OpenCL for Massively Parallel Graphic Processors

5

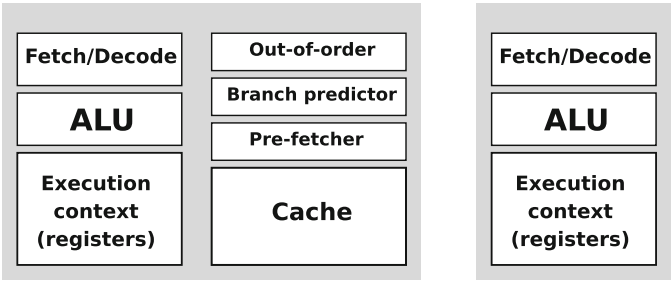
Chapter Summary
This chapter will teach us how to program GPUs using OpenCL. Almost all desktop computers ship with a quad-core processor and a GPU. Thus, we need a programming environment in which a programmer can write programs and run them on either a GPU or a quad-core CPU and a GPU. While CPUs are designed to handle complex tasks, such as time slicing, branching, etc., GPUs only do one thing well. They handle billions of repetitive low-level arithmetic operations. High-level languages, such as CUDA and OpenCL, that target the GPUs directly, are available today so GPU programming is rapidly becoming one of the mainstreams in the computer science community.

5.1 Anatomy of a GPU

In order to understand how to program massively parallel graphic processors, we must first understand how they are built. In the first part of this chapter, we will look behind the idea of processors in graphic processing units (GPU). The basic idea is to have many (hundreds or even thousands) simpler and weaker processing units in GPU instead of one or two powerful CPUs and let these many processors simultaneously perform the same instructions, but with different data. First, let us learn how a GPU is constructed. Then, we will learn how we program graphic processing units using the OpenCL language.

Processors in GPU differ from general-purpose CPUs in that they have a much simpler structure that is designed to execute hundreds of arithmetic instructions simultaneously. To understand how to implement such an efficient massively parallel processor, we will first briefly describe how general-purpose CPUs are built. The simplified structure of a general-purpose single-core CPU is presented in Fig. 5.1a.

R. Trobec et al., *Introduction to Parallel Computing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-98833-7_5



(a) Basic structure of a general-purpose single-core CPU.

(b) Basic structure of a slim single-core CPU.

Fig. 5.1 **a**) A general-purpose single-core CPU. **b**) A slimmed single-core CPU

It consists of the instruction fetch and instruction decode logic, an arithmetic-logic unit (ALU), and the execution context. The fetch/decode logic is responsible for fetching the instructions from memory, decoding them in order to prepare operands and select the required operation in ALU. The execution context comprises of the state of CPU such as a program counter, a stack pointer, a program-status register, and general-purpose registers. Such a general-purpose single-core CPU with a single ALU and execution context can run a single instruction from an instruction stream (*thread*) at a time. To increase the performance when executing a single thread, general-purpose single-core CPUs rely on out-of-order execution and branch prediction to reduce stalls. However, execution units are of no use without the instructions and the operands, which are stored in main memory. Transferring the instructions and operands to and from main memory requires considerable amount of power and time. This is addressed by the use of caches. Caches work on the principle of either spatial or temporal locality. They work well when an instruction stream is repeated many times (e.g., program loops) and when data is accessed from relatively close memory words. ALUs and fetch/decode logic run at high speed, consume little power, and require few hardware resources to build them. Contrary to execution units, a huge number of transistors is needed to build a cache (it may occupy up to 50% of the total die area) and they are very expensive. It is also one of the main energy absorbing element in general-purpose CPU.

5.1.1 Introduction to GPU Evolution

To build a GPU that comprises of tens or thousands of CPUs, we need a slimmer design of a CPU. For this reason, all complex and large units should be removed from general-purpose CPU: a branch predictor, out of order logic, caches, and a cache prefetcher. Such a single-core CPU with a slimmer design is presented in Fig. 5.1b.

Now, suppose we are running the following fragment of code on a slimmed single-core CPU from Fig. 5.1b:

```
void vectorAdd( float *vecA, float *vecB, float *vecC ) {
  int tid = 0;
  while (tid < 128) {
    vecC[tid] = vecA[tid] + vecB[tid];
    tid += 1;
  }
}
```

Listing 5.1 Vector addition

The C code in Listing 5.1 implements vector addition of two floating-point vectors, each containing 128 elements. A slimmed CPU executes a single instruction stream obtained after the compilation of the program in Listing 5.1. A compiled fragment of the function `VectorAdd` that runs on a single-core CPU is presented in Fig. 5.2. With the first two instructions in Fig. 5.2, we clear the registers `r2` and `r3` (suppose `r0` iz a zero register). The register `r2` is used to store loop counter (`tid` from Listing 5.1) while the register `r3` contains offset in the vectors `VecA` and `VecB`. Within the `L1` loop CPU loads adjacent elements from the vectors `VecA` and `VecB` into the floating-point registers `f1` and `f2`, adds them and stores the result from the register `f1` into the vector `VecC`. After that we increment the offset in the register `r3`. Recall that the vectors contain floating-point numbers, which are represented with 32 bits (4 bytes), thus the offset is incremented by 4. At the end of the loop, we increment the loop counter (variable `tid`) in the register `r2`, compare the loop counter with the value of 128 (the number of elements in each vector) and loop back if the counter is smaller, then the length of the vectors `VecA` and `VecB`.

Instead of using one slimmed CPU core from Fig. 5.2, we can use two such cores. Why? If we use two CPU cores form Fig. 5.2, we will be able to execute two instruction streams fully in parallel (Fig. 5.3). A two cores CPU from Fig. 5.3 replicates processing resources (Fetch/Decode logic, ALU, and execution context) and organizes them into two independent cores. When an application features two

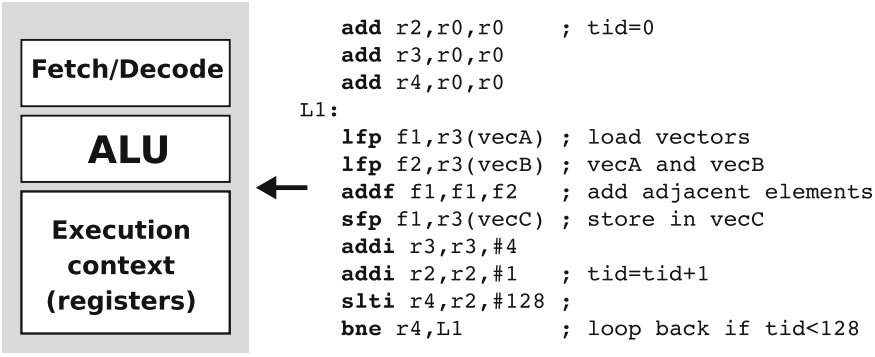


Fig. 5.2 A single instruction stream is executed on a single-core CPU

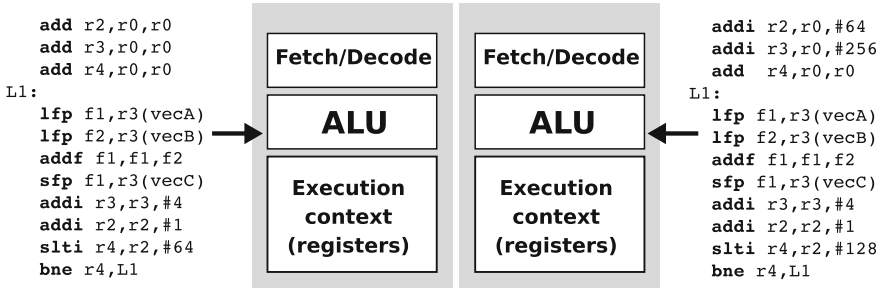


Fig. 5.3 Two instructions streams (two threads) are executed fully in parallel on two CPU cores

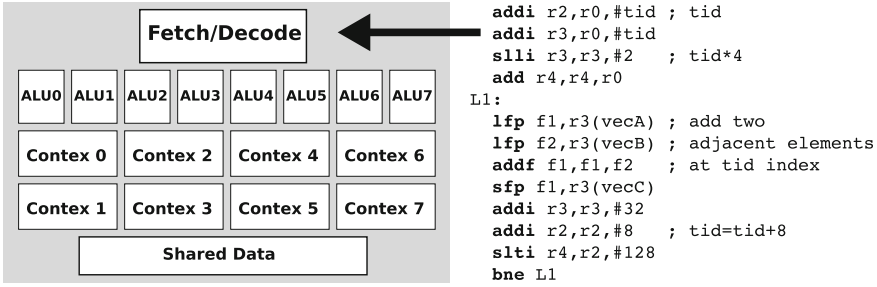


Fig. 5.4 A GPU core with eight ALUs, eight execution contexts, and shared fetch/decode logic

instruction streams (i.e., two threads), a two cores CPU provides increased throughput by simultaneously executing these instruction streams on each core. In the case of vector addition from Listing 5.1, we can now run two threads on each core. In this case, each thread will add 64 adjacent vector elements. Notice that both threads in Fig. 5.3 have the same instruction stream but use different data. The first thread adds the first 64 elements (the loop index `tid` in the register `r2` iterates from 0 to 63), while the second thread adds the last 64 elements (the loop index `tid` in the register `r2` iterates from 64 to 127).

We can achieve even higher performance by further replicating ALUs and execution contexts as in Fig. 5.4. Instead of replicating the complete CPU core from Fig. 5.2, we can replicate only ALU and execution context and leaving the fetch/decode logic shared among ALUs. As the fetch/decode logic is shared, all ALUs should execute the same operations contained in an instruction stream, but they can use different input data. Figure 5.4 depicts such a core with eight ALUs, eight execution contexts and shared fetch/decode logic. Such a core usually implements additional storage for data shared among the threads.

On such a core, we can add eight adjacent vector elements in parallel using one instruction stream. The instruction stream is now shared across threads with identical program counters (PC). The same instruction is executed for each thread but on different data. Thus, there is one ALU and one execution context per thread. Each thread should now use its own ID (`tid`) to identify data which is to be used in

Compute units and Processing elements

Terminology about processor cores in a modern GPU can be very confusing. The meaning of a term depends on who manufactured a particular GPU. To make thing simple, we opted for the following terminology. A compute unit can be considered equivalent to cores in CPU. Compute units are the basic computational building blocks of GPUs. CPU cores were designed for serial tasks like productivity applications, while GPUs were designed for more parallel and graphics-intensive tasks like video editing, gaming, and rich Web browsing. Compute units have many ALUs and execution context that share a common fetch/decode logic. ALUs execute same instructions in a *lock-step basis*, i.e., running the same instruction but on different data. These CUs implement an entirely new instruction set that is much simpler for compilers and software developers to use and delivers more consistent performance.

Top two GPU vendors, NVIDIA and AMD, use different names to describe compute units and processing elements. A compute unit is a *stream multiprocessor* in a NVidia GPU or a *SIMD engine* in an AMD GPU. A processing element is a *stream processor* in a NVidia GPU or an *ALU* in an AMD GPU.

instructions. The compiler for such a CPU core should be able to translate the code from Listing 5.1 into the assembly code from Fig. 5.4. When the first instruction is fetched it is dispatched to all eight ALUs within the core. Recall that each ALU has its own set of registers (execution context) so each ALU would add its own `tid` to its own register `r2`. The same holds also for the second and all following instructions in the instruction stream. For example, the instruction

```
lfp f1,r3(vecA)
```

is executed on all ALUs at the same time. This instruction loads the element from vector `vecA` at the address `vecA+r3`. Because the value in `r3` is based on different `tid`, each ALU will operate on different element form vector `vecA`. Most modern GPUs use this approach where the cores execute scalar instructions but one instruction stream is shared across many threads.

In this book, we well refer to a CPU core from Fig. 5.4 as **Compute Unit** (CU) and to ALU as **Processing Element**. Let us summarize the key-features of computer units. We can say that they are general-purpose processors, but they are designed very differently than the general-purpose cores in CPUs—they support so-called SIMD (Single Instruction Multiple Data) parallelism through replication of execution units (ALUs), and corresponding execution contexts, they do not support branch prediction or speculative execution and they have less cache than general-purpose CPUs.

We can further improve the execution speed of our vector addition problem replicating compute units. Figure 5.5 shows a GPU containing 16 compute units. Using 16 compute units as in Fig. 5.5 we can add 128 adjacent vector elements in parallel using one instruction stream. Each CU executes a code snippet in Fig. 5.5, which represents one thread. Let us suppose that we run 128 threads and each thread has its own ID, `tid`, where `tid` is in range 0 . . . 127. The first two instructions load

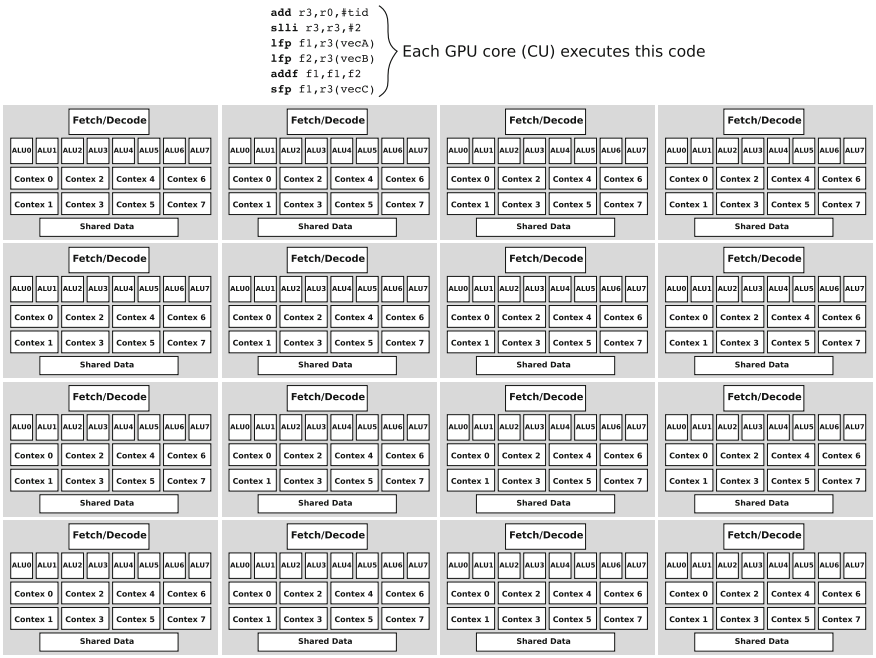


Fig. 5.5 Sixteen compute units each containing eigth processing elements and eigth separate contexts

the thread ID `tid` into `r3` and multiply it by 4 (in order to obtain the correct offset in floating-point vector). Now, the register `r3` that belongs to each thread contains the offset of the vector element that will be accessed in that thread. Each thread then adds two adjacent elements of `vecA` and `vecB` and stores the result into the corresponding element of `vecC`. Because each compute units has eight processing elements (128 processing elements in total), there is no need for the loop. Hopefully, we are now able to understand the basic idea behind modern GPUs: use as many ALUS as possible and let ALUs execute same instructions in a *lock-step basis*, i.e., running the same instruction at the same time but on different data.

5.1.2 A Modern GPU

Modern GPUs comprise of tens of compute units. The efficiency of wide SIMD processing allows GPUs to pack many CU cores densely with processing elements. For example, the NVIDIA GeForce GTX780 GPU contains 2304 processing elements. These processing elements are organized into 12 CU cores (192 PEs per CU). All modern GPUs maintain large numbers of execution contexts on chip to provide maximal memory latency-hiding ability. This represents a significant departure from CPU designs, which attempt to avoid or minimize stalls primarily using large, low-latency data caches and complicated out of order execution logic. Each CU contains

Table 5.1 Comparison of NVIDIA GPU generations

	GeForce GTX280	GeForce GTX580	GeForce GTX780
Microarchitecture	Tesla	Fermi	Kepler
CUs	30	16	12
PEs	240	512	2304
PEs per CU	8	32	192
32-bit registers per CU	16 K	32 K	64 K

thousands of 32-bit registers that are used to store execution context and are evenly allocated to threads (or PEs). Registers are both the fastest and most plentiful memory in the compute unit. As an example, CU in NVIDIA GeForce GTX780 (Kepler microarchitecture) contains 65,536 (64 K) 32-bit registers. To achieve large-scale multithreading, execution contexts must be compact. The number of thread contexts supported by a CU core is limited by the size of on-chip execution context storage. GPUs can manage many thread contexts (and provide maximal latency-hiding ability) when threads use fewer resources. When threads require large amounts of storage, the number of execution contexts (and latency-hiding ability) provided by a GPU drops. Table 5.1 shows the structure of some of the modern NVIDIA GPUs.

5.1.3 Scheduling Threads on Compute Units

The GPU device containing hundreds of simple processing elements is ideally suited for computations that can be run in parallel. That is, data parallelism is optimally handled on the GPU device. This typically involves arithmetic on large data sets (such as vectors, matrices, and images), where the same operation can be performed across thousands, if not millions, of data elements at the same time. To exploit such a huge parallelism, the programmers should partition their programs into thousands of threads and schedule them among compute units. To make it easier to switch to OpenCL later in this chapter, we will now define and use the same thread terminology as OpenCL does. In that sense, we will use the term **work-item** (WI) for a thread.

Work-items (or threads) are actually scheduled among compute units in two steps, which are given as follows:

Work-item (WI) and work-group (WG)

A **work-item** in OpenCL is actually a thread in terms of its control flow and its memory model. Work-items are organized into **work-groups**, which are the unit of work scheduled onto compute units. Because of this, work-groups also define the set of work-items that may share data using local memory and may synchronize at barriers.

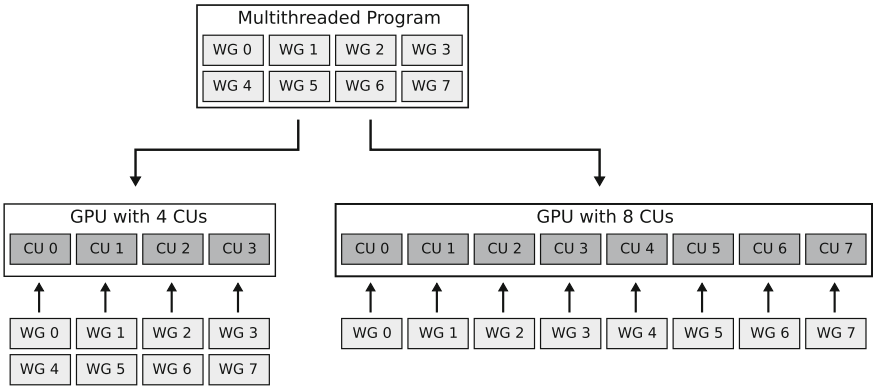


Fig. 5.6 A programmer partitions a program into blocks of *work-item* (threads) called *work-groups*. Work-groups execute independently from each other on CUs. Generally, a GPU with more CUs will execute the program faster than a GPU with fewer CUs

1. First, a programmer explicitly, within a program, partitions work-items into groups called **work-groups** (WG). A work-group is simply a block of work-items that are executed on the same compute unit. Besides that, a work-group also represents a set of work-items that can be synchronized by means of using barriers or memory fences. As a work-group runs on a compute-unit, all work-items within a work-group are able to share local memory that is present within a compute unit (this will be explained in more details in Sect. 5.1.4). After the program has been compiled and sent to execution, the hardware scheduler (which is a part of GPU) evenly assigns work-groups to compute-units. Work-groups execute independently from each other on CUs. If there are more work-groups than CUs, the work-groups are evenly assigned to CUs. Work-groups can be scheduled in any order by the hardware scheduler. In the following sections, we will learn how a programmer partitions a program into work-items and work-groups. Figure 5.6) shows how a multithreaded program is partitioned into work-groups that are assigned to several CUs.
2. Second, the compute unit schedules and executes work-items from the same work-group in groups of 32 parallel work-items called *warps*. When a compute unit is given one or more work-groups to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a work-group is partitioned into warps is always the same; each warp contains work-items of consecutive, increasing work-items IDs with the first warp containing work-item 0. Individual work-items composing a warp start together at the same program address, but they have their own instruction address counter and register state and are, therefore, free to branch and execute independently. However, the best performance is achieved when all work-items from the same warp execute the same instructions.

A warp executes one common instruction at a time. It means that work-items in a warp execute in a so-called *lock-step* basis, running the same instruction but on different data. Full efficiency is thus realized when all 32 work-items in a warp execute the same instruction sequence. If work-items in a warp diverge via a conditional branch (i.e., if we use conditional branches within the code executed by work-items), the warp serially executes each branch path taken, disabling work-items that are not on that path. When all branch paths complete, work-items converge back to the same execution path.

At every instruction issue time, a warp scheduler selects a warp that has work-items ready to execute its next instruction (Fig. 5.7), and issues the instruction to those work-items. Work-items that are ready to execute are called **active work-items**. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called **latency**. Full utilization is achieved when all warp schedulers have some instruction to issue for some warp at every clock cycle during that latency period. In that case, we say that *latency is completely hidden*. The most common reason a warp is not ready to execute its next instruction is that the instruction's input operands are not yet available. Another reason a warp is not ready to execute its next instruction is that it is waiting at some memory fence or barrier. A barrier can force CU to idle as more and more warps wait for other warps in the same work-group to complete execution of instructions prior to the barrier. Full utilization is achieved when more than one work-group is assigned to one CU, so that CU always have 32 work-items from some work-group that are ready to execute and are not waiting at the barrier.

Warp

A *warp* is a group of 32 work-items from the same work-group that are executed in parallel at the same time. Work-items in a warp execute in a so-called *lock-step* basis. Each warp contains work-items of consecutive, increasing work-items IDs. Individual work-items composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. However, the best performance is achieved when all work-items from the same warp execute the same instructions.

If processing elements within a CU remain idle during the period while a warp is stalled, then a GPU is inefficiently utilized. Instead, GPUs maintain more execution contexts on CU than they can simultaneously execute (recall that a huge register file is used to store context for each work-item). In such a way, PEs can execute instructions from active work-items when others are stalled. The execution context (program counters, registers, etc) for each warp processed by a CU is maintained on-chip during the entire lifetime of the warp. Therefore, switching from one execution context to another has no cost. Also, having multiple resident work-groups per CU can help reduce idling in the case of barriers, as warps from different work-groups do not need to wait for each other at barriers.

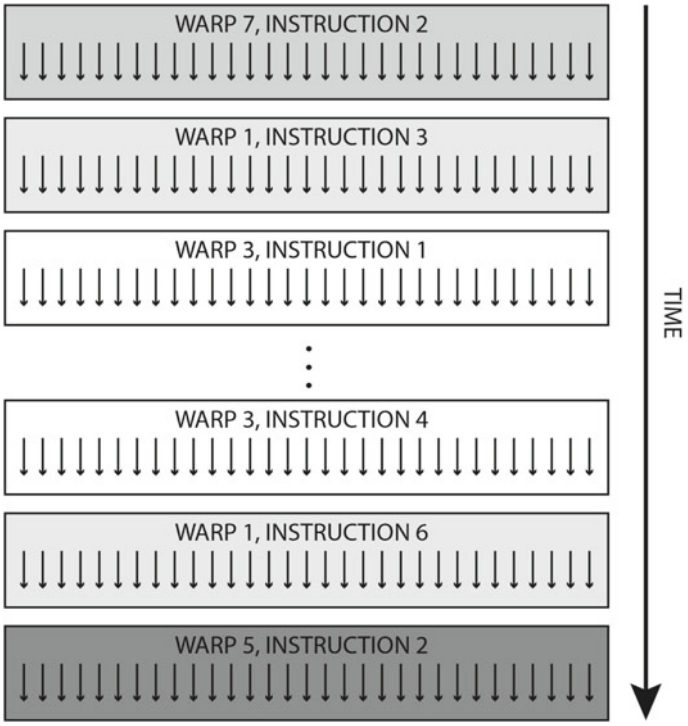


Fig. 5.7 Scheduling of warps within a compute unit. At every instruction issue time, a warp scheduler selects a warp that has work-items ready to execute its next instruction. Each warp always contains work-items of consecutive work-items IDs, but warps are executed out of order

Memory hierarchy on GPU

A GPU device has the following five memory regions accessible from a single work-item:

- Registers
- Local Memory
- Texture Memory
- Constant Memory
- Global Memory

5.1.4 Memory Hierarchy on GPU

Modern GPUs have several memories that can be accessed from a single work-item. Memory hierarchy of a modern GPU is shown in Fig. 5.8. A memory hierarchy has a

Table 5.2 Access time by memory level

Storage type	Access time
Registers	1 cycle
Local memory	1–32 cycles
Texture memory	~500 cycles
Constant memory	~500 cycles
Global memory	~500 cycles

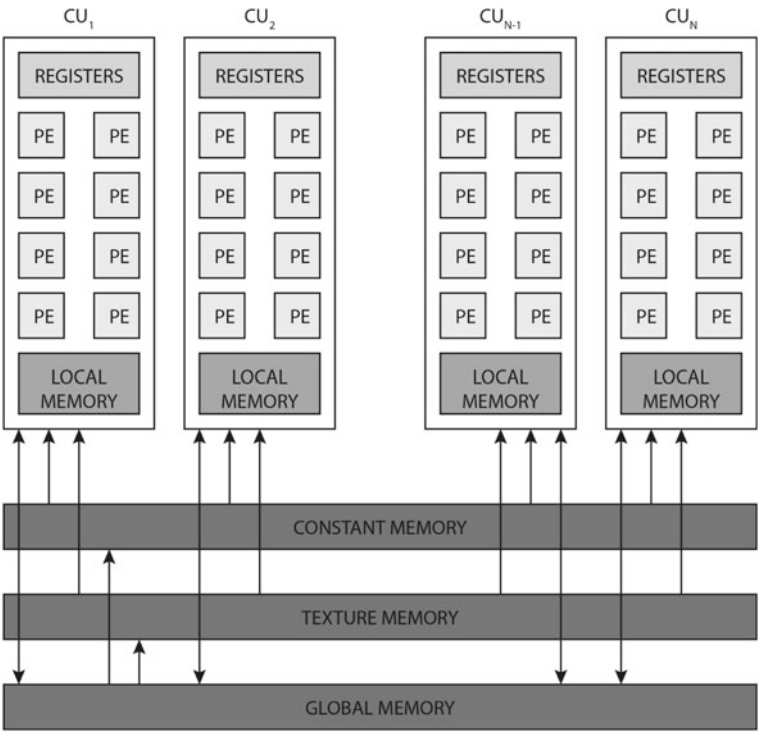


Fig. 5.8 Memory hierarchy on GPU

number of levels of areas where work-items can place data. Each level has its latency (i.e., access time) as shown in Table 5.2.

The GPU has thousands of *registers* per compute unit (CU). The registers are at the first and also the most preferable level, as their access time is 1 cycle. Recall that GPU dedicates real registers to each and every work-item. The number of registers per work-item is calculated at compile time. Depending on the particular microarchitecture of a CU, there are 16 K, 32 K, or 64 K registers for all work-items within an CU. For example, with Kepler microarchitecture you get 64 K of registers per CU. If you decide to partition your program such that there are 256 work-items per work-group, and that there are four work-groups per CU, you will get 65536 / (256*4) = 64 registers per work-item on a CU.

Each CU contains a small amount (64 kB) of very fast on-chip memory that can be accessed from the work-items running at the particular CU. It is mainly used for data interchange within a work-group running on CU. This memory is called *local or shared memory*. Local memory acts as a user-controlled L1 cache. Actually, on modern GPUs, this on-chip memory can be used as a user-controlled local memory or standard hardware-controlled L1 cache. For example, on Kepler CUs this memory can be split of 48 KB local memory/16 KB L1 cache. On CUs with the Tesla microarchitecture, there is 16 kB of local memory and no L1 cache. Local memory has around one-fifth of the speed of registers.

Memory coalescing

Coalesced memory access or *memory coalescing* refers to combining multiple memory accesses into a single transaction. Grouping of work-items into warps is not only relevant to computation, but also to global memory accesses. The GPU device coalesces global memory loads and stores issued by work-items of a warp into as few transactions as possible to minimize DRAM bandwidth. On the recent GPUs, every successive 128 bytes (e.g., 32 single precision words) memory can be accessed by a warp in a single transaction.

The largest memory space on GPU is the **global memory**. The global memory space is implemented in high-speed GDDR, or graphics dynamic memory, which achieves very high bandwidth, but like all memory, has a high latency. GPU global memory is global because it's accessible from both the GPU and the CPU. It can actually be accessed from any device on the PCI-E bus. For example, the GeForce GTX780 GPU has 3 GB of global memory implemented in GDDR5. Global memory resides in device DRAM and it is used for transfers between the host and device as well as for the data input to and output from work-items running on CUs. Reads and writes to global memory are always initiated from CU and are always 128 bytes wide starting at the address aligned at 128-bytes boundary. The blocks of memory that are accessed in one memory transactions are called **segments**. This has an extremely important consequence. If two work-items of the same warp access two data that fall into the same 128-bytes segment, data is delivered in a single transaction. If on the other hand there is data in a segment you fetch that no work-item requested—it is being read anyway and you (probably) waste bandwidth. And if two work-items from the same warp access two data that fall into two different 128-bytes segments, two memory transactions are required. The important thing to remember is that to ensure **memory coalescing** *we want work-items from the same warp to access contiguous elements in memory so to minimize the number of required memory transactions.*

There are also two additional read-only memory spaces within global memory that are accessible by all work-items: **constant memory** and **texture memory**. The constant memory space resides in device memory and is cached. This is where constants and program arguments are stored. Constant memory has two special properties: first, it is cached, and second, it supports broadcasting a single value to all work-items

OpenCL kernel

Code that gets executed on a GPU device is called a **kernel** in OpenCL. The kernels are written in a C dialect, which is mostly straightforward C with a lot of built-in functions and additional data types. The body of a kernel function implements the computation to be completed by all work-items.

within a warp. This broadcast takes place in just a single cycle. Texture memory is cached so an image read costs one memory read from device memory only on a cache miss, otherwise, it just costs one read from the texture cache. The texture cache is optimized for 2D spatial access pattern, so work-items of the same warp that read image addresses that are close together will achieve best performance.

5.2 Programmer's View

So far, we have learned how GPUs are built, what are compute units and processing elements, how work-groups and work-items are scheduled on CUs, which memory is present on a modern GPU, and what is the memory hierarchy of a modern GPU. We have mentioned that a programmer is responsible for partitioning programs into work-groups of work-items. In the following sections, we will learn what is a programmer's view of a heterogeneous system and how to use OpenCL to program for a GPU.

5.2.1 OpenCL

OpenCL (Open Computing Language) is the open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers, mobile devices, and embedded platforms. OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), and other types of processors or hardware accelerators. OpenCL specifies:

- programming language for programming these devices, and
- application programming interface to control the platform and execute programs on the compute devices.

OpenCL defines the OpenCL C programming language that is used to write compute **kernels**—the C like functions that implements the task which is to be executed by all work-items running on a GPU. Unfortunately, OpenCL has one significant

drawback: it is not easy to learn. Even the most introductory application is difficult for a newcomer to grasp. Prior to jump into OpenCL and take advantage of its parallel-processing capabilities, an OpenCL developer needs to clearly understand three basic concepts: heterogeneous system (also called platform model), execution model, and memory model.

5.2.2 Heterogeneous System

A heterogeneous system (also called platform model) consists of a single *host* connected to one or more OpenCL *devices* (e.g., GPUs, FPGA accelerators, DSP or even CPU). The device is where the OpenCL kernels execute. A typical heterogeneous system is shown in Fig. 5.9. An OpenCL program consists of the host program, that runs on the host (typically this is a desktop computer with a general-purpose CPU), and one or more kernels that run on the OpenCL devices. The OpenCL device comprises of several compute units. Each compute unit comprises of tens or hundreds of processing elements.

5.2.3 Execution Model

The OpenCL execution model defines how kernels execute. The most important concept to understand is NDRange (*N-Dimensional Range*) execution. The host program invokes a kernel over an index space. An example of an index space which is easy to understand is a for loop in C. In the for loop defined by the statement `for(int i=0; i<5; i++)`, any statements within this loop will execute five times, with i = 0, 1, 2, 3, 4. In this case, the index space of the loop is [0, 1, 2, 3, 4]. In OpenCL, index space is called NDRange, and can have 1, 2, or 3 dimensions. OpenCL kernel functions are executed exactly one time for each point in the NDRange index space. This unit of work for each point in the NDRange is called a work-item. Unlike for loops in C, where loop iterations are executed sequentially and in-order, an OpenCL device is free to execute work-items in parallel and in any order. Recall that work-items are not scheduled for execution individually onto OpenCL devices.

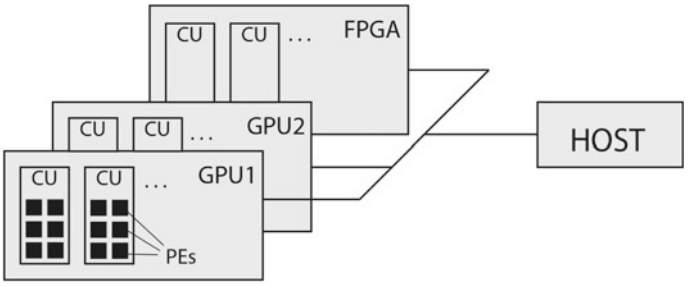


Fig. 5.9 A heterogeneous system

OpenCL execution model

The OpenCL Execution Model: Kernels are executed by one or more work-items. Work-items are collected into work-groups and each work-group executes on a compute unit. Kernels are invoked over an index space called NDRange. A work-item is a single kernel instance at a point in the NDRange. NDRange defines the total number of work-items that execute in parallel. In other words, each work-item executes the same kernel function.

Instead, work-items are organized into work-groups, which are the unit of work scheduled onto compute units. Because of this, work-groups also define the set of work-items that may share data using local memory. Synchronization is possible only between the work-items in a work-group.

Work-items have unique global IDs from the index space. Work-items are further grouped into work-groups and all work-items within a work-group are executed on the same compute unit. Work-groups have a unique work-group ID and work-items have a unique local ID within a work-group. NDRange defines the total number of work-items that execute in parallel. This number is called *global work size* and must be provided by a programmer before the kernel is submitted for execution. The number of work-items within a work-group is called *local work size*. The programmer may also set the local work size at runtime. Work-items within a work-group can communicate with each other and we can synchronize them. In addition, work-items within a work-group are able to share memory. Once the local work size has been determined, the NDRange (global work size) is divided automatically into work-groups, and the work-groups are scheduled for execution on the device.

A kernel function is written on the host. The host program then compiles the kernel and submits the kernel for execution on a device. The host program is thus responsible for creating a collection of work-items, each of which uses the same instruction stream defined by a single kernel. While the instruction stream is the same, each work-item operates on different data. Also, the behavior of each work-item may vary because of branch statements within the instruction stream.

Figure 5.10 shows an example of NDRange where each small square represents a work-item. NDRange in Fig. 5.10 is a two-dimensional index space of size (GX, GY). Each work-item within this NDRange has its own global index (g_x, g_y). For example, the shaded square has global index (10, 12). The work items are grouped into two-dimensional work-groups. Each work-group contains 64 work-items and is of size (LX, LY). Each work-item within a work-group has a unique local index (l_x, l_y). For example, the shaded square has local index (2, 4). Also, each work-group has its own work-group index (w_x, w_y). For example, the work-group containing the shaded square has work-group index (1, 1). And finally, the size of the NDRange index space can be expressed with the number of work-groups in each dimension, (WX, WY).

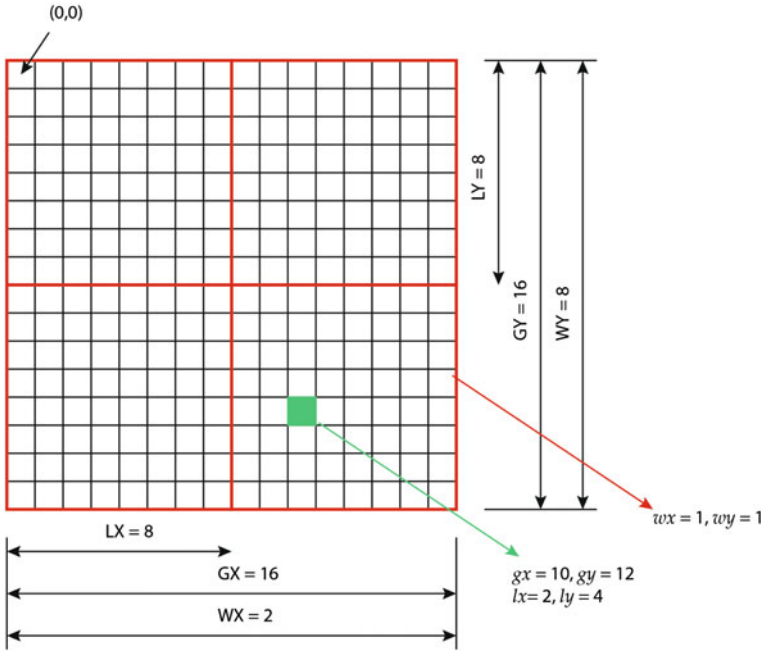


Fig. 5.10 NDRange

OpenCL memory model

The OpenCL memory model: Kernel data must be specifically placed in one of four address spaces: global memory, constant memory, local memory, or private memory. The location of the data determines how quickly it can be processed and how the data is shared within a work-group.

5.2.4 Memory Model

Since common memory address space is unavailable on the host and the OpenCL devices, the OpenCL memory model defines four regions of memory accessible to work-items when executing a kernel. Figure 5.11 shows the regions of memory accessible by the host and the compute device. OpenCL generalizes the different types of memory available on a device into private memory, local memory, global memory, and constant memory, as follows:

1. **Private memory**: a memory region that is private per work item. For example, on a GPU device this would be registered within the compute unit.

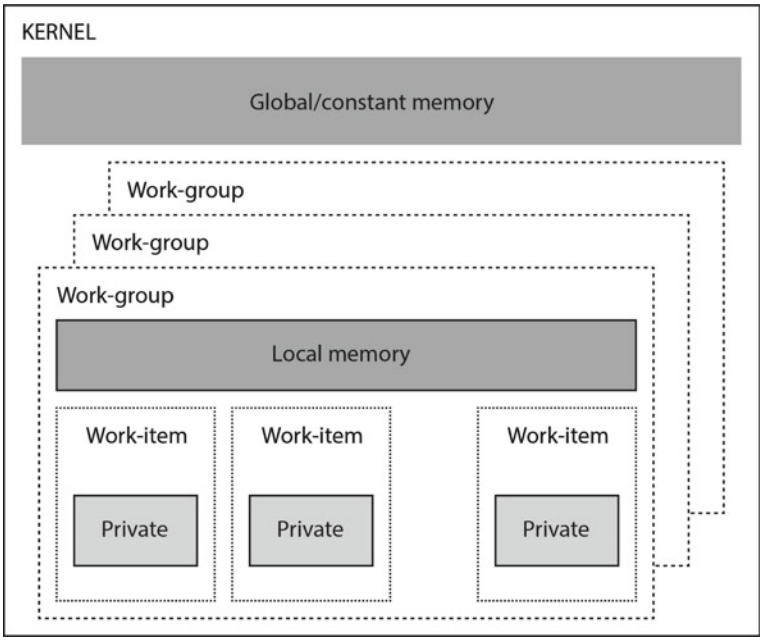


Fig. 5.11 The OpenCL memory model

2. **Local memory**: a memory region that is shared within a work-group. All work-items in the same work-group have both read and write access. On a GPU device, this is local memory within the compute unit.
3. **Global memory**: a memory region in which all work-items and work-groups have read and write access. It is visible to all work-items and all work-groups. On a GPU device, it is implemented in GDDR5. This region of memory can be allocated only by the host during runtime.
4. **Constant memory**: a region of global memory that stays constant throughout the execution of the kernel. Work-items have only read access to this region. The host is permitted both read and write access.

When writing kernels in the OpenCL language, we must declare memory with certain address space qualifiers to indicate whether the data resides in global (`__global`), constant (`__constant`), local (`__local`), or it will default to private within a kernel.

5.3 Programming in OpenCL

5.3.1 A Simple Example: Vector Addition

We will start with a simple C program that adds the adjacent elements of two arrays (vectors). with N elements each. The sample C code for vector addition that is intended to run on a single-core CPU is shown in Listing 5.2.

```
// add the elements of two arrays
void VectorAdd(float *a,
               float *b,
               float *c,
               int iNumElements) {

    int iGID = 0;

    while (iGID < iNumElements) {
        c[iGID] = a[iGID] + b[iGID];
        iGID += 1;
    }
}
```

Listing 5.2 Sequential vector addition

We compute the sum within a `while` loop. The index `iGID` ranges from 0 to `iNumElements - 1`. In each iteration, we add elements `a[iGID]` and `b[iGID]` and place the result in the `c[iGID]`.

Now, we will try to implement the same problem using OpenCL and execute it on a GPU. We will use this simple problem of adding two vectors because the emphasis will be on getting familiar with OpenCL and not on solving the problem itself. We will show how to split the code into two parts: the *kernel function* and the *host code*.

Kernel Function

We can accomplish the same addition on a GPU. To execute the vector addition function on a GPU device, we must write it as a kernel function that is executed on a GPU device. Each thread on the GPU device will then execute the same kernel function. *The main idea is to replace loop iterations with kernel functions executing at each point in a problem domain.* For example, process vectors with `iNumElements` elements with one kernel invocation per element or `iNumElements` threads (kernel executions). The OpenCL kernel is a code sequence that will be executed by every single thread running on a GPU. It is very similar in structure to a C function, but it has the qualifier `__kernel`. This qualifier alerts the compiler that a function is to be compiled to run on an OpenCL device instead of the host. The arguments are passed to a kernel as they are passed to any C function. The arguments in the global memory are described with `__global` qualifier and the arguments in the shared memory are described with `__local` qualifier. These arguments should be always passed as pointers.

OpenCL: Get global ID

The global ID for a working-item in NDRange is obtained by the `get_global_id` function:

`size_t get_global_id (uint dimindx)`

This function returns the unique global work-item ID value for dimension identified by its argument `dimindx`. Valid values of `dimindx` are 0 for the first dimension (row), 1 for the second dimension (column) and 2 for the third dimension in NDRange.

As each thread executing the kernel function operates on its own data, there should be a way to identify the thread end link it with particular data. To determine the thread id, we use the `get_global_id` function, which works for multiple dimensions.

The kernel function should look similar to the function `VectorAdd` from Listing 5.2. If we assume that each work-item calculates one element of array C, the kernel function looks like in Listing 5.3.

```
1  // OpenCL Kernel Function for element by element
2  //  vector addition
3  __kernel void VectorAdd(
4                          __global float* a,
5                          __global float* b,
6                          __global float* c,
7                          int iNumElements
8                          ) {
9
10     //find my global index and handle the data at this index
11     int iGID = get_global_id(0);
12
13     if (iGID < iNumElements) {
14         // add adjacent elements
15         c[iGID] = a[iGID] + b[iGID];
16     }
17 }
```

Listing 5.3 Vector Addition - the kernel function

We intend to run this kernel in `iNumElements` instances so that each work-item in NDRange will operate on one vector element. The kernel function has four arguments. The first two arguments are the pointers to input arrays in global memory, `a` and `b`, namely. The third parameter is the pointer to the output array `c` in global memory. And finally, the fourth argument `iNumElements` is the number of elements in arrays. Instead of summing in a `while` loop, each work-item discovers its global index in NDRange and process only the array elements at this index. For one-dimensional arrays we use one-dimensional index space. To discover its global index in one-dimensional index space, a work-item should call the `get_global_id(0)` function. Prior to run this kernel on a GPU device, we must setup the execution environment in the host code.

Host Code

In developing an OpenCL project, the first step is to code the host application. The host application runs on a user's computer (the host) and dispatches kernels to connected devices. The host application can be coded in C or C++. Because OpenCL supports a wide range of heterogeneous platforms, the programmer must first determine which OpenCL devices are connected to the platform. After he discovers the devices constituting the platform, the programmer chooses one or more devices on which he wants to run the kernel function. Only after that can he compile and execute the kernel function on the selected device. Thus, the kernel functions are compiled in runtime and the compilation process is initiated from the host code.

Prior to execute a kernel function, the host program for a heterogeneous system must carry out the following steps:

1. Discover the OpenCL devices that constitute the heterogeneous system. The OpenCL abstraction of the heterogeneous system is represented with *platform* and *devices*. The platform consists of one or more devices capable of executing the OpenCL kernels.
2. Probe the characteristics of these devices so that the software (kernel functions) can adapt to the specific features.
3. Read the program source containing the kernel function(s) and compile the kernel(s) that will run on the selected device(s).
4. Set up memory objects on the selected device(s) that will hold the data for the computation.
5. Compile and run the kernel(s) on the selected device(s).
6. Collect the final result from device(s).

The host code can be very difficult to understand for the beginner, but we will soon realize that a large part of the host code is repeated and can be reused in different applications. Once we understand the host code, we will only devote our attention to writing kernel functions. The above steps are accomplished through the following series of calls to OpenCL API within the host code:

1. Prepare and initialize data on host.
2. Discover and initialize the devices.
3. Create a context.
4. Create a command queue.
5. Create the program object for a context.
6. Build the OpenCL program.
7. Create device buffers.
8. Write host data to device buffers.
9. Create and compile the kernel.
10. Set the kernel arguments.
11. Set the execution model and enqueue the kernel for execution.
12. Read the output buffer back to the host.

Every host application requires five data structures: `cl_device_id`, `cl_context`, *cl_command_queue*, `cl_program`, and `cl_kernel`. This data structures must be initialized and filled-in prior to enqueue the kernel function for execution. Listing 5.4 shows the host code. In paragraphs that follows we explain each step within the host code and briefly describe the OpenCL API function used to accomplish the step. For more detailed description of API calls, you should refer to OpenCL™ 2.2 Specification.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <OpenCL/opencl.h>

cl_int status;
cl_int ciErr;
cl_device_id *devices = NULL;
cl_uint numDevices = 0;
char buffer[100000];
cl_uint buf_uint;
cl_ulong buf_ulong;
size_t buf_sizet;
cl_int iNumElements = 512*512;

cl_float* srcA;
cl_float* srcB;
cl_float* srcC;
cl_float result;

FILE* programHandle;            // File that contains kernel functions
size_t programSize;
char *programBuffer;
cl_program cpProgram;           // OpenCL program
cl_kernel ckKernel;             // OpenCL kernel

size_t szGlobalWorkSize;        // global work size
size_t szLocalWorkSize;         // local work size

// Main function
// ****************************************************
int main(int argc, char **argv)
{
    // set and log Global and Local work size dimensions
    szLocalWorkSize = 512;
    szGlobalWorkSize = iNumElements;
    // Allocate host arrays
    srcA = (void *)malloc(sizeof(cl_float) * iNumElements);
    srcB = (void *)malloc(sizeof(cl_float) * iNumElements);
    srcC = (void *)malloc(sizeof(cl_float) * iNumElements);
    // init arrays:
    for (int i = 0; i<iNumElements; i++ ) {
        *((cl_float*)srcA + i) = 1.0;
        *((cl_float*)srcB + i) = 1.0;
    }

    //************************************************
    // STEP 1: Discover and initialize the devices
    //************************************************
    // Use clGetDeviceIDs() to retrieve the number of
```

```
    // devices present
    status = clGetDeviceIDs(
                            NULL,
                            CL_DEVICE_TYPE_ALL,
                            0,
                            NULL,
                            &numDevices);
    if (status != CL_SUCCESS)
    {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
    }

    printf("The number of devices found = %d \n", numDevices);

    // Allocate enough space for each device
    devices = (cl_device_id*) malloc(numDevices*sizeof(cl_device_id));
    // Fill in devices with clGetDeviceIDs()
    status = clGetDeviceIDs(
                            NULL,
                            CL_DEVICE_TYPE_ALL,
                            numDevices,
                            devices,
                            NULL);
    if (status != CL_SUCCESS)
    {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
    }

    //*************************************************
    // STEP 2: Create a context
    //*************************************************

    cl_context context = NULL;
    // Create a context using clCreateContext() and
    // associate it with the devices
    context = clCreateContext(
                              NULL,
                              numDevices,
                              devices,
                              NULL,
                              NULL,
                              &status);
    if (!context)
    {
        printf("Error: Failed to create a compute context!\n");
        return EXIT_FAILURE;
    }

    //*************************************************
    // STEP 3: Create a command queue
    //*************************************************
    cl_command_queue cmdQueue;
    // Create a command queue using clCreateCommandQueue(),
    // and associate it with the device you want to execute
    // on
    cmdQueue = clCreateCommandQueue(
                                    context,
                                    devices[1], // GPU
                                    CL_QUEUE_PROFILING_ENABLE,
                                    &status);

    if (!cmdQueue)
    {
        printf("Error: Failed to create a command commands!\n");
        return EXIT_FAILURE;
```

```
    }

    //*****************************************************
    // STEP 4: Create the program object for a context
    //*****************************************************
    // 4 a: Read the OpenCL kernel from the source file and
    //      get the size of the kernel source
    programHandle = fopen("/Users/patriciobulic/FRICL/VectorAdd.cl", "r")↩
     ;
    fseek(programHandle, 0, SEEK_END);
    programSize = ftell(programHandle);
    rewind(programHandle);

    printf("Program size = %lu B \n", programSize);

    // 4 b: read the kernel source into the buffer programBuffer
    //      add null-termination-required by clCreateProgramWithSource
    programBuffer = (char*) malloc(programSize + 1);

    programBuffer[programSize] = '\0'; // add null-termination
    fread(programBuffer, sizeof(char), programSize, programHandle);
    fclose(programHandle);

    // 4 c: Create the program from the source
    //
    cpProgram = clCreateProgramWithSource(
                   context,
                   1,
                   (const char **)&programBuffer,
                   &programSize,
                   &ciErr);
    if (!cpProgram)
    {
        printf("Error: Failed to create compute program!\n");
        return EXIT_FAILURE;
    }
    free(programBuffer);

    //*****************************************************
    // STEP 5: Build the program
    //*****************************************************
    ciErr = clBuildProgram(
                           cpProgram,
                           0,
                           NULL,
                           NULL,
                           NULL,
                           NULL);

    if (ciErr != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];

        printf("Error: Failed to build program executable!\n");
        clGetProgramBuildInfo(cpProgram,
                              devices[1],
                              CL_PROGRAM_BUILD_LOG,
                              sizeof(buffer),
                              buffer,
                              &len);
        printf("%s\n", buffer);
        exit(1);
    }

    //*****************************************************
    // STEP 6: Create device buffers
```

```
    //*************************************************
    cl_mem bufferA; // Input array on the device
    cl_mem bufferB; // Input array on the device
    cl_mem bufferC; // Output array on the device
    //cl_mem noElements;

    // Size of data:
    size_t datasize = sizeof(cl_float) * iNumElements;

    // Use clCreateBuffer() to create a buffer object (d_A)
    // that will contain the data from the host array A
    bufferA = clCreateBuffer(
                            context,
                            CL_MEM_READ_ONLY,
                            datasize,
                            NULL,
                            &status);

    // Use clCreateBuffer() to create a buffer object (d_B)
    // that will contain the data from the host array B
    bufferB = clCreateBuffer(
                            context,
                            CL_MEM_READ_ONLY,
                            datasize,
                            NULL,
                            &status);

    // Use clCreateBuffer() to create a buffer object (d_C)
    // with enough space to hold the output data
    bufferC = clCreateBuffer(
                            context,
                            CL_MEM_WRITE_ONLY,
                            datasize,
                            NULL,
                            &status);

    //*************************************************
    // STEP 7: Write host data to device buffers
    //*************************************************
    // Use clEnqueueWriteBuffer() to write input array A to
    // the device buffer bufferA
    status = clEnqueueWriteBuffer(
                                cmdQueue,
                                bufferA,
                                CL_FALSE,
                                0,
                                datasize,
                                srcA,
                                0,
                                NULL,
                                NULL);

    // Use clEnqueueWriteBuffer() to write input array B to
    // the device buffer bufferB
    status = clEnqueueWriteBuffer(
                                cmdQueue,
                                bufferB,
                                CL_FALSE,
                                0,
                                datasize,
                                srcB,
                                0,
                                NULL,
                                NULL);

    //*************************************************
    // STEP 8: Create and compile the kernel
```

```
    //*****************************************************
    // Create the kernel
    ckKernel = clCreateKernel(
                               cpProgram,
                               "VectorAdd",
                               &ciErr);
    if (!ckKernel || ciErr != CL_SUCCESS)
    {
        printf("Error: Failed to create compute kernel!\n");
        exit(1);
    }

    //*****************************************************
    // STEP 9: Set the kernel arguments
    //*****************************************************
    // Set the Argument values
    ciErr = clSetKernelArg(ckKernel,
                           0,
                           sizeof(cl_mem),
                           (void*)&bufferA);
    ciErr |= clSetKernelArg(ckKernel,
                            1,
                            sizeof(cl_mem),
                            (void*)&bufferB);
    ciErr |= clSetKernelArg(ckKernel,
                            2,
                            sizeof(cl_mem),
                            (void*)&bufferC);
    ciErr |= clSetKernelArg(ckKernel,
                            3,
                            sizeof(cl_int),
                            (void*)&iNumElements);

    //*****************************************************
    // Start Core sequence... copy input data to GPU, compute,
    //      copy results back

    //*****************************************************
    // STEP 10: Enqueue the kernel for execution
    //*****************************************************
    // Launch kernel
    ciErr = clEnqueueNDRangeKernel(
                                    cmdQueue,
                                    ckKernel,
                                    1,
                                    NULL,
                                    &szGlobalWorkSize,
                                    &szLocalWorkSize,
                                    0,
                                    NULL,
                                    NULL);
    if (ciErr != CL_SUCCESS)
    {
        printf("Error launchung kernel!\n" );
    }

    // Wait for the command commands to get serviced before
    //  reading back results
    //
    clFinish(cmdQueue);

    //*****************************************************
    // STEP 11: Read the output buffer back to the host
    //*****************************************************
    // Synchronous/blocking read of results
ciErr = clEnqueueReadBuffer(
                                cmdQueue,
```

```
                                bufferC,
                                CL_TRUE,
                                0,
                                datasize,
                                srcC,
                                0,
                                NULL,
                                NULL);

    // Wait for the command commands to get serviced before reading back ↩
     results
    clFinish(cmdQueue);

    // check the result
    result = 0.0;
    for (int i=0; i<iNumElements; i++) {
        result += srcC[i];
    }
    printf("Result = %f \n", result);

    // Cleanup
    free(srcA);
    free(srcB);
    free(srcC);

    if(ckKernel) clReleaseKernel(ckKernel);
    if(cpProgram) clReleaseProgram(cpProgram);
    if(cmdQueue) clReleaseCommandQueue(cmdQueue);
    if(context) clReleaseContext(context);

    if(bufferA) clReleaseMemObject(bufferA);
    if(bufferB) clReleaseMemObject(bufferB);
    if(bufferC) clReleaseMemObject(bufferC);

    return 0;
}
```

Listing 5.4 Host code for vector addition

1. Discover and initialize the devices
Every OpenCL program requires an OpenCL context, including a list of all OpenCL devices available on the platform. To discover and initialize the devices, we use the `clGetDeviceIDs()` function. We must call the `clGetDeviceIDs()` function for two times. In the first call, we use `clGetDeviceIDs()` to retrieve the number of the OpenCL devices present on the platform. The code is shown in Listing 5.5. The number of OpenCL devices is returned in `num_Devices`. Once we know how many OpenCL devices are available on the platform, we can obtain the list of all devices available on a platform with the second call of the `clGetDeviceIDs()` function.

The sample code for discovering OpenCL devices is shown in Listing 5.6.

```
    //*****************************************************
    // STEP 1: Discover and initialize the devices
    //*****************************************************

    // Use clGetDeviceIDs() to retrieve the number of
    // devices present
    status = clGetDeviceIDs(
                            NULL,
                            CL_DEVICE_TYPE_ALL,
                            0,
```

OpenCL: Get device ID

To obtain the list of devices available on a platform we use the `clGetDeviceIDs()` function:

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

`clGetDeviceIDs` returns CL_SUCCESS if the function is executed successfully. Parameters are:

- `platform`: Refers to the platform ID or can be NULL.
- `device_type`: A bitfield that identifies the type of OpenCL device. Some of valid values are `CL_DEVICE_TYPE_CPU`, `CL_DEVICE_TYPE_GPU` and `CL_DEVICE_TYPE_ALL`. For all other values refer to OpenCL 2.2 Specification.
- `num_entries`: The number of `cl_device` entries that can be added to devices. If devices is not NULL, the `num_entries` must be greater than zero.
- `devices`: A list of OpenCL devices found. In the case that this is NULL, then `clGetDeviceIDs` returns the number of devices in `num_devices`. Otherwise it returns a pointer to the list of available OpenCL devices in `devices`.
- `num_devices`: The number of OpenCL devices available that match `device_type`. If `num_devices` is NULL, this argument is ignored.

Refer to OpenCL™ 2.2 Specification for more detailed description.

```
                            NULL,
                            &numDevices);
    if (status != CL_SUCCESS)
    {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
    }

    printf("The number of devices found = %d \n", numDevices);

    // Allocate enough space for each device
    devices = (cl_device_id*) malloc(numDevices*sizeof(↩
    cl_device_id));
    // Fill in devices with clGetDeviceIDs()
    status = clGetDeviceIDs(
                            NULL,
                            CL_DEVICE_TYPE_ALL,
                            numDevices,
                            devices,
                            NULL);
    if (status != CL_SUCCESS)
    {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
```

OpenCL: Get device info

To get information about an OpenCL device available on a platform we use the `clGetDeviceInfo()` function:

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

clGetDeviceInfo returns CL_SUCCESS if the function is executed successfully. Parameters are:

- `device`: Refers to the device returned by `clGetDeviceIDs`.
- `param_name`: An enumeration constant that identifies the device information being queried. Some of valid values are `CL_DEVICE_MAX_COMPUTE_UNITS`, `CL_DEVICE_MAX_WORK_GROUP_SIZE`, `CL_DEVICE_TYPE`, etc. For all other values refer to OpenCL 2.2 Specification.
- `param_value_size`: Specifies the size in bytes of memory pointed to by `param_value`.
- `param_value`: A pointer to memory location where appropriate values for a given `param_name` as specified in the table below will be returned.Specifies the size in bytes of memory pointed to by `param_value`.
- `param_value_size_ret`: Returns the actual size in bytes of data being queried by `param_value`. If `param_value_size_ret` is NULL, it is ignored.

Refer to OpenCL™ 2.2 Specification for more detailed description.

```
34     }
```

Listing 5.5 Discover and initialize devices

the first call is used to discover the number of present devices. This number is returned in the `numDevices` variable. On an Apple laptop with an Intel GPU, there are two discovered devices:

```
The number of devices found = 2
```

Once we know the number of devices, we make enough space in `devices` buffer with `malloc()`, and then we make the second call to `clGetDeviceIDs()` to obtain the list of all devices in the `devices` buffer.

We can get and print information about an OpenCL device with the `clGetDeviceInfo()` function.

The sample code for printing information about discovered OpenCL devices is shown in Listing 5.6.

```
printf("=== OpenCL devices: ===\n");
for (int i=0; i<numDevices; i++)
{
    printf("  -- The device with the index %d --\n", i);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_NAME,
                    sizeof(buffer),
                    buffer,
                    NULL);
    printf("  DEVICE_NAME = %s\n", buffer);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_VENDOR,
                    sizeof(buffer),
                    buffer,
                    NULL);
    printf("  DEVICE_VENDOR = %s\n", buffer);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_MAX_COMPUTE_UNITS,
                    sizeof(buf_uint),
                    &buf_uint,
                    NULL);
    printf("  DEVICE_MAX_COMPUTE_UNITS = %u\n",
           (unsigned int)buf_uint);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_MAX_WORK_GROUP_SIZE,
                    sizeof(buf_sizet),
                    &buf_sizet,
                    NULL);
    printf("  CL_DEVICE_MAX_WORK_GROUP_SIZE = %u\n",
           (unsigned int)buf_uint);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
                    sizeof(buf_uint),
                    &buf_uint,
                    NULL);
    printf("  CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS = %u\n",
           (unsigned int)buf_uint);

    size_t workitem_size[3];
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_MAX_WORK_ITEM_SIZES,
                    sizeof(workitem_size),
                    &workitem_size,
                    NULL);
    printf("  CL_DEVICE_MAX_WORK_ITEM_SIZES = %u, %u, %u \n",
           (unsigned int)workitem_size[0],
           (unsigned int)workitem_size[1],
           (unsigned int)workitem_size[2]);
}
```

Listing 5.6 Print devices' information

The following is the output of the code in Listing 5.6 for an Apple laptop with an Intel GPU:

```
=== OpenCL devices found on platform: ===
  -- Device 0 --
  DEVICE_NAME = Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz
  DEVICE_VENDOR = Intel
  DEVICE_MAX_COMPUTE_UNITS = 8
  CL_DEVICE_MAX_WORK_GROUP_SIZE = 2200
  CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS = 3
  CL_DEVICE_MAX_WORK_ITEM_SIZES = 1024, 1, 1
```

```
-- Device 1 --
DEVICE_NAME = Iris Pro
DEVICE_VENDOR = Intel
DEVICE_MAX_COMPUTE_UNITS = 40
CL_DEVICE_MAX_WORK_GROUP_SIZE = 1200
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS = 3
CL_DEVICE_MAX_WORK_ITEM_SIZES = 512, 512, 512
```

We can use this information about an OpenCL device later in our host program to automatically adapt the kernel to the specific features. In the above example, the device with index 1 is an Intel Iris Pro GPU. It has 40 compute units, each work-group can have up to 1200 work-items, which can span into three-dimensional NDRange and the maximum size in each dimension is 512. We can also see, that the device with index 0 is an Intel Core i7 CPU, which can also execute OpenCL code. It has eight compute units (four cores, two hardware threads per core).

2. Create a context

Once we have discovered the available OpenCL devices on the platform and have obtained ad least one device ID, we can create an OpenCL context. The context is used to group devices and memory objects together and to manage command queues, program objects, and kernel objects. An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program, and kernel objects and for executing kernels on one or more devices specified in the context. An OpenCL context is created with the `clCreateContext` function. Listing 5.7 shows a call to this function.

```
1  //*************************************************
2  // STEP 2: Create a context
3  //*************************************************
4
5  cl_context context = NULL;
6  // Create a context using clCreateContext() and
7  // associate it with the devices
8  context = clCreateContext(
9                            NULL,
10                           numDevices,
11                           devices,
12                           NULL,
13                           NULL,
14                           &status);
15 if (!context)
16 {
17     printf("Error: Failed to create a compute context!\n");
18     return EXIT_FAILURE;
19 }
```

Listing 5.7 Create a context

3. Create a command queue

When the context is created, command queues are created that allow commands to be sent to the OpenCL devices associated to the context. Commands are placed into the command queue in order the calls are made. The most common use of queues is to enqueue OpenCL kernel functions for execution on a device. The

OpenCL: Create a context

An OpenCL context is created by the `clCreateContext()` function:

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

An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context. `clCreateContext` returns a valid non-zero context and `errcode_ret` is set to CL_SUCCESS if the context is created successfully. Otherwise, it returns NULL value with the values returned in `errcode_ret`. Refer to OpenCL™ 2.2 Specification for more detailed description.

`clCreateCommandQueue` function is used to create a command queue. By enqueuing commands, we request that the OpenCL device execute the operations in the order. If we have multiple OpenCL devices, we must create a command queue for each OpenCL device and submit commands separately to each. Listing 5.8 shows how to create a command queue for an OpenCL device.

```
1  //*************************************************
2  // STEP 3: Create a command queue
3  //*************************************************
4
5  cl_command_queue cmdQueue;
6  // Create a command queue using clCreateCommandQueue(),
7  // and associate it with the device you want to execute
8  // on
9  cmdQueue = clCreateCommandQueue(
10                                 context,
11                                 devices[1], // GPU
12                                 CL_QUEUE_PROFILING_ENABLE,
13                                 &status);
14
15 if (!cmdQueue)
16 {
17     printf("Error: Failed to create a command commands!\n");
18     return EXIT_FAILURE;
19 }
```

Listing 5.8 Create a command queue

OpenCL: Create a command queue

To create a command-queue a specific device use the `clCreateCommandQueue()` function:

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

`clCreateCommandQueue` returns a valid non-zero command-queue and `errcode_ret` is set to CL_SUCCESS if the command-queue is created successfully. Otherwise, it returns NULL value with the values returned in `errcode_ret`. The third argument specifies if profiling is enabled (`CL_QUEUE_PROFILING_ENABLE`) to measure execution time of commands or disabled (0). Refer to OpenCL™ 2.2 Specification for more detailed description.

4. Create the program object for a context
An OpenCL program consists of a set of kernel functions that are identified as functions declared with the `__kernel` qualifier in the program source. Kernel functions are functions that are executed on a particular OpenCL device. OpenCL programs may also contain auxiliary functions and constant data that can be used by `__kernel` functions.

OpenCL allows applications to create a program object using the program source or binary and build appropriate program executables. This allows applications to determine whether they want to use the pre-built offline binary or load and compile the program source and use the executable compiled/linked online as the program executable. This can be very useful as it allows applications to load and build program executables online on its first instance for appropriate OpenCL devices in the system. These executables can now be queried and cached by the application. Future instances of the application launching will no longer need to compile and build the program executables. The cached executables can be read and loaded by the application, which can help significantly reduce the application initialization time.

To create a program object, we use the `clCreateProgramWithSource` function. Listing 5.9 shows how to:

- read the OpenCL kernel from the source file `VectorAdd.cl`,
- read the kernel source into the buffer `programBuffer`, and
- create the program from the source.

```
1  //*****************************************************
2  // STEP 4: Create the program object for a context
3  //*****************************************************
4
```

OpenCL: Create a program object

To create a program object for a context use the `clCreateProgramWithSource()` function:

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

`clCreateProgramWithSource` creates a program object for a context, and loads the source code specified by the text strings in the strings array into the program object. `clCreateCommandQueue` returns a valid non-zero program object and `errcode_ret` is set to CL_SUCCESS if the program object is created successfully. Otherwise, it returns NULL value with the values returned in `errcode_ret`. Refer to OpenCL™ 2.2 Specification for more detailed description.

```
5     // 4 a: Read the OpenCL kernel from the source file and
6     //       get the size of the kernel source
7     programHandle = fopen("/Users/patriciobulic/FRICL/VectorAdd↩
      .cl", "r");
8     fseek(programHandle, 0, SEEK_END);
9     programSize = ftell(programHandle);
10    rewind(programHandle);
11
12    printf("Program size = %d B \n", programSize);
13
14    // 4 b: read the kernel source into the buffer ↩
      programBuffer
15    //       add null-termination-required by ↩
      clCreateProgramWithSource
16    programBuffer = (char*) malloc(programSize + 1);
17
18    programBuffer[programSize] = '\0'; // add null-termination
19    fread(programBuffer, sizeof(char), programSize, ↩
      programHandle);
20    fclose(programHandle);
21
22    // 4 c: Create the program from the source
23    //
24    cpProgram = clCreateProgramWithSource(
25                   context,
26                   1,
27                   (const char **)&programBuffer,
28                   &programSize,
29                   &ciErr);
30    if (!cpProgram)
31    {
32        printf("Error: Failed to create compute program!\n");
33        return EXIT_FAILURE;
34    }
35
36    free(programBuffer);
```

Listing 5.9 Create the program object for a context

OpenCL: Build a program executable

To build (compile and link) a program executable from the program source or binary use the `clBuildProgram()` function:

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

`clBuildProgram` returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of errors. Refer to OpenCL™ 2.2 Specification for more detailed description.

5. Build the program

Once we have created a program object using the function `clCreateProgramWithSource`, we must build a program executable from the contents of the program object. Building the program compiles the source code in the program object and links the compiled code into an executable program. The program object can be built for one or more OpenCL devices using the function `clBuildProgram`. This function builds (compiles and links) a program executable from the program source or binary.The function `clBuildProgram` modifies the program object to include the executable, the build log and build options. Listing 5.10 shows how to build the program and read build information for the selected device in the program object in the case when the build process fails.

```
1   //*****************************************************
2   // STEP 5: Build the program
3   //*****************************************************
4
5   ciErr = clBuildProgram(
6                          cpProgram,
7                          0,
8                          NULL,
9                          NULL,
10                         NULL,
11                         NULL);
12
13  if (ciErr != CL_SUCCESS)
14  {
15      size_t len;
16      char buffer[2048];
17
18      printf("Error: Failed to build program executable!\n");
19      clGetProgramBuildInfo(cpProgram,
20                            devices[1],
21                            CL_PROGRAM_BUILD_LOG,
22                            sizeof(buffer),
23                            buffer,
```

OpenCL: Create a buffer

To create a device buffer use the `clCreateBuffer` function:

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

This function creates a buffer object within the context `context` of the size `size` bytes using flags `flags`. The pointer to the allocated buffer data `host_ptr` holds the address in the device memory. It returns a valid non-zero buffer object and `errcode_ret` is set to CL_SUCCESS if the program object is created successfully. Otherwise, it returns NULL value with the values returned in `errcode_ret`.
A bit-field `flags` is used to specify allocation and usage information such as the memory arena that should be used to allocate the buffer object and how it will be used. The following are some of the possible values for flags:

CL_MEM_READ_WRITE This flag specifies that the memory object will be read and written by a kernel. This is the default.

CL_MEM_WRITE_ONLY This flags specifies that the memory object will be written but not read by a kernel. Reading from a buffer object created with CL_MEM_WRITE_ONLY inside a kernel is undefined.

CL_MEM_READ_ONLY This flag specifies that the memory object is a read-only memory object when used inside a kernel. Writing to a buffer or image object created with CL_MEM_READ_ONLY inside a kernel is undefined.

Refer to OpenCL™ 2.2 Specification for more detailed description.

```
24                                  &len);
25          printf("%s\n", buffer);
26          exit(1);
27      }
```

Listing 5.10 Build the program

6. Create device buffers
Memory objects are reserved regions of global device memory that contains our data. There are two types of memory objects: device buffers and image objects. In this book, we use only device buffers. To create a device buffer we use the `clCreateBuffer` function.

One important thing to remember is that we should never try to de-reference the device pointer from the host code as the device memory is not directly accessible from the host, i.e., we cannot use these pointers to buffer objects to read or write memory from code that executes on the host. We use these pointers to read or write

memory from code that execute on device. Also, we pass these pointers as arguments to kernels, i.e., functions that execute on device. To read or write to device buffers from the host, we must use OpenCL dedicated functions `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`. Upon creation, the contents of the device buffers are undefined. We must explicitly fill the device buffers with our data from the host application. We will show this in the next subsection.

Listing 5.11 shows how to create three device buffers: `bufferA` and `bufferB` that are read-only and are used to store input vectors; and `bufferC` that is write-only and used to store the result of vector addition.

```
//*****************************************************
// STEP 6: Create device buffers
//*****************************************************

cl_mem bufferA; // Input array on the device
cl_mem bufferB; // Input array on the device
cl_mem bufferC; // Output array on the device
//cl_mem noElements;

// Size of data:
size_t datasize = sizeof(cl_float) * iNumElements;

// Use clCreateBuffer() to create a buffer object (d_A)
// that will contain the data from the host array A
bufferA = clCreateBuffer(
                          context,
                          CL_MEM_READ_ONLY,
                          datasize,
                          NULL,
                          &status);

// Use clCreateBuffer() to create a buffer object (d_B)
// that will contain the data from the host array B
bufferB = clCreateBuffer(
                          context,
                          CL_MEM_READ_ONLY,
                          datasize,
                          NULL,
                          &status);

// Use clCreateBuffer() to create a buffer object (d_C)
// with enough space to hold the output data
bufferC = clCreateBuffer(
                          context,
                          CL_MEM_WRITE_ONLY,
                          datasize,
                          NULL,
                          &status);
```

Listing 5.11 Create device buffers

7. Write host data to device buffers

After we have created the device buffers, we can enqueue reads and writes. To write data from host memory to a device buffer, we use the `clEnqueueWriteBuffer` function. We use this function to provide data for processing by a kernel executing on the device. Listing 5.12 shows how to write host data (input vectors `srcA` and `srcB`) to the device buffers `bufferA` and `buffferB`. The device buffers will be then accessed within the kernel.

OpenCL: Write to a buffer

To enqueue commands to write to a buffer object from the host memory use the `clEnqueueWriteBuffer` function:

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

This function enqueues commands into command queue `command_queue` to write `cb` bytes to a device buffer `buffer` from host memory pointed by `ptr`. This function does not block by default. To know when the command has completed, we can use a blocking form of the command by setting the `blocking_write` parameter to `CL_TRUE`. Refer to OpenCL™ 2.2 Specification for more detailed description.

```
1  //*************************************************
2  // STEP 7: Write host data to device buffers
3  //*************************************************
4  // Use clEnqueueWriteBuffer() to write input array A to
5  // the device buffer bufferA
6  status = clEnqueueWriteBuffer(
7                                 cmdQueue,
8                                 bufferA,
9                                 CL_FALSE,
10                                0,
11                                datasize,
12                                srcA,
13                                0,
14                                NULL,
15                                NULL);
16
17 // Use clEnqueueWriteBuffer() to write input array B to
18 // the device buffer bufferB
19 status = clEnqueueWriteBuffer(
20                                cmdQueue,
21                                bufferB,
22                                CL_FALSE,
23                                0,
24                                datasize,
25                                srcB,
26                                0,
27                                NULL,
28                                NULL);
```

Listing 5.12 Write host data to device buffers

OpenCL: Create a kernel object

To create a kernel object use the `clCreateKernel` function:

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

This function creates a kernel object from a function `kernel_name` contained within a program object `program` with a successfully built executable. Refer to OpenCL™ 2.2 Specification for more detailed description.

8. Create and compile the kernel

A kernel is a function we declare in an OpenCL program and is executed on the OpenCL device. We must identify kernels with the `__kernel` qualifier to let OpenCL know that the function is a kernel function. The kernel object is created after the executable has been successfully built in the program object. A kernel object is a data structure that includes the kernel function and the data on which the kernel operates. To create a single kernel object we use the `clCreateKernel` function. Before the kernel object is submitted to the command queue for execution, input or output buffers must be provided for any arguments required by the kernel function. If the arguments use device buffers, they must be created first and the data must be explicitly copied into the device buffers. Listing 5.13 shows how to create a kernel object from the kernel function `VectorAdd()`.

```
1   //*****************************************************
2   // STEP 8: Create and compile the kernel
3   //*****************************************************
4
5   // Create the kernel
6   ckKernel = clCreateKernel(
7                             cpProgram,
8                             "VectorAdd",
9                             &ciErr);
10  if (!ckKernel || ciErr != CL_SUCCESS)
11  {
12      printf("Error: Failed to create compute kernel!\n");
13      exit(1);
14  }
```

Listing 5.13 Create and compile the kernel

9. Set the kernel arguments

Prior to enqueue the kernel function for execution on device, we must set the kernel arguments. When the required memory objects have been successfully created, kernel arguments can be set using the `clSetKernelArg` function. Listing 5.14 shows how to set arguments for the kernel function `VectorAdd()`. In this example, the input arguments have indices 0, 1, and 3 and the output argument has index 2.

OpenCL: Set kernel arguments

To set the argument value for a specific argument of a kernel use the `clSetKernelArg` function:

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

Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to $n - 1$, where n is the total number of arguments declared by a kernel. The argument `index` refers to the specific argument position of the kernel definition that must be set. The last two arguments of `clSetKernelArg` specify the size of the argument data and the pointer to the actual data that should be used as the argument value. If a kernel function argument is declared to be a pointer of a built-in or user defined type with the __global or __constant qualifier, a buffer memory object must be used. Refer to OpenCL™ 2.2 Specification for more detailed description.

```
1  //*************************************************
2  // STEP 9: Set the kernel arguments
3  //*************************************************
4  // Set the Argument values
5  ciErr = clSetKernelArg(ckKernel,
6                         0,
7                         sizeof(cl_mem),
8                         (void*)&bufferA);
9  ciErr |= clSetKernelArg(ckKernel,
10                         1,
11                         sizeof(cl_mem),
12                         (void*)&bufferB);
13 ciErr |= clSetKernelArg(ckKernel,
14                         2,
15                         sizeof(cl_mem),
16                         (void*)&bufferC);
17 ciErr |= clSetKernelArg(ckKernel,
18                         3,
19                         sizeof(cl_int),
20                         (void*)&iNumElements);
```

Listing 5.14 Set the kernel arguments

10. Enqueue the kernel for execution

OpenCL always executes kernels in parallel, i.e., instances of the same kernel execute on different data set. Each kernel execution in OpenCL is called a work-item. Each work-item is responsible for executing the kernel once and operating on its assigned portion of the data set. OpenCL exploits parallel computation of the compute devices by having instances of the kernel execute on different portions of the N-dimensional problem space. In addition, each work-item is executed only with its assigned data. Thus, it is programmer's responsibility to tell OpenCL how many work-items are needed to process all data.

OpenCL: Enqueue the kernel for execution on a device

To enqueue a command to execute a kernel on a device use the function `clEnqueueNDRangeKernel`:

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

This function enqueues a command into `command_queue` to execute the kernel `kernel` on a device over NDRange. The argument `work_dim` denotes the number of dimensions used to specify the global work-items and work-items in the work-group. The number of global work-items in `work_dim` dimensions that will execute the kernel function is specified with `global_work_size`.The size of the work-group that will execute the kernel is specified with `local_work_size`. Refer to OpenCL™ 2.2 Specification for more detailed description.

Before the work-items total can be determined, the N-dimension to be used to represent the data must be determined. For example, a linear array of data would be a one-dimension problem space, while an image would be a two-dimensional problem space, and spatial data, such as a 3D object, would be a three-dimensional problem space.

When the dimension space is determined, the total work-items (also called the global work size) and group size can be calculated. When the work-items for each dimension and the group size (local work size) is determined (i.e., NDRange), the kernel can be sent to the command queue for execution. To execute the kernel function, we must enqueue the kernel object into a command queue. To enqueue the kernel to execute on a device, we use the function `clEnqueueNDRangeKernel`. Listing 5.15 shows how to enqueue the `VectorAdd` kernel on a device over one-dimensional space. The total number of work-items is `szGlobalWorkSize`, and the work-group size is `szLlobalWorkSize`.In this example (vector addition), `szGlobalWorkSize` is set to be equal to the number of elements in the input vector(s), while `szLocalWorkSize` is set to 256. Thus, the kernel `VectorAdd` will be executed by `szGlobalWorkSize` work-items and each SM on a GPU device will execute 256 work-items.

```
1    //*************************************************
2    // Start Core sequence... copy input data to GPU, compute,
3    //      copy results back
```

```
    //*************************************************
    // STEP 10: Enqueue the kernel for execution
    //*************************************************
    // Launch kernel
    ciErr = clEnqueueNDRangeKernel(
                                    cmdQueue,
                                    ckKernel,
                                    1,
                                    NULL,
                                    &szGlobalWorkSize,
                                    &szLocalWorkSize,
                                    0,
                                    NULL,
                                    NULL);
    if (ciErr != CL_SUCCESS)
    {
        printf("Error launchung kernel!\n" );
    }

    // Wait for the command commands to get serviced before
    //  reading back results
    //
    clFinish(cmdQueue);
```

Listing 5.15 Enqueue the kernel for execution

11. Read the output buffer back to the host
After the kernel function has been executed on the device, we should read the output data from the device. To read data from a device buffer to host memory, we use the `clEnqueueReadBuffer` function. Listing 5.16 shows how to read data from the device buffer `bufferC` to host memory `srcC`.

```
    //*************************************************
    // STEP 11: Read the output buffer back to the host
    //*************************************************

    // Synchronous/blocking read of results
    ciErr = clEnqueueReadBuffer(
                                  cmdQueue,
                                  bufferC,
                                  CL_TRUE,
                                  0,
                                  datasize,
                                  srcC,
                                  0,
                                  NULL,
                                  NULL);
```

Listing 5.16 Read the output buffer back to the host

5.3.2 Sum of Arbitrary Long Vectors

The OpenCL standard does not specify how the abstract execution model provided by OpenCL is mapped to the hardware. We can enqueue any number of threads (work items), and provide a work-group size (number of work_items in a work-group), with at least the following constraints:

OpenCL: Read from a buffer

To enqueue commands to read from a buffer object to the host memory use the `clEnqueueReadBuffer` function:

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

This function enqueues commands into command queue `command_queue` to read `cb` bytes from a device buffer `buffer` to host memory pointed by `ptr`. This function does not block by default. To know when the command has completed, we can use a blocking form of the command by setting the `blocking_write` parameter to `CL_TRUE`. Refer to OpenCL™ 2.2 Specification for more detailed description.

Occupancy

Occupancy is a ratio of active warps per compute unit to the maximum number of allowed warps. We should always keep the occupancy high because this is a way to hide latency when executing instructions. A compute unit should have a warp ready to execute in every cycle as this is the only way to keep hardware busy.

1. work-group size must divide the number of work items,
2. work-group size be at most `CL_DEVICE_MAX_WORK_GROUP_SIZE` (recall that for the CPU device used in the previous example this is 1200).

Maximum number of work-groups per compute unit is limited by the hardware resources. Each compute unit has a limited number of registers and a limited amount of local memory. Usually, no more than 16 work-groups can run simultaneously on a single compute unit with the Kepler microarchitecture and 8 work-groups can run simultaneously on a single compute unit with the Fermi microarchitecture. Also, there is a limit for the number of active warps on a single compute unit (64 on Kepler, 48 on Fermi). We usually want to keep as many active warps as possible because this affects *occupancy*. Occupancy is a ratio of active warps per compute unit to the maximum number of allowed warps. Keeping the occupancy high, we can hide latency when executing instructions. Recall that executing other warps when one warp is paused is the only way to hide latency and keep hardware busy. Finally, the

hardware also limits the number of work-groups in a single launch (usually this is 65535 in each NDRange dimension).

In our previous example, we have vectors with 512×512 elements (`iNumElements`) and we have launched the same number of work-items (`szGlobalWorkSize`). As each work-group contains 512 work-items (`szLocalWorkSize`), we have 512 work-groups in a single launch. Is it possible to add larger vectors and where is the limit? If we tried to add two vectors with 512×512 elements, we would fail to launch a kernel with such a large number of work-items (work-groups). So how would we use a GPU to add two arbitrary long vectors? First, we should limit the number of work-items and the number of work-groups. Secondly, one work-item should perform more than one addition. Let us first look at the new kernel function in Listing 5.17.

```
// OpenCL Kernel Function for element by element
//  vector addition of arbitrary long vectors
__kernel void VectorAddArbitrary(
                        __global float* a,
                        __global float* b,
                        __global float* c,
                        int iNumElements
                        ) {

    //find my global index
    int iGID = get_global_id(0);

    while (iGID < iNumElements) {
        // add adjacent elements
        c[iGID] = a[iGID] + b[iGID];
        iGID += get_global_size(0);
    }
}
```

Listing 5.17 Sum o arbitrary long vectors - the kernel function

We used a `while` loop to iterate through the data (this kernel is very similar to the function from Listing 5.2). Rather than incrementing `iGID` by 1, a many core GPU device could increment `iGID` by the number of work-items that we are using. We want each work-item to start on a different data index, so we use the thread global index:

```
int iGID = get_global_id(0);
```

After each thread finishes its work at the current index, we increment `iGID` by the total number of work-items in NDRange. This number is obtained from the function `get_global_size(0)`:

```
iGID += get_global_size(0);
```

The only remaining piece is to fix the execution model in the host code. To ensure that we never launch too many work-groups and work-items, we will fix the number of work-groups to a small number, but still large enough to have a good occupancy. We will launch 512 work-groups with 256 work-items per work-group (thus the total number of work-items will be 131072). The only change in the host code is

```
szLocalWorkSize = 256;
szGlobalWorkSize = 512*256;
```

5.3.3 Dot Product in OpenCL

We will now take a look at vector dot products. We will start with the simple version first to illustrate basic features of memory and work-item management in OpenCL programs. We will again recap the usage of NDRange and work-item ID. We will then analyze performance of the simple version and extend the simple version to version which employs local memory.

The computation of a vector dot product consists of two steps. First, we multiply corresponding elements of the two input vectors. This is very similar to vector addition but utilizes multiplication instead of addition. In the second step, we sum all the products instead of just storing them to an output vector. Each working-item multiplies a pair of corresponding vector elements and then moves on to its next pair. Because the result would be the sum of all these pairwise products, each working-item keeps a sum of its products. Just like in the addition example, the working-items increment their indices by the total number of threads. The kernel function for the first step is shown in Listing 5.18.

```
1  // OpenCL Kernel Function for Naive Dot Product
2  __kernel void DotProductNaive(
3                                  __global float* a,
4                                  __global float* b,
5                                  __global float* c,
6                                  int iNumElements
7                                  ) {
8
9      //find my global index
10     int iGID = get_global_id(0);
11     int index = iGID;
12
13     while (iGID < iNumElements) {
14         // add adjacent elements
15         c[iGID] = a[index] * b[index];
16         index += get_global_size(0);
17     }
18 }
```

Listing 5.18 Vector Dot Product Kernel - naive implementation

Each element of the array `c` holds the sum of products obtained form one work-item, i.e., `c[iGID]` holds a sum of products obtained by the work-item with the global index `iGID`. After all work-item finish their work, we should sum all the elements form the vector `c` to produce a single scalar product. But how do we know when have all work-items finished their work? We need a mechanism to synchronize work-items. The only way to synchronize all work-items in NDRange is to wait for the kernel function to finish. After the kernel function finishes, we can read the results (vector `c`) from a GPU device and sum its elements on host. The host code is

very similar to the host code from Listing 5.4. We have to make the following two changes:

1. the vector `c` has a different size than vectors `a` and `b`. It has the same number of elements as the number of all work-items in NDRange (`szGlobalWorkSize`):

```
//*****************************************************
// STEP 6: Create device buffers
//*****************************************************

...

cl_mem bufferC; // Output array on the device

// Size of data for bufferC:
size_t datasize_c = sizeof(cl_float) * szGlobalWorkSize;

...

// Use clCreateBuffer() to create a buffer object (d_C)
// with enough space to hold the output data
bufferC = clCreateBuffer(
                          context,
                          CL_MEM_READ_WRITE,
                          datasize_c,
                          NULL,
                          &status);
```

Listing 5.19 Create a buffer object C for naive implementation of vector dot product

2. after the kernel executes on a GPU device, we read vector `c` from device and serially sum all its elements to produce the final dot product:

How Fast is Your OpenCL Kernel

Our motivation for writing kernels in OpenCL is to speed up applications. Often, we want to measure the execution time of a kernel. As OpenCL is a performance-oriented language, performance analysis is an essential part of OpenCL programming. The OpenCL runtime provides a built-in mechanism (**profiling**) for timing the execution of kernels. A profiler is a performance analysis tool that gathers data from the OpenCL run-time using events. This information is used to discover bottlenecks in the application and find ways to optimize the application's performance. OpenCL supports 64-bit timing of commands submitted to command queues and events to keep track of a command's status. Events can be used with most commands placed on the command queue: commands to read, write, map, or copy memory objects, commands to enqueue kernels, etc. Profiling is enabled when a queue is created with the `CL_QUEUE_PROFILING_ENABLE` flag is set. The fact is that when you execute your kernels on GPU, no CPU clock is spent during the execution. When profiling is enabled, the function `clGetEventProfilingInfo` is used to extract the timing data. We need to follow next steps to measure the execution time of OpenCL kernel execution time:

OpenCL: Timing the execution

To return profiling information for the command associated with event if profiling is enabled use the function `clGetEventProfilingInfo`.

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

The function returns profiling information for the command associated with event if profiling is enabled. The first argument is the event being queried, and the second argument, `param_name` is an enumeration value describing the query. Most often used `param_name` values are:

CL_PROFILING_COMMAND_START : A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event starts execution on the device, and

CL_PROFILING_COMMAND_END : A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event has finished execution on the device.

Event objects are used to capture profiling information that measure execution time of a command. Profiling of OpenCL commands can be enabled using a command-queue created with CL_QUEUE_PROFILING_ENABLE flag set in properties argument to `clCreateCommandQueue`. OpenCL devices are required to correctly track time across changes in device frequency and power states. Refer to OpenCL™ 2.2 Specification for more detailed description.

1. Create a queue, profiling need to be enabled when the queue is created.

```
cmdQueue = clCreateCommandQueue(
              ...
              CL_QUEUE_PROFILING_ENABLE,
              &status);
```

2. Link an event when launching a kernel:

```
cl_event kernelevent;
ciErr = clEnqueueNDRangeKernel(
          ...
          &kernelevent);
```

3. Wait for the kernel to finish:

```
ciErr = clWaitForEvents (1, &kernelevent); // Wait for the event
```

4. Get profiling data using the function `clGetEventProfilingInfo()` and calculate the kernel execution time.
5. Release the event using the function `clReleaseEvent()`.

The code snippet form Listing 5.20 shows how to measure kernel execution time using OpenCL profiling events.

```
...

//*****************************************************
// STEP 3: Create a command queue
//*****************************************************

cl_command_queue cmdQueue;
// Create a command queue using clCreateCommandQueue(),
// and associate it with the device you want to execute
// on. Enable profiling.
cmdQueue = clCreateCommandQueue(
                                context,
                                devices[1], // GPU
                                CL_QUEUE_PROFILING_ENABLE,
                                &status);


...


//*****************************************************
// Start Core sequence... copy input data to GPU, compute,
//      copy results back
cl_event kernelevent;
//*****************************************************
// STEP 10: Enqueue the kernel for execution
//*****************************************************
// Launch kernel
ciErr = clEnqueueNDRangeKernel(
                                cmdQueue,
                                ckKernel,
                                1,
                                NULL,
                                &szGlobalWorkSize,
                                &szLocalWorkSize,
                                0,
                                NULL,
                                &kernelevent);

if (ciErr != CL_SUCCESS)
{
    printf("Error launchung kernel!\n" );
}
ciErr = clWaitForEvents (1, &kernelevent); // Wait for the ↩
    event

// Obtain the start- and end time for the event
unsigned long start = 0;
unsigned long end = 0;

// read device time counter in nanoseconds when the command
// identified by event starts execution on the device:
clGetEventProfilingInfo(kernelevent,
                        CL_PROFILING_COMMAND_START,
                        sizeof(cl_ulong),
```

```
                          &start,
                          NULL);
clGetEventProfilingInfo(kernelevent,
                          CL_PROFILING_COMMAND_END,
                          sizeof(cl_ulong),
                           &end,
                           NULL);

// Compute the duration in nanoseconds
float duration = (end - start) * 10e-9;

// Don't forget to release the event
clReleaseEvent(kernelevent);

printf("Kernel execution time = %f s \n", duration);

// Wait for the command commands to get serviced before
//  reading back results
//
clFinish(cmdQueue);
```

Listing 5.20 Measuring kernel execution time

This way, we can profile operations on both memory objects and kernels. Results for dot product of two vectors of size 16777216 ($512 \times 512 \times 64$) on an Apple laptop with an Intel GPU are as follows:

```
Kernel execution time = 0.127389 s
Result = 33554432.000000
```

5.3.4 Dot Product in OpenCL Using Local Memory

Host device data transfer has much lower bandwidth than global memory access. So we should perform as much computation on a GPU device as possible and read as small amount of data from a GPU device as possible. In this case, the threads should cooperate to calculate the final sum. Work-items can safely cooperate through local memory by means of synchronization. Local memory can be shared by all work-items in a work-group. Local memory on a GPU is implemented on a compute device. To allocate local memory, the `__local` address space qualifier is used in variable declaration. We will use a buffer in local memory named `ProductsWG` to store each work-item's running sum. This buffer will store `szLocalWorkSize` products so each work-item in the work-group will have a place to store its temporary result. Since the compiler will create a copy of the local variables for each work-group, we need to allocate only enough memory such that each thread in the work-group has an entry. It is relatively simple to declare local memory buffers as we just pass local arrays as arguments to the kernel:

```
__kernel void DotProductShared(
                __global float* a,
                __global float* b,
                __global float* c,
                __local* ProductsWG,
                int iNumElements)
```

OpenCL: Barrier

```
barrier(mem_fence_flag)
```

All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier. The `mem_fence_flag` can be either `CLK_LOCAL_MEM_FENCE` (the barrier function will queue a memory fence to ensure correct ordering of memory operations to local memory), or `CLK_GLOBAL_MEM_FENCE` (the barrier function will queue a memory fence to ensure correct ordering of memory operations to global memory. This can be useful when work-items, for example, write to buffer objects and then want to read the updated data). Refer to OpenCL™ 2.2 Specification for more detailed description.

We then set the kernel argument with a value of `NULL` and a size equal to the size we want to allocate for the argument (in byte). Therefore, it should be as follows:

```
ciErr |= clSetKernelArg(ckKernel,
                        3,
                        sizeof(float) * szLocalWorkSize,
                        NULL);
```

Now, each work-item computes a running sum of the product of corresponding entries in `a` and `b`. After reaching the end of the array, each thread stores its temporary sum into the local memory (buffer `ProductsWG`):

```
// work-item global index
int iGID = get_global_id(0);
// work-item local index
int iLID = get_local_id(0);
float temp = 0.0;
while (iGID < iNumElements) {
   // multiply adjacent elements
   temp += a[iGID] * b[iGID];
   iGID += get_global_size(0);
}
//store the product in local memory
ProductsWG[iLID] = temp;
```

At this point, we need to sum all the temporary values we have placed in the `ProductsWG`. To do this, we will need some of the threads to read the values that have been stored there. This is a potentially dangerous operation. We should place a synchronization barrier to guarantee that all of these writes to the local buffer `ProductsWG` complete before anyone tries to read from this buffer. The OpenCL C language provides functions to allow synchronization of work-items. However, as we mentioned, the synchronization can only occur between work-items in the same work-group. To achieve that, OpenCL implements a barrier memory fence for synchronization with the `barrier()` function. The function `barrier()` creates a barrier that blocks the current work-item until all other work-items in the same group has executed the barrier before allowing the work-item to proceed beyond the

barrier. All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier. The following call guarantees that every work-item in the work-group has completed instructions before the hardware will execute the next instruction on any work-item within the work-group

```
barrier(CLK_LOCAL_MEM_FENCE);
```

Now that we have guaranteed that our local memory has been filled, we can sum the values in it. We call the general process of taking an input array and performing some computations that produce a smaller array of results a *reduction*. The naive way to accomplish this reduction would be having one thread iterate over the shared memory and calculate a running sum. This will take us time proportional to the length of the array. However, since we have hundreds of threads available to do our work, we can do this reduction in parallel and take time that is proportional to the logarithm of the length of the array. Figure 5.12 shows a summation reduction. The idea is that each work-item adds two of the values in `ProductsWG` and store the result back to `ProductsWG`. Since each thread combines two entries into one, we complete the first step with half as many entries as we started with. In the next step, we do the same thing for the remaining half. We continue until we have the sum of every entry in the first element of `ProductsWG`. The code for the summation reduction is

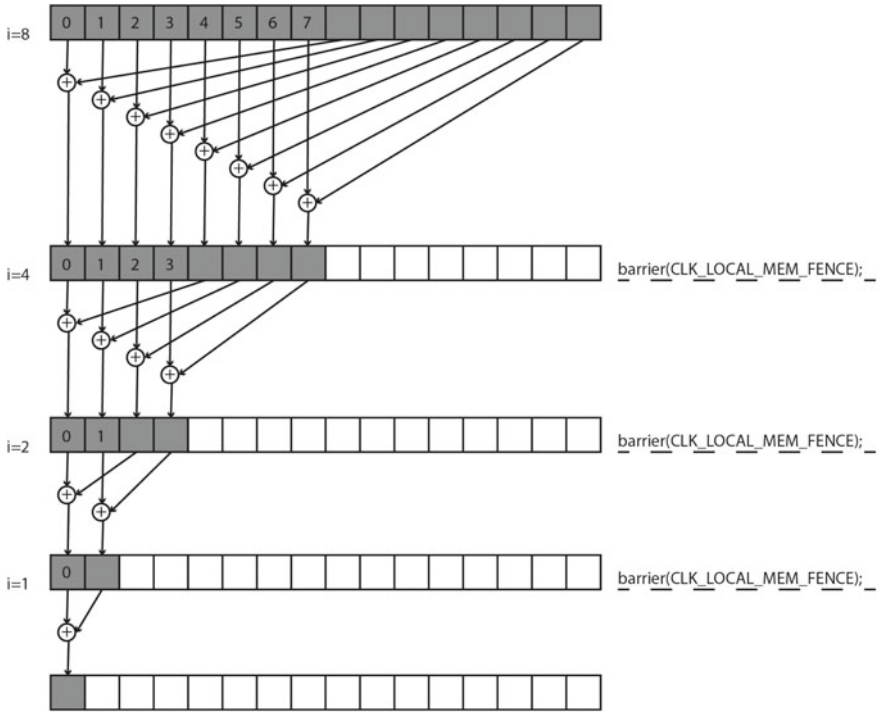


Fig. 5.12 Summation reduction

Reduction

In computer science, the reduction is a special type of operation that is commonly used in parallel programming to reduce the elements of an array into a single result.

```
// how many work-items are in WG?
int iWGS = get_local_size(0);
// Summation reduction:
int i = iWGS/2;
while(i!=0){
   if (iLID < i) {
      ProductsWG[iLID] += ProductsWG[iLID+i];
   }
   barrier(CLK_LOCAL_MEM_FENCE);
   i=i/2;
}
```

After we have completed one step, we have the same restriction we did after computing all the pairwise products. Before we can read the values we just stored in `ProductsWG`, we need to ensure that every thread that needs to write to `ProductsWG` has already done so. The `barrier(CLK_LOCAL_MEM_FENCE)` after the assignment ensures this condition is met. It is important to note that when using barrier, all work-items in the work-group must execute the barrier function. If the barrier function is called within a conditional statement, it is important to ensure that all work-items in the work-group enter the conditional statement to execute the barrier. For example, the following code is an illegal use of barrier because the barrier will not be encountered by all work-items:

```
if (iLID < i) {
   ProductsWG[iLID] += ProductsWG[iLID+i];
   barrier(CLK_LOCAL_MEM_FENCE);
}
```

Any work-item with the local index `iLID` greater than or equal to `i` will never execute the `barrier(CLK_LOCAL_MEM_FENCE)`. Because of the guarantee that no instruction after a `barrier(CLK_LOCAL_MEM_FENCE)` can be executed before every work-item of the work-group has executed it, the hardware simply continues to wait for these work-items. This effectively hangs the processor because it results in the GPU waiting for something that will never happen. Such a kernel will actually cause the GPU to stop responding, forcing you to kill your program.

After termination of the summation reduction, each work-group has a single number remaining. This number is sitting in the first entry of the `ProductsWG` buffer and is the sum of every pairwise product the work-items in that work-group computed. We now store this single value to global memory and end our kernel:

```
if (iLID == 0) {
   c[iWGID] = ProductsWG[0];
}
```

As there is only one element from `ProductsWG` that is transferred to global memory, only a single thread needs to perform this operation. Since each work-group writes exactly one value to the global array `c`, we can simply index it by `WGID`, which is the work-group index.

We are left with an array `c`, each entry of which contains the sum produced by one of the parallel work-groups. The last step of the dot product is to sum the entries of `c`. Because array `c` is relatively small, we return control to the host and let the CPU finish the final step of the addition, summing the array `c`.

Listing 5.21 shows the entire kernel function for dot product using shared memory and summation reduction.

```
//************************************************
// OpenCL Kernel Function for dot product
// using shared memory snd summation reduction
__kernel void DotProductShared(__global float* a,
                               __global float* b,
                               __global float* c,
                               __local* ProductsWG,
                               int iNumElements)
{

    // work-item global index
    int iGID = get_global_id(0);
    // work-item local index
    int iLID = get_local_id(0);
    // work-group index
    int iWGID = get_group_id(0);
    // how many work-items are in WG?
    int iWGS = get_local_size(0);

    float temp = 0.0;
    while (iGID < iNumElements) {
        // multiply adjacent elements
        temp += a[iGID] * b[iGID];
        iGID += get_global_size(0);
    }
    //store the product
    ProductsWG[iLID] = temp;

    // wait for all threads in WG:
    barrier(CLK_LOCAL_MEM_FENCE);

    // Summation reduction:
    int i = iWGS/2;
    while(i!=0){
        if (iLID < i) {
            ProductsWG[iLID] += ProductsWG[iLID+i];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        i=i/2;
    }

    // store partial dot product into global memory:
    if (iLID == 0) {
        c[iWGID] = ProductsWG[0];
    }
}
```

Listing 5.21 Vector Dot Product Kernel - implementation using local memory and summation reduction

In the host code for this example,we should create the `bufferC` memory object that will hold `szGlobalWorkSize/szLocalWorkSize` partial dot products. Listing 5.22 shows how to create `bufferC`.

```
size_t datasize_c = sizeof(cl_float) * (szGlobalWorkSize/↩
    szLocalWorkSize);

// Use clCreateBuffer() to create a buffer object (d_C)
// with enough space to hold the output data
bufferC = clCreateBuffer(
                          context,
                          CL_MEM_READ_WRITE,
                          datasize_c,
                          NULL,
                          &status);
```

Listing 5.22 Create `bufferC` for dot product using local memory

Listing 5.23 shows how to set kernel arguments.

```
//*****************************************************
// STEP 9: Set the kernel arguments
//*****************************************************
// Set the Argument values
ciErr = clSetKernelArg(ckKernel,
                       0,
                       sizeof(cl_mem),
                       (void*)&bufferA);
ciErr |= clSetKernelArg(ckKernel,
                        1,
                        sizeof(cl_mem),
                        (void*)&bufferB);
ciErr |= clSetKernelArg(ckKernel,
                        2,
                        sizeof(cl_mem),
                        (void*)&bufferC);
ciErr |= clSetKernelArg(ckKernel,
                        3,
                        sizeof(float) * szLocalWorkSize,
                        NULL);
ciErr |= clSetKernelArg(ckKernel,
                        4,
                        sizeof(cl_int),
                        (void*)&iNumElements);
```

Listing 5.23 Set kernel arguments for dot product using shared memory

The argument with index 3 is used to create local memory buffer of size `sizeof (float) * szLocalWorkSize` for each work-group. As this argument is declared in the kernel function with the `__local` qualifier, the last entry to `clSetKernelArg` must be NULL.

Results for dot product of two vectors of size 67108864 ($512 \times 512 \times 256$) on an Apple laptop with an Intel GPU are

```
Kernel execution time = 0.503470 s
Result = 33554432.000000
```

5.3.5 Naive Matrix Multiplication in OpenCL

This section describes a matrix multiplication application using OpenCL for GPUs in a step-by-step approach. We will start with the most basic version (naive) where focus will be on the code structure for the host application and the OpenCL GPU kernels. The naive implementation is rather straightforward, but it gives us a nice starting point for further optimization. For simplicity of presentation, we will consider only square matrices whose dimensions are integral multiples of 32 on a side. Matrix multiplication is a key building block for dense linear algebra and the same pattern of computation is used in many other algorithms. We will start with simple version first to illustrate basic features of memory and work-item management in OpenCL programs. After that we will extend to version which employs local memory.

Before starting, it is helpful to briefly recap how a matrix–matrix multiplication is computed. The element $c_{i,j}$ of **C** is the dot product of the ith row of **A** and the jth column of **B**. The matrix multiplication of two square matrices is illustrated in Fig. 5.13. For example, as illustrated in Fig. 5.13, the element $c_{5,2}$ is the dot product of the row 5 of **A** and the column 2 of **B**.

To implement matrix multiplication of two square matrices of dimension $N \times N$, we will launch $N \times N$ work-items. Indexing of work-items in NDRange will correspond to 2D indexing of the matrices. Work-item (i, j) will calculate the element $c_{i,j}$ using row i of **A** and column j of **B**. So, each work-item loads one row of matrix **A** and one column of matrix **B** from global memory, do the dot product, and store the result back to matrix **C** in the global memory. The matrix **A** is, therefore, read N times from global memory and **B** is read N times from global memory. The simple version of matrix multiplication can be implemented in the plain C language using three nested loops as in Listing 5.24. We assume data to be stored in row-major order (C-style).

```
void matrixmul(float *matrixA,
               float *matrixB,
               float *matrixC,
               int N) {

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
               matrixC[i*N + j] +=
                  matrixA[i * N + k] * matrixB[k * N + j];
            }
        }
    }
}
```

Listing 5.24 Simple matrix multiplication in C

Let us now implement the simple matrix multiplication in OpenCL.

The Naive Multiplication Kernel

To implement the above matrix multiplication in OpenCL $N \times N$ work-items will be needed. Let us have each work-item compute an element of the matrix **C**. Each work-item should first discover its ID within 2D NDRange and compute the corresponding element of **C**. Listing 5.25 shows the kernel function for naive matrix multiplication.

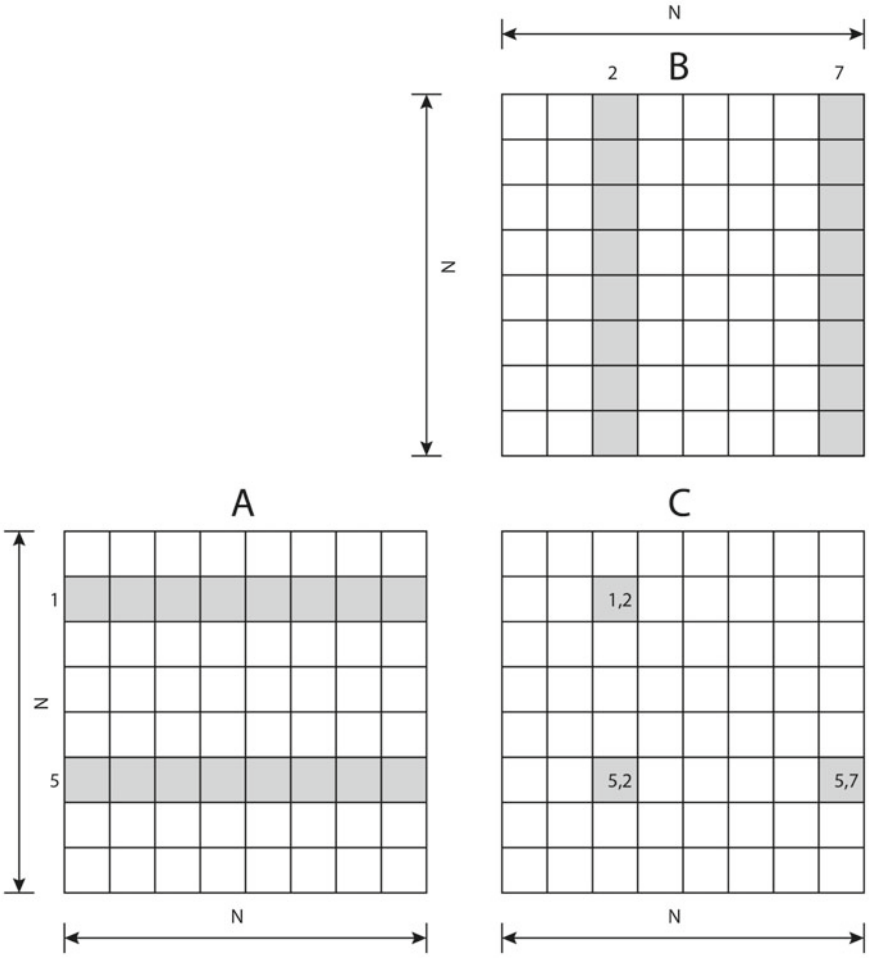


Fig. 5.13 Matrix multiplication

```
// OpenCL Kernel Function for naive matrix multiplication
__kernel void matrixmulNaive(
                              __global float* matrixA,
                              __global float* matrixB,
                              __global float* matrixC,
                              int N) {

    // global thread index
    int xGID = get_global_id(0); // column in NDRange
    int yGID = get_global_id(1); // row in NDRange

    float dotprod = 0.0;

    // each work item calculates one element of the matrix C:
    for (int i = 0; i < N; i++) {
```

```
        dotprod += matrixA[yGID * N + i] * matrixB[i * N + xGID];
    }
    matrixC[yGID * N + xGID] = dotprod;
}
```

Listing 5.25 The naive multiplication kernel

Each work item first discovers its global ID in 2D NDRange. The index of the column `xGID` is obtained from the first dimension of NDRange using `get_global_id(0)`. Similarly, the index of the row `yGID` is obtained from the second dimension of NDRange using `get_global_id(1)`. After obtaining its global ID, each work-item do the dot product between the `yGID`-th row of **A** and the `xGID`-th column of **B**. The dot product is stored to the element in the `yGID`-th row and the `xGID`-th column of **C**.

The Host Code

As we learned previously, the host code should probe for devices, create context, create buffers, and compile the OpenCL program containing kernels. These steps are the same as in the vector addition example from Sect. 5.3.1. Assuming you have already initialized OpenCL, created the context and the queue, and created the appropriate buffers and memory copies. Listing 5.26 shows how to compile the kernel for naive matrix multiplication.

```
//*****************************************************
// STEP 8: Create and compile the kernel
//*****************************************************
ckKernel = clCreateKernel(
                          cpProgram,
                          "matrixmulNaive",
                          &ciErr);
if (!ckKernel || ciErr != CL_SUCCESS)
{
    printf("Error: Failed to create compute kernel!\n");
    exit(1);
}
```

Listing 5.26 Create and compile the kernel for naive matrix multiplication

Prior to launch the kernel we should set the kernel arguments as in Listing 5.27.

```
//*****************************************************
// STEP 9: Set the kernel arguments
//*****************************************************
ciErr = clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&bufferA);
ciErr |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&bufferB);
ciErr |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&bufferC);
ciErr |= clSetKernelArg(ckKernel, 3, sizeof(cl_int), (void*)&iRows);
```

Listing 5.27 Set the kernel arguments for naive matrix multiplication

Finally, we are ready to launch the kernel `matrixmulNaive`. Listing 5.28 shows how you launch the kernel.

```
// ******************************************************
// Start Core sequence... copy input data to GPU, compute,
//      copy results back

// set and log Global and Local work size dimensions
const cl_int iWI = 16;
const size_t szLocalWorkSize[2] = { iWI, iWI };
const size_t szGlobalWorkSize[2] = { iRows, iRows };
cl_event kernelevent;

//*************************************************
// STEP 10: Enqueue the kernel for execution
//*************************************************
ciErr = clEnqueueNDRangeKernel(
                                cmdQueue,
                                ckKernel,
                                2,
                                NULL,
                                szGlobalWorkSize,
                                szLocalWorkSize,
                                0,
                                NULL,
                                &kernelevent);
```

Listing 5.28 Enqueue the kernel for naive matrix multiplication

As can be seen from the code in Listing 5.28, NDRange is of 2D size (`iRows`, `iRows`). That means that we launch (`iRows`, `iRows`) work-items. These work-items are further grouped into work-groups of dimension (16,16). If for example the size of matrices (`iRows`, `iRows`) is (4096, 4096), we launch 256×256 work-groups. As the number of work-groups is probably larger than the number of compute-units present in a GPU, we keep all compute-units busy. Recall that we should always keep the occupancy high, because this is a way to hide latency when executing instructions.

Execution time for naive matrix multiplication of two square matrices of size 3584×3584 on an Apple laptop with an Intel GPU is

```
Kernel execution time = 30.154823 s
```

5.3.6 Tiled Matrix Multiplication in OpenCL

Looking at the loop in the kernel code from Listing 5.25, we can notice that each work-item loads $2 \times N$ elements from global memory—two for each iteration through the loop, one from the matrix **A** and one from the matrix **B**. Since accesses to global memory are relatively slow, this can slow down the kernel, leaving the work-items idle for hundreds of clock cycles, for each access. Also, we can notice that for each element of **C** in a row, we use the same row of **A** and that each work-item in a work-group uses the same columns of **B**.

But not only are we accessing the GPU's off-chip memory way too much, we do not even care about memory coalescing! Assuming row-major order when storing matrices in global memory, the elements from the matrix **A** are accessed with unit stride, while elements from the matrix **B** are accessed with stride N.

Recall from Sect. 5.1.4 that to ensure memory coalescing, we want work-items from the same warp to access contiguous elements in memory so to minimize the number of required memory transactions. As work-items of the same warp access 32 contiguous floating-point elements from the same row of **A**, all these elements fall into the same 128-bytes segment and data is delivered in a single transaction. On the other hand, work-items of the same warp access 32 floating-point elements from **B** that are 4N bytes apart, so for each element from the matrix **B** a new memory transaction is needed. Although the GPU's caches probably will help us out a bit, we can get much more performance by *manually caching sub-blocks of the matrices (tiles)* in the GPU's on-chip local memory.

In other words, one way to reduce the number of accesses to global memory is to have the work-items load portions of matrices **A** and **B** into local memory, where we can access them much more quickly. So we will use local memory to avoid non-coalesced global memory access. Ideally, we would load both matrices entirely into local memory, but unfortunately, local memory is a rather limited resource and cannot hold two large matrices. Recall that older devices have 16kB of local memory per compute unit, and more recent devices have 48 kB of local memory per compute unit. So we will content ourselves with loading portions of **A** and **B** into local memory as needed, and making as much use of them as possible while they are there.

Assume that we multiply two matrices as shown in Fig. 5.14. To calculate the elements of the square submatrix **C** (*tile* C), we should multiply the corresponding rows and columns of matrices **A** and **B**. Also, we can subdivide the matrices **A** and **B** into submatrices (*tiles*) such as shown in Fig. 5.14. Now, we can multiply the corresponding row and the column from the **A** and **B** tiles and sum up these partial products. The process is shown in the lower part of Fig. 5.14. We can also observe that the individual rows and columns in tiles **A** and **B** are accessed several times. For example, in the 3×3 tiles from Fig. 5.14, all elements on the same row of the tile **C** are computed using the same data of the **A** tiles and all elements on the same column of the submatrix **C** are computed using the same data of the **B** tile. As the tiles are in local memory, these accesses are fast.

The idea of using tiles in matrix multiplication is as follows. The number of work-items that we start is equal to the number of elements in the matrix. Each work-item will be responsible for computing one element of the product matrix **C**. The index of the element in the matrix is equal to the global index of a work-item in NDrange. At the same time, we create the same number of work-groups as is the number of tiles. The number of elements in a tile will be equal to the number of threads in a work-group. This means that the element index within a tile will be the same as the local index in the group.

For reference, consider the matrix multiplication in Fig. 5.15. All matrices are of size 8×8, so we will have 64 work-items in NDrange. We divide matrices **A**, **B** and **C** in non-overlapping sub-blocks (tiles) of size $TW \times TW$, where $TW = 4$ as in

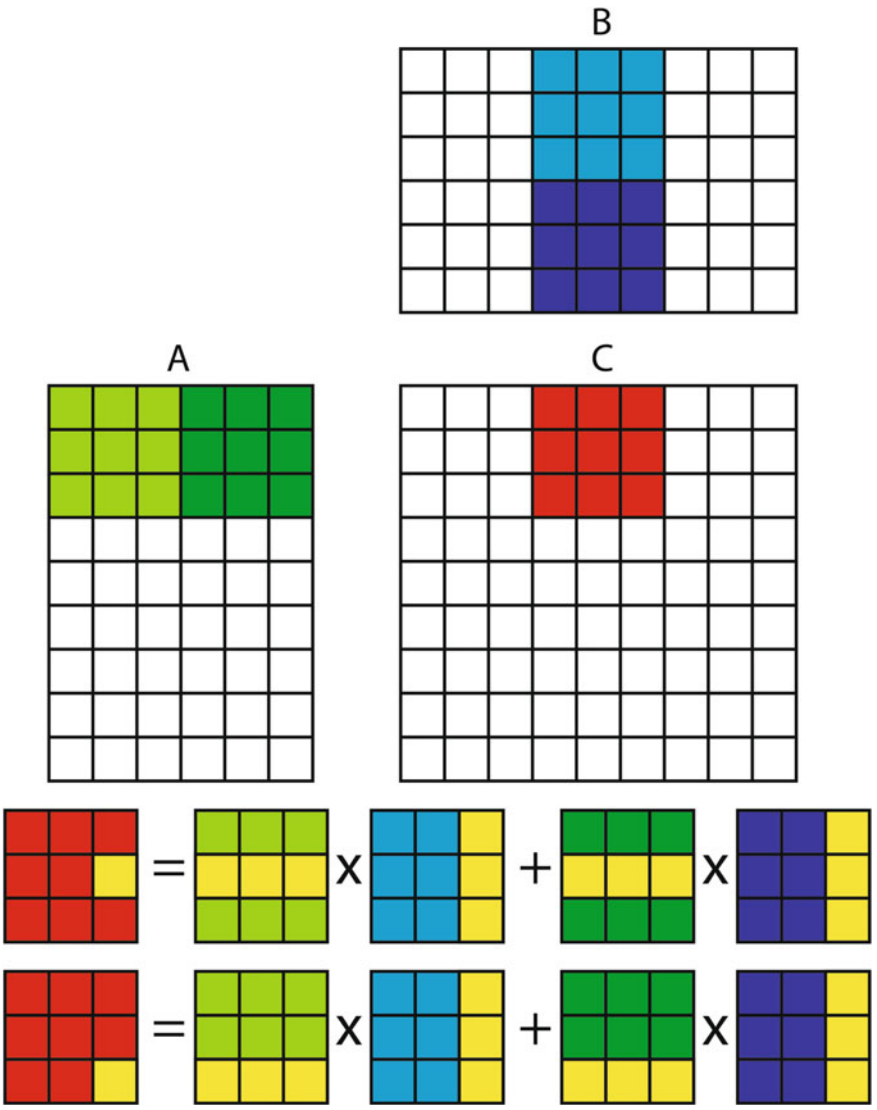


Fig. 5.14 Matrix multiplication using tiles

Fig. 5.15. Let us also suppose that tiles are indexed starting in the upper left corner. Now, consider the element $c_{5,2}$ in the matrix **C**, in Fig. 5.15. The element $c_{5,2}$ falls into the tile (0,1). The work-item responsible for computing the element $c_{5,2}$ has the global row index 5 and the global column index 2. Also, the same work-item has the local row index 1 and local column index 2. This work-item computes the element $c_{5,2}$ in **C** by multiplying together row 5 in **A**, and column 2 in **B**, but it will do it in pieces using tiles. As we already said, all work-items responsible for computing

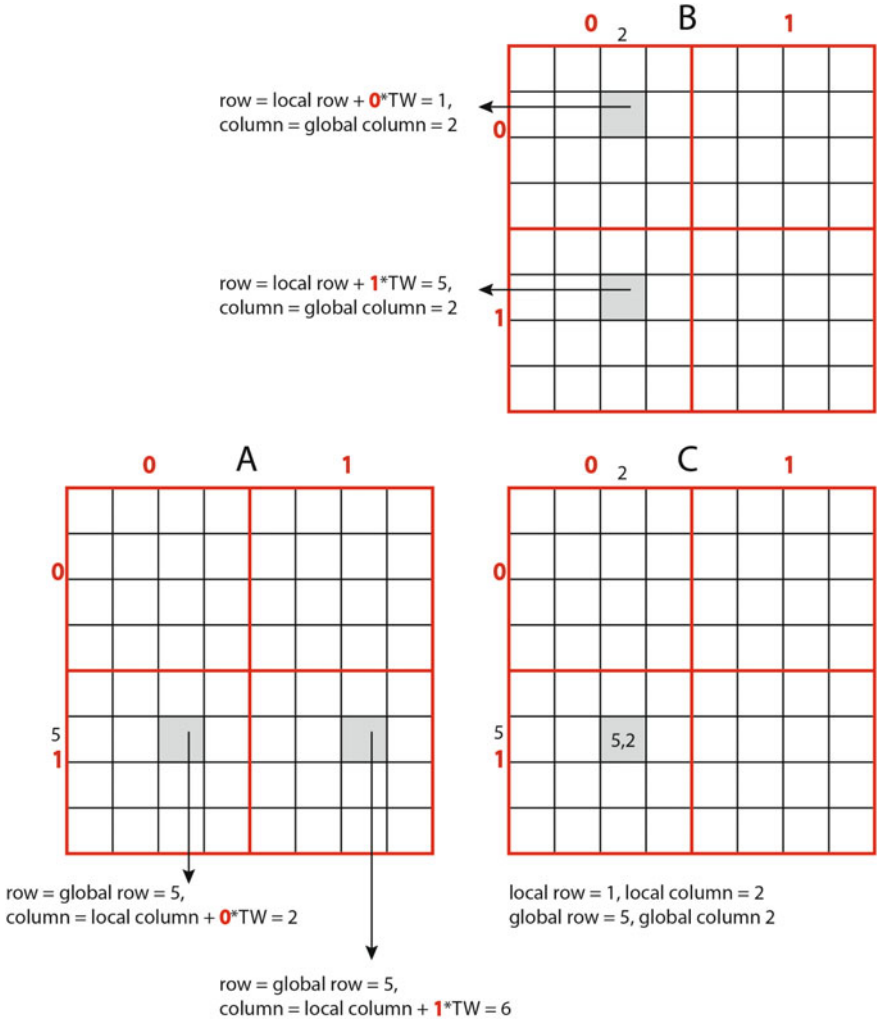


Fig. 5.15 Matrix multiplication with tiles

elements of the same tile in the matrix **C** should be in the same work-group. Let us explain this process for the work-item that computes the element $c_{5,2}$. The work-item should access the tiles (0,1) and (1,1) from **A** and tiles (0,0) and (1,1) from **B**. The computation is performed in two steps. First, the work-item computes dot product between the row 1 from the tile (0,1) in **A** and the column 2 from the tile (0,0) in **B**. In the second step, the same work-item computes dot product between the the row 1 from the tile (1,1) in **A** and the column 2 from the tile (1,0) in **B**. Finally, it adds this dot product to the one computed in the first step.

If we want to compute the first dot product as fast as possible, the elements form the row 1 from the tile (0,1) in **A** and the elements from the column 2 from the tile (0,0) in **B** should be in the local memory. The same is true for all rows in the tile (0,1) in **A** and all columns in the tile (0,0) in **B** because all work-items from the same work-group will access these elements concurrently. Also, once the first step is finished, the same will be true for the second step but this time for the tile (1,1) in **A** and the tile (1,0) in **B**.

So, before every step, all work-items from the same work-group should perform a *collaborative load* of tiles A and B into local memory. This is performed in such a way that the work-item in the ith local row and the jth local column performs two loads from global memory per tile: the element with local index (i, j) from the corresponding tile in matrix **A** and the element with local index (i, j) from the corresponding tile in matrix **B**. Figure 5.15 illustrates this process. For example, the work item that computes the element $c_{5,2}$ reads:

1. the elements $a_{5,2}$ and $b_{1,2}$ in the first step, and
2. the elements $a_{5,6}$ and $b_{5,2}$ in the second step.

Where is the benefit of using tiles? If we load the left-most (0,1) tile of matrix **A** into local memory, and the top-most (0,0) of those tiles of matrix **B** into local memory, then we can compute the first $TW \times TW$ products and add them together just by reading from local memory. But here is the benefit: as long as we have those tiles in local memory, every work-item from the work-group computing a tile form **C** can compute that portion of their sum from the same data in local memory. When each work item has computed this sum, we can load the next $TW \times TW$ tiles from **A** and **B**, and continue adding the term-by-term products to our value in **C**. And after all of the tiles have been processed, we will have computed our entries in **C**.

The Tiled Multiplication Kernel

The kernel code for the tiled matrix multiplication is shown in Listing 5.29.

```
#define TILE_WIDTH 16

// OpenCL Kernel Function for tiled matrix multiplication
__kernel void matrixmulTiled(
                              __global float* matrixA,
                              __global float* matrixB,
                              __global float* matrixC,
                              int N) {


    // Local memory to fit the tiles
    __local float matrixAsub[TILE_WIDTH][TILE_WIDTH];
    __local float matrixBsub[TILE_WIDTH][TILE_WIDTH];

    // global thread index
    int xGID = get_global_id(0); // column in NDRange
    int yGID = get_global_id(1); // row in NDRange

    // local thread index
    int xLID = get_local_id(0); // column in tile
    int yLID = get_local_id(1); // row in tile
```

```
22
23      float dotprod = 0.0;
24
25      for(int tile = 0; tile < N/TILE_WIDTH; tile++){
26          // Collaborative loading of tiles into shared memory:
27          // Load a tile of matrixA into local memory
28          matrixAsub[yLID][xLID] =
29              matrixA[yGID * N + (xLID + tile*TILE_WIDTH)];
30          // Load a tile of matrixB into local memory
31          matrixBsub[yLID][xLID] =
32              matrixB[(yLID + tile*TILE_WIDTH) * N  + xGID];
33
34          // Synchronise to make sure the tiles are loaded
35          barrier(CLK_LOCAL_MEM_FENCE);
36
37          for (int i = 0; i < TILE_WIDTH; i++) {
38              dotprod +=
39                  matrixAsub[yLID][i] * matrixBsub[i][xLID];
40          }
41
42          // Wait for other work-items to finish
43          //      before loading next tile
44          barrier(CLK_LOCAL_MEM_FENCE);
45      }
46
47      matrixC[yGID * N + xGID] = dotprod;
48  }
```

Listing 5.29 The tiled multiplication kernel

Tiles are stored in `matrixAsub` and `matrixBsub`. Each work-item finds its global index and its local index. The outer loop goes through all the tiles necessary to calculate the products in **C**. Each work-item in the work-group in one iteration of the outer loop first reads its elements from the global memory and writes them to the tile element with its local index. After loading its elements, each work-item waits at the barrier until the tiles are loaded. Then, in the innermost loop, each work-item calculates dot product between a row `yLID` form the tile `matrixAsub` and the column `xLID` from the tile `matrixBsub`. After that the work-item waits again at the barrier for the other work-items to finish their dot products. Then all work-items load next tiles and repeat the process.

The Host Code

To implement tiling, we will leave our host code from the previous naive kernel intact. The only thing we should change is to create and compile the appropriate kernel function:

```
1  //*****************************************************
2  // STEP 8: Create and compile the kernel
3  //*****************************************************
4  ckKernel = clCreateKernel(
5                              cpProgram,
6                              "matrixmulTiled",
7                              &ciErr);
8  if (!ckKernel || ciErr != CL_SUCCESS)
9  {
10     printf("Error: Failed to create compute kernel!\n");
11     exit(1);
12 }
```

Listing 5.30 Create and compile the kernel for tiled matrix multiplication

Note that it already uses 2D work-groups of 16 by 16. This means that the tiles are also 16 by 16.

Execution time for tiled matrix multiplication of two square matrices of size 3584×3584 on an Apple laptop with an Intel GPU is

```
Kernel execution time = 16.409384 s
```

5.4 Exercises

1. To verify that you understand how to control the argument definitions for a kernel, modify the kernel in Listing 5.3 so it adds four vectors together. Modify the host code to define four vectors and associate them with relevant kernel arguments. Read back the final result and verify that it is correct.
2. Use local memory to minimize memory movement costs and optimize performance of the matrix multiplication kernel in Listing 5.25. Modify the kernel so that each work-item copies its own row of **A** into local memory. Report kernel execution time.
3. Modify the kernel from the previous exercise so that each work-group collaboratively copies its own column of **B** into local memory. Report kernel execution time.
4. Write an OpenCL program that computes the Mandelbrot set. Start with the program in Listing 3.21.
5. Write an OpenCL program that computes π. Start with the program in Listing 3.15. Hint: the parallelization is similar to the parallelization of a dot product.
6. Write an OpenCL program that transposes a matrix. Use local memory and collaboratively reads to minimize memory movement costs and optimize performance of the kernel.
7. Given an input array $\{a_0, a_1, \ldots, a_{n-1}\}$ in pointer `d_a`, write an OpenCL program that stores the reversed array $\{a_{n-1}, a_{n-2}, \ldots, a_0\}$ in pointer `d_b`. Use multiple blocks. Try to revert data in local memory. Hint: using work-groups and local memory revert data in array slices. Then, revert slices in global memory.
8. Write an OpenCL program to detect edges on black and with images using the Sobel filter.

5.5 Bibliographical Notes

The primary source of information including all details of OpenCL is available at Khronos web site [15] where the complete reference guide is available. Another good online source of OpenCL tutorials and dozen of examples is *HandsOnOpenCL*

training course [14]. A comprehensive hands-on presentation of OpenCL can be found in the book *OpenCL in Action* by Matthew Scarpino [24]. A gentle introduction to OpenCL by the same author can be found in [23]. The books by Munshi et al. [17] and Gaster et al. [11] provide a deep dive into OpenCL.