CHAPTER 19

Memory Model and Atomics

Memory consistency is not an esoteric concept if we want to be parallel programmers. It helps us to ensure that data is where we need it, when we need it, and that its values are what we are expecting. This chapter brings to light key things we need to master to ensure our program hums along correctly. This topic is not unique to SYCL.

Having a basic understanding of the memory (consistency) model of a programming language is necessary for *any* programmer who wants to allow concurrent updates to memory (whether those updates originate from multiple work-items in the same kernel, multiple devices, or both). This is true regardless of how memory is allocated, and the content of this chapter is equally important to us whether we choose to use buffers or USM allocations.

In previous chapters, we have focused on the development of simple kernels, where work-items either operate on completely independent data or share data using structured communication patterns that can be expressed directly using language and/or library features. As we move toward writing more complex and realistic kernels, we are likely to encounter situations where work-items may need to communicate in less structured ways—understanding how the memory model relates to SYCL language features and the capabilities of the hardware we are targeting is a necessary precondition for designing correct, portable, and efficient programs.

J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_19

THREADS OF EXECUTION

C++17 introduced the concept of a "thread of execution" (often referred to simply as a "thread") to help describe the behaviors of library features related to parallelism and concurrency (e.g., the parallel algorithms). The C++ memory consistency model and execution model are defined entirely in terms of interactions between these "threads."

To simplify comparison between SYCL and C++, this chapter often uses the term "thread" to mean "thread of execution." A SYCL work-item is equivalent to a C++ thread of execution with weakly parallel forward progress guarantees, and so it is safe to use these terms interchangeably—occasionally, we may still use "work-item" to highlight when we are discussing SYCL-specific concepts.

The memory consistency model of C++ is sufficient for writing applications that execute entirely on the host, but it is modified by SYCL in order to address complexities that may arise when programming heterogeneous systems. Specifically, we need to be able to

- Reason about which types of memory allocation (buffers and USM) can be accessed by which devices in the system
- Prevent unsafe concurrent memory accesses (data races) during the execution of our kernels by using barriers and atomics
- Enable safe communication between work-items using barriers, fences, atomics, memory orders, and memory scopes

- Prevent optimizations that may unexpectedly alter the behavior of parallel applications—while still allowing other optimizations—using barriers, fences, atomics, memory orders, and memory scopes

Memory models are a complex topic, but for a good reason—processor architects care about making processors and accelerators execute our codes as efficiently as possible! We have worked hard in this chapter to break down this complexity and highlight the most critical concepts and language features. This chapter starts us down the path of not only knowing the memory model inside and out but also enjoying an important aspect of parallel programming that many people do not know exists. If questions remain after reading the descriptions and example codes here, we highly recommend visiting the websites listed at the end of this chapter or referring to the C++ and SYCL specifications.

What's in a Memory Model?

This section expands upon the motivation for programming languages to contain a memory model and introduces a few core concepts that parallel programmers should familiarize themselves with:

- Data races and synchronization
- Barriers and fences
- Atomic operations
- Memory ordering

Understanding these concepts at a high level is necessary to appreciate their expression and usage in C++ with SYCL. Readers with extensive experience in parallel programming, especially using C++, may wish to skip ahead.

Data Races and Synchronization

The *operations* that we write in our programs typically do not map directly to a single hardware instruction or micro-operation. A simple addition operation such as `data[i] += x` may be broken down into a sequence of several instructions or micro-operations:

- Load `data[i]` from memory into a temporary (register).
- Compute the result of adding `x` to `data[i]`.
- Store the result back to `data[i]`.

This is not something that we need to worry about when developing sequential applications—the three stages of the addition will be executed in the order that we expect, as depicted in Figure 19-1.

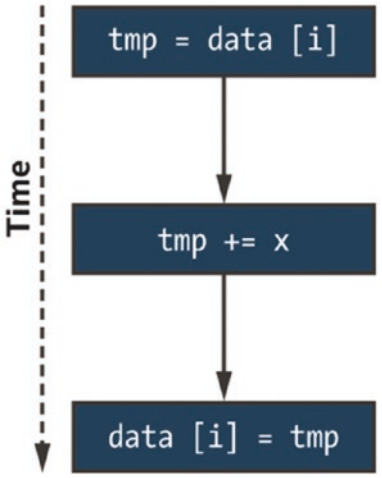

Figure 19-1. *Sequential execution of* `data[i] += x` *broken into three separate operations*

Switching to parallel application development introduces an extra level of complexity: if we have multiple operations being applied to the same data concurrently, how can we be certain that their view of that data is consistent? Consider the situation shown in Figure 19-2, where two executions of `data[i] += x` have been interleaved on two threads. If the two threads use different values of `i`, the application will execute correctly. If they use the same value of `i`, both load the same value from memory, and one of the results is overwritten by the other! This is just one of many ways in which their operations could be scheduled, and the behavior of our application depends on which thread gets to which data first—our application contains a *data race*.

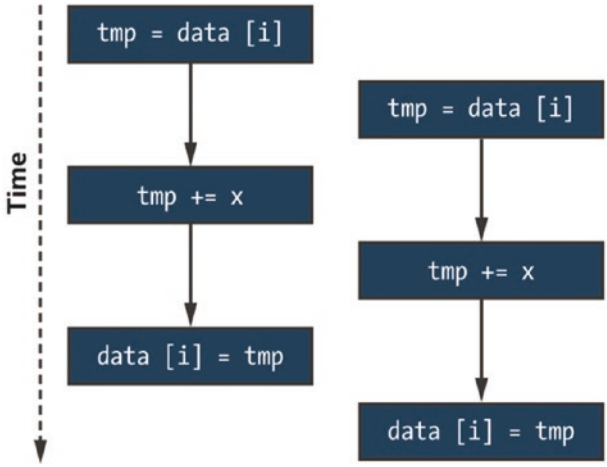

Figure 19-2. *One possible interleaving of* `data[i] += x` *executed concurrently*

The code in Figure 19-3 and its output in Figure 19-4 show how easily this can happen in practice. If `M` is greater than or equal to `N`, the value of `j` used by each thread is unique; if it is not, values of `j` will conflict, and updates may be lost. We say *may* be lost because a program containing a data race could still produce the correct answer some or all the time (depending on how work is scheduled by the implementation and hardware). Neither the compiler nor the hardware can possibly know

what this program is *intended* to do or what the values of `N` and `M` may be at runtime—it is our responsibility as programmers to understand whether our programs may contain data races and whether they are sensitive to execution order.

```
int* data = malloc_shared<int>(N, q);
std::fill(data, data + N, 0);

q.parallel_for(N, [=](id<1> i) {
   int j = i % M;
   data[j] += 1;
 }).wait();

for (int i = 0; i < N; ++i) {
  std::cout << "data [" << i << "] = " << data[i] << "\n";
}
```

Figure 19-3. *Kernel containing a data race*

```
N = 2, M = 2:
data [0] = 1
data [1] = 1

N = 2, M = 1:
data [0] = 1
data [1] = 0
```

Figure 19-4. *Sample output of the code in Figure 19-3 for small values of `N` and `M`*

In general, when developing massively parallel SYCL applications, we should not concern ourselves with the exact order in which individual work-items execute—there are hopefully hundreds (or thousands!) of work-items in each of our kernels, and trying to impose a specific ordering upon them will negatively impact both scalability and performance. Rather, our focus should be on developing portable applications that execute correctly, which we can achieve by providing the compiler (and hardware) with information about when work-items share data, what guarantees are needed when sharing occurs, and which execution orderings are legal.

Massively parallel applications should not be concerned with the exact order in which individual work-items execute!

Barriers and Fences

One way to prevent data races between work-items in the same group is to introduce synchronization across different threads using work-group barriers and appropriate memory fences. We could use a work-group barrier to order our updates of `data[i]` as shown in Figure 19-5, and an updated version of our example kernel is given in Figure 19-6. Note that because a work-group barrier does not synchronize work-items in different groups, our simple example is only guaranteed to execute correctly if we limit ourselves to a single work-group!

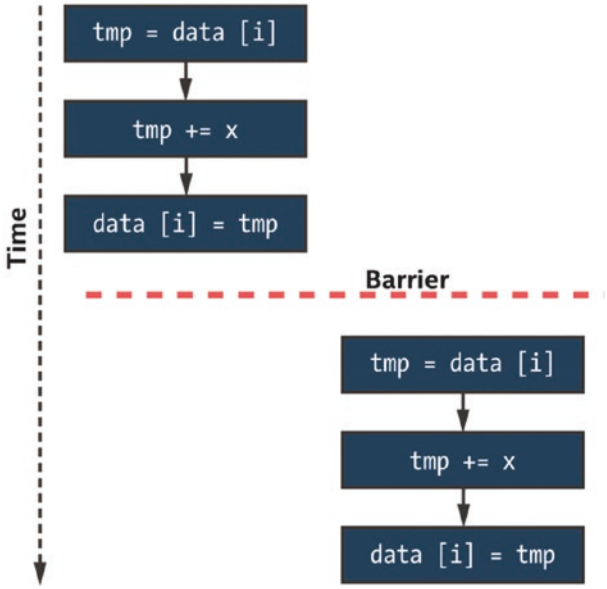

Figure 19-5. *Two executions of* `data[i] += x` *separated by a barrier*

```
int* data = malloc_shared<int>(N, q);
std::fill(data, data + N, 0);

// Launch exactly one work-group
// Number of work-groups = global / local
range<1> global{N};
range<1> local{N};

q.parallel_for(nd_range<1>{global, local},
               [=](nd_item<1> it) {
                 int i = it.get_global_id(0);
                 int j = i % M;
                 for (int round = 0; round < N; ++round) {
                   // Allow exactly one work-item update
                   // per round
                   if (i == round) {
                     data[j] += 1;
                   }
                   group_barrier(it.get_group());
                 }
               })
    .wait();

for (int i = 0; i < N; ++i) {
  std::cout << "data [" << i << "] = " << data[i] << "\n";
}
```

Figure 19-6. *Avoiding a data race using a barrier*

Although using a barrier to implement this pattern is possible, it is not typically encouraged—it forces the work-items in a group to execute sequentially and in a specific order, which may lead to long periods of inactivity in the presence of load imbalance. It may also introduce more synchronization than is strictly necessary—if the different work-items happen to use different values of `i`, they will still be forced to synchronize at the barrier.

Barrier synchronization is a useful tool for ensuring that all work-items in a work-group or sub-group complete some stage of a kernel before proceeding to the next stage, but is too heavy-handed for fine-grained (and potentially data-dependent) synchronization. For more general synchronization patterns, we must look to *atomic* operations.

Atomic Operations

Atomic operations enable concurrent access to a memory location without introducing a data race. When multiple atomic operations access the same memory, they are guaranteed not to overlap. Note that this guarantee does not apply if only some of the accesses use atomics and that it is our responsibility as programmers to ensure that we do not concurrently access the same data using operations with different atomicity guarantees.

Mixing atomic and non-atomic operations on the same memory location(s) at the same time results in undefined behavior!

If our simple addition is expressed using atomic operations, the result may look like Figure 19-8—each update is now an indivisible chunk of work, and our application will always produce the correct result. The corresponding code is shown in Figure 19-7—we will revisit the `atomic_ref` class and the meaning of its template arguments later in the chapter.

```
int* data = malloc_shared<int>(N, q);
std::fill(data, data + N, 0);

q.parallel_for(N, [=](id<1> i) {
   int j = i % M;
   atomic_ref<int, memory_order::relaxed,
              memory_scope::system,
              access::address_space::global_space>
       atomic_data(data[j]);
   atomic_data += 1;
 }).wait();

for (int i = 0; i < N; ++i) {
  std::cout << "data [" << i << "] = " << data[i] << "\n";
}
```

Figure 19-7. *Avoiding a data race using atomic operations*

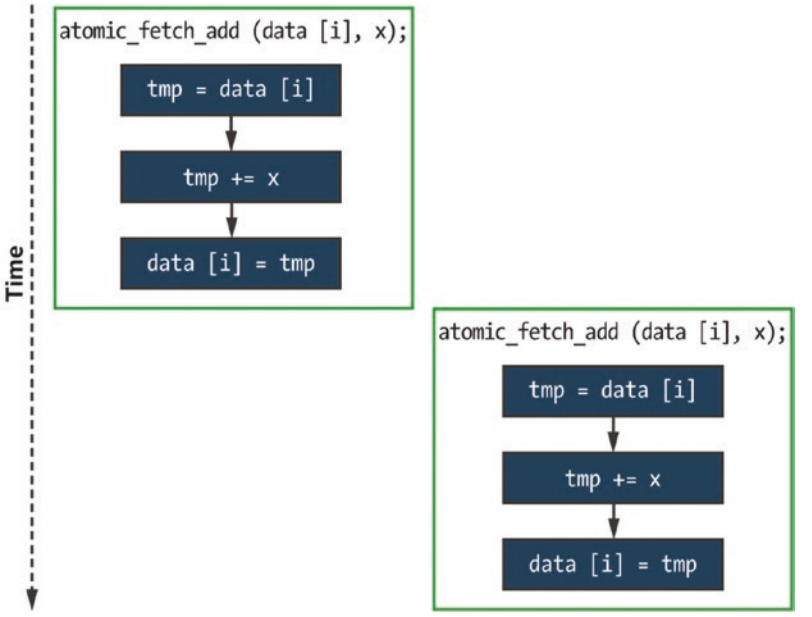

Figure 19-8. *An interleaving of* `data[i] += x` *executed concurrently with atomic operations*

However, it is important to note that this is still only one possible execution order. Using atomic operations guarantees that the two updates do not overlap (if both threads use the same value of `i`), but there is still no guarantee as to which of the two threads will execute first. Even more importantly, there are no guarantees about how these atomic operations are ordered with respect to any *non-atomic* operations in different threads.

Memory Ordering

Even within a sequential application, optimizing compilers and the hardware are free to reorder operations if they do not change the observable behavior of an application. In other words, the application must behave *as if* it ran exactly as it was written by the programmer.

Unfortunately, this as-if guarantee is not strong enough to help us reason about the execution of parallel programs. We now have two sources of reordering to worry about: the compiler and hardware may reorder the execution of statements within each sequential thread, and the threads themselves may be executed in any (possibly interleaved) order. To design and implement safe communication protocols between threads, we need to be able to constrain this reordering. By providing the compiler with information about our desired *memory order*, we can prevent reordering optimizations that are incompatible with the intended behavior of our applications.

Three commonly available memory orderings are:

1. A *relaxed* memory ordering
2. An *acquire-release* or release-acquire memory ordering
3. A *sequentially consistent* memory ordering

Under a relaxed memory ordering, memory operations can be reordered without any restrictions. The most common usage of a relaxed memory model is incrementing shared variables (e.g., a single counter, an array of values during a histogram computation).

Under an acquire-release memory ordering, one thread *releasing* an atomic variable and another thread *acquiring* the same atomic variable acts as a synchronization point between those two threads and guarantees that any prior writes to memory issued by the releasing thread are visible to the acquiring thread. Informally, we can think of atomic operations releasing side effects from other memory operations to other threads or acquiring the side effects of memory operations on other threads. Such a memory model is required if we want to communicate values between pairs of threads via memory, which may be more common than we would think. When a program *acquires* a lock, it typically goes on to perform some additional calculations and modify some memory before eventually

releasing the lock—only the lock variable is ever updated atomically, but we expect memory updates guarded by the lock to be protected from data races. This behavior relies on an acquire-release memory ordering for correctness, and attempting to use a relaxed memory ordering to implement a lock will not work.

Under a sequentially consistent memory ordering, the guarantees of acquire-release ordering still hold, but there additionally exists a single global order of all atomic operations. The behavior of this memory ordering is the most intuitive of the three and the closest that we can get to the original as-if guarantee we are used to relying upon when developing sequential applications. With sequential consistency, it becomes significantly easier to reason about communication between groups (rather than pairs) of threads, since all threads must agree on the global ordering of all atomic operations.

Understanding which memory orders are supported by a combination of programming model and device is a necessary part of designing portable parallel applications. Being explicit in describing the memory order required by our applications ensures that they fail predictably (e.g., at compile time) when the behavior we require is unsupported and prevents us from making unsafe assumptions.

The Memory Model

The chapter so far has introduced the concepts required to understand the memory model. The remainder of the chapter explains the memory model in detail, including

- How to express the memory ordering requirements of our kernels
- How to query the memory orders supported by a specific device

- How the memory model behaves with respect to disjoint address spaces and multiple devices
- How the memory model interacts with barriers, fences, and atomics
- How using atomic operations differs between buffers and USM

The memory model is based on the memory model of C++ but differs in some important ways. These differences reflect our long-term vision that SYCL should help inform the future of C++: the default behaviors and naming of classes are closely aligned with the C++ standard library and are intended to extend C++ functionality rather than to restrict it.

The table in Figure 19-9 summarizes how different memory model concepts are exposed as language features in C++ (C++11, C++14, C++17, C++20) vs. SYCL. The C++14, C++17, and C++20 standards additionally include some clarifications that impact implementations of C++. These clarifications should not affect the application code that we write, so we do not cover them here.

Feature	C++	SYCL
Atomic Objects	`std::atomic`	Not available.
Atomic References	`std::atomic_ref` (C++20 onwards)	`sycl::atomic_ref`

Memory Orders	`relaxed`
	`consume`
	`acquire`
	`release`
	`acq_rel`
	`seq_cst`
		`relaxed`
		
		`acquire`
		`release`
		`acq_rel`
		`seq_cst`

Memory Scopes	Not available.
	
	Behavior of atomics and fences matches SYCL `system` scope.
		`work_item`
		`sub_group`
		`work_group`
		`device`
		`system`

Fences	`std::atomic_thread_fence`	`sycl::atomic_fence`
Barriers	`std::barrier` (C++20 onwards)	`sycl::group_barrier`
Address Spaces	All memory is in a single (host) address space.	Host Device (Global) Device (Local) Device (Private) Shared (USM)

Figure 19-9. *Comparing C++ and SYCL memory models*

The `memory_order` Enumeration Class

The memory model exposes different memory orders through five values of the `memory_order` enumeration class (note: C++ "consume" is not part of SYCL), which can be supplied as arguments to fences and atomic operations. Supplying a memory order argument to an operation tells the compiler what memory ordering guarantees are required for all other memory operations (to any address) *relative to that operation*, as explained in the following:

- `memory_order::relaxed`

  Read and write operations can be reordered before or after the operation with no restrictions. There are no ordering guarantees.

- `memory_order::acquire`

  Read and write operations appearing after the operation in the program must occur after it (i.e., they cannot be reordered before the operation).

- `memory_order::release`

  Read and write operations appearing before the operation in the program must occur before it (i.e., they cannot be reordered after the operation), and preceding write operations are guaranteed to be visible to other work-items which have been synchronized by a corresponding acquire operation (i.e., an atomic operation using the same variable and `memory_order::acquire` or a barrier function).

- `memory_order::acq_rel`

  The operation acts as both an acquire and a release. Read and write operations cannot be reordered around the operation, and preceding writes must be made visible as previously described for `memory_order::release`.

- `memory_order::seq_cst`

  The operation acts as an acquire, release, or both depending on whether it is a read, write, or read–modify–write operation, respectively. All operations with this memory order are observed in a sequentially consistent order.

There are several restrictions on which memory orders are supported by each operation. The table in Figure 19-10 summarizes which combinations are valid.

Functions	Supported memory_order Values				
	relaxed	acquire	release	acq_rel	seq_cst
load	✓	✓	✗	✗	✓
store	✓	✗	✓	✗	✓
exchange compare_exchange_* fetch_*	✓	✓	✓	✓	✓
fence	✓	✓	✓	✓	✓

Figure 19-10. *Supporting atomic operations with* `memory_order`

Load operations do not write values to memory and are therefore incompatible with release semantics. Similarly, store operations do not read values from memory and are therefore incompatible with acquire semantics. The remaining read–modify–write atomic operations and fences are compatible with all memory orderings.

MEMORY ORDER IN C++

The C++ memory model additionally includes `memory_order::consume`, with similar behavior to `memory_order::acquire`. However, C++17 discourages its use, noting that its definition is being revised. Its inclusion in SYCL has therefore been left to consider for a future specification.

The `memory_scope` Enumeration Class

The C++ memory model assumes that applications execute on a single device with a single address space. Neither of these assumptions holds for SYCL applications: various parts of the application execute on different

devices (i.e., a host and one or more accelerator devices); each device has multiple address spaces (i.e., private, local, and global); and the global address space of each device may or may not be disjoint (depending on USM support).

To address this, SYCL extends the C++ notion of memory order to include the *scope* of an atomic operation, denoting the minimum set of work-items to which a given memory ordering constraint applies. The set of scopes are defined by way of a `memory_scope` enumeration class:

- `memory_scope::work_item`

  The memory ordering constraint applies only to the calling work-item. This scope is only useful for image operations, as all other operations within a work-item are already guaranteed to execute in program order.

- `memory_scope::sub_group, memory_scope::work_group`

  The memory ordering constraint applies only to work-items in the same sub-group or work-group as the calling work-item.

- `memory_scope::device`

  The memory ordering constraint applies only to work-items executing on the same device as the calling work-item.

- `memory_scope::system`

  The memory ordering constraint applies to all work-items in the system.

Barring restrictions imposed by the capabilities of a device, all memory scopes are valid arguments to all atomic and fence operations. However, a scope argument may be automatically demoted to a narrower scope in one of three situations:

1. If an atomic operation updates a value in work-group local memory, any scope broader than `memory_scope::work_group` is narrowed (because local memory is only visible to work-items in the same work-group).
2. If a device does not support USM, specifying `memory_scope::system` is always equivalent to `memory_scope::device` (because buffers cannot be accessed concurrently by multiple devices).
3. If an atomic operation uses `memory_order::relaxed`, there are no ordering guarantees, and the memory scope argument is effectively ignored.

Querying Device Capabilities

To ensure compatibility with devices supported by previous versions of SYCL and to maximize portability, SYCL supports OpenCL 1.2 devices and other hardware that may not be capable of supporting the full C++ memory model (e.g., certain classes of embedded devices). SYCL provides device queries to help us reason about the memory order(s) and memory scope(s) supported by the devices available in a system:

- `atomic_memory_order_capabilities`

  Return a list of all memory orderings supported by atomic operations on a specific device. All devices are required to support at least `memory_order::relaxed`.

- `atomic_fence_order_capabilities`

  Return a list of all memory orderings supported by fence operations on a specific device. All devices are required to support at least `memory_order::relaxed`, `memory_order::acquire`, `memory_order::release`, and `memory_order::acq_rel`. Note that the minimum requirement for fences is stronger than the minimum requirement for atomic operations, since such fences are essential for reasoning about memory order in the presence of barriers.

- `atomic_memory_scope_capabilities`

  `atomic_fence_scope_capabilities`

  Return a list of all memory scopes supported by atomic and fence operations on a specific device. All devices are required to support at least `memory_order::work_group`.

It may be difficult at first to remember which memory orders and scopes are supported for which combinations of function and device capability. In practice, we can avoid much of this complexity by following one of the two development approaches outlined in the following:

1. **Develop applications with sequential consistency and system fences.**

   Only consider adopting less strict memory orders during performance tuning.

2. **Develop applications with relaxed consistency and work-group fences.**

   Only consider adopting more strict memory orders and broader memory scopes where required for correctness.

The first approach ensures that the semantics of all atomic operations and fences match the default behavior of C++. This is the simplest and least error-prone option but has the worst performance and portability characteristics.

The second approach is more aligned with the default behavior of previous versions of SYCL and languages like OpenCL. Although more complicated—since it requires that we become more familiar with the different memory orders and scopes—it ensures that the majority of the SYCL code we write will work on any device without performance penalties.

Barriers and Fences

All previous usages of barriers and fences in the book so far have ignored the issue of memory order and scope, by relying on default behavior.

By default, every group barrier in SYCL acts as an acquire-release fence to all address spaces accessible by the calling work-item and makes preceding writes visible to at least all other work-items in the same group (as defined by the group's `fence_scope` member variable). This ensures memory consistency within a group of work-items after a barrier, in line with our intuition of what it means to synchronize (and the definition of the *synchronizes-with* relation in C++). It is possible to override this default behavior by passing an explicit `memory_scope` argument to the `group_barrier` function.

The `atomic_fence` function gives us even more fine-grained control than this, allowing work-items to execute fences specifying both a memory order and scope.

Atomic Operations in SYCL

SYCL provides support for many kinds of atomic operations on a variety of data types. All devices are guaranteed to support atomic versions of common operations (e.g., loads, stores, arithmetic operators), as well as the atomic *compare-and-swap* operations required to implement lock-free algorithms. The language defines these operations for all fundamental integer, floating-point, and pointer types—all devices must support these operations for 32-bit types, but 64-bit-type support is optional.

The `atomic` Class

The `std::atomic` class from C++11 provides an interface for creating and operating on atomic variables. Instances of the atomic class own their data, cannot be moved or copied, and can only be updated using atomic operations. These restrictions significantly reduce the chances of using the class incorrectly and introducing undefined behavior. Unfortunately, they also prevent the class from being used in SYCL kernels—it is impossible to create atomic objects on the host and transfer them to the device! We are free to continue using `std::atomic` in our host code, but attempting to use it inside of device kernels will result in a compiler error.

ATOMIC CLASS DEPRECATED IN SYCL 2020

The SYCL 1.2.1 specification included a `cl::sycl::atomic` class that is loosely based on the `std::atomic` class from C++11. We say loosely because there are some differences between the interfaces of the two classes, most notably that the SYCL 1.2.1 version does not own its data and defaults to a relaxed memory ordering.

The `cl::sycl::atomic` class is deprecated in SYCL 2020. The `atomic_ref` class (covered in the next section) should be used in its place.

The `atomic_ref` Class

The `std::atomic_ref` class from C++20 provides an alternative interface for atomic operations which provides greater flexibility than `std::atomic`. The biggest difference between the two classes is that instances of `std::atomic_ref` do not own their data but are instead constructed from an existing non-atomic variable. Creating an atomic reference effectively acts as a promise that the referenced variable will only be accessed atomically for the lifetime of the reference. These are exactly the semantics needed by SYCL, since they allow us to create non-atomic data on the host, transfer that data to the device, and treat it as atomic data only after it has been transferred. The `atomic_ref` class used in SYCL kernels is therefore based on `std::atomic_ref`.

We say *based on* because the SYCL version of the class includes three additional template arguments as shown in Figure 19-11.

```
template <typename T, memory_order DefaultOrder,
          memory_scope DefaultScope,
          access::address_space AddressSpace>
class atomic_ref {
 public:
  using value_type = T;
  static constexpr size_t required_alignment =
      /* implementation-defined */;
  static constexpr bool is_always_lock_free =
      /* implementation-defined */;
  static constexpr memory_order default_read_order =
      memory_order_traits<DefaultOrder>::read_order;
  static constexpr memory_order default_write_order =
      memory_order_traits<DefaultOrder>::write_order;
  static constexpr memory_order
      default_read_modify_write_order = DefaultOrder;
  static constexpr memory_scope default_scope =
      DefaultScope;

  explicit atomic_ref(T& obj);
  atomic_ref(const atomic_ref& ref) noexcept;
};
```

Figure 19-11. *Constructors and static members of the `atomic_ref` class*

As discussed previously, the capabilities of different SYCL devices are varied. Selecting a default behavior for the atomic classes of SYCL is a difficult proposition: defaulting to C++ behavior (i.e., `memory_order::seq_cst`, `memory_scope::system`) limits code to executing only on the most capable of devices; on the other hand, breaking with C++ conventions and defaulting to the lowest common denominator (i.e., `memory_order::relaxed`, `memory_scope::work_group`) could lead to unexpected behavior when migrating existing C++ code. The design adopted by SYCL offers a compromise, allowing us to define our desired default behavior as part of an object's type (using the `DefaultOrder` and `DefaultScope` template arguments). Other orderings and scopes can be provided as runtime arguments to specific atomic operations as we see fit—the `DefaultOrder` and `DefaultScope` only impact operations where we do not or cannot override the default behavior (e.g., when using a shorthand operator like +=). The final (optional) template argument denotes the address space in which the referenced object is allocated. Note that if the final template argument is not specified, the referenced variable can be allocated in any address space—although specifying an address space here is optional, we recommend providing explicit address spaces (where possible) to give compilers more information and to avoid unwanted performance overheads.

An atomic reference provides support for different operations depending on the type of object that it references. The basic operations supported by all types are shown in Figure 19-12, providing the ability to atomically move data to and from memory.

```
void store(
    T operand, memory_order order = default_write_order,
    memory_scope scope = default_scope) const noexcept;
T operator=(
    T desired) const noexcept;  // equivalent to store

T load(memory_order order = default_read_order,
       memory_scope scope = default_scope) const noexcept;
operator T() const noexcept;  // equivalent to load

T exchange(
    T operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_weak(
    T &expected, T desired, memory_order success,
    memory_order failure,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_weak(
    T &expected, T desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_strong(
    T &expected, T desired, memory_order success,
    memory_order failure,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_strong(
    T &expected, T desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;
```

Figure 19-12. *Basic operations with* `atomic_ref` *for all types*

Atomic references to objects of integral and floating-point types extend the set of available atomic operations to include arithmetic operations, as shown in Figure 19-13 and Figure 19-14. Devices are required to support atomic floating-point types irrespective of whether they feature native support for floating-point atomics in hardware, and many devices are expected to emulate atomic floating-point addition using an atomic compare exchange. This emulation is an important part of providing

performance and portability in SYCL, and we should feel free to use floating-point atomics anywhere that an algorithm requires them—the resulting code will work correctly everywhere and will benefit from future improvements in floating-point atomic hardware without any modification!

```
Integral fetch_add(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_sub(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_and(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_or(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_min(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_max(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral operator++(int) const noexcept;
Integral operator--(int) const noexcept;
Integral operator++() const noexcept;
Integral operator--() const noexcept;
Integral operator+=(Integral) const noexcept;
Integral operator-=(Integral) const noexcept;
Integral operator&=(Integral) const noexcept;
Integral operator|=(Integral) const noexcept;
```

Figure 19-13. *Additional operations with* `atomic_ref` *only for integral types*

```
Floating fetch_add(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_sub(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_min(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_max(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating operator+=(Floating) const noexcept;
Floating operator-=(Floating) const noexcept;
```

Figure 19-14. *Additional operations with* `atomic_ref` *only for floating-point types*

Using Atomics with Buffers

As discussed in the previous section, there is no way in SYCL to allocate atomic data and move it between the host and device. To use atomic operations in conjunction with buffers, we must create a buffer of non-atomic data to be transferred to the device and then access that data through an atomic reference.

```
q.submit([&](handler& h) {
  accessor acc{buf, h};
  h.parallel_for(N, [=](id<1> i) {
    int j = i % M;
    atomic_ref<int, memory_order::relaxed,
               memory_scope::system,
               access::address_space::global_space>
        atomic_acc(acc[j]);
    atomic_acc += 1;
  });
});
```

Figure 19-15. *Accessing a buffer via an explicitly created* `atomic_ref`

The code in Figure 19-15 is an example of expressing atomicity in SYCL using an explicitly created atomic reference object. The buffer stores normal integers, and we require an accessor with both read and write permissions. We can then create an instance of `atomic_ref` for each data access, using the `+=` operator as a shorthand alternative for the `fetch_add` member function.

This pattern is useful if we want to mix atomic and non-atomic accesses to a buffer within the same kernel, to avoid paying the performance overheads of atomic operations when they are not required. If we know that only a subset of the memory locations in the buffer will be accessed concurrently by multiple work-items, we only need to use atomic references when accessing that subset. Or, if we know that work-items in the same work-group only concurrently access local memory during one stage of a kernel (i.e., between two work-group barriers), we only need to use atomic references during that stage. When mixing atomic and non-atomic accesses like this, it is important to pay attention to object lifetimes—while any `atomic_ref` referencing a specific object exists, all accesses to that object must occur (atomically) via an instance of `atomic_ref`.

Using Atomics with Unified Shared Memory

As shown in Figure 19-16 (reproduced from Figure 19-7), we can construct atomic references from data stored in USM in exactly the same way as we could for buffers. Indeed, the only difference between this code and the code shown in Figure 19-15 is that the USM code does not require buffers or accessors.

```
q.parallel_for(N, [=](id<1> i) {
   int j = i % M;
   atomic_ref<int, memory_order::relaxed,
              memory_scope::system,
              access::address_space::global_space>
       atomic_data(data[j]);
   atomic_data += 1;
 }).wait();
```

Figure 19-16. *Accessing a USM allocation via an explicitly created* `atomic_ref`

Using Atomics in Real Life

The potential usages of atomics are so broad and varied that it would be impossible for us to provide an example of each usage in this book. We have included two representative examples, with broad applicability across domains:

1. Computing a histogram
2. Implementing device-wide synchronization

Computing a Histogram

The code in Figure 19-17 demonstrates how to use relaxed atomics in conjunction with work-group barriers to compute a histogram. The kernel is split by the barriers into three phases, each with their own atomicity requirements. Remember that the barrier acts both as a synchronization point and an acquire-release fence—this ensures that any reads and writes in one phase are visible to all work-items in the work-group in later phases.

The first phase sets the contents of some work-group local memory to zero. The work-items in each work-group update independent locations in work-group local memory by design—race conditions cannot occur, and no atomicity is required.

The second phase accumulates partial histogram results in local memory. Work-items in the same work-group may update the same locations in work-group local memory, but synchronization can be deferred until the end of the phase—we can satisfy the atomicity requirements using `memory_order::relaxed` and `memory_scope::work_group`.

The third phase contributes the partial histogram results to the total stored in global memory. Work-items in the same work-group are guaranteed to read from independent locations in work-group local memory, but may update the same locations in global memory—we no longer require atomicity for the work-group local memory and can satisfy the atomicity requirements for global memory using `memory_order::relaxed` and `memory_scope::system` as before.

```
q.submit([&](handler& h) {
   auto local = local_accessor<uint32_t, 1>{B, h};
   h.parallel_for(
       nd_range<1>{num_groups * num_items, num_items},
       [=](nd_item<1> it) {
         auto grp = it.get_group();

         // Phase 1: Work-items co-operate to zero local
         // memory
         for (int32_t b = it.get_local_id(0); b < B;
              b += it.get_local_range(0)) {
           local[b] = 0;
         }
         group_barrier(grp);  // Wait for all to be zeroed

         // Phase 2: Work-groups each compute a chunk of
         // the input. Work-items co-operate to compute
         // histogram in local memory
         const auto [group_start, group_end] =
             distribute_range(grp, N);
         for (int i = group_start + it.get_local_id(0);
              i < group_end; i += it.get_local_range(0)) {
           int32_t b = input[i] % B;
           atomic_ref<uint32_t, memory_order::relaxed,
                      memory_scope::work_group,
                      access::address_space::local_space>(local[b])++;
         }
         group_barrier(
             grp);  // Wait for all local histogram
                    // updates to complete

         // Phase 3: Work-items co-operate to update
         // global memory
         for (int32_t b = it.get_local_id(0); b < B;
              b += it.get_local_range(0)) {
           atomic_ref<uint32_t, memory_order::relaxed, memory_scope::system,
                      access::address_space::global_space>(histogram[b]) +=
               local[b];
         }
       });
 }).wait();
```

Figure 19-17. *Computing a histogram using atomic references in different memory spaces*

Implementing Device-Wide Synchronization

Back in Chapter 4, we warned against writing kernels that attempt to synchronize work-items across work-groups. However, we fully expect several readers of this chapter will be itching to implement their own device-wide synchronization routines atop of atomic operations and that our warnings will be ignored.

Device-wide synchronization is currently not portable and is best left to expert programmers. Future versions of SYCL will address this.

The code discussed in this section is dangerous and should not be expected to work on all devices, because of potential differences in device hardware features and SYCL implementations. The memory ordering guarantees provided by atomics are orthogonal to forward progress guarantees, and, at the time of writing, work-group scheduling in SYCL is completely implementation-defined. Formalizing the concepts and terminology required to describe SYCL's ND-range execution model and the forward progress guarantees associated with work-items, sub-groups, and work-groups is currently an area of active academic research—future versions of SYCL are expected to build on this work to provide additional scheduling queries and controls. For now, these topics should be considered expert-only.

Figure 19-18 shows a simple implementation of a device-wide latch (a single-use barrier), and Figure 19-19 shows a simple example of its usage. Each work-group elects a single work-item to signal arrival of the group at the latch and await the arrival of other groups using a naïve spin-loop, while the other work-items wait for the elected work-item using a work-group barrier. It is this spin-loop that makes device-wide synchronization unsafe; if any work-groups have not yet begun executing or the currently executing work-groups are not scheduled fairly, the code may deadlock.

Relying on memory order alone to implement synchronization primitives may lead to deadlocks in the absence of sufficiently strong forward progress guarantees!

For the code to work correctly, the following three conditions must hold:

1. The atomic operations must use memory orders at least as strict as those shown, to guarantee that the correct fences are generated.
2. The elected leader of each work-group in the ND-range must make progress independently of the leaders in other work-groups, to avoid a single work-item spinning in the loop from starving other work-items that have yet to increment the counter.
3. The device must be capable of executing all work-groups in the ND-range simultaneously, with strong forward progress guarantees, in order to ensure that the elected leaders of every work-group in the ND-range eventually reach the latch.

```
struct device_latch {
  explicit device_latch(size_t num_groups)
      : counter(0), expected(num_groups) {}

  template <int Dimensions>
  void arrive_and_wait(nd_item<Dimensions>& it) {
    auto grp = it.get_group();
    group_barrier(grp);
    // Elect one work-item per work-group to be involved in
    // the synchronization. All other work-items wait at the
    // barrier after the branch.
    if (grp.leader()) {
      atomic_ref<size_t, memory_order::acq_rel,
                 memory_scope::device,
                 access::address_space::global_space>
          atomic_counter(counter);

      // Signal arrival at the barrier.
      // Previous writes should be visible to all work-items
      // on the device.
      atomic_counter++;

      // Wait for all work-groups to arrive.
      // Synchronize with previous releases by all
      // work-items on the device.
      while (atomic_counter.load() != expected) {
      }
    }
    group_barrier(grp);
  }

  size_t counter;
  size_t expected;
};
```

Figure 19-18. *Building a simple device-wide latch on top of atomic references*

```
// Allocate a one-time-use device_latch in USM
void* ptr = sycl::malloc_shared(sizeof(device_latch), q);
device_latch* latch = new (ptr) device_latch(num_groups);
q.submit([&](handler& h) {
   h.parallel_for(R, [=](nd_item<1> it) {
     // Every work-item writes a 1 to its location
     data[it.get_global_linear_id()] = 1;

     // Every work-item waits for all writes
     latch->arrive_and_wait(it);

     // Every work-item sums the values it can see
     size_t sum = 0;
     for (int i = 0; i < num_groups * items_per_group;
          ++i) {
       sum += data[i];
     }
     sums[it.get_global_linear_id()] = sum;
   });
 }).wait();
free(ptr, q);
```

Figure 19-19. *Using the device-wide latch from Figure 19-18*

Although this code is not guaranteed to be portable, we have included it here to highlight two key points: (1) SYCL is expressive enough to enable device-specific tuning, sometimes at the expense of portability; and (2) SYCL already contains the building blocks necessary to implement higher-level synchronization routines, which may be included in a future version of the language.

Summary

This chapter provided a high-level introduction to memory model and atomic classes. Understanding how to use (and how not to use!) these classes is key to developing correct, portable, and efficient parallel programs.

Memory models are an overwhelmingly complex topic, and our focus here has been on establishing a base for writing real applications. If more information is desired, there are several websites, books, and talks dedicated to memory models referenced in the following.

For More Information

- A. Williams, C++ Concurrency in Action: Practical Multithreading, Manning, 2012, 978-1933988771
- H. Sutter, "atomic<> Weapons: The C++ Memory Model and Modern Hardware", herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/
- H-J. Boehm, "Temporarily discourage memory_order_consume," wg21.link/p0371
- C++ Reference, "std::atomic," en.cppreference.com/w/cpp/atomic/atomic
- C++ Reference, "std::atomic_ref," en.cppreference.com/w/cpp/atomic/atomic_ref