to 0. Then the appropriate device is selected by the **cuDeviceGet** function (line 2). The total number of devices can be obtained by calling the **cuDeviceGetCount** function (not shown here), where the first device in the list is indexed by 0. Other functions are possible too – for example, for getting the name of the device (**cuDeviceGetName**) or for obtaining its properties (**cuDeviceGetProperties**). In the third line, the context for the device is created (**cuCtxCreate**). This function not only creates the context but also binds it to the current caller thread. The second argument of the **cuCtxCreate** function is flags, here set to a default value of 0. The last argument is just the device selected in the previous line. The 4^{th} line is the **cuModuleLoad** function. It loads the compiled module into the current context. The first argument is a structure holding the loaded module, the second argument is a string containing the file name. The file loaded by this function has to be a PTX, *cubin* or *fatbin* file (all can be obtained from the nvcc compiler in the latest version). In the 5^{th} line, the SAXPY function handle is obtained and put into a *saxpy* pointer. If no such function exists in the module, then CUDA_ERORR_NOT_FOUND is returned.

```
1 hostX = (float*) malloc (mem);
2 initializeHostData(&a, hostX , hostY , n);
3 cuMemAlloc(&devX, mem);
4 cuMemcpyHtoD(devX, hostX, mem);
```

The next step in the program is to prepare data for calculation. This is done in a typical manner: first, the data are allocated; then, they are initialized with some reasonable values. This process is usually realized by classical memory allocation functions and by initialization done by hand. Memory allocation on the device is performed by the **cuMemAlloc(ptr,mem)** function, which allocates mem bytes and sets up the ptr pointer to the allocated area. The **cuMemcpyHtoD** function copies data located in the host memory and pointed by the *devX* pointer to the memory located on the device and pointed by the *devX* pointer. The copying process is synchronous.

```
void *args [] = { &devZ, &a, &devX, &devY, &n };
cuLaunchKernel(saxpy, 2, 1, 1, 8, 1, 1, 0, NULL, args, NULL);
```

The launch is done in two steps. First, the launch arguments which have to be passed to the kernel function are stored in one linear array. Some of them are far pointers, because they are put into the array as addresses. Second, the kernel is launched indirectly, by **cuLaunchKernel** function. This style of kernel launch does not require any special syntax and can be compiled by a regular gcc compiler. The

launch configuration is passed directly to the function as arguments, as well as the pointer to the kernel function and the array of kernel arguments. The whole argument list of **cuLaunchKernel** is defined as follows:

```
1 CUresult cuLaunchKernel(
2   CUfunction f,
3   unsigned int gridDimX, unsigned int gridDimY, unsigned int gridDimZ,
4   unsigned int blockDimX, unsigned int blockDimY, unsigned int blockDimZ,
5   unsigned int sharedMemBytes,
6   CUstream hStream,
7   void ** kernelParams,
8   void ** extra
9 );
```

Here, *f* is just the SAXPY function, called **saxpy** and loaded from the cubin file. The arguments from line 3 denote the grid size for the calculation. Line 4 contains the block dimensions in XYZ directions. Grid and block sizes are expressed by block and thread number, respectively. The next argument is the *sharedMemBytes* variable, which denotes how much dynamic shared memory will be available for each block in the grid. *hStream* is a stream identifier. All arguments passed to the kernel are collected in *kernelParams*. The number of those arguments does not have to be specified, because it is known from the kernel definition. Another way of passing the arguments to the kernel is to use the *extra* argument. This is just a buffer where all arguments are defined by the constant and are given by the pointers, as in the following example:

```
1   size_t argBufferSize;
2   char argBuffer[256];
3
4   // populate argBuffer and argBufferSize
5   void *config[] = {
6     CU_LAUNCH_PARAM_BUFFER_POINTER, argBuffer,
7     CU_LAUNCH_PARAM_BUFFER_SIZE, &argBufferSize,
8     CU_LAUNCH_PARAM_END
9   };
10  status = cuLaunchKernel(f, gx, gy, gz, bx, by, bz, sh, s, NULL, config);
```

In this example the *config* array contains a list of keys followed by values. The keys are predefined constants and are explained below. The values are just integer variables or pointers to data. The list has to be ended with CU_LAUNCH_PARAM_END. The preceding constants define data sizes (CU_LAUNCH_PARAM_BUFFER_POINTER) or pointers to data (CU_LAUNCH_PARAM_BUFFER_SIZE).

Finally, when the calculation is done, the data have to be copied back to the host memory:

```
cuMemcpyDtoH (hostZ, devZ, mem);
```

The vector *devZ* is allocated on the device and contains new calculated data. The data are copied to the memory pointed by *hostZ* on the host.

```
cuMemFree(devX);
free(hostZ);
cuCtxDetach(cuContext);
```

After the calculation is done, the memory and resources have to be released. In the above listing, allocated device memory and host memory are released. Also, the device context is freed.

Appendix B

Theoretical Foundations of Heterogeneous Computing

B.1. Parallel Computer Architectures

The key factor in determining a particular category of parallel computer architecture is its memory arrangement.

B.1.1. Clusters and SMP

The most popular current (2011) architecture is the cluster of interconnected computers where each node uses its own address space and accesses only the local memory, as shown in Fig. 1.1.

The message passing cluster architecture is efficient, since all nodes use their local memories except that processing parallel jobs requires moving data between different nodes. This communication is accomplished by external fast networks such as Myrinet, Infiniband or 10 Gbit Ethernet. The best of them are at least one order of magnitude slower than the internal computer communication. Clusters are efficient parallel machines for solving problems with a coarse granularity, i.e., where independent tasks computed in parallel are large and do not need too much inter-nodal communication. The main attractions of cluster computing are its affordable cost and the ease of increasing cluster power by adding new or faster nodes.

The second most common computer architecture family is shared memory multiprocessors, where all processors have access to a single address space that is either centralized or physically distributed. In the first case, the system is called *SMP* (Symmetric Multi-Processor), as shown in Fig. 1.3.

The SMP is also called the Uniform Memory Access (UMA) computer because all processors can access memory in the same amount of time. The interconnection network can be a simple bus or a more complex network such as cross-bar. No matter which interconnecting network is selected the UMA architecture suffers from the common bottleneck: access to centralized global memory.

B.1.2. DSM and *cc*NUMA

In large multiprocessors, memory is physically distributed. The address space is still single, but each processor has a chunk of memory and a part of the global address space. A version of this architecture is shown in Fig. 1.4.

A common version of DSM is the architecture called *cc*NUMA. In this name, NUMA stands for Non Uniform Memory Access and *cc* for cache coherent. The non-uniformity of memory access time stems from the unequal distance from the processors to the physically distributed data. For any given processor, some data may be closer and other further away, so the access time will be non-uniform. In *cc*NUMA architecture, the interconnect network supports cache coherence that is required for accessing the most recently modified data. Both architectures, the clusters and the SMPs, can be combined into a powerful **hybrid** machine where the cluster nodes are SMP machines. In this case, there are two levels of interconnect. One is the interconnect between nodes and the other is the interconnect within the SMP nodes. Some of the most powerful current supercomputers are hybrid cluster-SMP architectures programmed by combining MPI and Open MP.

B.1.3. Parallel Chip Computer

The most recent parallel computer architecture arrival is the Multi-core Microchip, whose generic structure is shown in Fig. B.1. In this architecture, one core may be responsible for control and others perform parallel processing, so the chip becomes an SMP parallel computer. Alternatively, some remote host CPUs can perform managing functions and the chip becomes an OpenCL device, a parallel co-processing device.

Multi-core processors are SMP systems. Networking them creates NUMA highly parallel architecture machines. A multicore CPU can be a part of a heterogeneous machine, as a device, and act as a parallel accelerator. Each core would process multiple threads called by OpenCL work items. One can say that GPUs are even more parallel. For example, The NVIDIA GPU series 10 can process up to 30000 threads [4] (in OpenCL vocabulary, threads are called work items).

B.1.4. Performance of OpenCL Programs

Amdahl's and Gustafson's Laws

In this section, performance of parallel multiprocessors and heterogeneous systems is considered. It is assumed that a multiprocessor has p processors and f is the fraction of program operations executed in parallel on p processors. The value of f is called the *parallel fraction*. The key notation used in this section is listed in equation B.1.

$$\begin{aligned} S_p &= \frac{T_1}{T_p} - \text{speedup} \\ E_p &= \frac{S_p}{p} - \text{efficiency} \end{aligned} \tag{B.1}$$

Parallel performance notation,
where T_1 – the serial compute time, T_p – the parallel wall clock time.

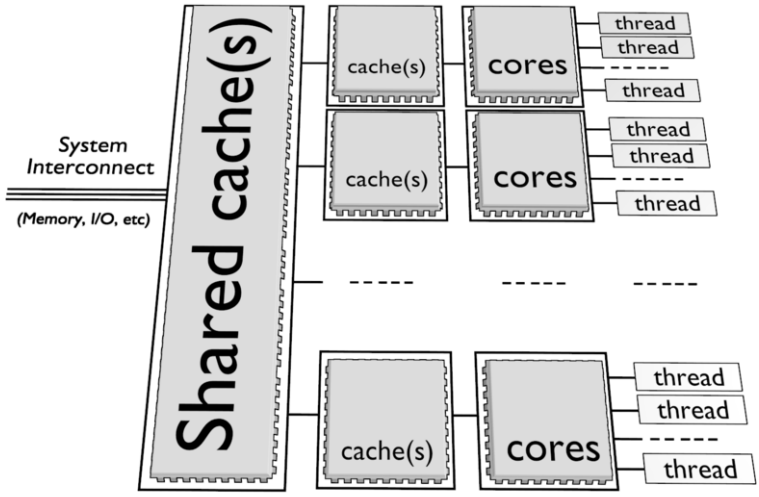

Figure B.1: A generic multi-core microprocessor.

For conventional parallel computing, two laws have been used to predict or estimate parallel performance. The first is Amdahl's Law and the second is Gustafson's Law. Amdahl's Law expresses the speedup in terms of f and p. By definition

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{T_1 \frac{f}{p} + T_1(1-f)} \leq \frac{1}{1-f} \tag{B.2}$$

and the maximal value of the speedup is

$$\max S_p = (1-f)^{-1} \tag{B.3}$$

for $p \to \infty$. The last equation is called Amdahl's Law. It tells us that unless f is very close to 1.0, the achievable speedup is not very impressive, regardless of how many processors are used. For example, if f=0.90 the maximal speedup is only 10. Fig. B.2 shows speedups for several values of f.

On the other hand, Amdahl's Law is too idealized because it ignores interprocessor communication that increases the parallel wall-clock time and reduces the speedup. To provide a more accurate speedup formula and Gustafson's Law the following additional notation is introduced: n - the problem size; T_{comm} - the communication time; $T_{1s} = T_1(1-f)$; $T_{1p} = T_1 f$

$$s = \frac{T_{1s}}{T_{1s} + \frac{T_{1p}}{p}} \tag{B.4}$$

Additional notation where s is the Gustafson serial fraction.

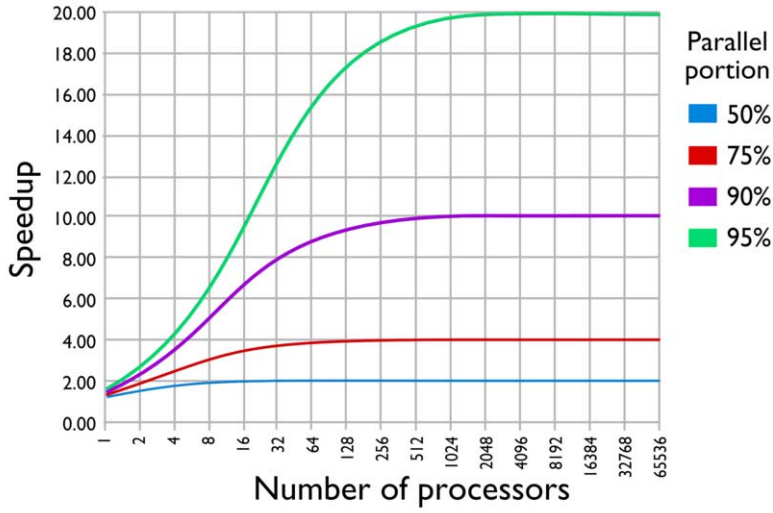

Figure B.2: Amdahl's Law.

Now the speedup, including communication, is depicted in Formula B.5.

$$S_p = \frac{T_{1s} + T_{1p}}{T_{1s} + \frac{T_{1p}}{p} + T_{\text{comm}}} \tag{B.5}$$

Speedup with the communication term.

Notice that these speedup equations apply to simple bimodal programs that are processed either serially or in parallel by p processors. In more general cases, algorithms and related programs have several parallel components with different levels of parallelism. The discussion in this section applies to heterogeneous computations (HC) that are bimodal: partly serial and partly parallel. But if these equations are used for estimating OpenCL platform speedups the variable p has to be replaced by t, which stands for the ratio of the GPU parallel speed to the CPU speed. Both are measured in flops.

$$\text{Speedup}_{\text{HC}} = \left(\frac{f}{t} + (1 - f) \right)^{-1} \tag{B.6}$$

Speedup estimate for heterogeneous computation.

Formula B.6 is analogous to the formula from which Amdahl's Law is derived. Using the definition of s in Formula B.4, Gustafson's Law shown in Formula B.7 can be obtained.

$$\begin{aligned} S_p &= p + (1 - p)s \\ \lim_{s \to 0} S_p &= p \end{aligned} \tag{B.7}$$

Gustafson's Law tells us what has been confirmed by benchmarking tests of large High Performance parallel computations' the parallel speedup increases with decreasing s where s is the Gustafson serial fraction defined in Formula B.4. The fraction s is smaller for larger problem sizes. So, for large problems the speedup approaches p. What has been considered so far is a simple bimodal case where the program runs either sequentially or in parallel on p processors. A more general case considered by [7] is when we deal with an algorithm that has several components whose sizes measured in flops are different and they are executed with variable flop rates.

Assume that the component P_i of size N_i Flops is processed at the rate R_i. All parts are executed consecutively in some order. The bimodal computing is a special case for i=2. In the general case, the average flop rate can be calculated from the equation B.8.

$$R_{\text{Average}} = \frac{N}{\sum_i \frac{N_i}{R_i}} \tag{B.8}$$

where

$$N = \sum_{i=1}^{n} N_i \tag{B.9}$$

General performance estimation.

Both Amdahl's Law and Gustafson's Law do not take into account communication in parallel computation. Several attempts have been made to include communication into the parallel computing performance analysis. One of them is by Karp-Flatt [3], who established an experimental method for detecting and analyzing impacts of communication on parallel computing performance. A sketch of the Karp-Flatt method is presented below. Recall that the total parallel time for parallel program execution is:

$$T_{Tot}(n,p) = T_{1s} + \frac{T_{1p}}{p} + T_{\text{comm}} \tag{B.10}$$

And $T_1 = T_{1s} + T_{1p}$. A serial fraction measuring total sequential time including communication is:

$$e = \frac{(T_{1s} + T_{\text{comm}})}{T_1} \tag{B.11}$$

Recall that speedup has been defined as

$$S_p = \frac{T_1}{T_{Tot}} \tag{B.12}$$

The value of e can be computed from these last four equations as a function of the experimentally determined speedup and p [3].

$$e = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{B.13}$$

Now it is possible to calculate the serial fraction e that measures the serial part of computing, including the communication time. Calculating the values of e can help in understanding the reasons for inefficiency in poorly running programs. Suppose, for example, that e is constant for benchmarks with increasing p but the tests show poor efficiency. This would indicate that the reason for poor performance is not communication but more likely the limited parallelism in the program. For more discussion of the Karp-Flatt method the reader is referred to [3].

At this point, it must be stressed that one of the most significant GPU performance limitations is global memory bandwidth. One of the consequences is the slow data transfer from the CPU to the GPU and vice-versa.

In applications of heterogeneous computers, transfer of data between the CPU and the GPU has to be reduced to the minimum to avoid very heavy performance degradation. To illustrate the data transfer problem consider two popular mathematical algorithms from the perspective of data transfer in heterogeneous computation. One is the computation of the product of two matrices. The kernel computes the product only once. No iterations are needed. Algorithm flow and data transfer are shown in Fig. B.3.

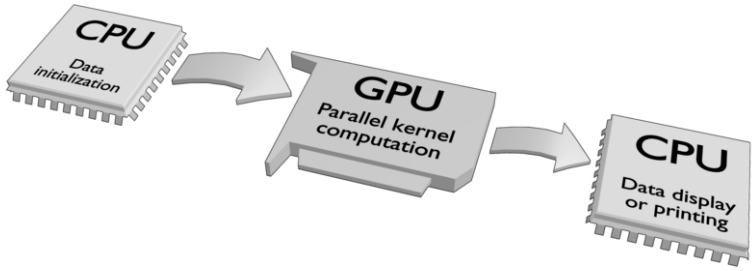

Figure B.3: The matrix/matrix product algorithm flow and data transfer.

Data transfer is indicated by the thick arrows. The matrix-matrix multiplication algorithm is executed by GPU in one shot without algorithmic iterations. The data transfer for a more complex CGM algorithm is shown in Fig. 1.13 in section 1.6.2.

In that version of CGM, it is assumed that the GPU executes the entire computation of every iteration. CGM is discussed further, and test results are presented, in sections 1.6 and 1.7. In this fast version, it is assumed that the entire iteration is executed in parallel by the GPU. Initially, the algorithm transfers data from the CPU to the global or local device memory. Later, after the iterations are finished, the

results computed by the GPU are transferred back to the CPU for display, printing or storage. The algorithm implementation requires only one bit transfer from the global memory to RAM after every iteration. This bit indicates if the iterations converged. In this implementation, all data transfers except the initial transfer have been avoided. The impact of one bit transfer between iterations can be ignored. Designing efficient algorithms for the OpenCL platforms requires minimization or elimination of data transfers between the host and the GPU and placing kernel data in the device local memory rather than in the slower global memory. This is analogous to reusing cache memory instead of RAM in a conventional computer.

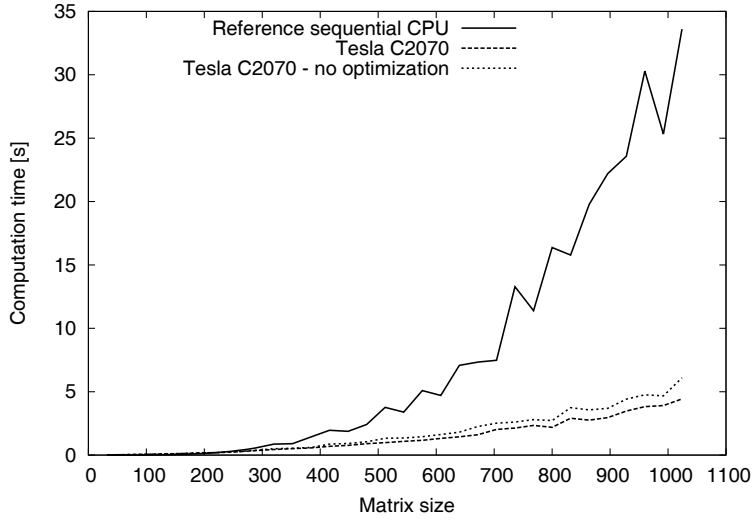

Figure B.4: The performance results for matrix/matrix multiplication: CPU and Tesla 2070 GPU.

The results shown in Fig. B.4 demonstrate a very substantial speedup over the straightforward use of the global memory. For large matrices, the execution time difference is about two orders of magnitude.

Comparing CPU with GPU

In 2010, HPCwire [31] published an interesting comparison of peak performances of an Intel processor and an NVIDIA GPU Tesla C2060. The Intel processor was a Xeon X6660 (Nehalem) 2.68 GHz, 48 GB memory, $7K and 0.66 kW energy consumption. The comparison included three values, shown in Table B.1.

More recently, Natoll [32] of Stone Ridge Technology compared the Intel CPU Westmere, which has 6 cores running at up to 3.4 GHz, with NVIDIA Fermi M2090 having 512 cores running at about 1.3 GHz. His bottom-line estimation was that the GPU would boost compute bound codes between 5 and 20 times.

Table B.1: Comparision between Intel CPU and Tesla GPU

	Unit	Intel CPU	Tesla GPU
Peak performance	Gflops	80.1	656.1
Peak performance per $1000	GF/dollars	11	60
Peak performance per watt	Gflops/watt	145	656

For the memory-bound codes, the expected boost is expected to be about 5 times in chip to chip comparison. Experiments we have seen or performed ourselves support these estimates.

B.2. Combining MPI with OpenCL

OpenCL can be combined with MPI in a way similar to how OpenMP has been combined with MPI on computer clusters whose nodes are shared memory SMPs (Fig. B.5). In the case of MPI and OpenMP, the assumption is that the cluster nodes are heterogeneous platforms.

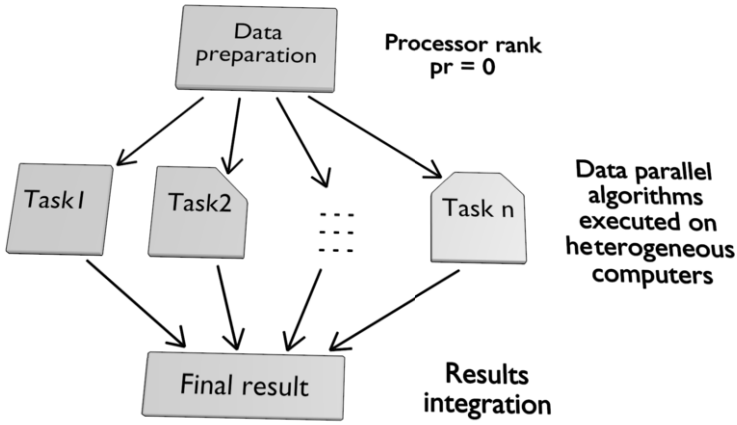

Figure B.5: MPI and OpenCL

It often happens that a scientific or an engineering computation problem can be partitioned into coarse-grain parallel tasks that contain internal fine-grain data parallelism. In this case, the job of major task definition and data preparation/communication could be done by MPI code, and data-parallel execution by the heterogeneous nodes of the cluster.

In his book [33], Ian Foster suggested an elegant method for developing parallel programs that included four steps: partition, communication, agglomeration and mapping. In the case of combining MPI and OpenCL, the method is a modification and includes: partition, data distribution, task execution on hererogeneous platforms, and solution integration from solution parts.

Appendix C

Matrix Multiplication – Algorithm and Implementation

C.1. Matrix Multiplication

One of the most commonly used linear algebra operations is matrix multiplication. Detailed notation is provided here, since it is going to be reused later in this section.

The matrices A and B are of size $m \times n$ and $n \times p$, respectively. These matrices will be called A and B, each of them having elements a_{ij} and b_{ij} accordingly. Matrix A will look like the representation in equation C.1.

$$d\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \\ a_{m1} & a_{m2} & & a_{mn} \end{bmatrix} \tag{C.1}$$

The matrix called C is the result of multiplication of two matrices called A and B, both of size $n \times n$. Each element of C is calculated as shown in equation C.2.

$$c_{ij} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \cdots + a_{i,n}b_{n,j} = \sum_{k=1}^{n} a_{i,k}b_{k,j} \tag{C.2}$$

C.2. Implementation

C.2.1. OpenCL Kernel

The formula C.2 can easily be translated into any programming language. On a standard CPU, the whole algorithm would involve three levels of loops – first two for traversing through the resulting matrix, and the innermost for calculating c_{ij}. This

```
// notation: i - row, j - column
kernel void matrix_mul(global float *A, global float *B, global float *C, int n) {
  int i = get_global_id(1);
  int j = get_global_id(0);
  int k;
  float c = 0;
  if (i > n) return;
  if (j > n) return;
  for (k = 0; k < n; k++) {
    c = c + A [i * n + k] * B [k * n + j];
  }
  C [i * n + j] = c;
}
```

Listing C.1: Matrix multiplication - the kernel code.

operation has the computational complexity $O(n^3)$ and is suitable for highly parallel computation using OpenCL[1].

In order to keep the examples simple, this code multiplies only the square matrices of size $n \times n$.

The function **get_global_id** retrieves a global work-item ID for a given dimension. The first two lines set i and j coordinates for the local ID-s. This construction supplants two levels of loops, as seen in the sequential version, and allows the programmer to focus on the merits of the algorithm. Here, it is the resulting item value as defined by equation C.2.

As the reader might notice, the matrices are stored as 1-dimensional arrays. This is because OpenCL allows only for passing one-dimensional buffers of simple or complex data types or simple variable values. This is why, instead of using $A[i][j]$, the construction $A[i * n + j]$ has to be used. This limitation can be bypassed using macros or functions in the OpenCL program.

C.2.2. Initialization and Setup

The setup of the OpenCL context is done in much the same way as in the example in section 2.12. The command queue is defined to be in-order and consists of only one device. The difference is that the OpenCL program is loaded from a file and errors are checked.

OpenCL program compilation is done by the C function shown in listing C.2. This code loads the whole OpenCL program from a selected file into the memory buffer. In the next step, it creates a program object. The last step is to build this program into executable form for a given computational context.

The OpenCL API does not have a function for loading a source file into the memory buffer[2]. The standard ANSI C library does not contain any function for check-

[1] There are also algorithms which compute matrix multiplication in $O(2^{2.8})$ but are not suitable for this book.

[2] Some OpenCL vendors provide functions for this task, like oclLoadProgSource on NVidia hardware, but it is not a part of the OpenCL API.

```
cl_program buildProgramFile(cl_context context, char *filename){
  cl_int clErrNum;
  size_t size;
  cl_program program;
  char *appsource;
  FILE *f = fopen(filename, "r");
  if (f == NULL) {
    printf("Error: Could not open '%s'.", filename);
    exit(EXIT_FAILURE);
  } else {
    fseek(f, 0L, SEEK_END);
    size = ftell(f);
    fseek(f, 0L, SEEK_SET);
    appsource = (char *)malloc(size);
    size = fread(appsource, 1, size, f);
    fclose(f);
  }
  program =
    clCreateProgramWithSource(context, 1,
      (const char **)&appsource, &size, &clErrNum);

  clErrNum = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  if (clErrNum != CL_SUCCESS) {
    size_t len;
    char buffer [4096];
    printf("Error: Could not build OpenCL program (%d)\n", clErrNum);
    cl_device_id device_id [16];
    clErrNum = clGetContextInfo(context,
      CL_CONTEXT_DEVICES, sizeof(device_id), &device_id, NULL);
    clGetProgramBuildInfo(program, device_id [0],
      CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
    printf("%s\n", buffer);
    exit(EXIT_FAILURE);
  }
  free(appsource);
  return program;
}
```

Listing C.2: Function for loading and building OpenCL program. This function checks for syntax and file access errors.

ing the file size. This is why it is checked using `fseek` and `ftell` commands[3]. This method has been chosen in order to keep code as portable as possible. The program source is stored in the memory buffer named *appsource*.

The function **clCreateProgramWithSource** creates an OpenCL program object based on source code. This function expects an array of pointers to character strings. The length of this array is passed in second parameter. In the example, only one source code buffer is used, but it is very easy to load multiple files at once. For more advanced use of this function, see chapter 2.12.2.

[3]This method does not work on files that do not permit seeking, like sockets and FIFOs.

The last step is to build the program. This is done using the function **clBuildProgram**. In contrast to the example in section 2, this time error checking is performed. As in every OpenCL API call, the error is denoted by an error number different than CL_SUCCESS.

clGetProgramBuildInfo returns build information for the device in the program object. In order to retrieve information about the compilation process, CL_PROGRAM_BUILD_LOG is passed as a third parameter. The output is in human-readable form, so it can be printed without any additional parsing. The output string is implementation-dependent, so it can look different on various OpenCL libraries. This function needs to know which device in the computation context to query. It is assumed that our context uses only one device. In this example, the function **clGetContextInfo** is used to obtain the list of devices in the computational context. Only the first one is passed to **clGetProgramBuildInfo**.

C.2.3. Kernel Arguments

Setting buffers and parameters for the kernel is shown in listing C.3. The type "cl_float" is used instead of "float". This is because in some rare circumstances, the data size of the types can differ and it is good to write code that is implementation independent. Note that for the simplicity of the example, the matrices are initialized statically and its size is set to 3. In C two-dimensional arrays are stored in memory contiguously by rows, so they can be treated as one-dimensional buffers. This example can easily be extended to dynamic matrices.

The function **clCreateBuffer** creates a memory buffer in the OpenCL context. This buffer can physically be stored in the host or the device memory. In the example, the second solution is used. If the buffer is stored in the host memory, the data can be cached by the device and the slow communication usually impacts program performance negatively. This function can also copy the data from a given host buffer.

In the example, all the data are copied into the device memory. The parameter CL_MEM_COPY_HOST_PTR orders OpenCL to copy data from host buffers "hostA" and "hostB" to newly created buffers "devA" and "devB" accordingly.

The kernel can access data only from its context. Unlike in CUDA, a kernel cannot be called with arguments set in brackets. The arguments have to be set using the API call **clSetKernelArg**. Variables with simple types can be passed directly via this call, but buffers have to be initialized in the computational context first. The kernel argument numbers are indexed from 0. The last line is an example of passing a single number as a kernel parameter.

In this part,errors are silently ignored because it has already been checked if the OpenCL program compiled correctly and the kernel is loaded. The most probable reason for potential errors is not enough memory, but it is very unlikely that this example would need more than a few kilobytes of RAM.

C.2.4. Executing Kernel

The problem space is two-dimensional. Because the goal is to gain as much performance and code simplicity as possible, the kernel has to be executed in two dimensions, so $n \times n$ kernel instances will be executed.

```
cl_int n = 3; // the size of square matrix
cl_float hostA [3] [3], hostB [3] [3], hostC [3] [3];
cl_mem devA, devB, devC; // pointers to the memory on the device

hostA [0] [0] = 1; hostA [0] [1] = 0; hostA [0] [2] = 3;
hostA [1] [0] = 0; hostA [1] [1] = 2; hostA [1] [2] = 2;
hostA [2] [0] = 0; hostA [2] [1] = 0; hostA [2] [2] = 3;

hostB [0] [0] = 1; hostB [0] [1] = 0; hostB [0] [2] = 0;
hostB [1] [0] = 0; hostB [1] [1] = 4; hostB [1] [2] = 0;
hostB [2] [0] = 0; hostB [2] [1] = 0; hostB [2] [2] = 1;

// we have to prepare memory buffer on the device
devA = clCreateBuffer(context,
  CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  n * n * sizeof(cl_float), hostA, NULL);
devB = clCreateBuffer(context,
  CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  n * n * sizeof(cl_float), hostB, NULL);
devC = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
  n * n * sizeof(cl_float), NULL, NULL);
// set parameters for kernel
clSetKernelArg(kernel, 0, sizeof(cl_mem), &devA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &devB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &devC);
clSetKernelArg(kernel, 3, sizeof(cl_int), &n);
```

Listing C.3: Setting kernel parameters for matrix multiplication.

```
// add execution of kernel to command queue
size_t dev_global_work_group_size [2] = { n, n };
ret = clEnqueueNDRangeKernel(queue, kernel,
  2, NULL,
  dev_global_work_group_size,
  NULL, 0, NULL, NULL);
// retrieve results
ret = clEnqueueReadBuffer(queue, devC, CL_TRUE,
  0, n * n * sizeof(cl_float),
  hostC, 0, NULL, NULL);
```

Listing C.4: Matrix multiplication kernel execution.

The function **clEnqueueNDRangeKernel** is used to enqueue kernel execution. This function does not wait for a command to finish. The first two parameters define which kernel to execute and on which command queue. The third parameter defines the problem dimensions count. It is very important to always match this value with the problem dimensions count when possible. The next parameter is NULL, meaning that OpenCL implementation should choose the local work group size. This parameter and its influence on execution performance will be discussed in detail in section 3.2. The next parameter defines the size of the global work group. This param-

eter is an array of two `size_t` elements.

The last three parameters are for events. These parameters are discussed in section 2.8.

The function **`clEnqueueReadBuffer`** is used for reading the memory buffer from the OpenCL device to the host memory. Here, it is set to work as a blocking function, so the use of **`clFinish(queue)`** is not necessary. The advantage of a queue is that all the needed commands can be put into it and only one synchronization point is needed at the very end. This model of synchronization is usually beneficial for application performance.

Comments and Exercises

The source code for this example is on the CD-ROM attached to the book.

This program can be extended to work with matrices of any given size. This can be done by adding another parameter to the kernel. Try to achieve this functionality.

Not all errors are checked; try to fix it by adding appropriate checking in memory initialization, setting kernel arguments and kernel execution.

Appendix D

Using Examples Attached to the Book

The book contains multiple listings extracted from the full example applications that are located on the attached storage. This approach allows the reader to focus on the important parts of the code. Every example is compatible with OpenCL 1.1, because there are no 1.2 platforms available yet (November 2011). The source codes attached to the book can be found in the directories named according to the section name and number. The common parts that are used by the examples are stored in the directory named `common`. There are applications written in the C and C++ programming languages. Both versions are always located in one directory. For example, consider these sections:

> OpenCL Context to Manage Devices
> OpenCL Context to Manage Devices – C++

The example application in both versions – C and C++ – is located in directory `2.05.OpenCL_Context_to_Manage_Devices`.

There are also directories named `0.00.skeleton.*`; these directories contain skeletons of OpenCL applications for different operating systems, languages and compilers. Skeleton applications were created in order to simplify the initial part of creating projects. The reader who would like to write an OpenCL application on his own is encouraged to try starting with the skeletons. It is also worth mentioning that the examples usually use the shared code located in `common` directory. The skeletons are independent and can be used without any additional modifications as a starting point of an OpenCL application.

Every example application was tested with GCC version 4.4, 4.5 and MinGW GCC version 4.6. Currently (fall 2011), only OpenCL 1.1 implementations are available. The code samples provided will work with both OpenCL 1.2 and OpenCL 1.1.

D.1. Compilation and Setup

D.1.1. Linux

The example applications will compile without any additional operations on almost all modern Linux distributions. There is, however, a requirement that header files for libraries be installed, as well as a compiler and OpenCL implementation. For example, on a Debian-like distribution, the packages that must be installed are:

- gcc
- libSDL-dev
- mesa-common-dev

For Linux console users, compilation of the attached sources is done using the **make** tool. Every example contains `Makefile`, which is required by the **make** tool. The user must execute the command that can be seen in listing D.1 in order to compile the example.

```
make
```

Listing D.1: Compilation of source code examples under Linux operating system.

D.1.2. Windows

The example applications were tested under Windows 7 and Windows XP with SP3. Available OpenCL software development kits (SDK) provide library and header files for developing OpenCL applications. Note that some OpenCL platform SDKs provide old versions of header files. In this case, the reader is advised to download OpenCL header files from the Khronos OpenCL standard webpage. Because the Windows operating system does not provide a standard directory for storing header files, they must be manually copied into a compiler `include` directory. The OpenCL–OpenGL example uses the SDL library. This library must be downloaded in the development version. It is freely available as open-source software. This library also provides its header files and libraries that must be copied into the compiler header and library directories.

There are instructions in the following subsections on how to configure an environment for compiling the examples. Note that some OpenCL Software Development Kits do not provide the most recent version of the OpenCL linker library. This can produce errors during the linking stage of compilation. To solve this, the user can try to use deprecated 1.0 or 1.1 API or use a different, more recent SDK. The book examples use the 1.1 and 1.2 API versions. There are also SDKs that do not provide `cl.hpp`, but remember that header files are available on the Khronos OpenCL web page.

MinGW

Before the compilation, the MinGW must know about OpenCL. The header files for OpenCL must be located in the MinGW `include` directory. The user must copy `opencl.h`, `cl_platform.h`, `cl.h`, `cl_gl.h`, `cl.hpp` and other header files listed on the Khronos OpenCL standard web page into the directory `include\CL`. For default MinGW installation, it should be `C:\MinGW\include\CL`. The OpenCL–OpenGL example also requires an SDL library. The header files must also be copied into the `include` directory of MinGW compiler. Remember that SDL header files must be in a subdirectory called `SDL`.

The library file that is needed by MinGW is called `libOpenCL.a`. This file must be copied into the MinGW library directory called `lib`. In a default installation, it is called `C:\MinGW\lib`. The OpenCL–OpenGL example requires library files provided by the SDL. The development version of this library provides appropriate files – the `.a` and `.h` files. The SDL.dll must be copied into `windows\system` directory. Compilation of OpenCL examples is done using the **`make`** command executed from the MinGW shell in the example directory.

Visual C++ 10

Visual C++ requires header and library files for OpenCL and SDL. The header files from both libraries must be put into the Visual C++ installation directory, under the subdirectory called `include`. This directory is located in
`C:\Program Files\Microsoft Visual Studio 10.0\VC\include`
by default. The header files for OpenCL must be put into the subdirectory called `CL`, and SDL files into `SDL` under the `include` directory.

Visual C++ searches library files in its installation directory, under the subdirectory called `lib`. In a default installation, this directory is called
`C:\Program Files\Microsoft Visual Studio 10.0\VC\lib`
The obligatory file for OpenCL is named `opencl.lib`, and library files for the SDL are called `SDL.lib` and `SDLmain.lib`.

OpenCL applications written in Visual C++ must be linked against `opencl.lib`. The option that performs this operation is located in project settings, under the `Linker/General` section. The option is called `Additional dependencies`.

Most of the examples are accompanied with a Visual C++ project file, so in most cases, the user can open example applications without any additional activity. The skeleton application for Visual C++ is also included and is located in `0.00.skeleton.windows.visualcpp.cpp`.

Note that some OpenCL software development kits integrate with Visual C++ during the installation process, so in this case it is not necessary to copy header and library files for VC++.

Bibliography and References

[1] E.Lusk W.Gropp and A.Skjellum. *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.

[2] G.Jost B.Chapman and R.van der Pas. *Using OpenMP, Portable Shared Memory Parallel Programming*. The MIT Press, 2008.

[3] M.J.Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, 2004.

[4] D.B.Kirk and Wen mei W.Hwu. *Programming Massively Parallel Processors, A hands-on Approach*. NVIDIA Morgan Kaufmann, 2010.

[5] R.Tsuchiyama et al. *The OpenCL Programming Book, Parallel Programming for Multicore CPU and GPU*. Fixstars Corporation and Impress Japan Corporation, 2009.

[6] G.H.Golub and C.F.Van Loan. *Matrix Computations, The second edition*. The Johns Hopkins University Press, 1990.

[7] D.C.Sorenson J.J.Dongarra, I.S.Duff and H.A.van der Vorst. *Numerical Linear Algebra for High-performance Computers*. Software Environments Tools SIAM, 199.

[8] R.W.Shonkwiler and L.Lefton. *An Introduction to Parallel and Vector Scientific Computing*. Cambridge Texts in Applied Mathematics, Cambridge University Press, 2006.

[9] R.A.van de Gejn with contributions from others. *Using LAPACK*. The MIT Press, 1997.

[10] Aaftab Manush, editor. *OpenCL Specification version 1.2*. Khronos OpenCL Working Group, 2011.

[11] *http://www.jocl.org/*, 2011.

[12] *http://mathema.tician.de/software/pyopencl*, 2011.

[13] *http://ruby-opencl.rubyforge.org/*, 2011.

[14] *http://www.nvidia.com/ content/ PDF/ fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf*, 2011.

[15] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[16] 754-1985 ieee standard for binary floating-point arithmetic. Technical report, IEEE, 1985.

[17] 754-2008 ieee standard for floating-point arithmetic. Technical report, IEEE, 2008.

[18] *http://www.opengl.org/registry/*, 2011.

[19] *http://www.libsdl.org/*, 2011.

[20] *http://developer.apple.com/ library/ mac/ #documentation/ GraphicsImaging/ Reference/ CGL_OpenGL/ Reference/ reference.html#// apple_ref/ c/ func/ CGLGetCurrentContext*, 2011.

[21] Pat Brown Chris Frazier, Jon Leech, editor. *The OpenGL Graphics System: A Specification (Version 4.2 (Core Profile) – August 8, 2011)*. The Khronos Group Inc, 2011.

[22] Sean Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[23] J.H.Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[24] D.E.Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, 1989.

[25] Z.Michalewicz. *Genetic algorithms + data structures = evolution programs (3rd ed.)*. Springer-Verlag, London, UK, 1996.

[26] Lenore Blum, Manuel Blum, and Mike Shub. A simple unpredictable pseudo-random number generator. *SIAM J. Comput.*, 15(2):364–383, 1986.

[27] NVIDIA Corporation. *CUDA 4.0 Release Notes*. NVIDIA Corporation, May 2011.

[28] NVIDIA Corporation. *CUDA Programming Guide, ver. 4.0*. NVIDIA Corporation, May 2011.

[29] *CUDA User Manual, ver. 4.0*. NVIDIA Corporation, May 2011.

[30] E.Kandrot J.Sanders. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. NVIDIA Corporation, 201.

[31] *http://hpcwire.com/*, May 2010.

[32] V.Natoll. *Top 10 Objections to GPU Computing Reconsidered*. HPCwire, June June 9 2011.

[33] I.Foster. *Designing and Building Parallel Programs*. Addison-Wesley Publishing Company, 1994.

Index

Symbols

LaTeX, viii
__device__, 260
__global__, 256, 257, 262
__CL_ENABLE_EXCEPTIONS, 53, 114, 128, 197
__CL_USER_OVERRIDE_ERROR_STRINGS, 54
__attribute__, 191
__device__, 259
__global__, 253, 256, 259
__host__, 259
__kernel, viii, 81, 90

A

Albert Einstein, vii
AMD, 39
application
  in OpenCL, 32
  OpenCL example, 106
atomic_inc, 99

B

barrier, 8, 73, 74, 95, 192, 233
  OpenMP, 8
barrier, 94, 95, 160, 192, 232–234
Blender, viii
blockDim, 256
blockDim, 256
blockIdx, 256
broadcast
  MPI, 2
Buffer Object, 32
build log, 107

C

calloc, 252
Cartesian, 2
CGM, 15, 19, 20, 24–26, 122, 139, 142, 169–171, 173, 177–182, 274
chromosome, 221
cl... constants:
  ..._khr_fp64, 153
  ..._khr_gl_sharing, 195, 201
cl... functions:
  ...BuildProgram, 85, 106, 109, 137, 157, 282
  ...CreateBuffer, 75, 112, 127, 135, 282
  ...CreateCommandQueue, 56, 110, 134
  ...CreateContext, 46, 106
  ...CreateContextFromType, 45, 47, 104, 134
  ...CreateFromGLBuffer, 208, 209
  ...CreateKernel, 87, 137
  ...CreateKernelsInProgram, 86, 87
  ...CreateProgramWithBinary, 106–108
  ...CreateProgramWithSource, 82, 106, 137, 281
  ...EnqueueAcquireGLObjects, 213–215
  ...EnqueueCopyBuffer, 78, 79
  ...EnqueueNDRangeKernel, 67, 112, 137, 159, 160, 283
  ...EnqueueReadBuffer, 77–79, 112, 127, 139, 284
  ...EnqueueReleaseGLObjects, 214
  ...EnqueueTask, 57, 58, 70
  ...EnqueueWriteBuffer, 77–79, 135
  ...Finish, 60, 61, 139
  ...Finish(queue), 284
  ...Flush, 60
  ...GetContextInfo, 47, 134, 282
  ...GetDeviceIDs, 37, 106
  ...GetDeviceInfo, 39, 104, 106, 108, 148, 149
  ...GetEventProfilingInfo, 62
  ...GetExtensionFunctionAddress, 150, 151
  ...GetExtensionFunctionAddressForPlatform, 150
  ...GetKernelInfo, 86, 87
  ...GetPlatformIDs, viii, 35, 45, 104, 106, 133
  ...GetPlatformInfo, 35, 36, 148
  ...GetProgramBuildInfo, 107, 282
  ...GetProgramInfo, 106, 109
  ...ReleaseCommandQueue, 113
  ...ReleaseContext, 113
  ...ReleaseEvent, 176
  ...ReleaseKernel, 113, 139
  ...ReleaseMemObject, 112, 139
  ...ReleaseProgram, 113, 139
  ...RetainEvent, 176

...SetKernelArg, 88, 106, 112, 137, 282
...WaitForEvents, 63
cl... types:
..._context_properties, 45, 47, 201
..._device_type, 37, 43
..._int3, 92
..._int4, 92
..._mem, 209
..._profiling_info, 61
cl::... constants:
...NullRange, 68
cl::... functions:
...Buffer, 76
...Buffer::Buffer, 76
...BufferGL, 208
...BufferGL::BufferGL, 209
...CommandQueue, 56
...CommandQueue::CommandQueue, 56
...CommandQueue::enqueueAcquireGLObjects, 213−215
...CommandQueue::enqueueCopyBuffer, 79
...CommandQueue::enqueueNDRangeKernel, 67
...CommandQueue::enqueueReadBuffer, 78, 79
...CommandQueue::enqueueReleaseGLObjects, 214
...CommandQueue::enqueueTask, 57, 59
...CommandQueue::enqueueWriteBuffer, 78, 79
...CommandQueue::finish, 60, 61
...CommandQueue::flush, 60
...Context::Context, 49
...Context::getInfo, 49, 50
...Device::getInfo, 41, 148, 149, 166
...Event::getProfilingInfo, 63
...Event::setCallback, 175
...Event::wait, 63
...Kernel::getInfo, 86, 87
...Kernel::Kernel, 87
...Kernel::setArg, 88
...KernelFunctor::KernelFunctor, 166
...Platform::get, 40
...Platform::getDevices, 41
...Platform::getInfo, viii, 40, 148
...Program, 82
...Program::build, 85
...Program::createKernels, 86, 87
...Program::Program, 82
...Queue::enqueueTask, 70
cl::... types:
...KernelFunctor, 88, 166
...NDRange, 67
...Platform, 40
...Program::Sources, 82
...Source, 128
cl::... variables:
...KernelFunctor, 236
CL_... constants:
...COMPLETE, 175
...DEVICE_DOUBLE_FP_CONFIG, 91, 166
...DEVICE_EXTENSIONS, 149
...DEVICE_MAX_WORK_ITEM_DIMENSIONS, 65
...DEVICE_NOT_FOUND, 53
...DEVICE_TYPE_ACCELERATOR, 43−45, 48
...DEVICE_TYPE_ALL, 40, 43, 45, 48, 201
...DEVICE_TYPE_CPU, 43, 48
...DEVICE_TYPE_CUSTOM, 43, 48
...DEVICE_TYPE_DEFAULT, 43, 48, 201
...DEVICE_TYPE_GPU, 43−45, 48, 134, 201
...INVALID_CONTEXT, 50
...MEM_ALLOC_HOST_PTR, 77
...MEM_COPY_HOST_NO_ACCESS, 77
...MEM_COPY_HOST_PTR, 77, 112, 135, 282
...MEM_COPY_HOST_READ_ONLY, 77
...MEM_COPY_HOST_WRITE_ONLY, 77
...MEM_READ_ONLY, 77, 112, 135
...MEM_READ_WRITE, 77
...MEM_USE_HOST_PTR, 77
...MEM_WRITE_ONLY, 112, 135
...MEM_WRITE_ONLY , 77
...OUT_OF_HOST_MEMORY, 50
...OUT_OF_RESOURCES, 50
...PLATFORM_EXTENSIONS, 36, 148, 149
...PLATFORM_PROFILE, 36
...PLATFORM_VERSION, 36
...PROGRAM_BINARIES, 109
...PROGRAM_BINARY_SIZES, 109
...PROGRAM_BUILD_LOG, 282
...QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, 58
...QUEUE_PROFILING_ENABLE, 62
...RUNNING, 175
...SUBMITTED, 175
...SUCCESS, 50
...TRUE, 139
CL_... variables:
...DEVICE_TYPE_ACCELERATOR, 38
...DEVICE_TYPE_ALL, 37
...DEVICE_TYPE_CPU, 37
...DEVICE_TYPE_CUSTOM, 38
...DEVICE_TYPE_DEFAULT, 38
...DEVICE_TYPE_GPU, 38
CLK_... constants:
...GLOBAL_MEM_FENCE, 95
...LOCAL_MEM_FENCE, 95
Command, 55
Command-Queue, 55
command-queue, 41, 55
communication, 1, 3, 11, 22, 99, 269, 271−274, 277, 282

compute capability, 247
consistency, 73, 74, 95
Context, 41
context, 41
CPU_... constants:
...COMPILER_OPTIONS, 158
...MAX_COMPUTE_UNITS, 158
crossover, 222
CU_... constants:
...LAUNCH_PARAM_BUFFER_POINTER, 266
...LAUNCH_PARAM_BUFFER_SIZE, 266
...LAUNCH_PARAM_END, 266
cubin, 262
CUDA, vii, 13, 15, 24, 88, 171, 181, 182, 236, 244–249, 251, 253–256, 259–264, 282
blockDim, 256, 258, 259
blockIdx, 256
compute capability, 247
deviceQuery, 247
threadIdx, 256
CUDA_... constants:
...ERORR_NOT_FOUND, 265
...ERROR_NOT_INITIALIZED, 265
cudacc, 262
cudaErrorMemoryAllocation, 252
cudaGetDeviceProperties, 247
cudaGetErrorString, 260
cudaGetLastError, 261
cudaMalloc, 252, 260
cudaMemcpy, 252, 253
cudaMemcpyDeviceToDevice, 253
cudaMemcpyDeviceToHost, 253, 260
cudaMemcpyHostToDevice, 253
cudaMemcpyHostToHost, 253
cudaPeekAtLastError, 261
cudaSuccess, 252
cutilSafeCall, 261

D

Data Parallel Programming Model, 65
decode, 221
device memory, 72
deviceQuery, 247
dim3, 254, 256
double, 166, 248

E

EMBEDDED_PROFILE, 36
event, 58
event wait list, 58

F

finish, 179
fitness function, 222
float, 248
FORTRAN, 1, 3, 4
FPGA, vii
FULL_PROFILE, 36

G

gcc, 262
get_global_id, 65, 66, 68, 106, 136, 280
get_global_offset, 66
get_global_size, 66
get_group_id, 66
get_local_id, 65, 66, 68
get_local_size, 66
get_num_groups, 66
get_work_dim, 67
gl... functions:
...BindBuffer, 196, 199, 206, 219
...BindTexture, 206, 207, 216, 220
...BufferData, 196, 200, 206
...ClearColor, 200
...DeleteBuffers, 196, 200, 219
...DeleteTextures, 220
...Enable, 200
...Finish, 219
...Flush, 214
...GenBuffers, 196, 199, 206
...GenTextures, 206, 207
...LoadIdentity, 200, 201
...MatrixMode, 200, 201
...Ortho, 200, 201
...PixelStorei, 206, 207
...TexParameteri, 207
...TexSubImage2D, 216, 217
...Viewport, 200, 201
GL_... constants:
...ARRAY_BUFFER, 206
...LINEAR, 207
...NEAREST, 207
...RGBA8, 207
...UNSIGNED_BYTE, 208
Global ID, 65
gridDim, 256

H

HPC, vii
HPCwire, 275

I

individual, 222

interconnected
computers, 27, 269
processes, 1
processors, 27

K

Kernel, 81
kernel, 11–14, 16, 17, 24, 28–32, 35, 41, 43, 44, 55, 57–59, 61, 65–71, 73, 75–77, 80–82, 85–94, 96, 97, 99, 100, 103, 106–109, 112–116, 121–124, 126, 127, 129, 130, 132, 135–144, 155–163, 165–167, 172–175, 177–183, 185–191, 193, 194, 201, 202, 209, 211–213, 215, 217, 218, 226–228, 230–232, 234–236, 238–241, 248, 253, 255–257, 259–262, 264–266, 274, 275, 280, 282–284
kernel keyword, 80, 81, 86, 90, 106
Kernel Object, 81
Khronos Group, vii, viii, 13, 28, 150

L

length, 213
Local ID, 65
loop, 2, 6–8, 29, 47, 50, 100, 120, 123, 132, 135, 140, 142, 156, 175, 183, 191, 218, 219, 226, 230, 233–235, 240, 260, 279, 280

M

make, 286
malloc, 252
matrix multiplication, 279
mem_fence, 95
memory pool, 72
memory region, 72
MPI, vii, 1–4, 9, 11, 13, 65, 270, 277
gather, 2
messages, 1–3
performance, 3
process, 1
scatter, 2
mutation, 222

N

NDRange, 65, 100
new, 252
normalization, 92
normalize, 93
NULL, 252
nvcc, 249, 262
NVIDIA, vii, 12, 13, 15, 19, 24, 33, 39, 85, 171, 181–183, 244–247, 249, 251, 260, 270, 275

O

OpenCL
applications, 18, 25
CGM results, 24, 25
components, 16, 17
CUDA, 13
goals, 13
history, 13
introduction, 1, 9, 11, 12
kernels, 17
machine, 25
preface, vii
SAXPY, 122, 123
SAXPY kernel, 123
with MPI, 277
with MPI, 277
OpenCL Compiler, 32
OpenCL Platform layer, 32
OpenCL preface, vii
OpenCL Runtime, 32
OpenMP, vii, 1, 3–9, 11–13, 44, 277

P

parallelism, 3, 7, 9, 10, 13, 22, 28, 30, 32, 70, 142, 155, 260, 272, 274, 277
PBO, 203, 205, 206, 208, 209, 216, 217, 220
phenotype, 222
Platform, 35
platform
OpenCL term, 34
population, 222
pragma, 167
printf, 155
process, 14, 16
automatically, 180, 181
CGM, 24
compilation, 107, 109, 115, 124, 259, 262, 282
context initialization, 39
CUDA SAXPY, 255
fork and join, 5
HTML, 181
installation, 246
matrix, 11
MPI, 1–3

OpenCL kernel, 123
OpenCL work-item, 65
parallel sum, 96, 97
picture, 195, 209, 218, 219
preprocessor, 118, 128, 236
queue, 112
solution, 4
transfer, 252
vector, 256
work offset, 70
Program, 32, 81
Program Object, 82

R

rank
MPI, 1–3, 11
OpenCL, 65
read_mem_fence, 95
relaxed consistency, 73
reqd_work_group_size, 191

S

SAXPY, 3, 19, 21, 23, 24, 122–124, 126, 128, 130–132, 136, 138, 249, 251, 252, 254, 256, 260, 262–266
definition, 2
MPI, 2
saxpy, 132
SDL_... constants:
...GL_DOUBLEBUFFER, 199
...OPENGL, 199
SDL_... functions:
...Delay, 219
...FreeSurface, 220, 221
...GL_GetProcAddress, 199
...GL_SetAttribute, 198, 199
...GL_SwapBuffers, 219
...Init, 198, 199
...PollEvent, 218, 219
...Quit, 220
...SetVideoMode, 199
selection, 222
size_t, 284
specimen, 222
sprofile, 180
synchronization, 3, 8, 30, 41, 60, 73–75, 79, 94, 95, 99, 121, 135, 137, 139, 156, 170, 173, 174, 179, 192, 214, 215, 231, 241, 284

T

Task Parallel Programming Model, 70
Tesla
2070, 253
C1060, 253
threadIdx, 256, 260

U

ULP, 164

V

vload3, 92
vloadn, 92
void, 90, 255
vstore3, 92
vstoren, 93

W

write_mem_fence, 95