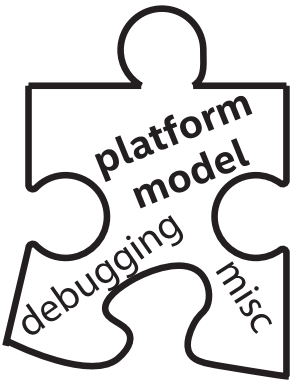
CHAPTER 13

Practical Tips



This chapter is home to a number of pieces of useful information, practical tips, advice, and techniques that have proven useful when programming SYCL and using DPC++. None of these topics are covered exhaustively, so the intent is to raise awareness and encourage learning more as needed.

Getting a DPC++ Compiler and Code Samples

Chapter 1 covers how to get the DPC++ compiler (oneapi.com/implementations or github.com/intel/llvm) and where to get the code samples (www.apress.com/9781484255735—look for Services for this book: Source Code). This is mentioned again to emphasize how useful

J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-5574-2_13

it can be to try the examples (including making modifications!) to gain hands-on experience. Join the club of those who know what the code in Figure 1-1 actually prints out!

Online Forum and Documentation

The Intel Developer Zone hosts a forum for discussing the DPC++ compiler, DPC++ Library (Chapter 18), DPC++ Compatibility Tool (for CUDA migration—discussed later in this chapter), and gdb included in the oneAPI toolkit (this chapter touches on debugging too). This is an excellent place to post questions about writing code, including suspected compiler bugs. You will find posts from some of the book authors on this forum doing exactly that, especially while writing this book. The forum is available online at `https://software.intel.com/en-us/forums/oneapi-data-parallel-c-compiler`.

The online oneAPI DPC++ language reference is a great resource to find a complete list of the classes and member definitions, details on compiler options, and more.

Platform Model

A SYCL or DPC++ compiler is designed to act and feel like any other C++ compiler we have ever used. A notable difference is that a regular C++ compiler produces code only for a CPU. It is worth understanding the inner workings, at a high level, that enable a compiler to produce code for a host CPU *and* devices.

The platform model (Figure 13-1), used by SYCL and DPC++, specifies a host that coordinates and controls the compute work that is performed on the devices. Chapter 2 describes how to assign work to devices, and Chapter 4 dives into how to program devices. Chapter 12 describes using the platform model at various levels of specificity.

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

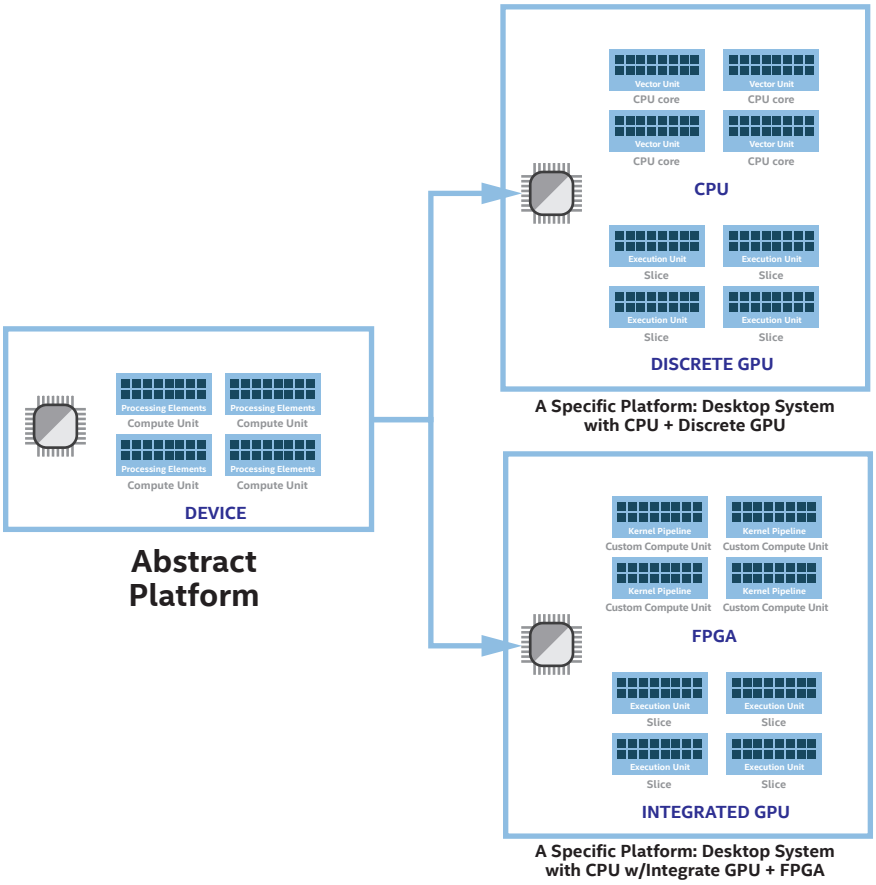


Figure 13-1. *Platform model: Can be used abstractly or with specificity*

Multiarchitecture Binaries

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

A multiarchitecture binary (aka a *fat binary*) is a single binary file that has been expanded to include all the compiled and intermediate code needed for our heterogeneous machine. The concept of multiarchitecture binaries is not new. For example, some operating systems support multiarchitecture 32-bit and 64-bit libraries and executables. A multiarchitecture binary acts like any other `a.out` or `A.exe` we are used to—but it contains everything needed for a heterogeneous machine. This helps to automate the process of picking the right code to run for a particular device. As we discuss next, one possible form of the device code in a fat binary is an intermediate format that defers the final creation of device instructions until runtime.

Compilation Model

The single-source nature of SYCL and DPC++ allows compilations to act and feel like regular C++ compilations. There is no need for us to invoke additional passes for devices or deal with bundling device and host code. That is all handled automatically for us by the compiler. Of course, understanding the details of what is happening can be important for several reasons. This is useful knowledge if we want to target specific architectures more effectively, and it is important to understand if we need to debug a failure happening in the compilation process.

We will review the compilation model so that we are educated for when that knowledge is needed. Since the compilation model supports code that executes on both a host and potentially several devices simultaneously, the commands issued by the compiler, linker, and other supporting tools are more complicated than the C++ compilations we are used to (targeting only one architecture). Welcome to the heterogeneous world!

This heterogeneous complexity is intentionally hidden from us by the DPC++ compiler and "just works."

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

The compiler can only compile ahead of time if the device target is known ahead of time (at the time when we compile our program). Deferring for just-in-time compilation gives more flexibility, but requires the compiler and the runtime to perform additional work while our application is running.

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

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

We can ask the compiler to compile ahead-of-time for specific devices or classes of devices. This has the advantage of saving runtime, but it has the disadvantage of added compile time and fatter binaries! Code that is compiled ahead-of-time is not as portable as just-in-time because it cannot adjust at runtime. We can include both in our binary to get the benefits of both.

Compiling for a specific device ahead-of-time also helps us to check at build time that our program should work on that device. With just-in-time compilation, it is possible that a program will fail to compile at runtime (which can be caught using the mechanisms in Chapter 5). There are a few debugging tips for this in the upcoming "Debugging" section of this chapter, and Chapter 5 details how these errors can be caught at runtime to avoid requiring that our applications abort.

Figure 13-2 illustrates the DPC++ compilation process from source code to fat binary (executable). Whatever combinations we choose are combined into a fat binary. The fat binary is employed by the runtime when the application executes (and it is the binary that we execute on the host!). At times, we may want to compile device code for a particular device in a separate compile. We would want the results of such a separate compilation to eventually be combined into our fat binary. This can be very useful for FPGA development when full compile (doing a full synthesis place-and-route) times can be very long and is in fact a requirement for FPGA development to avoid requiring the synthesis tools to be installed on a runtime system. Figure 13-3 shows the flow of the bundling/unbundling activity supported for such needs. We always have the option to compile everything at once, but during development, the option to break up compilation can be very useful.

Every SYCL and DPC++ compiler has a compilation model with the same goal, but the exact implementation details will vary. The diagrams shown here are for the DPC++ compiler toolchain.

One DPC++-specific component is shown in Figure 13-2 as the *integration header generator* that will not be mentioned again in this book. We can program without ever needing to know what it is or what it does. Nevertheless, to satisfy the curious, here is a little information: The integration header generator generates a header file providing information about SYCL kernels found in the translation unit. This includes how the names of SYCL kernel types map to symbolic names and information about kernel parameters and their locations within the corresponding lambda or functor object created by the compiler to capture them. The integration header is the mechanism used to implement the convenient style of kernel invocation from host code via C++ lambda/functor objects, which frees us from the time-consuming task of setting individual arguments, resolving kernels by name, and so on.

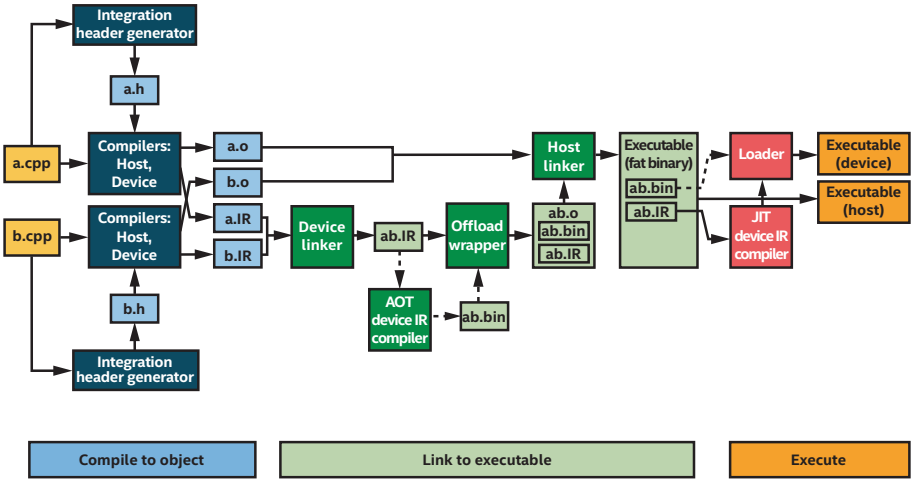


Figure 13-2. *Compilation process: Ahead-of-time and just-in-time options*

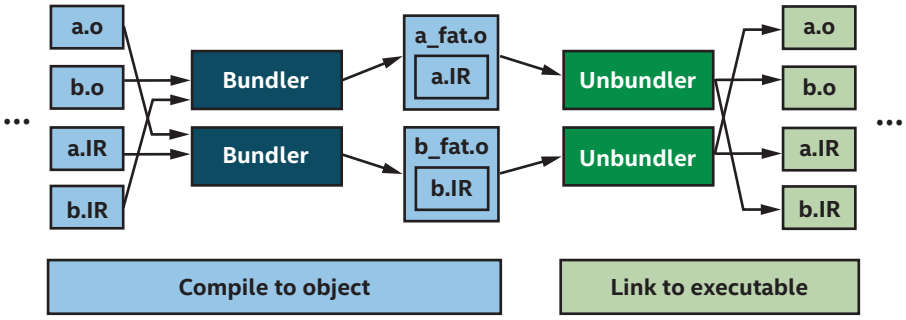


Figure 13-3. *Compilation process: Offload bundler/unbundler*

Adding SYCL to Existing C++ Programs

Adding the appropriate exploitation of parallelism to an existing C++ program is the first step to using SYCL. If a C++ application is already exploiting parallel execution, that may be a bonus, or it may be a headache. That is because the way we divide the work of an application into parallel execution greatly affects what we can do with it. When programmers talk

about *refactoring* a program, they are referring to rearranging the flow of execution and data within a program to get it ready to exploit parallelism. This is a complex topic that we will only touch briefly upon. There is no *one-size-fits-all* answer on how to prepare an application for parallelism, but there are some tips worth noting.

When adding parallelism to a C++ application, an easy approach to consider is to find an isolated point in the program where the opportunity for parallelism is the greatest. We can start our modification there and then continue to add parallelism in other areas as needed. A complicating factor is that refactoring (e.g., rearranging the program flow and redesigning data structures) may improve the opportunity for parallelism.

Once we find an isolated point in the program where the opportunity for parallelism is the greatest, we will need to consider how to use SYCL at that point in the program. That is what the rest of the book teaches.

At a high level, the key steps for introducing parallelism consist of

1. Safety with concurrency (commonly called *thread safety* in conventional CPU programming): Adjusting all shared mutable data (data that can change and is shared concurrently) to be used concurrently
2. Introducing concurrency and/or parallelism
3. Tuning for parallelism (best scaling, optimizing for throughput or latency)

It is important to consider step #1 first. Many applications have already been refactored for concurrency, but many have not. With SYCL as the sole source of parallelism, we focus on safety for the data being used within kernels and possibly shared with the host. If we have other techniques in our program (OpenMP, MPI, TBB, etc.) that introduce parallelism, that is an additional concern on top of our SYCL programming. It is important to note that it is okay to use multiple techniques inside a single program—SYCL

does not need to be the only source of parallelism within a program. This book does not cover the advanced topic of mixing with other parallelism techniques.

Debugging

This section conveys some modest debugging advice, to ease the challenges unique to debugging a parallel program, especially one targeting a heterogeneous machine.

We should never forget that we have the option to debug our applications while they are running on the host device. This debugging tip is described as Method#2 in Chapter 2. Because the architectures of devices often include fewer debugging hooks, tools can often probe code on a host more precisely. Another advantage of running *everything* on the host is that many errors relating to synchronization will disappear, including moving memory back and forth between the host and devices. While we eventually need to debug all such errors, this can allow incremental debugging so we can resolve some bugs before others.

Debugging tip Running on the host device is a powerful debugging tool.

Parallel programming errors, specifically data races and deadlocks, are generally easier for tools to detect and eliminate when running all code on the host. Much to our chagrin, we will most often see program failures from such parallel programming errors when running on a combination of host and devices. When such issues strike, it is very useful to remember that pulling back to host-only is a powerful debugging tool. Thankfully, SYCL and DPC++ are carefully designed to keep this option available to us and easy to access.

Debugging tip If a program is deadlocking, check that the host accessors are being destroyed properly.

The following DPC++ compiler options are a good idea when we start debugging:

- `-g`: Put debug information in the output.
- `-ferror-limit=1`: Maintain sanity when using C++ with template libraries such as SYCL/DPC++.
- `-Werror -Wall -Wpedantic`: Have the compiler enforce good coding to help avoid producing incorrect code to debug at runtime.

We really do not need to get bogged down fixing pedantic warnings just to use DPC++, so choosing to not use `-Wpedantic` is understandable.

When we leave our code to be compiled just-in-time during runtime, there is code we can inspect. This is *highly dependent* on the layers used by our compiler, so looking at the compiler documentation for suggestions is a good idea.

Debugging Kernel Code

While debugging kernel code, start by running on the host device (as advised in Chapter 2). The code for device selectors in Chapter 2 can easily be modified to accept runtime options, or compiler-time options, to redirect work to the host device when we are debugging.

When debugging kernel code, SYCL defines a C++-style `stream` that can be used within a kernel (Figure 13-4). DPC++ also offers an experimental implementation of a C-style `printf` that has useful capabilities, with some restrictions. Additional details are in the online oneAPI DPC++ language reference.

```
Q.submit([&](handler &h){
  stream out(1024, 256, h);
  h.parallel_for(range{8}, [=](id<1> idx){
    out << "Testing my sycl stream (this is work-item ID:" << idx << ")\n";
  });
});
```

Figure 13-4. *`sycl::stream`*

When debugging kernel code, experience encourages that we put breakpoints before `parallel_for` or inside `parallel_for`, but not actually on the `parallel_for`. A breakpoint placed on a `parallel_for` can trigger a breakpoint multiple times even after performing the next operation. This C++ debugging advice applies to many template expansions like those in SYCL, where a breakpoint on the template call will translate into a complicated set of breakpoints when it is expanded by the compiler. There may be ways that implementations can ease this, but the key point here is that we can avoid some confusion on all implementations by not setting the breakpoint precisely on the `parallel_for` itself.

Debugging Runtime Failures

When a runtime error occurs while compiling just-in-time, we are either dealing with a compiler/runtime bug, or we have accidentally programmed nonsense that was not detected until it tripped up the runtime and created difficult-to-understand runtime error messages. It can be a bit intimidating to dive into these bugs, but even a cursory look may allow us to get a better idea of what caused a particular issue. It might yield some additional knowledge that will guide us to avoid the issue, or it may just help us submit a short bug report to the compiler team. Either way, knowing that some tools exist to help can be important.

Output from our program that indicates a runtime failure may look like this:

```
origin>: error: Invalid record (Producer: 'LLVM9.0.0' Reader:
'LLVM 9.0.0')
terminate called after throwing an instance of
'cl::sycl::compile_program_error'
```

Seeing this throw noted here lets us know that our host program could have been constructed to catch this error. While that may not solve our problem, it does mean that runtime compiler failures do not need to abort our application. Chapter 5 dives into this topic.

When we see a runtime failure and have any difficulty debugging it quickly, it is worth simply trying a rebuild using ahead-of-time compilations. If the device we are targeting has an ahead-of-time compilation option, this can be an easy thing to try that may yield easier-to-understand diagnostics. If our errors can be seen at compile time instead of JIT or runtime, often much more useful information will be found in the error messages from the compiler instead of the small amount of error information we usually see from a JIT or the runtime. For specific options, check the online oneAPI DPC++ documentation for *ahead-of-time compilation*.

When our SYCL programs are running on top of an OpenCL runtime and using the OpenCL backend, we can run our programs with the OpenCL Intercept Layer: github.com/intel/opencl-intercept-layer. This is a tool that can inspect, log, and modify OpenCL commands that an application (or higher-level runtime) is generating. It supports a lot of controls, but good ones to set initially are `ErrorLogging`, `BuildLogging`, and maybe `CallLogging` (though it generates a lot of output). Useful dumps are possible with `DumpProgramSPIRV`. The OpenCL Intercept Layer is a separate utility and is not part of any specific OpenCL implementation, so it works with many SYCL compilers.

For suspected compiler issues on Linux systems with Intel GPUs, we can dump intermediate compiler output from the Intel Graphics Compiler. We do this by setting the environment variable `IGC_ShaderDumpEnable` equal to 1 (for some output) or the environment variable `IGC_ShaderDumpEnableAll` to 1 (for lots and lots of output). The dumped output goes in `/tmp/IntelIGC`. This technique may not apply to all builds of the graphics drivers, but it is worth a try to see if it applies to our system.

Figure 13-5 lists these and a few additional environment variables (supported on Windows and Linux) supported by compilers or runtimes to aid in advanced debugging. These are DPC++ implementation-dependent advanced debug options that exist to inspect and control the compilation model. They are not discussed or utilized in this book. The online oneAPI DPC++ language reference is a good place to learn more.

These options are not described more within this book, but they are mentioned here to open up this avenue of advanced debugging as needed. These options *may* give us insight into how to work around an issue or bug. It is possible that our source code is inadvertently triggering an issue that can be resolved by correcting the source code. Otherwise, the use of these options is for very advanced debugging of the compiler itself. Therefore, they are associated more with compiler developers than with users of the compiler. Some advanced users find these options useful; therefore, they are mentioned here and never again in this book. To dig deeper, the GitHub for DPC++ has a document for all environment variables under llvm / sycl / doc / EnvironmentVariables.md.

Debugging tip When other options are exhausted and we need to debug a runtime issue, we look for dump tools that might give us hints toward the cause.

Environment variables	Value	Description
SYCL_PI_TRACE	1 (basic), 2 (advanced), -1 (all)	Runtime: Value of 1 enables tracing of Runtime Plugin Interface (PI) for plugin and device discovery; Value of 2 enables tracing of all PI calls. Value of -1 unleashes all levels of tracing.
SYCL_PRINT_EXECUTION_GRAPH	always (or ask to dump only select files by specifying: before_addCG, after_addCG, before_addCopyBack, after_addCopyBack, before_addHostAcc, or after_addHostAcc)	Runtime: create text files (with DOT extension) tracing the execution graph. Relatively easy to browse traces of what is happening during runtime.
CL_CONFIG_USE_VECTORIZER	true or false	Runtime: ask the Intel CPU compiler to enable or disable vectorizer.
CL_CONFIG_CPU_TARGET_ARCH	skx,core-avx2	Runtime: ask the Intel CPU compiler to generate code for processors that support Intel Advanced Vector Extensions (Intel AVX512 and AVX2),
CL_CONFIG_DUMP_ASM	true or false	Runtime: ask the Intel CPU compiler to dump CPU assembly code.
IGC_ShaderDumpEnable	0 or 1	Linux only. Runtime: ask the Intel Graphics Compiler (JIT) to dump some information.
IGC_ShaderDumpEnableAll	0 or 1	Linux only. Runtime: ask the Intel Graphics Compiler (JIT) to dump *lots* of information.
SYCL_DUMP_IMAGES	true or false	Compile time: Request via SYCL_DUMP_IMAGES=1 that compiler to dump a SPV file containing the intermediate code which is passed to JIT compiler during execution.
SYCL_USE_KERNEL_SPV	<device binary>	Runtime: Load device image from the specified file. If runtime is unable to read the file, cl::sycl::runtime_error exception is thrown.

Figure 13-5. *DPC++ advanced debug options*

Initializing Data and Accessing Kernel Outputs

In this section, we dive into a topic that causes confusion for new users of SYCL and that leads to the most common (in our experience) first bugs that we encounter as new SYCL developers.

Put simply, when we create a buffer from a host memory allocation (e.g., array or vector), we can't access the host allocation directly until the buffer has been destroyed. The buffer owns any host allocation passed to

it at construction time, for the buffer's entire lifetime. There are rarely used mechanisms that *do* let us access the host allocation while a buffer is still alive (e.g., buffer mutex), but those advanced features don't help with the early bugs described here.

If we construct a buffer from a host memory allocation, we must not directly access the host allocation until the buffer has been destroyed! While it is alive, the buffer owns the allocation.

A common bug appears when the host program accesses a host allocation while a buffer still owns that allocation. All bets are off once this happens, because we don't know what the buffer is using the allocation for. Don't be surprised if the data is incorrect—the kernels that we're trying to read the output from may not have even started running yet! As described in Chapters 3 and 8, SYCL is built around an asynchronous task graph mechanism. Before we try to use output data from task graph operations, we need to be sure that we have reached synchronization points in the code where the graph has executed and made data available to the host. Both buffer destruction and creation of host accessors are operations that cause this synchronization.

Figure 13-6 shows a common pattern of code that we often write, where we cause a buffer to be destroyed by closing the block scope that it was defined within. By causing the buffer to go out of scope and be destroyed, we can then safely read kernel results through the original host allocation that was passed to the buffer constructor.

```
constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create host containers to initialize on the host
std::vector<int> in_vec(N), out_vec(N);

// Initialize input and output vectors
for (int i=0; i < N; i++) in_vec[i] = i;
std::fill(out_vec.begin(), out_vec.end(), 0);

// Nuance: Create new scope so that we can easily cause
// buffers to go out of scope and be destroyed
{

  // Create buffers using host allocations (vector in this case)
  buffer in_buf{in_vec}, out_buf{out_vec};

  // Submit the kernel to the queue
  q.submit([&](handler& h) {
    accessor in{in_buf, h};
    accessor out{out_buf, h};

    h.parallel_for(range{N}, [=](id<1> idx) {
      out[idx] = in[idx];
    });
  });

// Close the scope that buffer is alive within!  Causes
// buffer destruction which will wait until the kernels
// writing to buffers have completed, and will copy the
// data from written buffers back to host allocations (our
// std::vectors in this case).  After the buffer destructor
// runs, caused by this closing of scope, then it is safe
// to access the original in_vec and out_vec again!
}

// Check that all outputs match expected value
// WARNING: The buffer destructor must have run for us to safely
// use in_vec and out_vec again in our host code.  While the buffer
// is alive it owns those allocations, and they are not safe for us
// to use!  At the least they will contain values that are not up to
// date.  This code is safe and correct because the closing of scope
// above has caused the buffer to be destroyed before this point
// where we use the vectors again.
for (int i=0; i<N; i++)
  std::cout << "out_vec[" << i << "]=" << out_vec[i] << "\n";
```

Figure 13-6. *Common pattern—buffer creation from a host allocation*

There are two common reasons to associate a buffer with existing host memory like in Figure 13-6:

1. To simplify initialization of data in a buffer. We can just construct the buffer from host memory that we (or another part of the application) have already initialized.

2. To reduce the characters typed because closing scope with a '}' is slightly more concise (though more error prone) than creating a `host_accessor` to the buffer.

If we use a host allocation to dump or verify the output values from a kernel, we need to put the buffer allocation into a block scope (or other scopes) so that we can control when it is destroyed. We must then make sure that the buffer is destroyed before we access the host allocation to obtain the kernel output. Figure 13-6 shows this done correctly, while Figure 13-7 shows a common bug where the output is accessed while the buffer is still alive.

Advanced users may prefer to use buffer destruction to return result data from kernels into a host memory allocation. But for most users, and especially new developers, it is recommended to use scoped host accessors.

```
constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create host containers to initialize on the host
std::vector<int> in_vec(N), out_vec(N);

// Initialize input and output vectors
for (int i=0; i < N; i++) in_vec[i] = i;
std::fill(out_vec.begin(), out_vec.end(), 0);

// Create buffers using host allocations (vector in this case)
buffer in_buf{in_vec}, out_buf{out_vec};

// Submit the kernel to the queue
q.submit([&](handler& h) {
  accessor in{in_buf, h};
  accessor out{out_buf, h};

  h.parallel_for(range{N}, [=](id<1> idx) {
    out[idx] = in[idx];
  });
});

// BUG!!! We're using the host allocation out_vec, but the buffer out_buf
// is still alive and owns that allocation!  We will probably see the
// initialiation value (zeros) printed out, since the kernel probably
// hasn't even run yet, and the buffer has no reason to have copied
// any output back to the host even if the kernel has run.
for (int i=0; i<N; i++)
  std::cout << "out_vec[" << i << "]=" << out_vec[i] << "\n";
```

Figure 13-7. *Common bug: Reading data directly from host allocation during buffer lifetime*

Prefer to use host accessors instead of scoping of buffers, especially when getting started.

To avoid these bugs, we recommend using host accessors instead of buffer scoping when getting started with SYCL and DPC++. Host accessors provide access to a buffer from the host, and once their constructor has

finished running, we are guaranteed that any previous writes (e.g., from kernels submitted before the `host_accessor` was created) to the buffer have executed and are visible. This book uses a mixture of both styles (i.e., host accessors and host allocations passed to the buffer constructor) to provide familiarity with both. Using host accessors tends to be less error prone when getting started. Figure 13-8 shows how a host accessor can be used to read output from a kernel, without destroying the buffer first.

```
constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create host containers to initialize on the host
std::vector<int> in_vec(N), out_vec(N);

// Initialize input and output vectors
for (int i=0; i < N; i++) in_vec[i] = i;
std::fill(out_vec.begin(), out_vec.end(), 0);

// Create buffers using host allocations (vector in this case)
buffer in_buf{in_vec}, out_buf{out_vec};

// Submit the kernel to the queue
q.submit([&](handler& h) {
  accessor in{in_buf, h};
  accessor out{out_buf, h};

  h.parallel_for(range{N}, [=](id<1> idx) {
    out[idx] = in[idx];
  });
});

// Check that all outputs match expected value
// Use host accessor!  Buffer is still in scope / alive
host_accessor A{out_buf};

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

Figure 13-8. *Recommendation: Use a host accessor to read kernel results*

Host accessors can be used whenever a buffer is alive, such as at both ends of a typical buffer lifetime—for initialization of the buffer content and for reading of results from our kernels. Figure 13-9 shows an example of this pattern.

```
constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create buffers of size N
buffer<int> in_buf{N}, out_buf{N};

// Use host accessors to initialize the data
{ // CRITICAL: Begin scope for host_accessor lifetime!
  host_accessor in_acc{ in_buf }, out_acc{ out_buf };
  for (int i=0; i < N; i++) {
    in_acc[i] = i;
    out_acc[i] = 0;
  }
} // CRITICAL: Close scope to make host accessors go out of scope!

// Submit the kernel to the queue
q.submit([&](handler& h) {
  accessor in{in_buf, h};
  accessor out{out_buf, h};

  h.parallel_for(range{N}, [=](id<1> idx) {
    out[idx] = in[idx];
  });
});

// Check that all outputs match expected value
// Use host accessor!  Buffer is still in scope / alive
host_accessor A{out_buf};

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

Figure 13-9. *Recommendation: Use host accessors for buffer initialization and reading of results*

One final detail to mention is that host accessors sometime cause an opposite bug in applications, because they also have a lifetime. While a `host_accessor` to a buffer is alive, the runtime will not allow that buffer to be used by any devices! The runtime does not analyze our host programs to determine when they *might* access a host accessor, so the only way for it to know that the host program has finished accessing a buffer is for the `host_accessor` destructor to run. As shown in Figure 13-10, this can cause applications to appear to hang if our host program is waiting for some kernels to run (e.g., `queue::wait()` or acquiring another host accessor) and if the DPC++ runtime is waiting for our earlier host accessor(s) to be destroyed before it can run kernels that use a buffer.

When using host accessors, be sure that they are destroyed when no longer needed to unlock use of the buffer by kernels or other host accessors.

```
constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create buffers using host allocations (vector in this case)
buffer<int> in_buf{N}, out_buf{N};

// Use host accessors to initialize the data
host_accessor in_acc{ in_buf }, out_acc{ out_buf };
for (int i=0; i < N; i++) {
  in_acc[i] = i;
  out_acc[i] = 0;
}

// BUG: Host accessors in_acc and out_acc are still alive!
// Later q.submits will never start on a device, because the
// runtime doesn't know that we've finished accessing the
// buffers via the host accessors.  The device kernels
// can't launch until the host finishes updating the buffers,
// since the host gained access first (before the queue submissions).
// This program will appear to hang!  Use a debugger in that case.

// Submit the kernel to the queue
q.submit([&](handler& h) {
  accessor in{in_buf, h};
  accessor out{out_buf, h};

  h.parallel_for(range{N}, [=](id<1> idx) {
    out[idx] = in[idx];
  });
});

std::cout <<
  "This program will deadlock here!!!  Our host_accessors used\n"
  " for data initialization are still in scope, so the runtime won't\n"
  " allow our kernel to start executing on the device (the host could\n"
  " still be initializing the data that is used by the kernel). "
  " The next line\n of code is acquiring a host accessor for"
  " the output, which will wait for\n the kernel to run first. "
  " Since in_acc and out_acc have not been\n"
  " destructed, the kernel is not safe for the runtime to run, "
  " and we deadlock.\n";

// Check that all outputs match expected value
// Use host accessor!  Buffer is still in scope / alive
host_accessor A{out_buf};

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

Figure 13-10. *Bug (hang!) from improper use of* `host_accessors`

Multiple Translation Units

When we want to call functions inside a kernel that are defined in a different translational unit, those functions need to be labeled with `SYCL_EXTERNAL`. Without this attribute, the compiler will only compile a function for use outside of device code (making it illegal to call that external function from within device code).

There are a few restrictions on `SYCL_EXTERNAL` functions that do not apply if we define the function within the same translation unit:

- `SYCL_EXTERNAL` can only be used on functions.
- `SYCL_EXTERNAL` functions cannot use raw pointers as parameter or return types. Explicit pointer classes must be used instead.
- `SYCL_EXTERNAL` functions cannot call a `parallel_for_work_item` method.
- `SYCL_EXTERNAL` functions cannot be called from within a `parallel_for_work_group` scope.

If we try to compile a kernel that is calling a function that is not inside the same translation unit and is not declared with `SYCL_EXTERNAL`, then we can expect a compile error similar to

```
error: SYCL kernel cannot call an undefined function
without SYCL_EXTERNAL attribute
```

If the function itself is compiled without a `SYCL_EXTERNAL` attribute, we can expect to see either a link or runtime failure such as

```
terminate called after throwing an instance of
'cl::sycl::compile_program_error'
...error: undefined reference to ...
```

DPC++ supports `SYCL_EXTERNAL`. SYCL does not require compilers to support `SYCL_EXTERNAL`; it is an *optional* feature in general.

Performance Implications of Multiple Translation Units

An implication of the compilation model (see earlier in this chapter) is that if we scatter our device code into multiple translation units, that may trigger more invocations of just-in-time compilation than if our device code is co-located. This is highly implementation dependent and is subject to changes over time as implementations mature.

Such effects on performance are minor enough to ignore through most of our development work, but when we get to fine-tuning to maximize code performance, there are two things we can consider to mitigate these effects: (1) group device code together in the same translation unit, and (2) use ahead-of-time compilation to avoid just-in-time compilation effects entirely. Since both of these require some effort on our part, we only do this when we have finished our development and are trying to squeeze every ounce of performance out of our application. When we do resort to this detailed tuning, it is worth testing changes to observe their effect on the exact SYCL implementation that we are using.

When Anonymous Lambdas Need Names

SYCL provides for assigning names defined as lambdas in case tools need it and for debugging purposes (e.g., to enable displays in terms of user-defined names). Throughout most of this book, anonymous lambdas have been used for kernels because names are not needed when using DPC++ (except for passing of compile options as described with lambda naming

discussion in Chapter 10). They are also made optional as of the SYCL 2020 provisional.

When we have an advanced need to mix SYCL tools from multiple vendors on a codebase, the tooling may require that we name lambdas. This is done by adding a `<class uniquename>` to the SYCL action construct in which the lambda is used (e.g., `parallel_for`). This naming allows tools from multiple vendors to interact in a defined way within a single compilation and can also help by displaying kernel names that we define within debug tools and layers.

Migrating from CUDA to SYCL

Migrating CUDA code to SYCL or DPC++ is not covered in detail in this book. There are tools and resources available that explore doing this. Migrating CUDA code is relatively straightforward since it is a kernel-based approach to parallelism. Once written in SYCL or DPC++, the new program is enhanced by its ability to target more devices than supported by CUDA alone. The newly enhanced program can still be targeted to NVIDIA GPUs using SYCL compilers with NVIDIA GPU support.

Migrating to SYCL opens the door to the diversity of devices supported by SYCL, which extends far beyond just GPUs.

When using the DPC++ Compatibility Tool, the `--report-type=`*`value`* option provides very useful statistics about the migrated code. One of the reviewers of this book called it a "beautiful flag provided by Intel `dpct`." The `--in-root` option can prove very useful when migrating CUDA code depending on source code organization of a project.

To learn more about CUDA migration, two resources are a good place to start:

- Intel's DPC++ Compatibility Tool transforms CUDA applications into DPC++ code (tinyurl.com/CUDAtoDPCpp).
- Codeplay tutorial "Migrating from CUDA to SYCL" (tinyurl.com/codeplayCUDAtoSYCL).

Summary

Popular culture today often refers to tips as *life hacks*. Unfortunately, programming culture often assigns a negative connotation to *hack*, so the authors refrained from naming this chapter "SYCL Hacks." Undoubtedly, this chapter has just touched the surface of what practical tips can be given for using SYCL and DPC++. More tips can be shared by all of us on the online forum as we learn together how to make the most out of SYCL with DPC++.