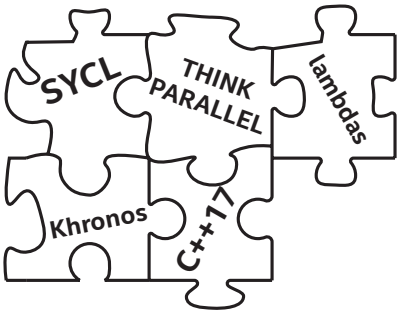
CHAPTER 1

Introduction



This chapter lays the foundation by covering core concepts, including terminology, that are critical to have fresh in our minds as we learn how to accelerate C++ programs using data parallelism.

Data parallelism in C++ enables access to parallel resources in a modern heterogeneous system. A single C++ application can use any combination of devices—including GPUs, CPUs, FPGAs, and AI Application-Specific Integrated Circuits (ASICs)—that are suitable to the problems at hand.

This book teaches data-parallel programming using C++ and SYCL.

SYCL (pronounced *sickle*) is an industry-driven Khronos standard that adds data parallelism to C++ for heterogeneous systems. SYCL programs perform best when paired with SYCL-aware C++ compilers such as the open source Data Parallel C++ (DPC++) compiler used in this book. SYCL is not an acronym; SYCL is simply a name.

J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-5574-2_1

DPC++ is an open source compiler project, initially created by Intel employees, committed to strong support of data parallelism in C++. The DPC++ compiler is based on SYCL, a few extensions,[1] and broad heterogeneous support that includes GPU, CPU, and FPGA devices. In addition to the open source version of DPC++, there are commercial versions available in Intel oneAPI toolkits.

Implemented features based on SYCL are supported by *both* the open source and commercial versions of the DPC++ compilers. All examples in this book compile and work with *either* version of the DPC++ compiler, and almost all will compile with recent SYCL compilers. We are careful to note where extensions are used that are DPC++ specific at the time of publication.

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 is no different. Like every great language specification, it is full of precision and light on motivation, usage, and teaching. This book is a "study guide" to teach SYCL and use of the DPC++ compiler.

As mentioned in the Preface, this book cannot 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 try to not get hung up on understanding the coding examples completely in Chapter 1 and trust it will get better with each chapter.

[1]The DPC++ team is quick to point out that they hope all their extensions will be considered, and hopefully accepted, by the SYCL standard at some time in the future.

SYCL 1.2.1 vs. SYCL 2020, and DPC++

As this book goes to press, the provisional SYCL 2020 specification is available for public comments. In time, there will be a successor to the current SYCL 1.2.1 standard. That anticipated successor has been informally referred to as SYCL 2020. While it would be nice to say that this book teaches SYCL 2020, that is not possible because that standard does not yet exist.

This book teaches SYCL with extensions, to approximate where SYCL will be in the future. These extensions are implemented in the DPC++ compiler project. Almost all the extensions implemented in DPC++ exist as new features in the provisional SYCL 2020 specification. Notable new features that DPC++ supports are USM, sub-groups, syntax simplifications enabled by C++17 (known as CTAD—class template argument deduction), and the ability to use anonymous lambdas without having to name them.

At publication time, *no* SYCL compiler (including DPC++) exists that implements *all* the functionality in the SYCL 2020 provisional specification.

Some of the features used in this book are specific to the DPC++ compiler. Many of these features were originally Intel extensions to SYCL that have since been accepted into the SYCL 2020 provisional specification, and in some cases their syntax has changed slightly during the standardization process. Other features are still being developed or are under discussion for possible inclusion in future SYCL standards, and their syntax may similarly be modified. Such syntax changes are actually highly desirable during language development, as we want features to evolve and improve to address the needs of wider groups of developers and the capabilities of a wide range of devices. All of the code samples in this book use the DPC++ syntax to ensure compatibility with the DPC++ compiler.

While endeavoring to approximate where SYCL is headed, there will almost certainly need to be adjustments to information in this book to align with the standard as it evolves. Important resources for updated

information include the book GitHub and errata that can be found from the web page for this book (www.apress.com/9781484255735), as well as the online oneAPI DPC++ language reference (tinyurl.com/dpcppref).

Getting a DPC++ Compiler

DPC++ is available from a GitHub repository (github.com/intel/llvm). Getting started with DPC++ instructions, including how to build the open source compiler with a clone from GitHub, can be found at intel.github.io/llvm-docs/GetStartedGuide.html.

There are also bundled versions of the DPC++ compiler, augmented with additional tools and libraries for DPC++ programming and support, available as part of a larger oneAPI project. The project brings broad support for heterogeneous systems, which include libraries, debuggers, and other tools, known as oneAPI. The oneAPI tools, including DPC++, are available freely (oneapi.com/implementations). The official oneAPI DPC++ Compiler Documentation, including a list of extensions, can be found at intel.github.io/llvm-docs.

The online companion to this book, the oneAPI DPC++ language reference online, is a great resource for more formal details building upon what is taught in this book.

Book GitHub

Shortly we will encounter code in Figure 1-1. If we want to avoid typing it all in, we can easily download all the examples in this book from a GitHub repository (www.apress.com/9781484255735—look for Services for this book: Source Code). The repository includes the complete code with build files, since most code listings omit details that are repetitive or otherwise

unnecessary for illustrating a point. The repository has the latest versions of the examples, which is handy if any are updated.

```
1. #include <CL/sycl.hpp>
2. #include <iostream>
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. const auto sz = secret.size();
9.
10. int main() {
11.  queue Q;
12.
13.  char*result = malloc_shared<char>(sz, Q);
14.  std::memcpy(result,secret.data(),sz);
15.
16.  Q.parallel_for(sz,[=](auto&i) {
17.     result[i] -= 1;
18.     }).wait();
19.
20.  std::cout << result << "\n";
21.  return 0;
22. }
```

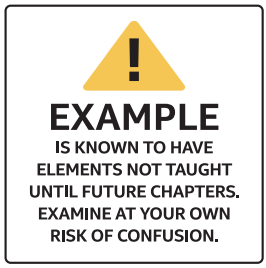


Figure 1-1. *Hello data-parallel programming*

Hello, World! and a SYCL Program Dissection

Figure 1-1 shows a sample SYCL program. Compiling with the DPC++ compiler, 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 particular example by the end of Chapter 4. Until then, we can observe the single include of `<CL/sycl.hpp>` (line 1) 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 11 establishes a queue for work requests directed to a particular device (Chapter 2).
- Line 13 creates an allocation for data shared with the device (Chapter 3).
- Line 16 enqueues work to the device (Chapter 4).
- Line 17 is the only line of code that will run on the device. All other code runs on the host (CPU).

Line 17 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 actually run asynchronously relative to the main program once the `parallel_for` queues the work. It is critical that there is a wait (line 18) before looking at the result to be sure that the kernel has completed, since in this particular 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 much more to discuss, but that is the job of later chapters.

Queues and Actions

Chapter 2 will discuss 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`, `parallel_for` (used in Figure 1-1), or `parallel_for_work_group`. 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 these 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 actually stretch out the time necessary to get a single work-item done, in order 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 one hundred GPUs, we would 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 presented an article titled "Reevaluating Amdahl's Law." He observed that parallelism was not used to speed up fixed workloads, but rather 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.

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 quite 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 will start 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

The phrase "heterogeneous system" snuck into the prior paragraph. 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*. Chapter 2 will begin the discussion of how to steer work (computations) to particular devices in a heterogeneous system.

GPUs have evolved to become high-performance computing 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 generally 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 SYCL, the DPC++ compiler, and the majority of what this book has to teach.

SYCL was created to address the challenges of C++ data-parallel programming for heterogeneous 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 SYCL and DPC++. Chapter 4 starts this discussion, and it continues through the rest of the book.

Key Attributes of DPC++ and SYCL

Every DPC++ (or SYCL) program is also a C++ program. Neither SYCL nor DPC++ relies on any language changes to C++. Both can be fully implemented with templates and lambda functions.

The reason SYCL compilers[2] exist is to optimize code in a way that relies on built-in knowledge of the SYCL specification. A standard C++ compiler that lacks any built-in knowledge of SYCL cannot lead to the same performance levels that are possible with a SYCL-aware compiler.

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

Single-Source

Programs can be single-source, meaning that the same translation unit[3] 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]It is probably more correct to call it a C++ compiler with support for SYCL.

[3]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 DPC++ and SYCL programs. As we will see shortly, devices do not need to support all of C++17.

Devices

Using multiple devices in a program is what makes it heterogeneous programming. That's 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 target for *acceleration offload* that SYCL promises. The idea of offloading computations is generally 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 or DPC++. 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 Data Parallel C++ 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 SYCL or DPC++: it is a core concept in other offload acceleration languages including OpenCL and CUDA.

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.

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

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

Kernel: Vector Addition (DAXPY)

Kernels should feel familiar to any programmer who has work 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 will 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.

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

```
1. // C++ loop
2. for (int i=0;i<n;i++) {
3.   z[i] = alpha * x[i] + y[i];
4. }
```

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

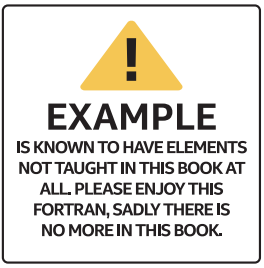


Figure 1-2. *DAXPY computations in Fortran, C++, and SYCL*

Asynchronous Task Graphs

The asynchronous nature of programming with SYCL/DPC++ 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 4 will start 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 18 to prevent line 20 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 14 uses `std::memcpy` to load the input. Since `std::memcpy` runs on the host, line 16 and later do not execute until line 15 has completed. After reading Chapter 3, we could be tempted to change this to use `myQ.memcpy` (using SYCL). We have done exactly that in Figure 1-3 in line 8. Since that is a queue submission, there is no guarantee that it will complete before line 10. 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 8 and then read it in line 10, we do not want a race that might have line 17 execute

before line 8 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 `myQ.memcpy` to complete before proceeding by adding `.wait()` to the end of line 8. 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 waiting automatically for us.

```
1. // ...we are changing one line from Figure 1-1
2. char *result = malloc_shared<char>(sz, Q);
3.
4. // Introduce potential data race!
5. // We don't define a dependence
6. // to ensure correct ordering with
7. // later operations.
8. Q.memcpy(result,secret.data(),sz);
9.
10. Q.parallel_for(sz,[=](auto&i) {
11.   result[i] -= 1;
12. }).wait();
13.
14. // ...
```

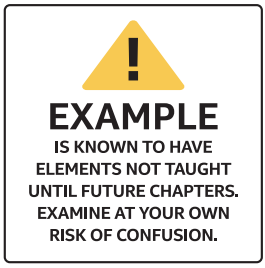


Figure 1-3. *Adding a race condition to illustrate a point about being asynchronous*

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 tradeoffs that balance program simplicity with efficient use of our systems.

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 somewhat 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.

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

C++ Lambda Functions

A feature of modern C++ that is heavily used by parallel programming techniques is the lambda function. Kernels (the code to run on a device) can be expressed in multiple ways, the most common one being a lambda function. Chapter 10 discusses all the various forms that a kernel can take, including lambda functions. Here we have a refresher on C++ lambda functions 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 function. 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, 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 almost 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`.
- *`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).

```
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 function 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 function demonstration code in Figure 1-4*

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.

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). Lambdas 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 (more on this in Chapter 10)*

Portability and Direct Programming

Portability is a key objective for SYCL and DPC++; however, neither can guarantee it. All a language and compiler can do is make portability a little easier for us to achieve in our applications when we want to do so.

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'd prefer that it got the most out of any platform that it is run upon. Given the wide variety of devices in a heterogeneous system, performance portability requires non-trivial 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 SYCL (or other direct 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 equivalent, although they are sometimes misconstrued as such. It is important to know that any programming consideration needed for concurrency is also important for parallelism.

The term concurrent refers to code that can be advancing but not necessarily at the same instant. On our computers, if we have a `Mail` program open and a `Web Browser`, then they are running concurrently. Concurrency can happen on systems with only one processor, through a

process of time slicing (rapid switching back and forth between running each program).

Tip Any programming consideration needed for concurrency is also important for parallelism.

The term parallel refers to code that can be advancing at the same instant. Parallelism requires systems that can actually do more than one thing at a time. A heterogeneous system can always do things in parallel, by its very nature of having at least two compute devices. Of course, a SYCL program does not require a heterogeneous system as it can run on a host-only system. Today, it is highly unlikely that any host system is not capable of parallel execution.

Concurrent execution of code generally faces the same issues as parallel execution of code, because any particular code sequence cannot assume that it is the only code changing the world (data locations, I/O, etc.).

Summary

This chapter provided terminology needed for SYCL and DPC++ and provided refreshers on key aspects of parallel programming and C++ that are critical to SYCL and DPC++. Chapters 2, 3, and 4 expand on three keys to SYCL programming: 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).