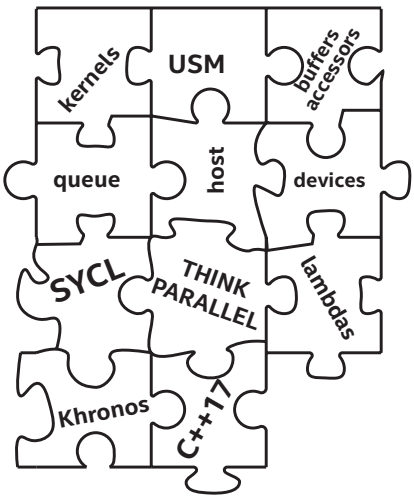
CHAPTER 4

Expressing Parallelism



Now we can put together our first collection of puzzle pieces. We already know how to place code (Chapter 2) and data (Chapter 3) on a device—all we must do now is engage in the art of deciding what to do with it. To that end, we now shift to fill in a few things that we have conveniently left out or glossed over so far. This chapter marks the transition from simple teaching examples toward real-world parallel code and expands upon details of the code samples we have casually shown in prior chapters.

J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-5574-2_4

Writing our first program in a new parallel language may seem like a daunting task, especially if we 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 these:

- 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 using working code examples, and highlight the most important aspects of the kernel execution model.

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 [Single Instruction, Multiple Data] instructions). Instead, kernels describe parallelism in terms of abstract concepts that an implementation (i.e., the combination of compiler and runtime) can then map to the hardware parallelism available on a specific target device. Although this mapping is implementation-defined, we can (and should) trust implementations to select a mapping that is sensible and capable of effectively exploiting hardware parallelism.

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, 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 15, 16, and 17.

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 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!

Multidimensional 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 multidimensional indexing provided by parallel kernels 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).

One important consideration is which dimension is *contiguous* or *unit-stride* (i.e., which locations in the multidimensional space are next to each other in the one-dimensional space). All multidimensional quantities related to parallelism in SYCL use the same convention: dimensions are numbered from 0 to N-1, where dimension N-1 corresponds to the contiguous dimension. Wherever a multidimensional quantity is written as a list (e.g., in constructors) or a class supports multiple subscript operators, this numbering applies left to right. This convention is consistent with the behavior of multidimensional arrays in standard C++.

An example of mapping a two-dimensional space 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.

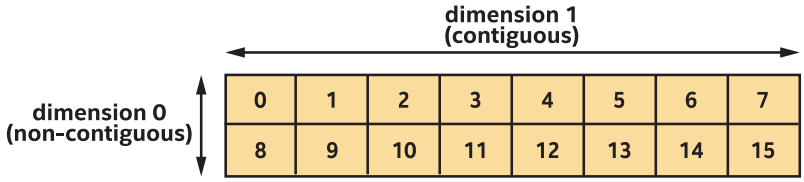


Figure 4-1. *Two-dimensional range of size (2, 8) mapped to linear indices*

If an application requires more than three dimensions, we must take responsibility for mapping between multidimensional 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. *Expressing a vector addition 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 runtime check! In order to address situations like this, 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[id] = a[id] + b[id];
}
```

Figure 4-3. *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. By writing this operation as a kernel, we are asserting that it is safe to run in parallel (and ideally should be).

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 parallel loops to kernels, the easier it will be to write effective parallel programs using Data Parallel C++.

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, and we need to familiarize ourselves with each of these options if we want to master the language.

Separating Kernels from Host Code

We have several alternative ways to separate host and device code, which we can mix and match within an application: 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 are shared by all these options. For consistency and brevity, all the code examples in this chapter express kernels using C++ lambdas.

LAMBDAS NOT CONSIDERED HARMFUL

There is no need to fully understand everything that the C++ specification says about lambdas in order to get started with DPC++—all we need to know is that the body of the lambda represents the kernel and that variables captured (by value) will be passed to the kernel as arguments.

There is no performance impact arising from the use of lambdas instead of more verbose mechanisms for defining kernels. A DPC++ compiler always understands when a lambda represents the body of a parallel kernel and can optimize for parallel execution accordingly.

For a refresher on C++ lambda functions, with notes about their use in SYCL, see Chapter 1. For more specific details on using lambdas to define kernels, see Chapter 10.

Different Forms of Parallel Kernels

There are three different kernel forms, supporting different 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 specific form to make a specific parallel algorithm easier to express or to make use of an otherwise inaccessible language feature.

The first form is used for *basic* data-parallel kernels and offers the gentlest introduction to writing kernels. 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 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 form provides an alternative syntax 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 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 complete control over the scheduling of 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 Single Program, Multiple Data (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.

One of the greatest strengths of a SPMD programming model is that it allows the same "program" to be mapped to multiple levels and types of parallelism, without any explicit direction from us. Instances of the same program could be pipelined, packed together and executed with SIMD instructions, distributed across multiple threads, or a mix of all three.

Understanding Basic Data-Parallel Kernels

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

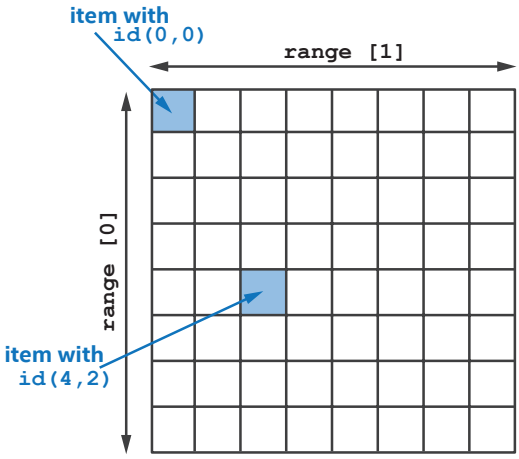


Figure 4-4. *Execution space of a basic parallel kernel, shown for a 2D range of 64 items*

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 cause programs to hang on some devices.

However, in order to guarantee correctness, we must always write our kernels under the assumption 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 (see Chapter 19) in order to prevent race conditions.

Writing Basic Data-Parallel Kernels

Basic data-parallel kernels are expressed using the `parallel_for` function. Figure 4-5 shows how to use this function to express a vector addition, which is our take on "Hello, world!" for parallel accelerator programming.

```
h.parallel_for(range{N}, [=](id<1> idx) {
  c[idx] = a[idx] + b[idx];
});
```

Figure 4-5. *Expressing a vector addition kernel 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. There are several different classes that can be accepted as arguments to a kernel function, and which should be used depends on which class exposes the functionality required—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 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 multidimensional `id`. As strange as it looks, this can be very powerful, enabling us to write kernels templated on the dimensionality of our data.

```
h.parallel_for(range{N, M}, [=](id<2> idx) {
  c[idx] = a[idx] + b[idx];
});
```

Figure 4-6. *Expressing a matrix addition kernel with* `parallel_for`

It is more common in C/C++ to use multiple indices and multiple subscript operators to index multidimensional data structures, and this explicit indexing is also supported by accessors. Using multiple indices in this way 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 4-7 must 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.

```
h.parallel_for(range{N, N}, [=](id<2> idx) {
  int j = idx[0];
  int i = idx[1];
  for (int k = 0; k < N; ++k) {
    c[j][i] += a[j][k]    * b[k][i];
//  c[idx]  += a[id(j,k) * b[id(k,i)];  <<< equivalent
  }
});
```

Figure 4-7. *Expressing a naïve matrix multiplication kernel for square matrices, with* `parallel_for`

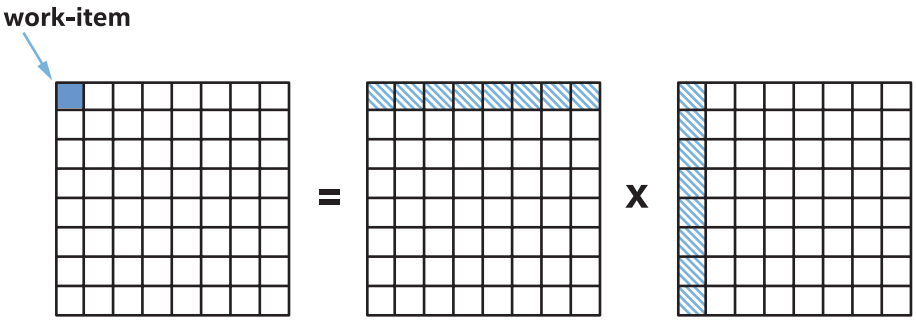


Figure 4-8. *Mapping matrix multiplication work to items in the execution range*

The diagram in Figure 4-8 shows how the work in our matrix multiplication kernel is mapped to individual items. Note that the number of items is derived from the size of the *output* range and that the same input values may be read by multiple 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.

Details of Basic Data-Parallel Kernels

The functionality of basic data-parallel kernels is exposed via three C++ classes: `range`, `id`, and `item`. We've already seen the `range` and `id` classes a few times in previous chapters, 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 runtime. Instances of the `range` class are used 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. *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, to obtain the `id` associated with a specific kernel instance, we must accept it (or an `item` containing it) as an argument to a kernel function. This `id` (or values returned by its member functions) must be forwarded to any function in which we want to query the index—there are not currently any free functions for querying the index at arbitrary points in a program, but this may be addressed by a future version of DPC++.

Each instance of a kernel accepting an `id` knows only the index in the range that it has been assigned to compute and knows nothing about the range itself. If we want our kernel instances to know about their own index *and* the range, we need to use the `item` class instead.

```
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. *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 specific kernel instance is to accept it as an argument to a kernel function.

```
template <int Dimensions = 1, bool WithOffset = 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. *Simplified definition of the* `item` *class*

Explicit ND-Range Kernels

The second form of parallel kernel 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.

Understanding Explicit ND-Range Parallel Kernels

The execution range of an ND-range kernel is divided into work-groups, sub-groups, and work-items. 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 for work-items and each type of group is an important part of writing correct and portable programs.

Figure 4-12 shows an example of an 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. Pay careful attention to the numbering of the dimensions: sub-groups are always one-dimensional, and so dimension 2 of the ND-range and work-group becomes dimension 0 of the sub-group.

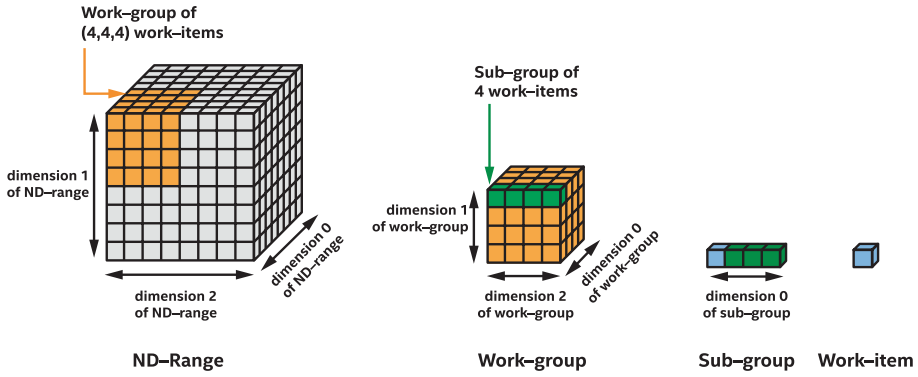


Figure 4-12. *Three-dimensional ND-range divided into work-groups, sub-groups, and work-items*

The exact mapping from each type of group to hardware resources is *implementation-defined*, and it is this flexibility that enables 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 specific 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. See Chapters 15, 16, and 17 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 (see Chapter 19).

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 with each other except by way of atomic memory operations to global memory (see Chapter 19). 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 (see Chapter 9).

2. Work-items in a work-group can synchronize using *work-group barriers* and guarantee memory consistency using *work-group memory fences* (see Chapter 9).

3. Work-items in a work-group have access to *group functions*, providing implementations of common communication routines (see Chapter 9) and common parallel patterns such as reductions and scans (see Chapter 14).

The number of work-items in a work-group is typically configured for each kernel at runtime, 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 specific device using the query functions of the `device` class (see Chapter 12), 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 can execute concurrently! We 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 against doing this—such kernels may appear to work today, but they are not guaranteed to work with future implementations and are highly likely to break when moved to a different device.

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 we're already familiar with a task-based programming model, we may find it useful to think of work-groups as though they are data-parallel tasks.

Sub-Groups

On many modern hardware platforms, subsets of the work-items in a work-group known as *sub-groups* are executed with additional scheduling guarantees. For example, the work-items in a sub-group could be executed simultaneously as a result of compiler vectorization, and/or the sub-groups themselves could be executed with forward progress guarantees because they are mapped to independent hardware threads.

When working with a single platform, it is tempting to bake assumptions about these execution models 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!

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, the work-items within a sub-group can synchronize, guarantee memory consistency, or execute common parallel patterns via group functions. However, there is no equivalent of work-group local memory for sub-groups (i.e., there is no sub-group local memory). Instead, the work-items in a sub-group can exchange data directly—without explicit memory operations—using *shuffle* operations (Chapter 9).

Some aspects of sub-groups are 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 (see Chapter 10). 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—an implementation is free to execute each work-item in a sub-group sequentially and only switch between work-items when a sub-group collective function is encountered. Where sub-groups are special is that some devices guarantee that they make independent forward progress—on some devices, all sub-groups within a work-group are guaranteed to execute (make progress) eventually, which is a cornerstone of several producer-consumer patterns. Whether or not this independent forward progress guarantee holds can be determined using a device query.

THINKING IN SUB-GROUPS

If we are coming from a programming model that requires us 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 forward progress, this mental model extends to treating each sub-group as though it were a separate stream of vector instructions executing in parallel.

```
range global{N, N};
range local{B, B};
h.parallel_for(nd_range{global, local}, [=](nd_item<2> it) {
  int j = it.get_global_id(0);
  int i = it.get_global_id(1);

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

Figure 4-13. *Expressing a naïve matrix multiplication kernel with ND-range* `parallel_for`

Writing Explicit ND-Range Data-Parallel Kernels

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 reused four times.

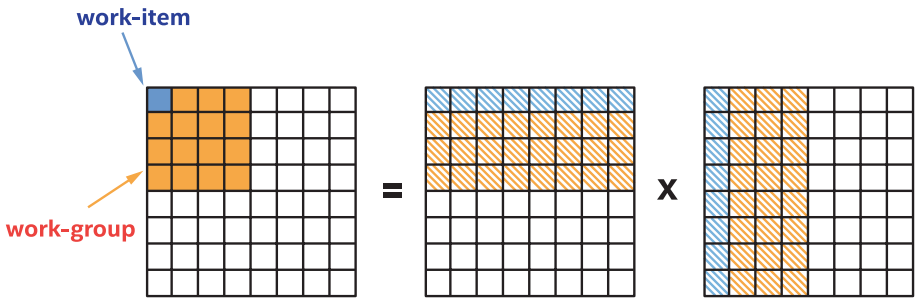


Figure 4-14. *Mapping matrix multiplication to work-groups and work-items*

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 can 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). Usage of this work-group local memory will be covered in Chapter 9.

Details of Explicit ND-Range Data-Parallel Kernels

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 4-15.

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 two 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. All functionality related to sub-groups is encapsulated in a dedicated class that will be discussed shortly.

```
template <int Dimensions = 1>
class nd_range {
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 4-15. *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 4-16.

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 4-16. *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 4-17.

```
template <int Dimensions = 1>
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;
};
```

Figure 4-17. *Simplified definition of the group class*

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 tradeoff 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 4-18: it is clear that body 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 for readers to follow.

```
void body(group& g);

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

Figure 4-18. *Using the group class to improve readability*

The sub_group Class

The sub_group class encapsulates all functionality related to sub-groups, and a simplified definition is shown in Figure 4-19. 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.

```
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 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;
};
```

Figure 4-19. *Simplified definition of the sub_group class*

Note that there are separate functions for querying the number of work-items in the current sub-group and the maximum number of work-items in any sub-group within the work-group. Whether and how these differ depends on exactly how sub-groups are implemented for a specific device, but the intent is to reflect any differences between the sub-group size targeted by the compiler and the runtime sub-group size. For example, very small work-groups may contain fewer work-items than the compile-time sub-group size, or sub-groups of different sizes may be used to handle work-groups that are not divisible by the 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 environment; 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.

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.

Understanding Hierarchical Data-Parallel Kernels

The underlying execution model of hierarchical data-parallel kernels is the same as the execution model of explicit ND-range data-parallel kernels. Work-items, sub-groups, and work-groups have identical semantics and execution guarantees.

However, the different scopes of a hierarchical kernel are mapped by the compiler to different execution resources: the outer scope is executed once per work-group (as if executed by a single work-item), while the inner scope is executed in parallel by work-items within the work-group. The different scopes also control where in memory different variables should be allocated, and the opening and closing of scopes imply work-group barriers (to enforce memory consistency).

Although the work-items in a work-group are still divided into sub-groups, the `sub_group` class cannot currently be accessed from a hierarchical parallel kernel; incorporating the concept of sub-groups into SYCL hierarchical parallelism requires more significant changes than introducing a new class, and work in this area is ongoing.

Writing Hierarchical Data-Parallel Kernels

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. Any code in a `parallel_for_work_group` scope is executed only once per work-group, and 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). Any code in a `parallel_for_work_item` scope is executed in parallel by the work-items of the work-group, and variables allocated in a `parallel_for_work_item` scope are visible to a single work-item (i.e., they are allocated in work-item private memory).

As shown in Figure 4-20, 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 must write.

```
range num_groups{N / B, N / B}; // N is a multiple of B
range group_size{B, B};
h.parallel_for_work_group(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> it) {
    int j = jb * B + it.get_local_id(0);
    int i = ib * B + it.get_local_id(1);
    for (int k = 0; k < N; ++k)
      c[j][i] += a[j][k] * b[k][i];
  });
});
```

Figure 4-20. *Expressing a naïve matrix multiplication kernel with hierarchical parallelism*

It is important to note that the ranges passed to the `parallel_for_work_group` 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, and the logical items of the logical range are assigned round-robin to physical work-items.

Figure 4-21 shows an example of the mapping between a logical range consisting of 11 logical work-items and an underlying physical range consisting of 8 physical work-items. The first three work-items are assigned two instances of the function, and all other work-items are assigned only one.

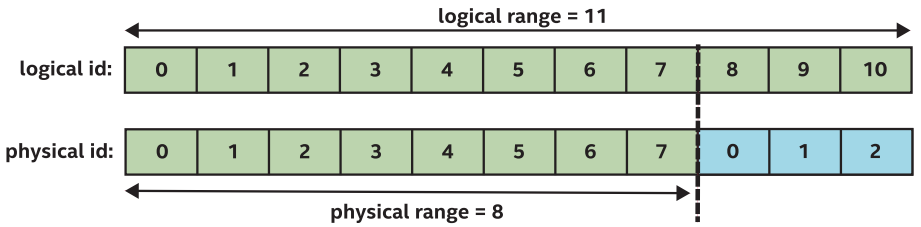


Figure 4-21. *Mapping a logical range of size 11 to a physical range of size 8*

As shown in Figure 4-22, 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 4-20, but that the amount of work has now been separated from the physical work-group size.

```
range num_groups{N / B, N / B}; // N is a multiple of B
range group_size{B, B};
h.parallel_for_work_group(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> it) {
    int j = jb * B + it.get_logical_local_id(0);
    int i = ib * B + it.get_logical_local_id(1);
    for (int k = 0; k < N; ++k)
      c[j][i] += a[j][k] * b[k][i];
  });
});
```

Figure 4-22. *Expressing a naïve matrix multiplication kernel with hierarchical parallelism and a logical range*

Details of Hierarchical Data-Parallel Kernels

Hierarchical data-parallel kernels reuse 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 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. As shown in Figure 4-23, 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 4-23. *Simplified definition of the* `h_item` *class*

The `private_memory` Class

The `private_memory` class provides a mechanism to declare variables that are private to each work-item, but which can be accessed across multiple `parallel_for_work_item` constructs nested within the same `parallel_for_work_group` scope.

This class is necessary because of how variables declared in different hierarchical parallelism scopes behave: 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 4-22. The `ib` and `jb` variables are declared at `parallel_for_work_group` scope and by default should be allocated 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 such 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 4-24.

```
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 4-24. *Simplified definition of the* `private_memory` *class*

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 results in code that is harder to read, and declaring 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 only when their analysis fails (remembering to also report the issue to the compiler vendor).

Mapping Computation to Work-Items

Most of the code examples so far have assumed that each instance of a kernel function corresponds to a single operation on a single piece of data. This is a simple way to write kernels, but such a one-to-one mapping is not

dictated by DPC++ or any of the kernel forms—we always have complete control over the assignment of data (and computation) to individual work-items, and making this assignment parameterizable can be a good way to improve performance portability.

One-to-One Mapping

When we write kernels such that there is a one-to-one mapping of work to work-items, those kernels must always be launched with a `range` or `nd_range` with a size exactly matching the amount of work that needs to be done. This is the most obvious way to write kernels, and in many cases, it works very well—we can trust an implementation to map work-items to hardware efficiently.

However, when tuning for performance on a specific combination of system and implementation, it may be necessary to pay closer attention to low-level scheduling behaviors. The scheduling of work-groups to compute resources is implementation-defined and could potentially be *dynamic* (i.e., when a compute resource completes one work-group, the next work-group it executes may come from a shared queue). The impact of dynamic scheduling on performance is not fixed, and its significance depends upon factors including the execution time of each instance of the kernel function and whether the scheduling is implemented in software (e.g., on a CPU) or hardware (e.g., on a GPU).

Many-to-One Mapping

The alternative is to write kernels with a many-to-one mapping of work to work-items. The *meaning* of the range changes subtly in this case: the range no longer describes the amount of work to be done, but rather the number of workers to use. By changing the number of workers and the amount of work assigned to each worker, we can fine-tune work distribution to maximize performance.

Writing a kernel of this form requires two changes:

1. The kernel must accept a parameter describing the total amount of work.
2. The kernel must contain a loop assigning work to work-items.

A simple example of such a kernel is given in Figure 4-25. Note that the loop inside the kernel has a slightly unusual form—the starting index is the work-item's index in the global range, and the stride is the total number of work-items. This *round-robin* scheduling of data to work-items ensures that all `N` iterations of the loop will be executed by a work-item, but also that linear work-items access contiguous memory locations (to improve cache locality and vectorization behavior). Work can be similarly distributed across groups or the work-items in individual groups to further improve locality.

```
size_t N = ...; // amount of work
size_t W = ...; // number of workers
h.parallel_for(range{W}, [=](item<1> it) {
  for (int i = it.get_id()[0]; i < N; i += it.get_range()[0]) {
    output[i] = function(input[i]);
  }
});
```

Figure 4-25. *Kernel with separate data and execution ranges*

These work distribution patterns are common, and they can be expressed very succinctly when using hierarchical parallelism with a logical range. We expect that future versions of DPC++ will introduce syntactic sugar to simplify the expression of work distribution in ND-range kernels.

Choosing a Kernel Form

Choosing between the different kernel forms is largely a matter of personal preference and heavily influenced by prior experience with other parallel programming models and languages.

The other main reason to choose a specific kernel form is that it is the only form to expose certain functionality required by a kernel. Unfortunately, it can be difficult to identify which functionality will be required before development begins—especially while we are still unfamiliar with the different kernel forms and their interaction with various classes.

We have constructed two guides based on our own experience in order to help us navigate this complex space. These guides should be considered rules of thumb and are definitely not intended to replace our 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 our application and development style.

The first guide is the flowchart in Figure 4-26, which selects a kernel form based on

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

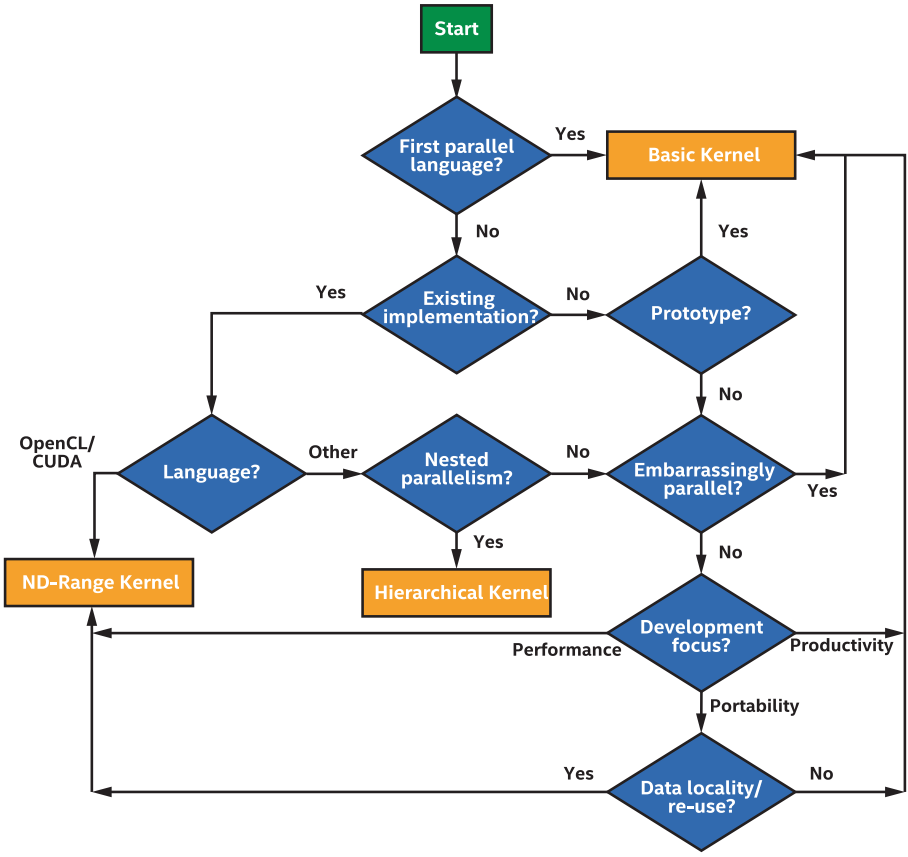


Figure 4-26. *Helping choose the right form for our kernel*

The second guide is the table in Figure 4-27, which summarizes the functionalities 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 publication for this book 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, 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
Sub-groups	No	Yes	No
Group Functions (e.g., `scan`, `reduce`)	No	Yes	No

Figure 4-27. *Features available to each kernel form*

Summary

This chapter introduced the basics of expressing parallelism in DPC++ and discussed the strengths and weaknesses of each approach to writing data-parallel kernels.

DPC++ and SYCL provide support for 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, and a deeper dive into many of the concepts and classes introduced in this chapter is forthcoming: the usage of local memory, barriers, and communication routines will be covered 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 15, 16, and 17; and best practices for expressing common parallel patterns using DPC++ will be presented in Chapter 14.