Quick Guide for DPC++

The content of this guide is written by adopting from the following sources:
1- Intel® oneAPI Programming Guide
2- Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Content:

Chapter1: Introduction

Data Parallel C++ is an extension of C++, incorporating SYCL and other new features. SYCL is an industry led for adding data parallelism to C++ for heterogeneous systems. To avoid writing “SYCL and DPC++” over and over, consider these: we speak of DPC++ in an inclusive manner, and all SYCL features are always DPC++ features as well.

What is Data-Parallel Programming?

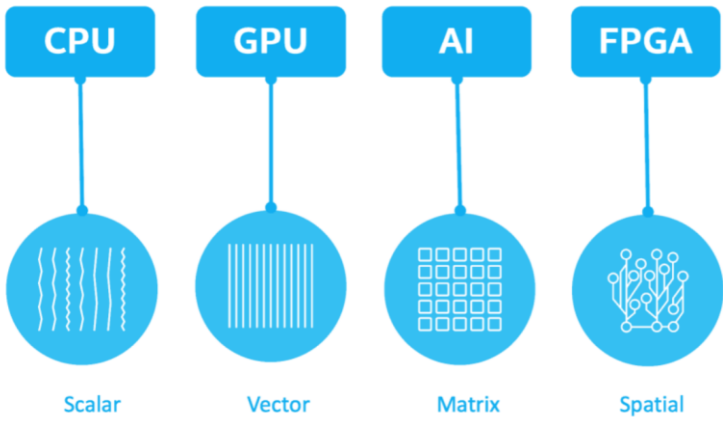
There are two approach for parallel programming: Task-parallel and Data parallel. In Data-parallel programming, Data is operated on in parallel by a collection of processing elements. Each processing element is hardware capable of some computation on the data. These processing elements may exist on a single device, or many devices in our computer systems. We specify our code to work on our data in the form of a kernel. Therefore, we should 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.

When we are 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. As a Parallel programmer, you should use elements of both data-parallel and task-parallel programming. Therefore, the features of DPC++ supports both parallel programming styles.

What are Heterogeneous Systems?

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) is a heterogeneous system. Today, the collection of devices includes 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++.

Accelerator architectures can be bucketed into broad categories that impact how we need to think about the programming model, and the types of workloads that an accelerator will perform well on. Following figure utilizes a taxonomy of Scalar, Vector, Matrix, and Spatial:



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.

Platform Model:

The platform model, used by 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. 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.

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. 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.

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.

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 the following figure:

```
#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

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++. DPC++ also allow for compilation of the device code to be completed at runtime for more flexibility. The single source nature 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. 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.

A single application may use multiple accelerators, but with specific purposes for each. In the following Figure, 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.

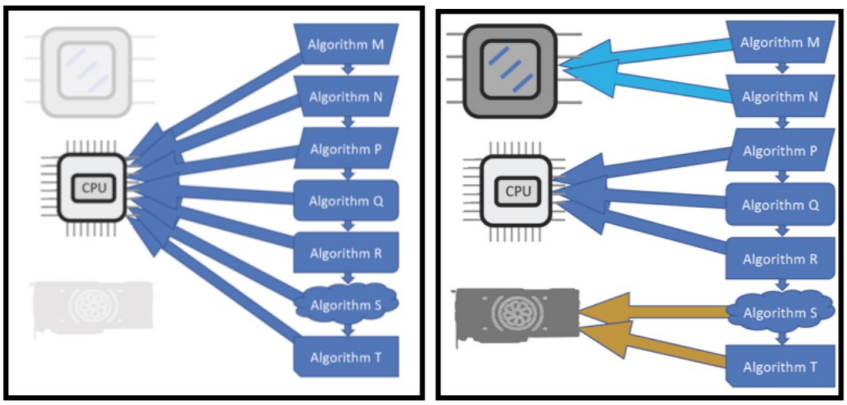


Figure 1-4: Left: Fallback on Host (CPU) if no device present, Right: Offload different algorithms to the devices with best match for the needs of each algorithm.

In the introduction, we provided a base rationale for DPC++. 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).

Chapter 2: Where Code Executes

In DPC++, the mechanisms used to select the destinations for code execution are a fundamental part of the solution. Here, we describe where code can execute, introduce when it will execute, and describe the mechanisms used to control the locations of execution.

DC++ applications contain C++ host code which is executed by the CPU(s) on which the operating system has launched the application. Host code is the backbone of the application in that it defines and controls the execution of compute on available devices. The host code orchestrates data movement and compute offload to devices, but can also perform compute-intensive work itself.

Devices correspond to accelerators or processors that are conceptually independent from the CPU that is executing host code. The host processor runs native C++ code, while devices run device code. Queues are the mechanism through which host code submits work to a device for future execution. Device code executes asynchronously from the host code, unless the developer explicitly ties the two together.

Choosing a device on which to execute

To explore the mechanisms that control where device code executes, you can consider the following use cases:

1- Running device code “somewhere”, with the developer not caring which device it ends up being.
2- Explicitly running device code on the host device, which is typical used for debugging.
3- Dispatching device code to another accelerator device.
4- Dispatching device code to a heterogeneous set of devices.
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.

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. 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.

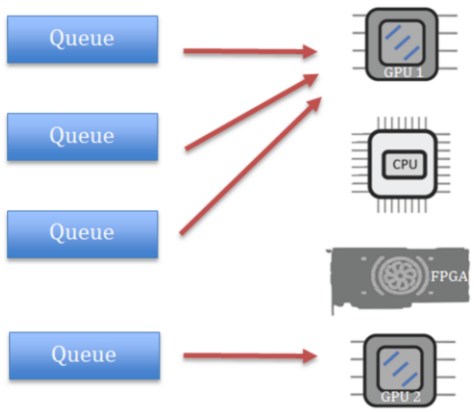


Figure 2-4: Multiple queues can be bound to a single device, and the combination of work submitted to such queues is combined onto the device. The converse is not allowed, though, and a single queue cannot be bound to more than one device, because it would lead to ambiguity in where device code will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

Following Figure 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, 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.

```
#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
```

The 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 follow:

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

There are a few categories of accelerator devices defined: CPU devices, GPUs, Accelerators (FPGAs). 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":

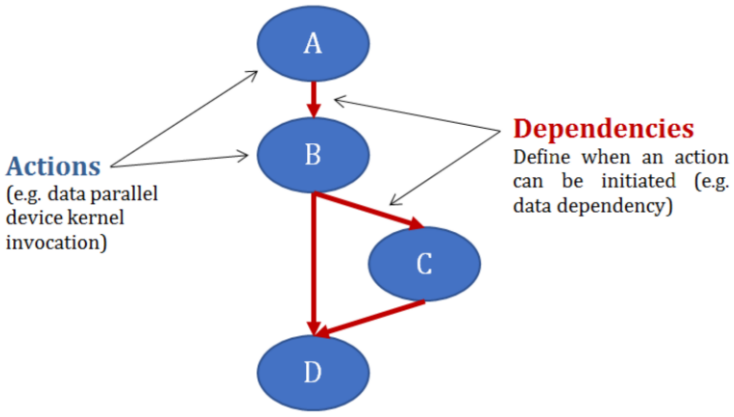
`intel::fpga_selector`	Select device that identifies itself as an FPGA

multiple queues can be constructed in the DPC++ application.

```
#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
```

Execution Model:

A fundamental concept in the DPC++ execution model is a graph of nodes (Directed Acyclic Graph (DAG)). Each node in this graph contains an action to be performed on a device. The nodes 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.



As we said, DPC++ applications typically contain a combination of both host code and device code. We introduce some of the basic DPC++ work dispatch constructs, with the goal to help readers understand and identify the division between device code, and the host 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.

Device dispatch and memory copy mechanisms:

A command group can be 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)

Following figure 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.

Only a single handler method from above 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.

```
q.submit([&](handler& h) {
    //COMMAND GROUP CODE
});
```

In the below figure, the code passed as the final argument to the *parallel_for* member of the sycl::handler class, defined as a lambda, is the device code to be executed on a device. The *parallel_for* in this case is the critical construct that is used to distinguish device code from host code.

```
queue my_queue;
std::cout << "Choose device: " <<
my_queue.get_device().get_info<info::device::name>() << "\n";

buffer<int, 1> my_buf { range<1> { num } };

my_queue.submit(
  [&](handler& cgh)
  {
    auto A =
      my_buf.get_access<access::mode::discard_write>(cgh);

    cgh.parallel_for<class my_kernel>(range<1>{num},
      [=](id<1> idx)
      {
        A[idx] = idx[0];
      }
  );
});
```

Host code

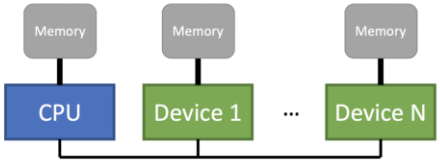
Immediate code to set up DAG node requirements and the work to perform.

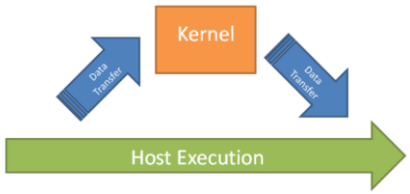
Device code run in the future when dependencies are met.

Chapter3: Data Management

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. Accelerator devices often have their own attached memories that cannot be directly accessed from the host. Hence, we introduce Unified Shared Memory and the Buffer abstractions for data management.

We refer to accesses to a directly attached memory to an accelerator as local accesses. Accesses to another device's memory are remote accesses. Remote accesses tend to be slower than local accesses.





Managing multiple memories can be accomplished, broadly, in two ways: explicitly by the programmer or implicitly by the runtime. Consider 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. This is explicit. In implicit, 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.

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. 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.

Buffers:

DPC++ does not require the programmer to manage the data copies. By creating Buffers and Accessors, DPC++ ensures that the data is available to host and device without any programmer effort. DPC++ also allows the programmer explicit control over data movement when it is necessary to achieve best performance. Buffers encapsulate data in a SYCL application across both devices and host. Accessors is the mechanism to access buffer data.

The buffer<Type, dims> class is generic over the element type and the number of dimensions, which can be one, two or three. When passed a raw pointer, the buffer(T* ptr, range size) constructor takes ownership of the memory it has been passed. This means that we absolutely cannot use that memory ourselves while the buffer exists, which is why we begin **a C++ scope**. At the end of their scope, the buffers will be destroyed and the memory returned to the user. The size argument is a range<dims> object, which has to have the same number of dimensions as the buffer and is initialized with the number of elements in each dimension.

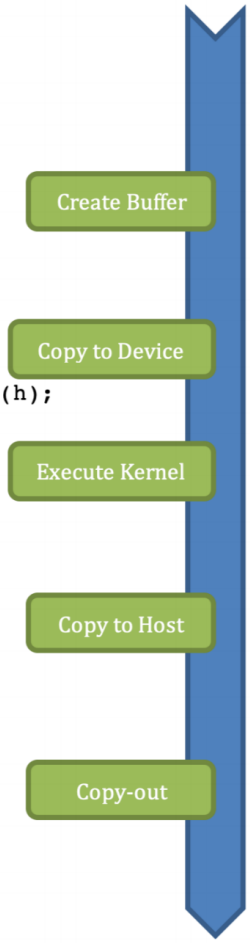
A **range** represents a one-, two- or three-dimensional range. The dimensionality of a range is a template argument and must therefore be known at compile-time, but its size in each dimension is dynamic and is passed to the constructor at run-time.

To use buffer, 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 way does not initialize its data. Buffers may not be directly accessed by the host and device. 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(). 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 below:

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.

In the code example shown in below, 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*.

```
#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;
}
```



Ordering the Uses of Data:

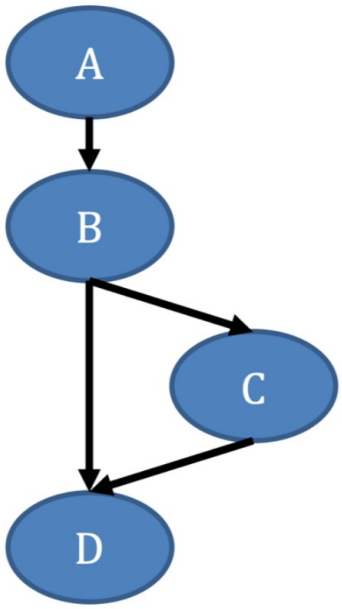
In DPC++, kernels can be viewed as asynchronous tasks that are submitted for execution. In many cases, kernels must execute in a specific order so that the correct result is computed. If obtaining the correct result requires task A to execute before task B, we say that a dependence exists between tasks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

Explicit dependences between tasks focuses on explicitly ordering tasks based on the computations that occur rather than on the data accessed by the computations. 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.

The second way is to use 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 below 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>(…);
  });
…
}
```

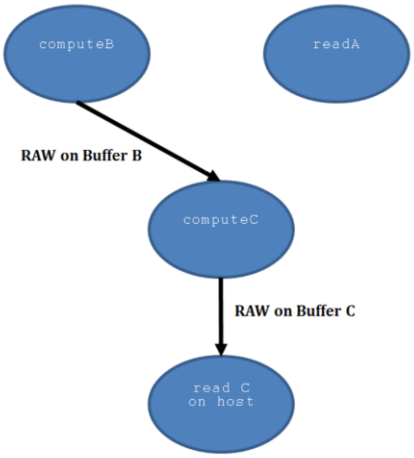


Implicit dependences between tasks in DPC++ are created from data dependences as shown below:

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

In Figures below, 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.



```
#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;
}
```

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.

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

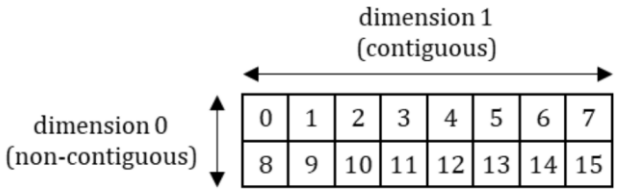
Chapter 4: Expressing Parallelism

We have already covered how to control where code runs (on what device), and how to get data to our code (accessible from a particular device). Now, we introduce the concept of a data parallel kernel.

Parallelism within Kernels:

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. It ensures that applications can scale up (or down) to fit the capabilities of different platforms. Whenever we want to achieve the highest levels of performance on a specific device, we should always perform some additional manual optimization work, regardless of the programming language we're using.

All multi-dimensional quantities related to parallelism in DPC++ use the same **rowmajor** convention, and in all contexts the contiguous dimension of a multi-dimensional quantity will be its **rightmost** as shown below:



A parallel kernel is not a loop, and does not have iterations as shown below. 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];
}
```

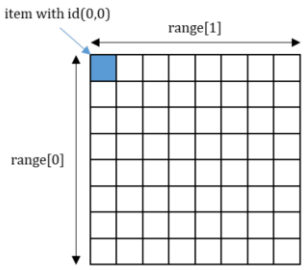
DPC++ provides three different kernel forms, each with their own execution models and syntax. The first kernel form is used for **basic** data parallel kernels. The second kernel form extends basic kernels to provide access to low-level performance-tuning features. This second form is known

as **ND-range** (N-dimensional range). The third form is referred to as **hierarchical data parallel**, referring to the hierarchy of the nested kernel constructs that appear in source code.

Basic Data Parallel Kernels:

The most basic form of parallel kernel in DPC++ is appropriate for operations that can be applied to every piece of data completely independently and in any order (embarrassingly parallel). Basic data parallel kernels are written in a Single Program Multiple Data (SPMD) style.

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.



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.

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

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.

Figure below 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];
});
```

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

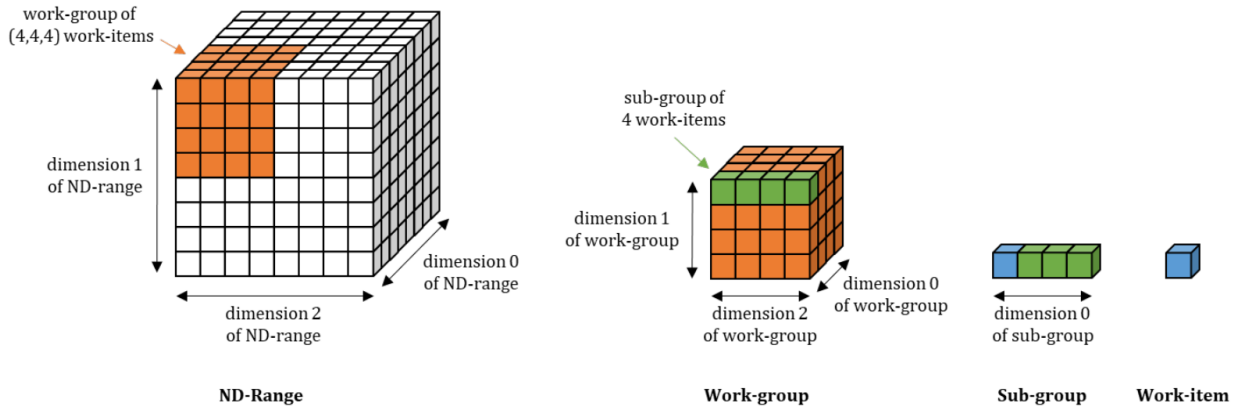
An item represents an individual instance of a kernel function, encapsulating both the execution range of the kernel and the instance's index within that range (using a range and an id, respectively). Like range and id, its dimensionality must be known at compile-time. The main difference between item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

ND-Range Kernels:

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.

The execution range of an ND-range kernel is divided into work-groups, sub-groups and work-items (as shown below). The ND-range represents the total execution range, which is divided into work-groups of uniform size. Each work-group can be further divided by the implementation into sub-groups.



The exact mapping from each type of group to hardware resources is implementation-defined. For example, work-items could be executed completely sequentially, or even executed by a hardware pipeline specifically configured for a particular kernel.

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 such as 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.

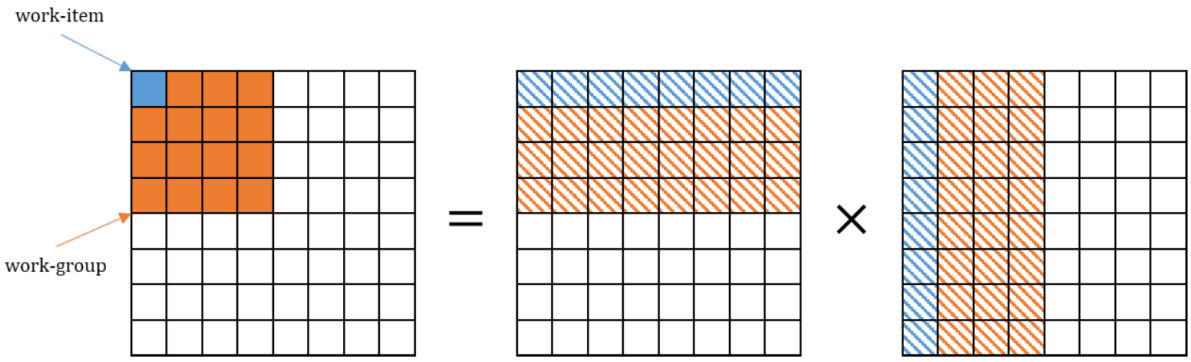
The number of work-items in a work-group is typically configured for each kernel at run-time, as the best grouping will depend upon both the amount of parallelism available (i.e. the size of the ND-range) and properties of the target device. We can determine the maximum number of work-items per work-group 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.

Sub-groups: 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. These subsets of work-items are known in DPC++ as sub-groups. Work-items in a sub-group can communicate directly, without explicit memory operations, using shuffle operations.

Figure below implements the matrix multiplication kernel using the ND-range parallel kernel:

```
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];
    }
});
```

diagram below 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 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.



ND-Kernel 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.

Hierarchical Data 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. 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. Mapping nested parallelism to accelerators is a challenge that is not unique to DPC++.

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.

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.

As shown in below, 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];
    }
  });
});
```

Choosing a Kernel Form:

Choosing between the different kernel forms is largely a matter of personal preference. However, the rule of thumb is the flowchart below, 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.

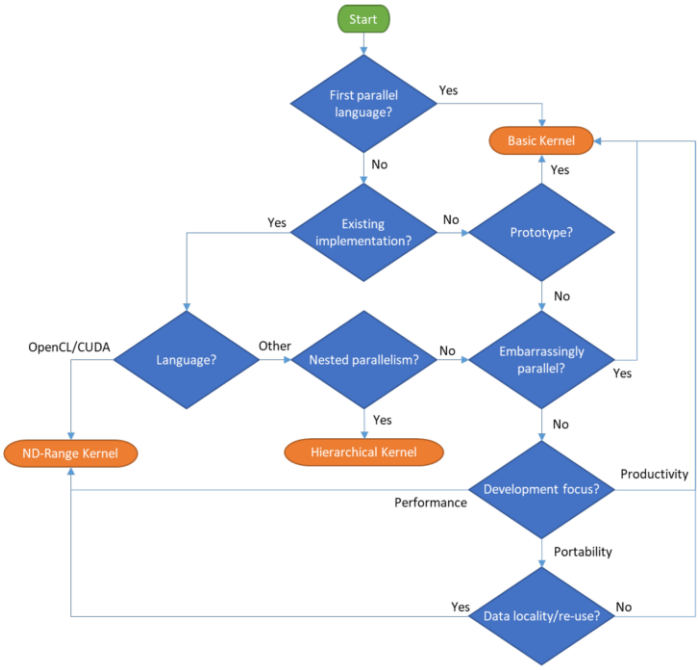


Figure 4-44, which summarizes the functionalities of DPC++ that are exposed to each of the kernel forms:

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.

Chapter 5: Simple profiling

First define and include following:

```
#include <chrono>
#define NS (1000000000.0) // number of nanoseconds in a second
```

Then setup profiling:

```
  //Profiling setup
  //Set things up for profiling at the host
  std::chrono::high_resolution_clock::time_point t1_host, t2_host;
  sycl::cl_ulong t1_kernel, t2_kernel;
  double time_kernel;
  auto property_list = sycl::property_list{sycl::property::queue::enable_profiling()};
```

Assign the property list to the queue:

```
 queue myq(selector,NULL,property_list);
```

And after the end of kernel, use these lines of codes:

```
    // Report kernel execution time and throughput
t1_kernel = e.get_profiling_info<sycl::info::event_profiling::command_start>();
t2_kernel = e.get_profiling_info<sycl::info::event_profiling::command_end>();
time_kernel = (t2_kernel - t1_kernel) / NS;
std::cout << "Kernel execution time: " << time_kernel << " seconds" << std::endl;
```