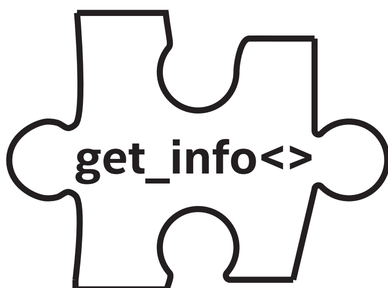
CHAPTER 12

Device Information



Chapter 2 introduced us to the mechanisms that direct work to a particular device—controlling *where code executes*. In this chapter, we explore how to adapt to the devices that are present at runtime.

We want our programs to be portable. In order to be portable, we need our programs to adapt to the capabilities of the device. We can parameterize our programs to only use features that are present and to tune our code to the particulars of devices. If our program is not designed to adapt, then bad things can happen including slow execution or program failures.

Fortunately, the creators of the SYCL specification thought about this and gave us interfaces to let us solve this problem. The SYCL specification defines a `device` class that encapsulates a device on which kernels may be executed. The ability to query the device class, so that our program can adapt to the device characteristics and capabilities, is the heart of what this chapter teaches.

© Intel Corporation 2021
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-5574-2_12

Many of us will start with having logic to figure out "Is there a GPU present?" to inform the choices our program will make as it executes. That is the start of what this chapter covers. As we will see, there is much more information available to help us make our programs robust and performant.

Parameterizing a program can help with correctness, portability, performance portability, and future proofing.

This chapter dives into the most important queries and how to use them effectively in our programs.

Device-specific properties are queryable using `get_info, but` DPC++ diverges from SYCL 1.2.1 in that it fully overloads `get_info` to alleviate the need to use `get_work_group_info` for work-group information that is really device-specific information. DPC++ does not support use of `get_work_group_info`. This change means that device-specific kernel and work-group properties are properly found as queries for device-specific properties (`get_info`). This corrects a confusing historical anomaly still present in SYCL 1.2.1 that was inherited from OpenCL.

Refining Kernel Code to Be More Prescriptive

It is useful to consider that our coding, kernel by kernel, will fall broadly into one of three categories:

- Generic kernel code: Run anywhere, not tuned to a specific class of device.
- Device type–specific kernel code: Run on a type of device (e.g., GPU, CPU, FPGA), not tuned to specific *models* of a device type. This is very useful because

many device types share common features, so it's safe to make some assumptions that would not apply to fully general code written for all devices.

- Tuned device-specific kernel code: Run on a type of device, with tuning that reacts to specific parameters of a device—this covers a broad range of possibilities from a small amount of tuning to very detailed optimization work.

It is our job as programmers to determine when different patterns (Chapter 14) are needed for different device types. We dedicate Chapters 14, 15, 16, and 17 to illuminating this important thinking.

It is most common to start by implementing generic kernel code to get it working. Chapter 2 specifically talks about what methods are easiest to debug when getting started with a kernel implementation. Once we have a kernel working, we may evolve it to target the capabilities of a specific device type or device model.

Chapter 14 offers a framework of thinking to consider parallelism first, before we dive into device considerations. It is our choice of pattern (aka algorithm) that dictates our code, and it is our job as programmers to determine when different patterns are needed for different devices. Chapters 15 (GPU), 16 (CPU), and 17 (FPGA) dive more deeply into the qualities that distinguish these device types and motivate a choice in pattern to use. It is these qualities that motivate us to consider writing distinct versions of kernels for different devices when the approaches (pattern choice) on different device types differ.

When we have a kernel written for a specific type of device (e.g., a specific CPU, GPU, FPGA, etc.), it is logical to adapt it to specific vendors or even models of such devices. Good coding style is to parameterize code based on features (e.g., item size support found from a device query).

We should write code to query parameters that describe the actual capabilities of a device instead of its marketing information; it is very bad programming practice to query the model number of a device and react to that—such code is less portable.

It is common to write a different kernel for each device type that we want to support (a GPU version of a kernel and an FPGA version of a kernel and maybe a generic version of a kernel). When we get more specific, to support a specific device vendor or even device model, we may benefit when we can parameterize a kernel rather than duplicate it. We are free to do either, as we see fit. Code cluttered with too many parameter adjustments may be hard to read or excessively burdened at runtime. It is common however that parameters can fit neatly into a single version of a kernel.

Parameterizing makes the most sense when the algorithm is broadly the same but has been tuned for the capabilities of a specific device. Writing a different kernel is much cleaner when using a completely different approach, pattern, or algorithm.

How to Enumerate Devices and Capabilities

Chapter 2 enumerates and explains five methods for choosing a device on which to execute. Essentially, Method#1 was the least prescriptive *run it somewhere,* and we evolve to the most prescriptive Method#5 which considered executing on a fairly precise model of a device from a family of devices. The enumerated methods in between gave a mix of flexibility and prescriptiveness. Figures 12-1, 12-2, and 12-3 help to illustrate how we can select a device.

Figure 12-1 shows that even if we allow the implementation to select a default device for us (Method#1 in Chapter 2), we can still query for information about the selected device.

Figure 12-2 shows how we can try to set up a queue using a specific device (in this case, a GPU), but fall back explicitly on the host if no GPU is available. This gives us some control of our device choice. If we simply used a default queue, we could end up with an unexpected device type (e.g., a DSP, FPGA). If we explicitly want to use the host device if there is no GPU device, this code does that for us. Recall that the host device is always guaranteed to exist, so we do not need to worry about using the `host_selector`.

It is not recommended that we use the solution shown in Figure 12-2. In addition to appearing a little scary and error prone, Figure 12-2 does not give us control over what GPU is selected because it is implementation dependent which GPU we get if more than one is available. Despite being both instructive and functional, there is a better way. It is recommended that we write custom device selectors as shown in the next code example (Figure 12-3).

Custom Device Selector

Figure 12-3 uses a custom device selector. Custom device selectors were first discussed in Chapter 2 as Method#5 for choosing where our code runs (Figure 2-15). The custom device selector causes its `operator()`, shown in Figure 12-3, to be invoked for each device available to the application. The device selected is the one that receives the highest score.[1] In this example, we will have a little fun with our selector:

[1]If our device selector returned only negative values, then the `my_selector()` would throw a `runtime_error` exception as expected on non-GPU systems in Figure 12-2. Since we return a positive value for the host, that cannot happen in Figure 12-3.

- Reject GPUs with a vendor name including the word "Martian" (return –1).
- Favor GPUs with a vendor name including the word "ACME" (return 824).
- Any other GPU is a good one (return 799).
- We pick the host device if no GPU is present (return 99).
- All other devices are ignored (return –1).

The next section, "Being Curious: `get_info<>`," dives into the rich information that `get_devices(), get_platforms()`, and `get_info<>` offer. Those interfaces open up any type of logic we might want to utilize to pick our devices, including the simple vendor name checks shown in Figures 2-15 and 12-3.

```
queue Q;

std::cout << "By default, we are running on "
  << Q.get_device().get_info<info::device::name>() << "\n";

// sample output:
// By default, we are running on Intel(R) Gen9 HD Graphics NEO.
```

Figure 12-1. *Device we have been assigned by default*

Queries about devices rely on installed software (special user-level drivers), to respond regarding a device. SYCL and DPC++ rely on this, just as an operating system needs drivers to access hardware—it is not sufficient that the hardware simply be installed in a machine.

```
auto GPU_is_available = false;

try {
  device testForGPU((gpu_selector()));
  GPU_is_available = true;
} catch (exception const& ex) {
  std::cout << "Caught this SYCL exception: " << ex.what() << std::endl;
}

auto Q = GPU_is_available ? queue(gpu_selector()) : queue(host_selector());

std::cout << "After checking for a GPU, we are running on:\n "
          << Q.get_device().get_info<info::device::name>() << "\n";

// sample output using a system with a GPU:
// After checking for a GPU, we are running on:
//  Intel(R) Gen9 HD Graphics NEO.
//
// sample output using a system with an FPGA accelerator, but no GPU:
// Caught this SYCL exception: No device of requested type available.
// ...(CL_DEVICE_NOT_FOUND)
// After checking for a GPU, we are running on:
//  SYCL host device.
```

Figure 12-2. *Using try-catch to select a GPU device if possible, host device if not*

```
class my_selector : public device_selector {
  public:
    int operator()(const device &dev) const {
      int score = -1;

      // We prefer non-Martian GPUs, especially ACME GPUs
      if (dev.is_gpu()) {
        if (dev.get_info<info::device::vendor>().find("ACME")
            != std::string::npos) score += 25;

        if (dev.get_info<info::device::vendor>().find("Martian")
            == std::string::npos) score += 800;
      }

      // Give host device points so it is used if no GPU is available.
      // Without these next two lines, systems with no GPU would select
      // nothing, since we initialize the score to a negative number above.
      if (dev.is_host()) score += 100;

      return score;
    }
};

int main() {
  auto Q = queue{ my_selector{} };

  std::cout << "After checking for a GPU, we are running on:\n "
    << Q.get_device().get_info<info::device::name>() << "\n";

  // Sample output using a system with a GPU:
  // After checking for a GPU, we are running on:
  //  Intel(R) Gen9 HD Graphics NEO.
  //
  // Sample output using a system with an FPGA accelerator, but no GPU:
  // After checking for a GPU, we are running on:
  //  SYCL host device.

  return 0;
}
```

Figure 12-3. *Custom device selector—our preferred solution*

Being Curious: get_info<>

In order for our program to “know” what devices are available at runtime, we can have our program query available devices from the device class, and then we can learn more details using `get_info<>` to inquire about a specific device. We provide a simple program, called *curious* (see Figure 12-4), that uses these interfaces to dump out information for us to look at directly. This can be very useful for doing a sanity check when developing or debugging a program that uses these interfaces. Failure of this program to work as expected can often tell us that the software drivers we need are not installed correctly. Figure 12-5 shows sample output from this program, with the high-level information about the devices that are present.

```
// Loop through available platforms
for (auto const& this_platform : platform::get_platforms() ) {
  std::cout << "Found platform: "
    << this_platform.get_info<info::platform::name>() << "\n";

  // Loop through available devices in this platform
  for (auto const& this_device : this_platform.get_devices() ) {
    std::cout << " Device: "
      << this_device.get_info<info::device::name>() << "\n";
  }
  std::cout << "\n";
}
```

Figure 12-4. *Simple use of device query mechanisms: curious.cpp*

```
% make curious
dpcpp curious.cpp -o curious

% ./curious
Found platform 1...
Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
Device: Intel(R) FPGA Emulation Device

Found platform 2...
Platform: Intel(R) OpenCL HD Graphics
Device: Intel(R) Gen9 HD Graphics NEO

Found platform 3...
Platform: Intel(R) OpenCL
Device: Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz

Found platform 4...
Platform: SYCL host platform
Device: SYCL host device
```

Figure 12-5. *Sample output from curious.cpp*

Being More Curious: Detailed Enumeration Code

We offer a program, which we have named verycurious.cpp (Figure 12-6), to illustrate some of the detailed information available using `get_info<>`. Again, we find ourselves writing code like this to help when developing or debugging a program. Figure 12-5 shows sample output from this program, with the lower-level information about the devices that are present.

Now that we have shown how to access the information, we will discuss the information fields that prove the most important to query and act upon in applications.

```
template <auto query, typename T>
void do_query( const T& obj_to_query, const std::string& name, int indent=4)
{
  std::cout << std::string(indent, ' ') << name << " is '"
    << obj_to_query.template get_info<query>() << "'\n";
}

// Loop through the available platforms
for (auto const& this_platform : platform::get_platforms() ) {
  std::cout << "Found Platform:\n";
  do_query<info::platform::name>(this_platform,
      "info::platform::name");
  do_query<info::platform::vendor>(this_platform,
      "info::platform::vendor");
  do_query<info::platform::version>(this_platform,
      "info::platform::version");
  do_query<info::platform::profile>(this_platform,
      "info::platform::profile");

  // Loop through the devices available in this plaform
  for (auto &dev : this_platform.get_devices() ) {
    std::cout << "  Device: "
              << dev.get_info<info::device::name>() << "\n";
    std::cout << "    is_host(): "
              << (dev.is_host() ? "Yes" : "No") << "\n";
    std::cout << "    is_cpu(): "
              << (dev.is_cpu() ? "Yes" : "No") << "\n";
    std::cout << "    is_gpu(): "
              << (dev.is_gpu() ? "Yes" : "No") << "\n";
    std::cout << "    is_accelerator(): "
              << (dev.is_accelerator() ? "Yes" : "No") << "\n";

    do_query<info::device::vendor>(dev, "info::device::vendor");
    do_query<info::device::driver_version>(dev,
               "info::device::driver_version");
    do_query<info::device::max_work_item_dimensions>(dev,
               "info::device::max_work_item_dimensions");
    do_query<info::device::max_work_group_size>(dev,
               "info::device::max_work_group_size");
    do_query<info::device::mem_base_addr_align>(dev,
               "info::device::mem_base_addr_align");
    do_query<info::device::partition_max_sub_devices>(dev,
               "info::device::partition_max_sub_devices");

    std::cout << "    Many more queries are available than shown here!\n";
  }
  std::cout << "\n";
}
```

Figure 12-6. *More detailed use of device query mechanisms: verycurious.cpp*

Inquisitive: get_info<>

The has_extension() interface allows a program to test directly for a feature, rather than having to walk through a list of extensions from get_info <info::platform::extensions> as printed out by the previous code examples. The SYCL 2020 provisional specification has defined new mechanisms to query extensions and detailed aspects of devices, but we don't cover those features (which are just being finalized) in this book. Consult the online oneAPI DPC++ language reference for more information.

Device Information Descriptors

Our "curious" program examples, used earlier in this chapter, utilize the most used SYCL device class member functions (i.e., is_host, is_cpu, is_gpu, is_accelerator, get_info, has_extension). These member functions are documented in the SYCL specification in a table titled "Member functions of the SYCL device class" (in SYCL 1.2.1, it is Table 4.18).

The "curious" program examples also queried for information using the get_info member function. There is a set of queries that must be supported by all SYCL devices, including a host device. The complete list of such items is described in the SYCL specification in a table titled "Device information descriptors" (in SYCL 1.2.1, it is Table 4.20).

Device-Specific Kernel Information Descriptors

Like platforms and devices, we can query information about our kernels using a get_info function. Such information (e.g., supported work-group sizes, preferred work-group size, the amount of private memory required per work-item) is device-specific, and so the get_info member function of the kernel class accepts a device as an argument.

DEVICE-SPECIFIC KERNEL INFORMATION IN SYCL 1.2.1

For historical reasons dating back to OpenCL naming, SYCL inherited a combination of queries named `kernel::get_info` and `kernel::get_work_group_info`, returning information about a kernel object and information pertaining to a kernel's execution on a specific device, respectively.

Use of overloading in DPC++ and SYCL (as of 2020 provisional) allows for both types of information to be supported through a single `get_info` API.

The Specifics: Those of "Correctness"

We will divide the specifics into information about necessary conditions (correctness) and information useful for tuning but not necessary for correctness.

In this first correctness category, we will enumerate conditions that should be met in order for kernels to launch properly. Failure to abide by these device limitations will lead to program failures. Figure 12-7 shows how we can fetch a few of these parameters in a way that the values are available for use in host code and in kernel code (via lambda capture). We can modify our code to utilize this information; for instance, it could guide our code on buffer sizing or work-group sizing.

Submitting a kernel that does not satisfy these conditions will generate an error.

```
std::cout << "We are running on:\n"
          << dev.get_info<info::device::name>() << "\n";

// Query results like the following can be used to calculate how
// large our kernel invocations should be.
auto maxWG = dev.get_info<info::device::max_work_group_size>();
auto maxGmem = dev.get_info<info::device::global_mem_size>();
auto maxLmem = dev.get_info<info::device::local_mem_size>();

std::cout << "Max WG size is " << maxWG
          << "\nMax Global memory size is " << maxGmem
          << "\nMax Local memory size is "  << maxLmem << "\n";
```

Figure 12-7. *Fetching parameters that can be used to shape a kernel*

Device Queries

`device_type`: `cpu`, `gpu`, `accelerator`, `custom`,[2] `automatic`, `host`, `all`. These are most often tested by `is_host()`, `is_cpu`, `is_gpu()`, and so on (see Figure 12-6):

> `max_work_item_sizes`: The maximum number of work-items that are permitted in each dimension of the work-group of the `nd_range`. The minimum value is `(1, 1, 1)` for non-custom devices.
>
> `max_work_group_size`: The maximum number of work-items that are permitted in a work-group executing a kernel on a single compute unit. The minimum value is 1.
>
> `global_mem_size`: The size of global memory in bytes.
>
> `local_mem_size`: The size of local memory in bytes. Except for custom devices, the minimum size is 32 K.

[2]Custom devices are not discussed in this book. If we find ourselves programming a device that identifies itself using the *custom* type, we will need to study the documentation for that device to learn more.

`extensions`: Device-specific information not specifically detailed in the SYCL specification, often vendor-specific, as illustrated in our `verycurious` program (Figure 12-6).

`max_compute_units`: Indicative of the amount of parallelism available on a device—implementation-defined, interpret with care!

`sub_group_sizes`: Returns the set of sub-group sizes supported by the device.

`usm_device_allocations`: Returns `true` if this device supports device allocations as described in explicit USM.

`usm_host_allocations`: Returns `true` if this device can access host allocations.

`usm_shared_allocations`: Returns `true` if this device supports shared allocations.

`usm_restricted_shared_allocations`: Returns `true` if this device supports shared allocations as governed by the restrictions of "restricted USM" on the device. This property requires that property `usm_shared_allocations` returns `true` for this device.

`usm_system_allocator`: Returns `true` if the system allocator may be used instead of USM allocation mechanisms for shared allocations on this device.

We advise avoiding max_compute_units in program logic.

We have found that querying the maximum number of compute units should be avoided, in part because the definition isn't crisp enough to be useful in code tuning. Instead of using `max_compute_units`, most programs should express their parallelism and let the runtime map it onto available parallelism. Relying on `max_compute_units` for correctness only makes sense when augmented with implementation- and device-specific information. Experts might do that, but most developers do not and do not need to do so! Let the runtime do its job in this case!

Kernel Queries

The mechanisms discussed in Chapter 10, under "Kernels in Program Objects," are needed to perform these kernel queries:

`work_group_size`: Returns the maximum work-group size that can be used to execute a kernel on a specific device

`compile_work_group_size`: Returns the work-group size specified by a kernel if applicable; otherwise returns (0, 0, 0)

`compile_sub_group_size`: Returns the sub-group size specified by a kernel if applicable; otherwise returns 0

`compile_num_sub_groups`: Returns the number of sub-groups specified by a kernel if applicable; otherwise returns 0

`max_sub_group_size`: Returns the maximum sub-group size for a kernel launched with the specified work-group size

`max_num_sub_groups`: Returns the maximum number of sub-groups for a kernel

The Specifics: Those of "Tuning/Optimization"

There are a few additional parameters that can be considered as fine-tuning parameters for our kernels. These can be ignored without jeopardizing the correctness of a program. These allow our kernels to really utilize the particulars of the hardware for performance.

Paying attention to the results of these queries can help when tuning for a cache (if it exists).

Device Queries

`global_mem_cache_line_size`: Size of global memory cache line in bytes.

`global_mem_cache_size`: Size of global memory cache in bytes.

`local_mem_type`: The type of local memory supported. This can be `info::local_mem_type::local` implying dedicated local memory storage such as SRAM or `info::local_mem_type::global`. The latter type means that local memory is just implemented as an abstraction on top of global memory with no performance gains. For custom devices (only), the local memory type can also be `info::local_mem_type::none`, indicating local memory is not supported.

Kernel Queries

`preferred_work_group_size`: The preferred work-group size for executing a kernel on a specific device.

`preferred_work_group_size_multiple`: The preferred work-group size for executing a kernel on a specific device

Runtime vs. Compile-Time Properties

The queries described in this chapter are performed through runtime APIs (`get_info`), meaning that the results are not known until runtime. This covers many use cases, but the SYCL specification is also undergoing work to provide compile-time querying of properties, when they can be known by a toolchain, to allow more advanced programming techniques such as templating of kernels based on properties of devices. Compile-time adaptation of code based on queries is not possible with the existing runtime queries, and this ability can be important for advanced optimizations or when writing kernels that use some extensions. The interfaces were not defined well enough yet at the time of writing to describe those interfaces in this book, but we can look forward to much more powerful query and code adaptation mechanisms that are coming soon in SYCL and DPC++! Look to the online oneAPI DPC++ language reference and the SYCL specifications for updates.

Summary

The most portable programs will query the devices that are available in a system and adjust their behavior based on runtime information. This chapter opens the door to the rich set of information that is available to allow such tailoring of our code to adjust to the hardware that is present at runtime.

Our programs can be made more portable, more performance portable, and more future-proof by parameterizing our application to adjust to the characteristics of the hardware. We can also test that the hardware present falls within the bounds of any assumptions we have made in the design of our program and either warn or abort when hardware is found that lies outside the bounds of our assumptions.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.