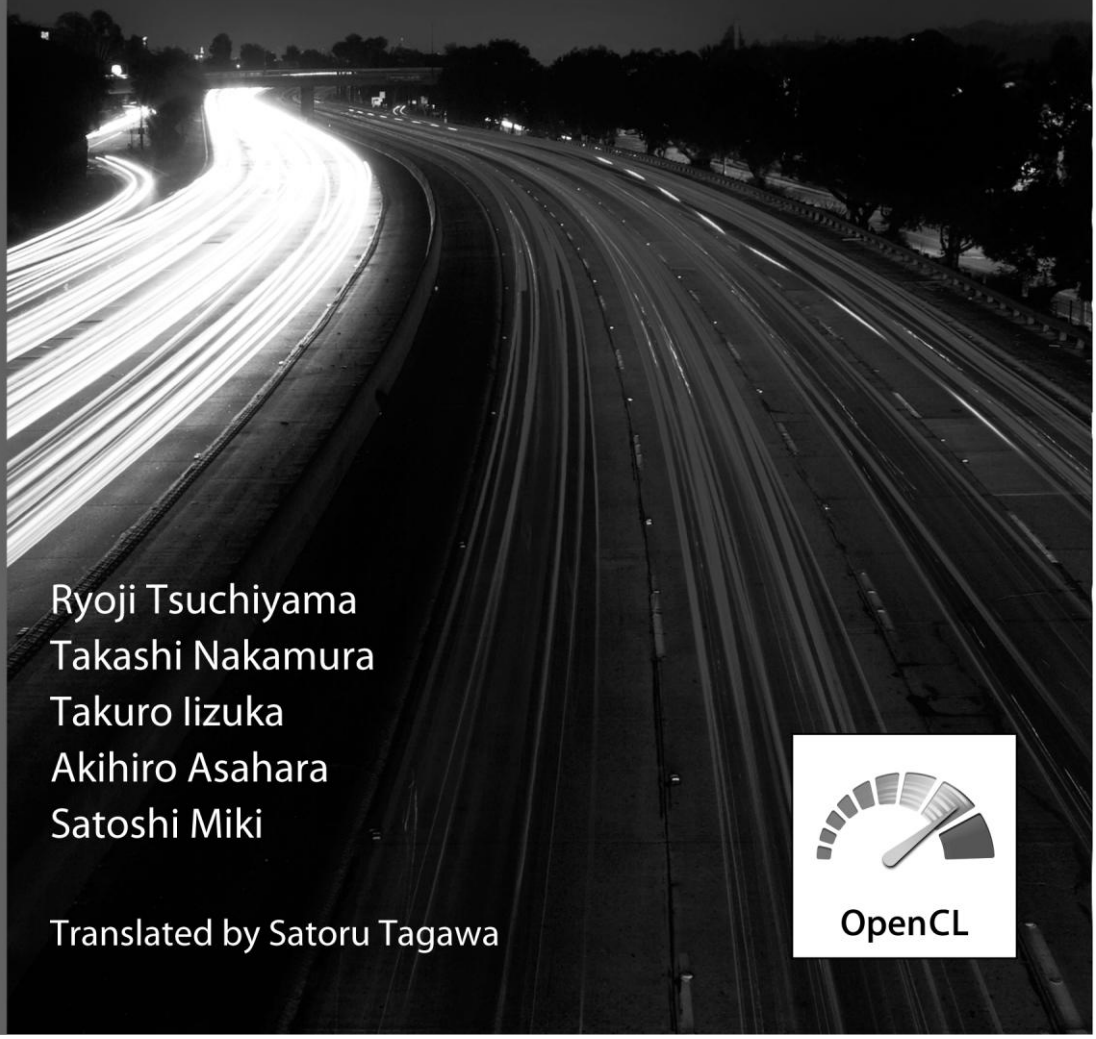
The OpenCL Programming Book

Parallel Programming for MultiCore CPU and GPU



Ryoji Tsuchiyama
Takashi Nakamura
Takuro Iizuka
Akihiro Asahara
Satoshi Miki

Translated by Satoru Tagawa

OpenCL

FIXSTARS®
Speed up your Business

Based on The Impress Japan Corporation publication "The OpenCL Programming Book (OpenCL Nyumon) " by Fixstars Corporation.

Contents

Foreword 4

Foreword 6

Acknowledgment 7

Abou the Authors 8

Introduction to Parallelization 10

Why Parallel 10

Parallel Computing (Hardware) 10

Parallel Computing (Software) 15

Conclusion 30

OpenCL 31

What is OpenCL? 31

Historical Background 31

An Overview of OpenCL 34

Why OpenCL? 36

Applicable Platforms 37

OpenCL Setup 41

Available OpenCL Environments 41

Developing Environment Setup 44

First OpenCL Program 51

Basic OpenCL 59

Basic Program Flow 59

Online/Offline Compilation 67

Calling the Kernel 77

Advanced OpenCL 98

OpenCL C 98

OpenCL Programming Practice 131

Case Study 164

FFT (Fast Fourier Transform) 164

Mersenne Twister 200

Notes 245

Foreword

“The free lunch is over.”

The history of computing has entered a new era. Until a few years ago, the CPU clock speed determined how fast a program will run. I vividly recall having a discussion on the topic of software performance optimization with a System Engineer back in 2000, to which his stance was, “the passing of time will take care of it all,” due to the improving technology. However, with the CPU clock speed leveling off at around 2005, it is now solely up to the programmer to make the software run faster. The free lunch is over. The processor venders have given up on increasing CPU clock speed, and are now taking the approach of raising the number of cores per processor in order to gain performance capability.

In recent years, many multi-core processors were born, and many developing methods were proposed. Programmers were forced to grudgingly learn a new language for every new type of processor architecture. Naturally, this caused a rise in demand for one language capable of handling any architecture types, and finally, an open standard was recently established. The standard is now known as “OpenCL”.

With the khronos group leading the way (known for their management of the OpenGL standard), numerous vendors are now working together to create a standard framework for the multi-core era.

Will this new specification become standardized? Will it truly be able to get the most out of multi-core systems? Will it help take some weight off the programmer’s shoulders? Will it allow for compatible programs to be written for various architectures? Whether the answers to these questions become “Yes” or “No” depends on the efforts of everyone involved. The framework must be designed to support various architectures. Many new tools and libraries must be developed. It must also be well-received by many programmers. This is no easy task, as evidenced by the fact that countless programming languages continually appear and disappear. However, one sure fact is that a new standard developing method is necessary for the new multi-core era. Another sure fact is that “OpenCL” is currently in the closest position to becoming that new standard.

The Fixstars Corporation, who authors this book, has been developing softwares for the Cell

Broadband Engine co-developed by Sony, Toshiba, and IBM, since 2004. We have been awed by the innovative idea of the Cell B.E. enough to battle its infamously difficult hardware design, in order to see the performance capability of the processor in action. We have also been amazed by the capability of the GPU, when its full power is unleashed by efficient usage of the hundreds of cores that exist within. Meanwhile, many of our clients had been expressing their distaste for the lack of standards between the different architectures, leading us to want an open framework. Therefore, "OpenCL" is a framework that developed solely out of need.

OpenCL is still in its infancy, and thus not every need can be fulfilled. We write this book in hopes that the readers of this book learns, uses, and contribute to the development of OpenCL, and thus become part of the ongoing evolution that is this multi-core era.

Fixstars Corporation
Chief Executive Officer
Satoshi Miki

Foreword

OpenCL is an exciting new open standard that is bringing the power of programming complex multi-processor systems to a much wider audience than ever before. Khronos is fully committed not only to creating the specification but also fostering a rich and diverse OpenCL ecosystem. A vital piece of any living standard is enabling the industry to truly understand and tap into the power full potential of the technology.

Fixstars is a skilled OpenCL practitioner and is ideally qualified to create state-of-the-art OpenCL educational materials. I wholeheartedly recommend to this book to anyone looking to understand and start using the amazing power of OpenCL.

Neil Trevett

President, The Khronos Group

Acknowledgment

The book is intended for those interested in the new framework known as OpenCL. While we do not assume any preexisting knowledge of parallel programming, since we introduce most of what you need to know in Chapter 1, we do however assume that the reader has a good grasp of the C language. Those who are already experts in parallel programming can start in Chapter 2 and dive straight into the new world of OpenCL.

The official reference manual for OpenCL can be found online on Khronous' website at: http://www.khronos.org/opencl/sdk/1.0/docs/man/xhtml/

Sample programs in this book can be downloaded at http://www.fixstars.com/en/books/opencl

About the Authors

Ryoji Tsuchiyama

Aside from being the leader of the OpenCL team at Fixstars, I am also involved in the development of a financial application for HPC. Since my first encounter with a computer (PC-8001) as a child, my fascination with computers have gotten me through a Master's Degree with an emphasis on distributed computing, and to a career where I orchestrate a bunch of CPUs and DSPs to work together in harmony with each other. Recently, I have been dealing with distributed computing on x86 clusters, as well as parallel processing using accelerators such as Cell/B.E. and GPU.

Takashi Nakamura

I am the main developer of the OpenCL Compiler at Fixstars. I get most of my vitamins and minerals from reading datasheets and manuals of chips from different vendors, and I particularly enjoy glancing at the pages on latency/throughput, as well as memory-mapped registers. I joined Fixstars in hopes that it will help me in achieving my life goal of creating the ultimate compiler as it necessitates an in-depth understanding of parallel processing, but I have yet to come up with a good solution. I'm becoming more convinced by day, however, that I/O and memory accesses must be conquered prior to tackling parallelization.

Takuro Iizuka

I am one of the developers of the OpenCL Compiler at Fixstars. I fell in love with the wickedly awesome capability of the GPU 2 years ago, and knew at first sight she was the one. My first computer was the Macintosh Classic, for which my only memory concerning it is playing the Invader game for hours on end. I often wonder how much geekier I could have been had I grew an appreciation for QuickDraw at that time instead. That being said, I still wish to write a book on how to make a GPU at some point.

Akihiro Asahara

At Fixstars, I take part in both the development and the marketing of middlewares and OS's for multi-core processors. Between teaching numerous seminars and writing technical articles, I am constantly trying to meet some sort of a deadline. My background, however, is in Astrophysics. Back in my researching days, I worked on projects such as construction of a data collection system (Linux kernel 2.2 base!) for large telescopes, and developing a software for astronomical gamma-ray simulation (in FORTRAN!). I'm constantly thinking of ways to

fit in some sort of astronomical work here at Fixstars.

Satoshi Miki

As one of the founding member, I currently serve the CEO position here at Fixstars. In 2004, I was convinced that software developing process will have to change drastically for the multi-core era, and decided to change the direction of the company to focus on the research and development of multi-core programs. Since then, I have been working hard to help spread multi-core technology that is already becoming the standard. My debut as a programmer was at the age of 25, and I am currently working on obtaining a PhD in information engineering.

About Fixstars Corporation

Fixstars Corporation is a software company that focuses on porting and optimization of programs for multi-core processors. Fixstars offers a total solution to improve the performance of multi-core based operations for fields that requires high computing power such as finance, medical, manufacturing, and digital media, using multi-core processors such as the Cell/B.E. and the GPU. Fixstars also works on spreading the multi-core technology through publication of articles introducing new technologies, as well as offer seminars. For more information, please visit http://www.fixstars.com/en.

Introduction to Parallelization

This chapter introduces the basic concepts of parallel programming from both the hardware and the software perspectives, which lead up to the introduction to OpenCL in the chapters to follow.

Why Parallel

In the good old days, software speedup was achieved by using a CPU with a higher clock speed, which significantly increased each passing year.
However, at around 2004 when Intel's CPU clock speed reached 4GHz, the increase in power consumption and heat dissipation formed what is now known as the "Power Wall", which effectively caused the CPU clock speed to level off.
The processor vendors were forced to give up their efforts to increase the clock speed, and instead adopt a new method of increasing the number of cores within the processor. Since the CPU clock speed has either remained the same or even slowed down in order to economize the power usage, old software designed to run on a single processor will not get any faster just by replacing the CPU with the newest model. To get the most out of the current processors, the software must be designed to take full advantage of the multiple cores and perform processes in parallel.
Today, dual-core CPUs are commonplace even for the basic consumer laptops. This shows that parallel processing is not just useful for performing advanced computations, but that it is becoming common in various applications.

Parallel Computing (Hardware)

First of all, what exactly is "parallel computing"? Wikipedia defines it as "a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently("in parallel").[1]
Many different hardware architectures exist today to perform a single task using multiple processors. Some examples, in order of decreasing scale is:

Grid computing - a combination of computer resources from multiple administrative domains applied to a common task.
MPP (Massively Parallel Processor) systems - known as the supercomputer architecture.
Cluster server system - a network of general-purpose computers.
SMP (Symmetric Multiprocessing) system - identical processors (in powers of 2) connected

together to act as one unit.
Multi-core processor - a single chip with numerous computing cores.

Flynn's Taxonomy

Flynn's Taxonomy is a classification of computer architectures proposed by Michael J. Flynn [2]. It is based on the concurrency of instruction and data streams available in the architecture. An instruction stream is the set of instructions that makes up a process, and a data stream is the set of data to be processed.

1. Single Instruction, Single Data stream (SISD)
SISD system is a sequential system where one instruction stream process one data stream. The pre-2004 PCs were this type of system.

2. Single Instruction, Multiple Data streams SIMD)
One instruction is broadcasted across many compute units, where each unit processes the same instruction on different data. The vector processor, a type of a supercomputer, is an example of this architecture type. Recently, various micro-processors include SIMD processors. For example, SSE instruction on Intel CPU, and SPE instruction on Cell Broadband Engines performs SIMD instructions.

3. Multiple Instruction, Single Data stream (MISD)
Multiple instruction streams process a single data stream. Very few systems fit within this category, with the exception for fault tolerant systems.

4. Multiple Instruction, Multiple Data streams (MIMD)
Multiple processing units each process multiple data streams using multiple instruction streams.

Using this classification scheme, most parallel computing hardware architectures, such as the SMP and cluster systems, fall within the MIMD category. For this reason, the MIMD architecture is further categorized by memory types.

The two main memory types used in parallel computing system are shared memory and distributed memory types. In shared memory type systems, each CPU that makes up the system is allowed access to the same memory space. In distributed memory type systems, each CPU that makes up the system uses a unique memory space.

Different memory types result in different data access methods. If each CPU is running a process, a system with shared memory type allows the two processes to communicate via Read/Write to the shared memory space. On the other hand, the system with distributed memory types requires data transfers to be explicitly performed by the user, since the two memory spaces are managed by two OS's.

The next sections explore the two parallel systems in detail.

Distributed Memory type

Tasks that take too long using one computer can be broken up to be performed in parallel using a network of processors. This is known as a cluster server system, which is perhaps the most commonly-seen distributed memory type system. This type of computing has been done for years in the HPC (High Performance Computing) field, which performs tasks such as large-scale simulation.

MPP (Massively Parallel Processor) system is also another commonly-seen distributed memory type system.
This connects numerous nodes, which are made up of CPU, memory, and a network port, via a specialized fast network. NEC's Earth Simulator and IBM's blue Gene are some of the known MPP systems.

The main difference between a cluster system and a MPP system lies in the fact that a cluster does not use specialized hardware, giving it a much higher cost performance than the MPP systems. Due to this reason, many MPP systems, which used to be the leading supercomputer type, have been replaced by cluster systems. According to the TOP500 Supercomputer Sites[3], of the top 500 supercomputers as of June 2009, 17.6% are MPP systems, while 82% are cluster systems.

One problem with cluster systems is the slow data transfer rates between the processors. This is due to the fact that these transfers occur via an external network. Some recent external networks include Myrinet, Infiniband, 10Gbit Ethernet, which has significantly become faster compared to the traditional Gigabit Ethernet. Even with these external networks, the transfer rates are still at least an order of magnitude slower than the local memory access by each processor.

For the reason given above, cluster systems are suited for parallel algorithms where the CPU's do

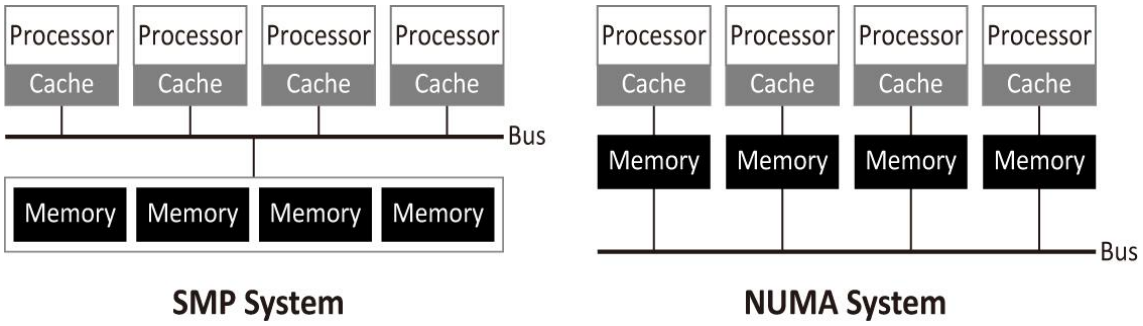
not have to communicate with each other too often. These types of algorithms are said to be "Course-grained Parallel." These algorithms are used often in simulations where many trials are required, but these trials have no dependency. An example is the risk simulation program used in Derivative product development in the finance field.

Shared Memory Type

In shared memory type systems, all processors share the same address space, allowing these processors to communicate with each other through Read/Writes to shared memory. Since data transfers/collections are unnecessary, this results in a much simpler system from the software perspective.

An example of a shared memory type system is the Symmetric Multiprocessing (SMP) system (Figure 1.1, left). The Intel Multiprocessor Specification Version 1.0 released back in 1994 describes the method for using x86 processors in a multi-processor configuration, and 2-Way work stations (workstations where up to 2 CPU's can be installed) are commonly seen today [4]. However, increasing the number of processors naturally increases the number of accesses to the memory, which makes the bandwidth between the processors and the shared memory a bottleneck. SMP systems are thus not scalable, and only effective up to a certain number of processors. Although 2-way servers are inexpensive and common, 32-Way or 64-Way SMP servers require specialized hardware, which can become expensive.

Figure 1.1: SMP and NUMA



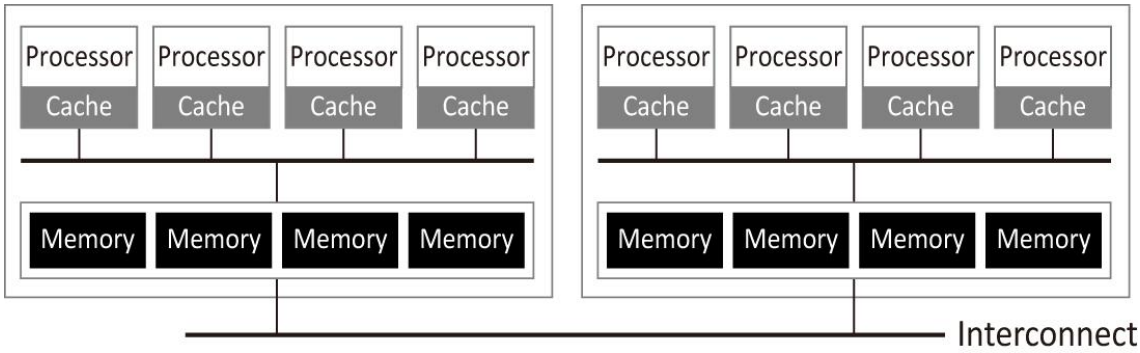
Another example of a shared memory type system is the Non-Uniform Memory Access (NUMA) system. The main difference from a SMP system is that the physical distance between the processor and the memory changes the access speeds. By prioritizing the usage of physically closer memory (local memory) rather than for more distant memory (remote memory), the bottleneck in SMP systems can be minimized. To reduce the access cost to the remote memory, a

processor cache and a specialized hardware to make sure the cache data is coherent has been added, and this system is known as a Cash Coherent NUMA (cc-NUMA).

Server CPUs such as AMD Opteron and Intel Xeon 5500 Series contains a memory controller within the chip. Thus, when these are used in multi-processor configuration, it becomes a NUMA system. The hardware to verify cache coherency is embedded into the CPU. In addition, Front Side Bus (FSB), which is a bus that connects multiple CPU as well as other chipsets, the NUMA gets rid of this to use a interconnect port that uses a Point-to-Point Protocol. These ports are called Quick Path Interconnect (QPI) by Intel, and Hyper Transport by AMD.

Now that the basic concepts of SMP and NUMA are covered, looking at typical x86 server products bring about an interesting fact. The Dual Core and Quad Core processors are SMP systems, since the processor cores all access the same memory space. Networking these multi-core processors actually end up creating a NUMA system. In other words, the mainstream 2+way x86 server products are "NUMA systems made by connecting SMP systems" (Figure 1.2) [5].

Figure 1.2: Typical 2Way x86 Server



Accelerator

The parallel processing systems discussed in the previous sections are all made by connecting generic CPUs. Although this is an intuitive solution, another approach is to use a different hardware more suited for performing certain tasks as a co-processor. The non-CPU hardware in this configuration is known as an Accelerator.

Some popular accelerators include Cell Broadband Engine (Cell/B.E.) and GPUs. Accelerators typically contain cores optimized for performing floating point arithmetic (fixed point arithmetic for some DSPs). Since these cores are relatively simple and thus do not take much area on the

chip, numerous cores are typically placed.

For example, Cell/B.E. contains 1 PowerPC Processor Element (PPE) which is suited for processes requiring frequent thread switching, and 8 Synergistic Processor Elements (SPE) which are cores optimized for floating point arithmetic. These 9 cores are connected using a high-speed bus called the Element Interconnect Bus (EIB), and placed on a single chip.

Another example is NVIDIA's GPU chip known as Tesla T10, which contains 30 sets of 8-cored Streaming Processors (SM), for a total of 240 cores on one chip.

In recent years, these accelerators are attracting a lot of attention. This is mainly due to the fact that the generic CPU's floating point arithmetic capability has leveled off at around 10 GFLOPS, while Cell/B.E. and GPUs can perform between 100 GFLOPS and 1 TFLOPS for a relatively inexpensive price. It is also more "Green", which makes it a better option than cluster server systems, since many factories and research labs are trying to cut back on the power usage.

For example, the circuit board and semiconductor fields use automatic visual checks. The number of checks gets more complex every year, requiring faster image processing so that the rate of production is not compromised. The medical imaging devices such as ultrasonic diagnosing devices and CT Scans are taking in higher and higher quality 2D images as an input every year, and the generic CPUs are not capable of processing the images in a practical amount of time. Using a cluster server for these tasks requires a vast amount of space, as well as high power usage. Thus, the accelerators provide a portable and energy-efficient alternative to the cluster. These accelerators are typically used in conjunction with generic CPUs, creating what's known as a "Hybrid System".

In summary, an accelerator allows for a low-cost, low-powered, high-performance system. However, the transfer speed between the host CPU and the accelerator can become a bottle neck, making it unfit for applications requiring frequent I/O operations. Thus, a decision to use a hybrid system, as well as what type of a hybrid system, needs to be made wisely.

OpenCL, in brief, is a development framework to write applications that runs on these "hybrid systems".

Parallel Computing (Software)

Up until this point, hardware architectures that involve performing one task using numerous processors had been the main focus. This section will focus on the method for parallel programming for the discussed hardware architectures.

Sequential vs Parallel Processing

Take a look at the pseudocode below.

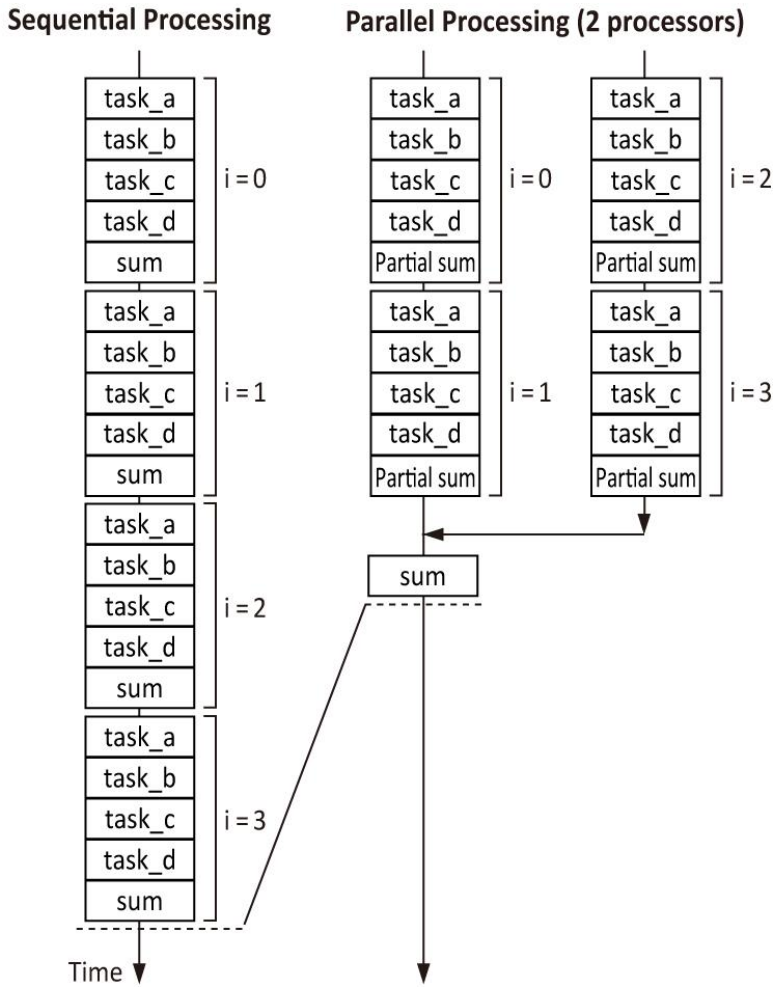
List1.1: Pseudocode for parallel processing

```
001: for(i=0;i<N;i++){
002:       resultA   = task_a(i);
003:       resultB   = task_b(i);
004:       resultC   = task_c(i);
005:       resultD   = task_d(i);
006:       resultAll  += resultA + resultB + resultC + resultD;
007: }
```

Executing the above code on a single-core, single-CPU processor (SISD system), the 4 tasks would be run sequentially in the order task_a, task_b, task_c, task_d, and the returned values are then summed up. This is then repeated N times, incrementing i after each iteration. This type of method is called Sequential processing. Running the code with N=4 is shown on the left hand side of Figure 1.3.

This type of code can benefit from parallelization. If this code is compiled without adding any options for optimization on a Dual Core system, the program would run sequentially on one core. In this scenario, the other core has nothing to do, so it becomes idle. This is clearly inefficient, so the intuitive thing to do here is to split the task into 2 subtasks, and run each subtasks on each core. This is the basis of parallel programming. As shown on Figure 1.3, the task is split up into 2 such that each subtask runs the loop N/2 times, and then performs addition at the end.

Figure 1.3: Parallel processing example



For actual implementation of parallel programs, it is necessary to follow the following steps.

1. Analyze the dependency within the data structures or within the processes, etc, in order to decide which sections can be executed in parallel.
2. Decide on the best algorithm to execute the code over multiple processors
3. Rewrite code using frameworks such as Message Passing Interface (MPI), OpenMP, or OpenCL.

In the past, these skills were required only by a handful of engineers, but since multi-core processors are becoming more common, the use of these distributed processing techniques are becoming more necessary. The following sections will introduce basic concepts required to implement parallel processes.

Where to Parallelize?

During the planning phase of parallel programs, certain existing laws must be taken into account. The first law states that if a program spends y% of the time running code that cannot be executed in parallel, the expected speed up from parallelizing is at best a 1/y fold improvement. The law can be proven as follows. Assume Ts to represent the time required to run the portion of the code that cannot be parallelized, and Tp to represent the time required to run the portion of the code that can benefit from parallelization. Running the program with 1 processor, the processing time is:

```
T(1) = Ts + Tp
```

The processing time when using N processors is:

```
T(N) = Ts + Tp/N
```

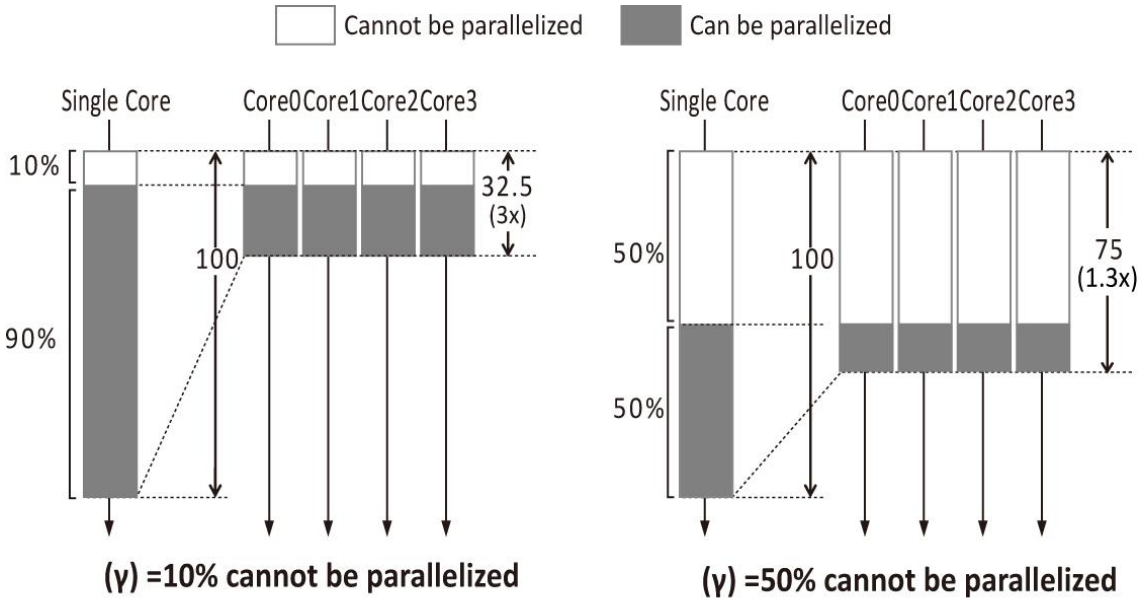
Using y to represent the proportion of the time spent running code that cannot be parallelized, the speedup S achieved is:

```
S = T(1)/T(N) = T(1)/(Ts + Tp/N) = 1/(y+(1-y)/N)
```

Taking the limit as N goes to infinity, the most speedup that can be achieved is S=1/y. This law is known as Amdahl's Law [6].

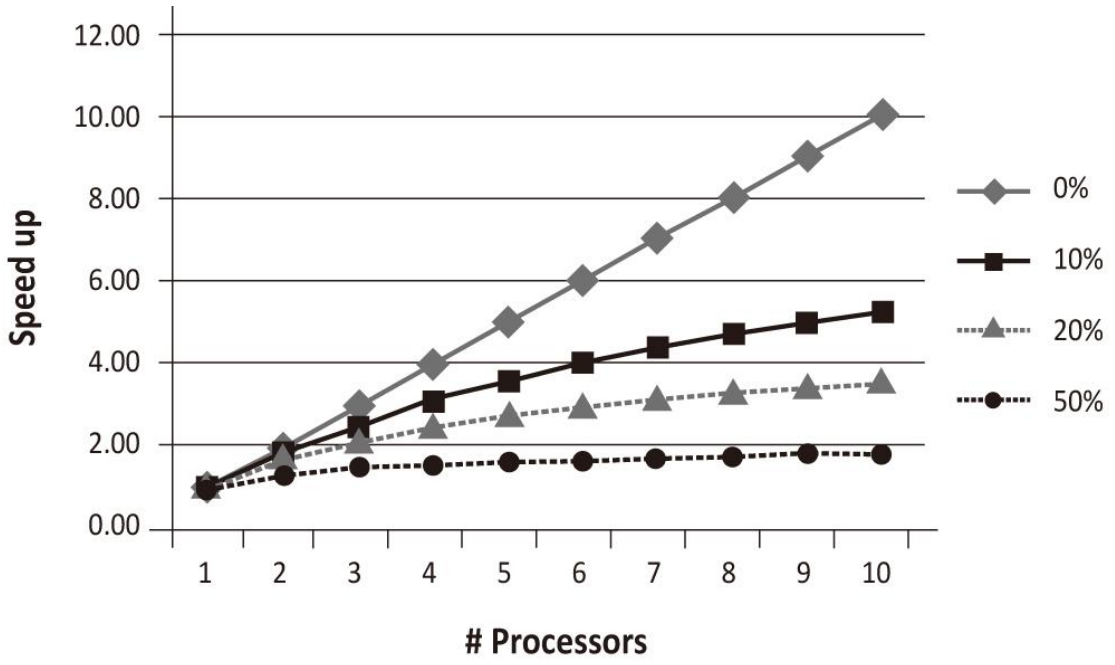
For a more concrete example, assume that a sequential code is being rewritten to run on a Quad Core CPU. Ideally, a 4-time speedup is achieved. However, as the law states, the speedup is limited by the portion of the code that must be run sequentially. Figure 1.4 shows the 2 cases where the proportion of the code that cannot be run in parallel (y) is 10%, and 50%. Even without taking overhead into account, the figure shows a difference of 3x and 1.3x speedup depending on y.

Figure 1.4: Amdahl's Law Example



This problem becomes striking as the number of processors is increased. For a common 2-way server that uses Intel's Xeon 5500 Series CPUs which supports hyper-threading, the OS sees 16 cores. GPUs such as NVIDIA's Tesla can have more 200 cores. Figure 1.5 shows the speedup achieved as a function of the sequentially processed percentage y and the number of cores. The graph clearly shows the importance of reducing the amount of sequentially processed portions, especially as the number of cores is increased. This also implies that the effort used for parallelizing a portion of the code that does not take up a good portion of the whole process may be in vain. In summary, it is more important to reduce serially processed portions rather than parallelizing a small chunk of the code.

Figure 1.5: Result of scaling for different percentage of tasks that cannot be parallelized



While Amdahl's law gives a rather pessimistic impression of parallel processing, the Gustafson Law provides a more optimistic view. This law states that as the program size increases, the fraction of the program that can be run in parallel also increases, thereby decreasing the portion that must be performed sequentially. Recall the previously stated equation:
T(N) = Ts + Tp/N

Gustafson states that of the changes in Ts, Tp is directly proportional to the program size, and that Tp grows faster than Ts. For example, assume a program where a portion that must be run sequentially is limited to initialization and closing processes, and all other processes can be performed in parallel. By increasing the amount of data to process by the program, it is apparent that Gustafson's law holds true. In other words, Gustafson's law shows that in order to efficiently execute code over multiple processors, a large-scale processing must take place. Development of parallel programming requires close attention to these 2 laws.

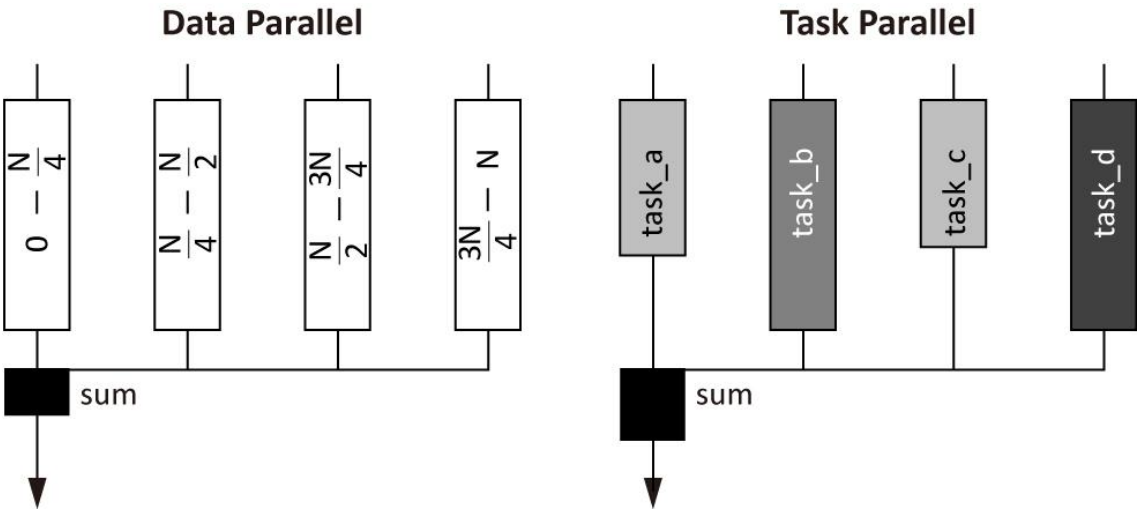
Types of Parallelism

After scrutinizing the algorithm and deciding where to parallelize, the next step is to decide on the type of parallelism to use.

Parallel processing requires the splitting up of the data to be handled, and/or the process itself. Refer back to the code on List 1.1. This time, assume the usage of a Quad-core CPU to run 4

processes at once. An intuitive approach is to let each processor perform N/4 iteration of the loop, but since there are 4 tasks within the loop, it also makes just as much sense to run each of these tasks in each processor. The former method is called "Data Parallel", and the latter method is called "Task Parallel" (Figure 1.6).

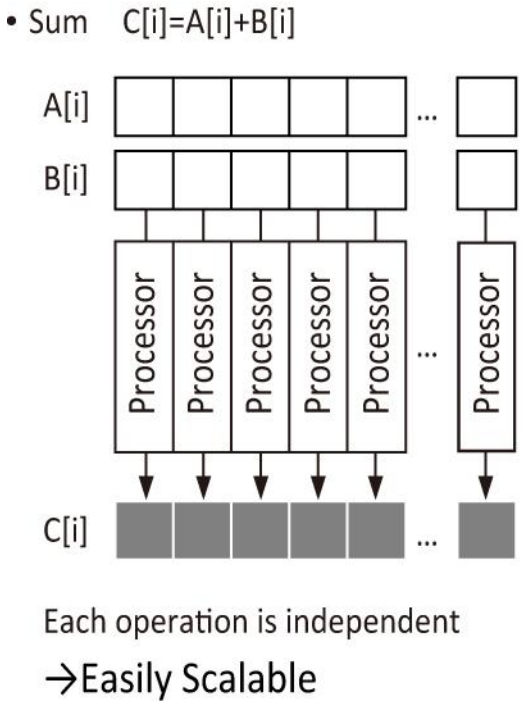
Figure 1.6: Data Parallel vs Task Parallel



Data Parallel

Main characteristics of data parallel method is that the programming is relatively simple since multiple processors are all running the same program, and that all processors finish their task at around the same time. This method is efficient when the dependency between the data being processed by each processor is minimal. For example, vector addition can benefit greatly from this method. As illustrated in Figure 1.7, the addition at each index can be performed completely independently of each other. For this operation, the number of processors is directly proportional to the speedup that may be achieved if overhead from parallelization can be ignored. Another, more concrete example where this method can be applied is in image processing. The pixels can be split up into blocks, and each of these blocks can be filtered in parallel by each processor.

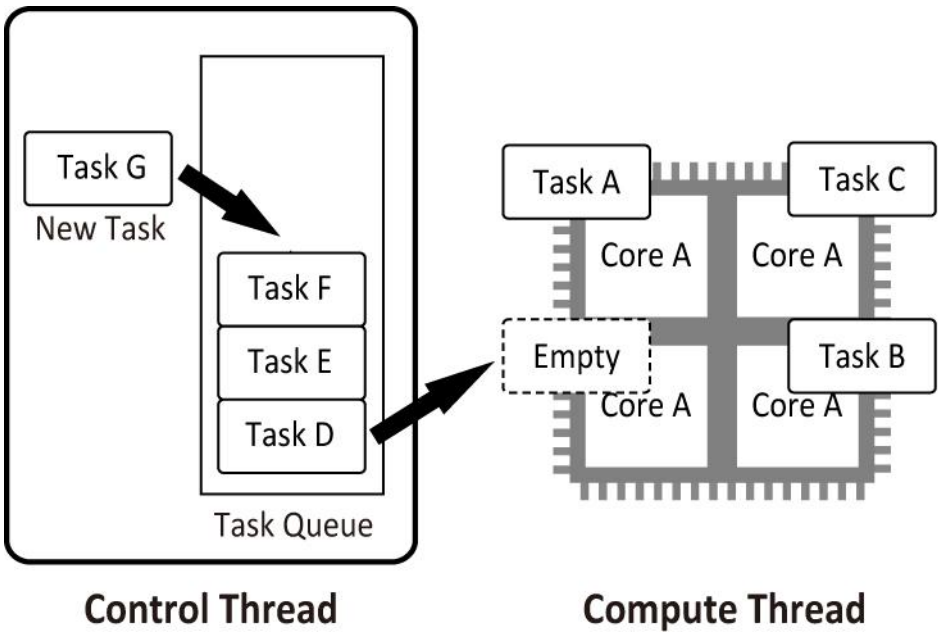
Figure 1.7: Vector Addition



Task Parallel

The main characteristic of the task parallel method is that each processor executes different commands. This increases the programming difficulty when compared to the data parallel method. Since the processing time may vary depending on how the task is split up, it is actually not suited for the example shown in Figure 1.6. Also, since task_a and task_c are doing nothing until task_b and task_d finishes, the processor utilization is decreased. Task parallel method requires a way of balancing the tasks to take full advantage of all the cores. One way is to implement a load balancing function on one of the processors. In the example in Figure 1.8, the load balancer maintains a task queue, and assigns a task to a processor that has finished its previous task.

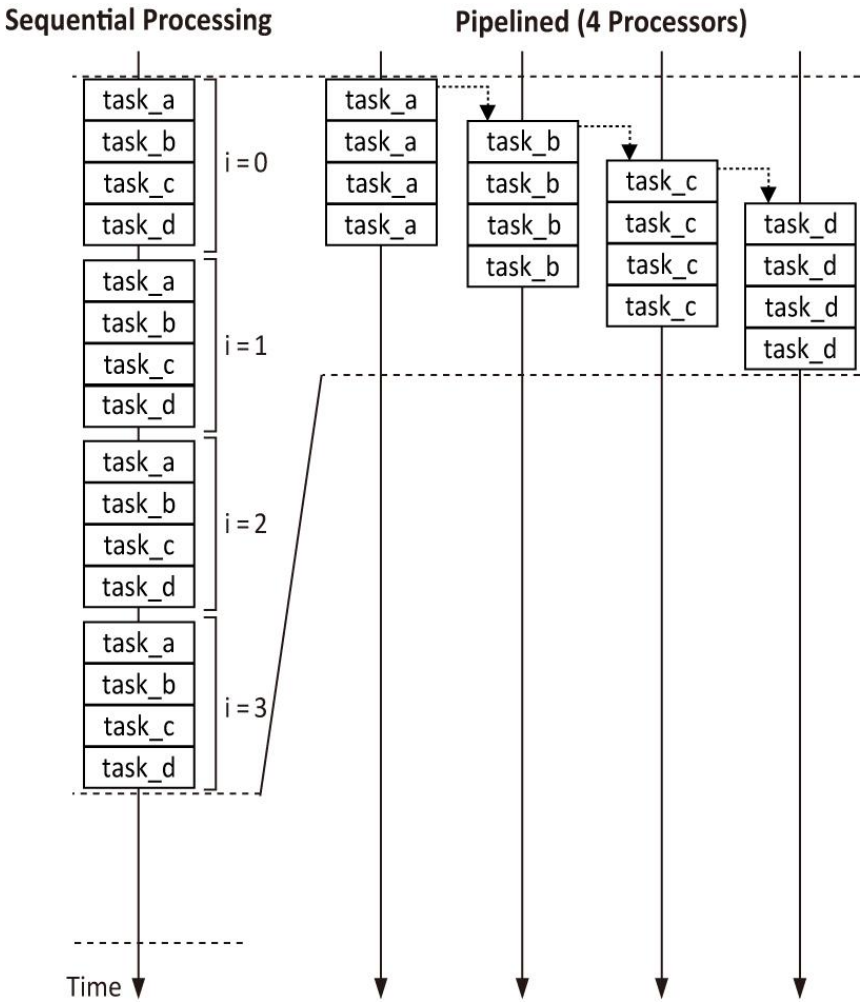
Figure 1.8: Load Balancing



Another method for task parallelism is known as pipelining. Pipelining is usually in reference to the "instruction pipeline" where multiple instructions, such as instruction decoding, arithmetic operation, and register fetch, are executed in an overlapped fashion over multiple stages. This concept can be used in parallel programming as well.

Figure 1.9 shows a case where each processor is given its own task type that it specializes in. In this example, the start of each task set is shifted in the time domain, such that task_b, task_c, task_d takes in the output of task_a, task_b, task_c as an input. The data moves as a stream across the processors. This method is not suited for the case where only one set of tasks is performed, but can be effective when processing, for example, videos, where processing frames are taken as inputs one after another.

Figure 1.9: Pipelining



Hardware Dependencies

When porting sequential code to parallel code, the hardware must be decided wisely. Programs usually have sections suited for data parallelism as well as for task parallelism, and the hardware is usually suited for one or the other.

For example, the GPU is suited for data parallel algorithms, due to the existence of many cores. However, since the GPU (at present) is not capable of performing different tasks in parallel, the Cell/B.E. is more suited for performing task parallel algorithms, since its 8 cores are capable of working independently of each other.

OpenCL allows the same code to be executed on either platforms, but since it cannot change the nature of the hardware, the hardware and the parallelization method must be chosen wisely.

Implementing a Parallel Program

After deciding on the parallelization method, the next step is the implementation. In decreasing order of user involvement:

1. Write parallel code using the operating system's functions.
2. Use a parallelization framework for program porting.
3. Use an automatic-parallelization compiler.

This section will explore the different methods.

Parallelism using the OS System Calls

Implementing parallel programs using the OS system call requires, at minimum, a call to execute and close a program, and some way of transferring data between the executed programs. If this is done on a cluster system, the data transfer between programs is performed using network transfer APIs such as the socket system call, but this is commonly done using a framework instead.

For performing parallel instructions performed within the processor itself, however, the OS system call may be used instead of the framework. The code can be further broken down into "parallel processes" and "parallel threads" to be run on the processor. The difference between processes and threads are as follows.

A process is an executing program given its own address space by the operating system. In general, the operating system performs execution, closing, and interruption within these process units, making sure each of these processes' distinct resources do not interfere with each other. Data transfer between programs may be performed by a system call to the operating system. For example, UNIX provides a system call shmget() that allocates shared memory that can be accessed by different processes.

A thread is a subset of a process that is executed multiple times within the program. These threads share the same address space as the processes. In general, since these threads execute in the same memory space, the overhead from starting and switching is much smaller than when compared to processes. In general, the operating system provides an API to create and manage these threads. For example, UNIX provides a library called Pthreads, which is a POSIX-approved thread protocol. POSIX is a standard for API specified by IEEE.

Whether to use parallel processes or parallel threads within a processor depends on the case, but in general, parallel threads are used if the goal is speed optimization, due to the light overhead.

List 1.2 shows an example where each member of an array is incremented using multithreading.

List 1.2: pthread example

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include <pthread.h>
004:
005: #define TH_NUM 4
006: #define N 100
007:
008: static void *thread_increment(void *array)
009: {
010: int i;
011: int *iptr;
012:
013: iptr = (int *)array;
014: for(i=0;i< N / TH_NUM;i++) iptr[i] += 1;
015:
016: return NULL;
017: }
018:
019: int main(void)
020: {
021: int i;
022: pthread_t thread[TH_NUM];
023:
024: int array[N];
025:
026: /* initialize array*/
027: for(i=0;i<N;i++){
028: array[i] = i;
029: }
```

```
030:
031: /* Start parallel process */
032: for(i=0;i<TH_NUM;i++){
033: if (pthread_create(&thread[i], NULL, thread_increment, array + i * N / TH_NUM) != 0)
034: {
035: return 1;
036: }
037: }
038:
039: /* Synchronize threads*/
040: for(i=0;i<TH_NUM;i++){
041: if (pthread_join(thread[i], NULL) != 0)
042: {
043: return 1;
044: }
045: }
046:
047: return 0;
048: }
```

The code is explained below.

003: Include file required to use the pthread API
008-017: The code ran by each thread. It increments each array elements by one. The start index is passed in as an argument.
022: Declaration of pthread_t variable for each thread. This is used in line 033.
032-037: Creation and execution of threads. In line 033, the third argument is the name of the function to be executed by the thread, and the fourth argument is the argument passed to the thread.
039-045: Waits until all threads finish executing

Parallelism using a Framework

Many frameworks exist to aid in parallelization, but the ones used in practical applications, such as in research labs and retail products, are limited. The most widely used frameworks are Message Passing Interface (MPI) for cluster servers, OpenMP for shared memory systems (SMP, NUMA), and the parallelization APIs in Boost C++. These frameworks require the user to

specify the section as well as the method used for parallelization, but take care of tasks such as data transfer and execution of the threads, allowing the user to focus on the main core of the program.

List 1.3 shows an example usage of OpenMP, which is supported by most mainstream compilers, such as GCC, Intel C, Microsoft C. From the original sequential code, the programmer inserts a "#pragma" directive, which explicitly tells the compiler which sections to run in parallel. The compiler then takes care of the thread creation and the commands for thread execution.

List 1.3: OpenMP Example

```
001: #include<stdio.h>
002: #include<stdlib.h>
003: #include<omp.h>
004: #define N 100
005: #define TH_NUM 4
006:
007: int main ()
008: {
009: int i;
010: int rootBuf[N];
011:
012: omp_set_num_threads(TH_NUM);
013:
014: /* Initialize array*/
015: for(i=0;i<N;i++){
016: rootBuf[i] = i;
017: }
018:
019: /* Parallel process */
020: #pragma omp parallel for
021: for (i = 0; i < N; i++) {
022: rootBuf[i] = rootBuf[i] + 1;
023: }
024:
025: return(0);
026: }
```

When compiling the above code, the OpenMP options must be specified. GCC (Linux) requires "-fopenmp", intel C (Linux) requires "-openmp", and Microsoft Visual C++ requires /OpenMP. The code is explained below.

003: Include file required to use OpenMP
004: Size of the array, and the number of times to run the loop. In general, this number should be somewhat large to benefit from parallelism.
012: Specify the number of threads to be used. The argument must be an integer.
020: Breaks up the for loops that follows this directive, into the number of threads specified in 012.

This example shows how much simpler the programming becomes if we use the OpenMP framework. Compare with List 2 that uses pthreads.

Automatic parallelization compiler

Compilers exist that examines for-loops to automatically decide sections that can be run in parallel, as well as how many threads to use.

For example, Intel C/C++ compiler does this when the option is sent.

(On Linux)

```
> icc -parallel -par-report3 -par-threshold0 -03 o parallel_test parallel_test.c
```

(Windows)

```
> Icc /Qparallel /Qpar-report3 /Q-par-threshold0 -03 o parallel_test parallel_test.c
```

The explanations for the options are discussed below.

• -parallel: Enables automatic parallelization
• -par-report3: Reports which section of the code was parallelized. There are 3 report levels, which can be specified in the form -par-report[n]
• -par-threshold0: Sets the threshold to decide whether some loops are parallelized. In order to benefit from parallelization, enough computations must be performed within each loop to hide the overhead from process/thread creation. This is specified in the form -par-threshold[n]. The

value for n must be between 0 ~ 100, with higher number implying higher number of computations. When this value is 0, all sections that can be parallelized become parallelized. The default value is 75.

At a glance, the automatic parallelization compiler seems to be the best solution since it does not require the user to do anything, but in reality, as the code becomes more complex, the compiler has difficulty finding what can be parallelized, making the performance suffer. As of August 2009, no existing compiler (at least no commercial) can auto-generate parallel code for hybrid systems such as the accelerator.

Conclusion

This section discussed the basics of parallel processing from both the hardware and the software perspectives. The content here is not limited to OpenCL, so those interested in parallel processing in general should have some familiarity with the discussed content. Next chapter will introduce the basic concepts of OpenCL.

OpenCL

The previous chapter discussed the basics of parallel processing. This chapter will introduce the main concepts of OpenCL.

What is OpenCL?

To put it simply, OpenCL (Open Computing Language) is "a framework suited for parallel programming of heterogeneous systems". The framework includes the OpenCL C language as well as the compiler and the runtime environment required to run the code written in OpenCL C.

In recent years, it is no longer uncommon for laptops to be equipped with relatively high-end GPUs. For desktops, it is possible to have multiple GPUs due to PCIe slots becoming more common. OpenCL provides an effective way to program these CPU + GPU type heterogeneous systems. OpenCL, however, is not limited to heterogeneous systems, and it can be used on homogeneous, multi-core processors such as the Intel Core i7, by using one core for control and the other cores for computations.

OpenCL is standardized by the Khronos Group, who is known for their management of the OpenGL specification. The group consists of members from companies like AMD, Apple, IBM, Intel, NVIDIA, Texas Instruments, Sony, Toshiba, which are all well-known processor vendors and/or multi-core software vendors. The goal of the standardization is ultimately to be able to program any combination of processors, such as CPU, GPU, Cell/B.E., DSP, etc. using one language. The company authoring this book, the Fixstars Corporation, is also a part of the standardization group, and involved in the specification process.

This chapter will give a glimpse into the history of how OpenCL came about, as well as an overview of what OpenCL is.

Historical Background

Multi-core + Heterogeneous Systems

In recent years, more and more sequential solutions are being replaced with multi-core solutions. The physical limitation has caused the processor capability to level off, so the effort is naturally being placed to use multiple processors in parallel. The heterogeneous system such as the

CPU+GPU system is becoming more common as well.

The GPU is a processor designed for graphics processing, but its parallel architecture containing up to hundreds of cores makes it suited for data parallel processes. The GPGPU (General Purpose GPU), which uses the GPU for tasks other than graphics processing, has been attracting attention as a result [7]. NVIDIA's CUDA has made the programming much simpler, as it allows the CPU for performing general tasks and the GPU for data parallel tasks [8]. NVIDIA makes GPUs for graphics processing, as well as GPUs specialized for GPGPU called "Tesla".

Another example of accelerator is Cell/B.E., known for its use on the PLAYSTATION3. Since this is suited for task parallel processing, the hardware can be chosen as follows:

- Data Parallel → GPU
- Task Parallel → Cell/B.E.
- General Purpose → CPU

Vendor-dependent Development Environment

We will now take a look at software development in a heterogeneous environment.
First, we will take a look at CUDA, which is a way to write generic code to be run on the GPU. Since no OS is running on the GPU, the CPU must perform tasks such as code execution, file system management, and the user interface, so that the data parallel computations can be performed on the GPU.
In CUDA, the control management side (CPU) is called the "host", and the data parallel side (GPU) is called the "device". The CPU side program is called the host program, and the GPU side program is called the kernel. The main difference between CUDA and the normal development process is that the kernel must be written in the CUDA language, which is an extension of the C language. An example kernel code is shown in List 2.1.

List 2.1: Example Kernel Code

```
001: __global__ void vecAdd(float *A, float *B, float *C) /* Code to be executed on the GPU
(kernel) */
002: {
003: int tid = threadIdx.x; /* Get thread ID */
004:
005: C[tid] = A[tid] + B[tid];
006:
```

```
007: return;
008: }
```

The kernel is called from the CPU-side, as shown in List 2.2 [9].

List 2.2: Calling the kernel in CUDA

```
001: int main(void) /* Code to be ran on the CPU */
002: {
003: ...
004: vecAdd<<<1, 256>>>(dA, dB, dC); /* kernel call (256 threads) */
005: ...
006: }
```

The Cell/B.E. is programmed using the "Cell SDK" released by IBM [10]. The Cell/B.E. comprises of a PPE for control, and 8 SPEs for computations, each requiring specific compilers. Although it does not extend a language like the CUDA, the SPE must be controlled by the PPE, using specific library functions provided by the Cell SDK. Similarly, specific library functions are required in order to run, for example, SIMD instructions on the SPE.
You may have noticed that the structures of the two heterogeneous systems are same, with the control being done on the CPU/PPE, and the computation being performed on GPU/SPE. However, the programmers must learn two distinct sets of APIs. (Table 2.1)

Table 2.1: Example Cell/B.E. API commands

Function	API
Open SPE program	spe_image_open()
Create SPE context	spe_context_create()
Load SPE program	spe_context_load()
Execute SPE program	spe_context_run()
Delete SPE context	spe_context_destroy()
Close SPE program	spe_image_close()

In order to run the same instruction on different processors, the programmer is required to learn processor-specific instructions. As an example, in order to perform a SIMD instruction, x86, PPE, SPE must call a SSE instruction, a VMX instruction, and a SPE-embedded instruction, respectively. (Table 2.2)

Table 2.2: SIMD ADD instructions on different processors

X86 (SSE3)	PowerPC (PPE)	SPE
_mm_add_ps()	vec_add()	spu_add()

Additionally, in the embedded electronics field, a similar model is used, where the CPU manages the DSPs suited for signal processing. Even here, the same sort of procedure is performed, using a completely different set of APIs.

To summarize, all of these combinations have the following characteristics in common:
• Perform vector-ized operation using the SIMD hardware
• Use multiple compute processors in conjunction with a processor for control
• The systems can multi-task

However, each combination requires its own unique method for software development.
This can prove to be inconvenient, since software development and related services must be rebuilt ground up every time a new platform hits the market. The software developers must learn a new set of APIs and languages. Often times, the acquired skills might quickly become outdated, and thus prove to be useless. This is especially more common in heterogeneous platforms, as programming methods can be quite distinct from each other. Also, there may be varying difficulty in the software development process, which may prevent the platform to be chosen solely on its hardware capabilities.
The answer to this problem is "OpenCL".

An Overview of OpenCL

OpenCL is a framework that allows a standardized coding method independent of processor types or vendors. In particular, the following two specifications are standardized:

1. OpenCL C Language Specification
Extended version of C to allow parallel programming

2. OpenCL Runtime API Specification
API used by the control node to send tasks to the compute cores

By using the OpenCL framework, software developers will be able to write parallel code that is independent of the hardware platform.

The heterogeneous model, made up of a control processor and multiple compute processors, is

currently being employed in almost all areas, including HPC, Desktops and embedded systems. OpenCL will allow all of these areas to be taken care of.
As an example, if an OpenCL implementation includes support for both multi-core CPU and GPU, a CPU core can use the other cores that include SSE units, as well as all the GPU to perform computations in parallel [11]. Furthermore, everything can be written using OpenCL.

OpenCL Software Framework

So what exactly is meant by "using OpenCL"? When developing software using OpenCL, the following two tools are required:
• OpenCL Compiler
• OpenCL Runtime Library

The programmer writes the source code that can be executed on the device using the OpenCL C language. In order to actually execute the code, it must first be compiled to a binary using the OpenCL Compiler designed for that particular environment.
Once the code is compiled, it must then be executed on the device. This execution, which includes the loading of the binary and memory allocation, can be managed by the controlling processor. The execution process is common to all heterogeneous combinations. This process must be coded by the programmer [12].

The control processer is assumed to have a C/C++ compiler installed, so the execution process can be written in normal C/C++. The set of commands that can be used for this task is what is contained in the "OpenCL Runtime Library," which is designed to be used for that particular environment [13].
The OpenCL Runtime Library Specification also requires a function to compile the code to be run on the device. Therefore, all tasks ranging from compiling to executing can be written within the control code, requiring only this code to be compiled and run manually.

Performance

OpenCL is a framework that allows for common programming in any heterogeneous environment. Therefore, OpenCL code is capable of being executed on any computer. This brings up the question of whether performance is compromised, as it is often the case for this type of common language and middleware. If the performance suffers significantly as a result of using OpenCL, its purpose becomes questionable.

OpenCL provides standardized APIs that perform tasks such as vectorized SIMD operations, data parallel processing, task parallel processing, and memory transfers. However, the abstraction layer is relatively close to the hardware, minimizing the overhead from the usage of OpenCL. In case lower-level programming is desired, OpenCL does allow the kernel to be written in its native language using its own API. This seems to somewhat destroy the purpose of OpenCL, as it would make the code non-portable. However, OpenCL prioritizes maximizing performance of a device rather than its portability. One of the functions that the OpenCL runtime library must support is the ability to determine the device type, which can be used to select the device to be used in order to run the OpenCL binary.

While the kernel written in OpenCL is portable, as it should be able to run on any device, the hardware selection must be kept in mind. If a data parallel algorithm is written, it should be executed on a device suited for data parallel processes. The performance can vary greatly just by the existence of SIMD units on the device. Thus, coding in OpenCL does not imply that it will run fast on any device. The hardware should be chosen wisely to maximize performance.

Why OpenCL?

This section will discuss the advantage of developing software using OpenCL.

Standardized Parallelization API

As mentioned previously, each heterogeneous system requires its own unique API/programming method. The aforementioned CUDA and Cell SDK are such examples. As far as standardized tools, Unix POSIX threads, OpenMP for shared memory systems, and MPI for distributed memory type systems exist. OpenCL is another standardized tool, but contrary to the other tools, it is independent of processors, operating systems, and memory types. By learning OpenCL, software developers will be able to code in any parallel environment. It is, however, necessary for these processors to support OpenCL, requiring the OpenCL compiler and the runtime library to be implemented. That being said, there are many reputable groups working on the standardization of OpenCL, making the switch over likely to be just a matter of time.

Optimization

OpenCL provides standard APIs at a level close to the hardware. To be specific, these include SIMD vector operations, asynchronous memory copy using DMA, and memory transfer between

the processors. Since high abstraction is not used, it is possible to tune the performance to get the most out of the compute processors of choice using OpenCL.
As mentioned earlier, OpenCL allows for the use of the processor-specific API to be used in case some specific functions are not supported in OpenCL. Therefore, the use of OpenCL would never result in limiting the performance.
This may seem counter-intuitive, since in order to maximize performance, it is necessary to learn the hardware-dependent methods. However, it should be kept in mind that if some code is written in OpenCL, it should work in any environment. Using the hardware-dependent coding method, the main steps taken is coding, debugging, and then tuning. If the code is already debugged and ready in OpenCL, the developer can start directly from the tuning step. Anyone who has developed code in these types of environment should see the benefit in this, since this would reduce the chance of wasting time on bugs that may arise from not having a full understanding of the hardware.

Learning Curve

OpenCL C language, as the name suggests, uses almost the exact same syntax as the C language. It is extended to support SIMD vector operations and multiple memory hierarchies, and some features not required for computations, such as printf(), were taken out.
The control program using the OpenCL runtime API can be written in C or C++, and does not require a special compiler. The developer is required to learn how to use the OpenCL runtime API, but this is not very difficult, especially to those who have experience with developing in heterogeneous environment such as CUDA. Also, once this is written to control one device type, the code will not have to be changed in order to control another device.

Applicable Platforms

This section will introduce the main concepts and key terms necessary to gain familiarity with OpenCL. Some terms may be shortened for the ease of reading (e.g. OpenCL Devices → Devices).

Host + Device

Up until this point, the processors in heterogeneous systems (OpenCL platforms) were called control processors and compute processors, but OpenCL defines these as follows.

1. Host

Environment where the software to control the devices is executed. This is typically the CPU as well as the memory associated with it.

2. OpenCL Devices

Environment where the software to perform computation is executed. GPU, DSP, Cell/B.E., CPU are some of the typically used devices, which contain numerous compute units. The memory associated with the processors is also included in the definition of a device.

There are no rules regarding how the host and the OpenCL device are connected. In the case of CPU + GPU, PCI Express is used most often. For CPU servers, the CPUs can be connected over Ethernet, and use TCP/IP for data transfer.
OpenCL device is expected to contain multiple Compute Units, which is made up of multiple Processing Elements. As an example, the GPU described in OpenCL terms is as follows.

• OpenCL Device - GPU
• Compute Unit - Streaming Multiprocessor (SM)
• Processing Element - Scalar Processor (SP)

OpenCL device will be called a device from this point forward.

Application Structure

This section will discuss a typical application that runs on the host and device.

OpenCL explicitly separates the program run on the host side and the device side.
The program run on the device is called a kernel. When creating an OpenCL application, this kernel is written in OpenCL C, and compiled using the OpenCL compiler for that device.
The host is programmed using C/C++ using the OpenCL runtime API, which is linked to an OpenCL runtime library implemented for the host-device combination. This code is compiled using compilers such as GCC and Visual Studio.
As stated earlier, the host controls the device using the OpenCL runtime API. Since OpenCL expects the device to not be running an operating system, the compiled kernel needs help from the host in order to be executed on the device. Since the OpenCL runtime implementation takes care of these tasks, the programmer will not have to know the exact details of how to perform these tasks, which is unique to each platform combination.

Parallel Programming Models

OpenCL provides API for the following programming models.

• Data parallel programming model
• Task parallel programming model

In the data parallel programming model, the same kernel is passed through the command queue to be executed simultaneously across the compute units or processing elements.
For the task parallel programming model, different kernels are passed through the command queue to be executed on different compute units or processing elements.

For data parallel programming models, each kernel requires a unique index so the processing is performed on different data sets. OpenCL provides this functionality via the concept of index space.
OpenCL gives an ID to a group of kernels to be run on each compute unit, and another ID for each kernel within each compute unit, which is run on each processing element. The ID for the compute unit is called the Workgroup ID, and the ID for the processing element is called the Work Item ID. When the user specifies the total number of work items (global item), and the number of work items to be ran on each compute unit (local item), the IDs are given for each item by the OpenCL runtime API. The index space is defined and accessed by this ID set. The number of workgroups can be computed by dividing the number of global items by the number of local items.

The ID can have up to 3 dimensions to correspond to the data to process. The retrieval of this ID is necessary when programming the kernel, so that each kernel processes different sets of data. The index space is defined for task parallel processing as well, in which both the number of work groups and the work items are 1.

Memory Model

OpenCL allows the kernel to access the following 4 types of memory.

1. Global Memory
Memory that can be read from all work items. It is physically the device's main memory.

2. Constant Memory

Also memory that can be read from all work items. It is physically the device's main memory, but can be used more efficiently than global memory if the compute units contain hardware to support constant memory cache. The cost memory is set and written by the host.

3. Local Memory

Memory that can be read from work items within a work group. It is physically the shared memory on each compute units.

4. Private Memory

Memory that can only be used within each work item. It is physically the registers used by each processing element.

The 4 memory types mentioned above all exist on the device. The host side has its own memory as well. However, only the kernel can only access memory on the device itself. The host, on the other hand, is capable of reading and writing to the global, constant, and the host memory.
Note that the physical location of each memory types specified above assumes NVIDIA's GPUs. The physical location of each memory types may differ depending on the platform. For example, if the devices are x86 multi-core CPUs, all 4 memory types will be physically located on the main memory on the host. However, OpenCL requires the explicit specification of memory types regardless of the platform.

OpenCL Setup

This chapter will setup the OpenCL development environment, and run a simple OpenCL program.

Available OpenCL Environments

This section will introduce the available OpenCL environment as of March, 2010.

FOXC (Fixstars OpenCL Cross Compiler)

FOXC, an OpenCL compiler, along with FOXC runtime, which is the OpenCL runtime implementation, are softwares currently being developed by the Fixstars Corporation. It is still a beta-release as of this writing (March, 2010).

OpenCL users can use the FOXC and FOXC runtime to compile and execute a multi-core x86 program.
Although OpenCL is a framework that allows for parallel programming of heterogeneous systems, it can also be used for homogeneous systems. Therefore, to run program compiled using FOXC, an additional accelerator is unnecessary. Also, since multi-threading using POSIX threads are supported, multiple devices can be running in parallel on a multi-core system.
FOXC is a source-to-source compiler that generates readable C code that uses embedded SSE functions from an OpenCL C code, giving the user a chance to verify the generated code.

Hardware	Multi-core x86 CPU, x86 server
Tested Environment	Yellow Dog Enterprise Linux (x86, 64-bit), Cent OS 5.3 (32-bit, 64-bit)

Further testing will be performed in the future. For the latest information, visit the FOXC website (http://www.fixstars.com/en/foxc)

NVIDIA OpenCL

NVIDIA has released their OpenCL implementation for their GPU. NVIDIA also has a GPGPU development environment known as CUDA, but their OpenCL implementation allows the user to choose either environment.

Hardware	CUDA-enabled graphics board; NVIDIA GeForce 8,9,200 Series NVIDIA Tesla NVIDIA Quadro series
Tested Environment	Windows XP (32-bit, 64-bit) Windows Vista (32-bit, 64-bit) Windows 7 (32-bit, 64-bit) Yellow Dog Enterprise Linux 6.3 (32-bit, 64-bit) Red Had Enterprise Linux 5.3 (32-bit, 64-bit) Ubuntu 8.10 (32-bit, 64-bit) Fedora 8 (32-bit, 64-bit)

The latest CUDA-enabled graphics boards are listed on NVIDIA's website (http://www.nvidia.com/object/cuda_learn_products.html). The latest supported environments are listed at (http://developer.nvidia.com/)

AMD (ATI) OpenCL

ATI used to be a reputable graphic-chip vendor like NVIDIA, but was bought out by AMD in 2006. Some graphics products from AMD still have the ATI brand name.
AMD has a GPGPU development environment called "ATI Stream SDK", which started to officially support OpenCL with their v2.01 release. (March 2010).

AMD's OpenCL environment supports both multi-core x86 CPU, as well as their graphics boards.

Hardware	Graphics boards by AMD; ATI Radeon HD 58xx/57xx/48xx ATI FirePro V8750/8700/7750/5700/3750 AMD FireStream 9270/9250 ATI Mobility Radeon HD 48xx ATI Mobility FirePro M7740 ATI Radeon Embedded E4690 Discrete GPU

Tested Environment	Windows XP SP3 (32-bit)/SP2(64-bit) Windows Vista SP1 (32-bit, 64-bit) Windows 7 (32-bit, 64-bit) OpenSUSE 11.0 (32-bit, 64-bit) Ubuntu 9.04 (32-bit, 64-bit)

The latest list of supported hardware is listed on AMD's website (http://developer.amd.com/gpu/ATIStreamSDKBetaProgram/Pages/default.aspx). The latest supported environments are listed on (http://developer.amd.com/gpu/ATIStreamSDKBetaProgram/Pages/default.aspx)

Apple OpenCL

One of the main features included in MacOS X 10.6 (Snow Leopard) was the Apple OpenCL, which was the first world-wide OpenCL release. The default support of OpenCL in the operating system, which allows developers the access to OpenCL at their fingertips, may help spread the use of OpenCL as a means to accelerate programs via GPGPU. Xcode, which is Apple's development environment, must be installed prior to the use of OpenCL.

Hardware	Any Mac with an Intel CPU
Tested Environment	Mac OS X 10.6 (Snow Leopard)

Apple's OpenCL compiler supports both GPUs and x86 CPUs.

IBM OpenCL

IBM has released the alpha version of OpenCL Development Kit for their BladeCenter QS22 and JS23, which can be downloaded from IBM Alpha Works (http://www.alphaworks.ibm.com/tech/opencl/). QS22 is a blade server featuring two IBM PowerXCell 8i processors, offering five times the double precision performance of the previous Cell/B.E. processor. JS23 is another blade server, made up of two POWER6+ processors, which is IBM's traditional RISC processor.

IBM's OpenCL includes compilers for both POWER processors, as well as for Cell/B.E.
As of this writing (March 2010), they are still alpha releases, which have yet to pass the OpenCL Spec test.

Hardware, OS	IBM BladeCenter QS22 running Fedora 9 IBM BladeCenter JS23 running Red Had Enterprise Linus 5.3

Developing Environment Setup

This section will walk through the setup process of FOXC, Apple, and NVIDIA OpenCL.

FOXC Setup

This setup procedure assumes the installation of FOXC in a 64-bit Linux environment, under the directory "/usr/local". FOXC can be downloaded for free from the Fixstars website (http://www.fixstars.com/en/foxc).

The install package can be uncompressed as follows.

```
> tar -zxf foxc-install-linux64.tar.gz -C /usr/local
```

The following environmental variables must be set.

List 3.1: CShell

```
setenv PATH /usr/local/foxc-install/bin:${PATH}
setenv LD_LIBRARY_PATH /usr/local/foxc-install/lib
```

List 3.2: Bourne Shell

```
export PATH=/usr/local/foxc-install/bin:${PATH}
export LD_LIBRARY_PATH=/usr/local/foxc-install/lib:${LD_LIBRARY_PATH}
```

This completes the installation of FOXC.

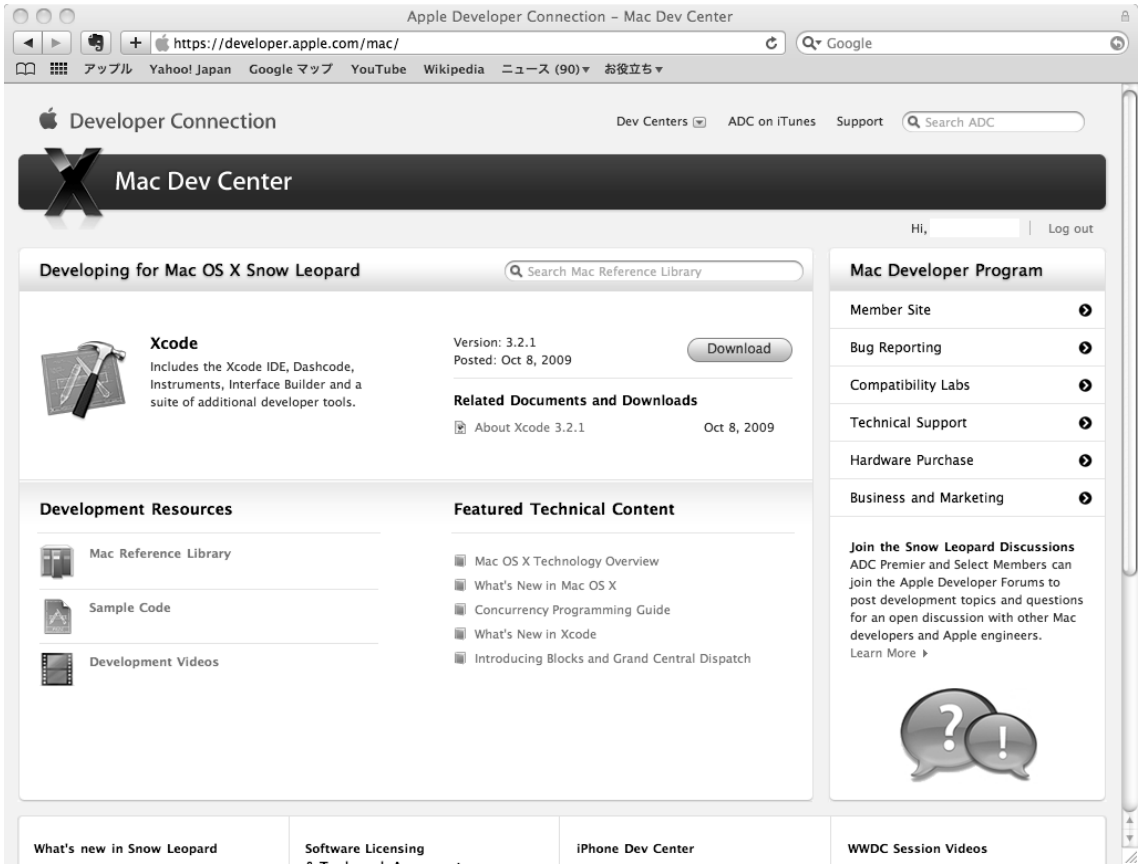
Apple OpenCL Setup

Mac OS X 10.6 (Snow Leopard) is required to use the Apple OpenCL. Also, since OS X does not have the necessary development toolkits (e.g. GCC) at default, these kits must first be downloaded from Apple's developer site.

Download the latest version of "Xcode" from http://developer.apple.com/mac/ (Figure 3.1).

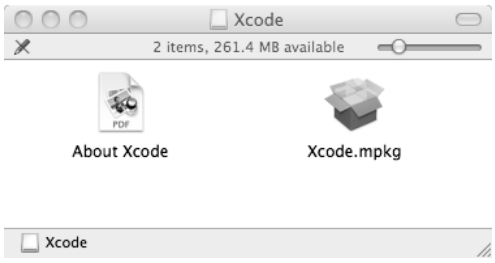
Xcode is a free IDE distributed by Apple, and this includes every tools required to start developing an OpenCL application. As of this writing (March 2010), the latest Xcode version is 3.2.1. You will need an ADC (Apple Developer Connection) account to download this file. The ADC Online account can be created for free.

Figure 3.1: Apple Developer Connection (http://developer.apple.com/mac/)



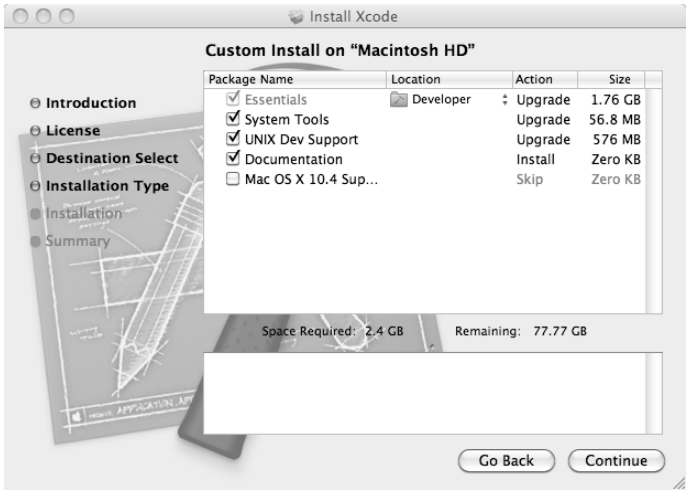
Double click the downloaded .dmg file, which automatically mounts the archive under /Volumes. The archive can be viewed from the Finder. You should see a file called "Xcode.mpkg" (the name can vary depending on the Xcode version, however) (Figure 3.2).

Figure 3.2: Xcode Archive Content



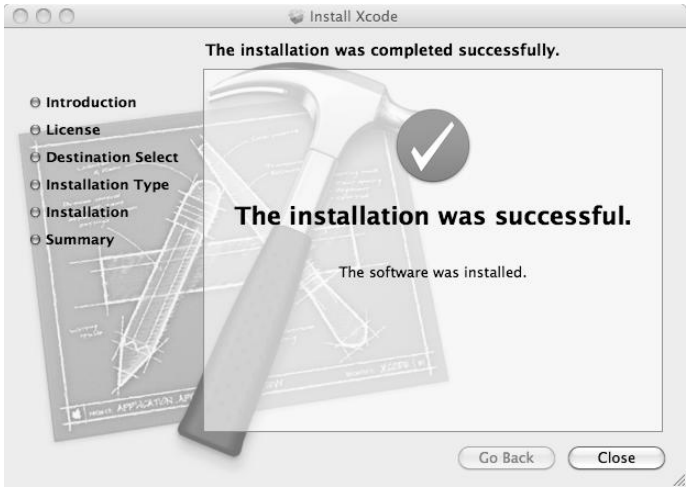
This file is the installation package for Xcode. Double click this file to start the installer. Most of the installation procedure is self-explanatory, but there is one important step. On the screen shown in Figure 3.3, make sure that the box for "UNIX Dev Support" is checked.

Figure 3.3: Custom Installation Screen



Continue onward after making sure that the above box is checked. When you reach the screen shown in Figure 3.4, the installation has finished successfully. You may now use OpenCL on Mac OS X.

Figure 3.4: Successful Xcode installation



NVIDIA OpenCL Setup

NVIDIA OpenCL is supported in multiple platforms. The installation procedure will be split up into that for Linux, and for Windows.

Install NVIDIA OpenCL on Linux

This section will walk through the installation procedure for 64-bit CentOs 5.3. First, you should make sure that your GPU is CUDA-enabled. Type the following on the command line to get the GPU type (# means run as root).

```
# lspci | grep -i nVidia
```

The list of CUDA-Enabled GPUs can be found online at the NVIDIA CUDA Zone (http://www.nvidia.com/object/cuda_gpus.html).
NVIDIA OpenCL uses GCC as the compiler. Make sure GCC is installed, and that its version is 3.4 or 4.x prior to 4.2. Type the following on the command line.

```
> gcc --version
```

If you get an error message such as "command not found: gcc", then GCC is not installed. If the version is not supported for CUDA, an upgrade or a downgrade is required.

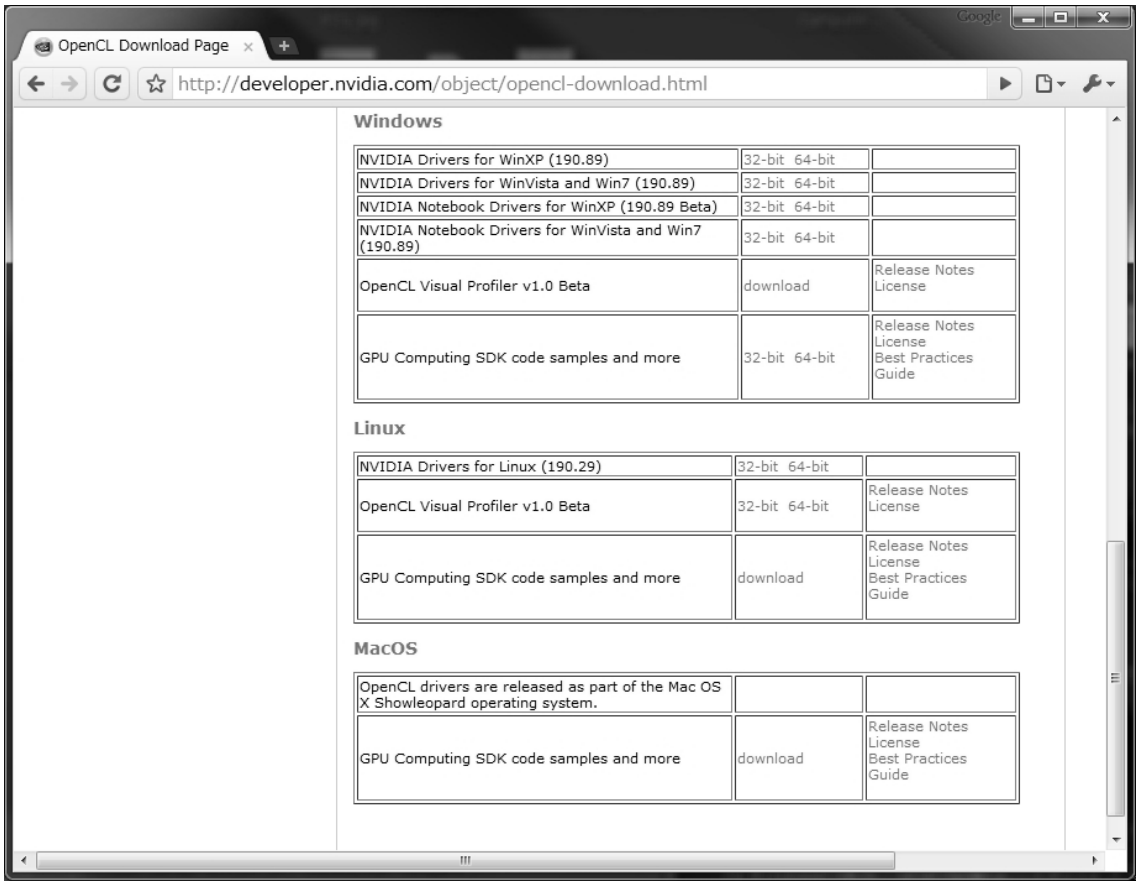
Next, go to the NVIDIA OpenCL download page (http://developer.nvidia.com/object/get-opencl.html) (Figure 3.5). Downloading from this site requires user registration. Download the following 2 files.

• NVIDIA Drivers for Linux (64-bit): nvdrivers_2.3_linux_64_190.29.run
• GPU Computing SDK code samples and more: gpucomputingsdk_2.3b_linux.run

The following is optional, but will prove to be useful.

• OpenCL Visual Profiler v1.0 Beta (64-bit): openclprof_1.0-beta_linux_64.tar.gz

Figure 3.5: NVIDIA OpenCL download page



First install the driver. This requires the X Window System to be stopped. If you are using a display manager, this must be stopped, since it automatically restarts the X Window System. Type the following command as root if using gdb.

```
# service gdm stop
```

Type the following to run Linux in command-line mode.

```
# init 3
```

Execute the downloaded file using shell.

```
# sh nvdrivers_2.3_linux_64_190.29.run
```

You will get numerous prompts, which you can answer "OK" unless you are running in a special environment. Restart the X Window System after a successful installation.

```
# startx
```

At this point, log off as root, and log back in as a user, as this is required to install the "GPU Computing SDK code samples and more" package. Type the following on the command line.

```
> sh gpucomputingsdk_2.3b_linux.run
```

You will be prompted for an install directory, which is $HOME by default. This completes the installation procedure for Linux.

Install NVIDIA OpenCL on Windows

This section will walk through the installation on 32-bit Windows Vista.

The GPU type can be verified on the "Device Manager", under "Display Adapter". Go to the CUDA-Enabled GPU list (link given in the Linux section) to make sure your GPU is supported. Building of the SDK samples requires either Microsoft Visual Studio 8 (2005) or Visual Studio 9 (2008) (Express Edition and above). Make sure this is installed.

Next, go to the NVIDIA OpenCL download page (http://developer.nvidia.com/object/get-opencl.html) (Figure 3.5). Downloading from this site requires user registration. Download the following 2 files.

• NVIDIA Drivers for WinVista and Win7 (190.89): nvdrivers_2.3_winvista_32_190.89_general.exe
• GPU Computing SDK code samples and more: gpucomputingsdk_2.3b_win_32.exe

The following is optional, but will prove to be useful.

• OpenCL Visual Profiler v1.0 Beta (64-bit): openclprof_1.0-beta_windows.zip

Double click the download executable for the NVIDIA driver in order to start the installation procedure. Restart your PC when prompted, after the NVIDIA driver is installed.

Next, install the GPU Computing SDK. Installation of the SDK requires user registration. After installation is completed successfully, the SDK will be installed in the following directories:

• **Windows XP**

C:¥Documents and Settings¥All Users¥Application Data¥NVIDIA Corporation¥NVIDIA GPU Computing SDK¥OpenCL

• **Windows Vista / Windows 7**

C:¥ProgramData¥NVIDIA Corporation¥NVIDIA GPU Computing SDK¥OpenCL

Restart your system after the SDK installation.

You should now be able to run the OpenCL sample program inside the SDK. To do so, double click on the "NVIDIA GPU Computing SDK Browser" icon which should have been created on your desktop. You will see a list of all the sample programs in the SDK. Open the "OpenCL Samples" tab to see all the sample code written in OpenCL. The sample code is listed in increasing level of difficulty. Some samples include a "Whitepaper", which is a detailed technical reference for that sample.
Syntax highlighting of OpenCL code on Visual Studio is not supported by default. The following procedure will enable syntax highlighting.

1. Copy usertype.dat

Under the "doc" folder found inside the SDK installation directory, you should see a file called "usertype.dat." Copy this file to the following directory.

If running on 32-bit Windows

```
C:¥Program Files¥Microsoft Visual Studio 8¥Common7¥IDE (VC8)
C:¥Program Files¥Microsoft Visual Studio 9¥Common7¥IDE (VC9)
```

If running on 64-bit Windows

```
C:¥Program Files (x86)¥Microsoft Visual Studio 8¥Common7¥IDE (VC8)
C:¥Program Files (x86)¥Microsoft Visual Studio 9¥Common7¥IDE (VC9)
```

If "usertype.dat" already exists in the directory, use an editor to append the content of NVIDIA SDK's "usertype.dat" to the existing file.

2. Visual Studio Setup

Start Visual Studio, go to the "Tool" Menu Bar, and then click on "Option". On the left panel, select "Text Editor", then "File Extension". On the right panel, type ".cl" to the box labeled

"Extension". Select "Microsoft Visual C++" on the drop-down menu, and then click the "Add" button.

Visual Studio will now perform syntax highlighting on .cl files.
When developing an OpenCL program, the required files for building are located as follows.

• Header File (Default: "NVIDIA GPU Computing SDK¥OpenCL¥common¥inc¥CL")
  cl.h, cl_gl.h, cl_platform.h, clext.h
• Library (Default: "NVIDIA GPU Computing SDK¥OpenCL¥common¥lib¥[Win32|x64]"
  OpenCL.lib (32-bit version and 64-bit versions exist)
• Dynamic Link Library (Default: "¥Windows¥system32")
  OpenCL.dll

If an application explicitly specifies the link to OpenCL.dll, OpenCL.lib is not required at runtime.

First OpenCL Program

From this section onward, we will start learning the OpenCL programming basics by building and running actual code. Since we have not yet gone over the OpenCL grammar, you should concentrate on the general flow of OpenCL programming.

Hello World

List 3.3 and 3.4 shows the familiar "Hello, World!" program, written in OpenCL. Since standard in/out cannot be used within the kernel, we will use the kernel only to set the char array. In this program, the string set on the kernel will be copied over to the host side, which can then be outputted. (The code can be downloaded from http://www.fixstars.com/books/opencl)

List 3.3: Hello World - kernel (hello.cl)

```
001: __kernel void hello(__global char* string)
002: {
003: string[0] = 'H';
004: string[1] = 'e';
005: string[2] = 'l';
006: string[3] = 'l';
```

```
007: string[4] = 'o';
008: string[5] = ',';
009: string[6] = ' ';
010: string[7] = 'W';
011: string[8] = 'o';
012: string[9] = 'r';
013: string[10] = 'l';
014: string[11] = 'd';
015: string[12] = '!';
016: string[13] = '¥0';
017: }
```

List 3.4: Hello World - host (hello.c)

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: #ifdef __APPLE__
005: #include <OpenCL/opencl.h>
006: #else
007: #include <CL/cl.h>
008: #endif
009:
010: #define MEM_SIZE (128)
011: #define MAX_SOURCE_SIZE (0x100000)
012:
013: int main()
014: {
015: cl_device_id device_id = NULL;
016: cl_context context = NULL;
017: cl_command_queue command_queue = NULL;
018: cl_mem memobj = NULL;
019: cl_program program = NULL;
020: cl_kernel kernel = NULL;
021: cl_platform_id platform_id = NULL;
022: cl_uint ret_num_devices;
023: cl_uint ret_num_platforms;
```

```
024: cl_int ret;
025:
026: char string[MEM_SIZE];
027:
028: FILE *fp;
029: char fileName[] = "./hello.cl";
030: char *source_str;
031: size_t source_size;
032:
033: /* Load the source code containing the kernel*/
034: fp = fopen(fileName, "r");
035: if (!fp) {
036: fprintf(stderr, "Failed to load kernel.¥n");
037: exit(1);
038: }
039: source_str = (char*)malloc(MAX_SOURCE_SIZE);
040: source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
041: fclose(fp);
042:
043: /* Get Platform and Device Info */
044: ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
045: ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);
046:
047: /* Create OpenCL context */
048: context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
049:
050: /* Create Command Queue */
051: command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
052:
053: /* Create Memory Buffer */
054: memobj = clCreateBuffer(context, CL_MEM_READ_WRITE,MEM_SIZE * sizeof(char), NULL,
&ret);
055:
056: /* Create Kernel Program from the source */
057: program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
```

```
058: (const size_t *)&source_size, &ret);
059:
060: /* Build Kernel Program */
061: ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
062:
063: /* Create OpenCL Kernel */
064: kernel = clCreateKernel(program, "hello", &ret);
065:
066: /* Set OpenCL Kernel Parameters */
067: ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);
068:
069: /* Execute OpenCL Kernel */
070: ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);
071:
072: /* Copy results from the memory buffer */
073: ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0,
074: MEM_SIZE * sizeof(char),string, 0, NULL, NULL);
075:
076: /* Display Result */
077: puts(string);
078:
079: /* Finalization */
080: ret = clFlush(command_queue);
081: ret = clFinish(command_queue);
082: ret = clReleaseKernel(kernel);
083: ret = clReleaseProgram(program);
084: ret = clReleaseMemObject(memobj);
085: ret = clReleaseCommandQueue(command_queue);
086: ret = clReleaseContext(context);
087:
088: free(source_str);
089:
090: return 0;
091: }
```

The include header is located in a different directory depending on the environment (Table 3.1).

Make sure to specify the correct location.

Table 3.1: Include header location (as of March 2010)

OpenCL implementation	Include Header
AMD	CL/cl.h
Apple	OpenCL/opencl.h
FOXC	CL/cl.h
NVIDIA	CL/cl.h

The sample code defines the following macro so that the header is correctly included in any environment.

List 3.1: Include Header Location (As of March, 2010)

```
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <cl.h>
#endif
```

Building in Linux/Mac OS X

Once the program is written, we are now ready to build and run the program. This section will describe the procedure under Linux/Mac OS X. The kernel and host code are assumed to exist within the same directory.

The procedures for building vary depending on the OpenCL implementation. "path-to-..." should be replaced with the corresponding OpenCL SDK path. The default SDK path is as shown in Figure 3.2.

Table 3.2: Path to SDK

SDK	Path
AMD Stream SDK 2.0 beta4	Path-to-AMD/ati-stream-sdk-v2.0-beta4-lnx32 (32-bit Linux) Path-to-AMD/ati-stream-sdk-v2.0-beta4-lnx64 (64-bit Linux)
FOXC	Path-to-foxc/foxc-install

NVIDIA GPU Computing SDK	$(HOME)/NVIDIA_GPU_Computing_SDK (Linux)

The build command on Linux/Max OS X are as follows:

AMD OpenCL

```
> gcc -I /path-to-AMD/include -L/path-to-AMD/lib/x86 -o hello hello.c
-Wl,-rpath,/path-to-AMD/lib/x86 -lOpenCL （32-bit Linux）
> gcc -I /path-to-AMD/include -L/path-to-AMD/lib/x86_64 -o hello hello.c
-Wl,-rpath,/path-to-AMD/lib/x86_64 -lOpenCL (64-bit Linux)
```

FOXC

```
> gcc -I /path-to-foxc/include -L /path-to-foxc/lib -o hello hello.c -Wl,-rpath,/path-to-foxc/lib
-lOpenCL
```

Apple

```
> gcc -o hello hello.c -framework opencl
```

NVIDIA

```
> gcc -I /path-to-NVIDIA/OpenCL/common/inc -L /path-to-NVIDIA/OpenCL/common/lib/Linux32
-o hello hello.c -lOpenCL （32-bit Linux）
> gcc -I /path-to-NVIDIA/OpenCL/common/inc -L /path-to-NVIDIA/OpenCL/common/lib/Linux64
-o hello hello.c -lOpenCL （64-bit Linux）
```

Alternatively, you can use the Makefile included with the sample code to run the OpenCL code in various platforms as written below.

```
> make amd (Linux)
> make apple (Mac OS X)
> make foxc (Linux)
> make nvidia (Linux)
```

This should create an executable with the name "hello" in working directory. Run the executable as follows. If successful, you should get "Hello World!" on the screen.

```
> ./hello
```

```
Hello World!
```

Building on Visual Studio

This section will walk through the building and execution process using Visual C++ 2008 Express under 32-bit Windows Vista environment. The OpenCL header file and library can be included to be used on a project using the following steps.

1. From the project page, go to “C/C++” -> “General”, then add the following in the box for “Additional include directories”:

NVIDIA

```
C:¥ProgramData¥NVIDIA Corporation¥NVIDIA GPU Computing SDK¥OpenCL¥common¥inc
```

AMD

```
C:¥Program Files¥ATI Stream¥include
```

2. From the project page, go to “Linker” -> “Input”, and in the box for “Additional library path”, type the following.

NVIDIA

```
C:¥ProgramData¥NVIDIA Corporation¥NVIDIA GPU Computing
SDK¥OpenCL¥common¥lib¥Win32
```

AMD

```
C:¥Program Files¥ATI Stream¥lib¥x86
```

3. From the project page, go to "Linker" -> "Input", and in the box for "Additional Dependencies", type the following.

Both NVIDIA and AMD

```
OpenCL.lib
```

These should apply to All Configurations, which can be selected on the pull-down menu located on the top left corner.
The environment should now be setup to allow an OpenCL code to be built on. Build and run the

sample code, and make sure you get the correct output.

Basic OpenCL

This chapter will delve further into writing codes for the host and the device. After reading this chapter, you should have the tools necessary for implementing a simple OpenCL program.

Basic Program Flow

The previous chapter introduced the procedure for building and running an existing sample code. By now, you should have an idea of the basic role of a host program and a kernel program. You should also know the difference between a host memory and a device memory, as well as when to use them.

This section will walk through the actual code of the "Hello, World!" program introduced in the last chapter.

OpenCL Program

Creating an OpenCL program requires writing codes for the host side, as well as for the device side. The device is programmed in OpenCL C, as shown in List 3.3 hello.cl. The host is programmed in C/C++ using an OpenCL runtime API, as shown in List 3.4 hello.c.

The sections to follow will walk through each code.

Kernel Code

The function to be executed on the device is defined as shown in List 4.1. The OpenCL grammar will be explained in detail in Chapter 5, but for the time being, you can think of it as being the same as the standard C language.

List 4.1: Declaring a function to be executed on the kernel

```
001:  __kernel void hello(__global char * string)
```

The only differences with standard C are the following.

• The function specifier "__kernel" is used when declaring the "hello" function
• The address specifier "__global" is used to define the function's string argument

The "__kernel" specifies the function to be executed on the device. It must be called by the host, which is done by using one of the following OpenCL runtime API commands.
• Task call: clEnqueueTask()
• Data-parallel call: clEnqueueNDRangeKernel()

Since the kernel is not of a data-parallel nature, the clEnqueueTask() is called from the host to process the kernel.

The __global address specifier for the variable "string" tells the kernel that the address space to be used is part of the OpenCL global memory, which is the device-side main memory.

The kernel is only allowed read/write access to global, constant, local, and private memory, which is specified by __global, __constant, __local, __private, respectively. If this is not specified, it will assume the address space to be __private, which is the device-side register.

Host Code

The host program tells the device to execute the kernel using the OpenCL runtime API.

Telling the device to execute the hello() kernel only requires the clEnqueueTask() command in the OpenCL runtime API, but other setup procedures must be performed in order to actually run the code. The procedure, which includes initialization, execution, and finalization, is listed below.

1. Get a list of available platforms
2. Select device
3. Create Context
4. Create command queue
5. Create memory objects
6. Read kernel file
7. Create program object
8. Compile kernel
9. Create kernel object
10. Set kernel arguments
11. Execute kernel (Enqueue task) ← hello() kernel function is called here
12. Read memory object

13. Free objects

We will go through each step of the procedure using hello.c as an example.

Get a List of Available Platform

The first thing that must be done on the host-side is to get a list of available OpenCL platforms. This is done in the following code segment from hello.c in List 3.4.

```
021:    cl_platform_id platform_id = NULL
...
023:    cl_uint ret_num_platforms;
...
044:    ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
```

The platform model in OpenCL consists of a host connected to one or more OpenCL devices.

The clGetPlatformIDs() function in line 44 allows the host program to discover OpenCL devices, which is returned as a list to the pointer platform_id of type cl_platform_id. The 1st argument specifies how many OpenCL platforms to find which most of the time is one. The 3rd argument returns the number of OpenCL platforms that can be used.

Select Device

The next step is to select a GPU device within the platform. This is done in the following code segment from hello.c in List 3.4.

```
015:    cl_device_id device_id = NULL;
...
022:    cl_uint ret_num_devices;
...
045:    ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);
```

The clGetDeviceIDs() function in line 45 selects the device to be used. The 1st argument is the platform that contains the desired device. The 2nd argument specifies the device type. In this case, CL_DEVICE_TYPE_DEFAULT is passed, which specifies whatever set as default in the platform to be used as the device. If the desired device is the GPU, then this should be CL_DEVICE_TYPE_GPU, and if it is CPU, then this should be CL_DEVICE_TYPE_CPU.

The 3rd argument specifies the number of devices to use. The 4th argument returns the handle to the selected device. The 5th argument returns the number of devices that corresponds to the device type specified in the 2nd argument. If the specified device does not exist, then ret_num_devices will be set to 0.

Create Context

After getting the device handle, the next step is to create an OpenCL Context. This is done in the following code segment from hello.c in List 3.4.

```
016:    cl_context context = NULL;
...
048:    context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
```

An OpenCL context is created with one or more devices, using the clCreateContext() function in line 48. The 2nd argument specifies the number of devices to use. The 3rd argument specifies the list of device handlers. The context is used by the OpenCL runtime for managing objects, which will be discussed later.

Create Command Queue

The command queue is used to control the device. In OpenCL, any command from the host to the device, such as kernel execution or memory copy, is performed through this command queue. For each device, one or more command queue objects must be created. This is done in the following code segment from hello.c in List 3.4.

```
017:    cl_command_queue command_queue = NULL;
...
051:    command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
```

The clCreateCommandQueue() function in line 51 creates the command queue. The 1st argument specifies the context in which the command queue will become part of. The 2nd argument specifies the device which will execute the command in the queue. The function returns a handle to the command queue, which will be used for memory copy and kernel execution.

Create Memory Object

To execute a kernel, all data to be processed must be on the device memory. However, the kernel does not have the capability to access memory outside of the device. Therefore, this action must be performed on the host-side. To do this, a memory object must be created, which allows the host to access the device memory. This is done in the following code segment from hello.c in List 3.4.

```
018:    cl_mem memobj = NULL;
...
054:    memobj = clCreateBuffer(context, CL_MEM_READ_WRITE, MEM_SIZE * sizeof(char),
NULL, &ret);
```

The clCreateBuffer() function in line 54 allocates space on the device memory [14]. The allocated memory can be accessed from the host side using the returned memobj pointer. The 1st argument specifies the context in which the memory object will become part of. The 2nd argument specifies a flag describing how it will be used. The CL_MEM_READ_WRITE flag allows the kernel to both read and write to the allocated device memory. The 3rd argument specifies the number of bytes to allocate. In this example, this allocated storage space gets the character string "Hello, World!", which is done in the hello.cl kernel in list 3.3.

Read Kernel File

As mentioned earlier, the kernel can only be executed via the host-side program. To do this, the host program must first read the kernel program. This may be in a form of an executable binary, or a source code which must be compiled using an OpenCL compiler. In this example, the host reads the kernel source code. This is done using the standard fread() function.

```
028: FILE *fp;
029: char fileName[] = "./hello.cl";
030: char *source_str;
031: size_t source_size;
032:
033: /* Load kernel code */
034: fp = fopen(fileName, "r");
035: if (!fp) {
036: fprintf(stderr, "Failed to load kernel.¥n");
037: exit(1);
038: }
```

```
039: source_str = (char*)malloc(MAX_SOURCE_SIZE);
040: source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
041: fclose(fp);
```

Create Program Object

Once the source code is read, this code must be made into a kernel program. This step is required, since the kernel program can contain multiple kernel functions [15]. This is done by creating a program object, as shown in the following code segment from hello.c in List 3.4.

```
019:    cl_program program = NULL;
...
057:    program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
058:                                        (const size_t *)&source_size, &ret);
```

The program object is created using the clCreateProgramWithSource() function. The 3rd argument specifies the read-in source code, and the 4th argument specifies the size of the source code in bytes. If the program object is to be created from a binary, clCreateProgramWithBinary() is used instead.

Compile

The next step is to compile the program object using an OpenCL C compiler.

```
061:    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

The clBuildProgram in line 61 builds the program object to create a binary. The 1st argument is the program object to be compiled. The 3rd argument is the target device for which the binary is created. The 2nd argument is the number of target devices. The 4th argument specifies the compiler option string.

Note that this step is unnecessary if the program is created from binary using clCreateProgramWithBinary().

Create Kernel Object

Once the program object is compiled, the next step is to create a kernel object. Each kernel object corresponds to each kernel function. Therefore, it is necessary to specify the kernel function's name when creating the kernel object.

```
020: cl_kernel kernel = NULL;
...
064: kernel = clCreateKernel(program, "hello", &ret);
```

The clCreateKernel() function in line 64 creates the kernel object from the program. The 1st argument specifies the program object, and the 2nd argument sets the kernel function name.

This example only has one kernel function for one program object, but it is possible to have multiple kernel functions for one program object. However, since each kernel object corresponds to one kernel function, the clCreateKernel() function must be called multiple times.

Set Kernel Arguments

Once the kernel object is created, the arguments to the kernel must be set. In this example, hello.cl in List 3.3 expects a pointer to a string array to be built on the device side. This pointer must be specified on the host-side. In this example, the pointer to the allocated memory object is passed in. In this way, the memory management can be performed on the host-side.

```
067:      ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &memobj);
```

The clSetKernelArg() function in line 67 sets the arguments to be passed into the kernel. The 1st argument is the kernel object. The 2nd argument selects which argument of the kernel is being passed in, which is 0 in this example, meaning the 0th argument to the kernel is being set. The 4th argument is the pointer to the argument to be past in, with the 3rd argument specifying this argument size in bytes. In this way, the clSetKernelArg() must be called for each kernel arguments.

Passing of the host-side data as a kernel argument can be done as follows.

```
int a =10;
clSetKernelArg(kernel, 0, sizeof(int), (void *)&a);
```

Execute Kernel (Enqueue task)

The kernel can now finally be executed. This is done by the code segment from hello.c in List 3.4

```
070:    ret = clEnqueueTask(command_queue, kernel, 0, NULL, NULL);
```

This throws the kernel into the command queue, to be executed on the compute unit on the device. Note that this function is asynchronous, meaning it just throws the kernel into the queue to be executed on the device. The code that follows the clEnqueueTask() function should account for this.

In order to wait for the kernel to finish executing, the 5th argument of the above function must be set as an event object. This will be explained in "4-3-3 Task Parallelism and Event Object". Also, the clEnqueueTask() function is used for task-parallel instructions. Data-parallel instructions should use the clEnqueueNDRangeKernel() function instead.

Read from the memory object

After processing the data on the kernel, the result must now be transferred back to the host-side. This is done in the following code segment from hello.c in List 3.4.

```
026:    char string[MEM_SIZE];
...
073:    ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0,
074:                                    MEM_SIZE * sizeof(char), string, 0, NULL, NULL);
```

The clEnqueueReadBuffer() function in lines 73~74 copies the data on the device-side memory to the host-side memory. To copy the data from the host-side memory to the device-side memory, the clEnqueueWriteBuffer() function is used instead. As the term "Enqueue" in the function indicates, the data copy instruction is placed in the command queue before it is processed. The 2nd argument is the pointer to the memory on the device to be copied over to the host side, while the 5th argument specifies the size of the data in bytes. The 6th argument is the pointer to the host-side memory where the data is copied over to. The 3rd argument specifies whether the command is synchronous or asynchronous. The "CL_TRUE" that is passed in makes the function synchronous, which keeps the host from executing the next command until the data copy finishes. If "CL_FALSE" is passed in instead, the copy becomes asynchronous, which queues the task and immediately executes the next instruction on the host-side.

Recall, however, that the "hello" kernel was queued asynchronously. This should make you question whether the memory copy from the device is reading valid data.

Looking just at the host-side code, it might make you think that this is a mistake. However, in this case it is OK. This is because when the command queue was created, the 3rd argument passed in was a 0, which makes the queued commands execute in order. Therefore, the data copy command waits until the previous command in the queue, the "hello" kernel, is finished. If the command queue was set to allow asynchronous execution, the data copy may start before the kernel finishes its processing of the data, which will achieve incorrect results. Asynchronous kernel execution may be required in some cases, however, and a way to do this is explained in "4-3 Kernel Call".

Free Objects

Lastly, all the objects need to be freed. This is done in the code segment shown below from hello.c in List 3.4.

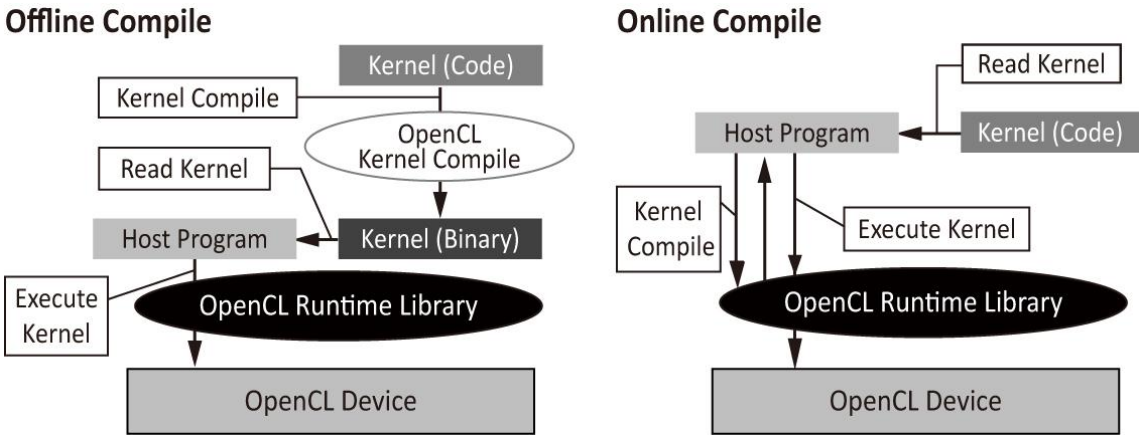
```
082:    ret = clReleaseKernel(kernel);
083:    ret = clReleaseProgram(program);
084:    ret = clReleaseMemObject(memobj);
085:    ret = clReleaseCommandQueue(command_queue);
086:    ret = clReleaseContext(context);
```

In real-life applications, the main course of action is usually a repetition of setting kernel arguments or the host-to-device copy -> kernel execution -> device-to-host copy cycle, so object creation/deletion cycle do not usually have to be repeated, since the same object can be used repeatedly. If too many objects are created without freeing, the host-side's object management memory space may run out, in which case the OpenCL runtime will throw an error.

Online/Offline Compilation

In OpenCL, a kernel can be compiled either online or offline (Figure 4.1).

Figure 4.1: Offline and Online Compilation



The basic difference between the 2 methods is as follows:

• Offline: Kernel binary is read in by the host code
• Online: Kernel source file is read in by the host code

In "offline-compilation", the kernel is pre-built using an OpenCL compiler, and the generated binary is what gets loaded using the OpenCL API. Since the kernel binary is already built, the time lag between starting the host code and the kernel getting executed is negligible. The problem with this method is that in order to execute the program on various platforms, multiple kernel binaries must be included, thus increasing the size of the executable file.

In "online-compilation", the kernel is built from source during runtime using the OpenCL runtime library. This method is commonly known as JIT (Just in time) compilation. The advantage of this method is that the host-side binary can be distributed in a form that's not device-dependent, and that adaptive compiling of the kernel is possible. It also makes the testing of the kernel easier during development, since it gets rid of the need to compile the kernel each time. However, this is not suited for embedded systems that require real-time processing. Also, since the kernel code is in readable form, this method may not be suited for commercial applications.

The OpenCL runtime library contains the set of APIs that performs the above operations. In a way, since OpenCL is a programming framework for heterogeneous environments, the online compilation support should not come as a shock. In fact, a stand-alone OpenCL compiler is not available for the OpenCL environment by NVIDIA, AMD, and Apple. Hence, in order to create a kernel binary in these environments, the built kernels has to be written to a file during runtime by the host program. FOXC on the other hand includes a stand-alone OpenCL kernel compiler, which makes the process of making a kernel binary intuitive.

We will now look at sample programs that show the two compilation methods. The first code shows the online compilation version.

List 4.2: Online Compilation version

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: #ifdef __APPLE__
005: #include <OpenCL/opencl.h>
006: #else
007: #include <CL/cl.h>
008: #endif
009:
010: #define MEM_SIZE (128)
011: #define MAX_SOURCE_SIZE (0x100000)
012:
013: int main()
014: {
015:     cl_platform_id platform_id = NULL;
016:     cl_device_id device_id = NULL;
017:     cl_context context = NULL;
018:     cl_command_queue command_queue = NULL;
019:     cl_mem memobj = NULL;
020:     cl_program program = NULL;
021:     cl_kernel kernel = NULL;
022:     cl_uint ret_num_devices;
023:     cl_uint ret_num_platforms;
024:     cl_int ret;
025:
026:     float mem[MEM_SIZE];
027:
028:     FILE *fp;
029:     const char fileName[] = "./kernel.cl";
030:     size_t source_size;
031:     char *source_str;
```

```
032:       cl_int i;
033:
034:       /* Load kernel source code */
035:       fp = fopen(fileName, "r");
036:       if (!fp) {
037:                  fprintf(stderr, "Failed to load kernel.¥n");
038:                  exit(1);
039:        }
040:       source_str = (char *)malloc(MAX_SOURCE_SIZE);
041:       source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
042:        fclose(fp);
043:
044:        /*Initialize Data */
045:       for (i = 0; i < MEM_SIZE; i++) {
046:                  mem[i] = i;
047:       }
048:
049:        /* Get platform/device information */
050:       ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
051:       ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);
052:
053:        /* Create OpenCL Context */
054:       context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
055:
056:        /* Create Command Queue */
057:       command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
058:
059:       /* Create memory buffer*/
060:       memobj = clCreateBuffer(context, CL_MEM_READ_WRITE, MEM_SIZE * sizeof(float),
NULL, &ret);
061:
062:       /* Transfer data to memory buffer */
063:       ret = clEnqueueWriteBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE *
sizeof(float), mem, 0, NULL, NULL);
064:
```

```
065:       /* Create Kernel program from the read in source */
066:       program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const
size_t *)&source_size, &ret);
067:
068:       /* Build Kernel Program */
069:       ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
070:
071:       /* Create OpenCL Kernel */
072:       kernel = clCreateKernel(program, "vecAdd", &ret);
073:
074:       /* Set OpenCL kernel argument */
075:       ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);
076:
077:       size_t global_work_size[3] = {MEM_SIZE, 0, 0};
078:       size_t local_work_size[3] = {MEM_SIZE, 0, 0};
079:
080:        /* Execute OpenCL kernel */
081:       ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global_work_size,
local_work_size, 0, NULL, NULL);
082:
083:       /* Transfer result from the memory buffer */
084:       ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE *
sizeof(float), mem, 0, NULL, NULL);
085:
086:        /* Display result */
087:       for (i=0; i < MEM_SIZE; i++) {
088:                  printf("mem[%d] : %f¥n", i, mem[i]);
089:        }
090:
091:        /* Finalization */
092:       ret = clFlush(command_queue);
093:       ret = clFinish(command_queue);
094:       ret = clReleaseKernel(kernel);
095:       ret = clReleaseProgram(program);
096:       ret = clReleaseMemObject(memobj);
097:       ret = clReleaseCommandQueue(command_queue);
```

```
098:      ret = clReleaseContext(context);
099:
100:      free(source_str);
101:
102:      return 0;
103:      }
```

The following code shows the offline compilation version (List 4.3).

List 4.3 Offline compilation version

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: #ifdef __APPLE__
005: #include <OpenCL/opencl.h>
006: #else
007: #include <CL/cl.h>
008: #endif
009:
010: #define MEM_SIZE (128)
011: #define MAX_BINARY_SIZE (0x100000)
012:
013: int main()
014: {
015:      cl_platform_id platform_id = NULL;
016:      cl_device_id device_id = NULL;
017:      cl_context context = NULL;
018:      cl_command_queue command_queue = NULL;
019:      cl_mem memobj = NULL;
020:      cl_program program = NULL;
021:      cl_kernel kernel = NULL;
022:      cl_uint ret_num_devices;
023:      cl_uint ret_num_platforms;
024:      cl_int ret;
025:
026:      float mem[MEM_SIZE];
```

```
027:
028:        FILE *fp;
029:        char fileName[] = "./kernel.clbin";
030:        size_t binary_size;
031:        char *binary_buf;
032:        cl_int binary_status;
033:        cl_int i;
034:
035:        /* Load kernel binary */
036:        fp = fopen(fileName, "r");
037:        if (!fp) {
038:                    fprintf(stderr, "Failed to load kernel.¥n");
039:                    exit(1);
040:        }
041:        binary_buf = (char *)malloc(MAX_BINARY_SIZE);
042:        binary_size = fread(binary_buf, 1, MAX_BINARY_SIZE, fp);
043:        fclose(fp);
044:
045:         /* Initialize input data */
046:        for (i = 0; i < MEM_SIZE; i++) {
047:                    mem[i] = i;
048:        }
049:
050:        /* Get platform/device information */
051:        ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
052:        ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);
053:
054:         /* Create OpenCL context*/
055:        context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
056:
057:        /* Create command queue */
058:        command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
059:
060:         /* Create memory buffer */
061:        memobj = clCreateBuffer(context, CL_MEM_READ_WRITE, MEM_SIZE * sizeof(float),
```

```
NULL, &ret);
062:
063:	/* Transfer data over to the memory buffer */
064:	ret = clEnqueueWriteBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE *
sizeof(float), mem, 0, NULL, NULL);
065:
066:	/* Create kernel program from the kernel binary */
067:	program = clCreateProgramWithBinary(context, 1, &device_id, (const size_t
*)&binary_size,
068:	(const unsigned char **)&binary_buf, &binary_status, &ret);
069:
070:	/* Create OpenCL kernel */
071:	kernel = clCreateKernel(program, "vecAdd", &ret);
072:	printf("err:%d¥n", ret);
073:
074:	/* Set OpenCL kernel arguments */
075:	ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);
076:
077:	size_t global_work_size[3] = {MEM_SIZE, 0, 0};
078:	size_t local_work_size[3] = {MEM_SIZE, 0, 0};
079:
080:	/* Execute OpenCL kernel */
081:	ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global_work_size,
local_work_size, 0, NULL, NULL);
082:
083:	/* Copy result from the memory buffer */
084:	ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE *
sizeof(float), mem, 0, NULL, NULL);
085:
086:	/* Display results */
087:	for (i=0; i < MEM_SIZE; i++) {
088:		printf("mem[%d] :%f¥n", i, mem[i]);
089:	}
090:
091:	/* Finalization */
092:	ret = clFlush(command_queue);
```

```
093:      ret = clFinish(command_queue);
094:      ret = clReleaseKernel(kernel);
095:      ret = clReleaseProgram(program);
096:      ret = clReleaseMemObject(memobj);
097:      ret = clReleaseCommandQueue(command_queue);
098:      ret = clReleaseContext(context);
099:
100:      free(binary_buf);
101:
102:      return 0;
103: }
```

The kernel program performs vector addition. It is shown below in list 4.4.

List 4.4: Kernel program

```
001: __kernel void vecAdd(__global float* a)
002: {
003:      int gid = get_global_id(0);
004:
005:      a[gid] += a[gid];
006: }
```

We will take a look at the host programs shown in List 4.2 and List 4.3. The two programs are almost identical, so we will focus on their differences.
The first major difference is the fact that the kernel source code is read in the online compile version (List 4.5).

List 4.5: Online compilation version - Reading the kernel source code

```
035:      fp = fopen(fileName, "r");
036:      if (!fp) {
037:                fprintf(stderr, "Failed to load kernel.¥n");
038:                exit(1);
039:      }
040:      source_str = (char *)malloc(MAX_SOURCE_SIZE);
041:      source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
042:      fclose(fp);
```

The source_str variable is a character array that merely contains the content of the source file. In order to execute this code on the kernel, it must be built using the runtime compiler. This is done by the code segment shown below in List 4.6.

List 4.6: Online compilation version - Create kernel program

```
065:    /* Create kernel program from the source */
066:    program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const
size_t *)&source_size, &ret);
067:
068:    /* Build kernel program */
069:    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

The program is first created from source, and then built.

Next, we will look at the source code for the offline compilation version (List 4.7).

List 4.7: Offline compilation - Reading the kernel binary

```
036:    fp = fopen(fileName, "r");
037:    if (!fp) {
038:            fprintf(stderr, "Failed to load kernel.¥n");
039:            exit(1);
040:    }
041:    binary_buf = (char *)malloc(MAX_BINARY_SIZE);
042:    binary_size = fread(binary_buf, 1, MAX_BINARY_SIZE, fp);
043:    fclose(fp);
```

The code looks very similar to the online version, since the data is being read into a buffer of type char. The difference is that the data on the buffer can be directly executed. This means that the kernel source code must be compiled beforehand using an OpenCL compiler. In FOXC, this can be done as follows.

```
> /path-to-foxc/bin/foxc -o kernel.clbin kernel.cl
```

The online compilation version required 2 steps to build the kernel program. With offline compilation, the clCreateProgramWithSource() is replaced with clCreateProgramWithBinary.

```
067:      program = clCreateProgramWithBinary(context, 1, &device_id, (const size_t
*)&binary_size,
068:                (const unsigned char **)&binary_buf, &binary_status, &ret);
```

Since the kernel is already built, there is no reason for another build step as in the online compilation version.
To summarize, in order to change the method of compilation from online to offline, the following steps are followed:

1. Read the kernel as a binary
2. Change clCreateProgramWithSource() to clCreateProgramWithBinary()
3. Get rid of clBuildProgram()

This concludes the differences between the two methods. See chapter 7 for the details on the APIs used inside the sample codes.

Calling the Kernel

Data Parallelism and Task Parallelism

As stated in the "1-3-3 Types of Parallelism" section, parallelizable code is either "Data Parallel" or "Task Parallel". In OpenCL, the difference between the two is whether the same kernel or different kernels are executed in parallel. The difference becomes obvious in terms of execution time when executed on the GPU.

At present, most GPUs contain multiple processors, but hardware such as instruction fetch and program counters are shared across the processors. For this reason, the GPUs are incapable of running different tasks in parallel.

As shown in Figure 4.2, when multiple processors perform the same task, the number of tasks equal to the number of processors can be performed at once. Figure 4.3 shows the case when multiple tasks are scheduled to be performed in parallel on the GPU. Since the processors can only process the same set of instructions across the cores, the processors scheduled to process Task B must be in idle mode until Task A is finished.

Figure 4.2: Efficient use of the GPU

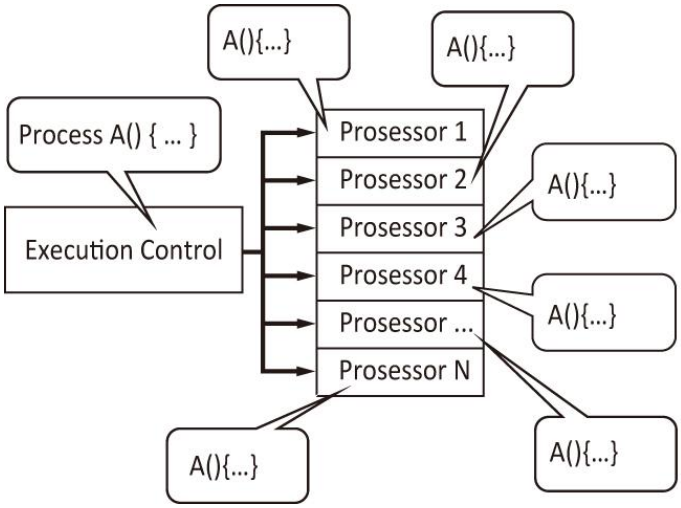
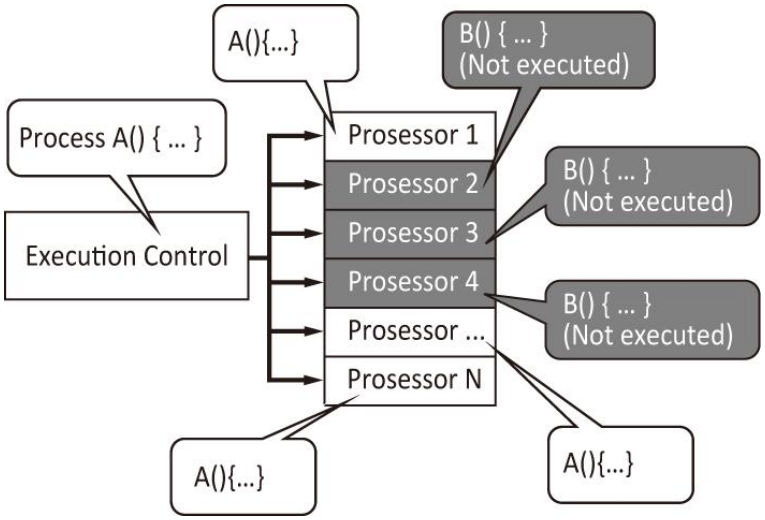


Figure 4.3: Inefficient use of the GPU

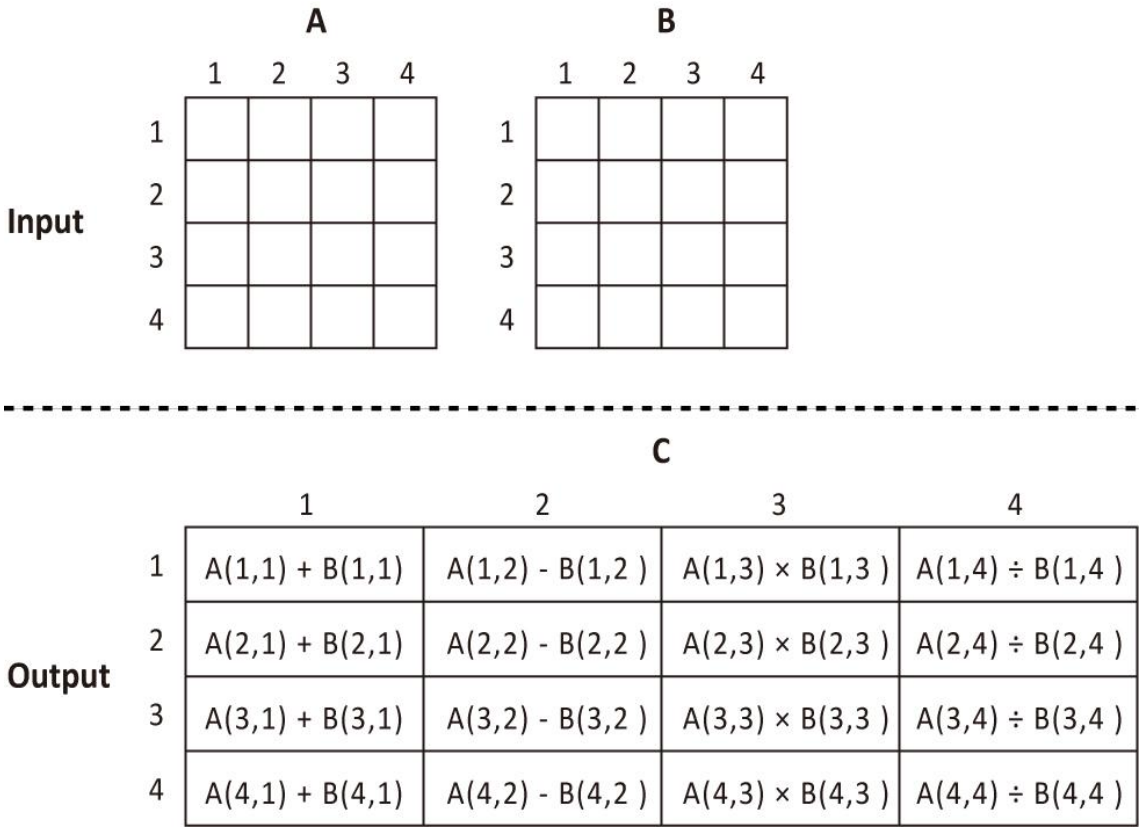


For data parallel tasks suited for a device like the GPU, OpenCL provides an API to run the same kernel across multiple processors, called clEnqueueNDRangeKernel(). When developing an application, the task type and the hardware need to be considered wisely, and use the appropriate API function.

This section will use vector-ized arithmetic operation to explain the basic method of implementations for data parallel and task parallel commands. The provided sample code is meant to illustrate the parallelization concepts.

The sample code performs the basic arithmetic operations, which are addition, subtraction, multiplication and division, between float values. The overview is shown in Figure 4.4.

Figure 4.4: Basic arithmetic operations between floats



As the figure shows, the input data consists of 2 sets of 4x4 matrices A and B. The output data is a 4x4 matrix C.

We will first show the data-parallel implementation (List 4.8, List 4.9). This program treats each row of data as one group in order to perform the computation.

List 4.8: Data parallel model - kernel dataParallel.cl

```
001: __kernel void dataParallel(__global float* A, __global float* B, __global float* C)
002: {
003:       int base = 4*get_global_id(0);
004:
005:       C[base+0] = A[base+0] + B[base+0];
006:       C[base+1] = A[base+1] - B[base+1];
007:       C[base+2] = A[base+2] * B[base+2];
008:       C[base+3] = A[base+3] / B[base+3];
009: }
```

List 4.9: Data parallel model - host dataParallel.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: #ifdef __APPLE__
005: #include <OpenCL/opencl.h>
006: #else
007: #include <CL/cl.h>
008: #endif
009:
010: #define MAX_SOURCE_SIZE (0x100000)
011:
012: int main()
013: {
014:       cl_platform_id platform_id = NULL;
015:       cl_device_id device_id = NULL;
016:       cl_context context = NULL;
017:       cl_command_queue command_queue = NULL;
018:       cl_mem Amobj = NULL;
019:       cl_mem Bmobj = NULL;
020:       cl_mem Cmobj = NULL;
021:       cl_program program = NULL;
022:       cl_kernel kernel = NULL;
023:       cl_uint ret_num_devices;
024:       cl_uint ret_num_platforms;
025:       cl_int ret;
026:
027:       int i, j;
028:       float *A;
029:       float *B;
030:       float *C;
031:
032:       A = (float *)malloc(4*4*sizeof(float));
033:       B = (float *)malloc(4*4*sizeof(float));
034:       C = (float *)malloc(4*4*sizeof(float));
035:
```

```
036:	FILE *fp;
037:	const char fileName[] = "./dataParallel.cl";
038:	size_t source_size;
039:	char *source_str;
040:
041:	/* Load kernel source file */
042:	fp = fopen(fileName, "r");
043:	if (!fp) {
044:		fprintf(stderr, "Failed to load kernel.¥n");
045:		exit(1);
046:	}
047:	source_str = (char *)malloc(MAX_SOURCE_SIZE);
048:	source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
049:	fclose(fp);
050:
051:	/* Initialize input data */
052:	for (i=0; i < 4; i++) {
053:		for (j=0; j < 4; j++) {
054:			A[i*4+j] = i*4+j+1;
055:			B[i*4+j] = j*4+i+1;
056:		}
057:	}
058:
059:	/* Get Platform/Device Information
060:	ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
061:	ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);
062:
063:	/* Create OpenCL Context */
064:	context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
065:
066:	* Create command queue */
067:	command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
068:
069:	* Create Buffer Object */
070:	Amobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
```

```
&ret);
071:       Bmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
&ret);
072:       Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
&ret);
073:
074:       /* Copy input data to the memory buffer */
075:       ret = clEnqueueWriteBuffer(command_queue, Amobj, CL_TRUE, 0, 4*4*sizeof(float),
A, 0, NULL, NULL);
076:       ret = clEnqueueWriteBuffer(command_queue, Bmobj, CL_TRUE, 0, 4*4*sizeof(float),
B, 0, NULL, NULL);
077:
078:       /* Create kernel program from source file*/
079:       program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const
size_t *)&source_size, &ret);
080:       ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
081:
082:       /* Create data parallel OpenCL kernel */
083:       kernel = clCreateKernel(program, "dataParallel", &ret);
084:
085:       /* Set OpenCL kernel arguments */
086:       ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&Amobj);
087:       ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&Bmobj);
088:       ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&Cmobj);
089:
090:       size_t global_item_size = 4;
091:       size_t local_item_size = 1;
092:
093:       /* Execute OpenCL kernel as data parallel */
094:       ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
095:       &global_item_size, &local_item_size, 0, NULL, NULL);
096:
097:       /* Transfer result to host */
098:       ret = clEnqueueReadBuffer(command_queue, Cmobj, CL_TRUE, 0, 4*4*sizeof(float),
C, 0, NULL, NULL);
099:
```

```
100:      /* Display Results */
101:      for (i=0; i < 4; i++) {
102:                for (j=0; j < 4; j++) {
103:                          printf("%7.2f ", C[i*4+j]);
104:                }
105:                printf("¥n");
106:      }
107:
108:
109:      /* Finalization */
110:      ret = clFlush(command_queue);
111:      ret = clFinish(command_queue);
112:      ret = clReleaseKernel(kernel);
113:      ret = clReleaseProgram(program);
114:      ret = clReleaseMemObject(Amobj);
115:      ret = clReleaseMemObject(Bmobj);
116:      ret = clReleaseMemObject(Cmobj);
117:      ret = clReleaseCommandQueue(command_queue);
118:      ret = clReleaseContext(context);
119:
120:      free(source_str);
121:
122:      free(A);
123:      free(B);
124:      free(C);
125:
126:      return 0;
127: }
```

Next, we will show the task parallel version of the same thing (List 4.10, List 4.11). In this sample, the tasks are grouped according to the type of arithmetic operation being performed.

List 4.10: Task parallel model - kernel taskParallel.cl

```
001: __kernel void taskParallelAdd(__global float* A, __global float* B, __global float* C)
002: {
003:      int base = 0;
```

```
004:
005:      C[base+0] = A[base+0] + B[base+0];
006:      C[base+4] = A[base+4] + B[base+4];
007:      C[base+8] = A[base+8] + B[base+8];
008:      C[base+12] = A[base+12] + B[base+12];
009: }
010:
011: __kernel void taskParallelSub(__global float* A, __global float* B, __global float* C)
012: {
013:      int base = 1;
014:
015:      C[base+0] = A[base+0] - B[base+0];
016:      C[base+4] = A[base+4] - B[base+4];
017:      C[base+8] = A[base+8] - B[base+8];
018:      C[base+12] = A[base+12] - B[base+12];
019: }
020:
021: __kernel void taskParallelMul(__global float* A, __global float* B, __global float* C)
022: {
023:      int base = 2;
024:
025:      C[base+0] = A[base+0] * B[base+0];
026:      C[base+4] = A[base+4] * B[base+4];
027:      C[base+8] = A[base+8] * B[base+8];
028:      C[base+12] = A[base+12] * B[base+12];
029: }
030:
031: __kernel void taskParallelDiv(__global float* A, __global float* B, __global float* C)
032: {
033:      int base = 3;
034:
035:      C[base+0] = A[base+0] / B[base+0];
036:      C[base+4] = A[base+4] / B[base+4];
037:      C[base+8] = A[base+8] / B[base+8];
038:      C[base+12] = A[base+12] / B[base+12];
039: }
```

List 4.11: Task parallel model - host taskParallel.c

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: #ifdef __APPLE__
005: #include <OpenCL/opencl.h>
006: #else
007: #include <CL/cl.h>
008: #endif
009:
010: #define MAX_SOURCE_SIZE (0x100000)
011:
012: int main()
013: {
014:       cl_platform_id platform_id = NULL;
015:       cl_device_id device_id = NULL;
016:       cl_context context = NULL;
017:       cl_command_queue command_queue = NULL;
018:       cl_mem Amobj = NULL;
019:       cl_mem Bmobj = NULL;
020:       cl_mem Cmobj = NULL;
021:       cl_program program = NULL;
022:       cl_kernel kernel[4] = {NULL, NULL, NULL, NULL};
023:       cl_uint ret_num_devices;
024:       cl_uint ret_num_platforms;
025:       cl_int ret;
026:
027:       int i, j;
028:       float* A;
029:       float* B;
030:       float* C;
031:
032:       A = (float*)malloc(4*4*sizeof(float));
033:       B = (float*)malloc(4*4*sizeof(float));
```

```
034:	C = (float*)malloc(4*4*sizeof(float));
035:
036:
037:	FILE *fp;
038:	const char fileName[] = "./taskParallel.cl";
039:	size_t source_size;
040:	char *source_str;
041:
042:	/* Load kernel source file */
043:	fp = fopen(fileName, "rb");
044:	if (!fp) {
045:		fprintf(stderr, "Failed to load kernel.¥n");
046:		exit(1);
047:	}
048:	source_str = (char *)malloc(MAX_SOURCE_SIZE);
049:	source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
050:	fclose(fp);
051:
052:	/* Initialize input data */
053:	for (i=0; i < 4; i++) {
054:		for (j=0; j < 4; j++) {
055:			A[i*4+j] = i*4+j+1;
056:			B[i*4+j] = j*4+i+1;
057:		}
058:	}
059:
060:	/* Get platform/device information */
061:	ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
062:	ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);
063:
064:	/* Create OpenCL Context */
065:	context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
066:
067:	/* Create command queue */
068:	command_queue = clCreateCommandQueue(context, device_id,
```

```
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &ret);
069:
070:        /* Create buffer object */
071:        Amobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
&ret);
072:        Bmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
&ret);
073:        Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
&ret);
074:
075:        /* Copy input data to memory buffer */
076:        ret = clEnqueueWriteBuffer(command_queue, Amobj, CL_TRUE, 0, 4*4*sizeof(float),
A, 0, NULL, NULL);
077:        ret = clEnqueueWriteBuffer(command_queue, Bmobj, CL_TRUE, 0, 4*4*sizeof(float),
B, 0, NULL, NULL);
078:
079:        /* Create kernel from source */
080:        program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const
size_t *)&source_size, &ret);
081:        ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
082:
083:        /* Create task parallel OpenCL kernel */
084:        kernel[0] = clCreateKernel(program, "taskParallelAdd", &ret);
085:        kernel[1] = clCreateKernel(program, "taskParallelSub", &ret);
086:        kernel[2] = clCreateKernel(program, "taskParallelMul", &ret);
087:        kernel[3] = clCreateKernel(program, "taskParallelDiv", &ret);
088:
089:        /* Set OpenCL kernel arguments */
090:        for (i=0; i < 4; i++) {
091:                ret = clSetKernelArg(kernel[i], 0, sizeof(cl_mem), (void *)&Amobj);
092:                ret = clSetKernelArg(kernel[i], 1, sizeof(cl_mem), (void *)&Bmobj);
093:                ret = clSetKernelArg(kernel[i], 2, sizeof(cl_mem), (void *)&Cmobj);
094:        }
095:
096:        /* Execute OpenCL kernel as task parallel */
097:        for (i=0; i < 4; i++) {
```

```
098:                    ret = clEnqueueTask(command_queue, kernel[i], 0, NULL, NULL);
099:        }
100:
101:        /* Copy result to host */
102:        ret = clEnqueueReadBuffer(command_queue, Cmobj, CL_TRUE, 0, 4*4*sizeof(float),
C, 0, NULL, NULL);
103:
104:         /* Display result */
105:        for (i=0; i < 4; i++) {
106:                    for (j=0; j < 4; j++) {
107:                                printf("%7.2f ", C[i*4+j]);
108:                    }
109:                    printf("¥n");
110:        }
111:
112:        /* Finalization */
113:        ret = clFlush(command_queue);
114:        ret = clFinish(command_queue);
115:        ret = clReleaseKernel(kernel[0]);
116:        ret = clReleaseKernel(kernel[1]);
117:        ret = clReleaseKernel(kernel[2]);
118:        ret = clReleaseKernel(kernel[3]);
119:        ret = clReleaseProgram(program);
120:        ret = clReleaseMemObject(Amobj);
121:        ret = clReleaseMemObject(Bmobj);
122:        ret = clReleaseMemObject(Cmobj);
123:        ret = clReleaseCommandQueue(command_queue);
124:        ret = clReleaseContext(context);
125:
126:        free(source_str);
127:
128:        free(A);
129:        free(B);
130:        free(C);
131:
132:        return 0;
```

```
133: }
```

As you can see, the source codes are very similar. The only differences are in the kernels themselves, and the way to execute these kernels. In the data parallel model, the 4 arithmetic operations are grouped as one set of commands in a kernel, while in the task parallel model, 4 different kernels are implemented for each type of arithmetic operation.

At a glance, it may seem that since the task parallel model requires more code, that it also must perform more operations. However, regardless of which model is used for this problem, the number of operations being performed by the device is actually the same. Despite this fact, some problems are easier, and performance can vary by choosing one over the other, so the parallelization model must be considered wisely in the planning stage of the application.

We will now walkthrough the source code for the data parallel model.

```
001: __kernel void dataParallel(__global float * A, __global float * B, __global float * C)
```

When the data parallel task is queued, work-items are created. Each of these work-items executes the same kernel in parallel.

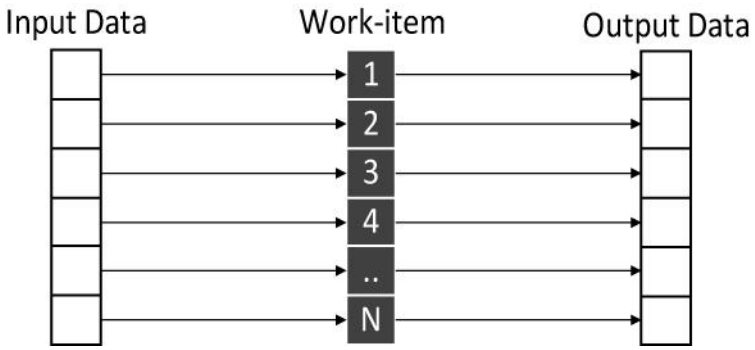
```
003:      int base = 4*get_global_id(0);
```

The get_global_id(0) gets the global work-item ID, which is used to decide the data to process, so that each work-items can process different sets of data in parallel. In general, data parallel processing is done using the following steps.

1. Get work-item ID
2. Process the subset of data corresponding to the work-item ID

A block diagram of the process is shown in Figure 4.5.

Figure 4.5: Block diagram of the data-parallel model in relation to work-items



In this example, the global work-item is multiplied by 4 and stored in the variable "base". This value is used to decide which element of the array A and B gets processed.

```
005:    C[base+0] = A[base+0] + B[base+0];
006:    C[base+1] = A[base+1] - B[base+1];
007:    C[base+2] = A[base+2] * B[base+2];
008:    C[base+3] = A[base+3] / B[base+3];
```

Since each work-item have different IDs, the variable "base" also have a different value for each work-item, which keeps the work-items from processing the same data. In this way, large amount of data can be processed concurrently.

We have discussed that numerous work items get created, but we have not touched upon how to decide the number of work-items to create. This is done in the following code segment from the host code.

```
090:    size_t global_item_size = 4;
091:    size_t local_item_size = 1;
092:
093:    /* Execute OpenCL kernel as data parallel */
094:    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
095:            &global_item_size, &local_item_size, 0, NULL, NULL);
```

The clEnqueueNDRangeKernel() is an OpenCL API command used to queue data parallel tasks. The 5th and 6th arguments determine the work-item size. In this case, the global_item_size is set to 4, and the local_item_size is set to 1. The overall steps are summarized as follows.

1. Create work-items on the host
2. Process data corresponding to the global work item ID on the kernel

We will now walkthrough the source code for the task parallel model. In this model, different kernels are allowed to be executed in parallel. Note that different kernels are implemented for each of the 4 arithmetic operations.

```
096:    /* Execute OpenCL kernel as task parallel */
097:    for (i=0; i < 4; i++) {
098:            ret = clEnqueueTask(command_queue, kernel[i], 0, NULL, NULL);
099:     }
```

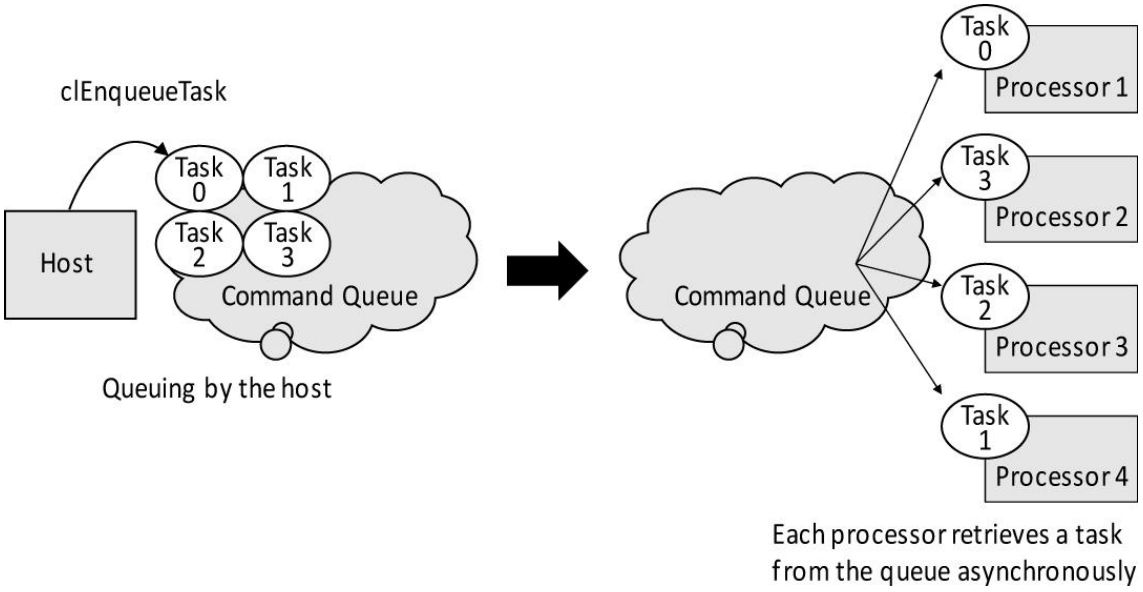
The above code segment queues the 4 kernels.

In OpenCL, in order to execute a task parallel process, the out-of-order mode must be enabled when the command queue is created. Using this mode, the queued task does not wait until the previous task is finished if there are idle compute units available that can be executing that task.

```
067: /* Create command queue */
068: command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &ret);
```

The block diagram of the nature of command queues and parallel execution are shown in Figure 4.6.

Figure 4.6: Command queues and parallel execution



The clEnqueueTask() is used as an example in the above figure, but a similar parallel processing could take place for other combinations of enqueue-functions, such as clEnqueueNDRangeKernel(), clEnqueueReadBuffer(), and clEnqueueWriteBuffer(). For example, since PCI Express supports simultaneous bi-directional memory transfers, queuing the clEnqueueReadBuffer() and clEnqueueWriteBuffer() can execute read and write commands simultaneously, provided that the commands are being performed by different processors. In the above diagram, we can expect the 4 tasks to be executed in parallel, since they are being queued in a command queue that has out-of-execution enabled.

Work Group

The last section discussed the concept of work-items. This section will introduce the concept of work-groups.

Work-items are grouped together into work-groups. A work-group must consist of at least 1 work-item, and the maximum is dependent on the platform. The work-items within a work-group can synchronize, as well as share local memory with each other.

In order to implement a data parallel kernel, the number of work-groups must be specified in addition to the number of work-items. This is why 2 different parameters had to be sent to the clEnqueueNDRangeKernel() function.

```
090:    size_t global_item_size = 4;
```

```
091:    size_t local_item_size = 1;
```

The above code means that each work-group is made up of 1 work-item, and that there are 4 work-groups to be processed.

The number of work-items per work-group is consistent throughout every work-group. If the number of work-items cannot be divided evenly among the work-groups, clEnqueueNDRangeKernel() fails, returning the error value CL_INVALID_WORK_GROUP_SIZE.

The code List 4.9 only used the global work-item ID to process the kernel, but it is also possible to retrieve the local work-item ID corresponding to the work-group ID. The relationship between the global work-item ID, local work-item ID, and the work-group ID are shown below in Figure 4.7. The function used to retrieve these ID's from within the kernel are shown in Table 4.1.

Figure 4.7: Work-group ID and Work-item ID

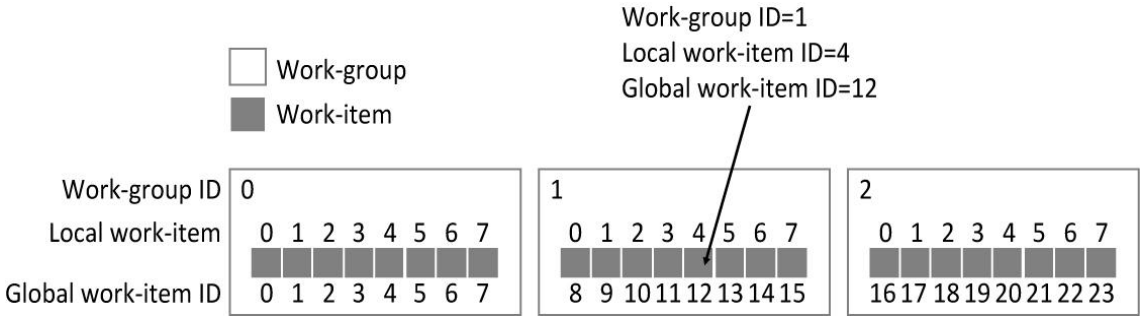
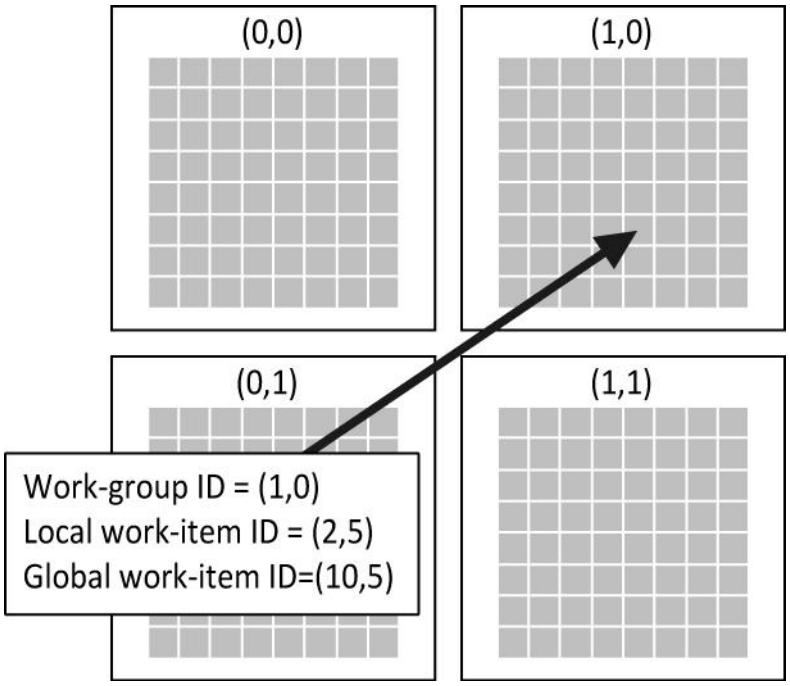


Table 4.1: Functions used to retrieve the ID's

Function	Retrieved value
get_group_id	Work-group ID
get_global_id	Global work-item ID
get_local_id	Local work-item ID

Since 2-D images or 3-D spaces are commonly processed, the work-items and work-groups can be specified in 2 or 3 dimensions. Figure 4.8 shows an example where the work-group and the work-item are defined in 2-D.

Figure 4.8: Work-group and work-item defined in 2-D



Since the work-group and the work-items can have up to 3 dimensions, the ID's that are used to index them also have 3 dimensions. The get_group_id(), get_global_id(), get_local_id() can each take an argument between 0 and 2, each corresponding to the dimension. The ID's for the work-items in Figure 4.8 are shown below in Table 4.2.

Table 4.2: The ID's of the work-item in Figure 4.8

Call	Retrieved ID
get_group_id(0)	1
get_group_id(1)	0
get_global_id(0)	10
get_global_id(1)	5
get_local_id(0)	2
get_local_id(1)	5

Note that the index space dimension and the number of work-items per work-group can vary depending on the device. The maximum index space dimension can be obtained using the clGetDeviceInfo() function to get the value of CL_DEVICE_WORK_ITEM_DIMENSIONS, and the maximum number of work-items per work-group can be obtained by getting the value of CL_DEVICE_IMAGE_SUPPORT. The data-type of CL_DEVICE_MAX_WORK_ITEM is cl_uint, and for CL_DEVICE_IMAGE_SUPPORT, it is an array of type size_t.

Also, as of this writing (12/2009), the OpenCL implementation for the CPU on Mac OS X only allows 1 work-item per work-group.

Task Parallelism and Event Object

The tasks placed in the command-queue are executed in parallel, but in cases where different tasks have data dependencies, they need to be executed sequentially. In OpenCL, the execution order can be set using an event object.

An event object contains information about the execution status of queued commands. This object is returned on all commands that start with "clEnqueue". In order to make sure task A executes before task B, the event object returned when task A is queued can be one of the inputs to task B, which keeps task B from executing until task A is completed.

For example, the function prototype for clPEnqueueTask() is shown below.

```
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)
```

The 4th argument is a list of events to be processed before this task can be run, and the 3rd argument is the number of events on the list. The 5th parameter is the event object returned by this task when it is placed in the queue.

List 4.12 shows an example of how the event objects can be used. In this example, kernel_A, kernel_B, kernel_C, kernel_D can all be executed in any order, but these must be completed before kernel_E is executed.

List 4.12: Event object usage example

```
clEnqueue events[4];
clEnqueueTask(command_queue, kernel_A, 0, NULL, &events[0]);
clEnqueueTask(command_queue, kernel_B, 0, NULL, &events[1]);
clEnqueueTask(command_queue, kernel_C, 0, NULL, &events[2]);
clEnqueueTask(command_queue, kernel_D, 0, NULL, &events[3]);
```

```
clEnqueueTask(command_queue, kernel_E, 4, events, NULL);
```

---COLUMN: Relationship between execution order and parallel execution---

In OpenCL, the tasks placed in the command queue are executed regardless of the order in which it is placed in the queue. In the specification sheet for parallel processors, there is usually a section on "order". When working on parallel processing algorithms, the concept of "order" and "parallel" need to be fully understood.

First, we will discuss the concept of "order". As an example, let's assume the following 4 tasks are performed sequentially in the order shown below.

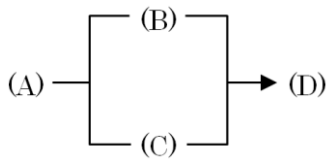
(A) → (B) → (C) → (D)

Now, we will switch the order in which task B and C are performed.

(A) → (C) → (B) → (D)

If tasks B and C are not dependent on each other, the two sets of tasks will result in the same output. If they are not, the two sets of tasks will not yield the correct result. This problem of whether a certain set of tasks can be performed in different order is a process-dependent problem.

Parallel processing is a type of optimization that deals with changing the order of tasks. For example, let's assume tasks B and C are processed in parallel.



In this case, task B may finish before task C, but task D waits for the results of both tasks B and C. This processing is only allowed in the case where the tasks B and C do not depend on each other. Thus, if task C must be performed after task B, then the 2 tasks cannot be processed in parallel.

On the other hand, it may make more sense to implement all the tasks in a single thread. Also, if the tasks are executed on a single-core processor, all the tasks must be performed sequentially. These need to be considered when implementing an algorithm.

In essence, two problems must be solved. One is whether some tasks can be executed out of order, and the other is whether to process the tasks in parallel.

• Tasks must be executed in a specific order = Cannot be parallelized
• Tasks can be executed out of order = Can be parallelized

Specifications are written to be a general reference of the capabilities, and do not deal with the actual implementation. Decision to execute certain tasks in parallel is thus not discussed inside the specification. Instead, it contains information such as how certain tasks are guaranteed to be processed in order.

Explanations on "ordering" is often times difficult to follow (Example: PowerPC's Storage Access Ordering), but often times, treating "order" as the basis when implementing parallel algorithms can clear up existing bottlenecks in the program.

For example, OpenMP's "parallel" construct specifies the code segment to be executed in parallel, but placing this construct in a single-core processor will not process the code segment in parallel. In this case, one may wonder if it meets OpenMP's specification. If you think about the "parallel" construct as something that tells the compiler and the processors that the loops can be executed out-of-order, it clears up the definition, since it means that the ordering can be changed or the tasks can be run parallel, but that they do not have to be.

Advanced OpenCL

Chapter 4 discussed the basic concepts required to write a simple OpenCL program. This chapter will go further in depth, allowing for a more a flexible and optimized OpenCL program.

OpenCL C

OpenCL C language is basically standard C (C99) with some extensions and restrictions. This language is used to program the kernel. Aside from these extensions and restrictions, the language can be treated the same as C. List 5.1 shows a simple function written in OpenCL C.

List 5.1: OpenCL C Sample Code

```
001: int sum(int *a, int n) { /* Sum every component in array "a" */
002:       int sum = 0;
003:       for (int i=0; i < n; i++) {
004:                  sum += a[i];
005:       }
006:       return sum;
007: }
```

Restrictions

Below lists the restricted portion from the C language in OpenCL C.

• The pointer passed as an argument to a kernel function must be of type __global, __constant, or __local.
• Pointer to a pointer cannot be passed as an argument to a kernel function.
• Bit-fields are not supported.
• Variable length arrays and structures with flexible (or unsized) arrays are not supported.
• Variadic macros and functions are not supported.
• C99 standard headers cannot be included
• The extern, static, auto and register storage-class specifiers are not supported.
• Predefined identifiers are not supported.
• Recursion is not supported.
• The function using the __kernel qualifier can only have return type void in the source code.
• Writes to a pointer of type char, uchar, char2, uchar2, short, ushort, and half are not supported.

• Support for double precision floating-point is currently an optional extension. It may or may not be implemented.

Address Space Qualifiers

OpenCL supports 4 memory types: global, constant, local, and private. In OpenCL C, these address space qualifiers are used to specify which memory on the device to use. The memory type corresponding to each address space qualifiers are shown in Table 5.1.

Table 5.1: Address Space Qualifiers and their corresponding memory

Qualifier	Corresponding memory
__global (or global)	Global memory
__local (or local)	Local memory
__private (or private)	Private memory
__constant (or constant)	Constant memory

The "__" prefix is not required before the qualifiers, but we will continue to use the prefix in this text for consistency. If the qualifier is not specified, the variable gets allocated to "__private", which is the default qualifier. A few example variable declarations are given in List 5.2.

List 5.2: Variables declared with address space qualifiers

```
001:    __global int global_data[128];  // 128 integers allocated on global memory
002:    __local float *lf;  // pointer placed on the private memory, which points to a single-precision float located on the local memory
003:    __global char * __local lgc[8];  // 8 pointers stored on the local memory that points to a char located on the global memory
```

The pointers can be used to read address in the same way as in standard C. Similarly, with the exception for "__constant" pointers, the pointers can be used to write to an address space as in standard C.

A type-cast using address space qualifiers is undefined. Some implementation allows access to a different memory space via pointer casting, but this is not the case for all implementations.

Built-in Function

OpenCL contains several built-in functions for scalar and vector operations. These can be used without having to include a header file or linking to a library. A few examples are shown below.

• **Math Functions**

Extended version of math.h in C. Allows usage of functions such as sin(), log(), exp().

• **Geometric Functions**

Includes functions such as length(), dot().

• **Work-Item Functions**

Functions to query information on groups and work-items. Get_work_dim(), Get_local_size() are some examples.

Refer to chapter 7 for a full listing of built-in functions.

List 5.3 below shows an example of a kernel that uses the sin() function. Note that header files do not have to be included.

List 5.3: Kernel that uses sin()

```
001:  __kernel void runtest(__global float *out, __global float *in) {
002:      for (int i=0; i<16; i++) {
003:                  out[i] = sin(in[i]/16.0f);
004       }
005:  }
```

Many built-in functions are overloaded. A function is said to be overloaded if the same function name can be used to call different function definitions based on the arguments passed into that function. For example, the sin() function is overloaded such that the precision of the returned value depends on whether a float or a double value is passed in as an argument, since different functions exist for each of the passed in data-types. In OpenCL, most math functions are overloaded, reducing the need for functions such as sinf() in standard C. An example of calls to overloaded functions is shown in List 5.4 below.

List 5.4: Calls to overloaded functions

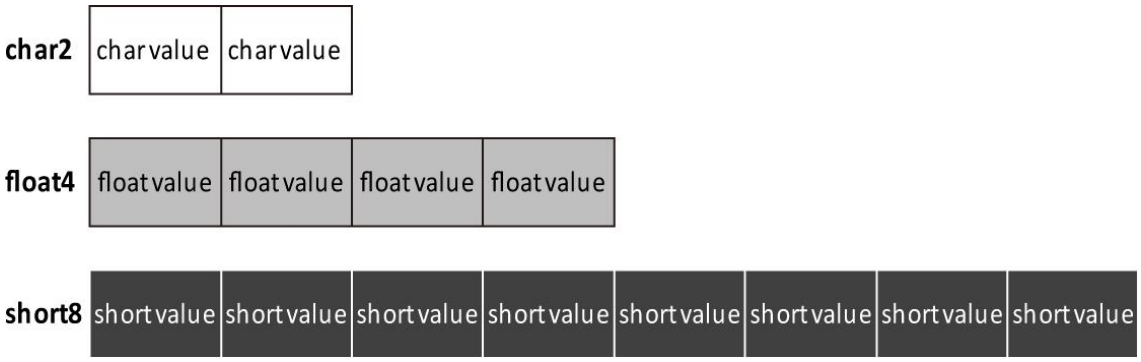
```
001: __kernel void runtest(__global float *out, __global float *in_single, __global double
*in_double) {
002:      for (int i=0; i < 16; i++) {
003:      float sf = sin(in_single[i]); /* The float version is called */
004:      double df = sin(in_double[i]); /* The double version is called */
```

```
005:      }
006: }
```

Vector Data

OpenCL defines vector data-types, which is basically a struct consisting of many components of the same data-type. Figure 5.1 shows some example of vector data-types.

Figure 5.1: Vector data-type examples



Each component in the vector is referred to as a scalar-type. A group of these scalar-type values defines a vector data-type.

Recent CPUs contain compute units called SIMD, which allows numerous data to be processed using one instruction. Usage of these SIMD units can accelerate some processes by many folds. The use of vector data-types in OpenCL can aid in efficient usage of these SIMD units.

In OpenCL C, adding a number after the scalar type creates a vector type consisting of those scalar types, such as in "int4". Declaring a variable using a vector type creates a variable that can contain the specified number of scalar types. List 5.5 shows examples of declaring vector-type variables.

List 5.5: Vector-type variable declaration

```
int i4;     // Variable made up of 4 int values
char2 c2;             // Variable made up of 2 char values
float8 f8;// Variable made up of 8 float values
```

OpenCL defines scalar types char, uchar, short, ushort, int, uint, long, ulong, float, double, and vector-types with size 2, 4, 8, and 16. Vector literals are used to place values in these variables.

A vector literal is written as a parenthesized vector type followed by a parenthesized set of expressions. List 5.6 shows some example usage of vector literals.

List 5.6: Vector Literal

```
(int4) (1,2,3,4);
(float8) (1.1f, 1.2f, 1.3f, 1.4f, 1.5f, 1.6f, 1.7f, 1.8f);
(int8) ((int2)(1,2), (int2)(3,4), (int4)(5,6,7,8));
```

The number of values set in the vector literals must equal the size of the vector.

The built-in math functions are overloaded such that an component-by-component processing is allowed for vector-types. Functions such as sin() and log() performs these operations on each component of the vector-type.

```
float4 g4 = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float4 f4 = sin(g4);
```

The above code performs the operation shown below.

```
float4 g4 = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float4 f4 = (float4)(sin(1.0f), sin(2.0f), sin(3.0f), sin(4.0f));
```

The vector component can be accessed by adding an extension to the variable. This can be done by adding .xyzw, accessing by number, or by odd/even/hi/lo.

• ".xyzw" accesses indices 0 through 3. It cannot access higher indices.
• The vector can be accessed via a number index using ".s" followed by a number. This number is in hex, allowing access to up to 16 elements by using a~f or A~F for indices above 9.
• ".odd" extracts the odd indices, and ".even" extracts the even indices. ".hi" extracts the upper half of the vector, and ".lo" extracts the lower half of the vector.

List 5.7 shows a few examples of component accesses.

List 5.7: Vector component access

```
int4 x = (int4)(1, 2, 3, 4);
int4 a = x.wzyx; /* a = (4, 3, 2, 1) */
```

```
int2 b = x.xx; /* b = (1, 1) */
int8 c = x.s01233210; /* c = (1, 2, 3, 4, 4, 3, 2, 1) */
int2 d = x.odd; /* d = (2, 4) */
int2 e = x.hi; /* e = (3, 4) */
```

We will now show an example of changing the order of data using the introduced concepts. List 5.8 shows a kernel code that reverses the order of an array consisting of 16 components.

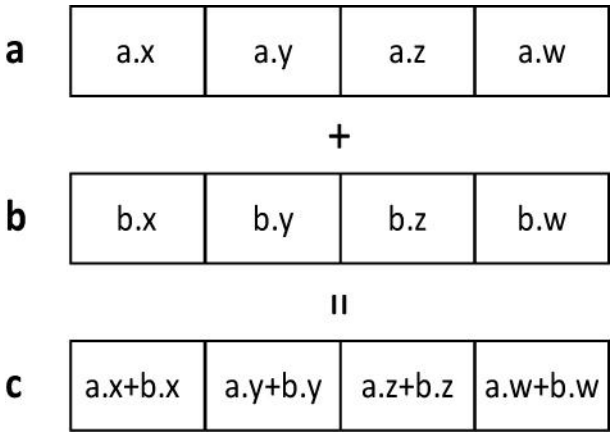
List 5.8: Reverse the order of a 16-element array

```
001: __kernel void runtest(__global float *out, __global float *in) {
002:       __global float4 *in4 = (__global float4*)in;
003:       __global float4 *out4 = (__global float4*)out;
004:       for (int i=0; i < 4; i++) {
005:                  out4[3-i] = in4[i].s3210;
006:       }
007: }
```

• 002~003: Type cast allow access as a vector type
• 004: Each loop processes 4 elements. The loop is run 4 times to process 16 elements.
• 005: Vector component access by index number.

Some operators are overloaded to support arithmetic operation on vector types. For these operations, arithmetic operation is performed for each component in the vector. An example is shown in Figure 5.2.

Figure 5.2: Addition of 2 vector-types



In general, the vector operation would be the same for each component, with the exception to comparison and ternary selection operations. Logic operations such as "&&" and "||" are undefined. The result of comparison and selection operations are shown in the sections to follow.

Comparison Operations

When 2 scalar-types are compared, an integer value of either 0 or 1 is returned. When 2 vector-types are compared, a vector type (not necessarily the same as the vector-types being compared) is returned, with each vector component having a TRUE or FALSE value. For example, a comparison of two-float4 types returns an int4 type. The operand types and the returned vector-type after a comparison are shown in Table 5.2 below.

Table 5.2: Resulting vector-type after a comparison

Operand type	Type after comparison
char, uchar	char
short, ushort	short
int, uint, float	int
long, ulong, double	long

Also, if the comparison is FALSE, a 0 (all bits are 0) is returned like usual, but if it is TRUE, a -1 (all bits are 1) is returned.

```
int4 tf = (float4)(1.0f, 1.0f, 1.0f, 1.0f) == (float4)(0.0f, 1.0f, 2.0f, 3.0f);
// tf = (int4)(0, -1, 0, 0)
```

A 0/-1 is returned instead of 0/1, since it is more convenient when using SIMD units. Since SIMD operation performs the same operation on different data in parallel, branch instructions should be avoided when possible. For example, in the code shown in List 5.9, the branch must be performed for each vector component serially, which would not use the SIMD unit effectively.

List 5.9: Branch process

```
// "out" gets the smaller of "in0" and "in1" for each component in the vector
int a = in0[i], b = in1[i];
if (a<b)   out[i] = a;
else       out[i] = b;
```

Since a "-1" is returned when the condition is TRUE, the code in List 5.9 can be replaced by the code in 5.10 that does not use a branch.

List 5.10: List 5.9 rewritten without the branch

```
// "out" gets the smaller of "in0" and "in1" for each component in the vector
int a = in0[i], b = in1[i];
int cmp = a < b;   // If TRUE, cmp=0xffffffff.   If FALSE, cmp=0x00000000;
out[i] = (a & cmp) | (b & ~cmp);   // a when TRUE and b when FALSE
```

In this case, the SIMD unit is used effectively, since each processing elements are running the same set of instructions.

To summarize, branch instruction should be taken out to use SIMD units, and to do this, it is much more convenient to have the TRUE value to be -1 instead of 1.

On a side note, OpenCL C language has a built-in function bitselect(), which can rewrite the line

```
out[i] = (a & cmp) | (b & ~cmp)   // a when TRUE and b when FALSE
```

with the line

```
out[i] = bitselect(cmp, b, a);
```

Ternary Selection Operations

The difference in comparison operation brings about a difference in ternary selection operations as well. In a selection operation, if the statement to the left of "?" evaluates to be FALSE, then the value between the "?" and ":" is selected, and if it evaluates to be TRUE, then the value to the right of ":" is selected. Using this operation, the code from List 5.10 can be rewritten as the code shown in List 5.11.

List 5.11: Ternary Selection Operator Usage Example

```
// "out" gets the smaller of "in0" and "in1" for each component in the vector
int a = in0[i], b = in1[i];
out[i] = a<b ? a : b; // a when TRUE and b when FALSE
```

--- COLUMN: Should you use a vector-type? ---

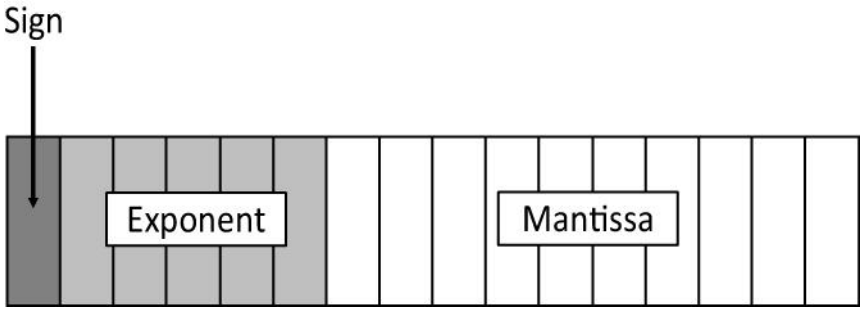
The OpenCL C language defines vector type, but it does not specify how its operations are implemented. Since not all processors contain SIMD units, (for example, NVIDIA GPU), and when vector operations are performed on these processors, the operations will be performed sequentially for each scalars. Also, large vector-types such as double16 may fill up the hardware register, resulting in slower execution. In these cases, the operation will not benefit from using a vector type. In fact, the compiler should take care of deciding on the vector length to use the SIMD unit for optimal performance.

However, compilers are still not perfect, and auto-vectorization is still under development. Thus, the optimization to effectively use SIMD units is placed on the hands of the programmers.

Half

OpenCL C language defines a 16-bit floating point type referred to as "half". One bit is used for sign, 5 bits are used for exponent, and the remaining 10 bits are used as the mantissa (also known as significant or coefficient). Figure 5.3 shows the bit layout of a half-type.

Figure 5.3: Bit layout of "half"



The "half" type is not as well known as 32-bit floats or 64-bit doubles, but this type is defined as a standard in IEEE 754.

OpenCL variable types

Some defined types in OpenCL C may differ, and some types may not be defined in the standard C language. A list of types defined in OpenCL C as well as their bit widths are shown in Table 5.3 below.

Table 5.3: OpenCL variable types

Data types	Bit width	Remarks
bool	Undefined	
char	8	
unsigned char, uchar	8	
short	16	
unsigned short, ushort	16	
int	32	
unsigned int, uint	32	
long	64	
unsigned long, ulong	64	
float	32	
half	16	
size_t	*	stddef.h not required
intptr_t	*	stddef.h not required
uintptr_t	*	stddef.h not required
void	-	

Rounding of floats

When a float is casted to integer, or when a double is casted to a float, rounding must take place from the lack of available bits. Table 5.4 shows a listing of situations when rounding occurs, as well what how the numbers are rounded in each of the situations.

Table 5.4: OpenCL C Rounding

Operation	Rounding Method
Floating point arithmetic	Round to nearest even by default. Others may be set as an option. The default may not be round to nearest even for Embedded Profiles
Builtin functions	Round to nearest even only
Cast from float to int	Round towards zero only
Cast from int to float	Same as for floating point arithmetic
Cast from float to fixed point	Same as for floating point arithmetic

Type casting can be performed in the same way as in standard C. Explicit conversion can also be

performed, which allows for an option of how the number gets rounded.

The explicit conversion can be done as follows.

```
convert_<type>[_<rounding>][_sat]
```

<type> is the variable type to be converted to. <rounding> sets the rounding mode. "_sat" can be used at the end to saturate the value when the value is above the max value or below the minimum value. <rounding> gets one of the rounding mode shown in Table 5.5. If this mode is not set, than the rounding method will be the same as for type casting.

Table 5.5: Explicit specification of the rounding mode

Rounding mode	Rounding toward
rte	Nearest even
rtz	0
rtp	+∞
rtn	-∞

List 5.13 shows an example where floats are explicitly converted to integers using different rounding modes.

List 5.13: Rounding example

```
001: __kernel void round_xyzw(__global int *out, __global float *in)
002: {
003:       out->x = convert_int_rte(in->x);    // Round to nearest even
004:       out->y = convert_int_rtz(in->y);    // Round toward zero
005:       out->z = convert_int_rtn(in->z);    // Round toward -∞
006:       out->w = convert_int_rtp(in->w);    // Round toward  ∞
007: }
```

Bit Reinterpreting

In OpenCL the union() function can be used to access the bits of a variable. List 5.14 shows the case where it is used to look at the bit value of a float variable.

List 5.14: Get bit values using union

```
001: // Get bit values of a float
002: int float_int(float a) {
003:       union { int i, float f; } u;
004:       u.f = a;
005:       return u.i;
006: }
```

The action of the above code using standard C is not defined, but when using OpenCL C, the bits of the float can be reinterpreted as an integer. OpenCL includes several built-in functions that make this reinterpretation easier. The function has the name "as_<type>", and is used to reinterpret a variable as another type without changing the bits.

List 5.15 shows an example where the "as_int()" function is called with a 32-bit float as an argument.

List 5.15: Reinterpretation using as_int()

```
001: // Get the bit pattern of a float
002: int float_int(float a) {
003:       return as_int(a);
004:}
005:
006: // 0x3f800000（=1.0f）
007: int float1_int(void) {
008:       return as_int(1.0f);
009: }
```

Reinterpretation cannot be performed, for example, between float and short, where the number of bits is different. Reinterpretation between short4 and int2, where the total number of bits is the same, is undefined and the answer may vary depending on the OpenCL implementation.

Local Memory

We have mentioned the OpenCL memory hierarchy from time to time so far in this text. OpenCL memory and pointer seems complicated when compared to the standard C language. This section will discuss the reason for the seemingly-complex memory hierarchy, as well as one of the memory types in the hierarchy, called the local memory.

There are two main types of memory. One is called SRAM (Static RAM) and the other is called DRAM (Dynamic RAM). SRAM can be accessed quickly, but cannot be used throughout due to cost and the complex circuitry. DRAM on the other hand is cheap, but cannot be accessed quickly as in the case for SRAM. In most computers, the frequently-used data uses the SRAM, while less frequently-used data is kept in the DRAM. In CPU, the last accessed memory content is kept in cache located in the memory space near the CPU. The cache is the SRAM in this case.

This cache memory may not be implemented for processors such as GPU and Cell/B.E., where numerous compute units are available. This is mainly due to the fact that caches were not of utmost necessity for 3D graphics, for which the processor was designed for, and that cache hardware adds complexity to the hardware to maintain coherency.

Processors that do not have cache hardware usually contain a scratch-pad memory in order to accelerate memory access. Some examples of scratch-pad memory include shared memory on NVIDIA's GPU, and local storage on Cell/B.E.

The advantage of using these scratch-pad memories is that it keeps the hardware simple since the coherency is maintained via software rather than on the cache hardware. Also, memory space can be made smaller than cache memory, since the memory content can be altered as needed via the software. One disadvantage is that everything has to be managed by the software.

OpenCL calls these scratch-pad memory "local memory" [16]. Local memory can be used by inserting the "__local" qualifier before a variable declaration. List 5.16 shows some examples of declaring variables to be stored on the local memory. Local memory is allocated per work-group. Work-items within a work-group can use the same local memory. Local memory that belongs to another work-group may not be accessed.

List 5.16: Declaring variables on the local memory

```
__local int lvar;               // Declare lvar on the local memory
__local int larr[128];          // Declare array larr on the local memory
__local int *ptr;     // Declare a pointer that points to an address on the local memory
```

The local memory size must be taken into account, since this memory was made small in order to speed up access to it. Naturally, memory space larger than the memory space cannot be allocated. The size of the local memory space that can be used on a work-group can be found by using the

clGetDeviceInfo function, passing CL_DEVICE_LOCAL_MEM_SIZE as an argument. The local memory size depends on the hardware used, but it is typically between 10 KB and a few hundred KB. OpenCL expects at least 16 KB of local memory.

Since this memory size varies depending on the hardware, there may be cases where you may want to set the memory size at runtime. This can be done by passing in an appropriate value to the kernel via clSetKernelArg(). List 5.17 and List 5.18 shows an example where the local memory size for a kernel is specified based on the available local memory on the device. The available local memory is retrieved using the clGetDeviceInfo function, and each kernel is given half of the available local memory.

List 5.17: Set local memory size at runtime (kernel)

```
001: __kernel void local_test(__local char *p, int local_size) {
002:       for (int i=0; i < local_size; i++) {
003:                   p[i] = i;
004:       }
005: }
```

List 5.18: Set local memory size at runtime (host)

```
001: #include <stdlib.h>
002: #ifdef __APPLE__
003: #include <OpenCL/opencl.h>
004: #else
005: #include <CL/cl.h>
006: #endif
007: #include <stdio.h>
008:
009: #define MAX_SOURCE_SIZE (0x100000)
010:
011: int main() {
012:       cl_platform_id platform_id = NULL;
013:       cl_uint ret_num_platforms;
014:       cl_device_id device_id = NULL;
015:       cl_uint ret_num_devices;
016:       cl_context context = NULL;
017:       cl_command_queue command_queue = NULL;
```

```
018:    cl_program program = NULL;
019:    cl_kernel kernel = NULL;
020:    size_t kernel_code_size, local_size_size;
021:    char *kernel_src_str;
022:    cl_int ret;
023:    FILE *fp;
024:    cl_ulong local_size;
025:    cl_int cl_local_size;
026:    cl_event ev;
027:
028:    clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
029:    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
030:                &ret_num_devices);
031:    context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
032:
033:    command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
034:    fp = fopen("local.cl", "r");
035:    kernel_src_str = (char*)malloc(MAX_SOURCE_SIZE);
036:    kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
037:    fclose(fp);
038:
039:    /* Get available local memory size */
040:    clGetDeviceInfo(device_id, CL_DEVICE_LOCAL_MEM_SIZE, sizeof(local_size),
&local_size, &local_size_size);
041:    printf("CL_DEVICE_LOCAL_MEM_SIZE = %d¥n", (int)local_size);
042:
043:    /* Build Program */
044:    program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
045:    (const size_t *)&kernel_code_size, &ret);
046:    clBuildProgram(program, 1, &device_id, "", NULL, NULL);
047:    kernel = clCreateKernel(program, "local_test", &ret);
048:
049:    cl_local_size = local_size / 2;
050:
051:    /* Set kernel argument */
052:    clSetKernelArg(kernel, 0, cl_local_size, NULL);
```

```
053:      clSetKernelArg(kernel, 1, sizeof(cl_local_size), &cl_local_size);
054:
055:      /* Execute kernel */
056:      ret = clEnqueueTask(command_queue, kernel, 0, NULL, &ev);
057:      if (ret == CL_OUT_OF_RESOURCES) {
058:                puts("too large local");
059:                return 1;
060:      }
061:      /* Wait for the kernel to finish */
062:      clWaitForEvents(1, &ev);
063:
064:      clReleaseKernel(kernel);
065:      clReleaseProgram(program);
066:      clReleaseCommandQueue(command_queue);
067:      clReleaseContext(context);
068:
069:      free(kernel_src_str);
070:
071:      return 0;
072: }
```

We will now go through the above code, starting with the kernel code.

```
001: __kernel void local_test(__local char *p, int local_size) {
```

The kernel receives a pointer to the local memory for dynamic local memory allocation.

```
002:      for (int i=0; i < local_size; i++) {
003:                p[i] = i;
004:       }
```

This pointer can be used like any other pointer. However, since the local memory cannot be read by the host program, the value computed in this kernel gets thrown away. In actual programs, data stored on the local memory must be transferred over to the global memory in order to be accessed by the host.

We will now look at the host code.

```
039:	/* Get available local memory size */
040:	clGetDeviceInfo(device_id, CL_DEVICE_LOCAL_MEM_SIZE, sizeof(local_size),
&local_size, &local_size_size);
041:	printf("CL_DEVICE_LOCAL_MEM_SIZE = %d¥n", (int)local_size);
```

The clGetDeviceInfo() retrieves the local memory size, and this size is returned to the address of "local_size", which is of the type cl_ulong.

```
049:	cl_local_size = local_size / 2;
```

The local memory may be used by the OpenCL implementation in some cases, so the actual usable local memory size may be smaller than the value retrieved using clGetDeviceInfo. We are only using half of the available local memory here to be on the safe side.

```
051: /* Set Kernel Argument */
052: clSetKernelArg(kernel, 0, cl_local_size, NULL);
```

The above code sets the size of local memory to be used by the kernel. This is given in the 3rd argument, which specifies the argument size. The 4th argument is set to NULL, which is the value of the argument.

```
056:	ret = clEnqueueTask(command_queue, kernel, 0, NULL, &ev);
057:	if (ret == CL_OUT_OF_RESOURCES) {
058:		puts("too large local");
059:		return 1;
060:	 }
```

The kernel is enqueued using clEnqueueTask(). If the specified local memory size is too large, the kernel will not be executed, returning the error code CL_OUT_OF_RESOURCES. For the record, the local memory does not need to be freed.

--- COLUMN: Parallel Programming and Memory Hierarchy ---

For those of you new to programming in a heterogeneous environment, you may find the

emphasis on memory usage to be a bit confusing and unnecessary (those familiar with GPU or Cell/B.E. probably find most of the content to be rather intuitive).

However, effective parallel programming essentially boils down to efficient usage of available memory. Therefore, memory management is emphasized in this text.

This column will go over the basics of parallel programming and memory hierarchy.

First important fact is that memory access cannot be accelerated effectively through the use of cache on multi-core/many-processor systems, as it had been previously with single cores. A cache is a fast storage buffer on a processor that temporarily stores the previously accessed content of the main memory, so that this content can be accessed quickly by the processor. The problem with this is that with multiple processors, the coherency of the cache is not guaranteed, since the data on the memory can be changed by one processor, while another processor still keeps the old data on cache.

Therefore, there needs to be a way to guarantee coherency of the cache coherency across multiple processors. This requires data transfers between the processors.

So how is data transferred between processors? For 2 processors A and B, data transfer occurs between A and B (1 transfer). For 3 processors A, B, and C, data is transferred from A to B, A to C, or B to C (3 transfers). For 4 processors A, B, C, and D, this becomes A to B, A to C, A to D, B to C, B to D, C to D (6 transfers). As the number of processors increase, the number of transfers explodes. The number of processing units (such as ALU), on the other hand, is proportional to the number of processors. This should make apparent of the fact that the data transfer between processors becomes the bottleneck, and not the number of processing units.

Use of a cache requires the synchronization of the newest content of the main memory, which can be viewed as a type of data transfer between processors. Since the data transfer is a bottleneck in multi-core systems, the use of cache to speed up memory access becomes a difficult task.

For this reason, the concept of scratch-pad memory was introduced to replace cache, which the programmer must handle instead of depending on the cache hardware.

On Cell/B.E. SPEs and NVIDIA GPUs, one thing they have in common is that neither of them have a cache. Because of this, the memory transfer cost is strictly just the data transfer cost,

which reduce the data transfer cost in cache that arises from using multiple processors.

Each SPE on Cell/B.E. has a local storage space and a hardware called MFC. The transfer between the local storage and the main memory (accessible from all SPE) is controlled in software. The software looks at where memory transfers are taking place in order to keep the program running coherently.

NVIDIA GPUs do not allow DMA transfer to the local memory (shared memory), and thus data must be loaded from the global memory on the device. When memory access instruction is called, the processor stalls for a few hundred clock cycles, which manifest the slow memory access speed that gets hidden on normal CPUs due to the existence of cache. On NVIDIA GPU, however, a hardware thread is implemented, which allows another process to be performed during the memory access. This can be used to hide the slow memory access speed.

Image Object

In image processing, resizing and rotation of 2-D images (textures) are often performed, which requires the processing of other pixels in order to produce the resulting image. The complexity of the processing is typically inversely proportional to the clarity of the processed image. For example, real-time 3-D graphics only performs relatively simple processing such as nearest neighbor interpolation and linear interpolation. Most GPUs implement some of these commonly used methods on a hardware called the texture unit.

OpenCL allows the use of these image objects as well as an API to process these image objects. The API allows the texture unit to be used to perform the implemented processes. This API is intended for the existing GPU hardware, which may not be fit to be used on other devices, but the processing may be accelerated if the hardware contains a texture unit.

Image object can be used using the following procedure.

1. Create an image object from the host (clCreateImage2D, clCreateImage3D)
2. Write data to the image object from the host (clEnqueueWriteImage)
3. Process the image on the kernel
4. Read data from the image object on the host (clEnqueueReadImage)

An example code is shown in List 5.19 and List 5.20 below.

List 5.19: Kernel (image.cl)

```
001: const sampler_t s_nearest = CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE;
002: const sampler_t s_linear = CLK_FILTER_LINEAR | CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE;
003: const sampler_t s_repeat = CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_REPEAT;
004:
005: __kernel void
006: image_test(__read_only image2d_t im,
007:                    __global float4 *out)
008: {
009:       /* nearest */
010:       out[0] = read_imagef(im, s_nearest, (float2)(0.5f, 0.5f));
011:       out[1] = read_imagef(im, s_nearest, (float2)(0.8f, 0.5f));
012:       out[2] = read_imagef(im, s_nearest, (float2)(1.3f, 0.5f));
013:
014:       /* linear */
015:       out[3] = read_imagef(im, s_linear, (float2)(0.5f, 0.5f));
016:       out[4] = read_imagef(im, s_linear, (float2)(0.8f, 0.5f));
017:       out[5] = read_imagef(im, s_linear, (float2)(1.3f, 0.5f));
018:
019:       /* repeat */
020:       out[6] = read_imagef(im, s_repeat, (float2)(4.5f, 0.5f));
021:       out[7] = read_imagef(im, s_repeat, (float2)(5.0f, 0.5f));
022:       out[8] = read_imagef(im, s_repeat, (float2)(6.5f, 0.5f));
023: }
```

List 5.20: Host code (image.cpp)

```
001: #include <stdlib.h>
002: #ifdef __APPLE__
003: #include <OpenCL/opencl.h>
004: #else
005: #include <CL/cl.h>
006: #endif
```

```
007: #include <stdio.h>
008:
009: #define MAX_SOURCE_SIZE (0x100000)
010:
011: int main()
012: {
013:       cl_platform_id platform_id = NULL;
014:       cl_uint ret_num_platforms;
015:       cl_device_id device_id = NULL;
016:       cl_uint ret_num_devices;
017:       cl_context context = NULL;
018:       cl_command_queue command_queue = NULL;
019:       cl_program program = NULL;
020:       cl_kernel kernel = NULL;
021:       size_t kernel_code_size;
022:       char *kernel_src_str;
023:       float *result;
024:       cl_int ret;
025:       int i;
026:       FILE *fp;
027:       size_t r_size;
028:
029:       cl_mem image, out;
030:       cl_bool support;
031:       cl_image_format fmt;
032:
033:       int num_out = 9;
034:
035:       clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
036:       clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
037:                   &ret_num_devices);
038:       context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
039:       result = (float*)malloc(sizeof(cl_float4)*num_out);
040:
041:        /* Check if the device support images */
042:       clGetDeviceInfo(device_id, CL_DEVICE_IMAGE_SUPPORT, sizeof(support), &support,
```

```
&r_size);
043:      if (support != CL_TRUE) {
044:                puts("image not supported");
045:                return 1;
046:      }
047:
048:      command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
049:      fp = fopen("image.cl", "r");
050:      kernel_src_str = (char*)malloc(MAX_SOURCE_SIZE);
051:      kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
052:      fclose(fp);
053:
054:      /* Create output buffer */
055:      out = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(cl_float4)*num_out,
NULL, &ret);
056:
057:      /* Create data format for image creation */
058:      fmt.image_channel_order = CL_R;
059:      fmt.image_channel_data_type = CL_FLOAT;
060:
061:      /* Create Image Object */
062:      image = clCreateImage2D(context, CL_MEM_READ_ONLY, &fmt, 4, 4, 0, 0, NULL);
063:
064:       /* Set parameter to be used to transfer image object */
065:       size_t origin[] = {0, 0, 0}; /* Transfer target coordinate*/
066:       size_t region[] = {4, 4, 1}; /* Size of object to be transferred */
067:
068:       float data[] = { /* Transfer Data */
069:                10, 20, 30, 40,
070:                10, 20, 30, 40,
071:                10, 20, 30, 40,
072:                10, 20, 30, 40,
073:      };
074:
075:      /* Transfer to device */
076:      clEnqueueWriteImage(command_queue, image, CL_TRUE, origin, region,
```

```
4*sizeof(float), 0, data, 0, NULL, NULL);
077:
078:        /* Build program */
079:        program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
080:        (const size_t *)&kernel_code_size, &ret);
081:        clBuildProgram(program, 1, &device_id, "", NULL, NULL);
082:        kernel = clCreateKernel(program, "image_test", &ret);
083:
084:        /* Set Kernel Arguments */
085:        clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&image);
086:        clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&out);
087:
088:        cl_event ev;
089:        clEnqueueTask(command_queue, kernel, 0, NULL, &ev);
090:
091:        /* Retrieve result */
092:        clEnqueueReadBuffer(command_queue, out, CL_TRUE, 0, sizeof(cl_float4)*num_out,
result, 0, NULL, NULL);
093:
094:        for (i=0; i < num_out; i++) {
095:        printf("%f,%f,%f,%f¥n",result[i*4+0],result[i*4+1],result[i*4+2],result[i*4+3]);
096:        }
097:
098:        clReleaseMemObject(out);
099:        clReleaseMemObject(image);
0100:
0101:       clReleaseKernel(kernel);
0102:       clReleaseProgram(program);
0103:       clReleaseCommandQueue(command_queue);
0104:       clReleaseContext(context);
0105:
0106:       free(kernel_src_str);
0107:       free(result);
0108:
0109:       return 0;
0110: }
```

The result is the following (the result may vary slightly, since OpenCL does not guarantee precision of operations).

```
10.000000,0.000000,0.000000,1.000000
10.000000,0.000000,0.000000,1.000000
10.000000,0.000000,0.000000,1.000000
10.000000,0.000000,0.000000,1.000000
13.000000,0.000000,0.000000,1.000000
18.000000,0.000000,0.000000,1.000000
10.000000,0.000000,0.000000,1.000000
30.000000,0.000000,0.000000,1.000000
10.000000,0.000000,0.000000,1.000000
```

We will start the explanation from the host side.

```
041:    /* Check if the device support images */
042:    clGetDeviceInfo(device_id, CL_DEVICE_IMAGE_SUPPORT, sizeof(support), &support,
&r_size);
043:    if (support != CL_TRUE) {
044:            puts("image not supported");
045:            return 1;
046:    }
```

The above must be performed, as not all OpenCL implementation may support image objects. It is supported if CL_DEVICE_IMAGE_SUPPORT returns CL_TRUE [17].

```
057:    /* Create data format for image creation */
058:    fmt.image_channel_order = CL_R;
059:    fmt.image_channel_data_type = CL_FLOAT;
```

The above code sets the format of the image object. The format is of type cl_image_format, which is a struct containing two elements. "image_channel_order" sets the order of the element, and "image_channel_data_type" sets the type of the element.

Each data in the image object is of a vector type containing 4 components. The possible values

for "image_channel_order" are shown in Table 5.6 below. The 0/1 value represent the fact that those components are set to 0.0f/1.0f. The X/Y/Z/W are the values that can be set for the format.

Table 5.6: Possible values for "image_channel_order"

Enum values for channel order	Corresponding Format
CL_R	(X,0,0,1)
CL_A	(0,0,0,X)
CL_RG	(X,Y,0,1)
CL_RA	(X,0,0,Y)
CL_RGB	(X,Y,Z,1)
CL_RGBA,CL_BGRA,CL_ARGB	(X,Y,Z,W)
CL_INTENSITY	(X,X,X,X)
CL_LUMINANCE	(X,X,X,1)

The values that can be set for "image_channel_data_type" is shown in Table 5.7 below. Of these, CL_UNORM_SHORT_565, CL_UNORM_SHORT_555, CL_UNORM_SHORT_101010 can only be set when "image_channel_order" is set to "CL_RGB".

Table 5.7: Possible values for "image_channel_data_types"

Image Channel Data Type	Corresponding Type
CL_SNORM_INT8	char
CL_SNORM_INT16	short
CL_UNORM_INT8	uchar
CL_UNORM_INT16	ushort
CL_UNORM_SHORT_565	ushort (with RGB)
CL_UNORM_SHORT_555	ushort (with RGB)
CL_UNORM_SHORT_101010	uint (with RGB)
CL_SIGNED_INT8	char
CL_SIGNED_INT16	short
CL_SIGNED_INT32	int
CL_UNSIGNED_INT8	uchar
CL_UNSIGNED_INT16	ushort
CL_UNSIGNED_INT32	uint
CL_FLOAT	float
CL_HALF_FLOAT	half

Now that the image format is specified, we are ready to create the image object.

```
061:    /* Create image object */
062:    image = clCreateImage2D(context, CL_MEM_READ_ONLY, &fmt, 4, 4, 0, 0, NULL);
```

As shown above, image object is created using clCreateImage2D(). The arguments are the corresponding context, read/write permission, image data format, width, height, image row pitch, host pointer, and error code. Host pointer is used if the data already exists on the host, and the data is to be used directly from the kernel. If this is set, the image row pitch must be specified. The host pointer is not specified in this case.

Now the data for the image object is transferred from the host to device.

```
064:    /* Set parameter to be used to transfer image object */
065:    size_t origin[] = {0, 0, 0}; /* Transfer target coordinate*/
066:    size_t region[] = {4, 4, 1}; /* Size of object to be transferred */
067:
068:    float data[] = { /* Transfer Data */
069:            10, 20, 30, 40,
070:            10, 20, 30, 40,
071:            10, 20, 30, 40,
072:            10, 20, 30, 40,
073:    };
074:
075:    /* Transfer to device */
076:    clEnqueueWriteImage(command_queue, image, CL_TRUE, origin, region,
4*sizeof(float), 0, data, 0, NULL, NULL);
```

The clEnqueueWriteImage() is used to transfer the image data. The arguments are command queue, image object, block enable, target coordinate, target size, input row pitch, input slice pitch, pointer to data to be transferred, number of events in wait list, event wait list, and event object. Refer to 4-1-3 for explanation on the block enable and events objects. The target coordinate and target size are specified in a 3-component vector of type size_t. If the image object is 2-D, the 3rd component to target coordinate is 0, and the 3rd component to the size is 1.
Passing this image object into the device program will allow the data to be accessed.

Now we will go over the device program.

```
001: const sampler_t s_nearest = CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE;
002: const sampler_t s_linear = CLK_FILTER_LINEAR | CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE;
003: const sampler_t s_repeat = CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_REPEAT;
```

The "sampler_t" type defines a sampler object. OpenCL allows different modes on how image objects are read, and these properties are defined in this object type, which gets passed in as a parameter when reading an image object. The properties that can be set are the following.

• Filter Mode
• Normalize Mode
• Addressing Mode

The values that can be set for each property are shown in Table 5.8 below.

Table 5.8: Sampler Object Property Values

Sampler state	Predefined Enums	Description
<filter mode>	CLK_FILTER_NEAREST	Use nearest defined coordinate
	CLK_FILTER_LINEAR	Interpolate between neighboring values
<normalized coords>	CLK_NORMALIZED_COORDS_FALSE	Unnormalized
	CLK_NORMALIZED_COORDS_TRUE	Normalized
<address mode>	CLK_ADDRESS_REPEAT	Out-of-range image coordinates wrapped to valid range

	CLK_ADDRESS_CLAMP_TO_EDGE	Out-of-range coord clamped to the extent
	CLK_ADDRESS_CLAMP	Out-of-range coord clamped to border color
	CLK_ADDRESS_NONE	Undefined.

The filter mode specifies the how non-exact coordinates are handled. The addressing mode specifies how out-of-range image coordinates are handled.

The filter mode decides how a value at non-exact coordinate is obtained. OpenCL defines the CLK_FILTER_NEAREST and CLK_FILTER_LINEAR modes. The CLK_FILTER_NEAREST mode determines the closest coordinate defined, and uses this coordinate to access the value at the coordinate. The CLK_FILTER_LINEAR mode interpolates the values from nearby coordinates (nearest 4 points if 2D and nearest 8 points if 3D).

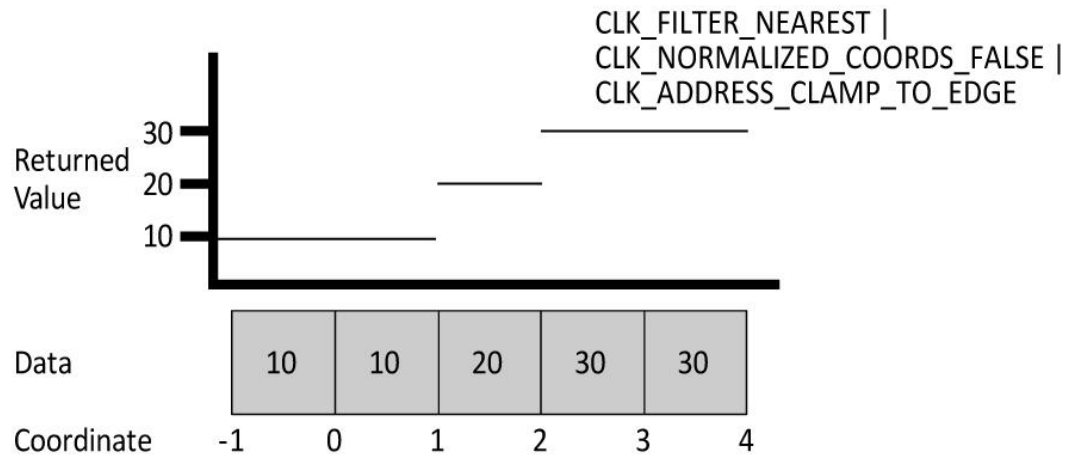
The address mode decides what to do when a coordinate outside the range of the image is accessed. If CLK_ADDRESS_CLAMP is specified, then the returned value is determined based on the "image_channel_order" as shown in Table 5.9.

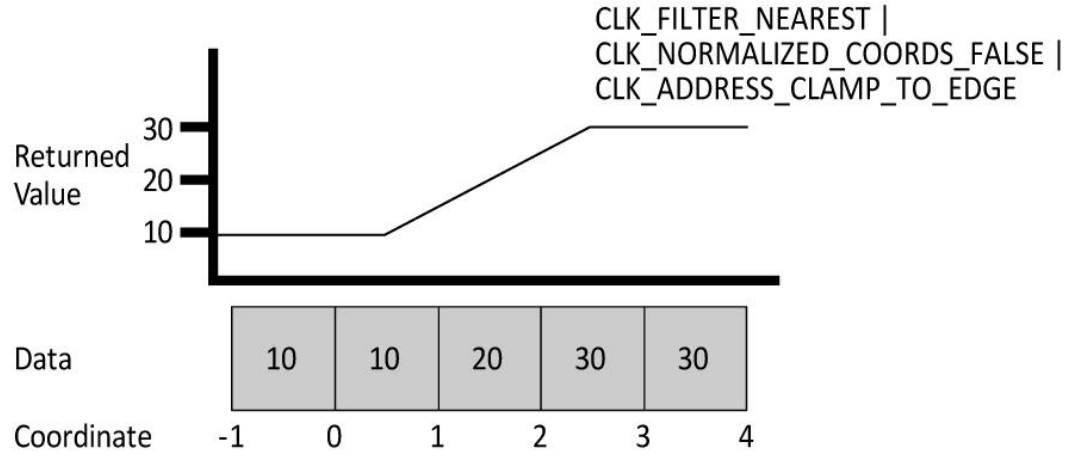
Table 5.9: CLK_ADDRESS_CLAMP Result as a function of "image_channel_order"

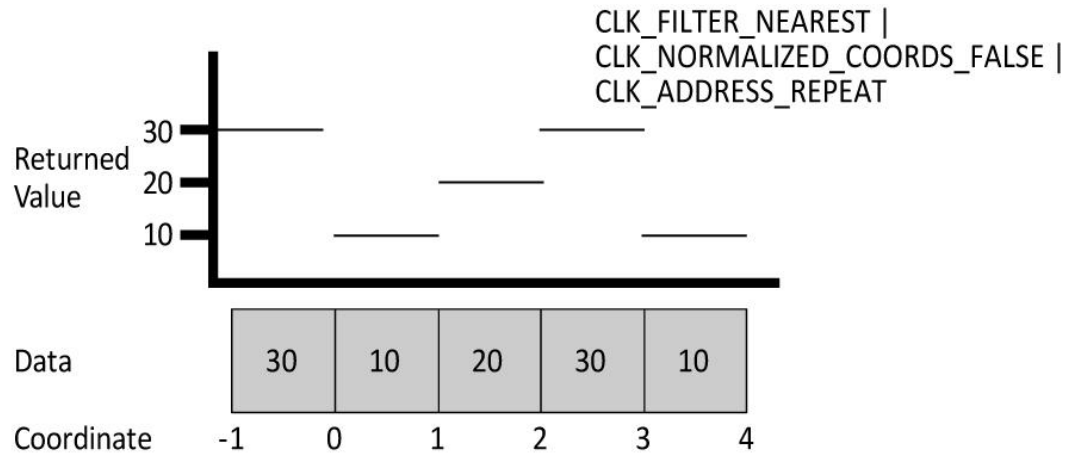
Enum values for channel order	Clamp point
CL_A, CL_INTENSITY, CL_RA, CL_ARGB, CL_BGRA, CL_RGBA	(0.0f, 0.0f, 0.0f, 0.0f)
CL_R, CL_RG, CL_RGB, CL_LUMINANCE	(0.0f, 0.0f, 0.0f, 1.0f)

In our sample code, the input data is set to (10, 20, 30). Plot of the results for different sample modes are shown in Figure 5.4.

Figure 5.4: Sampler object examples







The coordinates are normalized if the CLK_NORMALIZED_COORDS_TRUE is specified. This normalizes the image object so that its coordinates are accessed by a value between 0 and 1.0. However, the normalization must be either enabled or disabled for all image reads within a kernel.

Now we are ready to read the image object.

```
010:    out[0] = read_imagef(im, s_nearest, (float2)(0.5f, 0.5f));
011:    out[1] = read_imagef(im, s_nearest, (float2)(0.8f, 0.5f));
012:    out[2] = read_imagef(im, s_nearest, (float2)(1.3f, 0.5f));
```

The image object can be read using the built-in functions read_imagef(), read_imagei(), or read_imageui(). The format when the image object is read must match the format used when the object was created using clCreateImage2D. The formats are shown in Table 5.10 below. (Note: The action of the read_image*() function is undefined when there is a format mismatch)

Table 5.10: Format to specify when reading an image object

Function	Format	Description
read_imagef()	CL_SNORM_INT8	-127 ~ 127 normalized to -1.0 ~ 1.0 (-128 becomes -1.0)
	CL_SNORM_INT16	-32767 ~ 32767 normalized to -1.0 ~ 1.0 (-32768 becomes -1.0)
	CL_UNORM_INT8	0 ~ 255 normalized to 0.0 ~ 1.0
	CL_UNORM_INT16	0 ~ 65535 normalized to 0.0 ~ 1.0
	CL_FLOAT	as is
	CL_HALF_FLOAT	Half value converted to float
read_imagei()	CL_SIGNED_INT8	-128 ~ 127
	CL_SIGNED_INT16	-32768 ~ 32767
	CL_SIGNED_INT32	-2147483648 ~ 2147483647
read_imageui()	CL_SIGNED_INT8	0 ~ 255
	CL_SIGNED_INT16	0 ~ 65535
	CL_SIGNED_INT32	0 ~ 4294967296

The arguments of the read_image*() functions are the image object, sampler object, and the coordinate to read. The coordinate is specified as a float2 type. The returned type is float4, int4, or uint4 depending on the format.

Embedded Profile

OpenCL is intended to not only to be used on desktops environments, but for embedded systems

as well. However, it may not be possible to implement every functions defined in OpenCL for embedded systems. For this reason, OpenCL defines the Embedded Profile. The implementation that only supports the Embedded Profile may have the following restrictions.

• Rounding of floats
OpenCL typically uses the round-to-nearest-even method by default for rounding, but in the Embedded Profile, it may use the round-towards-zero method if the former is not implemented. This should be checked using the clGetDeviceInfo() function to get the "CL_DEVICE_SINGLE_FP_CONFIG" property.

• Precision of Built-in functions
Some built-in functions may have lower precision than the precision defined in OpenCL.

• Other optional functionalities
In addition to the above, 3D image filter, 64-bit long/ulong, float Inf and NaN may not be implemented, as they are optional. The result of operating on Inf or NaN is undefined, and will vary depending on the implementation.

Attribute Qualifiers

OpenCL allows the usage of attribute qualifiers, which gives specific instructions to the compiler. This is a language extension to standard C that is supported in GCC.

```
__attribute__(( value ))
```

This extension is starting to be supported by compilers other than GCC. The attribute can be placed on variables, types (typedef, struct), and functions.

Examples of an attribute being placed on types are shown below.

```
typedef int aligned_int_t __attribute__((aligned(16)));
typedef struct __attribute__(((packed))) packed_struct_t { ... };
```

Below is an example of an attribute being placed on a variable.

```
int a __attribute__((aligned(16)));
```

The basic grammar is to place an attribute after the variable, type or function to set the attribute to.

The attributes that can be set in OpenCL are shown below.

• aligned
Same as the "aligned" in GCC. When specified on a variable, the start address of that variable gets aligned. When specified on a type, the start address of that type gets aligned. This attribute on a function is undefined in OpenCL.

• packed
Same as the "packed" in GCC. When specified on a variable, the address between this variable and the previously defined variable gets padded. When specified on a struct, each member in the struct gets padded.

• endian
This specifies the byte order used on a variable. "host" or "device" can be passed in as an argument to make the byte order the same as either host or device.

There are also other attributes defined for function, which gives optimization hints to the compiler. Some hints are given below:

• vec_type_hint()
Suggests the type of vector to use for optimization

• work_group_size_hint()
Suggests the number of work-items/work-group.

• reqd_work_group_size()
Specifies the number of work-items/work-group. The kernel will throw an error when the number of work-items/work-group is a different number. (1,1,1) should be specified if the kernel is queued using clEnqueueTask().

Pragma

OpenCL defines several pragmas. The syntax is to use "#pragma OPENCL". 2 pragmas,

FP_CONTRACT, and EXTENSION are defined.

FP_CONTRACT

This is the same as the FP_CONTRACT defined in standard C.

```
#pragma OPENCL FP_CONTRACT on-off-switch
```

Some hardware supports "FMA" instructions, which sums a number with the product of 2 numbers in one instruction. These types of instructions, where multiple operations are performed as 1 operation, are known as contract operations. In some cases, the precision may be different from when the operations are performed separately. This FP_CONTRACT pragma enables or disables the usage of these contract operations.

EXTENSION

This enables or disables optional OpenCL extensions to be used.

```
#pragma OPENCL EXTENSION <extension_name> : <behavior>
```

<extension_name> gets the name of the OpenCL extension. The standard extensions are shown in Table 5.11 below.

Table 5.11: Extension name

Extension name	Extended capability
cl_khr_fp64	Support for double precision
cl_khr_fp16	Support for half precision
cl_khr_select_fprounding_mode	Support for setting the rounding type
cl_khr_global_int32_base_atomics, cl_khr_global_int32_extended_atomics	Support for atomic operations of 32-bit values
cl_khr_global_int64_base_atomics, cl_khr_global_int64_extended_atomics	Support for atomic operations of 64-bit values
cl_khr_3d_image_writes	Enable writes to 3-D image object
cl_khr_byte_addressable_store	Enable byte-size writes to address
all	Support for all extensions

<behavior> gets one of the value on Table 5.12. Specifying "require" when the enabled extension is "all" will display an error.

Table 5.12: Supported values for <behavior>

Behavior	Description
require	Check that extension is supported (compile error otherwise)
disable	Disable extensions
enable	Enable extensions

OpenCL Programming Practice

This section will go over some parallel processing methods that can be used in OpenCL.

We will use a sample application that analyzes stock price data to walk through porting of standard C code to OpenCL C in order to utilize a device. The analysis done in this application is to compute the moving average of the stock price for different stocks.

We will start from a normal C code, and gradually convert sections of the code to be processed in parallel. This should aid you in gaining intuition on how to parallelize your code.

A moving average filter is used commonly in image processing and signal processing as a low-pass filter.

Note that the sample code shown in this section is meant to be pedagogical in order to show how OpenCL is used. You may not experience any speed-up depending on the type of hardware you have.

Standard Single-Thread Programming

We will first walk through a standard C code of the moving average function. The function has the following properties.

• Stock price data in passed in as an int-array named "value". The result of the moving average is returned as an array of floats with the name "average".
• The array length is passed in as "length" of type int
• The width of the data to compute the average for is passed in as "width" of type int

To make the code more intuitive, the code on List 5.21 gets rid of all error checks that would

normally be performed. This function is what we want to process on the device, so this is the code will eventually be ported into kernel code.

List 5.21: Moving average of integers implemented in standard C

```
001: void moving_average(int *values,
002:       float *average,
003:       int length,
004:       int width)
005: {
006:       int i;
007:       int add_value;
008:
009:       /* Compute sum for the first "width" elements */
010:       add_value = 0;
011:       for (i=0; i < width; i++) {
012:                  add_value += values[i];
013:       }
014:       average[width-1] = (float)add_value;
015:
016:       /* Compute sum for the (width)th ~ (length-1)th elements */
017:       for (i=width; i < length; i++) {
018:                  add_value = add_value - values[i-width] + values[i];
019:                  average[i] = (float)(add_value);
020:       }
021:
022:       /* Insert zeros to 0th ~ (width-2)th element */
023:       for (i=0; i < width-1; i++) {
024:                  average[i] = 0.0f;
025:       }
026:
027:       /* Compute average from the sum */
028:       for (i=width-1; i < length; i++) {
029:                  average[i] /= (float)width;
030:       }
031: }
```

In this example, each indices of the average array contain the average of the previous width-1 values and the value of that index. Zeros are placed for average[0] ~ average[width-2] in lines 22~25, since this computation will require values not contained in the input array. In other words, if the width is 3, average[3] contain the average of value[1], value[2] and value[3]. The value of average[0] ~ average[width-2] = average[1] are zeros, since it would require value[-2] and value[-1].

The average itself is computed by first summing up the "width" number of values and storing it into the average array (lines 9-20). Lines 9-14 computes the sum of the first "width" elements, and stores it in average[width-1]. Lines 16-20 computes the sum for the remaining indices. This is done by starting from the previously computed sum, subtracting the oldest value and adding the newest value, which is more efficient than computing the sum of "width" elements each time. This works since the input data is an integer type, but if the input is of type float, this method may result in rounding errors, which can become significant over time. In this case, a method shown in List 5.22 should be used.

List 5.22: Moving average of floats implemented in standard C

```
001: void moving_average_float(float *values,
002:       float *average,
003:       int length,
004:       int width)
005: {
006:       int i, j;
007:       float add_value;
008:
009:       /* Insert zeros to 0th ~ (width-2)th elements */
010:       for (i=0; i < width-1; i++) {
011:                  average[i] = 0.0f;
012:       }
013:
014:       /* Compute average of (width-1) ~ (length-1) elements */
015:       for (i=width-1; i < length; i++) {
016:                  add_value = 0.0f;
017:                  for (j=0; j < width; j++) {
018:                             add_value += values[i - j];
019:                  }
```

```
020:                    average[i] = add_value / (float)width;
021:        }
022: }
```

We will now show a main() function that will call the function in List 5.21 to perform the computation (List 5.24). The input data is placed in a file called "stock_array1.txt", whose content is shown in List 5.23.

List 5.23: Input data (stock_array1.txt)

```
100,
109,
98,
104,
107,
...
50
```

List 5.24: Standard C main() function to all the moving average function

```
001: #include <stdio.h>
002: #include <stdlib.h>
003:
004: /* Read Stock data */
005: int stock_array1[] = {
006:        #include "stock_array1.txt"
007: };
008:
009: /* Define width for the moving average */
010: #define WINDOW_SIZE (13)
011:
012: int main(int argc, char *argv[])
013: {
014:
015:        float *result;
016:
017:        int data_num = sizeof(stock_array1) / sizeof(stock_array1[0]);
018:        int window_num = (int)WINDOW_SIZE;
```

```
019:
020:      int i;
021:
022:      /* Allocate space for the result */
023:      result = (float *)malloc(data_num*sizeof(float));
024:
025:      /* Call the moving average function */
026:      moving_average(stock_array1,
027:                  result,
028:                  data_num,
029:                  window_num);
030:
031:      /* Print result */
032:      for (i=0; i < data_num; i++) {
033:                  printf("result[%d] = %f¥n", i, result[i]);
034:      }
035:
036:      /* Deallocate memory */
037:      free(result);
038: }
```

The above code will be eventually transformed into a host code that will call the moving average kernel.

We have now finished writing a single threaded program to compute the moving average. This will now be converted to OpenCL code to use the device. Our journey has just begun.

Porting to OpenCL

The first step is to convert the moving_average() function to kernel code, or OpenCL C. This code will be executed on the device. The code in List 5.21 becomes as shown in List 5.25 after being ported.

List 5.25: Moving average kernel (moving_average.cl)

```
001: __kernel void moving_average(__global int *values,
002:      __global float *average,
```

```
003:      int length,
004:      int width)
005: {
006:      int i;
007:      int add_value;
008:
009:      /* Compute sum for the first "width" elements */
010:      add_value = 0;
011:      for (i=0; i < width; i++) {
012:                add_value += values[i];
013:      }
014:      average[width-1] = (float)add_value;
015:
016:      /* Compute sum for the (width)th ~ (length-1)th elements */
017:      for (i=width; i < length; i++) {
018:                add_value = add_value - values[i-width] + values[i];
019:                average[i] = (float)(add_value);
020:      }
021:
022:      /* Insert zeros to 0th ~ (width-2)th elements */
023:      for (i=0; i < width-1; i++) {
024:                average[i] = 0.0f;
025:      }
026:
027:      /* Compute average of (width-1) ~ (length-1) elements */
028:      for (i=width-1; i < length; i++) {
029:                average[i] /= (float)width;
030:      }
031: }
```

Note that we have only changed lines 1 and 2, which adds the __kernel qualifier to the function, and the address qualifier __global specifying the location of the input data and where the result will be placed.

The host code is shown in List 5.26.

List 5.26: Host code to execute the moving_average() kernel

```
001: #include <stdlib.h>
002: #ifdef __APPLE__
003: #include <OpenCL/opencl.h>
004: #else
005: #include <CL/cl.h>
006: #endif
007: #include <stdio.h>
008:
009: /* Read Stock data */
010: int stock_array1[] = {
011:       #include "stock_array1.txt"
012: };
013:
014: /* Define width for the moving average */
015: #define WINDOW_SIZE (13)
016:
017: #define MAX_SOURCE_SIZE (0x100000)
018:
019: int main(void)
020: {
021:       cl_platform_id platform_id = NULL;
022:       cl_uint ret_num_platforms;
023:       cl_device_id device_id = NULL;
024:       cl_uint ret_num_devices;
025:       cl_context context = NULL;
026:       cl_command_queue command_queue = NULL;
027:       cl_mem memobj_in = NULL;
028:       cl_mem memobj_out = NULL;
029:       cl_program program = NULL;
030:       cl_kernel kernel = NULL;
031:       size_t kernel_code_size;
032:       char *kernel_src_str;
033:       float *result;
034:       cl_int ret;
035:       FILE *fp;
```

```
036:
037:    int data_num = sizeof(stock_array1) / sizeof(stock_array1[0]);
038:    int window_num = (int)WINDOW_SIZE;
039:    int i;
040:
041:    /* Allocate space to read in kernel code */
042:    kernel_src_str = (char *)malloc(MAX_SOURCE_SIZE);
043:
044:    /* Allocate space for the result on the host side */
045:    result = (float *)malloc(data_num*sizeof(float));
046:
047:    /* Get Platform */
048:    ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
049:
050:    /* Get Device */
051:    ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
052:            &ret_num_devices);
053:
054:    /* Create Context */
055:    context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
056:
057:    /* Create Command Queue */
058:    command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
059:
060:    /* Read Kernel Code */
061:    fp = fopen("moving_average.cl", "r");
062:    kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
063:    fclose(fp);
064:
065:    /* Create Program Object */
066:    program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
067:            (const size_t *)&kernel_code_size, &ret);
068:    /* Compile kernel */
069:    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
070:
071:    /* Create Kernel */
```

```
072:    kernel = clCreateKernel(program, "moving_average", &ret);
073:
074:    /* Create buffer for the input data on the device */
075:    memobj_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
076:            data_num * sizeof(int), NULL, &ret);
077:
078:    /* Create buffer for the result on the device */
079:    memobj_out = clCreateBuffer(context, CL_MEM_READ_WRITE,
080:    data_num * sizeof(float), NULL, &ret);
081:
082:    /* Copy input data to the global memory on the device*/
083:    ret = clEnqueueWriteBuffer(command_queue, memobj_in, CL_TRUE, 0,
084:            data_num * sizeof(int),
085:            stock_array1, 0, NULL, NULL);
086:
087:    /* Set kernel arguments */
088:    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj_in);
089:    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobj_out);
090:    ret = clSetKernelArg(kernel, 2, sizeof(int), (void *)&data_num);
091:    ret = clSetKernelArg(kernel, 3, sizeof(int), (void *)&window_num);
092:
093:    /* Execute the kernel */
094:    ret = clEnqueueTask(command_queue, kernel, 0, NULL, NULL);
095:
096:    /* Copy result from device to host */
097:    ret = clEnqueueReadBuffer(command_queue, memobj_out, CL_TRUE, 0,
098:            data_num * sizeof(float),
099:            result, 0, NULL, NULL);
100:
101:
102:    /* OpenCL Object Finalization */
103:    ret = clReleaseKernel(kernel);
104:    ret = clReleaseProgram(program);
105:    ret = clReleaseMemObject(memobj_in);
106:    ret = clReleaseMemObject(memobj_out);
107:    ret = clReleaseCommandQueue(command_queue);
```

```
108:      ret = clReleaseContext(context);
109:
110:      /* Display Results */
111:      for (i=0; i < data_num; i++) {
112:                printf("result[%d] = %f¥n", i, result[i]);
113:      }
114:
115:      /* Deallocate memory on the host */
116:      free(result);
117:      free(kernel_src_str);
118:
119:      return 0;
120: }
```

This host is based on the code in List 5.24, adding the OpenCL runtime API commands required for the kernel execution. Note the code utilizes the online compile method, as the kernel source code is read in (Lines 60-69).

However, although the code is executable, it is not written to run anything in parallel. The next section will show how this can be done.

Vector Operations

First step to see whether vector-types can be used for the processing. For vector types, we can expect the OpenCL implementation to perform operations using the SIMD units on the processor to speed up the computation. We will start looking at multiple stocks from this section, as this is more practical. The processing will be vector-ized such that the moving average computation for each stock will be executed in parallel. We will assume that the processor will have a 128-bit SIMD unit, to operate on four 32-bit data in parallel. In OpenCL, types such as int4 and float4 can be used.

List 5.27 shows the price data for multiple stocks, where each row contains the price of multiple stocks at one instance in time. For simplicity's sake, we will process the data for 4 stocks in this section.

List 5.27: Price data for multiple stocks (stock_array_many.txt)

```
100, 212, 315, 1098, 763, 995, ..., 12
109, 210, 313, 1100, 783, 983, ..., 15
 98, 209, 310, 1089, 790, 990, ..., 18
104, 200, 319, 1098, 792, 985, ..., 21
107, 100, 321, 1105, 788, 971, ..., 18
…
 50, 33, 259, 980, 687, 950, ..., 9
```

List 5.28: Price data for 4 stocks (stock_array_4.txt)

```
100, 212, 315, 1098,
109, 210, 313, 1100,
 98, 209, 310, 1089,
104, 200, 319, 1098,
107, 100, 321, 1105,
…
 50, 33, 259, 980
```

For processing 4 values at a time, we can just replace int and float with int4 and float4, respectively. The new kernel code will look like List 5.29.

List 5.29: Vector-ized moving average kernel (moving_average_vec4.cl)

```
001: __kernel void moving_average_vec4(__global int4 *values,
002:       __global float4 *average,
003:       int length,
004:       int width)
005: {
006:       int i;
007:       int4 add_value; /* A vector to hold 4 components */
008:
009:       /* Compute sum for the first "width" elements for 4 stocks */
010:       add_value = (int4)0;
011:       for (i=0; i < width; i++) {
012:                   add_value += values[i];
013:       }
014:       average[width-1] = convert_float4(add_value);
015:
```

```
016:      /* Compute sum for the (width)th ～ (length-1)th elements for 4 stocks */
017:      for (i=width; i < length; i++) {
018:                add_value = add_value - values[i-width] + values[i];
019:                average[i] = convert_float4(add_value);
020:      }
021:
022:      /* Insert zeros to 0th ～ (width-2)th element for 4 stocks*/
023:      for (i=0; i < width-1; i++) {
024:                average[i] = (float4)(0.0f);
025:      }
026:
027:      /* Compute average of (width-1) ~ (length-1) elements for 4 stocks */
028:      for (i=width-1; i < length; i++) {
029:                average[i] /= (float4)width;
030:      }
031: }
```

The only differences from List 5.25 is the conversion of scalar to vector type (Lines 1,7,10 for int and Lines 2,24,29 for float), and the use of convert_float4() function. Note that the operators (+, -, *, /) are overloaded to be used on vector-types, so these do not need to be changed (Lines 12, 18, 29).

List 5.30: Host code to run the vector-ized moving average kernel

```
001: #include <stdlib.h>
002: #ifdef __APPLE__
003: #include <OpenCL/opencl.h>
004: #else
005: #include <CL/cl.h>
006: #endif
007: #include <stdio.h>
008:
009: #define NAME_NUM (4) /* Number of Stocks */
010: #define DATA_NUM (21) /* Number of data to process for each stock*/
011:
012: /* Read Stock data */
013: int stock_array_4[NAME_NUM*DATA_NUM]= {
```

```
014:        #include "stock_array_4.txt"
015: };
016:
017: /* Moving average width */
018: #define WINDOW_SIZE (13)
019:
020: #define MAX_SOURCE_SIZE (0x100000)
021:
022: int main(void)
023: {
024:        cl_platform_id platform_id = NULL;
025:        cl_uint ret_num_platforms;
026:        cl_device_id device_id = NULL;
027:        cl_uint ret_num_devices;
028:        cl_context context = NULL;
029:        cl_command_queue command_queue = NULL;
030:        cl_mem memobj_in = NULL;
031:        cl_mem memobj_out = NULL;
032:        cl_program program = NULL;
033:        cl_kernel kernel = NULL;
034:        size_t kernel_code_size;
035:        char *kernel_src_str;
036:        float *result;
037:        cl_int ret;
038:        FILE *fp;
039:
040:        int window_num = (int)WINDOW_SIZE;
041:        int point_num = NAME_NUM * DATA_NUM;
042:        int data_num = (int)DATA_NUM;
043:        int name_num = (int)NAME_NUM;
044:        int i, j;
045:
046:        /* Allocate space to read in kernel code */
047:        kernel_src_str = (char *)malloc(MAX_SOURCE_SIZE);
048:
049:        /* Allocate space for the result on the host side */
```

```
050:      result = (float *)malloc(point_num*sizeof(float));
051:
052:      /* Get Platform */
053:      ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
054:
055:      /* Get Device */
056:      ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
057:                &ret_num_devices);
058:
059:      /* Create Context */
060:      context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
061:
062:      /* Create command queue */
063:      command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
064:
065:      /* Read kernel source code */
066:      fp = fopen("moving_average_vec4.cl", "r");
067:      kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
068:      fclose(fp);
069:
070:      /* Create Program Object */
071:      program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
072:                (const size_t *)&kernel_code_size, &ret);
073:
074:       /* Compile kernel */
075:      ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
076:
077:      /* Create kernel */
078:      kernel = clCreateKernel(program, "moving_average_vec4", &ret);
079:
080:      /* Create buffer for the input data on the device */
081:      memobj_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
082:                point_num * sizeof(int), NULL, &ret);
083:
084:
085:      /* Create buffer for the result on the device */
```

```
086:    memobj_out = clCreateBuffer(context, CL_MEM_READ_WRITE,
087:            point_num * sizeof(float), NULL, &ret);
088:
089:    /* Copy input data to the global memory on the device*/
090:    ret = clEnqueueWriteBuffer(command_queue, memobj_in, CL_TRUE, 0,
091:            point_num * sizeof(int),
092:            stock_array_4, 0, NULL, NULL);
093:
094:    /* Set Kernel Arguments */
095:    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj_in);
096:    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobj_out);
097:    ret = clSetKernelArg(kernel, 2, sizeof(int), (void *)&data_num);
098:    ret = clSetKernelArg(kernel, 3, sizeof(int), (void *)&window_num);
099:
100:    /* Execute kernel */
101:    ret = clEnqueueTask(command_queue, kernel, 0, NULL, NULL);
102:
103:    /* Copy result from device to host */
104:    ret = clEnqueueReadBuffer(command_queue, memobj_out, CL_TRUE, 0,
105:            point_num * sizeof(float),
106:            result, 0, NULL, NULL);
107:
108:    /* OpenCL Object Finalization */
109:    ret = clReleaseKernel(kernel);
110:    ret = clReleaseProgram(program);
111:    ret = clReleaseMemObject(memobj_in);
112:    ret = clReleaseMemObject(memobj_out);
113:    ret = clReleaseCommandQueue(command_queue);
114:    ret = clReleaseContext(context);
115:
116:    /* Print results */
117:    for (i=0; i < data_num; i++) {
118:            printf("result[%d]:", i);
119:            for (j=0; j < name_num; j++) {
120:                    printf("%f, ", result[i*NAME_NUM+j]);
121:            }
```

```
122:                    printf("¥n");
123:        }
124:
125:        /* Deallocate memory on the host */
126:        free(result);
127:        free(kernel_src_str);
128:
129:        return 0;
130: }
```

The only difference from List 5.26 is that the data to process is increased by a factor of 4, and that the data length parameter is hard coded. In addition, the kernel code that gets read is changed to moving_average_vec4.cl (line 66), and the kernel name is changed to moving_average_vec4 (line 78).

We will now change the program to allow processing of more than 4 stocks, as in the data in List 5.27. For simplicity, we will assume the number of stocks to process is a multiple of 4. We could just call the kernel on List 5.29 and vector-ize the input data on the host side, but we will instead allow the kernel to take care of this.

Since we will be processing 4 stocks at a time, the kernel code will just have to loop the computation so that more than 4 stocks can be computed within the kernel. The kernel will take in a parameter "name_num", which is the number of stocks to process. This will be used to calculate the number of loops required to process all stocks.

The new kernel code is shown in List 5.31 below.

List 5.31: Moving average kernel of (multiple of 4) stocks (moving_average_many.cl)

```
001: __kernel void moving_average_many(__global int4 *values,
002:        __global float4 *average,
003:        int length,
004:        int name_num,
005:        int width)
006: {
007:        int i, j;
```

```
008:      int loop_num = name_num / 4; /* compute the number of times to loop */
009:      int4 add_value;
010:
011:      for (j=0; j < loop_num; j++) {
012:                /* Compute sum for the first "width" elements for 4 stocks */
013:                add_value = (int4)0;
014:                for (i=0; i < width; i++) {
015:                          add_value += values[i*loop_num+j];
016:                }
017:                average[(width-1)*loop_num+j] = convert_float4(add_value);
018:
019:                /* Compute sum for the (width)th ~ (length-1)th elements for 4 stocks */
020:                for (i=width; i < length; i++) {
021:                          add_value = add_value - values[(i-width)*loop_num+j] +
values[i*loop_num+j];
022:                          average[i*loop_num+j] = convert_float4(add_value);
023:                }
024:
025:                /* Insert zeros to 0th ~ (width-2)th element for 4 stocks*/
026:                for (i=0; i < width-1; i++) {
027:                          average[i*loop_num+j] = (float4)(0.0f);
028:                }
029:
030:                /* Compute average of (width-1) ~ (length-1) elements for 4 stocks */
031:                for (i=width-1; i < length; i++) {
032:                          average[i*loop_num+j] /= (float4)width;
033:                }
034:      }
035: }
```

The host code is shown in List 5.32.

List 5.32: Host code for calling the kernel in List 5.31

```
001: #include <stdlib.h>
002: #ifdef __APPLE__
003: #include <OpenCL/opencl.h>
```

```
004: #else
005: #include <CL/cl.h>
006: #endif
007: #include <stdio.h>
008:
009: #define NAME_NUM (8) /* Number of stocks */
010: #define DATA_NUM (21) /* Number of data to process for each stock */
011:
012: /* Read Stock data */
013: int stock_array_many[NAME_NUM*DATA_NUM]= {
014:       #include "stock_array_many.txt"
015: };
016:
017: /* Moving average width */
018: #define WINDOW_SIZE (13)
019:
020: #define MAX_SOURCE_SIZE (0x100000)
021:
022: int main(void)
023: {
024:       cl_platform_id platform_id = NULL;
025:       cl_uint ret_num_platforms;
026:       cl_device_id device_id = NULL;
027:       cl_uint ret_num_devices;
028:       cl_context context = NULL;
029:       cl_command_queue command_queue = NULL;
030:       cl_mem memobj_in = NULL;
031:       cl_mem memobj_out = NULL;
032:       cl_program program = NULL;
033:       cl_kernel kernel = NULL;
034:       size_t kernel_code_size;
035:       char *kernel_src_str;
036:       float *result;
037:       cl_int ret;
038:       FILE *fp;
039:
```

```
040:    int window_num = (int)WINDOW_SIZE;
041:    int point_num = NAME_NUM * DATA_NUM;
042:    int data_num = (int)DATA_NUM;
043:    int name_num = (int)NAME_NUM;
044:
045:    int i, j;
046:
047:    /* Allocate space to read in kernel code */
048:    kernel_src_str = (char *)malloc(MAX_SOURCE_SIZE);
049:
050:    /* Allocate space for the result on the host side */
051:    result = (float *)malloc(point_num*sizeof(float));
052:
053:    /* Get Platform*/
054:    ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
055:
056:    /* Get Device */
057:    ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
058:            &ret_num_devices);
059:
060:    /* Create Context */
061:    context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
062:
063:    /* Create Command Queue */
064:    command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
065:
066:    /* Read kernel source code */
067:    fp = fopen("moving_average_many.cl", "r");
068:    kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
069:    fclose(fp);
070:
071:    /* Create Program Object */
072:    program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
073:            (const size_t *)&kernel_code_size, &ret);
074:
075:    /* Compile kernel */
```

```
076:      ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
077:
078:      /* Create kernel */
079:      kernel = clCreateKernel(program, "moving_average_many", &ret);
080:
081:      /* Create buffer for the input data on the device */
082:      memobj_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
083:      point_num * sizeof(int), NULL, &ret);
084:
085:      /* Create buffer for the result on the device */
086:      memobj_out = clCreateBuffer(context, CL_MEM_READ_WRITE,
087:                point_num * sizeof(float), NULL, &ret);
088:
089:      /* Copy input data to the global memory on the device*/
090:      ret = clEnqueueWriteBuffer(command_queue, memobj_in, CL_TRUE, 0,
091:                point_num * sizeof(int),
092:                stock_array_many, 0, NULL, NULL);
093:
094:      /* Set Kernel Arguments */
095:      ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj_in);
096:      ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobj_out);
097:      ret = clSetKernelArg(kernel, 2, sizeof(int), (void *)&data_num);
098:      ret = clSetKernelArg(kernel, 3, sizeof(int), (void *)&name_num);
099:      ret = clSetKernelArg(kernel, 4, sizeof(int), (void *)&window_num);
100:
101:      /* Execute kernel */
102:      ret = clEnqueueTask(command_queue, kernel, 0, NULL, NULL);
103:
104:      /* Copy result from device to host */
105:      ret = clEnqueueReadBuffer(command_queue, memobj_out, CL_TRUE, 0,
106:                point_num * sizeof(float),
107:                result, 0, NULL, NULL);
108:
109:      /* OpenCL Object Finalization */
110:      ret = clReleaseKernel(kernel);
111:      ret = clReleaseProgram(program);
```

```
112:      ret = clReleaseMemObject(memobj_in);
113:      ret = clReleaseMemObject(memobj_out);
114:      ret = clReleaseCommandQueue(command_queue);
115:      ret = clReleaseContext(context);
116:
117:      /* Print results */
118:      for (i=0; i < data_num; i++) {
119:                printf("result[%d]:", i);
120:                for (j=0; j < name_num; j++) {
121:                          printf("%f, ", result[i*NAME_NUM+j]);
122:                }
123:                printf("¥n");
124:      }
125:
126:      /* Deallocate memory on the host */
127:      free(result);
128:      free(kernel_src_str);
129:
130:      return 0;
131: }
```

The only difference from List 5.30 is that the number of stocks to process has been increased to 8, which get passed in as an argument to the kernel (line 98). In addition, the kernel code that gets read is changed to moving_average_many.cl (line 67), and the kernel name is changed to moving_average_many (line 79).

This section concentrated on using SIMD units to perform the same process on multiple data sets in parallel. This is the most basic method of parallelization, which is done simply by replacing scalar-types with vector-types. The next step is expanding this to use multiple compute units capable of performing SIMD operations.

Data Parallel Processing

This section will focus on using multiple compute units to perform moving average for multiple stocks. Up until this point, only one instance of the kernel was executed, which processed all the data. To use multiple compute units simultaneously, multiple kernel instances must be executed

in parallel. They can either be the same kernel running in parallel (data parallel), or different kernels in parallel (task parallel). We will use the data parallel model, as this method is more suited for this process.

We will use the kernel in List 5.31 as the basis to perform the averaging on 8 stocks. Since this code operates on 4 data sets at once, we can use 2 compute units to perform operations on 8 data sets at once. This is achieved by setting the work group size to 2 when submitting the task.

In order to use the data parallel mode, each instance of the kernel must know where it is being executed within the index space. If this is not done, the same kernel will run on the same data sets. The get_global_id() function can be used to get the kernel instance's global ID, which happens to be the same value as the value of the iterator "j". Therefore, the code in List 5.31 can be rewritten to the following code in List 5.33.

List 5.33: Moving average kernel for 4 stocks (moving_average_vec4_para.cl)

```
001: __kernel void moving_average_vec4_para(__global int4 *values,
002:       __global float4 *average,
003:       int length,
004:       int name_num,
005:       int width)
006: {
007:       int i, j;
008:       int loop_num = name_num / 4;
009:       int4 add_value;
010:
011:       j = get_global_id(0);     /* Used to select different data set for each instance */
012:
013:       /* Compute sum for the first "width" elements for 4 stocks */
014:       add_value = (int4)0;
015:       for (i=0; i < width; i++) {
016:                   add_value += values[i*loop_num+j]; /* "j" decides on the data subset to
process for the kernel instance*/
017:       }
018:       average[(width-1)*loop_num+j] = convert_float4(add_value);
019:
020:       /* Compute sum for the (width)th ~ (length-1)th elements for 4 stocks */
```

```
021:      for (i=width; i < length; i++) {
022:                add_value = add_value - values[(i-width)*loop_num+j] +
values[i*loop_num+j];
023:                average[i*loop_num+j] = convert_float4(add_value);
024:      }
025:
026:      /* Insert zeros to 0th ～ (width-2)th element for 4 stocks*/
027:      for (i=0; i < width-1; i++) {
028:                average[i*loop_num+j] = (float4)(0.0f);
029:      }
030:
031:      /* Compute average of (width-1) ~ (length-1) elements for 4 stocks */
032:      for (i=width-1; i < length; i++) {
033:                average[i*loop_num+j] /= (float4)width;
034:      }
035: }
```

Since each compute unit is executing an instance of the kernel, which performs operations on 4 data sets, 8 data sets are processed over 2 compute units. In line 11, the value of "j" is either 0 or 1, which specifies the instance of the kernel as well as the data set to process. To take the change in the kernel into account, the host code must be changed as shown below in List 5.34.

List 5.34: Host code for calling the kernel in List 5.33

```
001: #include <stdlib.h>
002: #ifdef __APPLE__
003: #include <OpenCL/opencl.h>
004: #else
005: #include <CL/cl.h>
006: #endif
007: #include <stdio.h>
008:
009: #define NAME_NUM (8) /* Number of stocks */
010: #define DATA_NUM (21) /* Number of data to process for each stock */
011:
012: /* Read Stock data */
013: int stock_array_many[NAME_NUM*DATA_NUM]= {
```

```
014:        #include "stock_array_many.txt"
015: };
016:
017: /* Moving average width */
018: #define WINDOW_SIZE (13)
019:
020: #define MAX_SOURCE_SIZE (0x100000)
021:
022: int main(void)
023: {
024:        cl_platform_id platform_id = NULL;
025:        cl_uint ret_num_platforms;
026:        cl_device_id device_id = NULL;
027:        cl_uint ret_num_devices;
028:        cl_context context = NULL;
029:        cl_command_queue command_queue = NULL;
030:        cl_mem memobj_in = NULL;
031:        cl_mem memobj_out = NULL;
032:        cl_program program = NULL;
033:        cl_kernel kernel = NULL;
034:        size_t kernel_code_size;
035:        char *kernel_src_str;
036:        float *result;
037:        cl_int ret;
038:        FILE *fp;
039:
040:        int window_num = (int)WINDOW_SIZE;
041:        int point_num = NAME_NUM * DATA_NUM;
042:        int data_num = (int)DATA_NUM;
043:        int name_num = (int)NAME_NUM;
044:
045:        int i,j;
046:
047:        /* Allocate space to read in kernel code */
048:        kernel_src_str = (char *)malloc(MAX_SOURCE_SIZE);
049:
```

```
050:    /* Allocate space for the result on the host side */
051:    result = (float *)malloc(point_num*sizeof(float));
052:
053:    /* Get Platform*/
054:    ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
055:
056:    /* Get Device */
057:    ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
058:                &ret_num_devices);
059:
060:    /* Create Context */
061:    context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
062:
063:    /* Create Command Queue */
064:    command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
065:
066:    /* Read kernel source code */
067:    fp = fopen("moving_average_vec4_para.cl", "r");
068:    kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
069:    fclose(fp);
070:
071:    /* Create Program Object */
072:    program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
073:    (const size_t *)&kernel_code_size, &ret);
074:
075:    /* Compile kernel */
076:    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
077:
078:    /* Create kernel */
079:    kernel = clCreateKernel(program, "moving_average_vec4_para", &ret);
080:
081:    /* Create buffer for the input data on the device */
082:    memobj_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
083:    point_num * sizeof(int), NULL, &ret);
084:
085:    /* Create buffer for the result on the device */
```

```
086:      memobj_out = clCreateBuffer(context, CL_MEM_READ_WRITE,
087:      point_num * sizeof(float), NULL, &ret);
088:
089:      /* Copy input data to the global memory on the device*/
090:      ret = clEnqueueWriteBuffer(command_queue, memobj_in, CL_TRUE, 0,
091:                point_num * sizeof(int),
092:                stock_array_many, 0, NULL, NULL);
093:
094:      /* Set Kernel Arguments */
095:      ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj_in);
096:      ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobj_out);
097:      ret = clSetKernelArg(kernel, 2, sizeof(int), (void *)&data_num);
098:      ret = clSetKernelArg(kernel, 3, sizeof(int), (void *)&name_num);
099:      ret = clSetKernelArg(kernel, 4, sizeof(int), (void *)&window_num);
100:
101:
102:      /* Set parameters for data parallel processing (work item) */
103:      cl_uint work_dim = 1;
104:      size_t global_item_size[3];
105:      size_t local_item_size[3];
106:
107:      global_item_size[0] = 2; /* Global number of work items */
108:      local_item_size[0] = 1; /* Number of work items per work group */
109:      /* --> global_item_size[0] / local_item_size[0] becomes 2, which indirectly sets the
number of workgroups to 2*/
110:
111:      /* Execute Data Parallel Kernel */
112:      ret = clEnqueueNDRangeKernel(command_queue, kernel, work_dim, NULL,
113:                global_item_size, local_item_size,
114:                0, NULL, NULL);
115:
116:      /* Copy result from device to host */
117:      ret = clEnqueueReadBuffer(command_queue, memobj_out, CL_TRUE, 0,
118:                point_num * sizeof(float),
119:                result, 0, NULL, NULL);
120:
```

```
121:    /* OpenCL Object Finalization */
122:    ret = clReleaseKernel(kernel);
123:    ret = clReleaseProgram(program);
124:    ret = clReleaseMemObject(memobj_in);
125:    ret = clReleaseMemObject(memobj_out);
126:    ret = clReleaseCommandQueue(command_queue);
127:    ret = clReleaseContext(context);
128:
129:    /* Deallocate memory on the host */
130:    for (i=0; i < data_num; i++) {
131:            printf("result[%d]: ", i);
132:            for (j=0; j < name_num; j++) {
133:                    printf("%f, ", result[i*NAME_NUM+j]);
134:            }
135:            printf("¥n");
136:    }
137:
138:    /* Deallocate memory on the host */
139:    free(result);
140:    free(kernel_src_str);
141:
142:    return 0;
143: }
```

The data parallel processing is performed in lines 112 - 114, but notice the number of work groups are not explicitly specified. This is implied from the number of global work items (line 107) and the number of work items to process using one compute unite (line 108). It is also possible to execute multiple work items on 1 compute unit. This number should be equal to the number of processing elements within the compute unit for efficient data parallel execution.

Task Parallel Processing

We will now look at a different processing commonly performed in stock price analysis, known as the Golden Cross. The Golden Cross is a threshold point between a short-term moving average and a long-term moving average over time, which indicates a bull market on the horizon. This will be implemented in a task parallel manner.

Unlike data parallel programming, OpenCL does not have an API to explicitly specify the index space for batch processing. Each process need to be queued explicitly using the API function clEnqueueTask().

As mentioned in Chapter 4, the command queue only allows one task to be executed at a time unless explicitly specified to do otherwise. The host side must do one of the following:

• Allow out-of-order execution of the queued commands
• Create multiple command queues

Creating multiple command queues will allow for explicit scheduling of the tasks by the programmer. In this section, we will use the out-of-order method and allow the API to take care of the scheduling.

Allowing out-of-order execution in the command queue sends the next element in queue to an available compute unit. The out-of-order mode can be set as follows.

```
Command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &ret);
```

The 3rd argument CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE allows the command queue to send the next queued task to an available compute unit. This ends up in a scheduling of task parallel processing.

We will now perform task parallel processing to find the Golden Cross between a moving average over 13 weeks, and a moving average over 26 weeks. The two moving averages will be performed in a task parallel manner. We will use the code in List 5.29 (moving_average_vec4.cl), varying the 4th argument to 13 and 26 for each of the moving average to be performed. The host code becomes as shown in List 5.35.

List 5.35: Host code for task parallel processing of 2 moving averages

```
001: #include <stdlib.h>
002: #ifdef __APPLE__
003: #include <OpenCL/opencl.h>
004: #else
```

```
005: #include <CL/cl.h>
006: #endif
007: #include <stdio.h>
008:
009: #define NAME_NUM (4) /* Number of stocks */
010: #define DATA_NUM (100) /* Number of data to process for each stock */
011:
012: /* Read Stock data */
013: int stock_array_4[NAME_NUM*DATA_NUM]= {
014:       #include "stock_array_4.txt"
015: };
016:
017: /* Moving average width */
018: #define WINDOW_SIZE_13 (13)
019: #define WINDOW_SIZE_26 (26)
020:
021:
022: #define MAX_SOURCE_SIZE (0x100000)
023:
024:
025: int main(void)
026: {
027:       cl_platform_id platform_id = NULL;
028:       cl_uint ret_num_platforms;
029:       cl_device_id device_id = NULL;
030:       cl_uint ret_num_devices;
031:       cl_context context = NULL;
032:       cl_command_queue command_queue = NULL;
033:       cl_mem memobj_in = NULL;
034:       cl_mem memobj_out13 = NULL;
035:       cl_mem memobj_out26 = NULL;
036:       cl_program program = NULL;
037:       cl_kernel kernel13 = NULL;
038:       cl_kernel kernel26 = NULL;
039:       cl_event event13, event26;
040:       size_t kernel_code_size;
```

```
041:    char *kernel_src_str;
042:    float *result13;
043:    float *result26;
044:    cl_int ret;
045:    FILE *fp;
046:
047:    int window_num_13 = (int)WINDOW_SIZE_13;
048:    int window_num_26 = (int)WINDOW_SIZE_26;
049:    int point_num = NAME_NUM * DATA_NUM;
050:    int data_num = (int)DATA_NUM;
051:    int name_num = (int)NAME_NUM;
052:
053:    int i, j;
054:
055:    /* Allocate space to read in kernel code */
056:    kernel_src_str = (char *)malloc(MAX_SOURCE_SIZE);
057:
058:    /* Allocate space for the result on the host side */
059:    result13 = (float *)malloc(point_num*sizeof(float)); /* average over 13 weeks */
060:    result26 = (float *)malloc(point_num*sizeof(float)); /* average over 26 weeks */
061:
062:    /* Get Platform */
063:    ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
064:
065:    /* Get Device */
066:    ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
067:            &ret_num_devices);
068:
069:    /* Create Context */
070:    context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
071:
072:    /* Create Command Queue */
073:    command_queue = clCreateCommandQueue(context, device_id,
074:            CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &ret);
075:
076:    /* Read kernel source code */
```

```
077:    fp = fopen("moving_average_vec4.cl", "r");
078:    kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
079:    fclose(fp);
080:
081:    /* Create Program Object */
082:    program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
083:            (const size_t *)&kernel_code_size, &ret);
084:
085:    /* Compile kernel */
086:    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
087:
088:    /* Create kernel */
089:    kernel13 = clCreateKernel(program, "moving_average_vec4", &ret); /* 13 weeks */
090:    kernel26 = clCreateKernel(program, "moving_average_vec4", &ret); /* 26 weeks */
091:
092:    /* Create buffer for the input data on the device */
093:    memobj_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
094:            point_num * sizeof(int), NULL, &ret);
095:
096:    /* Create buffer for the result on the device */
097:    memobj_out13 = clCreateBuffer(context, CL_MEM_READ_WRITE,
098:            point_num * sizeof(float), NULL, &ret); /* 13 weeks */
099:    memobj_out26 = clCreateBuffer(context, CL_MEM_READ_WRITE,
100:            point_num * sizeof(float), NULL, &ret); /* 26 weeks */
101:
102:    /* Copy input data to the global memory on the device*/
103:    ret = clEnqueueWriteBuffer(command_queue, memobj_in, CL_TRUE, 0,
104:            point_num * sizeof(int),
105:            stock_array_4, 0, NULL, NULL);
106:
107:    /* Set Kernel Arguments (13 weeks) */
108:    ret = clSetKernelArg(kernel13, 0, sizeof(cl_mem), (void *)&memobj_in);
109:    ret = clSetKernelArg(kernel13, 1, sizeof(cl_mem), (void *)&memobj_out13);
110:    ret = clSetKernelArg(kernel13, 2, sizeof(int), (void *)&data_num);
111:    ret = clSetKernelArg(kernel13, 3, sizeof(int), (void *)&window_num_13);
112:
```

```
113:    /* Submit task to compute the moving average over 13 weeks */
114:    ret = clEnqueueTask(command_queue, kernel13, 0, NULL, &event13);
115:
116:    /* Set Kernel Arguments (26 weeks) */
117:    ret = clSetKernelArg(kernel26, 0, sizeof(cl_mem), (void *)&memobj_in);
118:    ret = clSetKernelArg(kernel26, 1, sizeof(cl_mem), (void *)&memobj_out26);
119:    ret = clSetKernelArg(kernel26, 2, sizeof(int), (void *)&data_num);
120:    ret = clSetKernelArg(kernel26, 3, sizeof(int), (void *)&window_num_26);
121:
122:    /* Submit task to compute the moving average over 26 weeks */
123:    ret = clEnqueueTask(command_queue, kernel26, 0, NULL, &event26);
124:
125:    /* Copy result for the 13 weeks moving average from device to host */
126:    ret = clEnqueueReadBuffer(command_queue, memobj_out13, CL_TRUE, 0,
127:            point_num * sizeof(float),
128:            result13, 1, &event13, NULL);
129:
130:    /* Copy result for the 26 weeks moving average from device to host */
131:    ret = clEnqueueReadBuffer(command_queue, memobj_out26, CL_TRUE, 0,
132:            point_num * sizeof(float),
133:            result26, 1, &event26, NULL);
134:
135:    /* OpenCL Object Finalization */
136:    ret = clReleaseKernel(kernel13);
137:    ret = clReleaseKernel(kernel26);
138:    ret = clReleaseProgram(program);
139:    ret = clReleaseMemObject(memobj_in);
140:    ret = clReleaseMemObject(memobj_out13);
141:    ret = clReleaseMemObject(memobj_out26);
142:    ret = clReleaseCommandQueue(command_queue);
143:    ret = clReleaseContext(context);
144:
145:    /* Display results */
146:    for (i=window_num_26-1; i < data_num; i++) {
147:            printf("result[%d]:", i );
148:            for (j=0; j < name_num; j++ ) {
```

```
149:                            /* Display whether the 13 week average is greater */
150:                            printf( "[%d] ", (result13[i*NAME_NUM+j] >
result26[i*NAME_NUM+j]) );
151:                    }
152:                    printf("¥n");
153:        }
154:
155:        /* Deallocate memory on the host */
156:        free(result13);
157:        free(result26);
158:        free(kernel_src_str);
159:
160:        return 0;
161: }
```

The Golden Cross Point can be determined by seeing where the displayed result changes from 0 to 1.

One thing to note in this example code is that the copying of the result from device to host should not occur until the processing is finished. Otherwise, the memory copy (clEnqueueReadBuffer) can occur during the processing, which would contain garbage.

Note in line 114 that "&event13" is passed back from the clEnqueueTask() command. This is known as an event object, which specifies whether this task has finished or not. This event object is seen again in line 128 to the clEnqueueReadBuffer() command, which specifies that the read command does not start execution until the computation of the moving average over 13 weeks is finished. This is done similarly for the moving average over 26 weeks, which is submitted in line 123 and written back in line 131.

In summary, the Enqueue API functions in general:

• Inputs event object(s) that specify what it must wait for until it can be executed
• Ouputs an event object that can be used to tell another task in queue to wait

The above two should be used to schedule the tasks in an efficient manner.

Case Study

This chapter will look at more practical applications than the sample codes that you have seen so far. You should have the knowledge required to write practical applications in OpenCL after reading this chapter.

FFT (Fast Fourier Transform)

The first application we will look at is a program that will perform band-pass filtering on an image. We will start by first explaining the process known as a Fourier Transform, which is required to perform the image processing.

Fourier Transform

"Fourier Transform" is a process that takes in samples of data, and outputs its frequency content. Its general application can be summarized as follows.

• Take in an audio signal and find its frequency content
• Take in an image data and find its spatial frequency content

The output of the Fourier Transform contains all of its input. A process known as the Inverse Fourier Transform can be used to retrieve the original signal.

The Fourier Transform is a process commonly used in many fields. Many programs use this procedure which is required for an equalizer, filter, compressor, etc.

The mathematical formula for the Fourier Transform process is shown below

$$X(\omega) = \int_{-\infty}^{\infty} x(t) \exp(-i\omega t) dt$$

The "i" is an imaginary number, and ω is the frequency in radians.

As you can see from its definition, the Fourier Transform operates on continuous data. However, continuous data means it contains infinite number of points with infinite precision. For this processing to be practical, it must be able to process a data set that contains a finite number of elements. Therefore, a process known as the Discrete Fourier Transform (DFT) was developed to estimate the Fourier Transform, which operates on a finite data set. The mathematical formula

is shown below.

$$X_j = \sum_{k=0}^{n-1} x_k \exp(-\frac{2\pi i}{n} jk) \qquad j = 0, 1, ..., n-1$$

This formula now allows processing of digital data with a finite number of samples. The problem with this method, however, is that its O(N^2). As the number of points is increased, the processing time grows by a power of 2.

Fast Fourier Transform

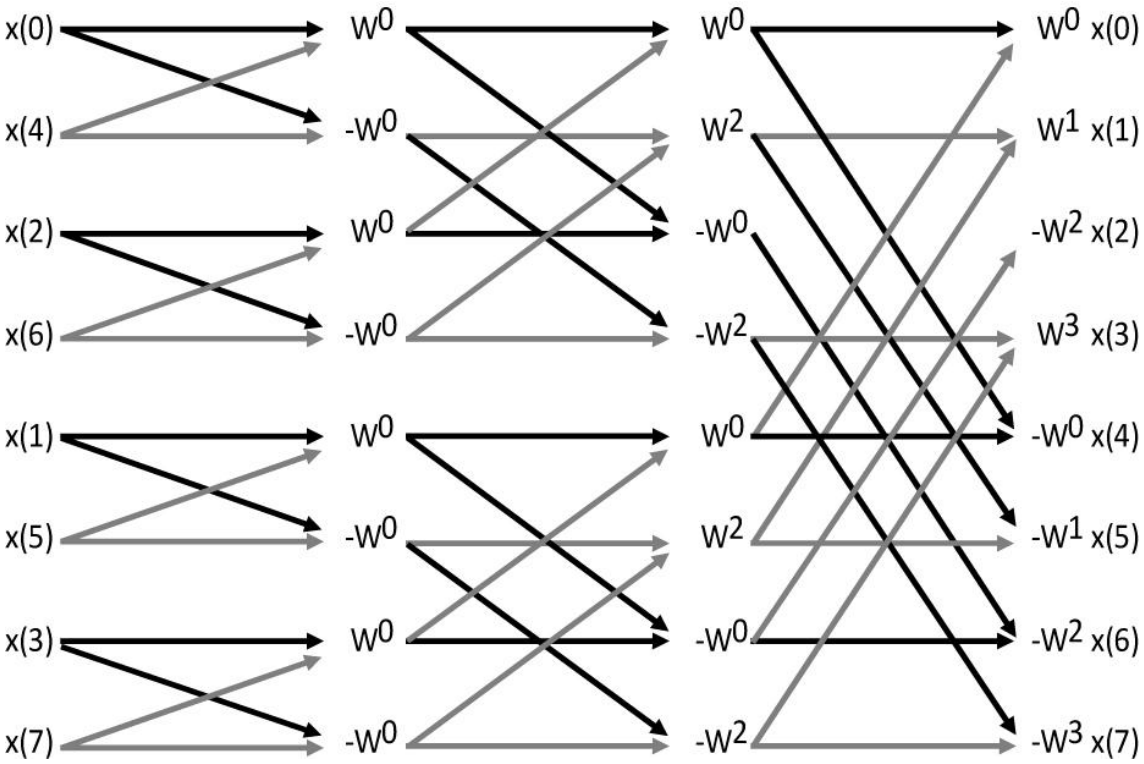
There exists an optimized implementation of the DFT, called the "Fast Fourier Transform (FFT)".

Many different implementations of FFT exist, so we will concentrate on the most commonly used Cooley-Tukey FFT algorithm. An entire book can be dedicated to explaining the FFT algorithm, so we will only explain the minimal amount required to implement the program.

The Cooley-Tukey algorithm takes advantage of the cyclical nature of the Fourier Transform, and solves the problem in O(N log N), by breaking up the DFT into smaller DFTs. The limitation with this algorithm is that the number of input samples must to be a power of 2. This limitation can be overcome by padding the input signal with zeros, or use in conjunction with another FFT algorithm that does not have this requirement. For simplicity, we will only use input signals whose length is a power of 2.

The core computation in this FFT algorithm is what is known as the "Butterfly Operation". The operation is performed on a pair of data samples at a time, whose signal flow graph is shown in Figure 6.2 below. The operation got its name due to the fact that each segment of this flow graph looks like a butterfly.

Figure 6.2: Butterfly Operation



The "W" seen in the signal flow graph is defined as below.

$$W = \exp(-\frac{2\pi i}{n})$$

Looking at Figure 6.2, you may notice the indices of the input are in a seemingly random order. We will not get into details on why this is done in this text except that it is an optimization method, but note that what is known as "bit-reversal" is performed on the indices of the input. The input order in binary is (000, 100, 010, 110, 001, 101, 011, 111). Notice that if you reverse the bit ordering of (100), you get (001). So the new input indices are in numerical order, except that the bits are reversed.

2-D FFT

As the previous section shows, the basic FFT algorithm is to be performed on 1 dimensional data. In order to take the FFT of an image, an FFT is taken row-wise and column-wise. Note that we are not dealing with time any more, but with spatial location.

When 2-D FFT is performed, the FFT is first taken for each row, transposed, and the FFT is again taken on this result. This is done for faster memory access, as the data is stored in row-major

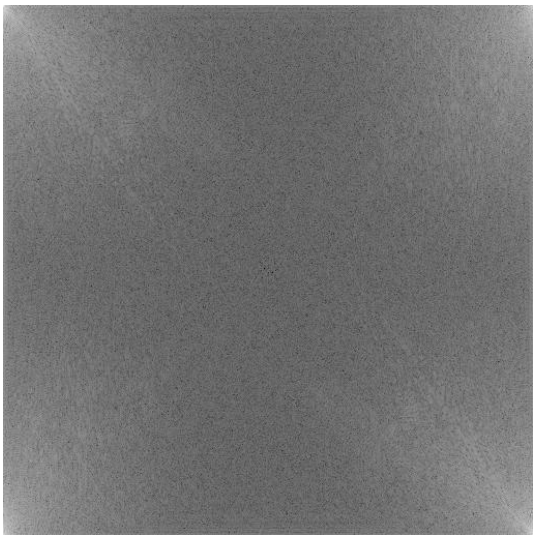
form. If transposition is not performed, interleaved accessing of memory occurs, which can greatly decrease speed of performance as the size of the image increases.

Figure 6.3(b) shows the result of taking a 2-D FFT of Figure 6.3(a).

Figure 6.3: 2-D FFT

(a) Original Image

(b) The result of taking a 2-D FFT



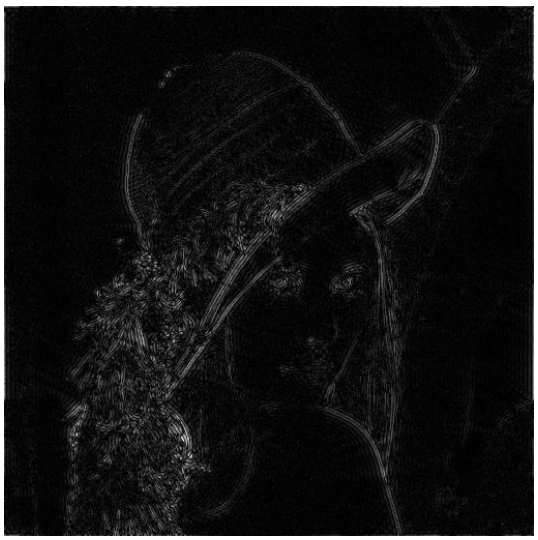
Bandpass Filtering and Inverse Fourier Transform

As stated earlier, the signal that has been transformed to the frequency domain via Fourier Transform can be transformed back using the Inverse Fourier Transform. Using this characteristic, it is possible to perform frequency-based filtering while in the frequency domain and transform back to the original domain. For example, the low-frequency components can be cut, which leaves the part of the image where a sudden change occurs. This is known as an "Edge Filter", and its result is shown in Figure 6.4a. If the high-frequency components are cut instead, the edges will be blurred, resulting in an image shown in Figure 6.4b. This is known as a "low-pass filter".

Figure 6.4: Edge Filter and Low-pass Filter

(a) Edged Filter

(b) Low-pass Filter



The mathematical formula for Inverse Discrete Fourier Transform is shown below.

$$x_j = \frac{1}{n}\sum_{k=0}^{n-1} X_k \exp(\frac{2\pi i}{n} jk) \qquad j = 0, 1, ..., n-1$$

Note its similarity with the DFT formula. The only differences are:

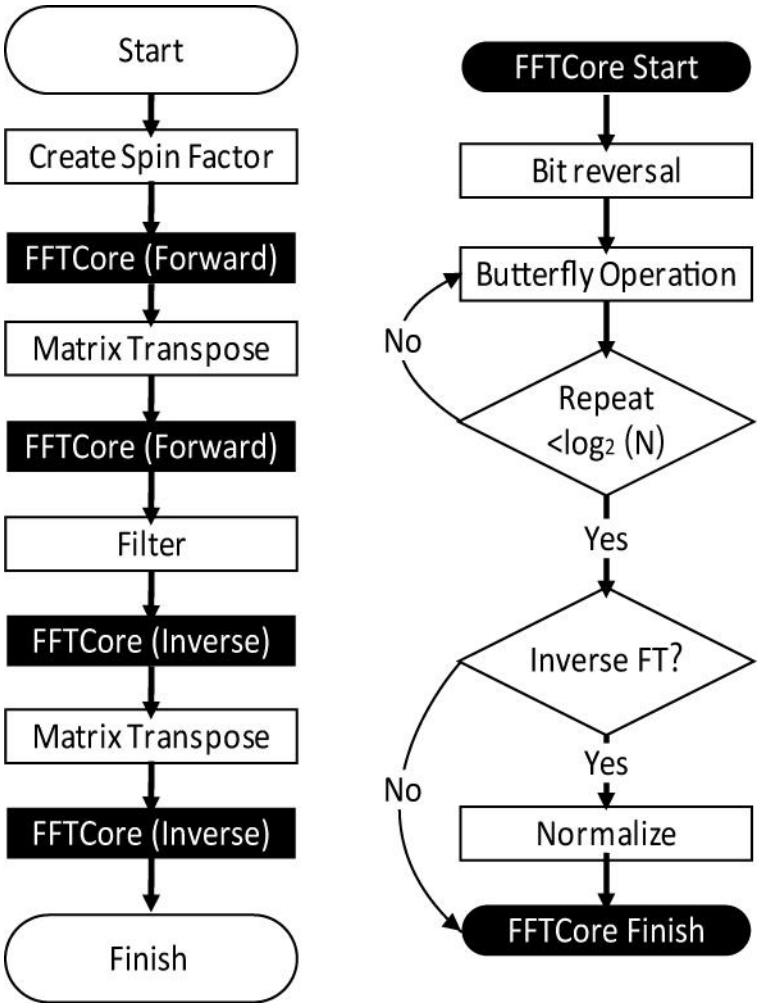
• Must be normalized by the number of samples
• The term within the exp() is positive.

The rest of the procedure is the same. Therefore, the same kernel can be used to perform either operation.

Overall Program Flow-chart

The overall program flow-chart is shown in Figure 6.5 below.

Figure 6.5: Program flow-chart



Each process is dependent on the previous process, so each of the steps must be followed in sequence. A kernel will be written for each of the processes in Figure 6.5.

Source Code Walkthrough

We will first show the entire source code for this program. List 6.1 is the kernel code, and List 6.2 is the host code.

List 6.1: Kernel Code

```
001: #define PI 3.14159265358979323846
002: #define PI_2 1.57079632679489661923
003:
004: __kernel void spinFact(__global float2* w, int n)
005: {
```

```
006:      unsigned int i = get_global_id(0);
007:
008:      float2 angle = (float2)(2*i*PI/(float)n,(2*i*PI/(float)n)+PI_2);
009:      w[i] = cos(angle);
010: }
011:
012: __kernel void bitReverse(__global float2 *dst, __global float2 *src, int m, int n)
013: {
014:      unsigned int gid = get_global_id(0);
015:      unsigned int nid = get_global_id(1);
016:
017:      unsigned int j = gid;
018:      j = (j & 0x55555555) << 1 | (j & 0xAAAAAAAA) >> 1;
019:      j = (j & 0x33333333) << 2 | (j & 0xCCCCCCCC) >> 2;
020:      j = (j & 0x0F0F0F0F) << 4 | (j & 0xF0F0F0F0) >> 4;
021:      j = (j & 0x00FF00FF) << 8 | (j & 0xFF00FF00) >> 8;
022:      j = (j & 0x0000FFFF) << 16 | (j & 0xFFFF0000) >> 16;
023:
024:      j >>= (32-m);
025:
026:      dst[nid*n+j] = src[nid*n+gid];
027: }
028:
029: __kernel void norm(__global float2 *x, int n)
030: {
031:      unsigned int gid = get_global_id(0);
032:      unsigned int nid = get_global_id(1);
033:
034:      x[nid*n+gid] = x[nid*n+gid] / (float2)((float)n, (float)n);
035: }
036:
037: __kernel void butterfly(__global float2 *x, __global float2* w, int m, int n, int iter, uint flag)
038: {
039:      unsigned int gid = get_global_id(0);
040:      unsigned int nid = get_global_id(1);
041:
```

```
042:      int butterflySize = 1 << (iter-1);
043:      int butterflyGrpDist = 1 << iter;
044:      int butterflyGrpNum = n >> iter;
045:      int butterflyGrpBase = (gid >> (iter-1))*(butterflyGrpDist);
046:      int butterflyGrpOffset = gid & (butterflySize-1);
047:
048:      int a = nid * n + butterflyGrpBase + butterflyGrpOffset;
049:      int b = a + butterflySize;
050:
051:      int l = butterflyGrpNum * butterflyGrpOffset;
052:
053:      float2 xa, xb, xbxx, xbyy, wab, wayx, wbyx, resa, resb;
054:
055:      xa = x[a];
056:      xb = x[b];
057:      xbxx = xb.xx;
058:      xbyy = xb.yy;
059:
060:      wab = as_float2(as_uint2(w[l]) ^ (uint2)(0x0, flag));
061:      wayx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x80000000, 0x0));
062:      wbyx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x0, 0x80000000));
063:
064:      resa = xa + xbxx*wab + xbyy*wayx;
065:      resb = xa - xbxx*wab + xbyy*wbyx;
066:
067:      x[a] = resa;
068:      x[b] = resb;
069: }
070:
071: __kernel void transpose(__global float2 *dst, __global float2* src, int n)
072: {
073:      unsigned int xgid = get_global_id(0);
074:      unsigned int ygid = get_global_id(1);
075:
076:      unsigned int iid = ygid * n + xgid;
077:      unsigned int oid = xgid * n + ygid;
```

```
078:
079:      dst[oid] = src[iid];
080: }
081:
082: __kernel void highPassFilter(__global float2* image, int n, int radius)
083: {
084:      unsigned int xgid = get_global_id(0);
085:      unsigned int ygid = get_global_id(1);
086:
087:      int2 n_2 = (int2)(n>>1, n>>1);
088:      int2 mask = (int2)(n-1, n-1);
089:
090:      int2 gid = ((int2)(xgid, ygid) + n_2) & mask;
091:
092:      int2 diff = n_2 - gid;
093:      int2 diff2 = diff * diff;
094:      int dist2 = diff2.x + diff2.y;
095:
096:      int2 window;
097:
098:      if (dist2 < radius*radius) {
099:           window = (int2)(0L, 0L);
100:      } else {
101:           window = (int2)(-1L, -1L);
102:      }
103:
104:      image[ygid*n+xgid] = as_float2(as_int2(image[ygid*n+xgid]) & window);
105: }
```

List 6.2: Host Code

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include <math.h>
004:
005: #ifdef __APPLE__
006: #include <OpenCL/opencl.h>
```

```
007: #else
008: #include <CL/cl.h>
009: #endif
010:
011: #include "pgm.h"
012:
013: #define PI 3.14159265358979
014:
015: #define MAX_SOURCE_SIZE (0x100000)
016:
017: #define AMP(a, b) (sqrt((a)*(a)+(b)*(b)))
018:
019: cl_device_id device_id = NULL;
020: cl_context context = NULL;
021: cl_command_queue queue = NULL;
022: cl_program program = NULL;
023:
024: enum Mode {
025:     forward = 0,
026:     inverse = 1
027: };
028:
029: int setWorkSize(size_t* gws, size_t* lws, cl_int x, cl_int y)
030: {
031:     switch(y) {
032:         case 1:
033:             gws[0] = x;
034:             gws[1] = 1;
035:             lws[0] = 1;
036:             lws[1] = 1;
037:             break;
038:         default:
039:             gws[0] = x;
040:             gws[1] = y;
041:             lws[0] = 1;
042:             lws[1] = 1;
```

```
043:                    break;
044:        }
045:
046:        return 0;
047: }
048:
049: int fftCore(cl_mem dst, cl_mem src, cl_mem spin, cl_int m, enum Mode direction)
050: {
051:        cl_int ret;
052:
053:        cl_int iter;
054:        cl_uint flag;
055:
056:        cl_int n = 1<<m;
057:
058:        cl_event kernelDone;
059:
060:        cl_kernel brev = NULL;
061:        cl_kernel bfly = NULL;
062:        cl_kernel norm = NULL;
063:
064:        brev = clCreateKernel(program, "bitReverse", &ret);
065:        bfly = clCreateKernel(program, "butterfly", &ret);
066:        norm = clCreateKernel(program, "norm", &ret);
067:
068:        size_t gws[2];
069:        size_t lws[2];
070:
071:        switch (direction) {
072:              case forward:flag = 0x00000000; break;
073:              case inverse:flag = 0x80000000; break;
074:        }
075:
076:        ret = clSetKernelArg(brev, 0, sizeof(cl_mem), (void *)&dst);
077:        ret = clSetKernelArg(brev, 1, sizeof(cl_mem), (void *)&src);
078:        ret = clSetKernelArg(brev, 2, sizeof(cl_int), (void *)&m);
```

```
079:      ret = clSetKernelArg(brev, 3, sizeof(cl_int), (void *)&n);
080:
081:      ret = clSetKernelArg(bfly, 0, sizeof(cl_mem), (void *)&dst);
082:      ret = clSetKernelArg(bfly, 1, sizeof(cl_mem), (void *)&spin);
083:      ret = clSetKernelArg(bfly, 2, sizeof(cl_int), (void *)&m);
084:      ret = clSetKernelArg(bfly, 3, sizeof(cl_int), (void *)&n);
085:      ret = clSetKernelArg(bfly, 5, sizeof(cl_uint), (void *)&flag);
086:
087:      ret = clSetKernelArg(norm, 0, sizeof(cl_mem), (void *)&dst);
088:      ret = clSetKernelArg(norm, 1, sizeof(cl_int), (void *)&n);
089:
090:      /* Reverse bit ordering */
091:      setWorkSize(gws, lws, n, n);
092:      ret = clEnqueueNDRangeKernel(queue, brev, 2, NULL, gws, lws, 0, NULL, NULL);
093:
094:      /* Perform Butterfly Operations*/
095:      setWorkSize(gws, lws, n/2, n);
096:      for (iter=1; iter <= m; iter++){
097:            ret = clSetKernelArg(bfly, 4, sizeof(cl_int), (void *)&iter);
098:            ret = clEnqueueNDRangeKernel(queue, bfly, 2, NULL, gws, lws, 0, NULL,
&kernelDone);
099:            ret = clWaitForEvents(1, &kernelDone);
100:      }
101:
102:      if (direction == inverse) {
103:            setWorkSize(gws, lws, n, n);
104:            ret = clEnqueueNDRangeKernel(queue, norm, 2, NULL, gws, lws, 0, NULL,
&kernelDone);
105:            ret = clWaitForEvents(1, &kernelDone);
106:      }
107:
108:      ret = clReleaseKernel(bfly);
109:      ret = clReleaseKernel(brev);
110:      ret = clReleaseKernel(norm);
111:
112:      return 0;
```

```
113: }
114:
115: int main()
116: {
117:     cl_mem xmobj = NULL;
118:     cl_mem rmobj = NULL;
119:     cl_mem wmobj = NULL;
120:     cl_kernel sfac = NULL;
121:     cl_kernel trns = NULL;
122:     cl_kernel hpfl = NULL;
123:
124:     cl_platform_id platform_id = NULL;
125:
126:     cl_uint ret_num_devices;
127:     cl_uint ret_num_platforms;
128:
129:     cl_int ret;
130:
131:     cl_float2 *xm;
132:     cl_float2 *rm;
133:     cl_float2 *wm;
134:
135:     pgm_t ipgm;
136:     pgm_t opgm;
137:
138:     FILE *fp;
139:     const char fileName[] = "./fft.cl";
140:     size_t source_size;
141:     char *source_str;
142:     cl_int i, j;
143:     cl_int n;
144:     cl_int m;
145:
146:     size_t gws[2];
147:     size_t lws[2];
148:
```

```
149:      /* Load kernel source code */
150:      fp = fopen(fileName, "r");
151:      if (!fp) {
152:           fprintf(stderr, "Failed to load kernel.¥n");
153:           exit(1);
154:      }
155:      source_str = (char *)malloc(MAX_SOURCE_SIZE);
156:      source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
157:      fclose( fp );
158:
159:      /* Read image */
160:      readPGM(&ipgm, "lena.pgm");
161:
162:      n = ipgm.width;
163:      m = (cl_int)(log((double)n)/log(2.0));
164:
165:      xm = (cl_float2 *)malloc(n * n * sizeof(cl_float2));
166:      rm = (cl_float2 *)malloc(n * n * sizeof(cl_float2));
167:      wm = (cl_float2 *)malloc(n / 2 * sizeof(cl_float2));
168:
169:      for (i=0; i < n; i++) {
170:           for (j=0; j < n; j++) {
171:                ((float*)xm)[(2*n*j)+2*i+0] = (float)ipgm.buf[n*j+i];
172:                ((float*)xm)[(2*n*j)+2*i+1] = (float)0;
173:           }
174:      }
175:
176:      /* Get platform/device    */
177:      ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
178:      ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);
179:
180:      /* Create OpenCL context */
181:      context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
182:
183:      /* Create Command queue */
```

```
184:      queue = clCreateCommandQueue(context, device_id, 0, &ret);
185:
186:      /* Create Buffer Objects */
187:      xmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, n*n*sizeof(cl_float2), NULL,
&ret);
188:      rmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, n*n*sizeof(cl_float2), NULL,
&ret);
189:      wmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, (n/2)*sizeof(cl_float2), NULL,
&ret);
190:
191:      /* Transfer data to memory buffer */
192:      ret = clEnqueueWriteBuffer(queue, xmobj, CL_TRUE, 0, n*n*sizeof(cl_float2), xm, 0,
NULL, NULL);
193:
194:      /* Create kernel program from source */
195:      program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const
size_t *)&source_size, &ret);
196:
197:      /* Build kernel program */
198:      ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
199:
200:      /* Create OpenCL Kernel */
201:      sfac = clCreateKernel(program, "spinFact", &ret);
202:      trns = clCreateKernel(program, "transpose", &ret);
203:      hpfl = clCreateKernel(program, "highPassFilter", &ret);
204:
205:      /* Create spin factor */
206:      ret = clSetKernelArg(sfac, 0, sizeof(cl_mem), (void *)&wmobj);
207:      ret = clSetKernelArg(sfac, 1, sizeof(cl_int), (void *)&n);
208:      setWorkSize(gws, lws, n/2, 1);
209:      ret = clEnqueueNDRangeKernel(queue, sfac, 1, NULL, gws, lws, 0, NULL, NULL);
210:
211:      /* Butterfly Operation */
212:      fftCore(rmobj, xmobj, wmobj, m, forward);
213:
214:      /* Transpose matrix */
```

```
215:      ret = clSetKernelArg(trns, 0, sizeof(cl_mem), (void *)&xmobj);
216:      ret = clSetKernelArg(trns, 1, sizeof(cl_mem), (void *)&rmobj);
217:      ret = clSetKernelArg(trns, 2, sizeof(cl_int), (void *)&n);
218:      setWorkSize(gws, lws, n, n);
219:      ret = clEnqueueNDRangeKernel(queue, trns, 2, NULL, gws, lws, 0, NULL, NULL);
220:
221:      /* Butterfly Operation */
222:      fftCore(rmobj, xmobj, wmobj, m, forward);
223:
224:      /* Apply high-pass filter */
225:      cl_int radius = n/8;
226:      ret = clSetKernelArg(hpfl, 0, sizeof(cl_mem), (void *)&rmobj);
227:      ret = clSetKernelArg(hpfl, 1, sizeof(cl_int), (void *)&n);
228:      ret = clSetKernelArg(hpfl, 2, sizeof(cl_int), (void *)&radius);
229:      setWorkSize(gws, lws, n, n);
230:      ret = clEnqueueNDRangeKernel(queue, hpfl, 2, NULL, gws, lws, 0, NULL, NULL);
231:
232:      /* Inverse FFT */
233:
234:      /* Butterfly Operation */
235:      fftCore(xmobj, rmobj, wmobj, m, inverse);
236:
237:      /* Transpose matrix */
238:      ret = clSetKernelArg(trns, 0, sizeof(cl_mem), (void *)&rmobj);
239:      ret = clSetKernelArg(trns, 1, sizeof(cl_mem), (void *)&xmobj);
240:      setWorkSize(gws, lws, n, n);
241:      ret = clEnqueueNDRangeKernel(queue, trns, 2, NULL, gws, lws, 0, NULL, NULL);
242:
243:      /* Butterfly Operation */
244:      fftCore(xmobj, rmobj, wmobj, m, inverse);
245:
246:      /* Read data from memory buffer */
247:      ret = clEnqueueReadBuffer(queue, xmobj, CL_TRUE, 0, n*n*sizeof(cl_float2), xm, 0,
NULL, NULL);
248:
249:      /*   */
```

```
250:      float* ampd;
251:      ampd = (float*)malloc(n*n*sizeof(float));
252:      for (i=0; i < n; i++) {
253:            for (j=0; j < n; j++) {
254:                  ampd[n*((i))+((j))] = (AMP(((float*)xm)[(2*n*i)+2*j],
((float*)xm)[(2*n*i)+2*j+1]));
255:            }
256:      }
257:      opgm.width = n;
258:      opgm.height = n;
259:      normalizeF2PGM(&opgm, ampd);
260:      free(ampd);
261:
262:      /* Write out image */
263:      writePGM(&opgm, "output.pgm");
264:
265:      /* Finalizations*/
266:      ret = clFlush(queue);
267:      ret = clFinish(queue);
268:      ret = clReleaseKernel(hpfl);
269:      ret = clReleaseKernel(trns);
270:      ret = clReleaseKernel(sfac);
271:      ret = clReleaseProgram(program);
272:      ret = clReleaseMemObject(xmobj);
273:      ret = clReleaseMemObject(rmobj);
274:      ret = clReleaseMemObject(wmobj);
275:      ret = clReleaseCommandQueue(queue);
276:      ret = clReleaseContext(context);
277:
278:      destroyPGM(&ipgm);
279:      destroyPGM(&opgm);
280:
281:      free(source_str);
282:      free(wm);
283:      free(rm);
284:      free(xm);
```

```
285:
286:      return 0;
287: }
```

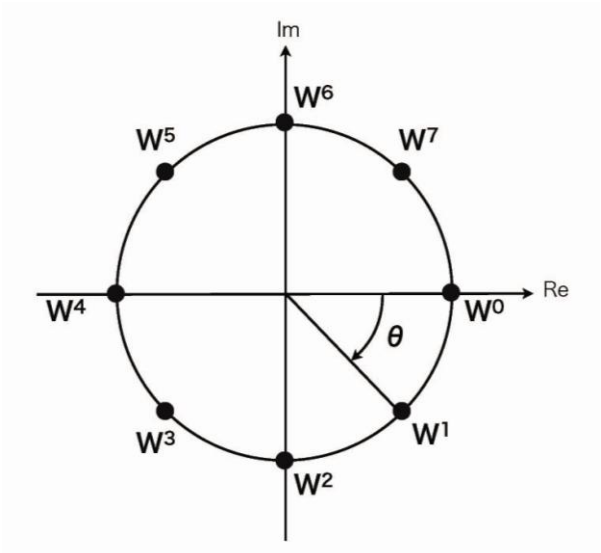
We will start by taking a look at each kernel.

List 6.3: Create Spin Factor

```
004: __kernel void spinFact(__global float2* w, int n)
005: {
006: unsigned int i = get_global_id(0);
007:
008: float2 angle = (float2)(2*i*PI/(float)n,(2*i*PI/(float)n)+PI_2);
009: w[i] = cos(angle);
010: }
```

The code in List 6.3 is used to pre-compute the value of the spin factor "w", which gets repeatedly used in the butterfly operation. The "w" is computed for radian angles that are multiples of $(2\pi/n)$, which is basically the real and imaginary components on the unit circle using cos() and -sin(). Note the shift by PI/2 in line 8 allows the cosine function to compute -sin(). This is done to utilize the SIMD unit on the OpenCL device if it has one.

Figure 6.6: Spin factor for n=8



The pre-computing of the values for "w" creates what is known as a "lookup table", which stores values to be used repeatedly on the memory. On some devices, such as the GPU, it may prove to

be faster if the same operation is performed each time, as it may be more expensive to access the memory.

List 6.4: Bit reversing

```
012: __kernel void bitReverse(__global float2 *dst, __global float2 *src, int m, int n)
013: {
014: unsigned int gid = get_global_id(0);
015: unsigned int nid = get_global_id(1);
016:
017: unsigned int j = gid;
018: j = (j & 0x55555555) << 1 | (j & 0xAAAAAAAA) >> 1;
019: j = (j & 0x33333333) << 2 | (j & 0xCCCCCCCC) >> 2;
020: j = (j & 0x0F0F0F0F) << 4 | (j & 0xF0F0F0F0) >> 4;
021: j = (j & 0x00FF00FF) << 8 | (j & 0xFF00FF00) >> 8;
022: j = (j & 0x0000FFFF) << 16 | (j & 0xFFFF0000) >> 16;
023:
024: j >>= (32-m);
025:
026: dst[nid*n+j] = src[nid*n+gid];
027: }
```

List 6.4 shows the kernel code for reordering the input data such that it is in the order of the bit-reversed index. Lines 18~22 performs the bit reversing of the inputs. The indices are correctly shifted in line 24, as the max index would otherwise be 2^32-1.

Also, note that a separate memory space must be allocated for the output on the global memory. These types of functions are known as an out-of-place function. This is done since the coherence of the data cannot be guaranteed if the input gets overwritten each time after processing. An alternative solution is shown in List 6.5, where each work item stores the output locally until all work items are finished, at which point the locally stored data is written to the input address space.

List 6.5: Bit reversing (Using synchronization)

```
012: __kernel void bitReverse(__global float2 *x, int m, int n)
013: {
014: unsigned int gid = get_global_id(0);
015: unsigned int nid = get_global_id(1);
```

```
016:
017: unsigned int j = gid;
018: j = (j & 0x55555555) << 1 | (j & 0xAAAAAAAA) >> 1;
019: j = (j & 0x33333333) << 2 | (j & 0xCCCCCCCC) >> 2;
020: j = (j & 0x0F0F0F0F) << 4 | (j & 0xF0F0F0F0) >> 4;
021: j = (j & 0x00FF00FF) << 8 | (j & 0xFF00FF00) >> 8;
022: j = (j & 0x0000FFFF) << 16 | (j & 0xFFFF0000) >> 16;
023:
024: j >>= (32-m);
025:
026: float2 val = x[nid*n+gid];
027:
028: SYNC_ALL_THREAD /* Synchronize all work-items */
029:
030: x[nid*n+j] = val;
031: }
```

However, OpenCL does not currently require the synchronization capability in its specification. It may be supported in the future, but depending on the device, these types of synchronization, especially when processing large amounts of data, can potentially decrease performance. If there is enough space on the device, the version on List 6.4 should be used.

List 6.6: Normalizing by the number of samples

```
029: __kernel void norm(__global float2 *x, int n)
030: {
031: unsigned int gid = get_global_id(0);
032: unsigned int nid = get_global_id(1);
033:
034: x[nid*n+gid] = x[nid*n+gid] / (float2)((float)n, (float)n);
035: }
```

The code in List 6.6 should be self-explanatory. It basically just dives the input by the value of "n". The operation is performed on a float2 type. Since the value of "n" is limited to a power of 2, you may be tempted to use shifting, but division by shifting is only possible for integer types. Shifting of a float value will result in unwanted results.

List 6.7: Butterfly operation

```
037: __kernel void butterfly(__global float2 *x, __global float2* w, int m, int n, int iter, uint flag)
038: {
039: unsigned int gid = get_global_id(0);
040: unsigned int nid = get_global_id(1);
041:
042: int butterflySize = 1 << (iter-1);
043: int butterflyGrpDist = 1 << iter;
044: int butterflyGrpNum = n >> iter;
045: int butterflyGrpBase = (gid >> (iter-1))*(butterflyGrpDist);
046: int butterflyGrpOffset = gid & (butterflySize-1);
047:
048: int a = nid * n + butterflyGrpBase + butterflyGrpOffset;
049: int b = a + butterflySize;
050:
051: int l = butterflyGrpNum * butterflyGrpOffset;
052:
053: float2 xa, xb, xbxx, xbyy, wab, wayx, wbyx, resa, resb;
054:
055: xa = x[a];
056: xb = x[b];
057: xbxx = xb.xx;
058: xbyy = xb.yy;
059:
060: wab = as_float2(as_uint2(w[l]) ^ (uint2)(0x0, flag));
061: wayx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x80000000, 0x0));
062: wbyx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x0, 0x80000000));
063:
064: resa = xa + xbxx*wab + xbyy*wayx;
065: resb = xa - xbxx*wab + xbyy*wbyx;
066:
067: x[a] = resa;
068: x[b] = resb;
069: }
```

The kernel for the butterfly operation, which performs the core of the FFT algorithm, is shown in

List 6.7 above. Each work item performs one butterfly operation for a pair of inputs. Therefore, (n * n)/2 work items are required.

Refer back to the signal flow graph for the butterfly operation in Figure 6.2. As the graph shows, the required inputs are the two input data and the spin factor, which can be derived from the "gid". The intermediate values required in mapping the "gid" to the input and output indices are computed in lines 42-46.

First, the variable "butterflySize" represents the difference in the indices to the data for the butterfly operation to be performed on. The "butterflySize" is 1 for the first iteration, and this value is doubled for each iteration.

Next, we need to know how the butterfly operation is grouped. Looking at the signal flow graph, we see that the crossed signal paths occur within independent groups. In the first iteration, the number of groups is the same as the number of butterfly operation to perform, but in the 2^{nd} iteration, it is split up into 2 groups. This value is stored in the variable butterflyGrpNum.

The differences of the indices between the groups are required as well. This is stored in the variable butterflyGrpDistance.

Next, we need to determine the indices to read from and to write to. The butterflyGrpBase variable contains the index to the first butterfly operation within the group. The butterflyGropOffset is the offset within the group. These are determined using the following formulas.

```
butterflyGrpBase = (gid / butterflySize) * butterflyGrpDistance);
butterflyGrpOffset = gid % butterflySize;
```

For our FFT implementation, we can replace the division and the mod operation with bit shifts, since we are assuming the value of n to be a power of 2.

Now the indices to perform the butterfly operation and the spin factor can be found. We will now go into the actual calculation.

Lines 55 ~ 65 are the core of the butterfly operation. Lines 60 ~ 62 takes the sign of the spin factor into account to take care of the computation for the real and imaginary components, as well

as the FFT and IFFT. Lines 64 ~ 65 are the actual operations, and Lines 67 ~ 68 stores the processed data.

List 6.8: Matrix Transpose

```
071: __kernel void transpose(__global float2 *dst, __global float2* src, int n)
072: {
073: unsigned int xgid = get_global_id(0);
074: unsigned int ygid = get_global_id(1);
075:
076: unsigned int iid = ygid * n + xgid;
077: unsigned int oid = xgid * n + ygid;
078:
079: dst[oid] = src[iid];
080: }
```

List 6.8 shows a basic implementation of the matrix transpose algorithm. We will not go into optimization of this algorithm, but this process can be speed up significantly by using local memory and blocking.

List 6.9: Filtering

```
082: __kernel void highPassFilter(__global float2* image, int n, int radius)
083: {
084: unsigned int xgid = get_global_id(0);
085: unsigned int ygid = get_global_id(1);
086:
087: int2 n_2 = (int2)(n>>1, n>>1);
088: int2 mask = (int2)(n-1, n-1);
089:
090: int2 gid = ((int2)(xgid, ygid) + n_2) & mask;
091:
092: int2 diff = n_2 - gid;
093: int2 diff2 = diff * diff;
094: int dist2 = diff2.x + diff2.y;
095:
096: int2 window;
097:
```

```
098: if (dist2 < radius*radius) {
099: window = (int2)(0L, 0L);
100: } else {
101: window = (int2)(-1L, -1L);
102: }
103:
104: image[ygid*n+xgid] = as_float2(as_int2(image[ygid*n+xgid]) & window);
105: }
```

List 6.9 is a kernel that filters an image based on frequency. As the kernel name suggests, the filter passes high frequencies and gets rid of the lower frequencies.

The spatial frequency obtained from the 2-D FFT shows the DC (direct current) component on the 4 edges of the XY coordinate system. A high pass filter can be created by cutting the frequency within a specified radius that includes these DC components. The opposite can be performed to create a low pass filter. In general, a high pass filter extracts the edges, and a low pass filter blurs the image.

Next, we will go over the host program. Most of what is being done is the same as for the OpenCL programs that we have seen so far. The main differences are:

• Multiple kernels are implemented
• Multiple memory objects are used, requiring appropriate data flow construction

Note that when a kernel is called repeatedly, the clSetKernelArg() only need to be changed for when an argument value changes from the previous time the kernel is called. For example, consider the butterfly operations being called in line 94 on the host side.

```
094:    /* Perform butterfly operations */
095:    setWorkSize(gws, lws, n/2, n);
096:    for (iter=1; iter <= m; iter++){
097:            ret = clSetKernelArg(bfly, 4, sizeof(cl_int), (void *)&iter);
098:            ret = clEnqueueNDRangeKernel(queue, bfly, 2, NULL, gws, lws, 0, NULL,
&kernelDone);
099:            ret = clWaitForEvents(1, &kernelDone);
100:    }
```

This butterfly operation kernel is executed log_2(n) times. The kernel must have its iteration number passed in as an argument. The kernel uses this value to compute which data to perform the butterfly operation on.

In this program, the data transfers between kernels occur via the memory objects. The types of kernels used can be classified to either in-place kernel or out-of-place kernel based on the data flow.

The in-place kernel uses the same memory object for both input and output, where the output data is written over the address space of the input data. The out-of-place kernel uses separate memory objects for input and output. The problem with these types of kernel is that it would require too much memory space on the device, and that a data transfer must occur between memory objects. Therefore, it would be wise to use as few memory objects as possible.

For this program, a memory object is required to store the pre-computed values of the spin factors. Since an out-of-place operation such as the matrix transposition exist, at least 2 additional memory objects are required. In fact, only these 3 memory objects are required for this program to run without errors due to race conditions.

To do this, the arguments to the kernel must be appropriately set using clSetKernelArg() for each call to the kernel. For example, when calling the out-of-place transpose operation which is called twice, the pointer to the memory object must be reversed the second time around.

The kernels in this program is called using clEnqueueNDRangeKernel() to operate on the data in a data parallel manner. When this is called, the number work items, whose values differ depending on the kernel used, must be set beforehand. To reduce careless errors and to make the code more readable, a setWorkSize() function is implemented in this program.

```
029:    int setWorkSize(size_t* gws, size_t* lws, cl_int x, cl_int y)
```

The program contains a set of procedures that are repeated numerous times, namely the bit reversal and the butterfly operation, for both FFT and IFFT. These procedures are all grouped into one function fftCore().

```
049:    int fftCore(cl_mem dst, cl_mem src, cl_mem spin, cl_int m, enum Mode direction)
```

This function takes memory objects for the input, output, and the spin factor, the sample number normalized by the log of 2, and the FFT direction. This function can be used for 1-D FFT if the arguments are appropriately set.

Lastly, we will briefly explain the outputting of the processed data to an image. The format used for the image is PGM, which is a gray-scale format that requires 8 bits for each pixel. The data structure is quite simple and intuitive to use. We will add a new file, called "pgm.h". This file will define numerous functions and structs to be used in our program. First is the pgm_t struct.

```
015: typedef struct _pgm_t {
016:       int width;
017:       int height;
018:       unsigned char *buf;
019: } pgm_t;
```

The width and the height gets the size of the image, and buf gets the image data. This can read in or written to a file using the following functions.

```
readPGM(pgm_t* pgm, const char*, filename);
writePGM(pgm_t* pgm, const char* filename);
```

Since each pixel of the PGM is stored as unsigned char, a conversion would need to be performed to represent the pixel information in 8 bits.

```
normalizePGM(pgm_t* pgm, double* data);
```

The full pgm.h file is shown below in List 6.10.

List 6.10: pgm.h

```
001: #ifndef _PGM_H_
002: #define _PGM_H_
003:
004: #include <math.h>
005: #include <string.h>
006:
```

```
007: #define PGM_MAGIC "P5"
008:
009: #ifdef _WIN32
010: #define STRTOK_R(ptr, del, saveptr) strtok_s(ptr, del, saveptr)
011: #else
012: #define STRTOK_R(ptr, del, saveptr) strtok_r(ptr, del, saveptr)
013: #endif
014:
015: typedef struct _pgm_t {
016:       int width;
017:       int height;
018:       unsigned char *buf;
019: } pgm_t;
020:
021: int readPGM(pgm_t* pgm, const char* filename)
022: {
023:       char *token, *pc, *saveptr;
024:       char *buf;
025:       size_t bufsize;
026:       char del[] = " ¥t¥n";
027:       unsigned char *dot;
028:
029:       long begin, end;
030:       int filesize;
031:       int i, w, h, luma, pixs;
032:
033:
034:       FILE* fp;
035:       if ((fp = fopen(filename, "rb"))==NULL) {
036:             fprintf(stderr, "Failed to open file¥n");
037:             return -1;
038:       }
039:
040:       fseek(fp, 0, SEEK_SET);
041:       begin = ftell(fp);
042:       fseek(fp, 0, SEEK_END);
```

```
043:     end = ftell(fp);
044:     filesize = (int)(end - begin);
045:
046:     buf = (char*)malloc(filesize * sizeof(char));
047:     fseek(fp, 0, SEEK_SET);
048:     bufsize = fread(buf, filesize * sizeof(char), 1, fp);
049:
050:     fclose(fp);
051:
052:     token = (char *)STRTOK_R(buf, del, &saveptr);
053:     if (strncmp(token, PGM_MAGIC, 2) != 0) {
054:          return -1;
055:     }
056:
057:     token = (char *)STRTOK_R(NULL, del, &saveptr);
058:     if (token[0] == '#' ) {
059:          token = (char *)STRTOK_R(NULL, "¥n", &saveptr);
060:          token = (char *)STRTOK_R(NULL, del, &saveptr);
061:     }
062:
063:     w = strtoul(token, &pc, 10);
064:     token = (char *)STRTOK_R(NULL, del, &saveptr);
065:     h = strtoul(token, &pc, 10);
066:     token = (char *)STRTOK_R(NULL, del, &saveptr);
067:     luma = strtoul(token, &pc, 10);
068:
069:     token = pc + 1;
070:     pixs = w * h;
071:
072:     pgm->buf = (unsigned char *)malloc(pixs * sizeof(unsigned char));
073:
074:     dot = pgm->buf;
075:
076:     for (i=0; i< pixs; i++, dot++) {
077:          *dot = *token++;
078:     }
```

```
079:
080:      pgm->width = w;
081:      pgm->height = h;
082:
083:      return 0;
084: }
085:
086: int writePGM(pgm_t* pgm, const char* filename)
087: {
088:      int i, w, h, pixs;
089:      FILE* fp;
090:      unsigned char* dot;
091:
092:      w = pgm->width;
093:      h = pgm->height;
094:      pixs = w * h;
095:
096:      if ((fp = fopen(filename, "wb+")) ==NULL) {
097:           fprintf(stderr, "Failed to open file¥n");
098:           return -1;
099:      }
100:
101:      fprintf (fp, "%s¥n%d %d¥n255¥n", PGM_MAGIC, w, h);
102:
103:      dot = pgm->buf;
104:
105:      for (i=0; i<pixs; i++, dot++) {
106:           putc((unsigned char)*dot, fp);
107:      }
108:
109:      fclose(fp);
110:
111:      return 0;
112: }
113:
114: int normalizeD2PGM(pgm_t* pgm, double* x)
```

```
115: {
116:     int i, j, w, h;
117:
118:     w = pgm->width;
119:     h = pgm->height;
120:
121:     pgm->buf = (unsigned char*)malloc(w * h * sizeof(unsigned char));
122:
123:     double min = 0;
124:     double max = 0;
125:     for (i=0; i < h; i++) {
126:         for (j=0; j < w; j++) {
127:             if (max < x[i*w+j])
128:             max = x[i*w+j];
129:             if (min > x[i*w+j])
130:                 min = x[i*w+j];
131:         }
132:     }
133:
134:     for (i=0; i < h; i++) {
135:         for (j=0; j < w; j++) {
136:             if((max-min)!=0)
137:                 pgm->buf[i*w+j] = (unsigned char)(255*(x[i*w+j]-min)/(max-min));
138:             else
139:                 pgm->buf[i*w+j]= 0;
140:         }
141:     }
142:
143:     return 0;
144: }
145:
146: int normalizeF2PGM(pgm_t* pgm, float* x)
147: {
148:     int i, j, w, h;
149:
150:     w = pgm->width;
```

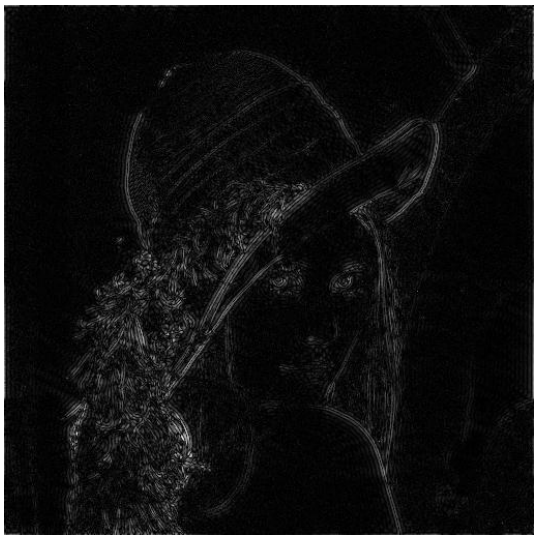
```
151:      h = pgm->height;
152:
153:      pgm->buf = (unsigned char*)malloc(w * h * sizeof(unsigned char));
154:
155:      float min = 0;
156:      float max = 0;
157:      for (i=0; i < h; i++) {
158:            for (j=0; j < w; j++) {
159:                  if (max < x[i*w+j])
160:                        max = x[i*w+j];
161:                  if (min > x[i*w+j])
162:                        min = x[i*w+j];
163:            }
164:      }
165:
166:      for (i=0; i < h; i++) {
167:            for (j=0; j < w; j++) {
168:                  if((max-min)!=0)
169:                        pgm->buf[i*w+j] = (unsigned char)(255*(x[i*w+j]-min)/(max-min));
170:                  else
171:                        pgm->buf[i*w+j]= 0;
172:            }
173:      }
174:
175:      return 0;
176: }
177:
178: int destroyPGM(pgm_t* pgm)
179: {
180:      if (pgm->buf) {
181:            free(pgm->buf);
182:      }
183:
184:      return 0;
185: }
186:
```

```
187: #endif /* _PGM_H_ */
```

When all the sources are compiled and executed on an image, the picture shown in Figure 6.7(a) becomes the picture shown in Figure 6.7(b). The edges in the original picture become white, while everything else becomes black.

Figure 6.7: Edge Detection

(a) Original Image (b) Edge Detection



Measuring Execution Time

OpenCL is an abstraction layer that allows the same code to be executed on different platforms, but this only guarantee that program can be executed. The speed of execution is dependent on the device, as well as the type of parallelism used. Therefore, in order to get the maximum performance, a device and parallelism dependent tuning must be performed.

In order to tune a program, the execution time must be measured, since it would otherwise be very difficult to see the result. We will now show how this can be done within the OpenCL framework for portability. Time measurement can be done in OpenCL, which is triggered by event objects associated with certain clEnqueue-type commands. This code is shown in List 6.11.

List 6.11: Time measurement using event objects

```
cl_context context;
```

```
cl_command_queue queue;
cl_event event;
cl_ulong start;
cl_ulong end;
…
queue = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &ret);
…
ret = clEnqueueWriteBuffer(queue, mobj, CL_TRUE, 0, MEM_SIZE, m, 0, NULL, &event);
…
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start,
NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
printf(" memory buffer write: %10.5f [ms]¥n", (end - start)/1000000.0);
```

The code on List 6.11 shows the code required to measure execution time. The code can be summarized as follows:

1. Command queue is created with the "CL_QUEUE_PROFILING_ENABLE" option
2. Kernel and memory object queuing is associated with events
3. clGetEventProfilingInfo() is used to get the start and end times.

One thing to note when getting the end time is that the kernel queuing is performed asynchronously. We need to make sure, in this case, that the kernel has actually finished executing before we get the end time. This can be done by using the clWaitForEvents() as in List 6.12.

List 6.12: Event synchronization

```
ret = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, gws, lws, 0, NULL, &event);
clWaitForEvents(1, &event);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start,
NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
```

The clWaitForEvents keeps the next line of the code to be executed until the specified events in the event list has finished its execution. The first argument gets the number of events to wait for, and the 2nd argument gets the pointer to the event list.

You should now be able to measure execution times using OpenCL.

Index Parameter Tuning

Recall that the number of work items and work groups had to be specified when executing data parallel kernels. This section focuses on what these values should be set to for optimal performance.

There is quite a bit of freedom when setting these values. For example, 512 work items can be split up into 2 work groups each having 256 work items, or 512 work groups each having 1 work item.

This raises the following questions:
1. What values are allowed for the number of work groups and work items?
2. What are the optimal values to use for the number of work groups and work items?

The first question can be answered using the clGetDeviceInfo() introduced in Section 5.1. The code is shown in List 6.13 below.

List 6.13: Get maximum values for the number of work groups and work items

```
cl_uint work_item_dim;
size_t work_item_sizes[3];
size_t work_group_size;
clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(cl_uint),
&work_item_dim, NULL);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_ITEM_SIZES, sizeof(work_item_sizes),
work_item_sizes, NULL);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t),
&work_group_size, NULL);
```

The first clGetDeviceInfo() gets the maximum number of dimensions allowed for the work item. This returns either 1, 2 or 3.

The second clGetDeviceInfo() gets the maximum values that can be used for each dimensions of the work item. The smallest value is [1,1,1].

The third clGetDeviceInfo() gets the maximum work item size that can be set for each work group. The smallest value for this is 1.

Running the above code using NVIDIA OpenCL would generate the results shown below.

```
Max work-item dimensions : 3
Max work-item sizes              : 512 512 64
Max work-group size              : 512
```

As mentioned before, the number of work items and the work groups must be set before executing a kernel. The global work item index and the local work item index each correspond to the work item ID of all the submitted jobs, and the work item ID within the work group.

For example, if the work item is 512 x 512, the following combination is possible. (gws=global work-item size, lws=local work-item size).

```
gws[] = {512,512,1}
lws[] = {1,1,1}
```

The following is also possible:

```
gws[] = {512,512,1}
lws[] = {16,16,1}
```

Yet another combination is:

```
gws[] = {512,512,1}
lws[] = {256,1,1}
```

The following example seems to not have any problems at a glance, but would result in an error, since the size of the work-group size exceeds that of the allowed size (32*32 = 1024 > 512).

```
gws[] = {512,512,1}
lws[] = {32,32,1}
```

Hopefully you have found the above to be intuitive. The real problem is figuring out the optimal

combination to use.

At this point, we need to look at the hardware architecture of the actual device. As discussed in Chapter 2, the OpenCL architecture assumes the device to contain compute unit(s), which is made up of several processing elements. The work-item corresponds to the processing element, and the work group corresponds to the compute unit. In other words, a work group gets executed on a compute unit, and a work-item gets executed on a processing element.

This implies that the knowledge of the number of compute units and processing elements is required in deducing the optimal combination to use for the local work group size. These can be found using clGetDeviceInfo(), as shown in List 6.14.

List 6.14: Find the number of compute units

```
cl_uint comput_unit = 0;
ret = clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint),
&compute_unit, NULL);
```

In NVIDIA GPUs, a compute unit corresponds to what is known as Streaming Multi-processor (SM). GT 200-series GPUs such as Tesla C1060 and GTX285 contain 30 compute units. A processing element corresponds to what is called CUDA cores. Each compute unit contains 8 processing elements, but 32 processes can logically be performed in parallel.

The following generations can be made from the above information:

• Processor elements can be used efficiently if the number of work items within a work group is a multiple of 32.
• All compute units can be used if the number of work groups is greater than 30.

We will now go back to the subject of FFT. We will vary the number of work-items per work group (local work-group size), and see how it affects the processing time. For simplicity, we will use a 512 x 512 image. Execution time was measured for 1, 16, 32, 64, 128, 256, and 512 local work-group size (Table 6.1).

Table 6.1: Execution time when varying lws (units in ms)

Process	1	16	32	64	128	256	512
membuf write	0.54	0.53	0.36	0.52	0.53	0.54	0.53

spinFactor	0.01	0.01	0.01	0.01	0.01	0.01	0.01
bitReverse	6.54	0.60	0.47	0.47	0.49	0.50	0.51
butterfly	37.75	3.12	2.87	3.02	3.41	3.60	3.88
normalize	2.86	0.20	0.12	0.09	0.09	0.10	0.10
transpose	2.83	1.53	1.49	1.42	1.07	0.95	0.96
highPassFilter	1.51	0.10	0.07	0.06	0.06	0.06	0.07
membuf read	0.62	0.57	0.57	0.61	0.58	0.56	0.58

As you can see, the optimal performance occurs when the local work-group size is a multiple of 32. This is due to the fact that 32 processes can logically be performed in parallel for each compute unit. The performance does not suffer significantly when the local work-group size is increased, since the NVIDIA GPU hardware performs the thread switching.

The FFT algorithm is a textbook model of a data parallel algorithm. We performed parameter tuning specifically for the NVIDIA GPU, but the optimal value would vary depending on the architecture of the device. For example, Apple's OpenCL for multi-core CPUs only allows 1 work-item for each work group. We can only assume that the parallelization is performed by the framework itself. At the time of this writing (December 2009), Mac OS X Snow Leopard comes with an auto-parallelization framework called the "Grand Central Dispatch", which lead us to assume a similar auto-parallelization algorithms are implemented to be used within the OpenCL framework.

We will now conclude our case study of the OpenCL implementation of FFT. Note that the techniques used so far are rather basic, but when combined wisely, a complex algorithm like the FFT can be implemented to be run efficiently over an OpenCL device. The implemented code should work over any platform where an OpenCL framework exists.

Mersenne Twister

This case study will use Mersenne Twister (MT) to generate pseudorandom numbers. This algorithm was developed by a professor at the Hiroshima University in Japan. MT has the following advantages.

• Long period
• Efficient use of memory
• High performance

• Good distribution properties

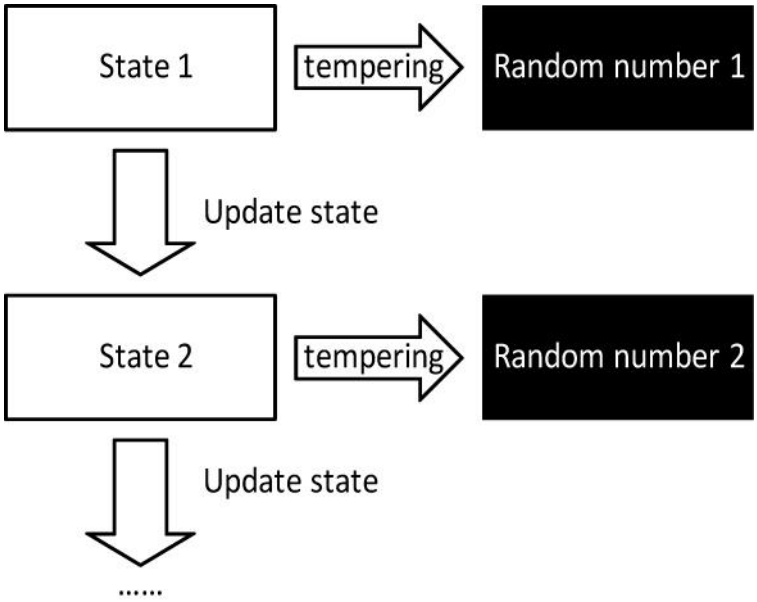
We will now implement this algorithm using OpenCL.

Parallelizing MT

A full understanding of the MT algorithm requires knowledge of advanced algebra, but the actual program itself is comprised of bit operations and memory accesses.

MT starts out with an initial state made up of an array of bits. By going through a process called "tempering" on this state, an array of random numbers is generated. The initial state then undergoes a set of operations, which creates another state. An array of random numbers is again generated from this state. In this manner, more random numbers are generated. A block diagram of the steps is shown in Figure 6.8.

Figure 6.8: MT processing



Studying the diagram in Figure 6.8, we see a dependency of the states, which gets in the way of parallel processing. To get around this problem, the following 2 methods come to mind.

• Parallelize the state change operation
The state is made up of a series of 32-bit words. Since the next state generation is done by operating on one 32-bit word at a time, each processing of the word can be parallelized. However, since the processing on each word does not require many instructions, and depending

on the cost of synchronization, performance may potentially suffer from parallelization. Also, since the number of processes that can be parallelized is dependent on the number of words that make up the state, it may not be enough to take advantage of the 100s of cores on GPUs. This type of processing would benefit more from using a SIMD unit that has low synchronization cost and designed to run few processes in parallel.

• Create numerous initial states, and process these in parallel
Since the dependency is between each state, generating an initial state for each processing element will allow for parallel execution of the MT algorithm. If the code is executed on Tesla C1060, for example, 240 MT can be processed in parallel. The generated random number, however, would not be the same as that generated sequentially from one state.

We will use the latter method, in order to run OpenCL on the GPU.

Dynamic Creator (dc)

NVIDIA's CUDA SDK sample contains a method called the Dynamic Creator (dc) which can be used to generate random numbers over multiple devices [18]. This section will show how to generate random numbers in OpenCL using "dc".

"dc" is a library that dynamically creates Mersenne Twisters parameters. The sample code in List 6.15 below shows how the get_mt_parameters() function in the "dc" library can be used. In this code, 32 sets of parameters are generated, but this can be changed by using a different value for the variable "n".

List 6.15: Using "dc" to generate Mersenne Twisters parameters

```
001: #include <stdio.h>
002: #include "dc.h"
003: int main(int argc, char **argv)
004: {
005: int i, n = 32;
006: mt_struct **mts;
007: init_dc(4172);
008: mts = get_mt_parameters(32,521,n);
009: for (i=0; i < n; i++) {
010: printf("{%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d}¥n",
```

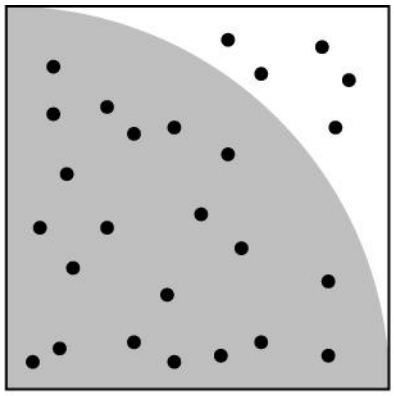
```
011:
mts[i]->aaa,mts[i]->mm,mts[i]->nn,mts[i]->rr,mts[i]->ww,mts[i]->wmask,mts[i]->umask,
012: mts[i]->lmask,mts[i]->shift0,mts[i]->shift1,mts[i]->shiftB,mts[i]->shiftC,
013: mts[i]->maskB,mts[i]->maskC);
014: }
015: return 0;
016: }
```

OpenCL MT

We will now implement MT in OpenCL. We will use the Mersenne Twister algorithm to generate random numbers, then use these numbers to compute the value of PI using the Monte Carlo method.

The Monte Carlo method can be used as follows. Assume that a quarter of a circle is contained in a square, such that the radius of the circle and the edge of the square are equivalent (Figure 6.9). Now points are chosen at random inside the square, which can be used to figure out if that point is within the quarter of a circle. Using the proportion of hit versus the number of points, and using the fact that the radius of the circle equals the edge of the square, the value of PI can be computed. Since the area of the quarter of a circle is PI * R^2 / 4, and the area of the square is R^2, the proportion of the area inside the circle is equal to PI / 4. When this is equated to the proportion found using the Monte Carlo method, solving for PI, we would get PI = 4 * (proportion).

Figure 6.9: Computing PI using Monte Carlo



Proportion of hit versus the number of points $\approx \frac{\pi R^2}{4}$

List 6.16 shows the kernel code and List 6.17 shows the host code for this program. The host program includes the commands required to measure the execution time, which we will use when we get to the optimization phase.

List 6.16: Kernel Code

```
001: typedef struct mt_struct_s {
002:     uint aaa;
003:     int mm,nn,rr,ww;
004:     uint wmask,umask,lmask;
005:     int shift0, shift1, shiftB, shiftC;
006:     uint maskB, maskC;
007: } mt_struct;
008:
009: /* Initialize state using a seed */
010: static void sgenrand_mt(uint seed, __global const mt_struct *mts, uint *state) {
011:     int i;
012:     for (i=0; i < mts->nn; i++) {
013:         state[i] = seed;
014:         seed = (1812433253 * (seed ^ (seed >> 30))) + i + 1;
015:     }
016:     for (i=0; i < mts->nn; i++)
017:         state[i] &= mts->wmask;
018: }
019:
020: /* Update state */
021: static void update_state(__global const mt_struct *mts, uint *st) {
022:     int n = mts->nn, m = mts->mm;
023:     uint aa = mts->aaa, x;
024:     uint uuu = mts->umask, lll = mts->lmask;
025:     int k,lim;
026:     lim = n - m;
027:     for (k=0; k < lim; k++) {
028:         x = (st[k]&uuu)|(st[k+1]&lll);
029:         st[k] = st[k+m] ^ (x>>1) ^ (x&1U ? aa : 0U);
030:     }
031:     lim = n - 1;
```

```
032:      for (; k < lim; k++) {
033:           x = (st[k]&uuu)|(st[k+1]&lll);
034:           st[k] = st[k+m-n] ^ (x>>1) ^ (x&1U ? aa : 0U);
035:      }
036:      x = (st[n-1]&uuu)|(st[0]&lll);
037:      st[n-1] = st[m-1] ^ (x>>1) ^ (x&1U ? aa : 0U);
038: }
039:
040: static inline void gen(__global uint *out, const __global mt_struct *mts, uint *state, int
num_rand) {
041:      int i, j, n, nn = mts->nn;
042:      n = (num_rand+(nn-1)) / nn;
043:      for (i=0; i < n; i++) {
044:           int m = nn;
045:           if (i == n-1) m = num_rand%nn;
046:           update_state(mts, state);
047:           for (j=0; j < m; j++) {
048:                /* Generate random numbers */
049:                uint x = state[j];
050:                x ^= x >> mts->shift0;
051:                x ^= (x << mts->shiftB) & mts->maskB;
052:                x ^= (x << mts->shiftC) & mts->maskC;
053:                x ^= x >> mts->shift1;
054:                out[i*nn + j] = x;
055:           }
056:      }
057: }
058:
059: __kernel void genrand(__global uint *out,__global mt_struct *mts,int num_rand){
060:      int gid = get_global_id(0);
061:      uint seed = gid*3;
062:      uint state[17];
063:      mts += gid; /* mts for this item */
064:      out += gid * num_rand; /* Output buffer for this item */
065:      sgenrand_mt(0x33ff*gid, mts, (uint*)state); /* Initialize random numbers */
066:      gen(out, mts, (uint*)state, num_rand); /* Generate random numbers */
```

```
067: }
068:
069: /* Count the number of points within the circle */
070: __kernel void calc_pi(__global uint *out, __global uint *rand, int num_rand) {
071:     int gid = get_global_id(0);
072:     int count = 0;
073:     int i;
074:
075:     rand += gid*num_rand;
076:
077:     for (i=0; i < num_rand; i++) {
078:         float x, y, len;
079:         x = ((float)(rand[i]>>16))/65535.0f; /* x coordinate */
080:         y = ((float)(rand[i]&0xffff))/65535.0f; /* y coordinate */
081:         len = (x*x + y*y); /* Distance from the origin */
082:
083:         if (len < 1) { /* sqrt(len) < 1 = len < 1 */
084:             count++;
085:         }
086:     }
087:
088:     out[gid] = count;
089: }
```

List 6.17: Host Code

```
001: #include <stdlib.h>
002: #include <CL/cl.h>
003: #include <stdio.h>
004: #include <math.h>
005:
006: typedef struct mt_struct_s {
007:     cl_uint aaa;
008:     cl_int mm,nn,rr,ww;
009:     cl_uint wmask,umask,lmask;
010:     cl_int shift0, shift1, shiftB, shiftC;
011:     cl_uint maskB, maskC;
```

```
012: } mt_struct;
013:
014: mt_struct mts[] = { /* Parameters generated by the Dynamic Creater */
015:     {-822663744,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-1514742400,-2785280},
016:     {-189104639,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-1263215232,-2785280},
017:
{-1162865726,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-1242112640,-2686976},
018:     {-1028775805,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-688432512,-2785280},
019:     {-1154537980,8,17,23,32,-1,-8388608,8388607,12,18,7,15,1702190464,-2785280},
020:     {-489740155,8,17,23,32,-1,-8388608,8388607,12,18,7,15,938827392,-34275328},
021:     {-2145422714,8,17,23,32,-1,-8388608,8388607,12,18,7,15,1722112896,-2818048},
022:     {-1611140857,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-173220480,-2785280},
023:     {-1514624952,8,17,23,32,-1,-8388608,8388607,12,18,7,15,1970625920,-2785280},
024:     {-250871863,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-605200512,-45776896},
025:     {-504393846,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-1535844992,-2818048},
026:     {-96493045,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-608739712,-36339712},
027:     {-291834292,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-151201152,-1736704},
028:     {-171838771,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-411738496,-36372480},
029:     {-588451954,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-1494951296,-2785280},
030:     {-72754417,8,17,23,32,-1,-8388608,8388607,12,18,7,15,1727880064,-688128},
031:
{-1086664688,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-1230685312,-8552448},
032:
{-558849839,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-1946722688,-34242560},
033:     {-1723207278,8,17,23,32,-1,-8388608,8388607,12,18,7,15,2000501632,-1114112},
034:     {-1779704749,8,17,23,32,-1,-8388608,8388607,12,18,7,15,1004886656,-2785280},
035:     {-2061554476,8,17,23,32,-1,-8388608,8388607,12,18,7,15,1958042496,-2785280},
036:     {-1524638763,8,17,23,32,-1,-8388608,8388607,12,18,7,15,653090688,-34177024},
037:     {-474740330,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-1263215232,-3833856},
038:     {-1001052777,8,17,23,32,-1,-8388608,8388607,12,18,7,15,921000576,-196608},
039:     {-1331085928,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-453715328,-2785280},
040:     {-883314023,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-184755584,-2785280},
041:     {-692780774,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-638264704,-2785280},
042:
{-1009388965,8,17,23,32,-1,-8388608,8388607,12,18,7,15,1001217664,-36339712},
043:     {-690295716,8,17,23,32,-1,-8388608,8388607,12,18,7,15,922049152,-1769472},
```

```
044:      {-1787515619,8,17,23,32,-1,-8388608,8388607,12,18,7,15,1970625920,-688128},
045:      {-1172830434,8,17,23,32,-1,-8388608,8388607,12,18,7,15,720452480,-2654208},
046:
{-746112289,8,17,23,32,-1,-8388608,8388607,12,18,7,15,2000509824,-2785280},};
047:
048: #define MAX_SOURCE_SIZE (0x100000)
049:
050: int main()
051: {
052:      cl_int num_rand = 4096*256; /* The number of random numbers generated using one
generator */
053:      int count_all, i, num_generator = sizeof(mts)/sizeof(mts[0]); /* The number of
generators */
054:      double pi;
055:      cl_platform_id platform_id = NULL;
056:      cl_uint ret_num_platforms;
057:      cl_device_id device_id = NULL;
058:      cl_uint ret_num_devices;
059:      cl_context context = NULL;
060:      cl_command_queue command_queue = NULL;
061:      cl_program program = NULL;
062:      cl_kernel kernel_mt = NULL, kernel_pi = NULL;
063:      size_t kernel_code_size;
064:      char *kernel_src_str;
065:      cl_uint *result;
066:      cl_int ret;
067:      FILE *fp;
068:      cl_mem rand, count;
069:      size_t global_item_size[3], local_item_size[3];
070:      cl_mem dev_mts;
071:      cl_event ev_mt_end, ev_pi_end, ev_copy_end;
072:      cl_ulong prof_start, prof_mt_end, prof_pi_end, prof_copy_end;
073:
074:      clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
075:      clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
076:           &ret_num_devices);
```

```
077:      context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
078:      result = (cl_uint*)malloc(sizeof(cl_uint)*num_generator);
079:
080:      command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_PROFILING_ENABLE, &ret);
081:      fp = fopen("mt.cl", "r");
082:      kernel_src_str = (char*)malloc(MAX_SOURCE_SIZE);
083:      kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
084:      fclose(fp);
085:
086:      /* Create output buffer */
087:      rand = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(cl_uint)*num_rand*num_generator, NULL, &ret);
088:      count = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(cl_uint)*num_generator, NULL, &ret);
089:
090:      /* Build Program*/
091:      program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
092:            (const size_t *)&kernel_code_size, &ret);
093:      clBuildProgram(program, 1, &device_id, "", NULL, NULL);
094:      kernel_mt = clCreateKernel(program, "genrand", &ret);
095:      kernel_pi = clCreateKernel(program, "calc_pi", &ret);
096:
097:      /* Create input parameter */
098:      dev_mts = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(mts), NULL, &ret);
099:      clEnqueueWriteBuffer(command_queue, dev_mts, CL_TRUE, 0, sizeof(mts), mts, 0,
NULL, NULL);
100:
101:      /* Set Kernel Arguments */
102:      clSetKernelArg(kernel_mt, 0, sizeof(cl_mem), (void*)&rand); /* Random numbers
(output of genrand) */
103:      clSetKernelArg(kernel_mt, 1, sizeof(cl_mem), (void*)&dev_mts); /* MT parameter
(input to genrand) */
104:      clSetKernelArg(kernel_mt, 2, sizeof(num_rand), &num_rand); /* Number of random
numbers to generate */
105:
```

```
106:	clSetKernelArg(kernel_pi, 0, sizeof(cl_mem), (void*)&count); /* Counter for points within circle (output of calc_pi) */
107:	clSetKernelArg(kernel_pi, 1, sizeof(cl_mem), (void*)&rand); /* Random numbers (input to calc_pi) */
108:	clSetKernelArg(kernel_pi, 2, sizeof(num_rand), &num_rand); /* Number of random numbers used */
109:
110:	global_item_size[0] = num_generator; global_item_size[1] = 1; global_item_size[2] = 1;
111:	local_item_size[0] = num_generator; local_item_size[1] = 1; local_item_size[2] = 1;
112:
113:	/* Create a random number array */
114:	clEnqueueNDRangeKernel(command_queue, kernel_mt, 1, NULL, global_item_size, local_item_size, 0, NULL, &ev_mt_end);
115:
116:	/* Compute PI */
117:	clEnqueueNDRangeKernel(command_queue, kernel_pi, 1, NULL, global_item_size, local_item_size, 0, NULL, &ev_pi_end);
118:
119:	/* Get result */
120:	clEnqueueReadBuffer(command_queue, count, CL_TRUE, 0, sizeof(cl_uint)*num_generator, result, 0, NULL, &ev_copy_end);
121:
122:	/* Average the values of PI */
123:	count_all = 0;
124:	for (i=0; i < num_generator; i++) {
125:		count_all += result[i];
126:	}
127:
128:	pi = ((double)count_all)/(num_rand * num_generator) * 4;
129:	printf("pi = %f¥n", pi);
130:
131:	/* Get execution time info */
132:	clGetEventProfilingInfo(ev_mt_end, CL_PROFILING_COMMAND_QUEUED, sizeof(cl_ulong), &prof_start, NULL);
133:	clGetEventProfilingInfo(ev_mt_end, CL_PROFILING_COMMAND_END, sizeof(cl_ulong),
```

```
&prof_mt_end, NULL);
134:      clGetEventProfilingInfo(ev_pi_end, CL_PROFILING_COMMAND_END, sizeof(cl_ulong),
&prof_pi_end, NULL);
135:      clGetEventProfilingInfo(ev_copy_end, CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &prof_copy_end, NULL);
136:
137:      printf(" mt: %f[ms]¥n"
138:            " pi: %f[ms]¥n"
139:            " copy: %f[ms]¥n",
140:            (prof_mt_end - prof_start)/(1000000.0),
141:            (prof_pi_end - prof_mt_end)/(1000000.0),
142:            (prof_copy_end - prof_pi_end)/(1000000.0));
143:
144:      clReleaseEvent(ev_mt_end);
145:      clReleaseEvent(ev_pi_end);
146:      clReleaseEvent(ev_copy_end);
147:
148:      clReleaseMemObject(rand);
149:      clReleaseMemObject(count);
150:      clReleaseKernel(kernel_mt);
151:      clReleaseKernel(kernel_pi);
152:      clReleaseProgram(program);
153:      clReleaseCommandQueue(command_queue);
154:      clReleaseContext(context);
155:      free(kernel_src_str);
156:      free(result);
157:      return 0;
158: }
```

We will start by looking at the kernel code

```
009: /* Initialize state using a seed */
010: static void sgenrand_mt(uint seed, __global const mt_struct *mts, uint *state) {
011:      int i;
012:      for (i=0; i < mts->nn; i++) {
013:            state[i] = seed;
```

```
014:            seed = (1812433253 * (seed ^ (seed >> 30))) + i + 1;
015:      }
016:      for (i=0; i < mts->nn; i++)
017:            state[i] &= mts->wmask;
018: }
```

The state bit array is being initialized. The above code is actually copied from the original code used for MT.

```
020: /* Update state */
021: static void update_state(__global const mt_struct *mts, uint *st) {
022:      int n = mts->nn, m = mts->mm;
023:      uint aa = mts->aaa, x;
024:      uint uuu = mts->umask, lll = mts->lmask;
025:      int k,lim;
026:      lim = n - m;
027:      for (k=0; k < lim; k++) {
028:            x = (st[k]&uuu)|(st[k+1]&lll);
029:            st[k] = st[k+m] ^ (x>>1) ^ (x&1U ? aa : 0U);
030:      }
031:      lim = n - 1;
032:      for (; k < lim; k++) {
033:            x = (st[k]&uuu)|(st[k+1]&lll);
034:            st[k] = st[k+m-n] ^ (x>>1) ^ (x&1U ? aa : 0U);
035:      }
036:      x = (st[n-1]&uuu)|(st[0]&lll);
037:      st[n-1] = st[m-1] ^ (x>>1) ^ (x&1U ? aa : 0U);
038: }
```

The above code updates the state bits.

```
040: static inline void gen(__global uint *out, const __global mt_struct *mts, uint *state, int
num_rand) {
041:      int i, j, n, nn = mts->nn;
042:      n = (num_rand+(nn-1)) / nn;
043:      for (i=0; i < n; i++) {
```

```
044:            int m = nn;
045:            if (i == n-1) m = num_rand%nn;
046:            update_state(mts, state);
047:            for (j=0; j < m; j++) {
048:                 /* Generate random numbers */
049:                 uint x = state[j];
050:                 x ^= x >> mts->shift0;
051:                 x ^= (x << mts->shiftB) & mts->maskB;
052:                 x ^= (x << mts->shiftC) & mts->maskC;
053:                 x ^= x >> mts->shift1;
054:                 out[i*nn + j] = x;
055:            }
056:       }
057: }
```

The code above generates "num_rand" random numbers. The update_state() is called to update the state bits, and a random number is generated from the state bits.

```
059: __kernel void genrand(__global uint *out,__global mt_struct *mts,int num_rand){
060:       int gid = get_global_id(0);
061:       uint seed = gid*3;
062:       uint state[17];
063:       mts += gid; /* mts for this item */
064:       out += gid * num_rand; /* Output buffer for this item */
065:       sgenrand_mt(0x33ff*gid, mts, (uint*)state); /* Initialize random numbers */
066:       gen(out, mts, (uint*)state, num_rand); /* Generate random numbers */
067: }
```

The above shows the kernel that generates random numbers by calling the above functions. The kernel takes the output address, MT parameters, and the number of random number to generate as arguments. The "mts" and "out" are pointers to the beginning of the spaces allocated to be used by all work-items. The global ID is used to calculate the addresses to be used by this kernel.

```
069: /* Count the number of points within the circle */
070: __kernel void calc_pi(__global uint *out, __global uint *rand, int num_rand) {
071:       int gid = get_global_id(0);
```

```
072:    int count = 0;
073:    int i;
074:
075:    rand += gid*num_rand;
076:
077:    for (i=0; i < num_rand; i++) {
078:        float x, y, len;
079:        x = ((float)(rand[i]>>16))/65535.0f; /* x coordinate */
080:        y = ((float)(rand[i]&0xffff))/65535.0f; /* y coordinate */
081:        len = (x*x + y*y); /* Distance from the origin */
082:
083:        if (len < 1) { /* sqrt(len) < 1 = len < 1 */
084:            count++;
085:        }
086:    }
087:
088:    out[gid] = count;
089: }
```

The above code counts the number of times the randomly chosen point within the square is inside the section of the circle. Since a 32-bit random number is generated, the upper 16 bits are used for the x coordinate, and the lower 16 bits are used for the y coordinate. If the distance from the origin is less than 1, then the chosen point is inside the circle.

Next we will take a look at the host code.

```
014: mt_struct mts[] = { /* Parameters generated by the Dynamic Creater */
015:    {-822663744,8,17,23,32,-1,-8388608,8388607,12,18,7,15,-1514742400,-2785280},
...
046:
{-746112289,8,17,23,32,-1,-8388608,8388607,12,18,7,15,2000509824,-2785280},};
```

The above is an array of struct for the MT parameters generated using "dc".

```
075:    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
076:        &ret_num_devices);
```

```
077:      context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
078:      result = (cl_uint*)malloc(sizeof(cl_uint)*num_generator);
079:
080:      command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_PROFILING_ENABLE, &ret);
081:      fp = fopen("mt.cl", "r");
082:      kernel_src_str = (char*)malloc(MAX_SOURCE_SIZE);
083:      kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
084:      fclose(fp);
085:
086:      /* Create output buffer */
087:      rand = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(cl_uint)*num_rand*num_generator, NULL, &ret);
088:      count = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(cl_uint)*num_generator, NULL, &ret);
089:
090:      /* Build Program*/
091:      program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
092:            (const size_t *)&kernel_code_size, &ret);
093:      clBuildProgram(program, 1, &device_id, "", NULL, NULL);
094:      kernel_mt = clCreateKernel(program, "genrand", &ret);
095:      kernel_pi = clCreateKernel(program, "calc_pi", &ret);
096:
097:      /* Create input parameter */
098:      dev_mts = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(mts), NULL, &ret);
099:      clEnqueueWriteBuffer(command_queue, dev_mts, CL_TRUE, 0, sizeof(mts), mts, 0,
NULL, NULL);
```

The code above transfers the MT parameters generated by "dc" to global memory.

```
101:      /* Set Kernel Arguments */
102:      clSetKernelArg(kernel_mt, 0, sizeof(cl_mem), (void*)&rand); /* Random numbers
(output of genrand) */
103:      clSetKernelArg(kernel_mt, 1, sizeof(cl_mem), (void*)&dev_mts); /* MT parameter
(input to genrand) */
104:      clSetKernelArg(kernel_mt, 2, sizeof(num_rand), &num_rand); /* Number of random
```

```
numbers to generate */
105:
106:      clSetKernelArg(kernel_pi, 0, sizeof(cl_mem), (void*)&count); /* Counter for points
within circle (output of calc_pi) */
107:      clSetKernelArg(kernel_pi, 1, sizeof(cl_mem), (void*)&rand); /* Random numbers
(input to calc_pi) */
108:      clSetKernelArg(kernel_pi, 2, sizeof(num_rand), &num_rand); /* Number of random
numbers used */
```

The code segment above sets the arguments to the kernels "kernel_mt", which generates random numbers, and "kernel_pi", which counts the number of points within the circle.

```
110:       global_item_size[0] = num_generator; global_item_size[1] = 1; global_item_size[2]
= 1;
111:      local_item_size[0] = num_generator; local_item_size[1] = 1; local_item_size[2] = 1;
112:
113:      /* Create a random number array */
114:       clEnqueueNDRangeKernel(command_queue, kernel_mt, 1, NULL, global_item_size,
local_item_size, 0, NULL, &ev_mt_end);
115:
116:      /* Compute PI */
117:       clEnqueueNDRangeKernel(command_queue, kernel_pi, 1, NULL, global_item_size,
local_item_size, 0, NULL, &ev_pi_end);
```

The global item size and the local item size are both set to the number of random numbers. This means that the kernels are executed such that only 1 work-group is created, which performs the processing on one compute unit.

```
122:      /* Average the values of PI */
123:      count_all = 0;
124:      for (i=0; i < num_generator; i++) {
125:            count_all += result[i];
126:      }
127:
128:      pi = ((double)count_all)/(num_rand * num_generator) * 4;
129:      printf("pi = %f¥n", pi);
```

The value of PI is calculated by tallying up the number of points chosen within the circle by each thread to calculate the proportion of hits. As discussed earlier, the value of PI is this proportion multiplied by 4.

Parallelization

Now we will go through the code in the previous section, and use parallelization where we can. Running the code as is on Core i7-920 + Tesla C 1060 environment, the execution time using different OpenCL implementations for kernel executions and memory copy are shown below in Figure 6.2.

Table 6.2: Processing Times

OpenCL	mt (ms)	pi (ms)	copy (ms)
NVIDIA	3420.8	2399.8	0.02
AMD	536.2	409.8	0.07
FOXC	243.8	364.9	0.002

We will look at how these processing times change by parallelization. Since the time required for the memory transfer operation is negligible compared to the other operations, we will not concentrate on this part.

The code in the previous section performed all the operation on one compute unit. The correspondence of the global work-item and the local work-item to the hardware (at least as of this writing) are summarized in Table 6.3.

Table 6.3: Correspondence of hardware with OpenCL work-item

OpenCL	Global work-item	Local work-item
NVIDIA	SM	CUDA core
AMD	OS thread	Fiber
FOXC	OS thread	Fiber

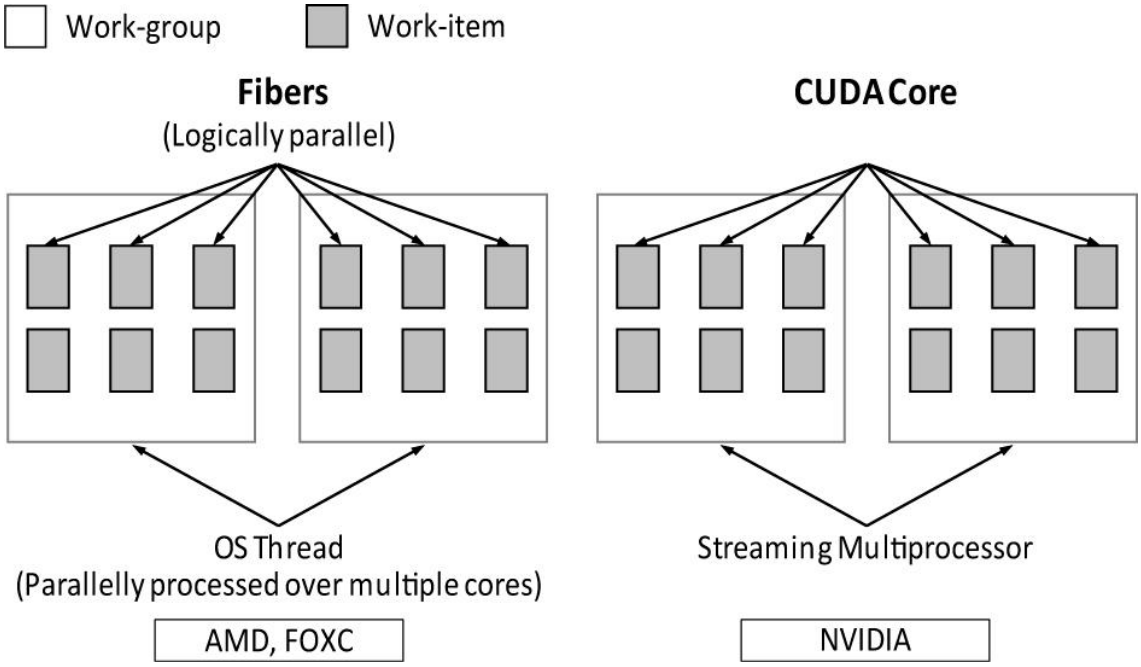
"OS threads" are threads managed by the operating systems, such as pthreads and Win32 threads.

"Fibers" are lightweight threads capable of performing context switching by saving the register contents. Unlike OS threads, fibers do not execute over multiple cores, but are capable of

performing context switching quickly.

"SM (Streaming Multiprocessor)" and "CUDA Core" are unit names specific to NVIDIA GPUs. A CUDA core is the smallest unit capable of performing computations. A group of CUDA cores make up the SM. igure 6.10 shows the basic block diagram.

Figure 6.10: OpenCL work-item and its hardware correspondences



Please refer to the "CUDA ™ Programming Guide" contained in the NVIDIA SDK for more details on the NVIDIA processor architecture [19].

Since the code on the previous section only uses one work-group, parallel execution via OS threads or SM is not performed. This would require multiple work-groups to be created.

We will change the code such that the local work-item size is the number of generated parameters created using "dc" divided by CL_DEVICE_MAX_COMPUTE_UNITS. The number of work-groups and work-items can be set using the code shown below.

```
clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint),
&num_compute_unit, NULL);
if (num_compute_unit > 32)
num_compute_unit = 32;
```

```
/* Number of work items */
num_work_item = (num_generator+(num_compute_unit-1)) / num_compute_unit;

/* Set work-group and work-item size */
global_item_size[0] = num_work_item * num_compute_unit; global_item_size[1] = 1;
global_item_size[2] = 1;
local_item_size[0] = num_work_item; local_item_size[1] = 1; local_item_size[2] = 1;
```

We have limited the maximum number of work-groups to 32, since we have only defined 32 sets of MT parameters in List 6.17, and each work-group requires a MT parameter.

```
/* Set Kernel Arguments */
clSetKernelArg(kernel_mt, 0, sizeof(cl_mem), (void*)&rand); /* Random numbers（output of genrand） */
clSetKernelArg(kernel_mt, 1, sizeof(cl_mem), (void*)&dev_mts); /* MT parameter（input to genrand） */
clSetKernelArg(kernel_mt, 2, sizeof(num_rand), &num_rand); /* Number of random numbers to generate */
clSetKernelArg(kernel_mt, 3, sizeof(num_generator), &num_generator); /* Number of MT parameters */

clSetKernelArg(kernel_pi, 0, sizeof(cl_mem), (void*)&count); /* Counter for points withincircle（output of calc_pi） */
clSetKernelArg(kernel_pi, 1, sizeof(cl_mem), (void*)&rand); /* Random numbers（input to calc_pi） */
clSetKernelArg(kernel_pi, 2, sizeof(num_rand), &num_rand); /* Random numbers (input to calc_pi) */
clSetKernelArg(kernel_pi, 3, sizeof(num_generator), &num_generator); /* Number of MT parameters */
```

On the device side, the number of random numbers to generate is limited by the number of available MT parameters.

```
__kernel void genrand(__global uint *out,__global mt_struct *mts,int num_rand, int num_generator){
```

```
int gid = get_global_id(0);
uint seed = gid*3;
uint state[17];
if (gid >= num_generator) return;
…
}

__kernel void calc_pi(__global uint *out, __global uint *rand, int num_rand, int
num_generator) {
int gid = get_global_id(0);
int count = 0;
int i;
if (gid >= num_generator) return;
…
}
```

The processing times after making the changes are shown below in Table 6.4.

Table 6.4: Processing time after parallelization

OpenCL	mt (ms)	pi (ms)
NVIDIA	2800.4	1018.8
AMD	73.6	94.6
FOXC	59.8	92.1

Note the processing time using FOXC and AMD are reduced by a factor of 4. This was expected, since it is executed on Core i7-920, which has 4 cores.

However, the processing time has not changed for NVIDIA's OpenCL. As shown in Figure 6.10, NVIDIA GPUs can perform parallel execution over multiple work-items. In the code in the previous section, all work items are performed in one compute unit, where parallelism is present over the multiple processing units. The code in this section allows one work-item to be executed on each compute-unit. So the difference is in whether the work-items were executed in parallel over the processing units, or the work-items were executed in parallel over the compute units. Thus, the processing time was not reduced significantly, and this speed up is merely from less chance of register data being stored in another memory during context switching.

To use the GPU effectively, parallel execution should occur in both the compute units and the CUDA cores.

Increasing Parallelism

This section will focus on increasing parallelism for efficient execution on the NVIDIA GPU. This will be done by increasing the number of MT parameters generated by "dc". The processing time is shown in Table 6.5 below.

Table 6.5: Processing times from increasing parallelism

# MT parameters		mt (ms)	pi (ms)
128	NVIDIA	767.4	288.8
	AMD	73.6	94.3
	FOXC	59.4	94.8
256	NVIDIA	491.8	214.4
	AMD	73.6	94.3
	FOXC	62.3	91.7
512	NVIDIA	379.0	117.5
	AMD	73.6	94.4
	FOXC	60.8	92.4

Note that the processing time does not change on the CPU, but that it reduces significantly on the GPU.

Optimization for NVIDIA GPU

Looking at Table 6.5, execution is slower on the NVIDIA despite increasing the parallelism. We will now look into optimizing the code to be executed on the GPU.

The first thing to look at for efficient execution on the GPU is the memory access. Since Tesla does not have any cache hardware, access to the global memory requires a few hundred clock cycles. Compared to processor with cache, accessing the global memory each time can slow down the average access times by a large margin. To get around this problem on the NVIDIA GPU, a wise usage of the local memory is required.

Looking at the code again, the variables "state" and "mts" accesses the same memory space many

times. We will change the code so that these data are written to the local memory.

First, we will change the host-side code so the data to be local memory is passed in as an argument to the kernel.

```
/* Set kernel arguments */
clSetKernelArg(kernel_mt, 0, sizeof(cl_mem), (void*)&rand); /* Random numbers（output of genrand）*/
clSetKernelArg(kernel_mt, 1, sizeof(cl_mem), (void*)&dev_mts); /* MT parameter（input to genrand） */
clSetKernelArg(kernel_mt, 2, sizeof(num_rand), &num_rand); /* Number of random numbers to generate */
clSetKernelArg(kernel_mt, 3, sizeof(num_generator), &num_generator); /* Number of MT parameters */
clSetKernelArg(kernel_mt, 4, sizeof(cl_uint)*17*num_work_item, NULL); /* Local Memory（state） */
clSetKernelArg(kernel_mt, 5, sizeof(mt_struct)*num_work_item, NULL); /* Local Memory（mts） */
```

The kernel will now be changed to use the local memory, by using the __local qualifier.

```
/* Initialize state using a seed */
static void sgenrand_mt(uint seed, __local const mt_struct *mts, __local uint *state) {
/* … */
}

/* Update State */
static void update_state(__local const mt_struct *mts, __local uint *st) {
/* … */
}

static inline void gen(__global uint *out, const __local mt_struct *mts, __local uint *state, int num_rand) {
/* … */
}
```

```
__kernel void genrand(__global uint *out,__global mt_struct *mts_g,int num_rand, int
num_generator,
__local uint *state_mem, __local mt_struct *mts){
int lid = get_local_id(0);
int gid = get_global_id(0);
__local uint *state = state_mem + lid*17; /* Store current state in local memory */
if (gid >= num_generator) return;
mts += lid;

/* Copy MT parameters to local memory */
mts->aaa = mts_g[gid].aaa;
mts->mm = mts_g[gid].mm;
mts->nn = mts_g[gid].nn;
mts->rr = mts_g[gid].rr;
mts->ww = mts_g[gid].ww;
mts->wmask = mts_g[gid].wmask;
mts->umask = mts_g[gid].umask;
mts->lmask = mts_g[gid].lmask;
mts->shift0 = mts_g[gid].shift0;
mts->shift1 = mts_g[gid].shift1;
mts->shiftB = mts_g[gid].shiftB;
mts->shiftC = mts_g[gid].shiftC;
mts->maskB = mts_g[gid].maskB;
mts->maskC = mts_g[gid].maskC;
out += gid * num_rand; /* Output buffer for this item*/
sgenrand_mt(0x33ff*gid, mts, (__local uint*)state); /* Initialize random numbers */
gen(out, mts, (__local uint*)state, num_rand); /* Generate random numbers */
}
```

The new execution times are shown in Table 6.6.

Table 6.6: Processing times on the Tesla using local memory

# MT parameters	mt (ms)	pi (ms)
128	246.2	287.8
256	143.4	206.7
512	108.7	113.1

We see a definite increase in speed.

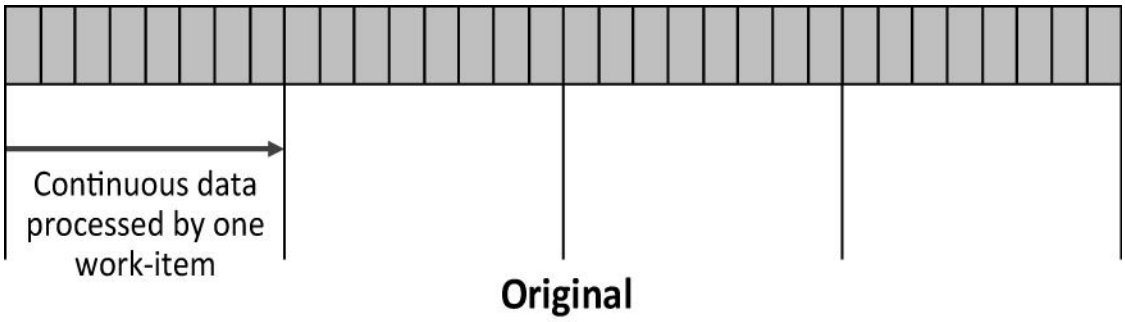
The thing to note about the local memory is that it is rather limited. On NVIDIA GPUs, the local memory size is 16,384 bytes. For our program, each work-item requires the storage of the state, which is 17 x 4 = 68 bytes, as well as the MT parameters, which is 14 x 4 = 56. Therefore, each work-item requires 124 bytes, meaning 16384/124 ≈ 132 work-items can be processed within a work-group. For GPUs that do not have many compute units, it may not be able to process if the number of parameters is increased to 256 or 512, since the number of work-items per work-group will be increased. This would not be a problem for Tesla, since the value of CL_DEVICE_MAX_COMPUT_UNITS is 30.

We will now further tune our program, which can be broken up into the next 3 processes:

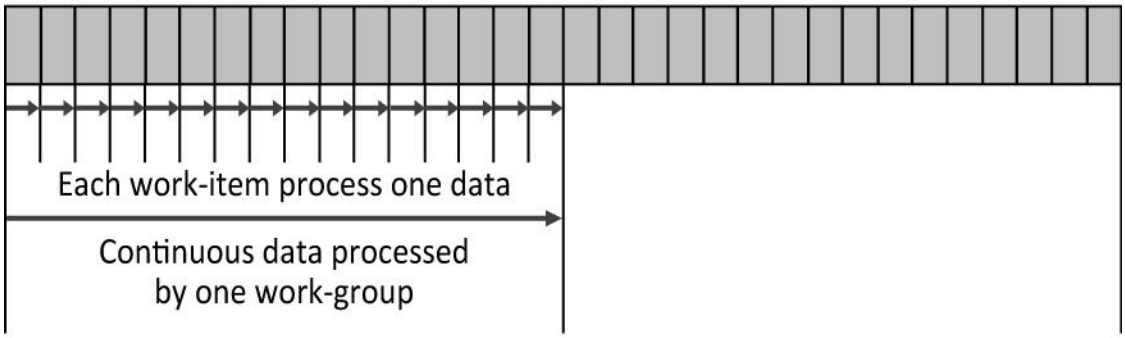
• Updating state
• Tempering
• Computing the value of PI

First, we will tackle the computation of PI. In the original code, each work-item processes a different part of an array. One of the properties of the NVIDIA GPU is that it is capable of accessing a block of continuous memory at once to be processed within a work-group. This is known as a coalesced access. Using this knowledge, we will write the code such that each work-group accesses continuous data on the memory. This change is illustrated in Figure 6.11.

Figure 6.11: Grouping of work-items



Original



After making the change

The changes in the code are shown below.

```
global_item_size_pi[0] = num_compute_unit * 128; global_item_size_pi[1] = 1;
global_item_size_pi[2] = 1;
local_item_size_pi[0] = 128; local_item_size_pi[1] = 1; local_item_size_pi[2] = 1;
/* Compute PI */
ret = clEnqueueNDRangeKernel(command_queue, kernel_pi, 1, NULL, global_item_size_pi,
local_item_size_pi, 0, NULL, &ev_pi_end);
```

The above is the change made on the host-side. The number of work-groups is changed to CL_DEVICE_MAX_COMPUT_UNITS, and the number of work-items is set to 128.

```
/* Count the number of points within the circle */
__kernel void calc_pi(__global uint *out, __global uint *rand, int num_rand_per_compute_unit,
int num_compute_unit, int num_rand_all, __local uint *count_per_wi)
{
int gid = get_group_id(0);
int lid = get_local_id(0);
int count = 0;
int i, end, begin;
```

```
/* Indices to be processed in this work-group */
begin = gid * num_rand_per_compute_unit;
end = begin + num_rand_per_compute_unit;
/* Reduce the end boundary index it is greater than the number of random numbers to
generate*/
if (end > num_rand_all)
end = num_rand_all;
            // Compute the reference address corresponding to the local ID
rand += lid;
/* Process 128 elements at a time */
for (i=begin; i < end-128; i+=128) {
float x, y, len;
x = ((float)(rand[i]>>16))/65535.0f; /* x coordinate */
y = ((float)(rand[i]&0xffff))/65535.0f; /* y coordinate */
len = (x*x + y*y); /* Distance from the origin */
if (len < 1) { /* sqrt(len) < 1 = len < 1 */
count++;
}
}

/* Process the remaining elements */
if ((i + lid) < end) {
float x, y, len;
x = ((float)(rand[i]>>16))/65535.0f; /* x coordinate */
y = ((float)(rand[i]&0xffff))/65535.0f; /* y coordinate */
len = (x*x + y*y); /* Distance from the origin */
if (len < 1) { /* sqrt(len) < 1 = len < 1 */
count++;
}
}
count_per_wi[lid] = count;
/* Sum the counters from each work-item */
barrier(CLK_LOCAL_MEM_FENCE); /* Wait until all work-items are finished */
if (lid == 0) {
int count = 0;
```

```
for (i=0; i < 128; i++) {
count += count_per_wi[i]; /* Sum the counters */
}
out[gid] = count;
}
}
```

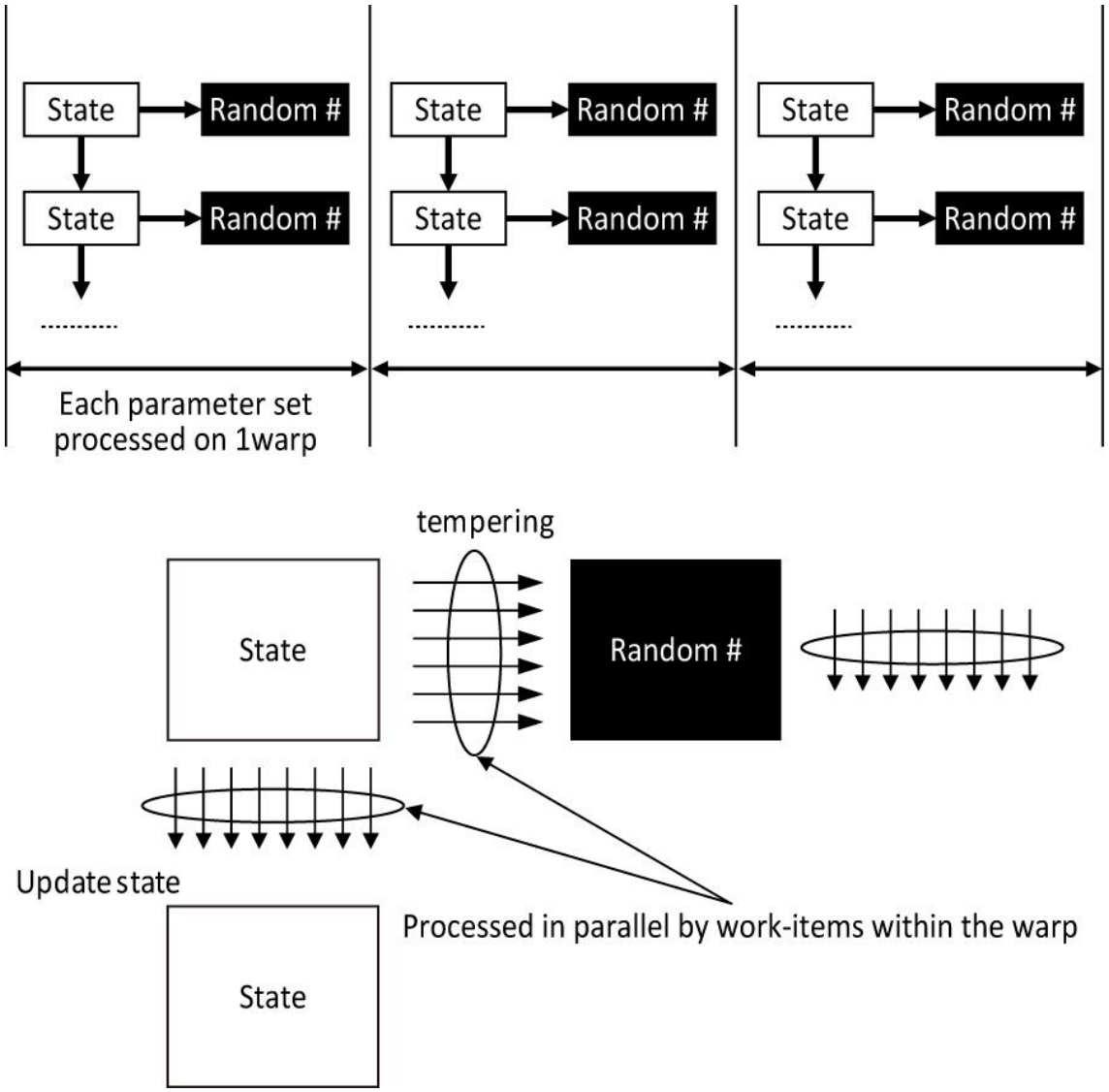
The above code shows the changes made on the device side. Note that each work-group processes the input data such that all the work-items within the work-group access continuous address space. The code includes the case when the number of data to process is not a multiple of 128. The processing time using this code is 9.05 ms, which is a 10-fold improvement over the previous code.

Next, we will optimize the tempering algorithm. As stated earlier, this process has a parallel nature. For the parameters in our example program, 9 state update and 17 tempering can be performed in parallel. However, since the processing on each word does not require many instructions, and depending on the cost of synchronization, performance may potentially suffer from parallelization. We had concluded previously that a use of SIMD units would be optimal for this process, to reduce the cost of synchronization.

NVIDIA GPUs performs its processes in "warps". A warp is made up of 32 work-items, and all work-items within a warp are executed synchronously. Taking advantage of this property, some operations can be performed in parallel without the need to synchronize.

We will use one warp to perform the update of the state, as well as the tempering process. This would enable these processes to be performed without increasing the synchronization cost. A new block diagram of the distribution of work-items and work-groups are shown in Figure 6.10 below.

Figure 6.10: Distribution of work-items for MT



Note that this optimization is done by using the property of NVIDIA GPUs, so making this change may not allow for proper operation on other devices.

The changes to the program are shown below.

```
/* Each compute unit process 4 warps */
warp_per_compute_unit = 4;
/* Total number of warps */
num_warp = warp_per_compute_unit * num_compute_unit ;
/* Number of MT parameters per warp (rounded up) */
num_param_per_warp = (num_generator + (num_warp-1)) / num_warp;
/* Number of work-items per group (warp = 32work items) */
```

```
num_work_item = 32 * warp_per_compute_unit;
/* Number of random numbers per group */
num_rand_per_compute_unit = (num_rand_all + (num_compute_unit-1)) / num_compute_unit;

/* Set Kernel Arguments */
clSetKernelArg(kernel_mt, 0, sizeof(cl_mem), (void*)&rand); /* Random numbers (output of genrand) */
clSetKernelArg(kernel_mt, 1, sizeof(cl_mem), (void*)&dev_mts); /* MT parameter (input to genrand) */
clSetKernelArg(kernel_mt, 2, sizeof(num_rand_per_generator), &num_rand_per_generator); /* Number of random numbers to generate for each MT parameter */
clSetKernelArg(kernel_mt, 3, sizeof(num_generator), &num_generator); /* Number of random numbers to generate */
clSetKernelArg(kernel_mt, 4, sizeof(num_param_per_warp), &num_param_per_warp); /* Number of parameters to process per work-group */
clSetKernelArg(kernel_mt, 5, sizeof(cl_uint)*17*num_work_item, NULL); /* Local Memory (state) */
clSetKernelArg(kernel_mt, 6, sizeof(mt_struct)*num_work_item, NULL); /* Local Memory (mts) */

/* Set the number of work-groups and work-items per work-group */
global_item_size_mt[0] = num_work_item * num_compute_unit; global_item_size_mt[1] = 1; global_item_size_mt[2] = 1;
local_item_size_mt[0] = num_work_item; local_item_size_mt[1] = 1; local_item_size_mt[2] = 1;
```

The above shows the changes made on the host-side. The number of random numbers to be generated by each work-group is first computed, and this is sent in as an argument. The number of work-items in each work-group is set to 128, which is 4 warps.

```
/* Update state */
static void update_state(__local const mt_struct *mts, __local uint *st, int wlid) {
int n = 17, m = 8;
uint aa = mts->aaa, x;
uint uuu = mts->umask, lll = mts->lmask;
int k,lim;
```

```
if (wlid < 9) {
/* Accessing indices k+1 and k+m would normally have dependency issues,
* but since the operations within a warp is synchronized, the write to
* st[k] by each work-item occurs after read from st[k+1] and st[k+m] */

k = wlid;
x = (st[k]&uuu)|(st[k+1]&lll);
st[k] = st[k+m] ^ (x>>1) ^ (x&1U ? aa : 0U);
}
if (wlid < 7) {
/* Same reasoning as above. No problem as long as the read operation
* for each work-item within a work-group is finished before writing */

k = wlid + 9;
x = (st[k]&uuu)|(st[k+1]&lll);
st[k] = st[k+m-n] ^ (x>>1) ^ (x&1U ? aa : 0U);
}
if (wlid == 0) {
x = (st[n-1]&uuu)|(st[0]&lll);
st[n-1] = st[m-1] ^ (x>>1) ^ (x&1U ? aa : 0U);
}
}

static inline void gen(__global uint *out, const __local mt_struct *mts, __local uint
*state, int num_rand, int wlid) {
int i, j, n, nn = mts->nn;
n = (num_rand+(nn-1)) / nn;

for (i=0; i < n; i++) {
int m = nn;
if (i == n-1) m = num_rand%nn;

update_state(mts, state, wlid);

if (wlid < m) { /* tempering performed within 1 warp */
```

```
int j = wlid;
uint x = state[j];
x ^= x >> mts->shift0;
x ^= (x << mts->shiftB) & mts->maskB;
x ^= (x << mts->shiftC) & mts->maskC;
x ^= x >> mts->shift1;
out[i*nn + j] = x;
}
}
}

__kernel void genrand(__global uint *out,__global mt_struct *mts_g,int num_rand, int
num_generator, uint num_param_per_warp, __local uint *state_mem,
__local mt_struct *mts){
int warp_per_compute_unit = 4;
int workitem_per_warp = 32;
int wid = get_group_id(0);
int lid = get_local_id(0);
int warp_id = wid * warp_per_compute_unit + lid / workitem_per_warp;
int generator_id, end;
int wlid = lid % workitem_per_warp; /* Local ID within the warp */

__local uint *state = state_mem + warp_id*17; /* Store state in local memory */

end = num_param_per_warp*warp_id + num_param_per_warp;

if (end > num_generator)
end = num_generator;

mts = mts + warp_id;

/* Loop for each MT parameter within this work-group */
for (generator_id = num_param_per_warp*warp_id;
generator_id < end;
generator_id ++)
{
```

```
if (wlid == 0) {
/* Copy MT parameters to local memory */
mts->aaa = mts_g[generator_id].aaa;
mts->mm = mts_g[generator_id].mm;
mts->nn = mts_g[generator_id].nn;
mts->rr = mts_g[generator_id].rr;
mts->ww = mts_g[generator_id].ww;
mts->wmask = mts_g[generator_id].wmask;
mts->umask = mts_g[generator_id].umask;
mts->lmask = mts_g[generator_id].lmask;
mts->shift0 = mts_g[generator_id].shift0;
mts->shift1 = mts_g[generator_id].shift1;
mts->shiftB = mts_g[generator_id].shiftB;
mts->shiftC = mts_g[generator_id].shiftC;
mts->maskB = mts_g[generator_id].maskB;
mts->maskC = mts_g[generator_id].maskC;
sgenrand_mt(0x33ff*generator_id, mts, (__local uint*)state); /* Initialize random numbers */
}
gen(out + generator_id*num_rand, mts, (__local uint*)state, num_rand, wlid); /* Generate
random numbers */
}
}
```

The above code shows the changes made on the device-side. It is changed such that one DC parameter is processed on each warp, and parallel processing is performed within the warp, reducing the need for synchronization. The processing times after making these changes are shown in Table 6.7 below.

Table 6.7: Processing times after optimization

# MT parameters	mt (ms)
128	57.0
256	42.8
512	35.7

Note the processing times are reduced significantly.

The final code for the program with all the changes discussed is shown in List 6.18 (host) and List 6.19 (device). Running this code on Tesla, there is a 90-fold improvement since the original program [20].

List 6.18: Host-side optimized for NVIDIA GPU

```
001:#include <stdlib.h>
002:#include <CL/cl.h>
003:#include <stdio.h>
004:#include <math.h>
005:
006:typedef struct mt_struct_s {
007: cl_uint aaa;
008: cl_int mm,nn,rr,ww;
009: cl_uint wmask,umask,lmask;
010: cl_int shift0, shift1, shiftB, shiftC;
011: cl_uint maskB, maskC;
012:} mt_struct ;
013:
014:#include "mts.h"
015:
016:#define MAX_SOURCE_SIZE (0x100000)
017:
018:int main(int argc, char **argv)
019:{
020:      cl_int num_rand_all = 4096*256*32; /* The number of random numbers to generate
*/
021:      cl_int num_rand_per_compute_unit, num_rand_per_generator;
022:      int count_all, i;
023:      cl_uint num_generator;
024:      unsigned int num_work_item;
025:      double pi;
026:      cl_platform_id platform_id = NULL;
027:      cl_uint ret_num_platforms;
028:      cl_device_id device_id = NULL;
029:      cl_uint ret_num_devices;
030:      cl_context context = NULL;
```

```
031:    cl_command_queue command_queue = NULL;
032:    cl_program program = NULL;
033:    cl_kernel kernel_mt = NULL, kernel_pi = NULL;
034:    size_t kernel_code_size;
035:    char *kernel_src_str;
036:    cl_uint *result;
037:    cl_int ret;
038:    FILE *fp;
039:    cl_mem rand, count;
040:
041:    size_t global_item_size_mt[3], local_item_size_mt[3];
042:    size_t global_item_size_pi[3], local_item_size_pi[3];
043:
044:    cl_mem dev_mts;
045:    cl_event ev_mt_end, ev_pi_end, ev_copy_end;
046:    cl_ulong prof_start, prof_mt_end, prof_pi_end, prof_copy_end;
047:    cl_uint num_compute_unit, warp_per_compute_unit, num_warp,
num_param_per_warp;
048:    int mts_size;
049:    mt_struct *mts = NULL;
050:
051:    if (argc >= 2) {
052:        int n = atoi(argv[1]);
053:        if (n == 128) {
054:            mts = mts128;
055:            mts_size = sizeof(mts128);
056:            num_generator = 128;
057:            num_rand_per_generator = num_rand_all / 128;
058:        }
059:        if (n == 256) {
060:            mts = mts256;
061:            mts_size = sizeof(mts256);
062:            num_generator = 256;
063:            num_rand_per_generator = num_rand_all / 256;
064:        }
065:    }
```

```
066:
067:      if (mts == NULL) {
068:            mts = mts512;
069:            mts_size = sizeof(mts512);
070:            num_generator = 512;
071:            num_rand_per_generator = num_rand_all / 512;
072:      }
073:
074:      clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
075:      clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
076:            &ret_num_devices);
077:      context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
078:      result = (cl_uint*)malloc(sizeof(cl_uint)*num_generator);
079:
080:      command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_PROFILING_ENABLE, &ret);
081:
082:      fp = fopen("mt.cl", "r");
083:      kernel_src_str = (char*)malloc(MAX_SOURCE_SIZE);
084:      kernel_code_size = fread(kernel_src_str, 1, MAX_SOURCE_SIZE, fp);
085:      fclose(fp);
086:
087:      /* Create output buffer */
088:      rand = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(cl_uint)*num_rand_all,
NULL, &ret);
089:      count = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(cl_uint)*num_generator, NULL, &ret);
090:
091:      /* Build Program*/
092:      program = clCreateProgramWithSource(context, 1, (const char **)&kernel_src_str,
093:            (const size_t *)&kernel_code_size, &ret);
094:      clBuildProgram(program, 1, &device_id, "", NULL, NULL);
095:
096:      kernel_mt = clCreateKernel(program, "genrand", &ret);
097:      kernel_pi = clCreateKernel(program, "calc_pi", &ret);
098:
```

```
099:      /* Create input parameter */
100:      dev_mts = clCreateBuffer(context, CL_MEM_READ_WRITE, mts_size, NULL, &ret);
101:      clEnqueueWriteBuffer(command_queue, dev_mts, CL_TRUE, 0, mts_size, mts, 0,
NULL, NULL);
102:
103:      clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint),
&num_compute_unit, NULL);
104:
105:      /* Each compute unit process 4 warps */
106:      warp_per_compute_unit = 4;
107:      /* Total number of warps */
108:      num_warp = warp_per_compute_unit * num_compute_unit ;
109:      /* Number of MT parameters per warp (rounded up) */
110:      num_param_per_warp = (num_generator + (num_warp-1)) / num_warp;
111:      /* Number of work-items per group (warp = 32work items) */
112:      num_work_item = 32 * warp_per_compute_unit;
113:      /* Number of random numbers per group */
114:      num_rand_per_compute_unit = (num_rand_all + (num_compute_unit-1)) /
num_compute_unit;
115:
116:      /* Set Kernel Arguments */
117:      clSetKernelArg(kernel_mt, 0, sizeof(cl_mem), (void*)&rand); /* Random numbers
(output of genrand) */
118:      clSetKernelArg(kernel_mt, 1, sizeof(cl_mem), (void*)&dev_mts); /* MT parameter
(input to genrand) */
119:      clSetKernelArg(kernel_mt, 2, sizeof(num_rand_per_generator),
&num_rand_per_generator); /* Number of random numbers to generate for each MT
parameter */
120:      clSetKernelArg(kernel_mt, 3, sizeof(num_generator), &num_generator); /* Number of
random numbers to generate */
121:      clSetKernelArg(kernel_mt, 4, sizeof(num_param_per_warp), &num_param_per_warp);
/* Number of parameters to process per work-group */
122:      clSetKernelArg(kernel_mt, 5, sizeof(cl_uint)*17*num_work_item, NULL); /* Local
Memory (state) */
123:      clSetKernelArg(kernel_mt, 6, sizeof(mt_struct)*num_work_item, NULL); /* Local
Memory (mts) */
```

```
124:
125:      clSetKernelArg(kernel_pi, 0, sizeof(cl_mem), (void*)&count); /* Counter for points
within circle (output of calc_pi) */
126:      clSetKernelArg(kernel_pi, 1, sizeof(cl_mem), (void*)&rand); /* Random numbers
(input to calc_pi) */
127:      clSetKernelArg(kernel_pi, 2, sizeof(num_rand_per_compute_unit),
&num_rand_per_compute_unit); /* Number of random numbers to process per work-group */
128:      clSetKernelArg(kernel_pi, 3, sizeof(num_compute_unit), &num_compute_unit); /*
Number of MT parameters */
129:      clSetKernelArg(kernel_pi, 4, sizeof(num_rand_all), &num_rand_all); /* Number of
random numbers used */
130:      clSetKernelArg(kernel_pi, 5, sizeof(cl_uint)*128, NULL); /* Memory used for counter */
131:
132:      /* Set the number of work-groups and work-items per work-group */
133:      global_item_size_mt[0] = num_work_item * num_compute_unit;
global_item_size_mt[1] = 1; global_item_size_mt[2] = 1;
134:      local_item_size_mt[0] = num_work_item; local_item_size_mt[1] = 1;
local_item_size_mt[2] = 1;
135:
136:      /* Create a random number array */
137:      clEnqueueNDRangeKernel(command_queue, kernel_mt, 1, NULL, global_item_size_mt,
local_item_size_mt, 0, NULL, &ev_mt_end);
138:
139:      global_item_size_pi[0] = num_compute_unit * 128; global_item_size_pi[1] = 1;
global_item_size_pi[2] = 1;
140:      local_item_size_pi[0] = 128; local_item_size_pi[1] = 1; local_item_size_pi[2] = 1;
141:
142:      /* Compute PI */
143:      clEnqueueNDRangeKernel(command_queue, kernel_pi, 1, NULL, global_item_size_pi,
local_item_size_pi, 0, NULL, &ev_pi_end);
144:
145:      /* Get result */
146:      clEnqueueReadBuffer(command_queue, count, CL_TRUE, 0,
sizeof(cl_uint)*num_generator, result, 0, NULL, &ev_copy_end);
147:
148:      /* Average the values of PI */
```

```
149:      count_all = 0;
150:      for (i=0; i<(int)num_compute_unit; i++) {
151:            count_all += result[i];
152:      }
153:
154:      pi = ((double)count_all)/(num_rand_all) * 4;
155:      printf("pi = %.20f¥n", pi);
156:
157:      /* Get execution time info */
158:      clGetEventProfilingInfo(ev_mt_end, CL_PROFILING_COMMAND_QUEUED,
sizeof(cl_ulong), &prof_start, NULL);
159:      clGetEventProfilingInfo(ev_mt_end, CL_PROFILING_COMMAND_END, sizeof(cl_ulong),
&prof_mt_end, NULL);
160:      clGetEventProfilingInfo(ev_pi_end, CL_PROFILING_COMMAND_END, sizeof(cl_ulong),
&prof_pi_end, NULL);
161:      clGetEventProfilingInfo(ev_copy_end, CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &prof_copy_end, NULL);
162:
163:      printf(" mt: %f[ms]¥n"
164:            " pi: %f[ms]¥n"
165:            " copy: %f[ms]¥n",
166:            (prof_mt_end - prof_start)/(1000000.0),
167:            (prof_pi_end - prof_mt_end)/(1000000.0),
168:            (prof_copy_end - prof_pi_end)/(1000000.0));
169:
170:      clReleaseEvent(ev_mt_end);
171:      clReleaseEvent(ev_pi_end);
172:      clReleaseEvent(ev_copy_end);
173:
174:      clReleaseMemObject(rand);
175:      clReleaseMemObject(count);
176:      clReleaseKernel(kernel_mt);
177:      clReleaseKernel(kernel_pi);
178:      clReleaseProgram(program);
179:      clReleaseCommandQueue(command_queue);
180:      clReleaseContext(context);
```

```
181:      free(kernel_src_str);
182:      free(result);
183:      return 0;
184:}
```

List 6.19: Device-side optimized for NVIDIA GPU

```
001:typedef struct mt_struct_s {
002:      uint aaa;
003:      int mm,nn,rr,ww;
004:      uint wmask,umask,lmask;
005:      int shift0, shift1, shiftB, shiftC;
006:      uint maskB, maskC;
007:} mt_struct ;
008:
009:/* Initialize state using a seed */
010:static void sgenrand_mt(uint seed, __local const mt_struct *mts, __local uint *state) {
011:      int i;
012:      for (i=0; i < mts->nn; i++) {
013:            state[i] = seed;
014:            seed = (1812433253 * (seed ^ (seed >> 30))) + i + 1;
015:      }
016:      for (i=0; i < mts->nn; i++)
017:            state[i] &= mts->wmask;
018:}
019:
020:/* Update state */
021:static void update_state(__local const mt_struct *mts, __local uint *st, int wlid) {
022:      int n = 17, m = 8;
023:      uint aa = mts->aaa, x;
024:      uint uuu = mts->umask, lll = mts->lmask;
025:      int k,lim;
026:
027:      if (wlid < 9) {
028:            /* Accessing indices k+1 and k+m would normally have dependency issues,
029:            * but since the operations within a warp is synchronized, the write to
030:            * st[k] by each work-item occurs after read from st[k+1] and st[k+m] */
```

```
031:
032:            k = wlid;
033:            x = (st[k]&uuu)|(st[k+1]&lll);
034:            st[k] = st[k+m] ^ (x>>1) ^ (x&1U ? aa : 0U);
035:      }
036:      if (wlid < 7) {
037:            /* Same reasoning as above. No problem as long as the read operation
038:            * for each work-item within a work-group is finished before writing */
039:
040:            k = wlid + 9;
041:            x = (st[k]&uuu)|(st[k+1]&lll);
042:            st[k] = st[k+m-n] ^ (x>>1) ^ (x&1U ? aa : 0U);
043:      }
044:      if (wlid == 0) {
045:            x = (st[n-1]&uuu)|(st[0]&lll);
046:            st[n-1] = st[m-1] ^ (x>>1) ^ (x&1U ? aa : 0U);
047:      }
048:}
049:
050:static inline void gen(__global uint *out, const __local mt_struct *mts, __local uint *state,
int num_rand, int wlid) {
051:      int i, j, n, nn = mts->nn;
052:      n = (num_rand+(nn-1)) / nn;
053:
054:      for (i=0; i < n; i++) {
055:            int m = nn;
056:            if (i == n-1) m = num_rand%nn;
057:
058:            update_state(mts, state, wlid);
059:
060:            if (wlid < m) { /* tempering performed within 1 warp */
061:                  int j = wlid;
062:                  uint x = state[j];
063:                  x ^= x >> mts->shift0;
064:                  x ^= (x << mts->shiftB) & mts->maskB;
065:                  x ^= (x << mts->shiftC) & mts->maskC;
```

```
066:                x ^= x >> mts->shift1;
067:                out[i*nn + j] = x;
068:            }
069:        }
070:}
071:
072:__kernel void genrand(__global uint *out,__global mt_struct *mts_g,int num_rand, int
num_generator,
073:            uint num_param_per_warp, __local uint *state_mem, __local mt_struct *mts){
074:        int warp_per_compute_unit = 4;
075:        int workitem_per_warp = 32;
076:        int wid = get_group_id(0);
077:        int lid = get_local_id(0);
078:        int warp_id = wid * warp_per_compute_unit + lid / workitem_per_warp;
079:        int generator_id, end;
080:        int wlid = lid % workitem_per_warp; /* Local ID within the warp */
081:
082:        __local uint *state = state_mem + warp_id*17; /* Store state in local memory */
083:
084:        end = num_param_per_warp*warp_id + num_param_per_warp;
085:
086:        if (end > num_generator)
087:            end = num_generator;
088:
089:        mts = mts + warp_id;
090:
091:
092:        /* Loop for each MT parameter within this work-group */
093:        for (generator_id = num_param_per_warp*warp_id;
094:            generator_id < end;
095:            generator_id ++)
096:        {
097:            if (wlid == 0) {
098:                /* Copy MT parameters to local memory */
099:                mts->aaa = mts_g[generator_id].aaa;
100:                mts->mm = mts_g[generator_id].mm;
```

```
101:                mts->nn = mts_g[generator_id].nn;
102:                mts->rr = mts_g[generator_id].rr;
103:                mts->ww = mts_g[generator_id].ww;
104:                mts->wmask = mts_g[generator_id].wmask;
105:                mts->umask = mts_g[generator_id].umask;
106:                mts->lmask = mts_g[generator_id].lmask;
107:                mts->shift0 = mts_g[generator_id].shift0;
108:                mts->shift1 = mts_g[generator_id].shift1;
109:                mts->shiftB = mts_g[generator_id].shiftB;
110:                mts->shiftC = mts_g[generator_id].shiftC;
111:                mts->maskB = mts_g[generator_id].maskB;
112:                mts->maskC = mts_g[generator_id].maskC;
113:                sgenrand_mt(0x33ff*generator_id, mts, (__local uint*)state); /* Initialize
random numbers */
114:            }
115:            gen(out + generator_id*num_rand, mts, (__local uint*)state, num_rand, wlid); /*
Generate random numbers */
116:      }
117:}
118:
119:/* Count the number of points within the circle */
120:__kernel void calc_pi(__global uint *out, __global uint *rand, int
num_rand_per_compute_unit,
121:            int num_compute_unit, int num_rand_all, __local uint *count_per_wi) {
122:      int gid = get_group_id(0);
123:      int lid = get_local_id(0);
124:
125:      int count = 0;
126:      int i, end, begin;
127:
128:      /* Indices to be processed in this work-group */
129:      begin = gid * num_rand_per_compute_unit;
130:      end = begin + num_rand_per_compute_unit;
131:
132:      /* Reduce the end boundary index it is greater than the number of random numbers
to generate*/
```

```
133:    if (end > num_rand_all)
134:    end = num_rand_all;
135:    // Compute the reference address corresponding to the local ID
136:    rand += lid;
137:
138:    /* Process 128 elements at a time */
139:    for (i=begin; i<end-128; i+=128) {
140:        float x, y, len;
141:        x = ((float)(rand[i]>>16))/65535.0f; /* x coordinate */
142:        y = ((float)(rand[i]&0xffff))/65535.0f; /* y coordinate */
143:        len = (x*x + y*y); /* Distance from the origin */
144:
145:        if (len < 1) { /* sqrt(len) < 1 = len < 1 */
146:            count++;
147:        }
148:    }
149:
150:
151:    /* Process the remaining elements */
152:    if ((i + lid) < end) {
153:        float x, y, len;
154:        x = ((float)(rand[i]>>16))/65535.0f; /* x coordinate */
155:        y = ((float)(rand[i]&0xffff))/65535.0f; /* y coordinate */
156:        len = (x*x + y*y); /* Distance from the origin*/
157:
158:        if (len < 1) { /* sqrt(len) < 1 = len < 1 */
159:            count++;
160:        }
161:    }
162:
163:    count_per_wi[lid] = count;
164:
165:    /* Sum the counters from each work-item */
166:
167:    barrier(CLK_LOCAL_MEM_FENCE); /* Wait until all work-items are finished */
168:
```

```
169:      if (lid == 0) {
170:            int count = 0;
171:            for (i=0; i < 128; i++) {
172:                  count += count_per_wi[i]; /* Sum the counters */
173:            }
174:            out[gid] = count;
175:      }
176:}
```

Notes

Chapter 1

1: Wikipedia, http://en.wikipedia.org/wiki/Parallel_computing

2: Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972.

3: TOP500 Supercomputer Sites, http://www.top500.org

4: Intel MultiProcessor Specification,
http://www.intel.com/design/pentium/datashts/242016.htm

5: http://software.intel.com/en-us/articles/optimizing-software-applications-for-numa/

6: Amdahl, G.M. Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

Chapter 2

7: NVIDIA uses the term "GPGPU computing" when referring to performing general-purpose computations through the graphics API. The term "GPU computing" is used to refer to performing computation using CUDA or OpenCL. This text will use the term "GPGPU" to refer to any general-purpose computation performed on the GPU.

8: http://www.nvidia.com/object/cuda_home.html

9: The kernel can be executed without using the <<<>>> constructor, but this will require calls to multiple CUDA-specific library functions.

10: http://www.ibm.com/developerworks/power/cell/index.html

11: The devices that can be used depend on the OpenCL implementation by that company. The OpenCL implementation should be chosen wisely depending on the platform.

12: Some reasons for this requirement are

(1) Each processor on the device is often only capable of running small programs, due to limited accessible memory.

(2) Because of the limited memory, it is common for the device to not run an OS. This is actually beneficial, as it allows the processors to concentrate on just the computations.

13: At present, it is not possible to use an OpenCL compiler by one company, and an OpenCL runtime library by another company. This should be kept in mind when programming a platform with multiple devices from different chip vendors.

Chapter 4

14: The flag passed in the 2nd argument only specifies how the kernel-side can access the memory space. If the "CL_MEM_READ_ONLY" is specified, this only allows the kernel to read from the specified address space, and does not imply that the buffer would be created in the constant memory. Also, the host is only allowed access to either the global memory or the constant memory on the device.
15: Otherwise, each kernel would have to be compiled independently.

Chapter 5

16: The OpenCL specification does not state that the local memory will correspond to the scratch-pad memory, but it is most probable that this would be the case. Also, for those experienced in CUDA, note that the OpenCL local memory does not correspond with the CUDA local memory, as it corresponds to the shared memory.
17: At the time of this writing (Dec 2009), the Mac OS X OpenCL returns CL_TRUE, but the image objects are not supported.

Chapter 6

18: Victor Podlozhnyuk, Parallel Mersenne Twister
19: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf
20: The "mts.h" is not explained here, as one only need to know that three variables are declared with the names "mts128", "mts256", "mts512", each containing corresponding number of DC parameters.