Data Parallel C++

Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Chapters 1-4

UNEDITED ADVANCE REVIEW COPY

Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, Xinmin Tian, and James R. Reinders

Apress®

Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

ISBN-13 (pbk): TBA

ISBN-13 (electronic): TBA

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Natalie Pao
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at TBA. For more detailed information, please visit http://www.apress.com/source-code

Table of Contents

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

FOR THIS BOOK PREVIEW (CHAPTERS 1-4):
ERRATA, NOTES, DOWNLOADS, FEEDBACK, ETC.

Please check our "preview book" website for information including errata, updates, and downloads (includes all sample code): https://tinyurl.com/book-dpcpp

We will also list some additional resources as they become available.

Your feedback is welcome. You can email James Reinders at dpc++@jamesreinders.com with any suggestions, encouragement, criticism, or questions that you may have. James will be sure to share any feedback that you send with all the authors.

Of course – watch for the full book, by mid-2020, available from Apress (no charge for PDF for the completed book, print copies will be available too).

https://tinyurl.com/book-dpcpp

We are pleased to share a preview of the beginning of our new book on data parallel programming in C++, for Heterogeneous Systems, using DPC++.

We have worked hard to make this text very useful — we believe it has been well-reviewed for content and correctness. However, we are far from done on formatting at this point, so we ask you to look past the imperfect layout and formatting in this preview edition.

While we will finish this book by mid-2020, we have decided to release the first four chapters to introduce DPC++ now. This will afford you, the reader, a chance to give DPC++ a try, to learn, and to give us feedback.

These four chapters form an excellent introduction to the entire topic. Remaining chapters will dive deeper into specific topics (we've attached a list at the end of this posting).

All the features of DPC++, and 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 DPC++ compiler.

Errata, Notes, Downloads, Feedback

Please check our "preview book" website for information including errata, updates, and downloads (includes all sample code): https://tinyurl.com/book-dpcpp

We will also list some additional resources as they become available. Your feedback is welcome. You can email James Reinders at dpc++@jamesreinders.com with any suggestions, encouragement, criticism, or questions that you may have. James will be sure to share any feedback that you send with all the authors.

oneAPI

Visit https://oneAPI.com for information on the oneAPI. Find resources for software developers at https://software.intel.com/oneAPI — including the oneAPI toolkits, discussion forums, online training, event information, and more.

The oneAPI website hosts some training on oneAPI, which contains a number of training modules dedicated to DPC++. While we expect the book to dig deeper than the online classes in many respects, listening to the training should be a very useful companion to the book.

The online training also explains how to register for an online account to access the tools already installed with access to CPUs, GPUs, and FPGAs.

Enjoy Learning DPC++ and SYCL

It is our hope that our book supports and helps grow the SYCL community, as we help promote data parallel programming in C++ using the DPC++ compiler.

We wish you the best as you learn DPC++, and SYCL. We welcome your feedback. We hope you will enjoy this preview, and be watching for the full book, by mid-2020, available from Apress in print, and as a free download.

Michael Kinsner,
James Brodman,
John Pennycook,
Xinmin Tian,
Ben Ashbaugh,
and James Reinders

November, 2019

Introduction

We are very excited to be teaching data-parallel programming for C++ in this book through use of DPC++ and SYCL.

SYCL is an industry led, Khronos standard, for adding data parallelism to C++ for heterogeneous systems. We will define data parallelism, and heterogeneous systems, in the next few pages.

DPC++ is an extension of C++, incorporating SYCL and other new features. There is an open-source DPC++ compiler, initially created by Intel, available from a GitHub repository. There is a commercial version of the DPC++ compiler, augmented with additional tools and libraries for DPC++ programming and support, available from Intel.

All the features of DPC++, and 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 DPC++ compiler.

In this book we introduce DPC++ for the first time, while also being the first full book teaching SYCL.

It is our hope that this book supports and helps grow the SYCL community.

To avoid writing "SYCL and DPC++" or "DPC++ and SYCL" over and over in this book, we do two things:

(1) we speak of DPC++ in an inclusive manner, and

(2) we speak of SYCL features with the understanding they are always DPC++ features as well.

Most of the examples in this book should work with any up-to-date SYCL compiler including the DPC++ compilers. Early in each chapter we clearly state when extensions unique to DPC++ are used.

DPC++ AND SYCL IN CHAPTER 1

This chapter introduces DPC++ and SYCL. A box like this one, early in each chapter, will clearly state when extensions unique to DPC++ are used within the chapter. The sample code in this chapter does not use any DPC++ specific extensions.

DPC++ includes all of SYCL. When something is specific to only DPC++, we will go out of our way to be clear.

Our book will do considerably more than introduce — we will dive into the key issues and solutions that are encountered while constructing effective data-parallel programs, with C++, to program the powerful heterogeneous systems we have available today.

What is Data-Parallel Programming?

DPC++ and SYCL are designed to encourage and support a data-parallel programming style, although they do not limit us to only data-parallel programming.

Data-parallel programming might be described as both a mindset, as well as a way of programming. Data is operated on in parallel by a collection of processing elements. Each processing element is hardware capable of some computation on the data. These processing elements may exist on a single device, or many devices in our computer systems. We specify our code to work on our data in the form of a *kernel*.

When programming in a data-parallel fashion, we focus on specifying what operations (written as a *kernel*) should apply to every data element. Data-parallelism applies best on regular data structures like arrays and matrices because they may offer many opportunities for parallel computations that are easy to specify.

An important concept in data-parallel programming is the kernel — a function that will be executed on a device containing processing elements. The term kernel is used in data-parallel programming including SYCL, OpenCL, CUDA, and DPC++.

In contrast, when programming in a task-parallel fashion, we specify a distribution of code among processing elements and then work to have the data travel to the computations as dictated by the code.

Parallel programmers generally use elements of both data-parallel and task-parallel programming, and therefore the features of DPC++ and SYCL are not all strictly data parallel only. One example is that SYCL allows running two *separate* submissions in parallel. Blending data-parallel programming and task-parallel programming is to be encouraged, and we hope that will seem natural and normal after reading this book. A "Think Parallel" mindset will remain critical to make the most of such knowledge — know where your parallelism is, and have a plan to exploit it!

DPC++ and SYCL are designed to encourage and support a data-parallel programming style, although they do not limit us to only data-parallel programming.

What are Heterogeneous Systems?

For our purposes, a heterogenous system is any system which contains multiple types of computational devices. For instance, a system with both a CPU (Central Processing Unit, often simply called a processor) and a GPU (Graphics Processing Unit, with computationally intensive versions often referred to as General-Purpose GPU, or GPGPUs) is a heterogeneous system. Today, the collection of devices includes CPUs, GPUs, FPGAs (Field Programmable Gate Arrays), DSPs (Digital Signal Processors), ASICs (Application Specific Integrated Circuits), and AI chips (graph, neuromorphic, etc.).

Having multiple types of devices, each with different architectures and therefore different characteristics, leads to different programming and optimization needs for each device. This creates a number of challenges —helping solve these being the motivation behind DPC++.

DPC++ exists to address the challenges of data-parallel programming for heterogeneous systems.

Enjoy the Journey – This is something BIG

Seeking to support data parallelism for programming heterogeneous systems within C++ is no small vision. We hope you enjoy the journey as much as we do. The emergence of both open source, and commercial, support for SYCL and DPC++ is strong evidence that we are on the cusp of something big.

Parallel Programming in C++

Parallel programming is a wide and complex topic – and we focus on a key aspect of it, namely language support for data-parallel programming. This is distinct from, and complementary to, the highly portable facilities of Threading Building Blocks (TBB) and today's C++ standard that address other key aspects of parallel programming. Language support for data parallelism is critical because it allows developers to express their data parallel algorithms more directly, and enables compiler optimization technologies. Prior efforts including OpenMP and OpenCL have given us substantial real-world experience upon which to build. That experience has led to a deep understanding of how critical it is to make data-parallel programming a first-class citizen in a programming language and support that directly with compiler technology.

DPC++ is modern C++.

DPC++ should feel familiar to C++11 programmers.

DPC++ will act as a proving ground for C++ features supporting heterogeneous platforms and parallelism.

DPC++ will provide valuable input to future ISO C++ standards.

Journey to Parallelism, Scaling, Portability

Parallel programming has come a long way over multiple decades and we can now better assess the most effective ways to support data-parallel programming. The heterogeneous nature of today's computing systems must be accounted for as well. Being effective at both data parallelism and heterogeneous programming requires good solutions for scaling and portability.

Real Estate is all about location, location, and location.
Parallel Programming is all about scaling, scaling, and scaling.

Scaling is a measure of the effectiveness of parallel programming, which measures the degree to which a program effectively uses more parallel hardware when available. We must never lose sight of the objective, for any parallel programming system, to allow for effective scaling. This includes scaling as we move our application across platforms. Heterogeneous platforms confound the problem because different architectures can have substantially different needs. We do not pretend that DPC++ solves scaling and all heterogeneous platform needs for us. Rather, DPC++ is a tool that allows us to succeed by helping us to deal with issues that would otherwise interfere with our scaling goals, and by offering us techniques to address the needs of heterogeneous platforms. That may sound easier than it is; that DPC++ succeeds at this is quite an accomplishment.

DPC++ forms a solid foundation on which to build scalable portable applications.

Portability is a complex topic and includes the concept of *functional portability* (the program works) as well as *performance portability* (the program achieves high performance when ported to a new machine). DPC++ addresses both by giving us, as programmers, the tools we need to succeed. DPC++ does not magically make the issues disappear — DPC++ forms a solid foundation on which to build portable applications that also scale. Not every program can scale, but DPC++ helps us when they can — and even then, we are responsible for devising an effective plan for performance portability ourselves as programmers.

Constantly question your own approaches with respect to their impact on both scaling and the multiple facets of portability.

We will touch on scaling and performance portability throughout this book. We will teach to reinforce both capabilities, usually without dwelling on them specifically. We encourage you to constantly question your own approaches with respect to their impact on both scaling and the multiple facets of portability for your own applications.

First look at DPC++ code

Figure 1-1 shows a sample SYCL program. In this example, only a short segment of code is destined to run on a *device* (the default accelerator if one exists on the system, or the host via the always available *host device* if no accelerator is present). Coding in DPC++ or SYCL has a "single source" property, which means that host code and device code can exist within the same source file. This is illustrated in the example in Figure 1-1. Supporting single source allows host and device code to be easily considered, and also optimized, together by a compiler. This, it turns out, is a critical advantage in using DPC++ or SYCL. DPC++ and SYCL also allow for compilation of the device code to be completed at runtime for more flexibility. Even in such a case, significant optimization can occur, in advance, during the compilation of the application. We will discuss this more in an upcoming section on the Compilation Model.

In Figure 1-1, the motivations for chapters 2, 3, 4 are highlighted. Chapter 2 will describe how to control *where* our code runs — in other words, on what device(s) code will run. Chapter 3 will describe how to manage data effectively — so that it shows up where and when it is needed. Chapter 4 will describe how to write kernels — the essence of code to be run on a device. Additional chapters follow to expand on the basics learned in the first four chapters.

Why DPC++?

There are several reasons that DPC++ is needed, and is the right answer for us today:

- We need to address the critical need for supporting data-parallel programming as a first-class citizen — it needs to be portable, high-level, and non-proprietary while addressing modern heterogeneous computer architectures.
- We want the same programming environment across host and device code: support for modern C++ features (in particular, templates), better type safety, and for data parallel code to look and feel like sequential code.

- The future of computer architecture includes accelerators that span scalar, vector, matrix, and spatial (SVMS) operations (coming up in Figure 1-2); support for heterogeneous machines including the SVMS capabilities are needed. Support should span highly complex and programmable devices, as well as fixed function or specialized devices that are less programmable.

```
#include <CL/sycl.hpp>
#include <iostream>
#include <array>
#define SIZE 1024
using namespace cl::sycl;
int main() {
  std::array<int, SIZE> a, b, c;
  for (int i = 0; i<SIZE; ++i) {
    a[i] = i;
    b[i] = -i;
    c[i] = i;
  }
  {
    range<1> a_size{SIZE};
    auto platforms = platform::get_platforms();
    for (auto &platform : platforms) {
      std::cout << "Platform: "
        << platform.get_info<info::platform::name>() << std::endl;
      auto devices = platform.get_devices();
      for (auto &device : devices ) {
        std::cout << " Device: "
          << device.get_info<info::device::name>()
          << std::endl;
      }
    }
    queue d_queue;
    buffer<int, 1> a_device(a.data(), a_size);
    buffer<int, 1> b_device(b.data(), a_size);
    buffer<int, 1> c_device(c.data(), a_size);
    d_queue.submit([&](handler &cgh) {
      auto c_res =
        c_device.get_access<access::mode::write>(cgh);
      auto a_in =
        a_device.get_access<access::mode::read>(cgh);
      auto b_in =
        b_device.get_access<access::mode::read>(cgh);
      cgh.parallel_for<class ex1>(a_size,[=](id<1> idx) {
        c_res[idx] = a_in[idx] + b_in[idx];
      });
    });
  }
}
```

Chapter 2: controlling where device code will be run

Chapter 3: sharing data within the system

Chapter 4: writing device code

Figure 1-1: First look at DPC++/SYCL programming

- When targeting multiple architectures (e.g., CPUs., GPUs, FPGAs), access to performance is important – even if it takes programmer effort. Hiding the controls where a programmer cannot access them is unacceptable.
- Programming should expose higher level abstractions and language mechanisms than those presented by OpenCL or CUDA, and in a non-proprietary form. Doing so will give us higher programmer productivity and lower the barrier to entry for new programmers.

DPC++ addresses all of these needs. DPC++ serves as an open alternative to proprietary, device-specific languages.

Why must it be a first-class citizen?

We use the phrase 'first-class citizen in the language' to express a need that the features we discuss be a known part of the language, so that it is both portable and can be well supported (and especially optimized!) by a compiler.

A compiler "knowing" SYCL and DPC++ will always be able to generate parallel code more effectively than a pure template library approach. However, we must note that SYCL and DPC++, are technically implementable as template libraries. They do not rely on new keywords or syntactic changes to C++. Nevertheless, effective implementations of DPC++ or SYCL *do* require compile support and a tuned runtime. This differs from Threading Building Blocks (TBB), which relies only on its template implementation and does not require modified compilers for performance. The DPC++ approach is similar to OpenMP, which will compile with any compiler, but will not yield benefits without an optimizing compiler that "knows" OpenMP.

Standardization into a language such as C++ is a long and careful process with constraints aimed to ensure a consistent experience far into the future. We hope that DPC++, and readers of this book, will provide valuable input into this process.

Made for each other: Data parallelism and Big Data

In addition to discussing how to program and control devices, we will also discuss data management repeatedly throughout this book. This is because assigning work to a device is complicated by data, beyond simply saying "do the work over there!" Data movement costs time and power and has to be managed as a first-class citizen. With data parallelism,

we are likely to manage a *lot* of data (think "Big Data"), so this is not a trivial concern. We cannot afford to behave like the proverbial ostrich sticking its head in the sand hoping all the data movement will be optimally handled with no effort from us. On the other hand, we hope that much of it is automatic so we can focus our programming where it matters most.

Data parallelism and Big Data go hand-in-hand; data movement costs time and power, and has to be managed as a first-class citizen.

SYCL, DPC++, C++

The focus of this book is "Data-Parallel C++" and it is realized via "SYCL with extensions." Please allow us to explain.

Khronos SYCL

SYCL (pronounced 'sickle') represents an industry standardization effort that includes support for data-parallel programming for C++. It is summarized as "C++ Single-source Heterogeneous Programming for OpenCL." The SYCL standard, like OpenCL, is managed by the Khronos Group.

SYCL is a cross-platform abstraction layer that builds on OpenCL. It enables code for heterogeneous processors to be written in a "single source" style using C++. This is not only useful for us as programmers, but it also gives a compiler the ability to analyze and optimize across the entire program regardless of the device on which the code is to be run.

Unlike OpenCL, SYCL includes templates and lambda functions to enable higher-level application software to be cleanly coded with optimized acceleration of kernel code. Developers program at a higher level than OpenCL but always have access to lower-level code through seamless integration with OpenCL, as well as C/C++ libraries.

DPC++

In order to fully address the needs of C++ programmers, a number of features are needed that are not currently within SYCL. For now, critical support for Unified Shared Memory (USM), and subgroups, exist only in DPC++.

The extensions we include in DPC++, to the extent they can be applied across many platforms, may be submitted for consideration in future SYCL specifications. We

cannot guarantee that they will be incorporated, or be unchanged. We do work hard in the SYCL community, and in our book, to minimize the need for changes.

DPC++ includes *all* of SYCL. We may not dive into all aspects of DPC++ and SYCL in this book because our focus is teaching effective data-parallel programming. Regardless, all of SYCL is included in DPC++ and so these features are available even if not mentioned in this book.

While we teach DPC++ and SYCL in this book – we do not intend this book to be an exhaustive reference for all capabilities.

We focus on DPC++ and SYCL features (including extensions) that teach effective data-parallel programming.

For the parts of DPC++ which are SYCL features, we will describe them as SYCL features. For any extensions, which are not part of the SYCL specification at the time we write this (SYCL 1.2.1r5), we will not label them as SYCL features. Of course, the SYCL standard will evolve and may incorporate a feature in the future that will not be labeled as SYCL in this book because of the timing.

Each chapter in this book includes a summary box near the beginning of the chapter as shown next. It is possible that by the time you read this book, some features we call out as unique to DPC++ may have been incorporated in some form by a future SYCL standard. We encourage you to browse our online errata (see Preface and Epilogue for URL), where we plan to make notes about such developments.

DPC++ AND SYCL IN CHAPTER X

Features that are discussed, but unique to DPC++ are called out clearly.

Otherwise, most or all of each chapter will be SYCL features and will entirely be included in DPC++.

C++

C++ lacks direct support for data parallelism in the language, and it lacks the controls needed for modern heterogeneous systems. We expect consensus to grow on the need and

the methods to solve these challenges. Therefore, you can see why we say that the focus of this book is "Data-Parallel C++" and it is realized via "SYCL with extensions."

Like the rest of the SYCL community, we hope many of the key features of SYCL will be considered for future C++ standards. Time will tell.

Why Heterogeneous Systems?

Heterogeneous computing matters as computer architects work to limit power consumption, reduce latency, and increase throughput. From the early 1990s until 2006, a so-called "free lunch" (continual performance boosts to virtually all applications) existed because processor performance was reliably doubling every two to three years, mostly because clock rates were doubling every couple of years. That era ended around 2006, and a new era of multicore and then many core processors emerged. The architectural shift to parallel processing gave multitasking systems a boost but did not give most existing *single* applications a performance boost without alterations in programming. This new era also saw accelerators, such as GPUs, gain popularity for accelerating more parts of applications than ever before. That gave rise to an era of heterogeneous computing, and a deluge of proposed accelerators with their own specialized processing capabilities, and with many different programming models.

Trading Generality for Performance: Architects of Accelerators boost performance for specific algorithms by reducing generality.

By being more specialized, accelerator designs can offer higher performance on specific problems because they do not have to be designed to handle every problem. This is a classic computer architecture trade-off. This generally means accelerators can only support subsets of the programming languages that were designed for processors. In fact, in DPC++ only code written in a kernel can be targeted to an accelerator. As we will see in Chapter 4, kernel code has some restrictions on it that do not exist for C++ code in general. Worrying about these restrictions would be a needless distraction because the uniqueness of each accelerator should be on our minds as programmers.

We need to think about the programming model, formulation of algorithms, and the types of workloads that an accelerator will perform well on.

Accelerator architectures can be bucketed into broad categories that impact how we need to think about the programming model, formulation of algorithms, and the types of workloads that an accelerator will perform well on. Like all attempts to categorize, there are many possibilities. Figure 1-2 utilizes a taxonomy of Scalar, Vector, Matrix, and Spatial (SVMS).

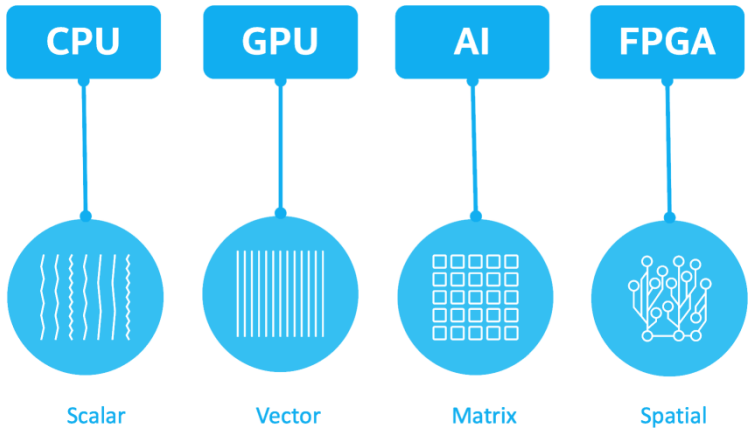


Figure 1-2: One possible taxonomy for highlighting unique strengths of devices within a heterogeneous system — should not be read to diminish the multifaceted capabilities of particular devices.

CPUs are the best choice for general purpose code including scalar and decision making (frequently branching) code, and often have built in accelerators for vectors. GPUs seek to accelerate vectors, and the closely related tensors. DSPs seek to accelerate specific mathematical operations with low latency, often in designs dealing with analog signals such as a cellphone. Accelerators for AI generally seek to accelerate matrix operations, although some may accelerate graphs as well. FPGAs and ASICs are particularly suited to accelerate spatial problems, including problems expressed in terms of flow graphs or pipelines.

While these buckets may be as good as any, they are only rough buckets for thinking — they are certainly not absolute rules. We can note that virtually all modern

processors have some form of vector, or even matrix, support. The point of Figure 1-2 focuses more on observations about unique advantages, such as the fact that processors can prove uniquely capable at scalar when accelerators falter. Categorization can be useful for thinking, but we should not get hung up on them too much.

Platform Model

The platform model, used by SYCL and DPC++, specifies a host that coordinates and controls the compute work that is performed on the devices. A device is an accelerator, presumably with specialized capabilities. Chapter 2 describes how to assign work to devices, and Chapter 4 dives into how to program devices.

As we discuss in Chapter 2, there is always a device corresponding to the host, known as the *host device*. Providing this guaranteed-to-be-available target for device code allows device code to be written assuming at least one device is available, even if it is the host itself! The choice of the devices on which to run device code is under program control — it is entirely our choice as programmer if, and how, we want to execute code on specific devices.

Amdahl's Law is a formula to predict the theoretical speedup when using multiple processors to do a fixed workload. Maximum gain from parallelism is limited to `(1/(1-p))` where `p` is the fraction of the program that runs in parallel (e.g., 2/3rd run in parallel, limits speed-up to 3X).

We must take care in our program to overlap work in the system as well as to hide latencies caused by data movement. Otherwise, using the host to dispatch work serially can become a serious performance bottleneck due to Amdahl's Law. This is why the programming model has us queue work to a device and not idly wait for its conclusion, gives methods to describe dependencies between work items, and provides numerous data management capabilities. These help to lower the time spent outside of parallel execution, as well as to free the host for doing important work itself. The most effective programs make efficient use of both the host and devices to get work done.

Multiarchitecture (aka "Fat") Binaries

Since our goal is to have a single source code to support a heterogeneous machine, it is only natural to want a single executable file to be the result.

A multiarchitecture binary (also called a *fat binary*) is a single binary file that has been *fattened* (expanded) to include all the compiled and intermediate code needed for our heterogeneous machine. A multiarchitecture binary acts like we are used to having an `a.out`, or `A.EXE`, operate for us — but it contains everything needed for a heterogeneous machine. This helps automate the process of picking the right code to run for a particular machine. As we discuss next, one form of the device code in a fat binary is an intermediate form that defers the final code creation until runtime.

Compilation Model

The single source nature of SYCL and DPC++ allows compilation to "just work." In other words, we do not need to know all the details of how it works because of the similarity to a standard C++ compilation.

Since the compilation model supports code that executes on both a host and potentially several accelerators simultaneously, the commands issued by the compiler, linker, and other supporting tools are more complicated than the C++ compilations we are used to (targeting only one architecture). Welcome to a heterogeneous world!

While this complexity is hidden from us by default and "just works," advanced users may want to understand these details to better target specific architectures.

In particular, compiling for multiple architectures presents several challenges, particularly because some may be known and some unknown at the time of compilation.

The DPC++ compiler can generate target-specific executable code similar to traditional C++ compilers (*ahead-of-time* compilation, sometimes referred to as 'offline kernel compilation'), or it can generate an intermediate representation that can be *just-in-time* compiled to a specific target at run time.

DPC++ compilation can be "ahead of time" or "just in time."

By default, when we compile our code (for most devices), the output for device code is in an intermediate form. At runtime, the device handler on the system will *just-in-time* compile the intermediate form into code to run on the device(s) to match what is available on the system.

We can ask the compiler to compile ahead-of-time for specific devices, or class of devices, in advance. This has the advantage of saving runtime, but it has the disadvantage of added compile time and fatter binaries! Compiling for a specific device ahead-of-time also helps us check at build time that our program should work on that device. With just-in-time, it is possible that a program will fail to compile for a specific device only when we try to run on it if device code uses features that are only optionally exposed by devices such as images.

Figure 1-3 illustrates the compilation process from source code to fat binary (executable). Whatever combinations we choose are bundled together in a fat binary. The fat binary is employed by the runtime when the application executes and the particular form needed for the target device is determined at runtime.

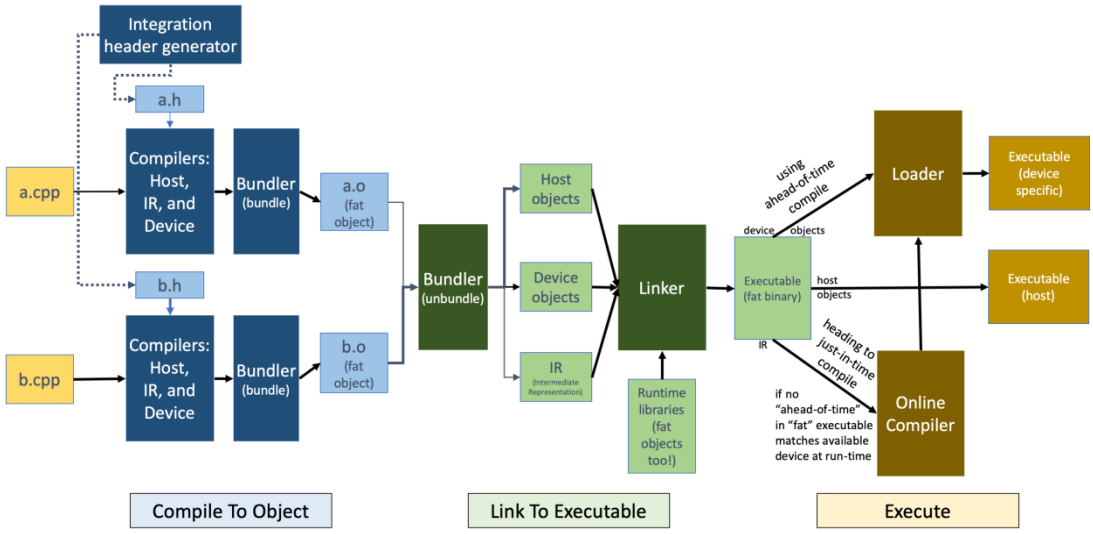


Figure 1-3: Compilation process showing ahead-of-time and just-in-time options using fat binary (executable).

Truth and Fallacy of Write Once, Run Everywhere

When we consider real heterogeneous systems, we might notice that the accelerators are there for specific reasons. The accelerators are not added into a computer design in order to run all the code, and in systems with multiple accelerators they are not there to be redundant with each other.

A single application may use multiple accelerators, but with specific purposes for each. This gives rise to two different usage scenarios to consider (illustrated in Figure 1-4 and Figure 1-5), both of which are equally well supported by DPC++.

In Figure 1-4, we consider that an application may use different algorithms that may be designed to perform on a particular type of accelerator if present. Such a program can run everywhere, and the CPU is used unless a well-suited accelerator is available for a particular algorithm. However, we do not try to make each algorithm run efficiently on all possible accelerators. While DPC++ code will be able run on all accelerators, there are options for "fallback" if and when an accelerator does not support a particular feature. We discuss coding for "fallback" in Chapter 2.

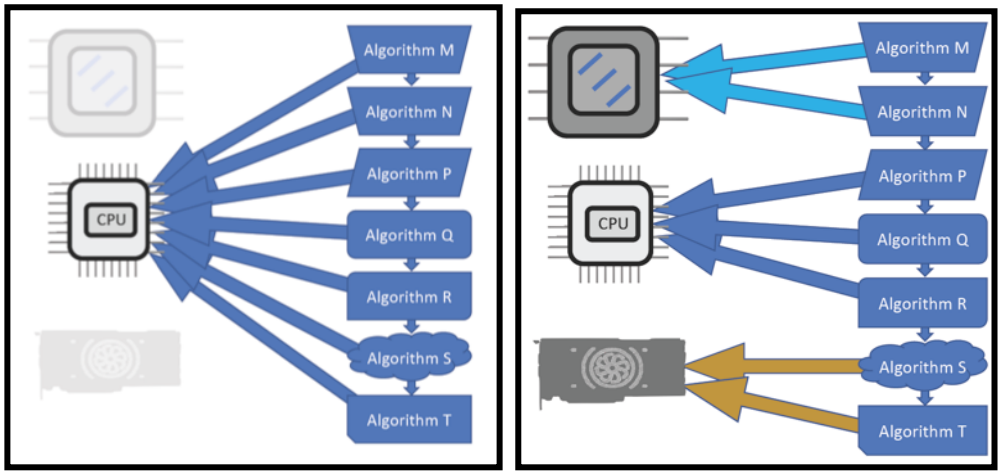


Figure 1-4: Left: Fallback on Host (CPU) if no device present, Right: Offload different algorithms to the devices with best match for the needs of each algorithm.

In Figure 1-5, we consider that some algorithms may be suitable for acceleration on a variety of devices. Inferencing, for instance, might be suitable for different accelerators — perhaps offering different tradeoffs in latency, performance, and power consumption. Again, such a program can be designed to run everywhere and the particular algorithm may be suitable for multiple accelerators. Even in this case, making each algorithm run efficiently on many different accelerators is likely to be a complex task.

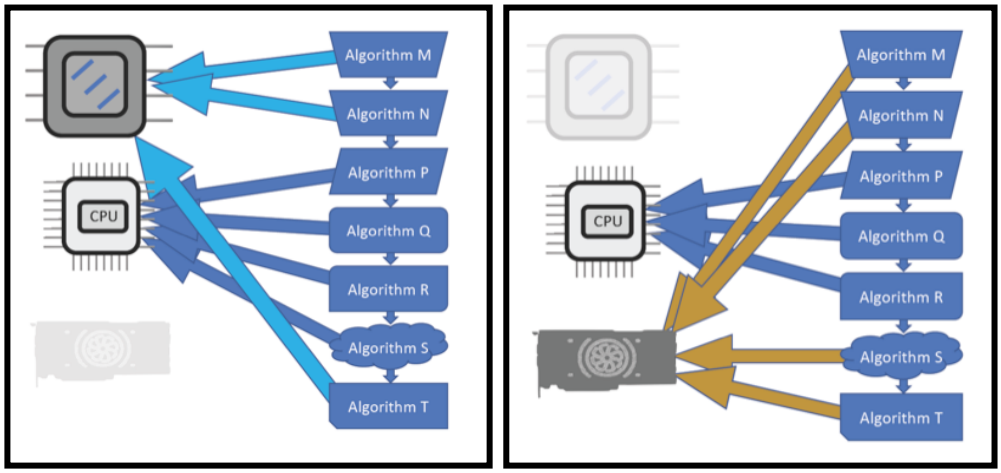


Figure 1-5: Left: the top device can serve the needs of Algorithms M, N, and T; Right: The bottom device can serve the needs of Algorithms M, N, S, and T. If both devices, or neither, are present then refer to Figure 1-4.

Direct programming

In our two scenarios illustrated in Figure 1-4 and Figure 1-5, we can write our code once and let the compiler translate it for each accelerator. However, it is unlikely that a single coding will run efficiently on all accelerators when you consider the variety of architectures that give rise to accelerators being interesting in the first place! As we illustrated in Figure 1-2, specific accelerators excel at specific styles of algorithms. Therefore, DPC++ gives us a way to specify different coding for different accelerators when we want to do so. This is called direct programming. As we will show in Chapters 13, 14, and 15, this direct programming is primarily motivated by architecture of a particular class of accelerator, and only a little by vendor specific features. In other words, coding for GPU is largely an exercise to aligning to the style of processing GPUs excel at doing, and coding for an FPGA is largely an exercise in aligning to the strengths of an FPGA. Of course, for the case that an accelerator is not always present we want to use the full capabilities of the CPU which may require we not rely on a version specifically optimized for an accelerator.

Using direct programming is necessary because compilers simply do not perform algorithm conversions.

Using direct programming is necessary because compilers simply do *not* perform algorithm conversions. For instance, if we code a *bubble sort* (not a great sorting algorithm in general) we do not expect the compiler to refactor the code into a quicksort (generally considered a very good sort algorithm). Instead, it is our job as programmers to either code a *quick sort*, or to use a library routine called 'sort' and hope it uses the best algorithm.

Compilers do not convert a bubble sort algorithm to a quick sort algorithm for us. That is our job as programmers. Likewise, a completely generic matrix multiply will not be compiled into an optimized matrix multiply for a GPU.

Similarly, DPC++ gives us the opportunity to use coding styles aligned with the accelerator we anticipate using, or to use a different aspect of oneAPI and leverage APIs (e.g., library functions) that are supported across multiple different architectures.

Why Data Parallelism?

Data parallelism is one of the most important forms of parallelism for a parallel programmer to exploit. In practice, with a good data-parallel programming model, one can achieve high degrees of *scaling*. Scaling refers to a desire to see performance increase (scale up) when compute resources increase (scale up).

Effective parallel programming is all about *scaling*. If we expand our computing resources by 100X, ideally, we hope our application will run a hundred times as fast. Of course, there are complications — chief of which is expressed by Amdahl's Law.

Think (Data) Parallel

Effective parallel programming starts with a clear vision, by the programmer, of where the parallelism is. That is followed by a strategy on how to exploit the parallelism in a manner that yields speed-up. Learning to be an effective parallel programmer rests squarely on developing a knack for finding the best way to break up a problem into a large number of independent tasks. Independence is critical, because interaction between tasks will limit scaling — interaction is effectively a serialization, and Amdahl's Law tells us that will limit us.

Data parallelism, in practice, tends to work very well for parallel programming. If we can figure out a kernel of computation that we want to apply on all of our vast data, we can see amazing speed-ups in our applications. The two key concepts in our favor here are:

- Independence: Thinking in terms of *kernels* encourages exactly the sort of independence we need in our strategy to achieve high scaling.
- Scale with problem size: problems that involve processing data will scale with the problem size when we have independence. This is wonderful – because tackling bigger data sets, or otherwise finding more Big Data, is not considered hard to do these days!

Intranode, not multinode, parallelism

DPC++ is focused on parallelism within a single node, often call intranode parallelism. A node, in this case, is defined as scope for the DPC++ host with its attached devices. The host may be a single, or multiple, CPUs for instance — a node is the shared memory environment of those one or more CPUs with attached accelerators (DPC++ devices). The topic of multinode parallelism (parallel programming often associated with MPI – which connects nodes together to operate in parallel), is not directly addressed by DPC++, SYCL, or OpenCL. Multinode, in our context, would be a collection of multiple hosts. Mechanisms to allow efficient multinode parallelism are present in DPC++, SYCL, and OpenCL. Multinode parallelism with DPC++, for instance, could use MPI to orchestra multiple nodes each running DPC++ in a fashion learned from this book. Multinode parallelism is beyond the scope of this book.

Other Accelerator Programming Models

The concept of accelerators in computing is not a new one, and over the decades they have been accompanied with a wide variety of programming models. Early floating-point processors were programmed separately, and even the introduction of float-point for x86 processors occurred as a coprocessor with a distinct set of instructions that were executed outside the processor but with memory accesses coordinated by the processor.

Having accelerator programming in the same high-level programming language has been a goal of many modern programming environments. The end of rising clock rate (circa 2006) has given rise to many new accelerators and interest in programming them more easily and effectively.

Rather than look at a historical view of accelerator models, we have chosen to briefly list ones with the most contemporary relevance and offer a few notes on how they relate to DPC++, and in some instances oneAPI.

oneAPI: An Intel-led initiative to provide a common support model for accelerators regardless of programming language. First public beta Q4 2019. http://software.intel.com/oneAPI

DPC++ (Data-Parallel C++): consists of C++ with SYCL and extensions, the subject of this book. DPC++ outfits C++ for data parallelism. DPC++ supports the execution of compute kernels. First public beta Q4 2019. http://software.intel.com/oneAPI

SYCL: also the subject of this book, a Khronos specification for extending C++ with a cross-platform abstraction layer. SYCL offers a higher-level programming interface for OpenCL devices. SYCL enables code for heterogeneous processors to be written in a "single-source" style using C++. The SYCL standards team enjoys widespread participation. SYCL supports the execution of compute kernels. First specification May 2015. https://www.khronos.org/sycl/

OpenCL (Open Computing Language): a Khronos specification for a framework for writing programs that execute across heterogeneous platforms including CPUs, GPUs, DSPs, FPGAs, etc. Most vendors participate in the OpenCL standards body. Such broad support for OpenCL has resulted in the broadest list of accelerator devices being supported of any accelerator programming system. OpenCL focuses on support for the execution of compute kernels. First specification in August 2009. https://www.khronos.org/opencl/

CUDA (Compute Unified Device Architecture): A programming model for "CUDA enabled" Nvidia GPUs. The CUDA platform is a software layer to connect with the virtual instruction set of a large range of Nvidia GPUs in order to execute compute kernels. Initial release in June 2007. https://developer.nvidia.com/cuda-zone

TBB (Threading Building Blocks): An Intel-led open-source project that adds support for parallelism to C++ through template libraries. The most popular abstract programming model for parallel programming in C++, and is supported on all modern systems. Does not itself implement SIMD or vectorization support — rather than duplicate such efforts it relies on compiler extensions (OpenMP, OpenCL, SYCL, DPC++) for that. Support for heterogeneous systems is achieved through the ability to offload functionality within a TBB flow graph using other models such as OpenCL, CUDA, SYCL, or DPC++. Initial release in mid-2006. https://github.com/intel/tbb

OpenMP (Open Multiprocessing directives): A specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran, C, and C++ programs. Vendors and HPC users participate in

the OpenMP standards body. OpenMP is the most widely adopted standard for systems, has been implemented by all major compiler vendors, and is supported on all modern systems. First specification in late 1997. https://www.openmp.org/

OpenACC (Open Accelerator directives): A specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C program — developed as an offshoot of OpenMP by Cray, CAPS, Nvidia and PGI to address the needs of Nvidia GPUs. OpenACC adopted a more compiler-friendly descriptive specification, versus the more user-friendly prescriptive specifications of OpenMP. Initial specification November 2011. Cray announced, in 2019, plans to phase out OpenACC support in favor of the unified OpenMP support for heterogeneous systems. https://www.openacc.org/

C++ AMP (C++ Accelerated Massive Parallelism) is a library implemented on DirectX 11 and an open specification from Microsoft for implementing data parallelism using GPUs directly in C++. C++ AMP provides a single-source way to write programs that compile and execute kernels on GPU data-parallel hardware. First specification August 2012. https://docs.microsoft.com/cpp/parallel/amp/cpp-amp-overview

AMD HCC (Heterogeneous Compute Compiler): single source C++ with code generation to both x86 processors and *HSAIL* (heterogeneous offload to AMD APUs and discrete GPUs via *HSA* enabled runtimes and drivers). *HSAIL* is short for HSA Intermediate Language. *HSA* is short for Heterogeneous System Architecture. HSA is a set of specifications, initially created by AMD, that allows for the integration of central processing units and graphics processors on the same bus, with shared memory and tasks. First specification March 2015. https://gpuopen.com/compute-product/hcc-heterogeneous-compute-compiler/

Kokkos (C++ Performance Portability Programming EcoSystem): a project from Sandia National Labs; Kokkos provides abstractions for both parallel execution of code and data management with the stated objective to implement a programming model in C++ for writing performance portable applications targeting all major HPC platforms. First released in 2008 as a collection of the handful of sparse and dense kernels as part of the Trilinos Project at Sandia, it has developed notoriety in its innovations for heterogeneous support over the years since. https://github.com/kokkos

RAJA: a collection of C++ software abstractions, being developed at Lawrence Livermore National Laboratory (LLNL). RAJA makes heavy use of C++ templates and is best when C++ lambda expressions are used to express the computational kernels. CHAI is a closely related library that complements RAJA by providing pointer abstractions that hide run-time data copies that move kernel data to execution memory spaces as needed. Umpire is a closely related library that provides a simple, unified interface to access

capabilities for memory resources. Described in 2014, first release 2016. https://github.com/LLNL/RAJA

MPI (Message Passing Interface): widely used and supported standard for portable message-passing on parallel computing architectures. Many vendors and major HPC users participate in the MPI standards body. MPI defines the syntax and semantics of a core of library routines to enable writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. First draft specification in 1992, 1.0 released in 1994. https://www.mpi-forum.org/

Evolution of SYCL (thus far)

What we know of as SYCL today started as the "High Level Model for OpenCL" (HLM for OpenCL) in 2010 when some members of the OpenCL working group wanted to develop a higher-level interface to target OpenCL devices. A sub-group was formed to study what could be done to provide a “high-level programming model, unifying host and device execution environments through language syntax for increased usability and broader optimization opportunities.”

Early SYCL implementations relied on utilizing OpenCL to reach accelerator devices. However, SYCL has proven to be a more general heterogeneous framework able to target other systems. For example, the ComputeCpp and hipSYCL implementations each target Nvidia GPUs by outputting to CUDA, and hipSYCL targets AMD GPUs by outputting to AMD HIP on the ROCm platform.

March 2015 – SYCL 1.2

The first release of a specification from The Khronos Group was called SYCL 1.2 in March 2015. SYCL continues as a complementary effort alongside the ongoing evolution of the OpenCL language. SYCL 1.2 built on the features of C++11.

SYCL generated a lot of interest and support, largely among tool and framework creators. For instance, Tensorflow targeting of OpenCL has been implemented by using SYCL. Multiple efforts to develop compilers and tools for SYCL arose based on SYCL 1.2.

However, for application writers, SYCL really starts with 1.2.1r5.

April 2019 – SYCL 1.2.1r5

On April 18, 2019, SYCL 1.2.1 revision 5 became the latest SYCL based on OpenCL 1.2, and (despite the slight change in numbering) was a *major* update representing four years of efforts by the SYCL group. Khronos created an open-source project, hosted on github, to support Parallel STL on top of SYCL, running on OpenCL devices. The SYCL conformance test suite (CTS), an open-source project on github, was also introduced to help vendors test compliance. The announcements from Khronos noted that *SYCL brings the power of single-source modern C++ to the OpenCL and SPIR world, it also prepares the convergence with other standards such as ISO C++, and Khronos' Vulkan, OpenVX, and NNEF.*

SYCL 1.2.1r6 and beyond

Revisions to the 1.2.1 standard are needed to clarify or adjust the standard. In particular, implementers of the standard are finding opportunities for clarifications, refinements, and corrections. While these do not materially impact the intent of SYCL, they will impact some topics within this book. We have endeavored to be current with the specification at time of publication — but changes are inevitable. We encourage the reader to visit our web site (https://tinyurl.com/book-dpcpp) for errata notes that will include notes when an adjustment is needed because of a standard revision.

SYCL Provisional 2.2

An early and provisional SYCL 2.2 was previewed in May 2016 targeting C++14 and OpenCL 2.2. As related on Wikipedia, the SYCL committee decided to not finalize this version and is working on a more flexible SYCL specification to address the increasing diversity of current accelerators.

The SYCL committee actively engages and examines proposals for future SYCL features. Intel, as a member of the SYCL standard committee, is open about experiences extending SYCL with DPC++ features. It is logical to hope that those experiences will lead to further refinements to SYCL in the future.

Summary

This chapter provided a base rationale for DPC++ and SYCL. Next, we will consider accelerators as devices. Such 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). We will devote Chapters 2, 3, and 4, respectively, to expanding on facets of these three needs.

FOR THIS BOOK PREVIEW (CHAPTERS 1-4):
ERRATA, NOTES, DOWNLOADS, FEEDBACK, ETC.

Please check our "preview book" website for information including errata, updates, and downloads (includes all sample code): https://tinyurl.com/book-dpcpp

We will also list some additional resources as they become available.

Your feedback is welcome. You can email James Reinders at dpc++@jamesreinders.com with any suggestions, encouragement, criticism, or questions that you may have. James will be sure to share any feedback that you send with all the authors.

Of course – watch for the full book, by mid-2020, available from Apress (no charge for PDF for the completed book, print copies will be available too).

https://tinyurl.com/book-dpcpp

Where Code Executes

SYCL provides a heterogeneous programming framework in which code executes on a mixture of a host CPU and one or more devices. In such a framework, the mechanisms used to select the destinations for code execution are a fundamental part of the solution. This chapter describes where code can execute, introduces when it will execute, and describes the mechanisms used to control the locations of execution. Chapter 3 will describe how to manage data so it arrives where we are executing our code, and then Chapter 4 returns to the code itself and discusses the writing of kernels. These chapters combine to teach the fundamentals of writing a SYCL application, because before writing data parallel code in Chapter 4, we first need to understand the assignment of code to targets (this chapter) and the management of data (Chapter 3).

DPC++ AND SYCL IN CHAPTER 2

Everything discussed in this chapter is SYCL and is therefore fully supported by DPC++.

Single source

SYCL programs can be single source, meaning that the same file contains both the code that defines the compute kernels to be executed on SYCL devices and also the host code that orchestrates execution of those compute kernels.

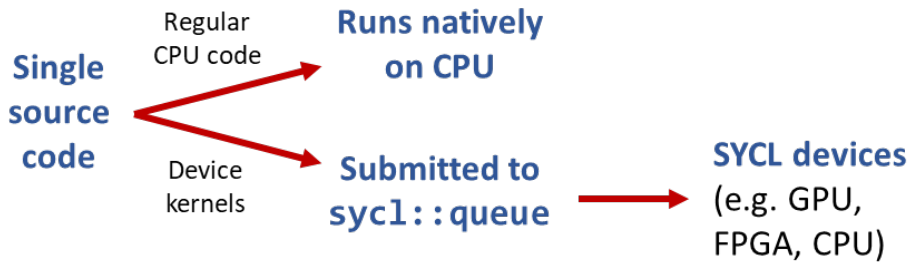


Figure 2-1 shows these two code paths graphically, and Figure 2-2 provides an example SYCL application with the host and device code regions marked.

Combining both device and host code into a single source file can make it easier to understand and maintain a heterogeneous compute application. The combination also provides improved language type safety and can lead to additional compiler optimizations.

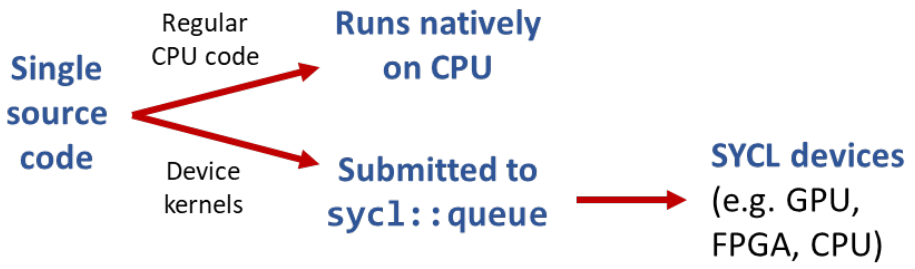


Figure 2-1: SYCL single source code contains both host code that runs natively on the host CPU, and device code that is executed on SYCL devices.

```
constexpr int num=16;

#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

int main() {
  int data[num];

  {
    // Create queue on implementation-chosen default device
    queue myQueue;

    // Create buffer using host allocated "data" array
    buffer<int, 1> buf { data, range<1> { num } };

      myQueue.submit([&](handler& h) {
        auto writeResult =
           buf.get_access<access::mode::discard_write>(h);
        h.parallel_for<class simple_test>(range<1> { num },
           [=](id<1> idx) {
      ------------------------------------------------------------
          writeResult[idx] = idx[0];
      ------------------------------------------------------------
        });
      });
  } // Buffer goes out of scope.  Blocks until kernel output
    // available in "data" array.

for (int i = 0; i < num; i++)
  std::cout << "data[" << i << "] = " << data[i] << "\n";

  return 0;
}
```

Host code
Device code
Host code

Figure 2-2: A simple SYCL program.

Host code

SYCL applications contain C++ host code which is executed by the CPU(s) on which the operating system has launched the SYCL application. Host code is the backbone of a SYCL application in that it defines and controls the execution of compute on available devices. It is also the interface through which users define the data and dependencies that should be managed by the SYCL runtime.

SYCL host code is standard C++11 and SYCL-specific constructs and classes are designed to be implementable as a C++ library. This makes it easier to reason about what is allowed within host code (anything that is allowed in C++), and can simplify integration with build systems.

The host code in a SYCL application orchestrates data movement and compute offload to devices, but can also perform compute-intensive work itself, and can use libraries like any C++ application.

Device code

Devices correspond to accelerators or processors that are conceptually independent from the CPU that is executing SYCL host code (the host processor). A SYCL implementation may expose the host processor also as a device, as described later in this chapter, but the host processor and devices should be thought of as logically independent from each other. The host processor runs native C++ code, while devices run device code.

SYCL queues are the mechanism through which host code submits work to a device for future execution. There are three critical properties of device code to understand:

1. **It executes asynchronously from the host code.** The host program submits device code for future execution on a device, and the SYCL runtime tracks and initiates that work only when all dependencies for execution are satisfied (more on this in Chapter 3). The host program execution carries on before the submitted work is initiated on a device, providing the property that execution on devices is asynchronous to host program execution, unless the developer explicitly ties the two together.

2. **There are restrictions on device code** to make it possible to compile and achieve performance on accelerator devices. For example, C++ exceptions and run time type information (RTTI) are not supported within device code, because they would lead to performance degradation on many modern accelerators. The small set of device code restrictions are covered in detail in Chapter 10.

3. **Some functions and queries defined by SYCL are available only within device code**, because they only make sense in that context. For example, work item identifier queries that allow an executing instance of device code to query its position in a larger data parallel range (described in Chapter 4).

Choosing a device on which to execute

To explore the mechanisms that control where device code executes, we consider five use cases:

Method#1:	Running device code "somewhere", with the developer not caring which device it ends up being. This is often the first step in application development, because it requires the least code.
Method#2:	Explicitly running device code on the host device, which is typical used for debugging and is guaranteed to be always available on any system.
Method#3:	Dispatching device code to a GPU or another accelerator device.
Method#4:	Dispatching device code to a heterogeneous set of devices, such as a GPU and an FPGA.
Method#5:	Selecting specific devices from a more general class of devices, such as a specific type of FPGA from a collection of available FPGA devices.

Developers will typically debug their code as much as possible with Method#2, and only move to Methods 3-5 when code is proven as much as it can be with Method #2.

Method#1: Just run on a device (don't care what type)

By design, SYCL makes it easy to run device code on a device chosen by the runtime. This automatic selection of an available accelerator is designed to make it easy to start writing and running device code, where the developer doesn't yet care about what device is chosen. This device selection does not take into account the code to be executed, so should be considered an arbitrary choice.

To talk about choice of a device, even one that the implementation has selected for the user, it's important to first introduce the primary mechanism through which the program interacts with a device: the queue.

Queues

A `sycl::queue` is an abstraction to which work is submitted for execution on a single device. This work is most commonly the invocation of a data parallel computation, although other commands are also available such as for manual control of data motion, where the user wants more control than the automatic data movement provided by the SYCL runtime. Work submitted to a `sycl::queue` can execute once certain prerequisites tracked by the SYCL runtime are met, such as availability of input data. These prerequisites are described in more detail in Chapters 3 and 8.

A `sycl::queue` is bound to a single `sycl::device`, and that binding occurs on construction of the queue. It is important to understand that work submitted to a queue is executed on the single device to which that queue is bound. Queues do not map to collections of devices, for example, where there would be ambiguity on which device would perform the enqueued work. Similarly, a queue cannot spread the work submitted to it across multiple devices. Instead, there is an unambiguous and one-to-one mapping between a queue, and the device on which work submitted to that queue will execute, as shown in Figure 2-3.

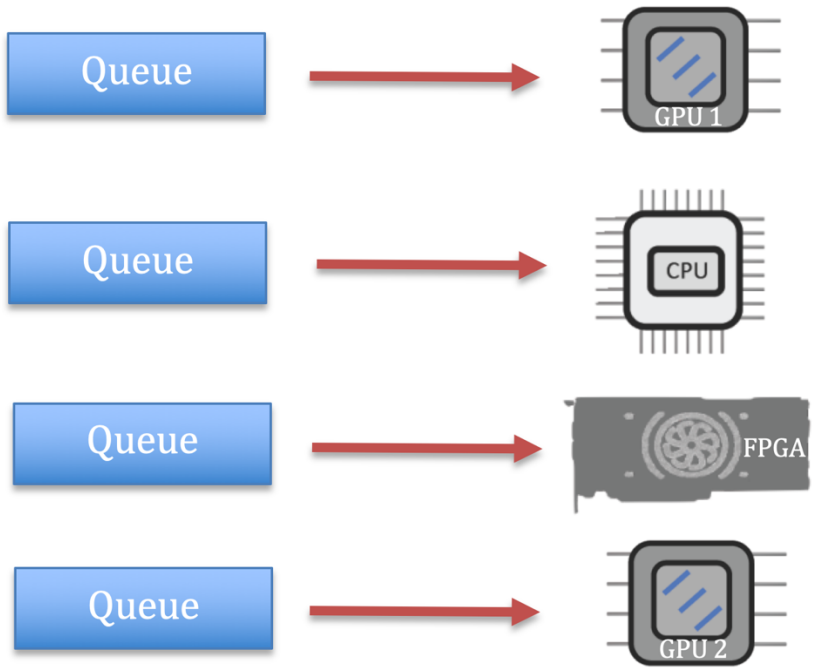


Figure 2-3: Queues are bound one-to-one to a specific device. Work submitted to the queue executes on that device. SYCL developers construct queues in their host code, so choose how many are needed for an application.

Multiple queues may be created in a SYCL application, as desired for application architecture or programming style. For example, multiple queues may be created to each bind with a different device, or to be used by different threads in a host program. Any number of queues can be bound to a single device, such as a GPU, and submissions to different queues bound to the same device will result in the combined work being performed on that single device. An example of such a binding is shown in Figure 2-4. Conversely, a queue cannot be bound to more than one device. Queues bind only to a single device so that there is no ambiguity on where work submitted to a queue will execute. If a user wants a queue that will load balance work across multiple devices, for example, then they can create that abstraction in their code.

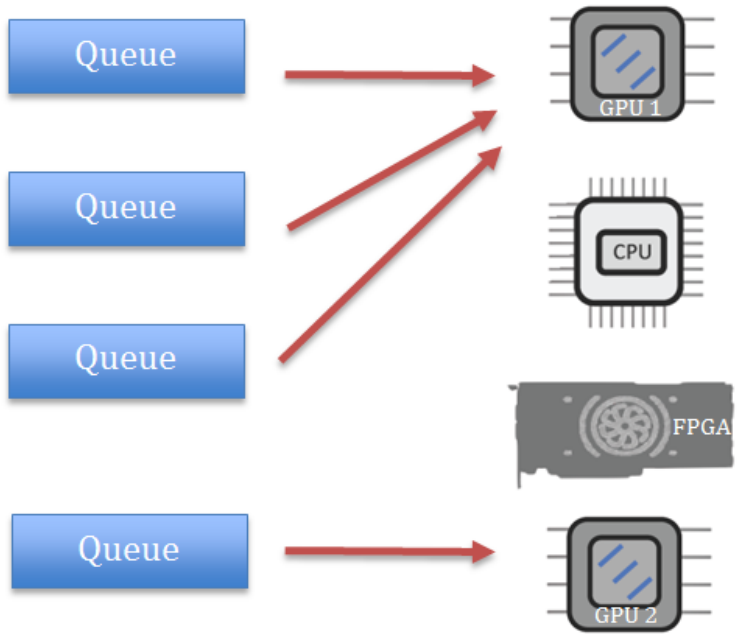


Figure 2-4: Multiple queues can be bound to a single device, and the combination of work submitted to such queues is combined onto the device. The converse is not allowed, though, and a single queue cannot be bound to more than one device, because it would lead to ambiguity in where device code will execute.

Because a queue is bound one-to-one with a device, queue construction is the most common location in SYCL code to choose the device on which the queue submissions will execute. Selection of the device is achieved through a device selector abstraction and associated `sycl::device_selector` class.

Binding a queue to a device, when any device will do

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

int main() {
  // Create queue on whatever default device SYCL
  // implementation chooses. Implicit use of the default_selector().
  queue myQueue;

  std::cout << "Selected device: " <<
    myQueue.get_device().get_info<info::device::name>() << "\n";

  return 0;
}

Possible output:
Selected device: SYCL host device
```

Figure 2-5 provides an example code listing, where the device with which a queue will be bound is not specified. The trivial queue constructor, that doesn't take any arguments (as in Figure 2-5), simply chooses some available device behind the scenes. SYCL guarantees that at least some device will be available, which might be the SYCL host device that is a device abstraction of the processor(s) on which the host code is executing.


```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

int main() {
  // Create queue on whatever default device SYCL
  // implementation chooses. Implicit use of the default_selector().
  queue myQueue;

  std::cout << "Selected device: " <<
    myQueue.get_device().get_info<info::device::name>() << "\n";

  return 0;
}

Possible output:
Selected device: SYCL host device
```


Figure 2-5: Implicit default device selector through default construction of a queue.

Using the default queue constructor is a simple way to start application development, and to get device code running quickly. Additional control over selection of

the device bound to a queue can be added as it becomes relevant for application development.

Method#2: Using the host device for development and debugging

The host device can be thought of as enabling the host CPU to act as if it was an independent device, allowing device code (with the associated built-ins) to be executed regardless of the accelerators available in a system. It also has the fundamental property that it is always available to a SYCL application. The host device therefore provides a guarantee that device code can always be run within an application (no dependency on accelerator hardware), and has three primary uses:

1. **Development of device code** on less capable systems that don't have any accelerators. One common use is development and testing of SYCL device code on a local system, before deploying to an HPC cluster for performance testing and optimization.
2. **Debugging of device code** with non-accelerator tooling. Accelerators are often exposed through lower level APIs that may not have debug tooling as advanced as is available for host CPUs. With this in mind, the host device is expected to support debugging using standard tools familiar to CPU developers.
3. **Backup** if no other devices are available, to guarantee that device code can be executed functionally. The host device implementation may not have performance as a primary goal, so should be considered as a functional backup to ensure that device code can always execute in any SYCL application, but not necessarily a path to performance.

The host device is functionally like a hardware accelerator device in that a queue can bind to it, and it can execute device code. Figure 2-6 shows how the host device is a peer to other accelerators that might be available in a system. It can execute device code, just like a CPU, GPU or FPGA, and can have one or more queues constructed that bind to it.

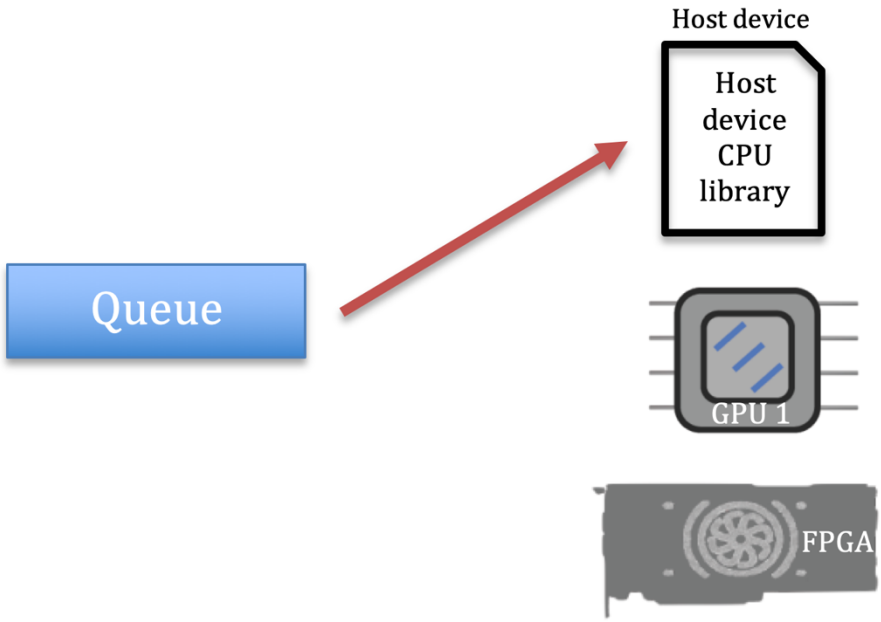


Figure 2-6: The host device is logically like any accelerator device (e.g. CPU, GPU, FPGA) in that it can execute device code, and can have a queue bound to it that is used to enqueue device code kernels. The host device is guaranteed to be available in every SYCL application, regardless of what other hardware devices are present.

A SYCL application can choose to create a queue that is bound to the host device by explicitly passing the `sycl::host_selector` class to a queue constructor, as shown in Figure 2-7.

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

int main() {
  // Create queue on whatever default device SYCL
  // implementation chooses. Explicit default_selector.
  queue myQueue( host_selector{} );

  std::cout << "Selected device: " <<
    myQueue.get_device().get_info<info::device::name>() << "\n";

  return 0;
}

Possible Output:
Device: SYCL host device
```

Figure 2-7: Selecting the host device using the `sycl::host_selector` class.

Even when not explicitly requested (using `sycl::host_selector` for example), the host device may be selected by the default device selector, as occurred in the output in Figure 2-5Figure 2-5.

Figure 2-7 passes the `sycl::host_selector` class to the queue constructor, to define the expected target binding of the queue. `sycl::host_selector` is an example of a SYCL device selector class, of which a few variants are defined to make it easy to target a device of a specific type.

Method#3: Using a GPU (or other accelerator)

GPUs are highlighted in this example use case, but any class of accelerator applies equally. To make it easy to target common classes of accelerators, SYCL groups devices into several broad categories, and provides built-in selector classes for them. To choose from a broad category of device type such as "any GPU available in the system," the corresponding SYCL code is very brief, as described in this section.

Device types

There are two main categories of device to which a queue can be bound:

1. The host device, which has already been described.

2. Accelerator devices such as a GPU, an FPGA, or a CPU device, which are used to accelerate workloads in real applications.

Accelerator devices

There are a few categories of accelerator devices defined by SYCL:

1. CPU devices
2. GPUs
3. Accelerators, which capture devices that don't identify as either a CPU device or a GPU. Today this includes FPGAs and DSPs.

These categories are easy to bind to a queue using SYCL's built-in selector classes, which can be passed to queue (and some other class) constructors.

Device selectors

Classes in SYCL that must be bound to a specific device, such as a queue, have constructors that can accept a class derived from `sycl::device_selector`. For example, the queue constructor is:

```
queue( const device_selector &deviceSelector,
       const property_list &propList = {});
```

SYCL defines five built-in selectors for the broad classes of common devices:

default_selector	Any device of the implementation's choosing
host_selector	Select the host device (always available)
cpu_selector	Select device that identifies itself as a CPU device in device queries
gpu_selector	Select device that identifies itself as a GPU in device queries
accelerator_selector	Select device that identifies itself as an "accelerator", which includes FPGAs

One additional selector is shipped as part of DPC++ (not available in SYCL), and is available by including the header "`CL/sycl/intel/fpga_extensions.hpp`":

intel::fpga_selector	Select device that identifies itself as an FPGA

A queue can be constructed using one of the built-in selectors, such as:

```
queue myQueue( cpu_selector{} );
```

Figure 2-8 shows a complete example using the `sycl::cpu_selector`, and Figure 2-9 shows the corresponding binding of a queue with an available CPU device.

Figure 2-10 shows an example using a variety of built-in selector classes, and also demonstrates use of device selectors with another class (`sycl::device`) that accepts a `sycl::device_selector` on construction.

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

int main() {
  // Create queue on whatever default device SYCL
  // implementation chooses. Explicit default_selector.
  queue myQueue( cpu_selector{} );

  std::cout << "Selected device: " <<
    myQueue.get_device().get_info<info::device::name>() << "\n";
  std::cout << " -> Device vendor: " <<
    myQueue.get_device().get_info<info::device::vendor>() << "\n";

  return 0;
}

Possible Output:
Selected device: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
 -> Device vendor: Intel(R) Corporation
```

Figure 2-8: CPU device selector example.

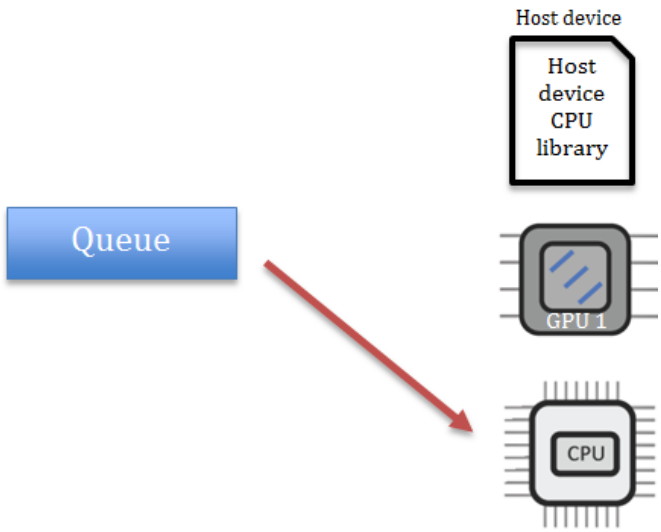


Figure 2-9: CPU device selector example. The queue is bound to a CPU device available to the SYCL application.

```
#include <CL/sycl.hpp>
#include <CL/sycl/intel/fpga_extensions.hpp> // fpga_selector
#include <iostream>
#include <string>

using namespace cl::sycl;

void output_dev_info( const device& dev, std::string selector_name) {
  std::cout << selector_name << ": Selected device: " <<
    dev.get_info<info::device::name>() << "\n";
  std::cout << "                    -> Device vendor: " <<
    dev.get_info<info::device::vendor>() << "\n";
}

int main() {

  output_dev_info( device( default_selector{}), "default_selector" );
  output_dev_info( device( host_selector{}), "host_selector" );
  output_dev_info( device( cpu_selector{}), "cpu_selector" );
  output_dev_info( device( gpu_selector{}), "gpu_selector" );
  output_dev_info( device( accelerator_selector{}),
                                          "accelerator_selector" );
  output_dev_info( device( intel::fpga_selector{}), "fpga_selector" );

  return 0;
}

Possible Output:
default_selector: Selected device: Intel(R) Gen9 HD Graphics NEO
                  -> Device vendor: Intel(R) Corporation
host_selector: Selected device: SYCL host device
                  -> Device vendor:
cpu_selector: Selected device: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
                  -> Device vendor: Intel(R) Corporation
gpu_selector: Selected device: Intel(R) Gen9 HD Graphics NEO
                  -> Device vendor: Intel(R) Corporation
accelerator_selector: Selected device: Intel(R) FPGA Emulation Device
(preview)
                  -> Device vendor: Intel(R) Corporation
fpga_selector: Selected device: pac_a10 : PAC Arria 10 Platform
(pac_a10_f300000)
                  -> Device vendor: Intel Corp
```

Figure 2-10: Example device identification output from various classes of device selectors, and demonstration that device selectors can be used for construction of more than just a queue (in this case construction of a device class instance).

When device selection fails

If a `sycl::gpu_selector` is used when creating an object such as a queue, and if there are no GPU devices available to the SYCL runtime, then the selector throws a `sycl::runtime_error` exception. This is true for all device selector classes, in that if no device of the required class is available, then a `sycl::runtime_error` exception is

thrown. It is reasonable for complex applications to catch that error, and instead acquire a less desirable (for the application/algorithm) device class as an alternative.

Method#4: Using multiple devices

As shown in Figures 2-3 and 2-4, multiple queues can be constructed in a SYCL application. These queues can be bound to a single device (the sum of work to the queues is funneled into the single device), to multiple devices, or to some combination of these. Figure 2-11 provides an example code listing that creates one queue bound to a GPU, and another queue bound to an FPGA. The corresponding mapping is shown graphically in Figure 2-12.

```
#include <CL/sycl.hpp>
#include <CL/sycl/intel/fpga_extensions.hpp> // fpga_selector
#include <iostream>

using namespace cl::sycl;

int main() {
  queue my_gpu_queue( gpu_selector{} );
  queue my_fpga_queue( intel::fpga_selector{} );


  std::cout << "Selected device 1: " <<
    my_gpu_queue.get_device().get_info<info::device::name>() << "\n";

  std::cout << "Selected device 2: " <<
    my_fpga_queue.get_device().get_info<info::device::name>() << "\n";

  return 0;
}

Possible Output:
Selected device 1: Intel(R) Gen9 HD Graphics NEO
Selected device 2: pac_a10 : PAC Arria 10 Platform
```

Figure 2-11: Creating queues to both GPU and FPGA devices.

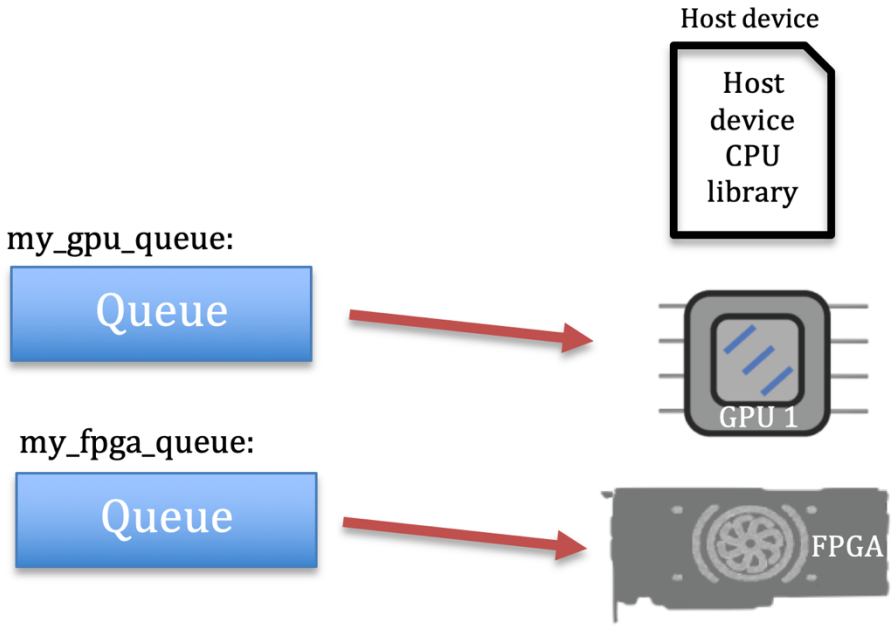


Figure 2-12: GPU and FPGA device selector example. One queue is bound to a GPU available to the SYCL application, and another queue is bound to an available FPGA.

Method#5: Custom (very specific) device selection

Writing a custom selector

The SYCL built-in device selectors are intended to get code up and running quickly. Real applications typically require specialized selection of a device, such as picking a desired GPU from a set of GPU models available in a system. The device selection mechanism is easily extended to arbitrarily complex selection logic, as required for an application.

`device_selector` base class

All device selectors derive from the abstract `sycl::device_selector` base class, and define the function call operator in the derived class:

```
virtual int operator()(const device &dev) const {
  /* User logic */
}
```

Defining this operator in a class that derives from `sycl::device_selector` is all that is required to define any complexity of device selection logic, resulting from three properties:

1. The function call operator is automatically called on each device that the SYCL runtime finds as accessible to the application, including the host device.
2. The operator returns an integer score. The highest score across all available devices defines the selected device.
3. A negative integer returned by the function call operator means that the device must not be returned by the device selector.

Mechanisms to score a device

Many approaches are possible to create an integer score corresponding to a specific device or style of device, such as:

1. Return positive value for a specific device class
2. String match on device name and/or device vendor strings
3. Anything you can imagine in code leading to an integer value, based on device or platform queries

For example, one possible approach to select an Intel Arria-family FPGA device is shown in Figure 2-13.

```
class my_selector : public device_selector {
  public:
    virtual int operator()(const device &dev) const {
      if (
          dev.get_info<info::device::name>().find("Arria")
              != std::string::npos &&
          dev.get_info<info::device::vendor>().find("Intel")
              != std::string::npos) {
        return 1;
      } else {
        return -1;
      }
    }
};
```

Figure 2-13: Custom selector for Intel Arria FPGA device.

The `select_device()` method provided by the `sycl::device_selector` base class executes the function call operator for each device available in the system, and returns the device that saw the highest non-negative function call operator return value. If multiple devices end up returning the same high score, then one of those devices will be

returned, but which isn't defined. If no devices cause the function call operator to return a non-negative score, then a `sycl::runtime_error` exception is thrown automatically by the `select_device()` function.

Three paths to device code execution on CPU

A potential source of confusion comes from the multiple mechanisms through which a CPU can have code executed on it, as summarized in Figure 2-14.

The first and most obvious path to CPU execution is host code, which is either part of the single source application (host code regions), or which is linked to and called from the host code such as a library function.

The other two available paths execute device code. The first CPU path for device code is through the host device, which was described earlier in this chapter. It is always available and is expected to execute the device code on the same CPU(s) that the host code is executing on.

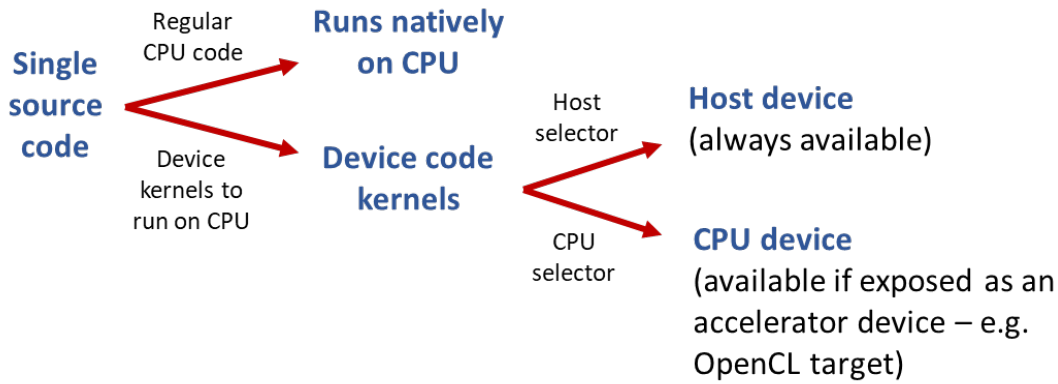


Figure 2-14: SYCL mechanisms to execute on a CPU.

A second path to execution of device code on a CPU is optionally exposed by an implementation, and is a CPU accelerator device that is optimized for performance. This philosophy of implementation is described by the SYCL specification, where the host device is intended to be debuggable with a native CPU debugger, while CPU devices may be built on implementations optimized for performance where a native CPU debugger isn't applicable.

Language constructs that generate work on a device

SYCL applications typically contain a combination of both host code and device code, with device code running on one or more of the devices described previously in this chapter, including possibly the host device. Device code is defined in the language by passing it to a small set of language constructs, which makes it easy to distinguish from host code.

The remainder of this chapter introduces some of the basic SYCL work dispatch constructs, with the goal to help readers understand and identify the division between device code in an application, and the host code that executes natively on the host processor.

Introduction to the SYCL Graph

A fundamental concept in the SYCL execution model is a graph of nodes. Each node in this graph contains an action to be performed on a device, with the most common action being a data parallel device kernel invocation. Figure 2-15 shows an example graph with four nodes, where each node can be considered to be a device kernel invocation for this example.

The nodes in Figure 2-15 have dependency edges defining when it is legal for a node's work to begin execution. The dependency edges are most commonly generated automatically from data dependencies, although there are mechanisms to manually add additional dependencies. Node B in the graph, for example, has a dependence edge from node A. This edge means that node A must complete execution, and most likely (depending on specifics of the dependency) make generated data available on the device where node B will execute, before node B's action is started. The SYCL runtime controls resolution of dependencies and triggering of node executions, completely asynchronously from the host program's execution. The graph of nodes defining an application will be referred to in this book as the SYCL Directed Acyclic Graph (DAG), and is covered in more detail in Chapter 3.

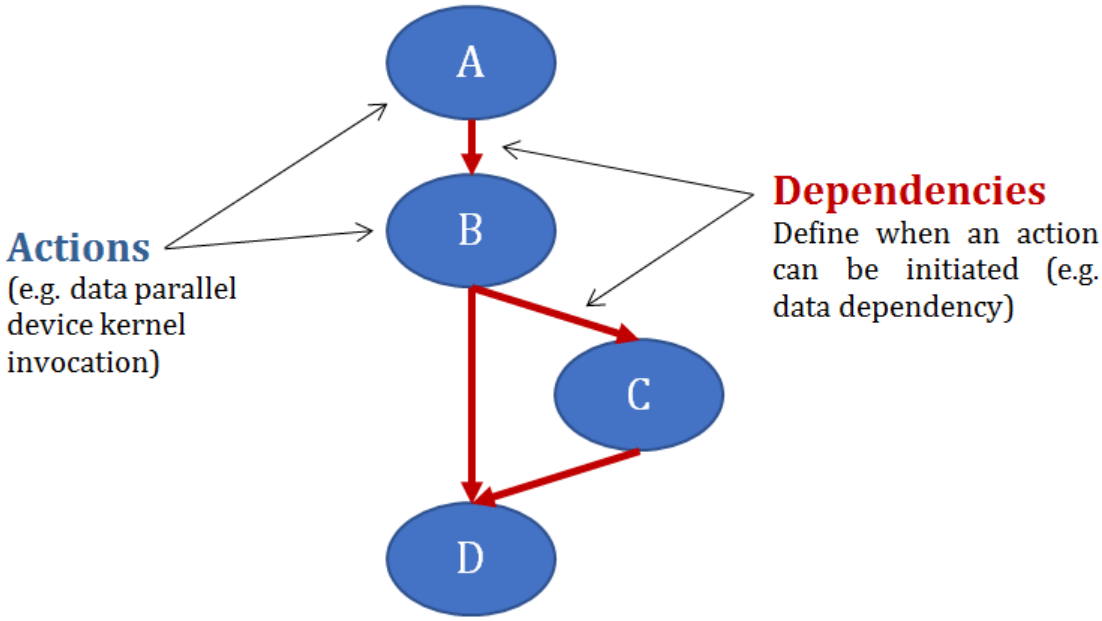


Figure 2-15: The SYCL Directed Acyclic Graph (DAG) defines the actions to perform (asynchronously from the host program) on one or more devices, and also the dependencies between actions that determine when it is safe for the SYCL runtime to initiate an action.

Where is the device code?

There are multiple mechanisms that can be used to define code that will be executed on a device, but a simple example with a lambda suffices to show how to identify such code. Even if the pattern in the example appears complex at first glance, the pattern remains the same across all device code definitions so quickly becomes second nature.

```
deviceQueue.submit([&](handler& h)
    {
       auto writeResult =
          buf.get_access<access::mode::discard_write>(h);

       h.parallel_for<class simple_test>
          (range<1> { num }, [=](id<1> idx) {

         writeResult[idx] = idx[0];

       });
     });
```

Command group

Device code

Figure 2-16: A simple submission of device code

The code passed as the final argument to the `parallel_for` member of the `sycl::handler` class, defined as a lambda in Figure 2-16, is the device code to be executed on a SYCL device. The `parallel_for` in this case is the critical construct that is used to distinguish device code from host code. `parallel_for` is one of a small set of device dispatch mechanisms, all members of the `handler` class, that define the code to be executed on a device.

Device dispatch and memory copy mechanisms

The code snip in Figure 2-16 contains a `parallel_for` construct, which defines work to be performed on a device through the C++ callable or `sycl::kernel` object passed as its final argument (a C++ lambda in this case). The `parallel_for` is within a command group submitted to a `queue`, and the `queue` defines the device on which the work is to be performed. Within the command group, there are two categories of code:

1. **At most one call to a handler class method** that either dispatches work to be performed on the device (device code forming a kernel), or that performs a manual data movement command such as `copy`.
2. **Host code** that typically sets up dependencies defining when it is safe for the SYCL runtime to start execution of the work defined in (1), such as creation of accessors to buffers (described in Chapter 3).

The handler class contains a small set of member functions that define the work to be performed when a DAG node's dependencies are met, and the SYCL runtime chooses to execute the node's actions. Figure 2-17 summarizes these methods.

Work Type	Handler class method	Summary
Device code execution	`single_task`	Execute a single instance of a device function.
	`parallel_for`	Multiple overloads are available to launch device code with different combinations of work sizes.
	`parallel_for_work_group`	Launch a kernel using hierarchical parallelism, described in Chapter 4.
Explicit memory operation	`copy`	Copy data between locations specified by accessor, pointer, and/or `shared_ptr`. The copy occurs as part of the DAG, including dependency tracking.
	`update_host`	Trigger update of host data backing of a buffer object.
	`fill`	Initialize data in a buffer to a specified value.

Figure 2-17: Handler class methods that invoke device code or perform explicit memory operations

Only a single handler method from Figure 2-17 may be called within a command group (it is an error to call more than one), and only a single command group can be submitted to a queue per `submit` call. The result of this is that a single operation from Figure 2-17 exists per SYCL DAG node, to be executed when the DAG node dependencies are met and the SYCL runtime determines that the node is safe to execute.

The notion of asynchronous execution in the future, when a DAG node's dependencies have been satisfied, forms the critical difference between the two classes of code listed previously that can exist within a command group, and also with host code.

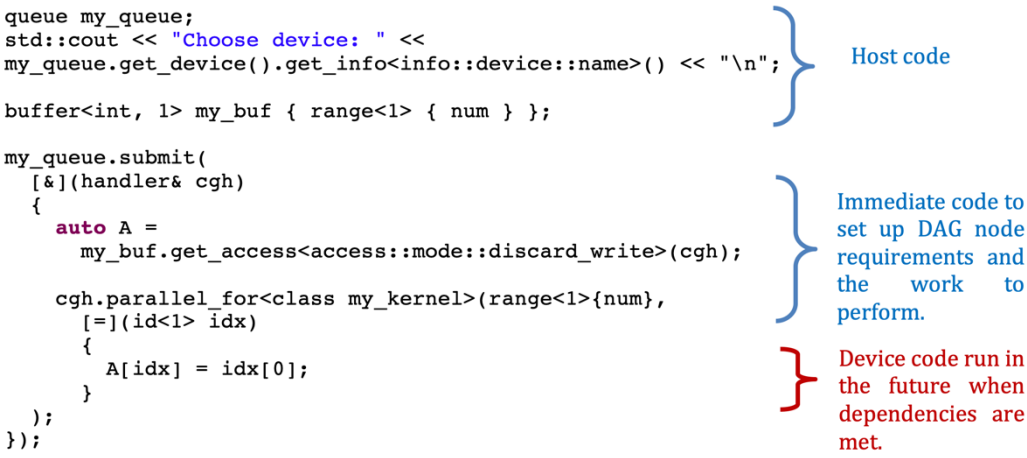


Figure 2-18: A simple submission of device code

There are three classes of code in Figure 2-18:

1. **Host code** that drives the SYCL application, including creating and managing data buffers, and enqueueing nodes into the SYCL DAG for asynchronous execution.

2. **Code within a command group that is not a handler method listed in Figure 2-17.** This code is run on the processor that the host code is executing on, and executes immediately (before the `submit` call returns). This code sets up the DAG node dependencies by creating accessors, for example, and is effectively CPU-executed setup code for the DAG node. Any arbitrary CPU can execute here, but best practice is to restrict it to code that configures the DAG node dependencies.

3. **Device dispatch method or an explicit memory operation.** One `handler` member listed in Figure 2-17 can be included in a command group, and it defines the work to be performed asynchronously in the future, when the DAG node requirements are met (set up by (2)). If a device dispatch function is called from the handler class, then the callable (or `sycl::kernel` object) passed to the appropriate argument will be compiled as device code, and executed on the device once the DAG dependencies are satisfied. The device on which execution occurs is simply the device associated with the queue to which the command group was submitted.

To understand when code in a SYCL application will run, the distinction is simple. Anything passed to a `sycl::handler` method listed in Figure 2-17 that initiates device code execution, or a `sycl::handler` explicit memory operation listed in Figure 2-17, will execute asynchronously in the future, when the DAG node dependencies have been met. All other code runs as part of the host program immediately, as expected in typical C++ code.

Fallback

Typically, a command group is executed on the command queue to which it has been submitted. However, there may be cases where the command group fails to be submitted to a queue (e.g. when the enqueue size is too large for the device's limits), or when a successfully submitted device operation such as a kernel execution is unable to begin execution (e.g. when a hardware device has failed). To handle such cases, it is possible to specify a fallback command queue for the command group to be executed on.

The fallback queue mechanism must be explicitly enabled by passing a secondary queue to a `submit` call. The fallback is of limited value, and it is instead recommended that you catch any exceptions thrown by a `submit` call (that doesn't have a fallback queue) and re-submit the work to a different queue through another `submit` call, if appropriate. Your exception catching code can record failure of the first submission through a debug mechanism of your choosing.

Figure 2-19 shows code that will fail if executed on Intel Processor Graphics, such as Intel HD Graphics 530, due to the requested size of the workgroup. The SYCL specification allows specifying a secondary queue as a parameter to the submit function, and this secondary queue (the host device in this case) is used if the command group fails to be enqueued to the primary queue (if an exception is thrown by the `submit` call).

```
#include<CL/sycl.hpp>
#include<iostream>
#define N 1024
#define M 32
using namespace cl::sycl;
int main(){

  queue gpuQueue( gpu_selector{} );
  queue hostQueue( host_selector{} );

  buffer<int,2> buf(range<2>(N,N));
  gpuQueue.submit([&](handler &cgh){
    auto bufacc = buf.get_access<access::mode::read_write>(cgh);
    cgh.parallel_for<class ndim>(nd_range<2>(range<2>(N,N),
      range<2>(M,M)), [=](nd_item<2> i){
       id<2> ind = i.get_global_id();
       bufacc[ind[0]][ind[1]] = ind[0]+ind[1];
    });
  },hostQueue); /** <<== Fallback Queue Specified **/
  auto bufacc1 = buf.get_access<access::mode::read>();
  for(inti=0;i<N;i++){
    for(int j = 0; j < N; j++){ if(bufacc1[i][j] != i+j){
       std::cout<<"Wrong result\n";
       return 1; }
    } }
  std::cout<<"Correct results\n";
  return 0; }
}
```

Figure 2-19: Simple fallback queue example.

Summary

In this chapter, we provided an overview of queues, selection of the device with which a queue will be associated, and how to create custom device selectors. We also introduced which code will execute on a device asynchronously when dependencies are met, versus the code that executed as part of the C++ application host code. Chapter 3 describes the SYCL DAG and associated concepts in more depth.

Please check our “preview book” website for information including errata, updates, and downloads (includes all sample code). https://tinyurl.com/book-dpcpp

Data Management

This chapter provides an overview of managing data, including controlling the ordering of data usage. This complements the prior chapter, which discussed controlling where code runs. This chapter helps us efficiently make our data appear where we have asked the code to run. This is important not only for correct execution of our application, but optimizing data movement will also help minimize execution time and power consumption. Since we covered how to control where code runs in Chapter 2 (on what device), once we discuss how to get data to our code (accessible from a particular device) in this chapter, we can get down to writing data parallel code in Chapter 4.

DPC++ AND SYCL IN CHAPTER 3

Unified Shared Memory (USM) and ordered queues are DPC++ extensions at the time this book is going to press and are not part of the SYCL 1.2.1 specification.

Everything else discussed in this chapter is SYCL, and is therefore fully supported by DPC++.

Introduction

Compute is nothing without data. The whole point of accelerating a computation is to produce an answer more quickly, so one of the most important aspects of data parallel computations is how they access data. Introducing accelerator devices into a machine further complicates the picture. In traditional single-socket CPU-based systems, we have a single memory. Accelerator devices often have their own attached memories that cannot be directly accessed from the host. Consequently, parallel programming models that support discrete devices must provide mechanisms to manage these multiples memories and move data between them.

In this chapter, we present an overview of the various mechanisms for data management in DPC++. We introduce Unified Shared Memory and the Buffer abstractions for data management and describe the relationship between kernel execution and data movement.

The Data Management Problem

Historically, one of the advantages of shared memory models for parallel programming is that they provide programmers a single, shared, view of memory. Having this single view of memory simplifies life for the programmer, as he or she is not required to do anything special to access memory from parallel tasks (aside from proper synchronization to avoid data races). While some types of accelerator devices (integrated GPUs, for example) share memory with a host CPU, many discrete accelerators often have their own local memories separate from that of the CPU as seen in Figure 3-1.

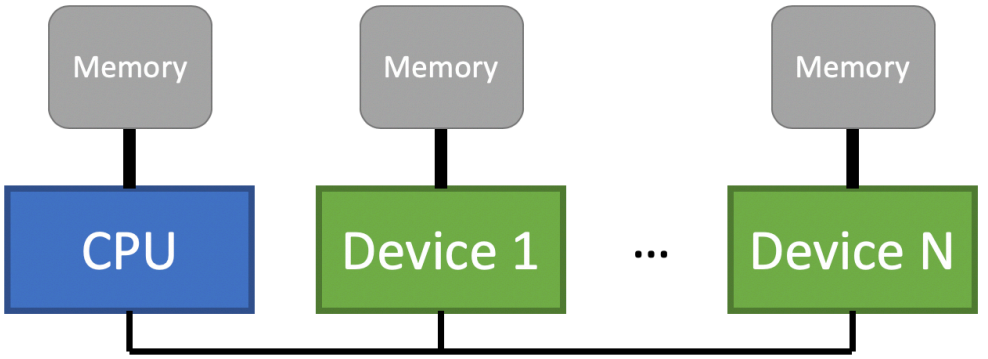


Figure 3-1: Multiple Discrete Memories

Device Local vs. Device Remote

Parallel programs running on a device prefer to read and write data to memory attached directly to the device. We refer to accesses to a directly attached memory as *local* accesses. Accesses to another device's memory are *remote* accesses. Remote accesses tend to be slower than local accesses because they must travel over data links with lower bandwidth and higher latency. This means it is advantageous to co-locate both a computation and the data it will use. However, the programmer must somehow ensure that data is copied or migrated between different memories in order to move it closer to where computation occurs.

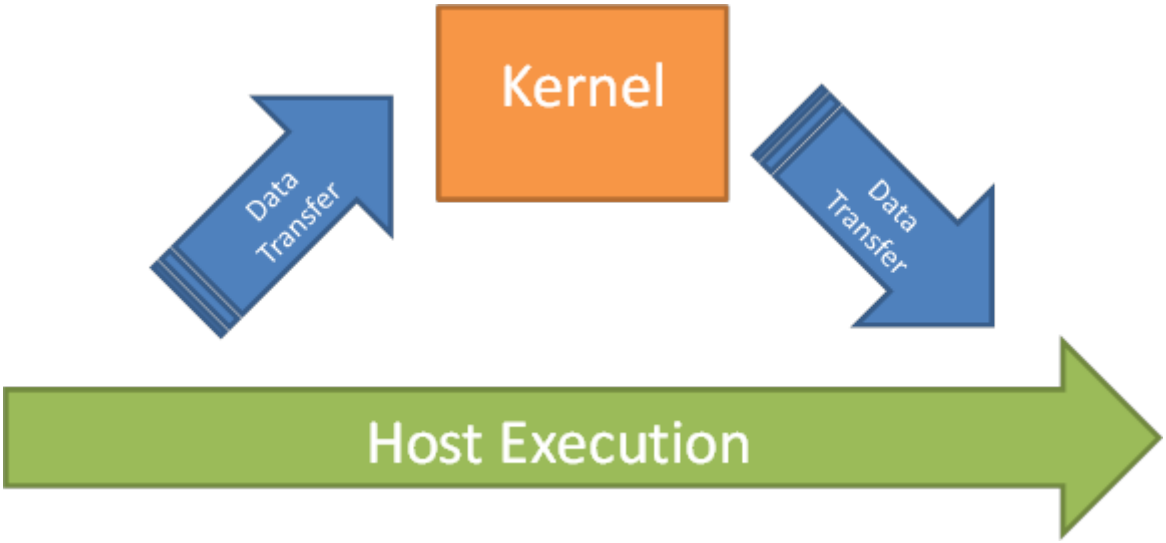


Figure 3-2: Data Movement and Kernel Execution

Managing Multiple Memories

Managing multiple memories can be accomplished, broadly, in two ways: *explicitly* by the programmer or *implicitly* by the runtime. Each method has its advantages and drawbacks, and programmers may choose one or the other depending on circumstances or personal preference.

Explicit data movement

One option for managing multiple memories is for the programmer to explicitly copy data between different memories. Figure 3-2 shows a system with a discrete GPU where the programmer must first copy any data that the GPU kernel will require from the host memory to GPU memory. After the kernel computes new results, the programmer must also copy these results back to the CPU before the host program can use that data.

The primary advantage of explicit data movement is that the programmer has full control over when data is transferred between different memories. This is important because overlapping computation with data transfer can be essential to obtaining the best performance on some hardware.

The drawback of explicit data movement is that specifying all data movements can be tedious and error prone. Transferring an incorrect amount of data or not ensuring that all data has been transferred before a kernel begins computing can lead to incorrect

results. These drawbacks create a barrier to entry when prototyping or evaluating the computational power of a particular accelerator. Getting all of the data movement correct up front can be a time-consuming task.

Implicit data movement

The alternative to programmer-controlled explicit data movement is implicit data movement controlled by a parallel runtime or driver. In this case, instead of the programmer inserting explicit copies between different memories, the parallel runtime is responsible for ensuring that data is transferred to the appropriate memory before it is used.

The advantage of implicit data movement is that it requires less effort on the programmer's part to get an application to take advantage of device local memory. All the heavy lifting is done by the runtime. This also reduces the opportunity to introduce errors into the program since the runtime will automatically identify both when data transfers must be performed and how much data must be transferred.

The drawback of implicit data movement is that the programmer has less or no control over the behavior of the runtime's implicit mechanisms. The runtime will provide functional correctness but may not move data in an optimal fashion that ensures maximal overlap of computation with data transfer, and this could have a negative impact on program performance.

Selecting the right strategy: explicit or implicit

Picking the best strategy for a program can depend on many different factors. Different strategies might be appropriate for different phases of program development. A programmer could even decide that the best solution is to mix and match the explicit and implicit methods for different pieces of the program, as they are not mutually exclusive. A programmer might choose to begin using implicit data movement to simplify porting an application to a new device. As we begin tuning the application for performance, we might start replacing implicit data movement with explicit in performance-critical parts of the code. Future chapters will cover how data transfers can be overlapped with computation in order to optimize performance.

USM, Buffers, and Images

DPC++ provides three abstractions for managing memory: Unified Shared Memory (USM), buffers, and images. USM is a pointer-based approach that should be familiar to C/C++ programmers. One advantage of USM is easier integration with existing C++ routines that operate on pointers as rewriting code to replace pointers with buffers can be time-consuming. Buffers, as represented by the `buffer` template class, describe one-, two-, or three-dimensional arrays. Buffers provide an abstract view of memory that can be accessed on either the host or a device. Buffers are not directly accessed by the programmer and are instead accessed through `accessor` objects. Images act as a special type of buffer that provides extra functionality specific to image processing. This functionality includes support for special image formats, reading images using sampler objects, and more. Buffers and images are powerful abstractions that solve many problems for the programmer, but rewriting all interfaces in existing routines to accept buffers or accessors can be time-consuming. Since the interface for buffers and images is largely the same, the rest of this chapter will only focus on USM and buffers.

Unified Shared Memory

Unified Shared Memory (USM) is one of DPC++'s tools for data management. USM is a pointer-based approach that should be familiar to C and C++ programmers that use malloc or new to allocate data. USM simplifies life for the programmer when porting existing C/C++ code to DPC++ since code that accepted pointers can continue to accept pointers. USM ensures that this works by requiring devices that support USM to support a unified virtual address space. Having a unified virtual address space means that any pointer value returned by a USM allocation routine on the host will be valid pointer value on the device. It is unnecessary to map the pointer value on the host to a "device version".

A more detailed description of USM can be found in Chapter 6.

Accessing memory through pointers

Since not all memories are created equal when a system contains host memory and some number of device-attached local memories, USM defines three different types of allocations: `device`, `host`, and `shared`. All types of allocations are performed on the host. Figure 3-3 summarizes the characteristics of each allocation type.

`device` allocations are allocations in device-local memory. They can be read from and written to on a device but are not directly accessible from the host. Programmers must

use explicit copy operations to move data between regular allocations in host memory and `device` allocations.

Allocation Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	✘	✓	device
host	Allocations in host memory	✓	✓	host
shared	Allocations shared between host and device	✓	✓	can migrate back and forth

Figure 3-3: USM Allocation Types

`host` allocations are allocations in host memory that are accessible both on the host and on a device. This means the same pointer value is valid both in host code and in device kernels. However, when these pointers are accessed, the data always comes from host memory. If they are accessed on a device, the data does not migrate from the host to device-local memory. Instead, data is typically sent over a bus, such as PCI-Express (PCI-E), that connects the device to the host.

`shared` allocations are allocations that are accessible on both the host and the device. In this regard they are very similar to host allocations, but they differ in that data can now migrate between host memory and device-local memory. This means that accesses on a device, after the migration has occurred, happen from much faster device-local memory instead of remotely accessing host memory. Typically, this is accomplished through mechanisms inside the DPC++ runtime and lower-level drivers that are mostly hidden from the programmer.

USM and Data Movement

USM supports both explicit and implicit data movement strategies, and different allocation types map to different strategies. Device allocations explicitly move data between host and device while host and shared allocations use implicit data movement.

Explicit Data Movement in USM

Explicit data movement with USM is accomplished with `device` allocations and a DPC++-specific `memcpy()` found in the queue and handler classes. The programmer enqueues `memcpy()` operations to transfer data either from the host to the device or from the device to the host.

Figure 3-4 contains one kernel that operates on a device allocation. Data is copied between `hostArray` and `deviceArray` before and after the kernel executes using `memcpy()` operations. Calls to `wait()` on the queue appear to ensure that the copy to the device has completed before the kernel executes and to ensure that the kernel has completed before the data is copied back to the host. We will learn how we can eliminate these calls later in the chapter.

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

int main() {
  queue myQueue;
  auto dev = myQueue.get_device();
  auto ctxt = myQueue.get_context();

  int hostArray[42];
  int *deviceArray = (int*) malloc_device(42 * sizeof(int), dev, ctxt);
  for (int i = 0; i < 42; i++) {
    hostArray[i] = 42;
  }

  myQueue.submit([&](handler& h) {
    // copy hostArray to deviceArray
    h.memcpy(deviceArray, &hostArray[0], 42 * sizeof(int));
  });
  myQueue.wait();

  myQueue.submit([&](handler& h) {
    h.parallel_for<class foo>(range<1>{42}, [=](id<1> ID) {
      int i = ID[0];
      deviceArray[i]++;
    });
  });
  myQueue.wait();

  myQueue.submit([&](handler& h) {
    // copy deviceArray back to hostArray
    h.memcpy(&hostArray[0], deviceArray, 42 * sizeof(int));
  });
  myQueue.wait();

  free(deviceArray, ctxt);
}
```

Figure 3-4: USM Explicit Data Movement

Implicit Data Movement in USM

Implicit data movement with USM is accomplished with `host` and `shared` allocations. With these types of allocations, the programmer does not need to explicitly insert copy operations to move data between host and device. Instead, the programmer simply accesses the pointers inside a kernel, and any required data movement is performed automatically without programmer intervention. This greatly simplifies porting existing codes to DPC++: simply replace any malloc or new with the appropriate DPC++ USM allocation functions, and everything should just work.

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

int main() {
  queue myQueue;
  auto dev = myQueue.get_device();
  auto ctxt = myQueue.get_context();

  int *hostArray = (int*) malloc_host(42 * sizeof(int), ctxt);
  int *sharedArray = (int*) malloc_shared(42 * sizeof(int),
                                          dev, ctxt);

  for (int i = 0; i < 42; i++) {
    // Initialize hostArray on host
    hostArray[i] = 1234;
  }

  myQueue.submit([&](handler& h) {
    h.parallel_for<class theKernel>(range<1>{42}, [=](id<1> myID) {
      int i = myID[0];

      // access sharedArray and hostArray on device
      sharedArray[i] = hostArray[i] + 1;
    });
  });
  myQueue.wait();

  for (int i = 0; i < 42; i++) {
    // access sharedArray on host
    hostArray[i] = sharedArray[i];
  }

  free(sharedArray, ctxt);
  free(hostArray, ctxt);
}
```

Figure 3-5: USM Implicit Data Movement

In Figure 3-5, we create two arrays, `hostArray` and `sharedArray`, that are host and shared allocations, respectively. We can directly initialize `hostArray` in the host code since it is valid pointer on both host and device. Similarly, it can be directly accessed inside the kernel, performing remote reads of the data. The runtime ensures that `sharedArray` is available on the device before the kernel accesses it and that it is moved back when it is later read by the host code, all without programmer intervention.

Buffers

The other abstraction DPC++ provides for data management is the buffer object. Buffers are a data abstraction that represent one or more objects of a given C++ type. While a buffer itself is a single object, the C++ type encapsulated by the buffer could be an array that contains multiple objects. Buffers represent data objects rather than specific memory addresses, so they cannot be directly accessed like regular C++ arrays. Indeed, a buffer object might map to multiple different memory locations on several different devices, or even on the same device for performance reasons. Instead, we use *accessor* objects to read and write buffers.

A more detailed description of buffers can be found in Chapter 7.

Creating buffers

Buffers can be created in a variety of ways. The simplest method is to simply construct a new buffer object only passing a range that specifies the size of the buffer. However, creating a buffer in this fashion does not initialize its data, meaning that the programmer must first initialize the buffer through other means before attempting to read useful data.

Buffers can also be created from existing data on the host. This is done by invoking one of the several constructors that take either a pointer to an existing host allocation or a set of `InputIterators`. Data is copied during buffer construction from the existing host allocation into the buffer object. A buffer may also be created from an existing `cl_mem` object if the programmer is using the SYCL interoperability features with OpenCL.

Accessing buffers

Buffers may not be directly accessed by the host and device (except through advanced and infrequently-used mechanisms not described here). Instead, we must create accessors in order to read and write buffers. The `accessor` class is heavily templated, so we typically do not directly create `accessor` objects. Instead, we use helper methods contained in the `buffer` class: `get_access()`.

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;

int main() {
  int myData[42];
  for (int i = 0; i < 42; i++) {
    myData[i] = 0;
  }
  {
    queue myQueue;
    buffer<int, 1> myBuffer(myData, range<1>{42});

    myQueue.submit([&](handler& h) {
      // create an accessor to update
      // the buffer on the device
      auto myAccessor =
        myBuffer.get_access<access::mode::read_write>(h);
      h.parallel_for<class theKernel>(
        range<1>{42}, [=](id<1> myID) {
          myAccessor[myID]++;
      });
    });

    // create host accessor
    auto hostAccessor =
      myBuffer.get_access<access::mode::read>();
    for (int i = 0; i < 42; i++) {
      // access myBuffer on host
      std::cout << hostAccessor[i] << " ";
    }
    std::cout << std::endl;
  }
  // myData is updated when myBuffer is
  // destroyed upon exiting scope
  for (int i = 0; i < 42; i++) {
    std::cout << myData[i] << " ";
  }
  std::cout << std::endl;
}
```

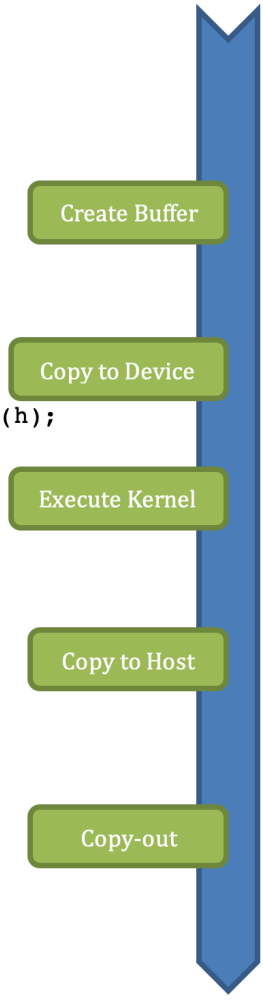


Figure 3-6: Buffers and Accessors

Access Mode	Description
read	Read-only access.
write	Write-only access. Previous contents not discarded.
read_write	Read and write access.
discard_write	Write-only access. Previous contents discarded.
discard_read_write	Read and write access. Previous contents discarded.
atomic	Read and write atomic access.

Figure 3-7: Buffer Access Modes

Access Modes

When creating an accessor, we must inform the runtime how we are going to use it. We do this by specifying an access mode. Access modes are defined in the `sycl::access::mode` `enum` described in Figure 3-7. In the code example shown in Figure 3-6, the accessor `myAccessor` is created with `access::mode::read_write`. This lets the runtime know that we intend to both read and write to the buffer through `myAccessor`. Access modes are how the runtime is able to perform implicit data movement. For example, `access::mode::read` tells the runtime that the data needs to be available on the device before this kernel can begin executing. If a kernel only reads data through an accessor, there is no need to copy data back to the host after the kernel has completed as we haven't modified it. Likewise, `access::mode::write` lets the runtime know that we will modify the contents of a buffer and may need to copy the results

back after computation has ended. Modes `access::mode::discard_write` and `access::mode::discard_read_write` are used to optimize kernels that do not require the original contents of a buffer to be copied to the device before execution, typically because the kernel will overwrite the data.

Creating accessors with the proper modes gives the runtime more information about how we use data in a DPC++ program. The runtime uses accessors to order the use of data, but it can also use this data to optimize scheduling of kernels and data movement. The other access modes are described in greater detail in Chapter 7.

Ordering the Uses of Data

In DPC++, kernels can be viewed as asynchronous tasks that are submitted for execution. These tasks must be submitted to a queue where they are scheduled for execution on a device. In many cases, kernels must execute in a specific order so that the correct result is computed. If obtaining the correct result requires `task A` to execute before `task B`, we say that a *dependence* exists between `tasks A` and `B`.

However, kernels are not the only form of task that arises. Any data that is accessed by a kernel needs to be available on the device before the kernel can start executing. These data dependences can create additional tasks in the form of data transfers from one device to another. Data transfer tasks may be either explicit in the form of programmer-specified copy operations or implicit data movements performed by the DPC++ runtime.

If you take all the tasks in a program and the dependences that exist between them, you can use this to visualize this information as a graph. This task graph is specifically a directed acyclic graph (DAG) where the nodes are the tasks and the edges are the dependences. The graph is *directed* because dependences are one-way: `task A` must happen before `task B`. The graph is *acyclic* because it does not contain any cycles, or paths from a node that lead back to itself.

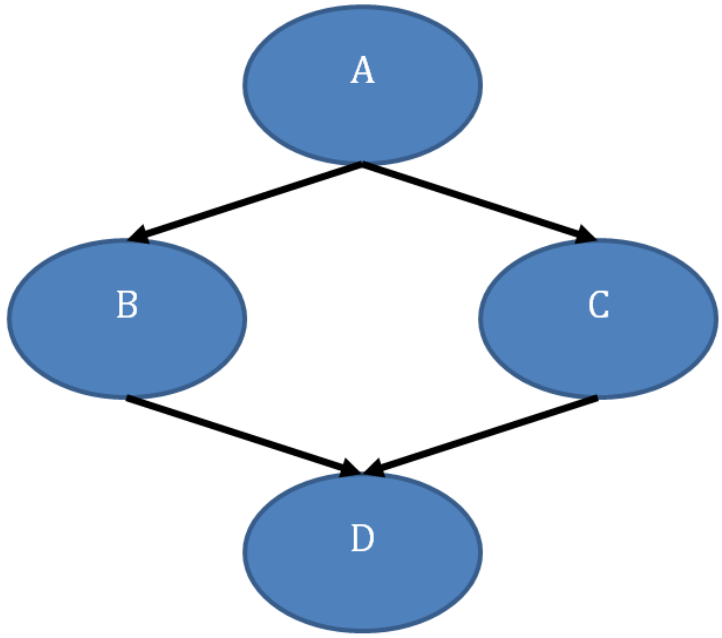


Figure 3-8: Simple Task Graph

In Figure 3-8, `task A` must execute before `tasks B` and `C`. Likewise, `B` and `C` must execute before `task D`. Since `B` and `C` do not have a dependence between each other, the runtime is free to execute them in any order (or even in parallel) as long as `task A` has already executed. Therefore, the possible legal orderings of this graph are `A ⇒ B ⇒ C ⇒ D`, `A ⇒ C ⇒ B ⇒ D`, and even `A ⇒ B, C ⇒ D` if `B` and `C` can concurrently execute.

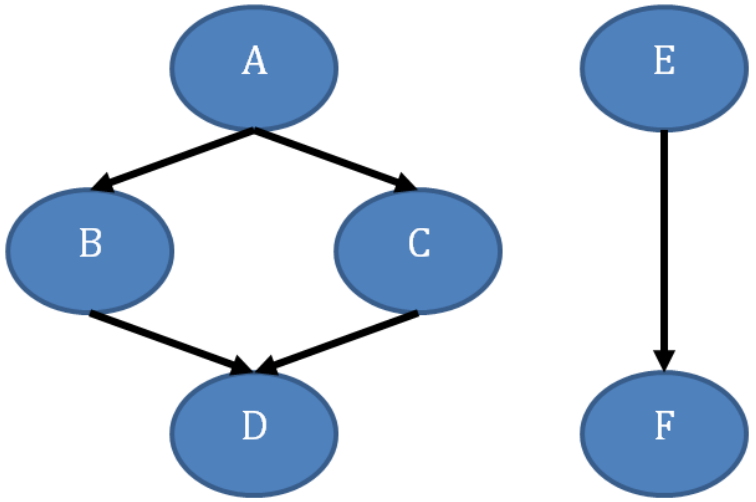


Figure 3-9: Task Graph with Disjoint Dependences

Tasks may have a dependence with a subset of all tasks. In these cases, we only want to specify the dependences that matter for correctness. This slack gives the runtime latitude to optimize the execution order of the task graph. In Figure 3-9, we extend the earlier task graph from Figure 3-8 to add tasks `E` and `F` where `E` must execute before `F`. However, tasks `E` and `F` have no dependences with nodes `A, B, C,` and `D`. This allows the runtime to choose from many possible legal orderings to execute all the tasks.

There are two different ways to model the execution of tasks in a queue: the queue could either execute tasks in the order of submission or it could execute tasks in *any* order. However, executing tasks in an arbitrary order could lead to unexpected results, particularly if the tasks have dependences between each other. DPC++ provides several mechanisms for programmers to order tasks using different strategies.

In-order Queues

The simplest option to order tasks is to submit them to an `ordered_queue` object. An `ordered_queue` is an in-order queue that executes tasks in the order in which they were submitted as seen in Figure 3-10. While the intuitive task ordering of in-order queues provides an advantage in simplicity, it provides a disadvantage in that the execution of tasks will not overlap even if no dependences exist between them. Ordered queues are useful when bringing up applications because they are simple, intuitive, deterministic on execution ordering, and suitable for many codes.

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

int main() {
…
  // Example meant to illustrate
  // - not be a complete application!

  ordered_queue myQueue;
  myQueue.submit([&](handler& h) {
    h.parallel_for<class taskA>(…);
  });
  myQueue.submit([&](handler& h) {
    h.parallel_for<class taskB>(…);
  });
  myQueue.submit([&](handler& h) {
    h.parallel_for<class taskC>(…);
  });
…
}
```

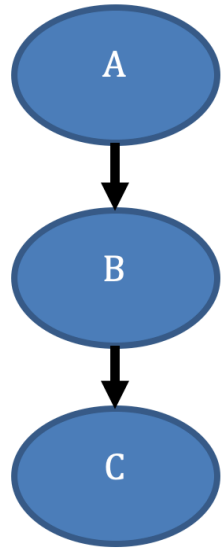


Figure 3-10: `ordered_queue` *Usage*

Out-of-Order (OoO) Queues

Since `sycl::queue` objects in DPC++ are out-of-order queues (unlike `ordered_queue`), they must provide ways to order tasks submitted to them. Queues order tasks by letting the programmers inform the runtime of dependences between them. These dependences can be specified, either explicitly or implicitly, using *command groups*.

A command group is an object that specifies a task and its dependences. Command groups are typically written as C++ lambdas passed as an argument to the `submit()` method of a queue object. This lambda's only parameter is a reference to the handler object representing the command group. The handler object is used to specify tasks, create accessors, and specify dependences.

Explicit Dependences with Events

Explicit dependences between tasks look like the examples we have seen in the previous sections where task `A` must execute before task `B`. This method of expressing dependences focuses on explicitly ordering tasks based on the computations that occur rather than on the data accessed by the computations. Note that this method is primarily relevant for

codes that use USM. In DPC++, we can express these dependences through *event* objects. When submitting a command group to a queue, the `submit()` method returns an event object. These events can then be used in two ways.

First, we can synchronize through the host by explicitly calling the `wait()` method on the event. This forces the runtime to wait for the task that generated the event to finish executing before host program execution may continue. Explicitly waiting on events can be very useful for debugging an application, but `wait()` can overly constrain the asynchronous execution of tasks since it halts all execution on the host thread. Similarly, one could also call `wait()` on a queue object, which would block execution on the host until all enqueued tasks have completed. This can be a useful tool if the programmer does not want to keep track of all the events returned by enqueueing tasks.

This brings us to the second way events can be used. The handler class contains a method named `depends_on()`. This method accepts either a single event or a vector of events and informs the runtime that the command group being submitted requires the specified events to complete before the specified task may execute. Figure 3-11 shows an example of how `depends_on()` may be used to order tasks.

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

int main() {
…
  // Example meant to illustrate
  // - not be a complete application!

  queue myQueue;
  auto eA = myQueue.submit([&](handler& h) {
    h.parallel_for<class taskA>(…);
  });
  eA.wait();
  auto eB = myQueue.submit([&](handler& h) {
    h.parallel_for<class taskB>(…);
  });
  auto eC = myQueue.submit([&](handler& h) {
    h.depends_on(eB);
    h.parallel_for<class taskC>(…);
  });
  myQueue.submit([&](handler& h) {
    h.depends_on({eB, eC });
    h.parallel_for<class taskD>(…);
  });
…
}
```

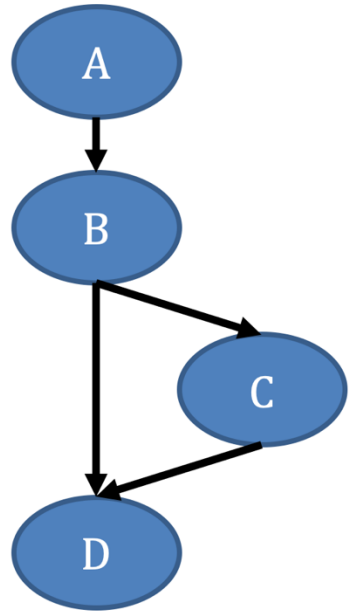


Figure 3-11: Using events and `depends_on`

Implicit Dependences with Accessors

Implicit dependences between tasks in DPC++ are created from data dependences. Data dependences between tasks take three forms, shown in Figure 3-12.

Data dependences are expressed to the runtime through two components: accessors and program order. Both components must be used for the runtime to properly compute data dependences. This is illustrated in Figure 3-13 and Figure 3-14.

Dependence Type	Description
`Read-after-Write (RAW)`	Occurs when task B needs to read data computed by task A.
`Write-after-Read (WAR)`	Occurs when task B writes data after it has been read by task A.
`Write-after-Write(WAW)`	Occurs when task B also writes over data computed by task A.

Figure 3-12: Three forms of Data Dependencies

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;
constexpr int N = 42;

int main() {
  int a[N], b[N], c[N];
  for (int i = 0; i < N; i++) {
    a[i] = b[i] = c[i] = 0;
  }

  queue myQueue;
  buffer<int, 1> A(a, range<1>{N});
  buffer<int, 1> B(b, range<1>{N});
  buffer<int, 1> C(c, range<1>{N});

  myQueue.submit([&](handler& h) {
    auto accA = A.get_access<access::mode::read>(h);
    auto accB = B.get_access<access::mode::write>(h);

    h.parallel_for<class computeB>(range<1>{N}, [=](id<1> ID) {
      accB[ID] = accA[ID] + 1;
    });
  });

  myQueue.submit([&](handler& h) {
    auto accA = A.get_access<access::mode::read>(h);

    h.parallel_for<class readA>(range<1>{N}, [=](id<1> ID) {
      // Useful only as an example
      int data = accA[ID];
    });
  });

  myQueue.submit([&](handler& h) {
    // RAW of buffer B
    auto accB = B.get_access<access::mode::read>(h);
    auto accC = C.get_access<access::mode::write>(h);

    h.parallel_for<class computeC>(range<1>{N}, [=](id<1> ID) {
      accC[ID] = accB[ID] + 2;
    });

  });

  // read C on host
  auto hostAccC = C.get_access<access::mode::read>();
  for (int i = 0; i < N; i++) {
    std::cout << hostAccC[i] << " ";
  }
  std::cout << std::endl;
}
```

Figure 3-13: Read-After-Write

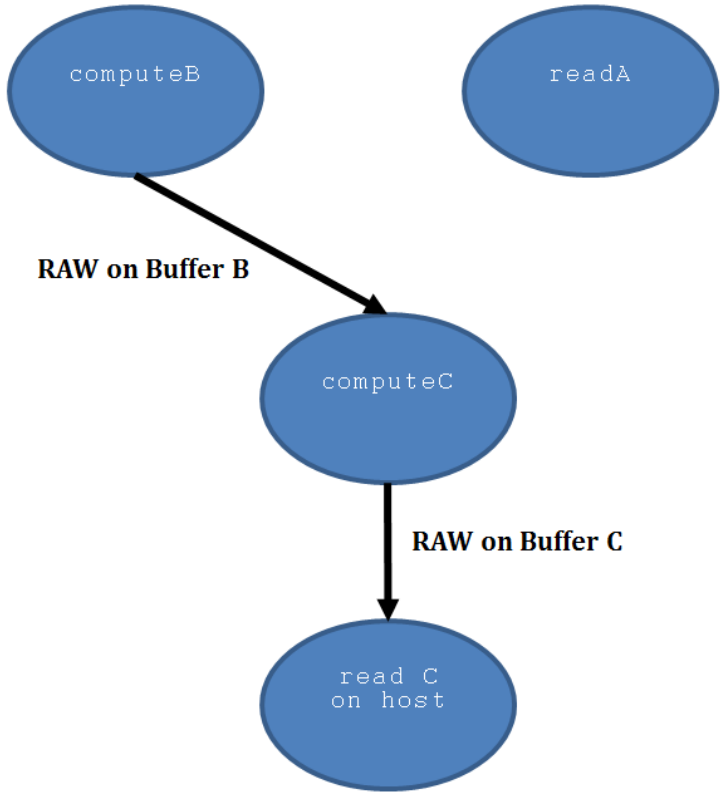


Figure 3-14: RAW Task Graph

In Figure 3-13 and Figure 3-14, we execute three kernels: `computeB`, `readA`, and `computeC`, then read the final result back on the host. The command group for kernel `computeB` creates two accessors, `accA` and `accB`. Kernel `computeB` reads buffer `A` and writes buffer `B`. Buffer `A` must be copied from the host to the device before the kernel begins execution.

Kernel `readA` also creates a read-only accessor for buffer `A`. Since kernel `readA` is submitted after kernel `computeB`, this creates a Read-after-Read (RAR) scenario. However, RARs do not place extra restrictions on the runtime, and the kernels are free to execute in any order. Indeed, a runtime might prefer to execute kernel `readA` before kernel `computeB` or even execute both at the same time. Both require buffer `A` to be copied to the device, but kernel `computeB` also requires buffer `computeB` to be copied since we didn't specify a `discard_write` access mode. This means that the runtime could execute kernel `readA` while the data transfer for buffer `B` occurs.

Kernel `computeC` reads buffer `B`, which we computed in kernel `computeB`. Since we submitted kernel `computeC` after we submitted kernel `computeB`, this means that kernel `computeC` has a RAW data dependence on buffer `B`. RAW dependences are also called true dependences or flow dependences, as data needs to flow from one computation to another in order to compute the correct result. Finally, we also create a RAW dependence on buffer `c` between kernel `computeC` and the host, since the host wants to `readC` after the kernel has finished. This forces the runtime to copy buffer `c` back to the host. Since kernel `computeC` does not later write to buffer `A`, the runtime does not need to copy it back to the host.

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;
constexpr int N = 42;

int main() {
  int a[N], b[N];
  for (int i = 0; i < N; i++) {
    a[i] = b[i] = 0;
  }
  {
    queue myQueue;
    buffer<int, 1> A(a, range<1>{N});
    buffer<int, 1> B(b, range<1>{N});

    myQueue.submit([&](handler& h) {
      auto accA = A.get_access<access::mode::read>(h);
      auto accB = B.get_access<access::mode::write>(h);

      h.parallel_for<class computeB>(range<1>{N}, [=](id<1> ID) {
        accB[ID] = accA[ID] + 1;
      });
    });

    myQueue.submit([&](handler& h) {
      // WAR of buffer A
      auto accA = A.get_access<access::mode::write>(h);
      h.parallel_for<class rewriteA>(range<1>{N}, [=](id<1> ID) {
        accA[ID] = 21 + 21;
      });
    });

    myQueue.submit([&](handler& h) {
      // WAW of buffer B
      auto accB = B.get_access<access::mode::write>(h);
      h.parallel_for<class rewriteB>(range<1>{N}, [=](id<1> ID) {
        accB[ID] = 30 + 12;
      });
    });
  }
  for (int i = 0; i < N; i++) {
    std::cout << a[i] << " " << b[i] << " ";
  }
  std::cout << std::endl;
}
```

Figure 3-15: Write-After-Read and Write-After-Write

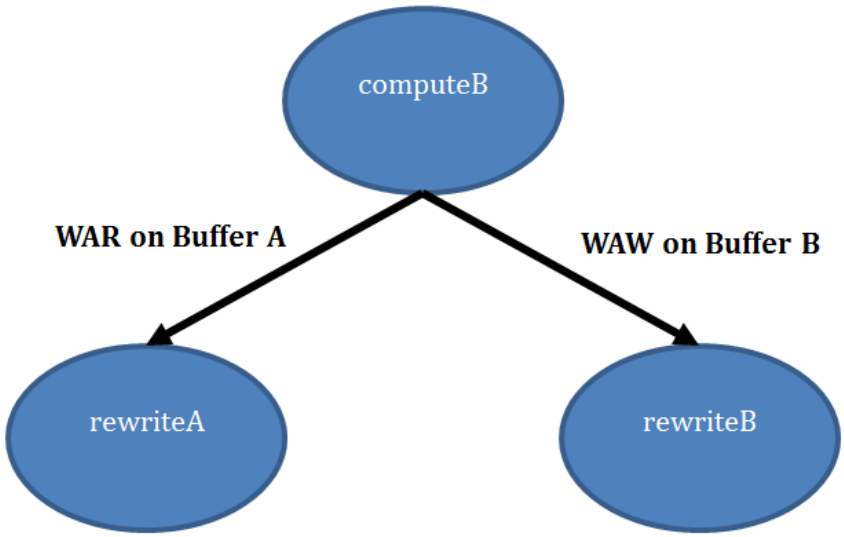


Figure 3-16: WAR and WAW Task Graph

In Figure 3-15 and Figure 3-16, we again execute three kernels: `computeB`, `rewriteA`, and `rewriteB`. Kernel `computeB` once again reads buffer `A` and writes buffer `B`, kernel `rewriteA` writes to buffer `A`, and kernel `rewriteB` writes to buffer `B`. Kernel `rewriteA` could theoretically execute earlier than kernel `computeB` since less data needs to be transferred before the kernel is ready, but it must wait until after kernel `computeB` finishes since there is a WAR dependence on buffer `A`. In this example, kernel `computeB` requires the original value of `A` from the host, and it would read the wrong values if kernel `rewriteA` executed before kernel `computeB`. WAR dependences are also called anti-dependences. RAW dependences ensure that data properly flows in the correct direction while WAR dependences ensure existing values are not overwritten before they are read. The WAW dependence on buffer `B` found in kernel rewrite functions similarly. If there were any reads of buffer `B` submitted in between kernels `computeB` and `rewriteB`, they would result in RAW and WAR dependences that would properly order the tasks. However, there is an implicit dependence between kernel `rewriteB` and the host in this example since the final data must be written back to the host. The WAW dependence, also called an output dependence, ensures that the final output will be correct on the host.

Choosing a Data Management Strategy

Selecting the right data management strategy for your applications is largely a matter of personal preference. Indeed, you may begin with one strategy and switch to another as your DPC++ program matures. However, there are few useful guidelines to help you pick a strategy that will serve your needs.

The first decision to make is whether you want to use explicit or implicit data movement since this greatly affects what needs to be done to bring a program into DPC++. Implicit data movement is generally an easier place to start because DPC++ will handle all the data movement, letting you focus on expressing the computation.

If you decide that you'd rather have full control over all data movement from the beginning, then explicit data movement using USM device allocations is where you want to start. Just be sure to add the necessary copies between host and devices!

When selecting an implicit data movement strategy, you still have a choice of whether to use buffers or USM host or shared pointers. Again, this choice is a matter of personal preference, but there are a few questions that could help guide you to one over the other. If you're porting an existing C/C++ program that that uses pointers, USM might be an easier path since most code won't need to change. Another question to ask is how you would like to express your dependences between kernels. If you prefer to think about data dependences between kernels, choose buffers. If you prefer to think about dependences as performing one computation before another, choose USM.

When using USM pointers (with either explicit or implicit data movement), you have a choice of which type of queue you want to use. In-order queues are simple and intuitive, but they constrain the runtime and may limit performance. Out-of-order queues are more complex, but they give the runtime more freedom to reorder and overlap execution. The out-of-order queue class is the right choice if your program will have complex dependences between kernels. If your program simply runs many kernels one after another, then the in-order `ordered_queue` class will be a better option for you.

Summary

In this chapter, we have introduced the mechanisms that DPC++ uses to address the problems of data management and how to order the uses of data. Managing access to different memories is a key challenge when using accelerator devices, and DPC++ gives programmers different options to suit their needs.

We provided an overview of the different types of dependences that can exist between the uses of data, and we described how to provide information about these dependences to queues so that they properly order tasks.

This chapter provided a very brief introduction to Unified Shared Memory and buffers. We will explore all the modes and behaviors of USM in greater detail in Chapter 6. Chapter 7 will do a deep dive on buffers including all the different ways to create buffers and control their behavior. Chapter 8 will revisit the DAG scheduling mechanism of DPC++ queues.

FOR THIS BOOK PREVIEW (CHAPTERS 1-4):
ERRATA, NOTES, DOWNLOADS, FEEDBACK, ETC.

Please check our "preview book" website for information including errata, updates, and downloads (includes all sample code): https://tinyurl.com/book-dpcpp

We will also list some additional resources as they become available.

Your feedback is welcome. You can email James Reinders at dpc++@jamesreinders.com with any suggestions, encouragement, criticism, or questions that you may have. James will be sure to share any feedback that you send with all the authors.

Of course – watch for the full book, by mid-2020, available from Apress (no charge for PDF for the completed book, print copies will be available too).

https://tinyurl.com/book-dpcpp

Expressing Parallelism

We have already covered how to control where code runs in Chapter 2 (on what device), and how to get data to our code (accessible from a particular device) in Chapter 3. This chapter fills in missing details from the code samples shown so far and starts to transition from simple teaching examples towards real-world parallel code.

Writing your first program in a new parallel language may seem like a daunting task, especially if you are new to parallel programming. Language specifications are not written for application developers, and often assume some familiarity with terminology; they do not contain answers to questions like:

- Why is there more than one way to express parallelism?
- Which method of expressing parallelism should I use?
- How much do I really need to know about the execution model?

This chapter seeks to address these questions, and more. We introduce the concept of a data parallel kernel, discuss the strengths and weaknesses of the different kernel forms available in DPC++ using working code examples, and highlight the most important aspects of the kernel execution model.

DPC++ AND SYCL IN CHAPTER

Work-group collectives and sub-groups are DPC++ extensions at the time this book is going to press, and are not part of the SYCL 1.2.1 specification.

Everything else discussed in this chapter is SYCL, and is therefore fully supported by DPC++.

Parallelism within Kernels

Parallel kernels have emerged in recent years as a powerful means of expressing data parallelism. The primary design goals of a kernel-based approach are *portability* across a wide range of devices and high programmer *productivity*. As such, kernels are typically not hard-coded to work with a specific number or configuration of hardware resources (e.g.

cores, hardware threads, SIMD instructions). Instead, kernels describe parallelism in terms of abstract concepts that an implementation (i.e. a combination of compiler and runtime) can then map to the hardware parallelism available on a particular target device.

Exposing a great deal of parallelism in a hardware-agnostic way ensures that applications can scale up (or down) to fit the capabilities of different platforms, but...

Guaranteeing functional portability is not the same as guaranteeing high performance!

There is a significant amount of diversity in the devices supported by DPC++, and we must remember that different architectures are designed and optimized for different use-cases. Whenever we hope to achieve the highest levels of *performance* on a specific device, we should always expect that some additional manual optimization work will be required — regardless of the programming language we're using! Examples of such device-specific optimizations include: blocking for a particular cache size; choosing a grain size that amortizes scheduling overheads; making use of specialized instructions or hardware units; and most importantly, choosing an appropriate algorithm. Some of these examples will be revisited in Chapters 13, 14 and 15.

Striking the right balance between performance, portability and productivity during application development is a challenge that we must all face — and a challenge that this book cannot address in its entirety. However, we hope to show that DPC++ provides all of the tools required to maintain both generic portable code and optimized target-specific code using a single high-level programming language. The rest is left as an exercise to the reader!

Multi-dimensional Kernels

The parallel constructs of many other languages are one-dimensional, mapping work directly to a corresponding one-dimensional hardware resource (e.g. number of hardware threads). Parallel kernels are a higher-level concept than this, and their dimensionality is more reflective of the problems that our codes are typically trying to solve (in a one-, two- or three-dimensional space).

However, we must remember that the multi-dimensional indexing provided by the SYCL execution model is a programmer convenience implemented on top of an underlying one-dimensional space. Understanding how this mapping behaves can be an important part of certain optimizations (e.g. tuning memory access patterns).

All multi-dimensional quantities related to parallelism in DPC++ use the same *row-major* convention, and in all contexts the contiguous dimension of a multi-dimensional quantity will be its *rightmost*. Unfortunately, we often find ourselves in situations where terms like "row-major" and "rightmost" are of little help: for example, we may need to refer to a single, specific dimension! To avoid confusion, dimensions are also numbered from 0 to `N-1`, where dimension `N-1` corresponds to the contiguous dimension in an `N`-dimensional space. An example of mapping two dimensions to a linear index using the SYCL convention is shown in Figure 4-1. We are of course free to break from this convention and adopt our own methods of linearizing indices, but must do so carefully — breaking from the SYCL convention may have a negative performance impact on devices that benefit from stride-one accesses.

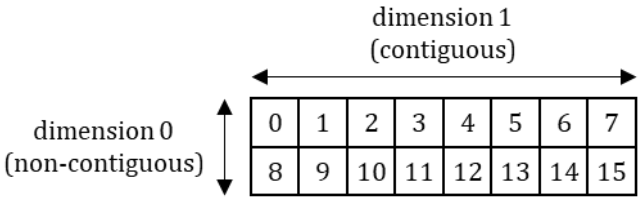


Figure 4-1: A two-dimensional range of size (2, 8) mapped to linear indices.

If an application requires more than three dimensions, we must take responsibility for mapping between multi-dimensional and linear indices manually, using modulo arithmetic.

Loops vs. Kernels

An iterative loop is an inherently serial construct: each iteration of the loop is executed sequentially (i.e. in order). An optimizing compiler may be able to determine that some or all iterations of a loop can execute in parallel, but it must be conservative — if the compiler isn't smart enough or doesn't have enough information to prove that parallel execution is always safe, it must preserve the loop's sequential semantics for correctness.

```
for (int i = 0; i < N; ++i) {
  c[i] = a[i] + b[i];
};
```

Figure 4-2: A vector addition expressed as a serial loop.

Consider the loop in Figure 4-2, which describes a simple vector addition. Even in a simple case like this, proving that the loop can be executed in parallel is not trivial: parallel execution is only safe if `c` does not overlap `a` or `b`, which in the general case cannot be proven without a run-time check! In order to address situations like these, languages have added features enabling us to provide compilers with extra information that may simplify analysis (e.g. asserting that pointers do not overlap with `restrict`), or to override all analysis altogether (e.g. declaring that all iterations of a loop are independent, or defining exactly how the loop should be scheduled to parallel resources).

The exact meaning of a "parallel loop" is somewhat ambiguous — due to overloading of the term by different parallel programming languages — but many common parallel loop constructs represent compiler transformations applied to sequential loops. Such programming models enable us to write sequential loops and only later provide information about how different iterations can be executed safely in parallel. These models are very powerful, integrate well with other state-of-the-art compiler optimizations, and greatly simplify parallel programming, but do not always encourage us to think about parallelism at an early stage of development.

A parallel kernel is not a loop, and does not have iterations. Rather, a kernel describes a single operation, which can be instantiated many times and applied to different input data; when a kernel is launched in parallel, multiple instances of that operation are executed simultaneously.

```
launch N kernel instances {
  int id = get_instance_id(); // unique identifier in [0, N)
  c[i] = a[i] + b[i];
}
```

Figure 4-3: The example loop rewritten (in pseudocode) as a parallel kernel.

Figure 4-3 shows our simple loop example rewritten as a kernel using pseudocode. The opportunity for parallelism in this kernel is clear and explicit: the kernel can be executed in parallel by any number of instances, and each instance independently applies to a separate piece of data.

In short: kernel-based programming is not a way to retrofit parallelism into existing sequential codes, but a methodology for writing explicitly parallel applications.

The sooner that we can shift our thinking from loops to kernels, the easier it will be to write effective parallel programs using DPC++.

Overview of Language Features

Once we've decided to write a parallel kernel, we must decide what type of kernel we want to launch and how to represent it in our program. There are a multitude of ways to express parallel kernels in DPC++, and we must familiarize ourselves with all of these options if we want to master the language.

Separating Kernels from Host Code

DPC++ offers a number of alternative ways to separate host and device code, which we can mix and match within an application: "single-source" C++ lambda expressions or function objects (functors); OpenCL C source strings; or binaries. Some of these options were already covered in Chapter 2, and all of them will be covered in more detail in Chapter 10.

The fundamental concepts of expressing parallelism in DPC++ are shared by all of these options. For consistency and conciseness, all of the code examples in this chapter express kernels using C++ lambdas.

LAMBDAS NOT CONSIDERED HARMFUL

Not all programmers like using C++ lambda expressions. They're (relatively) new to the language, they have unfamiliar syntax, and some compilers have historically had some problems optimizing around them. However, there is no need to fully understand everything that the C++ specification says about lambdas in order to get started with DPC++ — all you need to know is how to copy the simple code examples found in this book and how to write the body of a loop. There is no need to worry about any potential negative performance impacts, either — a DPC++ compiler always understands when a lambda represents the body of a parallel kernel, and can optimize for parallel execution accordingly.

Different Forms of Parallel Kernel

DPC++ provides three different kernel forms, each with their own execution models and syntax. It is possible to write portable kernels using any of the kernel forms, and kernels written in any form can be tuned to achieve high performance on a wide variety of device types. However, there will be times when we may want to use a particular form to make a specific parallel algorithm easier to express, or to make use of an otherwise inaccessible language feature.

The first kernel form is used for *basic* data parallel kernels, and offers the gentlest introduction to writing kernels in DPC++. With basic kernels, we sacrifice control over low-

level features like scheduling in order to make the expression of the kernel as simple as possible. How the individual kernel instances are mapped to hardware resources is controlled entirely by the implementation, and so as basic kernels grow in complexity it becomes harder and harder to reason about their performance.

The second kernel form extends basic kernels to provide access to low-level performance-tuning features. This second form is known as *ND-range* (N-dimensional range) data parallel for historical reasons, and the most important thing to remember is that it enables certain kernel instances to be grouped together, allowing us to exert some control over data locality and the mapping between individual kernel instances and the hardware resources that will be used to execute them.

The third and final form provides syntactic sugar to simplify the expression of ND-range kernels using nested kernel constructs. This third form is referred to as *hierarchical* data parallel, referring to the hierarchy of the nested kernel constructs that appear in user source code.

We will revisit how to choose between the different kernel forms again at the end of this chapter, once we've discussed their features in more detail.

Basic Data Parallel Kernels

The most basic form of parallel kernel in DPC++ is appropriate for operations that are *embarrassingly parallel* (i.e. operations that can be applied to every piece of data completely independently and in any order). By using this form, we give an implementation ultimate freedom to schedule work. It is thus an example of a *descriptive* programming construct — we *describe* that the operation is embarrassingly parallel, and all scheduling decisions are made by the implementation.

Basic data parallel kernels are written in a **S**ingle **P**rogram **M**ultiple **D**ata (SPMD) style — a single "program" (the kernel) is applied to multiple pieces of data. Note that this programming model still permits each instance of the kernel to take different paths through the code, as a result of data-dependent branches.

Basic Data Parallel Kernels: Execution Model

The execution space of a basic parallel kernel is referred to in DPC++ as its execution *range*, and each instance of the kernel functor is referred to as an *item*. This is represented diagrammatically in Figure 4-4.

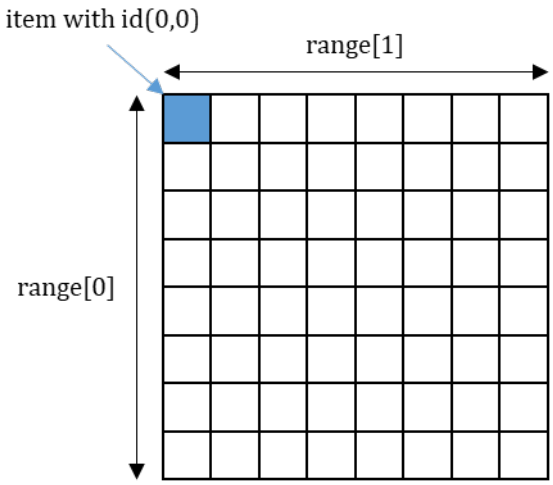


Figure 4-4: The execution space of a basic parallel kernel, shown for a 2D range of 64 items.

The execution model of basic data parallel kernels is very simple: it *allows* for completely parallel execution, but does not *guarantee* or *require* it. Items can be executed in *any* order, including sequentially on a single hardware thread (i.e. without any parallelism)! Kernels that assume that all items *will* be executed in parallel (e.g. by attempting to synchronize items) could therefore very easily lead programs to hang on some devices.

However, in order to guarantee correctness we must always write our kernels assuming that they *could* be executed in parallel. For example, it is our responsibility to ensure that concurrent accesses to memory are appropriately guarded by atomic memory operations in order to prevent race conditions.

Basic Data Parallel Kernels: Syntax

Basic data parallel kernels are expressed using the `parallel_for` function, which is a member of the `handler` class and can only be called at command-group scope. Figure 4-5 shows how to use this function to express a vector addition, which is the equivalent of "Hello world!" for parallel accelerator programming.

```
cgh.parallel_for<class vector_add>(range<1>(N), [=](id<1> i)
{
  c[i] = a[i] + b[i];
});
```

Figure 4-5: A vector addition kernel expressed with `parallel_for`.

The function only takes two arguments: the first is a `range` specifying the number of items to launch in each dimension; and the second is a kernel function to be executed for each index in the range. The kernel can take either an `id` or an `item` as its argument, and which should be used depends on which class exposes the functionality required by your use-case — we'll revisit this later.

Figure 4-6 shows a very similar use of this function to express a matrix addition, which is (mathematically) identical to vector addition except working with two-dimensional data. This is reflected by the kernel — the only difference between the two code snippets is the dimensionality of the `range` and `id` classes used! It is possible to write the code this way because a SYCL `accessor` can be indexed by a multi-dimensional `id`. As strange as it looks, this can be very powerful, enabling us to write templated kernels that operate on data of any dimensionality.

```
cgh.parallel_for<class matrix_add>(range<2>(N, M), [=](id <2> i)
{
  c[i] = a[i] + b[i];
});
```

Figure 4-6: A matrix addition kernel expressed with `parallel_for`.

It is more common in C/C++ to use multiple indices and multiple subscript operators to index multi-dimensional data structures, and this explicit indexing is also supported by DPC++. This method of indexing can improve readability when a kernel operates on data of different dimensionalities simultaneously, or when the memory access patterns of a kernel are more complicated than can be described by using an item's `id` directly.

For example, the matrix multiplication kernel in Figure 4-7must extract the two individual components of the index in order to be able to describe the dot-product between rows and columns of the two matrices. In our opinion, consistently using multiple subscript operators (e.g. `[j][k]`) is more readable than mixing multiple indexing modes and constructing two-dimensional `id` objects (e.g. `id(j,k)`), but this is simply a matter of personal preference.

The examples in the remainder of this chapter all use multiple subscript operators, to ensure that there is no ambiguity in the dimensionality of the buffers being accessed.

```
cgh.parallel_for<class matrix_mul>(range<2>(N, N), [=](id<2> ji)
{
    int j = ji[0];
    int i = ji[1];

    for (int k = 0; k < N; ++k)
    {
        c[j][i] += a[j][k] * b[k][i];
        // or c[ji] += a[id(j,k)] * b[id(k,i)];
    }
});
```

Figure 4-7: A naïve matrix multiplication kernel expressed with `parallel_for`.

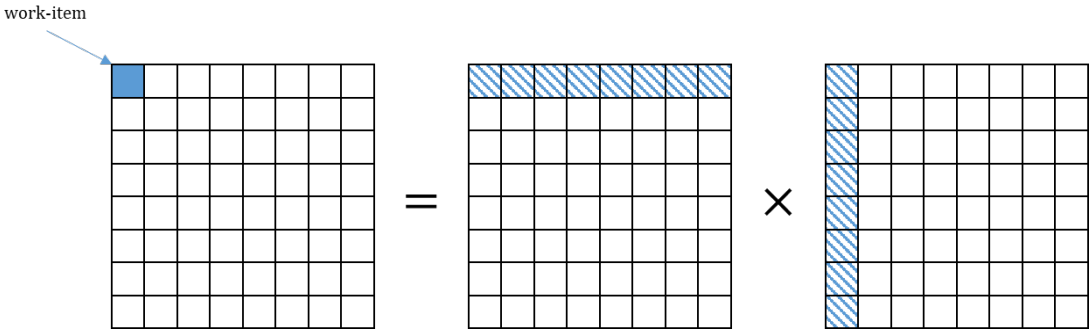


Figure-8: Mapping of matrix multiplication work to items in the execution range.

The diagram in Figure-8 shows how the work in our matrix-multiplication kernel is mapped to individual items. Each item computes a single value of the C matrix, by iterating sequentially over a (contiguous) row of the `A` matrix and a (non-contiguous) column of the `B` matrix.

Basic Data Parallel Kernels: Important Classes

The functionality of basic data parallel kernels is exposed via three C++ classes: `range`, `id` and `item`. The `range` and `id` classes were also discussed during the introduction of buffers in Chapter 3, but we revisit them here with a different focus.

The `range` Class

A `range` represents a one-, two- or three-dimensional range. The dimensionality of a `range` is a template argument and must therefore be known at compile-time, but its size in each dimension is dynamic and is passed to the constructor at run-time. Instances of

the `range` class are used in DPC++ to describe both the execution ranges of parallel constructs and the sizes of buffers.

A simplified definition of the `range` class, showing the constructors and various methods for querying its extent, is shown in Figure 4-9.

```
template <int dimensions = 1>
class range {
public:
  // Construct a range with one, two or three dimensions
  range(size_t dim0);
  range(size_t dim0, size_t dim1);
  range(size_t dim0, size_t dim1, size_t dim2);

  // Return the size of the range in a specific dimension
  size_t get(int dimension) const;
  size_t &operator[](int dimension);
  size_t operator[](int dimension) const;

  // Return the product of the size of each dimension
  size_t size() const;

  // Arithmetic operations on ranges are also supported
};
```

Figure 4-9: A simplified definition of the `range` class.

The `id` Class

An `id` represents an index into a one, two- or three-dimensional range. The definition of `id` is similar in many respects to `range`: its dimensionality must also be known at compile-time; and it may be used to index an individual instance of a kernel in a parallel construct or an offset into a buffer.

As shown by the simplified definition of the `id` class in Figure 4-10, an `id` is conceptually nothing more than a container of one, two or three integers. The operations available to us are also very simple: we can query the component of an index in each dimension, and we can perform simple arithmetic to compute new indices.

Although we can construct an `id` to represent an arbitrary index, the only way to obtain the `id` associated with a particular kernel instance is to accept it as an argument to a kernel function. This `id` (or values returned by its member functions) must be forwarded to any device function in which we want to query the index. If a kernel functor

accepts an instance of `id`, there is no way to identify how many instances of the kernel function were launched.

```
template <int dimensions = 1>
class id {
public:
  // Construct an id with one, two or three dimensions
  id(size_t dim0);
  id(size_t dim0, size_t dim1);
  id(size_t dim0, size_t dim1, size_t dim2);

  // Return the component of the id in a specific dimension
  size_t get(int dimension) const;
  size_t &operator[](int dimension);
  size_t operator[](int dimension) const;

  // Arithmetic operations on ids are also supported
};
```

Figure 4-10: A simplified definition of the `id` class.

The `item` Class

An `item` represents an individual instance of a kernel function, encapsulating both the execution range of the kernel and the instance's index within that range (using a `range` and an `id`, respectively). Like `range` and `id`, its dimensionality must be known at compile-time.

A simplified definition of the `item` class is given in Figure 4-11. The main difference between `item` and `id` is that `item` exposes additional functions to query properties of the execution range (e.g. size, offset) and a convenience function to compute a linearized index. As with `id`, the only way to obtain the `item` associated with a particular kernel instance is to accept it as an argument to a kernel function.

```
template <int dimensions = 1, bool with_offset = true>
class item {
public:
  // Return the index of this item in the kernel's execution range
  id<dimensions> get_id() const;
  size_t get_id(int dimension) const;
  size_t operator[](int dimension) const;

  // Return the execution range of the kernel executed by this item
  range<dimensions> get_range() const;
  size_t get_range(int dimension) const;

  // Return the offset of this item (if with_offset == true)
  id<dimensions> get_offset() const;

  // Return the linear index of this item
  // e.g. id(0) * range(1) * range(2) + id(1) * range(2) + id(2)
  size_t get_linear_id() const;
};
```

Figure 4-11: A simplified definition of the `item` *class.*

Explicit ND-Range Kernels

The second form of parallel kernel in DPC++ replaces the flat execution range of basic data parallel kernels with an execution range where items belong to groups, and is appropriate for cases where we would like to express some notion of locality within our kernels. Different behaviors are defined and guaranteed for different types of groups, giving us more insight into and/or control over how work is mapped to specific hardware platforms.

These explicit ND-range kernels are thus an example of a more *prescriptive* parallel construct — we *prescribe* a mapping of work to each type of group, and the implementation must obey that mapping. However, it is not completely prescriptive, as the groups themselves may execute in any order and an implementation retains some freedom over how each type of group is mapped to hardware resources. This combination of prescriptive and descriptive programming enables us to design and tune our kernels for locality without impacting their portability.

Like basic data parallel kernels, ND-range kernels are written in a SPMD style where all work-items execute the same kernel "program" applied to multiple pieces of data. The key difference is that each program instance can query its position within the groups that contain it, and can access additional functionality specific to each type of group.

Explicit ND-Range Parallel Kernels: Execution Model

The execution range of an ND-range kernel is divided into work-groups, sub-groups and work-items (as shown in Figure 4-12). The ND-range represents the total execution range, which is divided into work-groups of uniform size (i.e. the work-group size must divide the ND-range size exactly in each dimension). Each work-group can be further divided by the implementation into sub-groups. Understanding the execution model defined by DPC++ for work-items and each type of group is an important part of writing correct and portable programs.

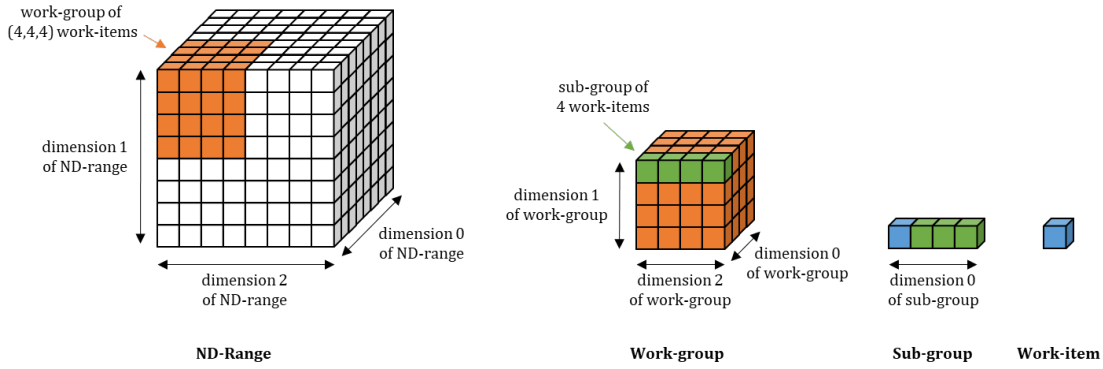


Figure 4-12: A three-dimensional ND-range of size (8, 8, 8) divided into 8 work-groups of size (4, 4, 4). Each work-group contains 16 one-dimensional sub-groups of 4 work-items.

The exact mapping from each type of group to hardware resources is *implementation-defined*, and it is this flexibility that enables DPC++ programs to execute on a wide variety of hardware. For example, work-items could be executed completely sequentially, executed in parallel by hardware threads and/or SIMD instructions, or even executed by a hardware pipeline specifically configured for a particular kernel.

In this chapter we are focused only on the semantic guarantees of the ND-range execution model in terms of a generic target platform, and we will not cover its mapping to any one platform. The reader is referred to Chapters 13, 14 and 15 for details of the hardware mapping and performance recommendations for GPUs, CPUs and FPGAs respectively.

Work-items

Work-items represent the individual instances of a kernel function. In the absence of other groupings, work-items can be executed in any order and cannot communicate or synchronize with each other except by way of atomic memory operations to global memory.

Work-groups

The work-items in an ND-range are organized into work-groups. Work-groups can execute in any order, and work-items in different work-groups cannot communicate or synchronize with each other except by way of atomic memory operations to global memory. However, the work-items within a work-group have concurrent scheduling guarantees when certain constructs are used, and this locality provides some additional capabilities:

1. Work-items in a work-group have access to *work-group local memory*, which may be mapped to a dedicated fast memory on some devices

2. Work-items in a work-group can synchronize using *work-group barriers* and guarantee memory consistency using *work-group memory fences*

3. Work-items in a work-group have access to *work-group collectives*, providing fast implementations of common parallel patterns

The number of work-items in a work-group is typically configured for each kernel at run-time, as the best grouping will depend upon both the amount of parallelism available (i.e. the size of the ND-range) and properties of the target device. We can determine the maximum number of work-items per work-group supported by a particular device using the query functions of the `device` class, and it is our responsibility to ensure that the work-group size requested for each kernel is valid.

There are some subtleties in the work-group execution model that are worth emphasizing.

First, although the work-items in a work-group are scheduled to a single compute unit, there need not be any relationship between the number of work-groups and the number of compute units. In fact, the number of work-groups in an ND-range can be many times larger than the number of work-groups that a given device is capable of executing concurrently! If you are a parallel programming expert already, you may be

tempted to try and write kernels that synchronize across work-groups by relying on very clever device-specific scheduling, but we strongly recommend that you do not attempt this — such kernels are very sensitive to implementation details that may change or be non-deterministic.

Second, although the work-items in a work-group are scheduled concurrently, they are not guaranteed to make *independent forward progress* — executing the work-items within a work-group sequentially between barriers and collectives is a valid implementation. Communication and synchronization between work-items in the same work-group is only guaranteed to be safe when performed using the barrier and collective functions provided, and hand-coded synchronization routines may deadlock.

THINKING IN WORK-GROUPS

Work-groups are similar in many respects to the concept of a task in other programming models (e.g. Threading Building Blocks): tasks can execute in any order (controlled by a scheduler); it's possible (and even desirable) to oversubscribe a machine with tasks; and it's often not a good idea to try and implement a barrier across a group of tasks (as it may be very expensive or incompatible with the scheduler). If you're already familiar with a task-based programming model, you may find it useful to think of work-groups as though they are data-parallel tasks.

Sub-groups

On many modern hardware platforms, a subset of the work-items in a work-group are executed simultaneously (e.g. as a result of compiler vectorization) or with additional scheduling guarantees (e.g. because subsets are mapped to independent hardware threads). When working with a single platform it is tempting to bake such execution model assumptions into our codes, but this makes them inherently unsafe and non-portable — they may break when moving between different compilers, or even when moving between different generations of hardware from the same vendor!

These subsets of work-items are known in DPC++ as *sub-groups*. Defining sub-groups as a core part of the language gives us a safe alternative to making assumptions that may later prove to be device-specific. Leveraging sub-group functionality also allows us to reason about the execution of work-items at a low level (i.e. close to hardware) and is key to achieving very high levels of performance across many platforms.

As with work-groups, work-items within a sub-group are able to access additional capabilities:

1. Work-items in a sub-group can communicate directly, without explicit memory operations, using *shuffle operations*

2. Work-items in a sub-group can synchronize using *sub-group barriers* and guarantee memory consistency using *sub-group memory fences*

3. Work-items in a sub-group have access to *sub-group collectives*, providing fast implementations of common parallel patterns

The number of sub-groups per work-group is implementation-defined and outside of our control. However, a sub-group has a fixed (one-dimensional) size for a given combination of device, kernel and ND-range, and we can query this size using the query functions of the `kernel` class. By default, the number of work-items per sub-group is also chosen by the implementation — we can override this behavior by requesting a particular sub-group size at compile-time, but must ensure that the sub-group size we request is compatible with the device.

Like work-groups, the work-items in a sub-group are only guaranteed to execute concurrently between sub-group functions, and an implementation is free to execute the work-items in a sub-group sequentially. Where sub-groups are special is that they are the only part of DPC++ capable of providing some guarantee of independent forward progress — on some devices, the sub-groups within a work-group are guaranteed to make independent forward progress with respect to other sub-groups in the same work-group. Whether or not this forward progress guarantee holds can be determined using a device query.

THINKING IN SUB-GROUPS

If you are coming from a programming model that requires you to think about explicit vectorization, it may be useful to think of each sub-group as a set of work-items packed into a SIMD register, where each work-item in the sub-group corresponds to a SIMD lane. When multiple sub-groups are in flight simultaneously and a device guarantees they will make independent forward progress, this mental model extends to treating each sub-group as though it were a separate stream of vector instructions executing in parallel.

Explicit ND-Range Data Parallel Kernels: Syntax

Figure 4-13 re-implements the matrix multiplication kernel that we saw previously using the ND-range parallel kernel syntax, and the diagram in Figure 4-14 shows how the work in this kernel is mapped to the work-items in each work-group. Grouping our work-items in this way ensures locality of access and hopefully improves cache hit rates: for example, the work-group in Figure 4-14 has a local range of (4, 4) and contains 16 work-items, but only accesses four times as much data as a single work-item — in other words, each value we load from memory can be re-used four times. Tuning the work-group size (i.e. the `B` parameter) to realize this benefit based on the cache size of specific devices is left as an exercise to the reader.

```
range<2> global(N, N);
range<2> local(B, B);
cgh.parallel_for<class matrix_mul>(nd_range<2>(global, local),
                                   [=](nd_item<2> item)
{
    int j = item.get_global_id(0);
    int i = item.get_global_id(1);

    for (int k = 0; k < N; ++k)
    {
        c[j][i] += a[j][k] * b[k][i];
    }
});
```

Figure 4-13: A naïve matrix multiplication kernel expressed with ND-range `parallel_for`.

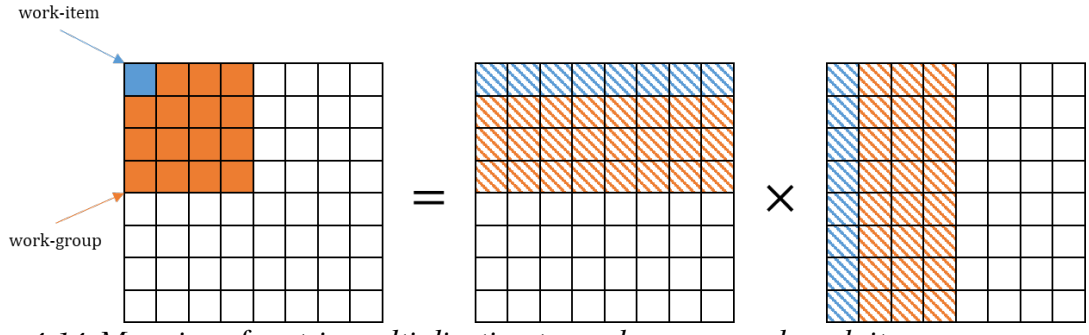


Figure 4-14: Mapping of matrix multiplication to work-groups and work-items.

So far, our matrix multiplication example has relied on a hardware cache to optimize repeated accesses to the A and B matrices from work-items in the same work-group. Such hardware caches are commonplace on traditional CPU architectures and are becoming increasingly so on GPU architectures, but there are other architectures (e.g.

previous generation GPUs, FPGAs) with explicitly managed "scratchpad" memories. ND-range kernels are able to use *local accessors* to describe allocations that should be placed in work-group local memory, and an implementation is then free to map these allocations to special memory (where it exists). A version of our matrix multiplication kernel updated to use local accessors is shown in Figure 4-15. Usage of local memory will be covered in more detail in Chapter 9.

```
range<2> global(N, N);
range<2> local(B, B);
auto a_tile = local_accessor<float,2>(range<2>(B, B), cgh);
auto b_tile = local_accessor<float,2>(range<2>(B, B), cgh);
cgh.parallel_for<class matrix_mul>(nd_range<2>(global, local),
                                   [=](nd_item<2> item)
{
    int j = item.get_global_id(0);
    int i = item.get_global_id(1);

    int lj = item.get_local_id(0);
    int li = item.get_local_id(1);

    for (int kb = 0; kb < N/B; ++kb)
    {
        // Load tiles of A and B matrices into local memory
        a_tile[lj][ti] = a[j][kb*B + li];
        b_tile[lj][ti] = b[kb*B + lj][i];

        // Wait for load into local memory to complete
        // Barrier synchronizes all work-items in this work-group
        item.barrier(access::fence_space::local_space);

        // Compute matrix multiply using results in local memory
        for (int k = 0; k < B; ++k)
        {
            c[j][i] += a_tile[lj][k] * b_tile[k][li];
        }

        // Ensure all work-items are done before overwriting local memory
        // Barrier synchronizes all work-items in this work-group
        item.barrier(access::fence_space::local_space);
    }
});
```

Figure 4-15: A tiled matrix multiplication kernel expressed with ND-range `parallel_for` and work-group local memory.

The operation of this new kernel can be thought of as two distinct phases: in the first, the work-items in the work-group collaborate to load shared data from the A and B matrices into work-group local memory; and in the second, the work-items perform their own private computations using that data. In order to ensure that the first phase has completed on all work-items in the work-group before any work-item begins executing the second phase, they are separated by a call to `barrier()`, which synchronizes all work-

items and also acts as a memory fence. This pattern is a common one, and the use of work-group local memory in a kernel almost always necessitates the use of work-group local barriers. Note that a call to `mem_fence()` is insufficient in cases like these: a memory fence imposes an ordering on the execution of instructions from a single work-item, but does not force each work-item to wait for all other work-items to reach the same point in kernel execution.

We can go one step further to improve locality by leveraging the ability of sub-groups to communicate without explicit memory accesses. There are several ways that work could be mapped to sub-groups, and one potential mapping is shown in Figure 4-16: in this mapping, each work-group remains responsible for computing one tile of the output matrix, and each sub-group is therefore responsible for computing one row of its parent work-group's tile. Each sub-group accesses one row of the A matrix, and each work-item in the sub-group accesses one column of the B matrix.

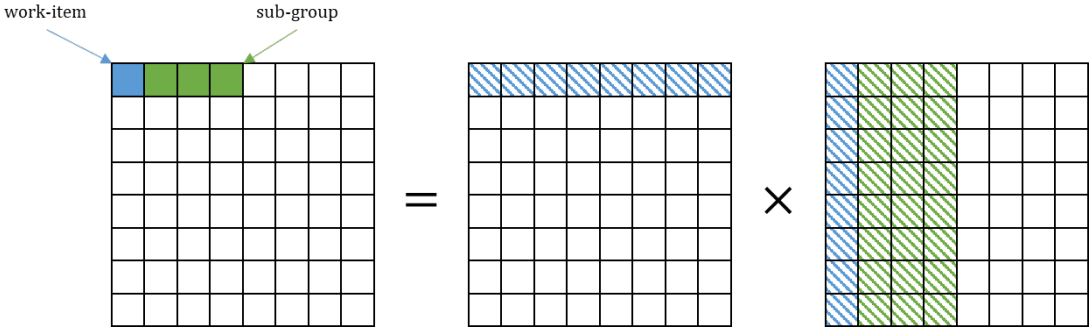


Figure 4-16: Mapping of matrix multiplication to work-groups and work-items.

The corresponding code for this mapping is shown in Figure 4-17. In order to ensure that the number of work-items in a sub-group is the same as the width of the work-group, we have two options: 1) set the work-group size based on the sub-group size chosen by the compiler; or 2) request a particular sub-group size at compile-time. The code shown goes with the latter option, decorating the kernel lambda with the `[[intel::reqd_sub_group_size]]` attribute to guarantee that the compiler selects a sub-group size of 8.

Since we have now guaranteed that each sub-group will access an independent row of the A matrix, we can replace the `a_tile` local accessor with direct communication between work-items. Each sub-group loads a row (with the `load function`), and iterates through the elements of that row using a *shuffle* (specifically, the `broadcast` function).

```
q.submit([&](handler& cgh)
{
  auto a = a_buf.get_access<access::mode::read>(cgh);
  auto b = b_buf.get_access<access::mode::read>(cgh);
  auto c = c_buf.get_access<access::mode::read_write>(cgh);

  auto b_tile = local_accessor<float, 2>(range<2>(B, B), cgh);

  range<2> global(N, N);
  range<2> local(B, B);
  cgh.parallel_for<class matrix_mul>(nd_range<2>(global, local),
                                     [=](nd_item<2> item)
                                     [[intel::reqd_sub_group_size(8)]]
  {
    int j = item.get_global_id(0);
    int i = item.get_global_id(1);

    int li = item.get_local_id(0);
    int lj = item.get_local_id(1);

    intel::sub_group sg = item.get_sub_group();

    for (int kb = 0; kb < N/B; ++kb)
    {
      // Load row of A into private memory
      // Each work-item in the sub-group loads one float
      float a_row = sg.load(a.get_pointer() + j*N + kb*B);

      // Load tile of B matrix into local memory
      b_tile[lj][li] = b[kb*B + lj][i];

      // Wait for load into local memory to complete
      // Barrier synchronizes all work-items in this work-group
      item.barrier(access::fence_space::local_space);

      // Compute matrix multiply using results in private/local memory
      for (int k = 0; k < B; ++k)
      {
        // Extract element k from a_row using a broadcast
        // equivalent to sg.shuffle<float>(a_row, id<1>(k));
        float a_row_k = sg.broadcast<float>(a_row, id<1>(k));

        c[j][i] += a_row_k * b_tile[k][li];
      }

      // Ensure all work-items are done before overwriting local memory
      // Barrier synchronizes all work-items in this work-group
      item.barrier(access::fence_space::local_space);

    }
  });
});
```

Figure 4-17: Tiled matrix multiply using sub-groups and one work-item per element.

Unfortunately, this mapping does not permit us to replace the `b_tile` local accessor with shuffles in a similar fashion, because the reads and writes to `b_tile` are transposed—when loading the B matrix into `b_tile`, each sub-group loads a contiguous row, but when performing the matrix multiplication each sub-group reads a non-

contiguous column. As written, this transpose operation requires communication between different sub-groups, which can only take place via memory.

We could work around this issue by changing our mapping of work to work-items, making each sub-group responsible for a full (8, 8) tile of the output matrix. Such a mapping is interesting because it demonstrates that there need not be a one-to-one mapping between work-items and the dimensionality of the problem space: here, each work-item would be responsible for computing multiple values in the output matrix. The resulting code would likely be quite difficult to read, and its portability would suffer too (since it would rely on a compile-time tile size), but such tricks may be necessary to extract every last ounce of performance from a given device.

Explicit ND-Range Data Parallel Kernels: Important Classes

ND-range data parallel kernels use different classes compared to basic data parallel kernels: `range` is replaced by `nd_range`, and `item` is replaced by `nd_item`. There are also two new classes, representing the different types of groups to which a work-item may belong: functionality tied to work-groups is encapsulated in the `group` class, and functionality tied to sub-groups is encapsulated in the `sub_group` class.

The `nd_range` Class

An `nd_range` represents a grouped execution range using two instances of the `range` class: one denoting the global execution range, and another denoting the local execution range of each work-group. A simplified definition of the `nd_range` class is given in Figure 4-18.

It may be a little surprising that the `nd_range` class does not mention sub-groups at all: the sub-group range is not specified during construction, and cannot be queried. There are several reasons for this omission. First, sub-groups are a low-level implementation detail that can be ignored for many kernels. Second, there are several devices supporting exactly one valid sub-group size, and specifying this size everywhere would be unnecessarily verbose. Finally, sub-groups as a DPC++ feature are an Intel extension to SYCL, and all functionality related to sub-groups is encapsulated in a dedicated class that will be discussed shortly.

```
template <int dimensions = 1>
class nd_item {
public:
  // Construct an nd_range from global and work-group local ranges
  nd_range(range<dimensions> global, range<dimensions> local);

  // Return the global and work-group local ranges
  range<dimensions> get_global_range() const;
  range<dimensions> get_local_range() const;

  // Return the number of work-groups in the global range
  range<dimensions> get_group_range() const;
};
```

Figure 4-18: A simplified definition of the `nd_range` class.

The `nd_item` Class

An `nd_item` is the ND-range form of an `item`, again encapsulating the execution range of the kernel and the item's index within that range. Where `nd_item` differs from `item` is in how its position in the range is queried and represented, as shown by the simplified class definition in Figure 4-19. For example, we can query the item's index in the (global) ND-range using the `get_global_id()` function, or the item's index in its (local) parent work-group using the `get_local_id()` function.

The `nd_item` class also provides functions for obtaining handles to classes describing the group and sub-group that an item belongs to. These classes provide an alternative interface for querying an item's index in an ND-range. We strongly recommend writing kernels using these classes instead of relying on `nd_item` directly: using the `group` and `sub_group` classes is often cleaner; conveys intent more clearly; and is more aligned with the future direction of DPC++.

```
template <int dimensions = 1>
class nd_item {
public:
  // Return the index of this item in the kernel's execution range
  id<dimensions> get_global_id() const;
  size_t get_global_id(int dimension) const;
  size_t get_global_linear_id() const;

  // Return the execution range of the kernel executed by this item
  range<dimensions> get_global_range() const;
  size_t get_global_range(int dimension) const;

  // Return the index of this item within its parent work-group
  id<dimensions> get_local_id() const;
  size_t get_local_id(int dimension) const;
  size_t get_local_linear_id() const;

  // Return the execution range of this item's parent work-group
  range<dimensions> get_local_range() const;
  size_t get_local_range(int dimension) const;

  // Return a handle to the work-group
  // or sub-group containing this item
  group<dimensions> get_group() const;
  sub_group get_sub_group() const;
};
```

Figure 4-19: A simplified definition of the `nd_item` class.

The `group` Class

The `group` class encapsulates all functionality related to work-groups, and a simplified definition is shown in Figure 4-20.

Many of the functions that the `group` class provides each have equivalent functions in the `nd_item` class: for example, calling `group.get_id()` is equivalent to calling `item.get_group_id()`, and calling `group.get_local_range()` is equivalent to calling `item.get_local_range()`. If we're not using any of the work-group functions exposed by the class, should we still use it? Wouldn't it be simpler to use the functions in `nd_item` directly, instead of creating an intermediate `group` object? There is a trade-off here: using `group` requires us to write slightly more code, but that code may be easier to read. For example, consider the code snippet in Figure 4-21: it is clear that `foo` expects to be called by all work-items in the `group`, and it is clear that the `range` returned by `get_local_range()` in the body of the `parallel_for` is the range of the `group`. The same code could very easily be written using only `nd_item`, but it would likely be harder to follow.

```
class group {
public:
  // Return the index of this group in the kernel's execution range
  id<dimensions> get_id() const;
  size_t get_id(int dimension) const;
  size_t get_linear_id() const;

  // Return the number of groups in the kernel's execution range
  range<dimensions> get_group_range() const;
  size_t get_group_range(int dimension) const;

  // Return the number of work-items in this group
  range<dimensions> get_local_range() const;
  size_t get_local_range(int dimension) const;

  // Query Boolean conditions across all work-items in this group
  bool any(bool predicate) const;
  bool all(bool predicate) const;

  // Broadcast a value to all work-items in a group
  template <typename T>
  T broadcast(T x, id<1> local_id) const;

  // Execute parallel algorithms using values from all work-items
  // in this group
  template <typename T, class BinaryOp>
  T reduce(T x, T init, BinaryOp binary_op) const;

  template <typename T, class BinaryOp>
  T exclusive_scan(T x, T init, BinaryOp binary_op) const;

  template <typename T, class BinaryOp>
  T inclusive_scan(T x, BinaryOp binary_op, T init) const;
};
```

Figure 4-20: A simplified definition of the `group` class.

```
void foo(group& g);

cgh.parallel_for(nd_range<1>(global, local), [=](nd_item<1> it) {
  group<1> g = it.get_group();
  range<1> r = g.get_local_range();
  ...
  foo(g);
});
```

Figure 4-21: An example of using the `group` class to improve readability.

Broadcast

The `broadcast` function of the `group` class enables one work-item in a group to share the value of a variable with all other work-items in the group. An example is shown in Figure 4-22.

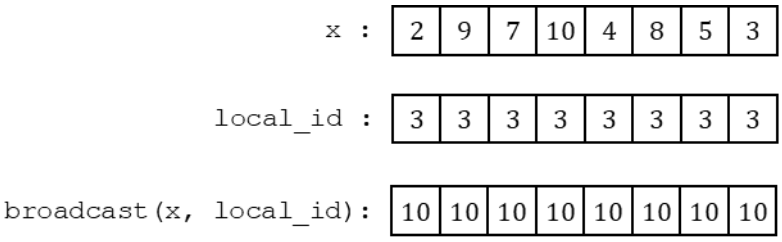


Figure 4-22: An example of the `broadcast` function.

Votes

The `any` and `all` functions of the `group` class (henceforth referred to collectively as "vote" functions) enable work-items to compare the result of a Boolean condition across their group: `any` returns true if the condition is true for at least one work-item in the group; and `all` returns true only if the condition is true for all work-items in the group. A comparison of these two functions for an example input is shown in Figure 4-23.

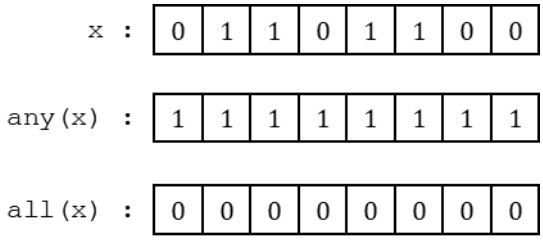


Figure 4-23: A comparison of the `any` and `all` functions.

Collectives

The collective functions provide implementations of closely-related common parallel patterns. It would be straightforward to implement functional versions of these patterns manually by building on top of barriers and local memory, but providing them as library functions instead increases developer productivity and gives implementations the ability

to generate highly optimized code for individual target devices. A comparison of `reduce`, `exclusive_scan`, and `inclusive_scan` is given in Figure 4-24.

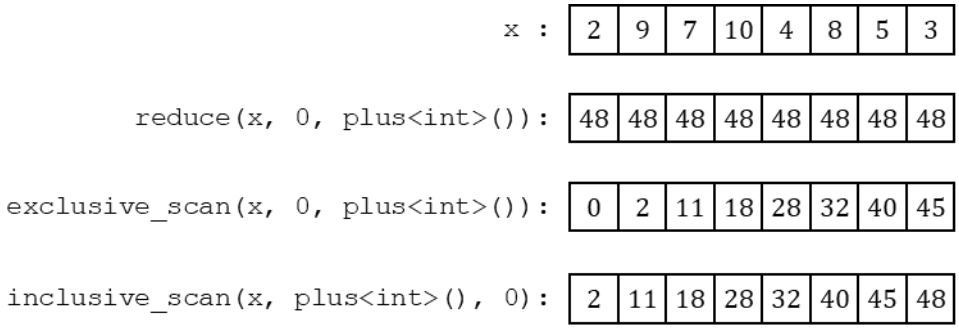


Figure 4-24: A comparison of `reduce`, `exclusive_scan` and `inclusive_scan`.

At the time of writing, the collectives are limited to supporting only primitive data types and the most common reduction operators (`plus`, `minimum` and `maximum`). This is sufficient for many use-cases, but future versions of DPC++ are expected to extend collective support to user-defined types and operators.

```
class sub_group {
public:
  // Return the index of the sub-group
  id<1> get_group_id() const;

  // Return the number of sub-groups in this item's parent work-group
  range<1> get_group_range() const;

  // Return the number of uniform sub-groups in this item's parent work-group
  range<1> get_uniform_group_range() const;

  // Return the index of the work-item in this sub-group
  id<1> get_local_id() const;

  // Return the number of work-items in this sub-group
  range<1> get_local_range() const;

  // Return the maximum number of work-items in any
  // sub-group in this item's parent work-group
  range<1> get_max_local_range() const;

  // Query Boolean conditions across all work-items in this sub-group
  bool any(bool predicate) const;
  bool all(bool predicate) const;

  // Broadcast a value to all work-items in this sub-group
  template <typename T>
  T broadcast(T x, id<1> local_id) const;
```

Figure 4-25: 1 of 2: A simplified definition of the `sub_group` class.

```
  // Execute parallel algorithms using values from
  // all work-items in this sub-group
  template <typename T, class BinaryOp>
  T reduce(T x, T init, BinaryOp binary_op) const;

  template <typename T, class BinaryOp>
  T exclusive_scan(T x, T init, BinaryOp binary_op) const;

  template <typename T, class BinaryOp>
  T inclusive_scan(T x, BinaryOp binary_op, T init) const;

  // One input shuffles
  template <typename T>
  T shuffle(T x, id<1> local_id) const;

  template <typename T>
  T shuffle_down(T x, uint32_t delta) const;

  template <typename T>
  T shuffle_up(T x, uint32_t delta) const;

  template <typename T>
  T shuffle_xor(T x, id<1> value) const;

  // Two input shuffles
  template <typename T>
  T shuffle(T x, T y, id<1> local_id) const;

  template <typename T>
  T shuffle_down(T current, T next, uint32_t delta) const;

  template <typename T>
  T shuffle_up(T previous, T current, uint32_t delta) const;

  // Loads and stores
  template <typename T, access::address_space Space>
  T load(const multi_ptr<T,Space> src) const;

  template <typename T, int N, access::address_space Space>
  vec<T,N> load(const multi_ptr<T,Space> src) const;

  template <typename T, int N, access::address_space Space>
  void store(multi_ptr<T,Space> dst, const T& x) const;

  template <typename T, int N, access::address_space Space>
  void store(multi_ptr<T,Space> dst, const vec<T,N>& x) const;
};
```

Figure 4-26: 2 of 2: A simplified definition of the `sub_group` *class.*

The `sub_group` Class

The `sub_group` class encapsulates all functionality related to sub-groups, and a simplified definition is shown in Figure 4-25 and Figure 4-26. Unlike with work-groups, the `sub_group` class is the only way to access sub-group functionality; none of its functions are duplicated in `nd_item`. The queries in the `sub_group` class are all interpreted relative to the calling work-item: for example, `get_local_id()` returns the local index of the calling work-item within its sub-group.

The code sample in Figure 4-27 demonstrates the most basic usage of sub-groups, specifically: obtaining a handle to the sub-group an item belongs to (`get_sub_group()`); identifying an item's location in its sub-group (`get_local_id()`); and querying the size of the sub-group (`get_local_range()`). Possible output from this application running on an Intel Iris Graphics 540 is given in Figure 4-28.

There are two important things to note in the output. First, the print statements from each sub-group do not appear in order (and are expected to change between runs). As with work-groups, we should not assume that sub-groups will execute in any particular order. Second, there is one sub-group with a smaller size than the others. If the size of a work-group does not divide evenly by the sub-group size chosen by an implementation, then the size of the last sub-group will be equal to the remainder (e.g. 63 mod 8 = 7).

```
cgh.parallel_for(nd_range<1>(global, local), [=](nd_item<1> it) {
  // Get handle to this item's sub-group
  intel::sub_group sg = it.get_sub_group();

  // Print only once per sub-group
  if (sg.get_local_id() == 0) {
    out << "Hello from sub-group " << sg.get_group_id()[0] << " of "
        << sg.get_group_range() << "!  Sub-group size = "
        << sg.get_local_range()[0] << endl;
  }
});
```

Figure 4-27: Getting a sub-group handle.

```
Hello from sub-group 0 of 8!  Sub-group size = 8
Hello from sub-group 2 of 8!  Sub-group size = 8
Hello from sub-group 1 of 8!  Sub-group size = 8
Hello from sub-group 4 of 8!  Sub-group size = 8
Hello from sub-group 3 of 8!  Sub-group size = 8
Hello from sub-group 5 of 8!  Sub-group size = 8
Hello from sub-group 7 of 8!  Sub-group size = 7
Hello from sub-group 6 of 8!  Sub-group size = 8
```

Figure 4-28: Possible output from the code in Figure 4-27.

Shuffles

One of the most useful features of sub-groups is the ability to communicate directly between individual work-items without explicit memory operations. In many cases, these *shuffle* operations enable us to remove work-group local memory usage from our kernels and/or to avoid unnecessary repeated accesses to global memory. There are several flavors of these shuffle functions available.

The most general of the shuffle functions is called `shuffle`, and as shown in Figure 4-29 it allows for arbitrary communication between any pair of work-items in the sub-group. This generality may come at a cost, however, and we strongly encourage making use of the more specialized shuffle functions wherever possible.

The `shuffle_up` and `shuffle_down` functions effectively *shift* the contents of a sub-group by a fixed number of elements in a given direction, as shown in Figure 4-30. Shifting can be useful for parallelizing loops with loop-carried dependencies, or implementing common algorithms like exclusive/inclusive scans. The two-input versions are especially useful, allowing a known value to be shifted into the end of a sub-group.

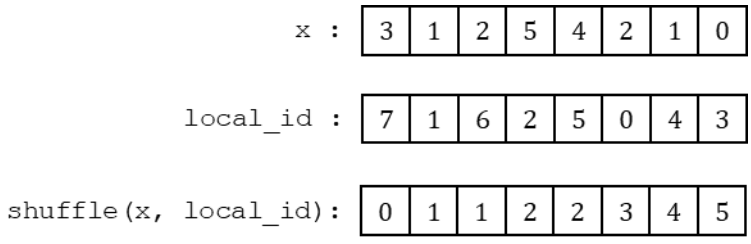


Figure 4-29: A generic `shuffle` used to sort the `x` values of an 8-item sub-group using pre-computed permutation indices.

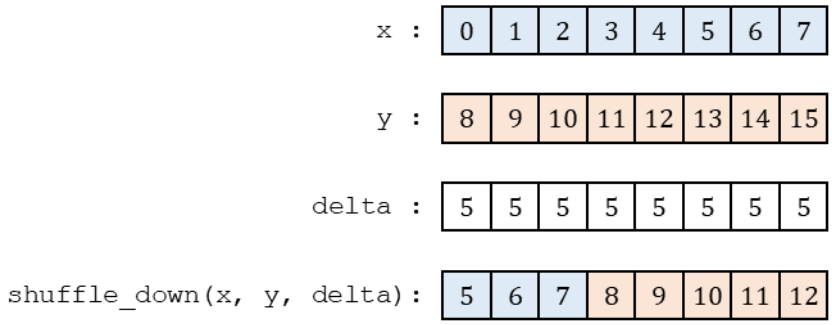


Figure 4-30: A two-input `shuffle_down` used to shift the `x` values of an 8-item sub-group by 5 items. The value returned to the last 5 work-items in the sub-group are taken from `y`.

The `shuffle_xor` function swaps the values of two work-items, as specified by the result of an XOR operation applied to the work-item's sub-group local ID and a fixed constant. As shown in Figure 4-31 and Figure 4-32, several common communication patterns can be expressed using an XOR: for example, swapping pairs of neighboring values, or reversing the sub-group values.

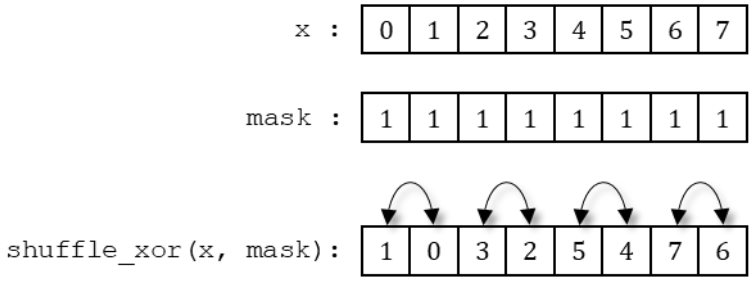


Figure 4-31: A `shuffle_xor` used in an 8-item sub-group to swap neighboring pairs of `x`.

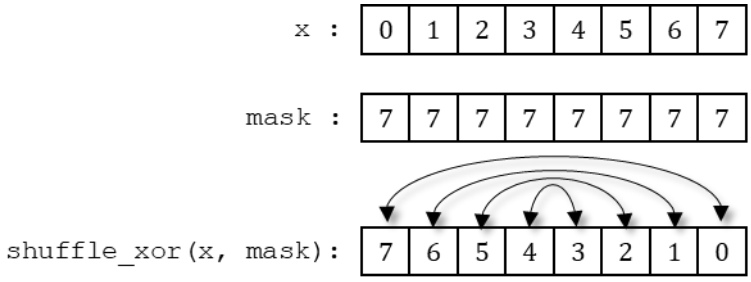


Figure 4-32: A `shuffle_xor` used in an 8-item sub-group to reverse the values of `x`.

```
cgh.parallel_for(nd_range<1>(N, N), [=](nd_item<1> it)
{
  // Get handle to this item's sub-group
  intel::sub_group sg = it.get_sub_group();

  // Load the value for this item
  uint32_t i = it.get_global_id(0);
  float centre = input[i];

  // Obtain the left and right values directly from neighboring work-items
  // If the end of the sub-group is reached, substitute a 0
  float left = sg.shuffle_up(0.0f, centre, 1);
  float right = sg.shuffle_down(centre, 0.0f, 1);

  float result = (left + centre + right) / 3.0f;
  output[i] = result;
});
```

Figure 4-33: Using sub-group shuffle functions to optimize loads in a stencil.

The code sample in Figure 4-33 demonstrates the use of `shuffle_up` and `shuffle_down` for a simple one-dimensional stencil. Once each work-item has loaded its center value, the values to the left and right are known to have been loaded by other work-items. Therefore, rather than loading the left and right values again from memory, we can use shuffles to access them directly. Note that this example assumes that the total domain is very small (the size of the sub-group) and that `0.0f` is an appropriate boundary value. Real-life use-cases may require additional logic to ensure correct values at the boundaries.

Broadcast, Vote and Collectives

The behavior of these `sub_group` functions are almost exactly the same as their equivalents in the `group` class, but they deserve additional attention because they may enable aggressive optimizations in certain compilers. For example, a compiler may be able to reduce register usage for variables that are broadcast to all work-items in a sub-group, or may be able to reason about control-flow divergence based on usage of the `any` and `all` functions. An example of such optimizations being used in practice is shown in Figure 4-34.

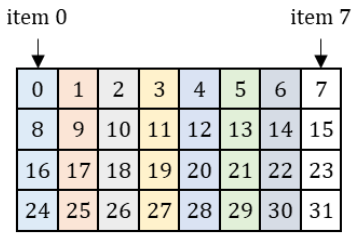
```
cgh.parallel_for(nd_range<1>(N, N), [=](nd_item<1> it) {
  // Get handle to this item's sub-group
  intel::sub_group sg = it.get_sub_group();

  // Load the (index, value) pair for this item
  int32_t i = it.get_global_id(0);
  int idx = index[i];
  int x = values[i];

  // All work-items in the sub-group take the same branch below
  // If all elements are the same, use a sub-group reduction and one atomic
  if (sg.all(idx == sg.broadcast<int>(idx, 0))) {
    int sum = sg.reduce(x, intel::plus<>());
    if (sg.get_local_id() == 0) {
      histogram[idx].fetch_add(sum);
    }
  }

  // Otherwise, use an atomic for each work-item in the sub-group
  // A simple "else" would be sufficient -- extra condition here used to
  // demonstrate "any"
  else if (sg.any(idx != sg.broadcast<int>(idx, 0))) {
    histogram[idx].fetch_add(x);
  }
});
```

Figure 4-34: Using sub-group collective functions to optimize atomic accesses.

Loads and Stores

The sub-group load and store functions serve two purposes: 1) informing the compiler that all work-items in the sub-group are loading contiguous data starting from the same (uniform) location in memory; and 2) enabling us to request optimized loads/stores of large amounts of contiguous data.

In a SPMD programming context like an ND-range `parallel_for`, it may not be clear to the compiler how addresses computed by different work-items relate to one another. For example, as shown in Figure 4-35, accessing a contiguous block of memory from indices [0, 32) appears to have a strided access pattern from the perspective of each work-item.




```
for (int b = 0; b < 4; ++b)
{
  int offset = b * sg.get_max_local_range();
  array[offset + sg.get_local_id()];
  ...
}
```

Figure 4-35: Memory access pattern of an 8-wide sub-group accessing four contiguous blocks.

Some architectures include dedicated hardware to detect when work-items in a sub-group access contiguous data and combine their memory requests, while other architectures require this to be known ahead of time and encoded in the load/store instruction. Sub-group loads and stores are not required for correctness on any platform but may improve performance on some platforms, and should therefore be considered as an optimization hint.

```
cgh.parallel_for<class vector_add>(range<1>(N), [=](item<1> it)
{
  intel::sub_group sg = it.get_sub_group();
  uint32_t sg_id = sg.get_group_id();
  uint32_t sg_size = sg.get_max_local_range();

  // Compute base pointers shared by all work-items in the sub-group
  int* a_base = a.get_pointer() + sg_id * sg_size;
  int* b_base = b.get_pointer() + sg_id * sg_size;
  int* c_base = c.get_pointer() + sg_id * sg_size;

  // Load four contiguous blocks of sub-group size
  // Each work-item loads four elements, with a stride of the sub-group size
  float4 a4 = sg.load<4>(a_base);
  float4 b4 = sg.load<4>(b_base);

  // Compute and store four elements per work-item
  float4 c4 = a4 + b4;
  sg.store<4>(c_base, c4);
});
```

Figure 4-36: Using sub-group loads and stores to access four contiguous blocks of sub-group size.

Hierarchical Parallel Kernels

Hierarchical data parallel kernels offer an experimental alternative syntax for expressing kernels in terms of work-groups and work-items, where each level of the hierarchy is programmed using a nested invocation of the `parallel_for` function. This *top-down* programming style is intended to be similar to writing parallel loops, and may feel more familiar than the *bottom-up* programming style used by the other two kernel forms.

One complexity of hierarchical kernels is that each nested invocation of `parallel_for` creates a separate SPMD context; each scope defines a new "program" that should be executed by all parallel workers associated with that scope. This complexity requires compilers to perform additional analysis, and can complicate code generation for

some devices; compiler technology for hierarchical parallel kernels on some platforms is still relatively immature, and performance will be closely tied to the quality of a particular compiler implementation. Mapping nested parallelism to accelerators is a challenge that is not unique to DPC++: the parallel STL already supports nested algorithms with different execution policies; there is an ISO C++ proposal to associate execution resources with levels of nesting using executors; and languages outside of C++ (e.g. OpenMP) have long supported nested parallel constructs. This topic is the subject of much interest and research, and we expect that the performance of hierarchical parallelism will improve over time.

Since the relationship between a hierarchical data parallel kernel and the code generated for a specific device is compiler-dependent, hierarchical kernels should be considered a more *descriptive* construct than explicit ND-range kernels. However, since hierarchical kernels retain the ability to control the mapping of work to work-items and work-groups, they remain more *prescriptive* than basic kernels.

Hierarchical Data Parallel Kernels: Execution Model

The underlying execution model of hierarchical data parallel kernels is the same as the execution model of explicit ND-range data parallel kernels. Individual kernel instances are still mapped to work-items, sub-groups and work-groups, with identical semantics and execution guarantees.

Hierarchical Data Parallel Kernels: Syntax

In hierarchical kernels, the `parallel_for` function is replaced by the `parallel_for_work_group` and `parallel_for_work_item` functions, which correspond to work-group and work-item parallelism respectively. At the time of writing, there is no access to sub-groups within hierarchical kernels (i.e. there is no equivalent of a `parallel_for_sub_group` function), but this is expected to be addressed by future versions of DPC++.

DPC++ compilers must guarantee that hierarchical kernels execute as if code in a `parallel_for_work_group` scope is executed only once per work-group, and as if variables allocated in a `parallel_for_work_group` scope are visible to all work-items (i.e. they are allocated in work-group local memory). However, the code that is actually generated will be highly dependent on the results of compiler analysis: an optimizing compiler may be able to prove that it is safe to execute work-group code

redundantly on all work-items, or that it is safe to allocate a work-group variable in work-item private memory (e.g. because its value is constant).

As shown in Figure-37, kernels expressed using hierarchical parallelism are very similar to ND-range kernels. We should therefore view hierarchical parallelism primarily as a productivity feature; it doesn't expose any functionality that isn't already exposed via ND-range kernels, but it may improve the readability of our code and/or reduce the amount of code that we have to write.

```
range<2> num_groups(N/B, N/B);
range<2> group_size(B, B);
cgh.parallel_for_work_group<class matrix_mul>(num_groups, group_size,
[=](group<2> grp)
{
  int jb = grp.get_id(0);
  int ib = grp.get_id(1);
  grp.parallel_for_work_item([&](h_item<2> item)
  {
    int j = jb*B + item.get_local_id(0);
    int i = ib*B + item.get_local_id(1);
    for (int k = 0; k < N; ++k)
    {
      c[j][i] += a[j][k] * b[k][i];
    }
  });
});
```

Figure-37: A naïve matrix multiplication kernel expressed with hierarchical parallelism.

Like `parallel_for`, `parallel_for_work_group` is a member function of the `handler` class and can only be called inside of a command-group scope. It is important to note that the ranges passed to the function specify the number of groups and an optional group size, **not** the total number of work-items and group size as was the case for ND-range `parallel_for`. The kernel function accepts an instance of the `group` class, reflecting that the outer scope is associated with work-groups rather than individual work-items.

`parallel_for_work_item` is a member function of the `group` class, and can only be called inside of a `parallel_for_work_group` scope. In its simplest form, its only argument is a function accepting an instance of the `h_item` class, and the number of times that the function is executed is equal to the number of work-items requested per work-group; the function is executed once per *physical* work-item. An additional productivity feature of `parallel_for_work_item` is its ability to support a *logical* range, which is passed as an additional argument to the function. When a logical range is

specified, each physical work-item executes zero or more instances of the function; the logical items of the logical range are assigned round-robin to physical work-items, as shown in Figure 4-38.

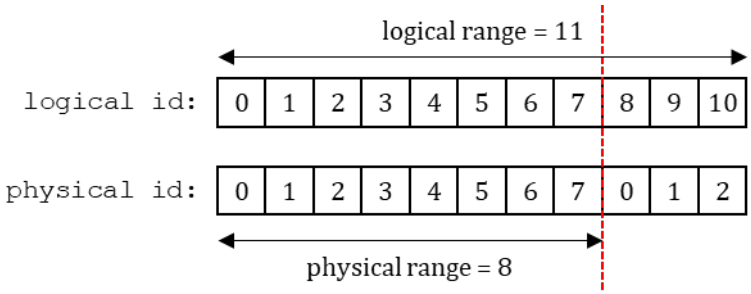


Figure 4-38: A logical range of size 11 mapped to a physical range of size 8. The first three work-items are assigned two instances of the function, and all other work-items are assigned only one.

As shown in Figure 4-39, combining the optional group size of `parallel_for_work_group` with the logical range of `parallel_for_work_item` gives an implementation the freedom to choose work-group sizes without sacrificing our ability to conveniently describe the execution range using nested parallel constructs. Note that the amount of work performed per group remains the same as in Figure-37, but that the amount of work has now been separated from the physical work-group size.

```
range<2> num_groups(N/B, N/B);
range<2> group_size(B, B);
cgh.parallel_for_work_group<class matrix_mul>(num_groups, [=](group<2> grp)
{
  int jb = grp.get_id(0);
  int ib = grp.get_id(1);
  grp.parallel_for_work_item(group_size, [&](h_item<2> item)
  {
    int j = jb*B + item.get_logical_local_id(0);
    int i = ib*B + item.get_logical_local_id(1);
    for (int k = 0; k < N; ++k)
    {
      c[j][i] += a[j][k] * b[k][i];
    }
  });
});
```

Figure 4-39: A naïve matrix multiplication kernel expressed with hierarchical parallelism and a logical range.

```
cgh.parallel_for_work_group<class matrix_mul>(range<2>(N/B, N/B),
                                               [=](group<2> grp)
{
  // Work-group local arrays can be allocated at work-group scope
  // if their sizes are known at compile-time
  float a_tile[B][B];
  float b_tile[B][B];

  int jb = grp.get_id(0);
  int ib = grp.get_id(1);

  for (int kb = 0; kb < N/B; ++kb)
  {
    // Load tiles of A and B matrices into local memory
    // Implicit barrier ensures all code at work-group scope has executed
    grp.parallel_for_work_item(range<2>(B, B), [&](h_item<2> item)
    {
      int lj = item.get_logical_local_id(0);
      int li = item.get_logical_local_id(1);

      int j = jb*B+lj;
      int i = ib*B+li;

      a_tile[lj][li] = a[j][kb*B + li];
      b_tile[lj][li] = b[kb*B + lj][i];
    });

    // Implicit barrier ensures load into local memory is complete

    // Compute matrix multiply using results in local memory
    grp.parallel_for_work_item(range<2>(B, B), [&](h_item<2> item)
    {
      int lj = item.get_logical_local_id(0);
      int li = item.get_logical_local_id(1);

      int j = jb*B+lj;
      int i = ib*B+li;

      for (int k = 0; k < B; ++k)
      {
        c[j][i] += a_tile[lj][k] * b_tile[k][li];
      }
    });

    // Implicit barrier ensures work-item scope has executed for all items
  }
});
```

Figure 4-40: A tiled matrix multiplication kernel using work-group local memory expressed with hierarchical parallelism and a logical range.

The code in Figure 4-40 shows how to extend Figure 4-39 to use work-group local memory in hierarchical parallelism. There are two notable differences between this kernel and the version we wrote as an ND-range kernel. The first difference is that we can declare the work-group local memory inside of the kernel (without using a local accessor) as long as the amount of local memory is known at compile-time. The second difference is that the two phases of the kernel each has a corresponding `parallel_for_work_item`

scope and its own logical range, with work-group barriers occurring implicitly between phases.

Making `B` a compile-time parameter and using logical ranges effectively separates the block size from the work-group size, allowing us to tune both separately. While it would have been possible to express this same pattern using an ND-range kernel, manually distributing the iterations of a loop over the work-items in each work-group, it would not have been as simple.

Hierarchical Data Parallel Kernels: Important Classes

Hierarchical data parallel kernels re-use the `group` class from ND-range data parallel kernels, but replace `nd_item` with `h_item`. A new `private_memory` class is introduced to provide tighter control over allocations occurring in `parallel_for_work_group` scope.

The `h_item` Class

An `h_item` is a variant of `item` that is only available within a `parallel_for_work_item` scope. It provides a similar interface to an `nd_item`, with one notable difference: the item's index can be queried relative to the physical execution range of a work-group (with `get_physical_local_id()`) or the logical execution range of a `parallel_for_work_item` construct (with `get_logical_local_id()`).

```
template <int dimensions>
class h_item {
public:
  // Return item's index in the kernel's execution range
  id<dimensions> get_global_id() const;
  range<dimensions> get_global_range() const;

  // Return the index in the work-group's execution range
  id<dimensions> get_logical_local_id() const;
  range<dimensions> get_logical_local_range() const;

  // Return the index in the logical execution range of the parallel_for
  id<dimensions> get_physical_local_id() const;
  range<dimensions> get_physical_local_range() const;
}
```

Figure 4-41: Simplified definition of the `h_item` class.

```
template <typename T, int dimensions = 1>
class private_memory
{
  public:
  // Construct a private variable for each work-item in the group
  private_memory(const group<dimensions>&);

  // Return the private variable associated with this work-item
  T& operator(const h_item<dimensions>&);
};
```

Figure 4-42: Simplified definition of the `private_memory` *class.*

The `private_memory` Class

The combination of `parallel_for_work_group` scope implying work-group local variables and `parallel_for_work_item` accepting a logical range complicates the declaration of variables that are intended to be private to a work-item: variables declared at the outer scope are only private if the compiler can prove it is safe to make them so; and variables declared at the inner scope are private to a logical work-item rather than a physical one. It is impossible using scope alone for us to convey that a variable is intended to be private for each physical work-item.

To see why this is a problem, let's refer back to our matrix multiplication kernels in Figure 4-39 and Figure 4-40. The `ib` and `jb` variables are declared at `parallel_for_work_group` scope, and a compiler may therefore choose to allocate them in work-group local memory! There's a good chance that an optimizing compiler would not make this mistake, because the variables are read-only and their value is simple enough to compute redundantly on every work-item, but the language makes no guarantees. If we want to be certain that a variable is declared in work-item private memory, we must wrap the variable declaration in an instance of the `private_memory` class, shown in Figure 4-42.

For example, if we were to rewrite our matrix multiplication kernel using the `private_memory` class, we would define the variables as `private_memory<int> ib(grp)`, and each access to these variables would become `ib[item]`. In this case, using the `private_memory` class actually results in code that is harder to read, and manually re-computing the values at `parallel_for_work_item` scope is clearer.

Our recommendation is to only use the `private_memory` class if a work-item private variable is used across multiple `parallel_for_work_item` scopes within the same `parallel_for_work_group`, it is too expensive to compute repeatedly, or its computation has side effects that prevent it from being computed redundantly. Otherwise,

we should rely on the abilities of modern optimizing compilers by default, and declare variables at `parallel_for_work_item` scope when their analysis fails (remembering to also file a bug report with the compiler vendor).

Choosing a Kernel Form

Choosing between the different kernel forms is largely a matter of personal preference, and we expect that preference to be heavily influenced by the amount of prior experience you have had with other parallel programming models and languages.

The other main reason to choose a particular kernel form is that it is the only form to expose certain functionality required by your kernel. Unfortunately, identifying which functionality you will need *a priori* may be complicated, especially while you are still unfamiliar with the different kernel forms and their interaction with various DPC++ classes.

We have constructed two guides based on our own experience with DPC++ in order to help you navigate this complex space. These guides should be considered rules-of-thumb, and are definitely not intended to replace your own experimentation — the best way to choose between the different kernel forms will always be to spend some time writing in each of them, in order to learn which form is the best fit for your application and development style.

The first guide is the flowchart in Figure 4-43, which selects a kernel form based on:

1. Whether you have previous experience with parallel programming.
2. Whether you are writing a new code from scratch, or are porting an existing parallel program written in a different language.
3. Whether your kernel is embarrassingly parallel, already contains nested parallelism, or re-uses data between different instances of the kernel function.
4. Whether you are writing a new kernel in SYCL to maximize performance, to improve the portability of your code, or because it provides a more productive means of expressing parallelism than lower-level languages.

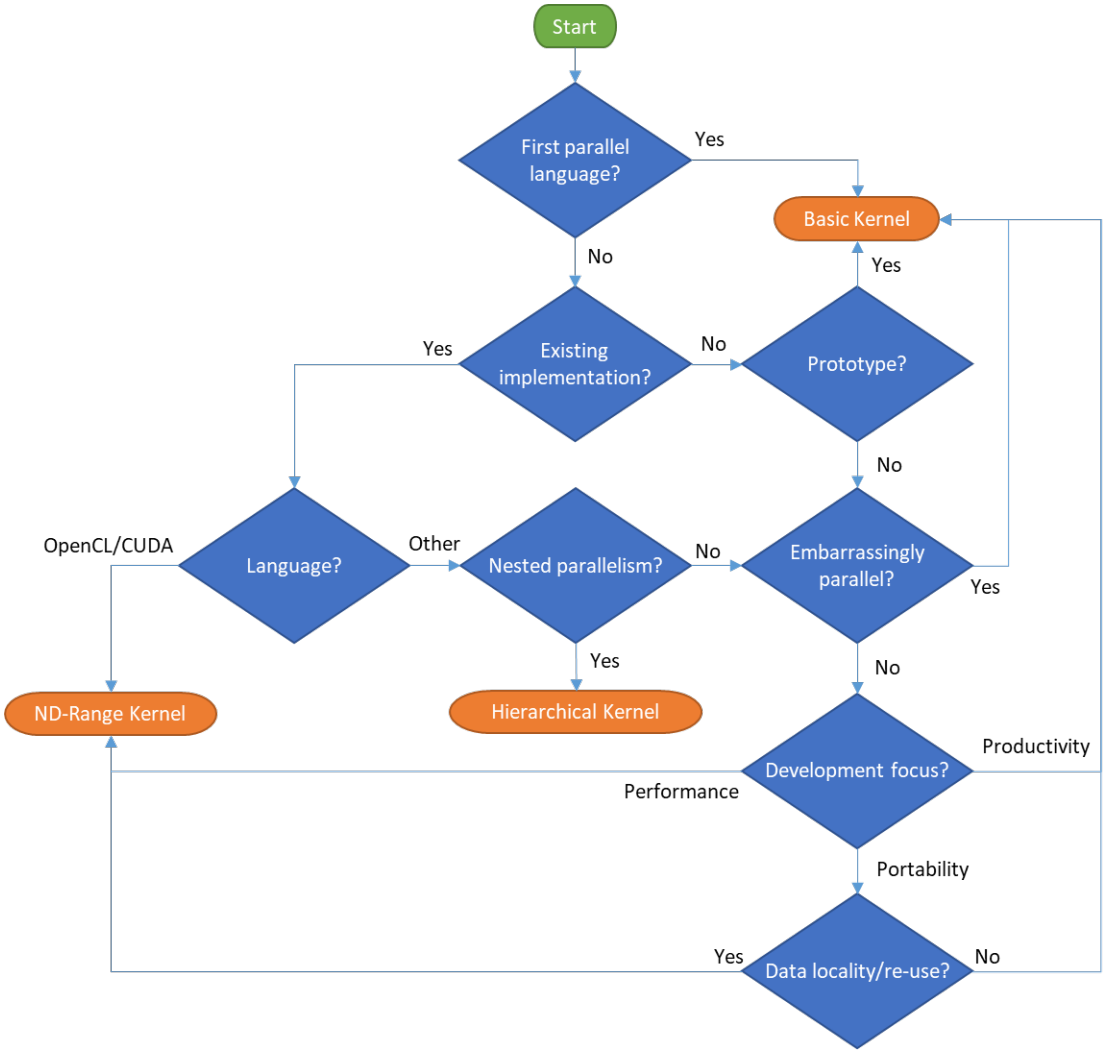


Figure 4-43: A flowchart designed to help you choose the right form for your kernel.

The second guide is the table in Figure 4-44, which summarizes the functionalities of DPC++ that are exposed to each of the kernel forms. It is important to note that this table reflects the state of DPC++ at the time of writing, and that the features available to each kernel form should be expected to change as the language evolves. However, we expect the basic trend to remain the same: basic data parallel kernels will not expose locality-aware features; explicit ND-range kernels will expose all performance-enabling features of DPC++; and hierarchical kernels will lag behind explicit ND-range kernels in exposing features, but their expression of those features will use higher-level abstractions.

Feature	Basic Kernel	ND-Range Kernel	Hierarchical Kernel
Work-group Local Memory	No	Yes	Yes
Work-group Barriers	No	Yes	Yes
Work-group Functions (e.g. `scan`, `reduce`)	No	Yes	No
Sub-groups	No	Yes	No

Figure 4-44: A summary of the features available to each kernel form.

Summary

In this chapter, we have introduced the basics of expressing parallelism in DPC++ and discussed the strengths and weaknesses of each approach to writing data-parallel kernels.

DPC++ is a rich language supporting many forms of parallelism, and we hope that we have provided enough information to prepare readers to dive in and start coding!

We have only scratched the surface of DPC++'s feature set, and a deeper dive into many of the concepts and classes introduced in this chapter are forthcoming: the usage of work-group local memory and barriers will be expanded upon in Chapter 9; different ways of defining kernels besides using lambda expressions will be discussed in Chapter 10; detailed mappings of the ND-range execution model to specific hardware will be explored in Chapters 13, 14 and 15; and best practices for expressing common parallel patterns using DPC++ will be presented in Chapter TBD.

For More Information

- SYCL Specification, Version 1.2.1, Section 3.4 and 3.6
  https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
- A Unified Executors Proposal for C++
  http://wg21.link/p044

FOR THIS BOOK PREVIEW (CHAPTERS 1-4):
ERRATA, NOTES, DOWNLOADS, FEEDBACK, ETC.

Please check our "preview book" website for information including errata, updates, and downloads (includes all sample code): https://tinyurl.com/book-dpcpp

We will also list some additional resources as they become available.

Your feedback is welcome. You can email James Reinders at dpc++@jamesreinders.com with any suggestions, encouragement, criticism, or questions that you may have. James will be sure to share any feedback that you send with all the authors.

Of course – watch for the full book, by mid-2020, available from Apress (no charge for PDF for the completed book, print copies will be available too).

https://tinyurl.com/book-dpcpp

FOR THIS BOOK PREVIEW (CHAPTERS 1-4):
ERRATA, NOTES, DOWNLOADS, FEEDBACK, ETC.

Please check our "preview book" website for information including errata, updates, and downloads (includes all sample code): https://tinyurl.com/book-dpcpp

We will also list some additional resources as they become available.

Your feedback is welcome. You can email James Reinders at dpc++@jamesreinders.com with any suggestions, encouragement, criticism, or questions that you may have. James will be sure to share any feedback that you send with all the authors.

Of course – watch for the full book, by mid-2020, available from Apress (no charge for PDF for the completed book, print copies will be available too).

https://tinyurl.com/book-dpcpp