CHAPTER 1

Introduction

We have undeniably entered the age of accelerated computing. In order to satisfy the world's insatiable appetite for more computation, accelerated computing drives complex simulations, AI, and much more by providing greater performance *and* improved power efficiency when compared with earlier solutions.

Heralded as a "New Golden Age for Computer Architecture,"[1] we are faced with enormous opportunity through a rich diversity in compute devices. We need portable software development capabilities that are not tied to any single vendor or architecture in order to realize the full potential for accelerated computing.

SYCL (pronounced *sickle*) is an industry-driven Khronos Group standard adding advanced support for data parallelism with C++ to support accelerated (heterogeneous) systems. SYCL provides mechanisms for C++ compilers to exploit accelerated (heterogeneous) systems in a way that is highly synergistic with modern C++ and C++ build systems. SYCL is not an acronym; SYCL is simply a name.

[1] *A New Golden Age for Computer Architecture* by John L. Hennessy, David A. Patterson; Communications of the ACM, February 2019, Vol. 62 No. 2, Pages 48-60.

ACCELERATED VS. HETEROGENEOUS

These terms go together. *Heterogeneous* is a technical description acknowledging the combination of compute devices that are programmed differently. *Accelerated* is the motivation for adding this complexity to systems and programming. There is no guarantee of acceleration ever; programming heterogeneous systems will only accelerate our applications when we do it right. This book helps teach us how to do it right!

Data parallelism in C++ with SYCL provides access to all the compute devices in a modern accelerated (heterogeneous) system. A single C++ application can use any combination of devices—including GPUs, CPUs, FPGAs, and application-specific integrated circuits (ASICs)—that are suitable to the problems at hand. No proprietary, single-vendor, solution can offer us the same level of flexibility.

This book teaches us how to harness accelerated computing using data-parallel programming using C++ with SYCL and provides practical advice for balancing application performance, portability across compute devices, and our own productivity as programmers. This chapter lays the foundation by covering core concepts, including terminology, which are critical to have fresh in our minds as we learn how to accelerate C++ programs using data parallelism.

Read the Book, Not the Spec

No one wants to be told "Go read the spec!"—specifications are hard to read, and the SYCL specification (`www.khronos.org/sycl/`) is no different. Like every great language specification, it is full of precision but is light on motivation, usage, and teaching. This book is a "study guide" to teach C++ with SYCL.

No book can explain *everything at once*. Therefore, this chapter does what no other chapter will do: the code examples contain programming constructs that go unexplained until future chapters. We should not get hung up on understanding the coding examples completely in Chapter 1 and trust it will get better with each chapter.

SYCL 2020 and DPC++

This book teaches C++ with SYCL 2020. The first edition of this book preceded the SYCL 2020 specification, so this edition includes updates including adjustments in the header file location (`sycl` instead of `CL`), device selector syntax, and removal of an explicit host device.

DPC++ is an open source compiler project based on LLVM. It is our hope that SYCL eventually be supported by default in the LLVM community and that the DPC++ project will help make that happen. The DPC++ compiler offers broad heterogeneous support that includes GPU, CPU, and FPGA. All examples in this book work with the DPC++ compiler and should work with any C++ compiler supporting SYCL 2020.

Important resources for updated SYCL information, including any known book errata, include the book GitHub (`github.com/Apress/data-parallel-CPP`), the Khronos Group SYCL standards website (`www.khronos.org/sycl`), and a key SYCL education website (`sycl.tech`).

As of publication time, no C++ compiler claims full conformance or compliance with the SYCL 2020 specification. Nevertheless, the code in this book works with the DPC++ compiler and should work with other C++ compilers that have most of SYCL 2020 implemented. We use only standard C++ with SYCL 2020 excepting for a few DPC++-specific extensions that are clearly called out in Chapter 17 (Programming for FPGAs) to a small degree, Chapter 20 (Backend Interoperability) when connecting to Level Zero backends, and the Epilogue when speculating on the future.

Why Not CUDA?

Unlike CUDA, SYCL supports data parallelism in C++ for all vendors and all types of architectures (not just GPUs). CUDA is focused on NVIDIA GPU support only, and efforts (such as HIP/ROCm) to reuse it for GPUs by other vendors have limited ability to succeed despite some solid success and usefulness. With the explosion of accelerator architectures, only SYCL offers the support we need for harnessing this diversity and offering a multivendor/multiarchitecture approach to help with portability that CUDA does not offer. To more deeply understand this motivation, we highly recommend reading (or watching the video recording of their excellent talk) "A New Golden Age for Computer Architecture" by industry legends John L. Hennessy and David A. Patterson. We consider this a must-read article.

Chapter 21, in addition to addressing topics useful for migrating code from CUDA to C++ with SYCL, is valuable for those experienced with CUDA to bridge some terminology and capability differences. The most significant capabilities beyond CUDA come from the ability for SYCL to support multiple vendors, multiple architectures (not just GPUs), and multiple backends even for the same device. This flexibility answers the question "Why not CUDA?"

SYCL does not involve any extra overhead compared with CUDA or HIP. It is not a compatibility layer—it is a generalized approach that is open to all devices regardless of vendor and architecture while simultaneously being in sync with modern C++. Like other open multivendor and multiarchitecture techniques, such as OpenMP and OpenCL, the ultimate proof is in the implementations including options to access hardware-specific optimizations when absolutely needed.

Why Standard C++ with SYCL?

As we will point out repeatedly, every program using SYCL is first and foremost a C++ program. SYCL does not rely on any language changes to C++. SYCL does take C++ programming places it cannot go without SYCL. We have no doubt that all programming for accelerated computing will continue to influence language standards including C++, but we do not believe the C++ standard should (or will) evolve to displace the need for SYCL any time soon. SYCL has a rich set of capabilities that we spend this book covering that extend C++ through classes and rich support for new compiler capabilities necessary to meet needs (already existing today) for multivendor and multiarchitecture support.

Getting a C++ Compiler with SYCL Support

All examples in this book compile and work with all the various distributions of the DPC++ compiler and should compile with other C++ compilers supporting SYCL (see "SYCL Compilers in Development" at www.khronos.org/sycl). We are careful to note the very few places where extensions are used that are DPC++ specific at the time of publication.

The authors recommend the DPC++ compiler for a variety of reasons, including our close association with the DPC++ compiler. DPC++ is an open source compiler project to support SYCL. By using LLVM, the DPC++ compiler project has access to backends for numerous devices. This has already resulted in support for Intel, NVIDIA, and AMD GPUs, numerous CPUs, and Intel FPGAs. The ability to extend and enhance support openly for multiple vendors and multiple architecture makes LLVM a great choice for open source efforts to support SYCL.

There are distributions of the DPC++ compiler, augmented with additional tools and libraries, available as part of a larger project to offer broad support for heterogeneous systems, which include libraries, debuggers, and other tools, known as the oneAPI project. The oneAPI tools, including the DPC++ compiler, are freely available (`www.oneapi.io/implementations`).

```
 1. #include <iostream>
 2. #include <sycl/sycl.hpp>
 3. using namespace sycl;
 4.
 5. const std::string secret{
 6.     "Ifmmp-!xpsme\"\012J(n!tpssz-!Ebwf/!"
 7.     "J(n!bgsbje!J!dbo(u!ep!uibu/!.!IBM\01"};
 8.
 9. const auto sz = secret.size();
10.
11. int main() {
12.   queue q;
13.
14.   char* result = malloc_shared<char>(sz, q);
15.   std::memcpy(result, secret.data(), sz);
16.
17.   q.parallel_for(sz, [=](auto& i) {
18.      result[i] -= 1;
19.    }).wait();
20.
21.   std::cout << result << "\n";
22.   free(result, q);
23.   return 0;
24. }
```

EXAMPLE
IS KNOWN TO HAVE ELEMENTS NOT TAUGHT UNTIL FUTURE CHAPTERS. EXAMINE AT YOUR OWN RISK OF CONFUSION.

Figure 1-1. *Hello data-parallel programming*

Hello, World! and a SYCL Program Dissection

Figure 1-1 shows a sample SYCL program. Compiling and running it results in the following being printed:

`Hello, world!` (and some additional text left to experience by running it)

We will completely understand this example by the end of Chapter 4. Until then, we can observe the single include of `<sycl/sycl.hpp>` (line 2) that is needed to define all the SYCL constructs. All SYCL constructs live inside a namespace called `sycl`.

- Line 3 lets us avoid writing `sycl::` over and over.
- Line 12 instantiates a queue for work requests directed to a particular device (Chapter 2).
- Line 14 creates an allocation for data shared with the device (Chapter 3).
- Line 15 copies the secret string into device memory, where it will be processed by the kernel.
- Line 17 enqueues work to the device (Chapter 4).
- Line 18 is the only line of code that will run on the device. All other code runs on the host (CPU).

Line 18 is the *kernel* code that we want to run on devices. That kernel code decrements a single character. With the power of `parallel_for()`, that kernel is run on each character in our secret string in order to decode it into the `result` string. There is no ordering of the work required, and it is run asynchronously relative to the main program once the `parallel_for` queues the work. It is critical that there is a `wait` (line 19) before looking at the result to be sure that the kernel has completed, since in this example we are using a convenient feature (Unified Shared Memory, Chapter 6). Without the wait, the output may occur before all the characters have been decrypted. There is more to discuss, but that is the job of later chapters.

Queues and Actions

Chapter 2 discusses queues and actions, but we can start with a simple explanation for now. Queues are the only connection that allows an application to direct work to be done on a device. There are two types of actions that can be placed into a queue: (a) code to execute and (b) memory operations. Code to execute is expressed via either `single_task` or `parallel_for` (used in Figure 1-1). Memory operations perform copy operations between host and device or fill operations to initialize memory. We only need to use memory operations if we seek more control than what is done automatically for us. These are all discussed later in the book starting with Chapter 2. For now, we should be aware that queues are the connection that allows us to command a device, and we have a set of actions available to put in queues to execute code and to move around data. It is also very important to understand that requested actions are placed into a queue without waiting. The host, after submitting an action into a queue, continues to execute the program, while the device will eventually, and asynchronously, perform the action requested via the queue.

QUEUES CONNECT US TO DEVICES

We submit actions into queues to request computational work and data movement.

Actions happen asynchronously.

It Is All About Parallelism

Since programming in C++ for data parallelism is all about parallelism, let's start with this critical concept. The goal of parallel programming is to compute something faster. It turns out there are two aspects to this: *increased throughput* and *reduced latency*.

Throughput

Increasing throughput of a program comes when we get more work done in a set amount of time. Techniques like pipelining may stretch out the time necessary to get a single work-item done, to allow overlapping of work that leads to more work-per-unit-of-time being done. Humans encounter this often when working together. The very act of sharing work involves overhead to coordinate that often slows the time to do a single item. However, the power of multiple people leads to more throughput. Computers are no different—spreading work to more processing cores adds overhead to each unit of work that likely results in some delays, but the goal is to get more total work done because we have more processing cores working together.

Latency

What if we want to get one thing done faster—for instance, analyzing a voice command and formulating a response? If we only cared about throughput, the response time might grow to be unbearable. The concept of latency reduction requires that we break up an item of work into pieces that can be tackled in parallel. For throughput, image processing might assign whole images to different processing units—in this case, our goal may be optimizing for images per second. For latency, image processing might assign each pixel within an image to different processing cores—in this case, our goal may be maximizing pixels per second from a single image.

Think Parallel

Successful parallel programmers use both techniques in their programming. This is the beginning of our quest to *Think Parallel*.

We want to adjust our minds to think first about where parallelism can be found in our algorithms and applications. We also think about how different ways of expressing the parallelism affect the performance we ultimately achieve. That is a *lot* to take in all at once. The quest to *Think Parallel* becomes a lifelong journey for parallel programmers. We can learn a few tips here.

Amdahl and Gustafson

Amdahl's Law, stated by the supercomputer pioneer Gene Amdahl in 1967, is a formula to predict the theoretical maximum speed-up when using multiple processors. Amdahl lamented that the maximum gain from parallelism is limited to `(1/(1-p))` where `p` is the fraction of the program that runs in parallel. If we only run two-thirds of our program in parallel, then the most that program can speed up is a factor of 3. We definitely need that concept to sink in deeply! This happens because no matter how fast we make that two-thirds of our program run, the other one-third still takes the same time to complete. Even if we add 100 GPUs, we will only get a factor of 3 increase in performance.

For many years, some viewed this as proof that parallel computing would not prove fruitful. In 1988, John Gustafson wrote an article titled "Reevaluating Amdahl's Law." He observed that parallelism was not used to speed up fixed workloads, but it was used to allow work to be scaled up. Humans experience the same thing. One delivery person cannot deliver a single package faster with the help of many more people and trucks. However, a hundred people and trucks can deliver one hundred packages more quickly than a single driver with a truck. Multiple drivers will definitely increase throughput and will also generally reduce latency for package deliveries. Amdahl's Law tells us that a single driver cannot deliver one package faster by adding ninety-nine more drivers with their own trucks. Gustafson noticed the opportunity to deliver one hundred packages faster with these extra drivers and trucks.

This emphasizes that parallelism is most useful because the size of problems we tackle keep growing in size year after year. Parallelism would not nearly as important to study if year after year we only wanted to run the same size problems faster. This quest to solve larger and larger problems fuels our interest in exploiting data parallelism, using C++ with SYCL, for the future of computer (heterogeneous/accelerated systems).

Scaling

The word “scaling” appeared in our prior discussion. Scaling is a measure of how much a program speeds up (simply referred to as “speed-up”) when additional computing is available. Perfect speed-up happens if one hundred packages are delivered in the same time as one package, by simply having one hundred trucks with drivers instead of a single truck and driver. Of course, it does not reliably work that way. At some point, there is a bottleneck that limits speed-up. There may not be one hundred places for trucks to dock at the distribution center. In a computer program, bottlenecks often involve moving data around to where it will be processed. Distributing to one hundred trucks is similar to having to distribute data to one hundred processing cores. The act of distributing is not instantaneous. Chapter 3 starts our journey of exploring how to distribute data to where it is needed in a heterogeneous system. It is critical that we know that data distribution has a cost, and that cost affects how much scaling we can expect from our applications.

Heterogeneous Systems

For our purposes, a heterogeneous system is any system which contains multiple types of computational devices. For instance, a system with both a central processing unit (CPU) and a graphics processing unit (GPU) is a heterogeneous system. The CPU is often just called a processor, although that can be confusing when we speak of all the processing units in a heterogeneous system as compute processors. To avoid this confusion, SYCL refers to processing units as *devices*. An application always runs on a *host* that in turn sends work to *devices*. Chapter 2 begins the discussion of how our main application (*host* code) will steer work (computations) to particular *devices* in a heterogeneous system.

A program using C++ with SYCL runs on a *host* and issues kernels of work to *devices*. Although it might seem confusing, it is important to know that the host will often be able to serve as a device. This is valuable for two key reasons: (1) the host is most often a CPU that will run a kernel if no accelerator is present—a key promise of SYCL for application portability is that a kernel can always be run on any system even those without accelerators—and (2) CPUs often have vector, matrix, tensor, and/or AI processing capabilities that are accelerators that kernels map well to run upon.

Host code invokes code on *devices*. The capabilities of the *host* are very often available as a *device* also, to provide both a back-up device and to offer any acceleration capabilities the host has for processing kernels as well. Our *host* is most often a CPU, and as such it may be available as a *CPU device*. There is no guarantee by SYCL of a *CPU device*, only that there is at least one device available to be the default device for our application.

While heterogeneous describes the system from a technical standpoint, the reason to complicate our hardware and software is to obtain higher performance. Therefore, the term *accelerated computing* is popular for marketing heterogeneous systems or their components. We like to emphasize that there is no guarantee of acceleration. Programming of heterogeneous systems will only accelerate our applications when we do it right. This book helps teach us how to do it right!

GPUs have evolved to become high-performance computing (HPC) devices and therefore are sometimes referred to as general-purpose GPUs, or GPGPUs. For heterogeneous programming purposes, we can simply assume we are programming such powerful GPGPUs and refer to them as GPUs.

Today, the collection of devices in a heterogeneous system can include CPUs, GPUs, FPGAs (field-programmable gate arrays), DSPs (digital signal processors), ASICs (application-specific integrated circuits), and AI chips (graph, neuromorphic, etc.).

The design of such devices will involve duplication of compute processors (multiprocessors) and increased connections (increased bandwidth) to data sources such as memory. The first of these, multiprocessing, is particularly useful for raising throughput. In our analogy, this was done by adding additional drivers and trucks. The latter of these, higher bandwidth for data, is particularly useful for reducing latency. In our analogy, this was done with more loading docks to enable trucks to be fully loaded in parallel.

Having multiple types of devices, each with different architectures and therefore different characteristics, leads to different programming and optimization needs for each device. That becomes the motivation for C++ with SYCL and the majority of what this book has to teach.

SYCL was created to address the challenges of C++ data-parallel programming for heterogeneous (accelerated) systems.

Data-Parallel Programming

The phrase "data-parallel programming" has been lingering unexplained ever since the title of this book. Data-parallel programming focuses on parallelism that can be envisioned as a bunch of data to operate on in parallel. This shift in focus is like Gustafson vs. Amdahl. We need one hundred packages to deliver (effectively lots of data) in order to divide up the work among one hundred trucks with drivers. The key concept comes down to what we should divide. Should we process whole images or process them in smaller tiles or process them pixel by pixel? Should we analyze a collection of objects as a single collection or a set of smaller groupings of objects or object by object?

Choosing the right division of work and mapping that work onto computational resources effectively is the responsibility of any parallel programmer using C++ with SYCL. Chapter 4 starts this discussion, and it continues through the rest of the book.

Key Attributes of C++ with SYCL

Every program using SYCL is first and foremost a C++ program. SYCL does not rely on any language changes to C++.

C++ compilers with SYCL support will optimize code based on built-in knowledge of the SYCL specification as well as implement support so heterogeneous compilations "just work" within traditional C++ build systems.

Next, we will explain the key attributes of C++ with SYCL: *single-source* style, host, devices, kernel code, and asynchronous task graphs.

Single-Source

Programs are single-source, meaning that the same translation unit[2] contains both the code that defines the compute kernels to be executed on devices and also the host code that orchestrates execution of those compute kernels. Chapter 2 begins with a more detailed look at this capability. We can still divide our program source into different files and translation units for host and device code if we want to, but the key is that we don't have to!

[2] We could just say "file," but that is not entirely correct here. A translation unit is the actual input to the compiler, made from the source file after it has been processed by the C preprocessor to inline header files and expand macros.

Host

Every program starts by running on a host, and most of the *lines* of code in a program are usually for the host. Thus far, hosts have always been CPUs. The standard does not require this, so we carefully describe it as a host. This seems unlikely to be anything other than a CPU because the host needs to fully support C++17 in order to support all C++ with SYCL programs. As we will see shortly, devices (accelerators) do not need to support all of C++17.

Devices

Using multiple devices in a program is what makes it heterogeneous programming. That is why the word *device* has been recurring in this chapter since the explanation of heterogeneous systems a few pages ago. We already learned that the collection of devices in a heterogeneous system can include GPUs, FPGAs, DSPs, ASICs, CPUs, and AI chips, but is not limited to any fixed list.

Devices are the targets to gain acceleration. The idea of offloading computations is to transfer work to a device that can accelerate completion of the work. We have to worry about making up for time lost moving data—a topic that needs to constantly be on our minds.

Sharing Devices

On a system with a device, such as a GPU, we can envision two or more programs running and wanting to use a single device. They do not need to be programs using SYCL. Programs can experience delays in processing by the device if another program is currently using it. This is really the same philosophy used in C++ programs in general for CPUs. Any system can be overloaded if we run too many active programs on our CPU (mail, browser, virus scanning, video editing, photo editing, etc.) all at once.

On supercomputers, when nodes (CPUs + all attached devices) are granted exclusively to a single application, sharing is not usually a concern. On non-supercomputer systems, we can just note that the performance of a program may be impacted if there are multiple applications using the same devices at the same time.

Everything still works, and there is no programming we need to do differently.

Kernel Code

Code for a device is specified as kernels. This is a concept that is not unique to C++ with SYCL: it is a core concept in other offload acceleration languages including OpenCL and CUDA. While it is distinct from loop-oriented approaches (such as commonly used with OpenMP target offloads), it may resemble the body of code within the innermost loop without requiring the programmer to write the loop nest explicitly.

Kernel code has certain restrictions to allow broader device support and massive parallelism. The list of features *not* supported in kernel code includes dynamic polymorphism, dynamic memory allocations (therefore no object management using new or delete operators), static variables, function pointers, runtime type information (RTTI), and exception handling. No virtual member functions, and no variadic functions, are allowed to be called from kernel code. Recursion is not allowed within kernel code.

VIRTUAL FUNCTIONS?

While we will not discuss it further in this book, the DPC++ compiler project does have an experimental extension (visible in the open source project, of course) to implement some support for virtual functions within kernels. Thanks to the nature of offloading to accelerator efficiently, virtual functions cannot be supported well without some restrictions, but many users have expressed interest in seeing SYCL offer such support even with some restrictions. The beauty of open source, and the open SYCL specification, is the opportunity to participate in experiments that can inform the future of C++ and SYCL specifications. Visit the DPC++ project (github.com/intel/llvm) for more information.

Chapter 3 describes how memory allocations are done before and after kernels are invoked, thereby making sure that kernels stay focused on massively parallel computations. Chapter 5 describes handling of exceptions that arise in connection with devices.

The rest of C++ is fair game in a kernel, including functors, lambda expressions, operator overloading, templates, classes, and static polymorphism. We can also share data with the host (see Chapter 3) and share the read-only values of (non-global) host variables (via lambda expression captures).

Kernel: Vector Addition (DAXPY)

Kernels should feel familiar to any programmer who has worked on computationally complex code. Consider implementing DAXPY, which stands for "double-precision A times X Plus Y." A classic for decades. Figure 1-2 shows DAXPY implemented in modern Fortran, C/C++, and SYCL. Amazingly, the computation lines (line 3) are virtually identical. Chapters 4 and 10 explain kernels in detail. Figure 1-2 should help remove any concerns that kernels are difficult to understand—they should feel familiar even if the terminology is new to us.

```
! Fortran loop
do i = 1, n
  z(i) = alpha * x(i) + y(i)
end do
```

```
// C/C++ loop
for (int i=0;i<n;i++) {
  z[i] = alpha * x[i] + y[i];
}
```

```
// SYCL kernel
q.parallel_for(range{n},[=](id<1> i) {
  z[i] = alpha * x[i] + y[i];
}).wait();
```

EXAMPLE

IS KNOWN TO HAVE ELEMENTS NOT TAUGHT IN THIS BOOK AT ALL. PLEASE ENJOY THIS FORTRAN, SADLY THERE IS NO MORE IN THIS BOOK.

Figure 1-2. *DAXPY computations in Fortran, C/C++, and SYCL*

Asynchronous Execution

The asynchronous nature of programming using C++ with SYCL must *not* be missed. Asynchronous programming is critical to understand for two reasons: (1) proper use gives us better performance (better scaling), and (2) mistakes lead to parallel programming errors (usually race conditions) that make our applications unreliable.

The asynchronous nature comes about because work is transferred to devices via a "queue" of requested actions. The host program submits a requested action into a queue, and the program continues without waiting for any results. This *no waiting* is important so that we can try to keep computational resources (devices and the host) busy all the time. If we had to wait, that would tie up the host instead of allowing the host to do useful work. It would also create serial bottlenecks when the device finished, until we queued up new work. Amdahl's Law, as discussed earlier, penalizes us for time spent not doing work in parallel. We need to construct our programs to be moving data to and from devices while the devices are busy and keep all the computational power of the devices and host busy any time work is available. Failure to do so will bring the full curse of Amdahl's Law upon us.

Chapter 3 starts the discussion on thinking of our program as an asynchronous task graph, and Chapter 8 greatly expands upon this concept.

Race Conditions When We Make a Mistake

In our first code example (Figure 1-1), we specifically did a "`wait`" on line 19 to prevent line 21 from writing out the value from `result` before it was available. We must keep this asynchronous behavior in mind. There is another subtle thing done in that same code example—line 15 uses `std::memcpy` to load the input. Since `std::memcpy` runs on the host, line 17 and later do not execute until line 15 has completed. After reading Chapter 3, we could be tempted to change this to use `q.memcpy` (using SYCL). We have done exactly that in Figure 1-3 on line 7. Since that is a queue submission, there is no guarantee that it will execute before line 9. This creates a *race condition*, which is a type of parallel programming bug. A race condition exists when two parts of a program access the same data without coordination. Since we expect to write data using line 7 and then read it in line 9, we do not want a race that might have line 9 execute before line 7 completes! Such a race condition would make our program unpredictable—our program could get different results on different runs and on different systems. A fix for this would be to explicitly wait for `q.memcpy` to complete before proceeding by adding `.wait()` to the end of line 7. That is not the best fix. We could have used event dependences to solve this (Chapter 8). Creating the queue as an ordered queue would also add an implicit dependence between the `memcpy` and the `parallel_for`. As an alternative, in Chapter 7, we will see how a buffer and accessor programming style can be used to have SYCL manage the dependences and waits automatically for us.

```
// ...we are changing one line from Figure 1-1
char* result = malloc_shared<char>(sz, q);

// Introduce potential data race!  We don't define a
// dependence to ensure correct ordering with later
// operations.
q.memcpy(result, secret.data(), sz);

q.parallel_for(sz, [=](auto& i) {
   result[i] -= 1;
 }).wait();

// ...
```

EXAMPLE
IS KNOWN TO HAVE ELEMENTS NOT TAUGHT UNTIL FUTURE CHAPTERS. EXAMINE AT YOUR OWN RISK OF CONFUSION.

Figure 1-3. *Adding a race condition to illustrate a point about being asynchronous*

RACE CONDITIONS DO NOT ALWAYS CAUSE A PROGRAM TO FAIL

An astute reader noticed that the code in Figure 1-3 did not fail on every system they tried. Using a GPU with `partition_max_sub_devices==0` did not fail because it was a small GPU not capable of running the `parallel_for` until the `memcpy` had completed. Regardless, the code is flawed because the race condition exists even if it does not universally cause a failure at runtime. We call it a race—sometimes we win, and sometimes we lose. Such coding flaws can lay dormant until the right combination of compile and runtime environments lead to an observable failure.

Adding a `wait()` forces host synchronization between the `memcpy` and the kernel, which goes against the previous advice to keep the device busy all the time. Much of this book covers the different options and trade-offs that balance program simplicity with efficient use of our systems.

OUT-OF-ORDER QUEUES VS. IN-ORDER QUEUES

We will use out-of-order queues in this book because of their potential performance benefits, but it is important to know that support for in-order queues does exist. In-order is simply an attribute we can request when creating a queue. CUDA programmers will know that CUDA streams are unconditionally *in-order*. SYCL queues instead are *out-of-order* by default but may optionally be in-order by passing the `in_order` queue property when the SYCL queue is created (refer to Chapter 8). Chapter 21 provides information on this and other considerations for programmers coming from using CUDA.

For assistance with detecting data race conditions in a program, including kernels, tools such as Intel Inspector (available with the oneAPI tools mentioned previously in "Getting a DPC++ Compiler") can be helpful. The sophisticated methods used by such tools often do not work on all devices. Detecting race conditions may be best done by having all the kernels run on a CPU, which can be done as a debugging technique during development work. This debugging tip is discussed as Method#2 in Chapter 2.

TO TEACH THE CONCEPT OF *DEADLOCK*, THE DINING PHILOSOPHERS PROBLEM IS A CLASSIC ILLUSTRATION OF A SYNCHRONIZATION PROBLEM IN COMPUTER SCIENCE

Imagine a group of philosophers sitting around a circular table, with a single chopstick placed between each philosopher. Every philosopher needs two chopsticks to eat their meal, and they always pick up chopsticks one at a time. Regrettably, if all philosophers first grab the chopstick to their left and then hold it waiting for the chopstick from their right, we have a problem if they all get hungry at the same time. Specifically, they will end up all waiting for a chopstick that will never be available.

Poor algorithm design (grab left, then wait until grab right) in this case can result in deadlock and all the philosophers starving to death. That is sad. Discussing the numerous ways to design an algorithm that starves fewer philosophers to death, or hopefully is fair and feeds them all (none starve), is a topic that is fun to consider and has been written about many times.

Realizing how easy it is to make such programming errors, looking for them when debugging, and gaining a feel for how to avoid them are all essential experiences on the journey to become an effective parallel programmer.

Deadlock

Deadlocks are bad, and we will emphasize that understanding concurrency vs. parallelism (see last section of this chapter) is essential to understanding how to avoid deadlock.

Deadlock occurs when two or more actions (processes, threads, kernels, etc.) are blocked, each waiting for the other to release a resource or complete a task, resulting in a standstill. In other words, our application will never complete. Every time we use a wait, synchronization, or lock, we can create deadlocks. Lack of synchronization can lead to deadlock, but more often it manifests as a race condition (see prior section).

Deadlocks can be difficult to debug. We will revisit this in the "Concurrency vs. Parallelism" section at the end of this chapter.

Chapter 4 will tell us "lambda expressions not considered harmful." We should be comfortable with lambda expressions in order to use DPC++, SYCL, and modern C++ well.

C++ Lambda Expressions

A feature of modern C++ that is heavily used by parallel programming techniques is the lambda expression. Kernels (the code to run on a device) can be expressed in multiple ways, the most common one being a lambda expression. Chapter 10 discusses all the various forms that a kernel can take, including lambda expressions. Here we have a refresher on C++ lambda expressions plus some notes regarding use to define kernels. Chapter 10 expands on the kernel aspects after we have learned more about SYCL in the intervening chapters.

The code in Figure 1-3 has a lambda expression. We can see it because it starts with the very definitive `[=]`. In C++, lambdas start with a square bracket, and information before the closing square bracket denotes how to *capture* variables that are used within the lambda but not explicitly passed to it as parameters. For kernels in SYCL, the capture must be *by value* which is denoted by the inclusion of an equals sign within the brackets.

Support for lambda expressions was introduced in C++11. They are used to create anonymous function objects (although we can assign them to named variables) that can capture variables from the enclosing scope. The basic syntax for a C++ lambda expression is

```
[ capture-list ] ( params ) -> ret { body }
```

where

- *`capture-list`* is a comma-separated list of captures. We capture a variable by value by listing the variable name in the capture-list. We capture a variable by reference by prefixing it with an ampersand, for example, &v**.** There are also shorthands that apply to all in-scope automatic variables: `[=]` is used to capture all automatic variables used in the body by value and the current object by reference, `[&]` is used to capture all automatic variables used in the body as well as the current object by reference, and `[]` captures nothing. With SYCL, `[=]` is always used because no variable is allowed to be captured by reference for use in a kernel. Global variables are *not* captured in a lambda, per the C++ standard. Non-global static variables *can* be used in a kernel but *only* if they are `const`. The few restrictions noted here allow kernels to behave consistently across different device architectures and implementations.

- *`params`* is the list of function parameters, just like for a named function. SYCL provides for parameters to identify the element(s) the kernel is being invoked to process: this can be a unique `id` (one-dimensional) or a 2D or 3D `id`. These are discussed in Chapter 4.
- *`ret`* is the return type. If `->ret` is not specified, it is inferred from the return statements. The lack of a return statement, or a return with no value, implies a return type of `void`. SYCL kernels must *always* have a return type of `void`, so we should not bother with this syntax to specify a return type for kernels.
- *body* is the function body. For a SYCL kernel, the contents of this kernel have some restrictions (see earlier in this chapter in the "Kernel Code" section).

Figure 1-4 shows a C++ lambda expression that captures one variable, i, by value and another, j, by reference. It also has a parameter k0 and another parameter l0 that is received by reference. Running the example will result in the output shown in Figure 1-5.

```
int i = 1, j = 10, k = 100, l = 1000;

auto lambda = [i, &j](int k0, int& l0) -> int {
  j = 2 * j;
  k0 = 2 * k0;
  l0 = 2 * l0;
  return i + j + k0 + l0;
};

print_values(i, j, k, l);
std::cout << "First call returned " << lambda(k, l)
          << "\n";
print_values(i, j, k, l);
std::cout << "Second call returned " << lambda(k, l)
          << "\n";
print_values(i, j, k, l);
```

Figure 1-4. *Lambda expression in C++ code*

```
i == 1
j == 10
k == 100
l == 1000
First call returned 2221
i == 1
j == 20
k == 100
l == 2000
Second call returned 4241
i == 1
j == 40
k == 100
l == 4000
```

Figure 1-5. *Output from the lambda expression demonstration code in Figure 1-4*

We can think of a lambda expression as an instance of a function object, but the compiler creates the class definition for us. For example, the lambda expression we used in the preceding example is analogous to an instance of a class as shown in Figure 1-6. Wherever we use a C++ lambda expression, we can substitute it with an instance of a function object like the one shown in Figure 1-6.

Whenever we define a function object, we need to assign it a name (Functor in Figure 1-6). Lambda expressions expressed inline (as in Figure 1-4) are anonymous because they do not need a name.

```
class Functor {
 public:
  Functor(int i, int &j) : my_i{i}, my_jRef{j} {}

  int operator()(int k0, int &l0) {
    my_jRef = 2 * my_jRef;
    k0 = 2 * k0;
    l0 = 2 * l0;
    return my_i + my_jRef + k0 + l0;
  }

 private:
  int my_i;
  int &my_jRef;
};
```

Figure 1-6. *Function object instead of a lambda expression (more on this in Chapter 10)*

Functional Portability and Performance Portability

Portability is a key objective for using C++ with SYCL; however, nothing can guarantee it. All a language and compiler can do is to make portability a little easier for us to achieve in our applications when we want to do so. It is true that higher-level (more abstract) programming—such as domain-specific languages, libraries, and frameworks—can offer more portability in large part because they allow less prescriptive programming. Since we are focused on *data-parallel* programming in C++ in this book, we assume a desire to have more control and with that comes more responsibility to understand how our coding affects portability.

Portability is a complex topic and includes the concept of *functional portability* as well as *performance portability*. With functional portability, we expect our program to compile and run equivalently on a wide variety of platforms. With performance portability, we would like our program to get reasonable performance on a wide variety of platforms. While that is a pretty soft definition, the converse might be clearer—we do not want to write a program that runs superfast on one platform only to find that it is unreasonably slow on another. In fact, we would prefer that it got the most out of any platform upon which it is run. Given the wide variety of devices in a heterogeneous system, performance portability requires nontrivial effort from us as programmers.

Fortunately, SYCL defines a way to code that can improve performance portability. First of all, a generic kernel can run everywhere. In a limited number of cases, this may be enough. More commonly, several versions of important kernels may be written for different types of devices. Specifically, a kernel might have a generic GPU *and* a generic CPU version. Occasionally, we may want to specialize our kernels for a specific device such as a specific GPU. When that occurs, we can write multiple versions and specialize each for a different GPU model. Or we can parameterize one version to use attributes of a GPU to modify how our GPU kernel runs to adapt to the GPU that is present.

While we are responsible for devising an effective plan for performance portability ourselves as programmers, SYCL defines constructs to allow us to implement a plan. As mentioned before, capabilities can be layered by starting with a kernel for all devices and then gradually introducing additional, more specialized kernel versions as needed. This sounds great, but the overall flow for a program can have a profound impact as well because data movement and overall algorithm choice matter. Knowing that gives insight into why no one should claim that C++ with SYCL (or other programming solution) solves performance portability. However, it is a tool in our toolkit to help us tackle these challenges.

Concurrency vs. Parallelism

The terms *concurrent* and *parallel* are not necessarily equivalent, although they are sometimes misconstrued as such. Any discussion of these terms is further complicated by the fact that various sources rarely agree on the same definitions.

Consider these definitions from the Sun Microsystems *Multithreaded Programming Guide*:[3]

- **Concurrency**: A condition that exists when at least two threads are making progress
- **Parallelism**: A condition that exists when two threads are executing simultaneously

To fully appreciate the difference between these concepts, we need to seek an intuitive understanding of what matters here. The following observations can help us gain that understanding:

- **Executing simultaneously can be faked**: Even without hardware support for doing more than one thing at a time, software can fake doing multiple things at once by multiplexing. Multiplexing is a good example of concurrency without parallelism.

[3] The authors are fans of this programming guide's coverage of the fundamentals that never go away. It is online at docs.oracle.com/cd/E19253-01/816-5137/816-5137.pdf.

- **Hardware resources are limited**: Hardware is never infinitely "wide" because hardware always has a finite number of execution resources (e.g., processors, cores, execution units). When hardware can execute each of our threads using dedicated resources, we have both concurrency and parallelism.

When we as programmers say, "do X, Y and Z at the same time," we often do not actually care whether hardware provides concurrency or parallelism. We probably do not want our program (with three tasks) to fail to launch on a machine that can only run two of them simultaneously. We would prefer that as many tasks as possible are processed in parallel, repeatedly stepping through batches of tasks until they are all complete.

But sometimes, we *do* care. And mistakes in our thinking can have disastrous effects (like "deadlock"). Imagine that our example from the last paragraph was modified such that the last thing a task (X, Y, or Z) does is "wait until *all* the tasks are done." Our program will run just fine if the number of tasks never exceeds the limits of the hardware. But if we break our tasks into batches, a task in our first batch will wait forever. Unfortunately, that means our application never finishes.

This is a common mistake that is easy to make, which is why we are emphasizing these concepts. Even expert programmers must focus to try to avoid this—and we all find that we will need to debug issues when we miss something in our thinking. These concepts are not simple, and the C++ specification includes a lengthy section detailing the precise conditions in which threads are guaranteed to make progress. All we can do in this introductory section is highlight the importance of understanding these concepts as much as we can.

Developing an intuitive grasp of these concepts is important for effective programming of heterogeneous and accelerated systems. We all need to give ourselves time to gain such intuition—it does not happen all at once.

Summary

This chapter provided terminology needed for understanding C++ with SYCL and provided refreshers on key aspects of parallel programming and C++ that are critical to SYCL. Chapters 2, 3, and 4 expand on three keys to data-parallel programming while using C++ with SYCL: devices need to be given work to do (send code to run on them), be provided with data (send data to use on them), and have a method of writing code (kernels).