Heterogeneous Computing with SYCL

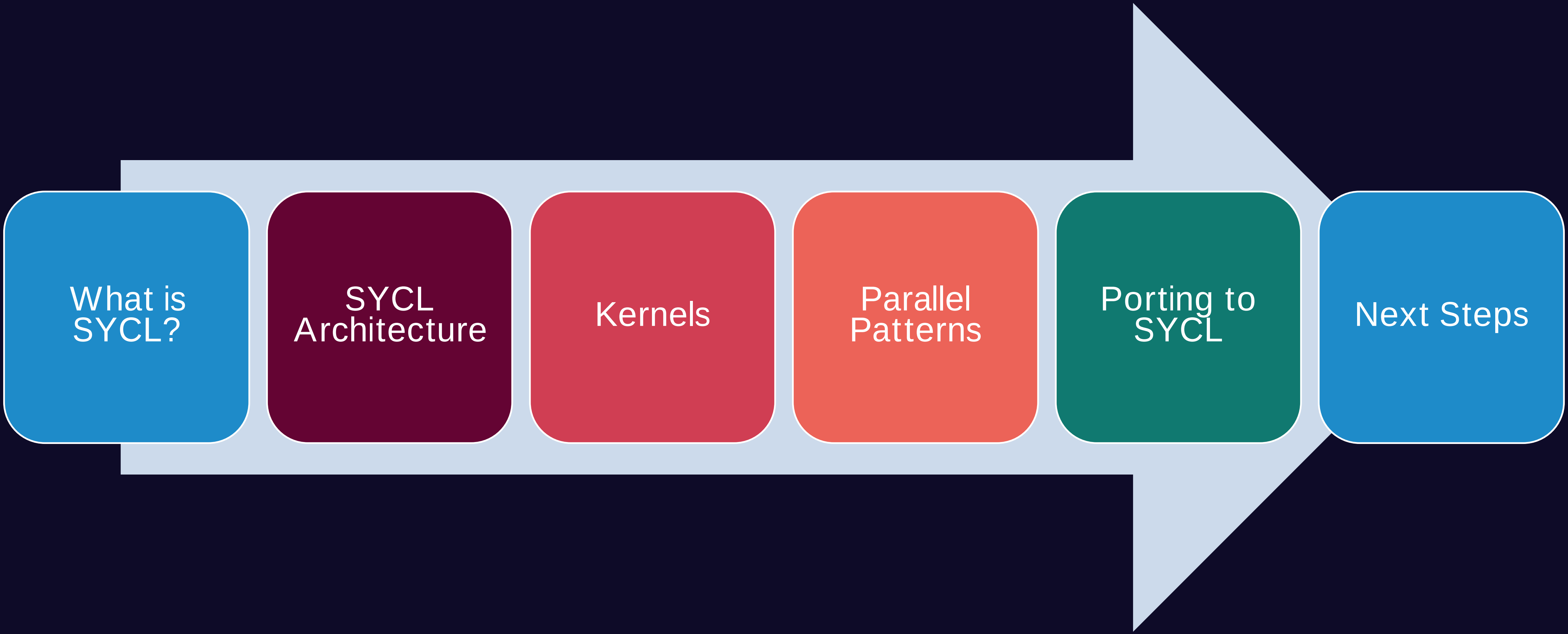
Kris Rowe
Argonne Leadership Computing Facility

Programming GPUs | 2022 Compute Ontario Summer School | July 8, 2022

Course Material on GitHub

- https://github.com/kris-rowe/coss-2022-sycl-tutorial
- Code is not production grade, is meant for learning
- Examples are flat-coded for readability
- Exercises are somewhat refactored to hide boilerplate code
- Ask for help in GitHub discussions incase someone else has a similar question

The Big Picture



Approach

USM instead of buffers/accessors

- Most HPC applications will use MPI
- Memory management is like CUDA
- Generally, provides better performance
- Explicit expression of data-dependencies
- Easily use libraries

Lambdas instead of functors

- Source code is easier to understand
- Simpler to port existing code
- Kernels are defined where they are used

What is SYCL?

What is SYCL?

- An open, royalty-free, cross-platform standard for parallel programming on heterogeneous systems
- Developed by the Khronos Group
- A single-source programming model for modern standard C++ (C++17)
- Builds on concepts, portability, and efficiency of OpenCL
- *However*, SYCL >> OpenCL + C++17
  - Uses concepts that make modern C++ great
  - Higher-level interface is much more approachable than the OpenCL API

- Used by industry, government, and academia internationally
- Applications on systems ranging from laptops to world's largest supercomputers
- Runs on different types of hardware
  - CPUs
  - GPUs
  - FPGAs
- Runs on different vendor's hardware
  - NVIDIA
  - Intel
  - AMD

Example SYCL Program

```cpp
#include <CL/sycl.hpp>
#include <iostream>
#include <vector>

int main() {
  const size_t vector_length = 2000;
  std::vector<float> a_host(vector_length);
  std::vector<float> b_host(vector_length, 1.0);
  std::vector<float> c_host(vector_length, 1.0);
  const float s = 1.0;

  // Create a sycl::queue using the default device selector
  sycl::device sycl_device{sycl::default_selector()};
  sycl::context sycl_context{sycl_device};
  sycl::queue sycl_queue{sycl_context, sycl_device};

  // Allocate vectors on the device
  float* a =
      sycl::malloc_device<float>(vector_length, sycl_device, sycl_context);
  float* b =
      sycl::malloc_device<float>(vector_length, sycl_device, sycl_context);
  float* c =
      sycl::malloc_device<float>(vector_length, sycl_device, sycl_context);

  // Copy from the host to the device; synchronize.
  sycl_queue.copy(b_host.data(), b, b_host.size());
  sycl_queue.copy(c_host.data(), c, c_host.size());
  sycl_queue.wait();

  // Submit work to the queue using a kernel defined via lambdas; synchronize.
  sycl_queue.parallel_for({vector_length},
                          [=](sycl::id<1> i) { a[i] = b[i] + s * c[i]; });
  sycl_queue.wait();

  // Copy from the device to the host; synchronize.
  sycl_queue.copy(a, a_host.data(), a_host.size());
  sycl_queue.wait();

  // Verify the results.
  for (const auto& a_i : a_host) {
    // Don't check for equality of floating-point values in production code!
    if (2.0 != a_i) {
      std::cout << "Verification failed!\n";
      return EXIT_FAILURE;
    }
  }

  std::cout << "Success!\n";

  // Free device memory
  sycl::free(a, sycl_context);
  sycl::free(b, sycl_context);
  sycl::free(c, sycl_context);
  return EXIT_SUCCESS;
}
```

Compiling, Linking, & Running Code

- Since SYCL programs use single-source C++ code building a program is straightforward
- To use LLVM clang and friends, pass the `-fsycl` flag
- Device functions should be in the same translation unit as the kernels which use them
- Like CUDA compilation, a fat-binary is produced
  - Contains binaries and/or low-level (SPIR-V, ptx) code needed to run on one or more target platforms
  - Enables cross-compilation, good for HPC systems
  - Can prescribe specific targets using triples like `-fsycl-targets=nvptx64-nvidia-cuda`
  - Ahead-of-time compilation is available for some backends via linker flags (e.g., `-Xsycl`)
- Consult compiler documentation for a full set of options
  - https://intel.github.io/llvm-docs/UsersManual.html

SYCL Architecture

Backends

- A SYCL Backend implements the SYCL programming model
- Typically, an implementation uses low-level, highly-optimized, vendor-specific drivers
- Intel oneAPI DPC++ Compiler comes with several backends
  - OpenCL – for Intel CPUs and GPUs using the Intel OpenCL Driver
  - Level Zero – for Intel GPUs using the Intel oneAPI Level Zero Driver
  - CUDA – for NVIDIA GPUs using the NVIDIA CUDA Driver
  - HIP – for AMD GPUs using the AMD ROCm Driver
- Direct interaction with SYCL backend should not be necessary for most applications
- Interoperability functions provide an interface to backend objects
  - Use case: calling a vendor-specific library in a SYCL application

Platforms & Devices

- A SYCL device represents hardware which can execute SYCL kernels
- Three main types
  - CPU
  - GPU
  - Accelerator (FPGA)
- A SYCL platform is a collection of devices managed by a single backend
- The same hardware can appear as different devices in different platforms
- E.g., a single Intel GPU could appear in both the Level-Zero and OpenCL platforms

```
Platform 0: Intel(R) OpenCL
Device 0: Intel(R) Xeon(R) CPU E3-1585 v5 @ 3.50GHz
Type: CPU
Memory: 62 GB
Max Work Group Size: 8192

Platform 1: Intel(R) Level-Zero
Device 0: Intel(R) Iris(R) Pro Graphics P580 [0x193a]
Type: GPU
Memory: 49 GB
Max Work Group Size: 256

Platform 2: SYCL host platform
Device 0: SYCL host device
Type: host
Memory: 62 GB
Max Work Group Size: 1
```

A single CPU system with integrated graphics

Information Descriptors

SYCL runtime classes provide the function `get_info`

The return value depends on the template parameter—called an information descriptor

```
for (auto& p : platforms) {
  std::string platform_name = p.get_info<sycl::info::platform::name>();
  std::cout << "Platform " << platform_id << ": " << platform_name << "\n";

  auto devices = p.get_devices();
  int device_id = 0;
  for (auto& d : devices) {
    std::string device_name = d.get_info<sycl::info::device::name>();
    std::cout << "Device " << device_id << ": " << device_name << "\n";

    printDeviceType(d);

    uint64_t memory = d.get_info<sycl::info::device::global_mem_size>();
    std::cout << "Memory: " << (memory / gigabyte) << " GB\n";

    uint64_t max_wg_size =
        d.get_info<sycl::info::device::max_work_group_size>();
    std::cout << "Max Work Group Size: " << max_wg_size << "\n";
```

Contexts

- A SYCL context is
  - A collection of one or more SYCL devices
  - Associated memory allocations
  - Any work-queues on each device
- This defines the fundamental “world” in which SYCL programs operate
- Convenient shortcuts avoid the need to deal with platforms and contexts explicitly

Devices Selectors

- A device selector is a function which returns an integral score for a given SYCL device
- When passed to a constructor, the SYCL runtime scores all available SYCL devices and uses the highest-scoring device
- Several built-in device selectors are defined by the SYCL standard
  - `default_selector`
  - `cpu_selector`
  - `gpu_selector`

```
// Create a sycl::queue using the default device selector
sycl::device sycl_device{sycl::default_selector()};
sycl::context sycl_context{sycl_device};
sycl::queue sycl_queue{sycl_context, sycl_device};
```

Exercise #2 involves implementing a custom device selector!

Memory Management

- The SYCL Unified Shared Memory model uses a single virtual address space to ensure pointers in host and device code have consistent values.
- C-style malloc functions provide three flavors of memory allocation
  - Device allocations
    - Exist in on-device memory
    - Accessible on the device, but not the host
    - Data must be explicitly transferred between the host and device
  - Host allocations
    - Page-locked memory on the host
    - Accessible on both the host and device
    - Device can write to host memory directly in kernels
  - Shared allocations
    - Accessible on both the host and device
    - Automatically migrated by the runtime
- USM allocations must be explicitly freed by calling `sycl::free`

Queues

- SYCL queues are work-queues on a specific device to which work can be submitted
- By default, SYCL queues are out-of-order
- No need to manage multiple in-order queues
  - Runtime handles the hard-work for you!
- Need to ensure data transfer operations are finished before they are needed by a kernel
- If necessary, can force queue to be in-order
  - Useful for porting CUDA code
- Most common functions that enqueue work
  - `parallel_for`
  - `memcpy`

Host Synchronization & Events

- Constructor and malloc calls are synchronous
- Functions that submit work to a device are asynchronous, but return SYCL event objects
  - E.g., `memcpy` and `parrallel_for`
- To synchronize with host
  - Call `queue.wait()` or `wait_and_throw()`
    - This waits for all enqueued work to finish—could block other queues with dependencies
  - Wait on a specific event (or set of events)
- Events can also be passed to work enqueuing functions as dependencies
  - Enables the programmer to directly express data dependencies

Exceptions

SYCL defines an exception class that extends C++ standard exceptions

Exception handling is done with the usual `try-catch` pattern

```cpp
#include <CL/sycl.hpp>
#include <iostream>

int main() {
  try {
    // Create a host device
    sycl::device sycl_device{sycl::default_selector()};
    sycl::context sycl_context{sycl_device};
    sycl::queue sycl_queue{sycl_context, sycl_device};

    int i{};
    // Mock device_malloc failing and returning nullptr
    int* j = nullptr;
    sycl_queue.copy<int>(&i, j, 1);

  } catch (const sycl::exception& e) {
    std::cerr << "Synchronous exception\n";
    std::cerr << e.what() << std::endl;
  }
  std::cout << "Exception handled.\n";
  return EXIT_SUCCESS;
}
```

Kernels

Data Parallelism

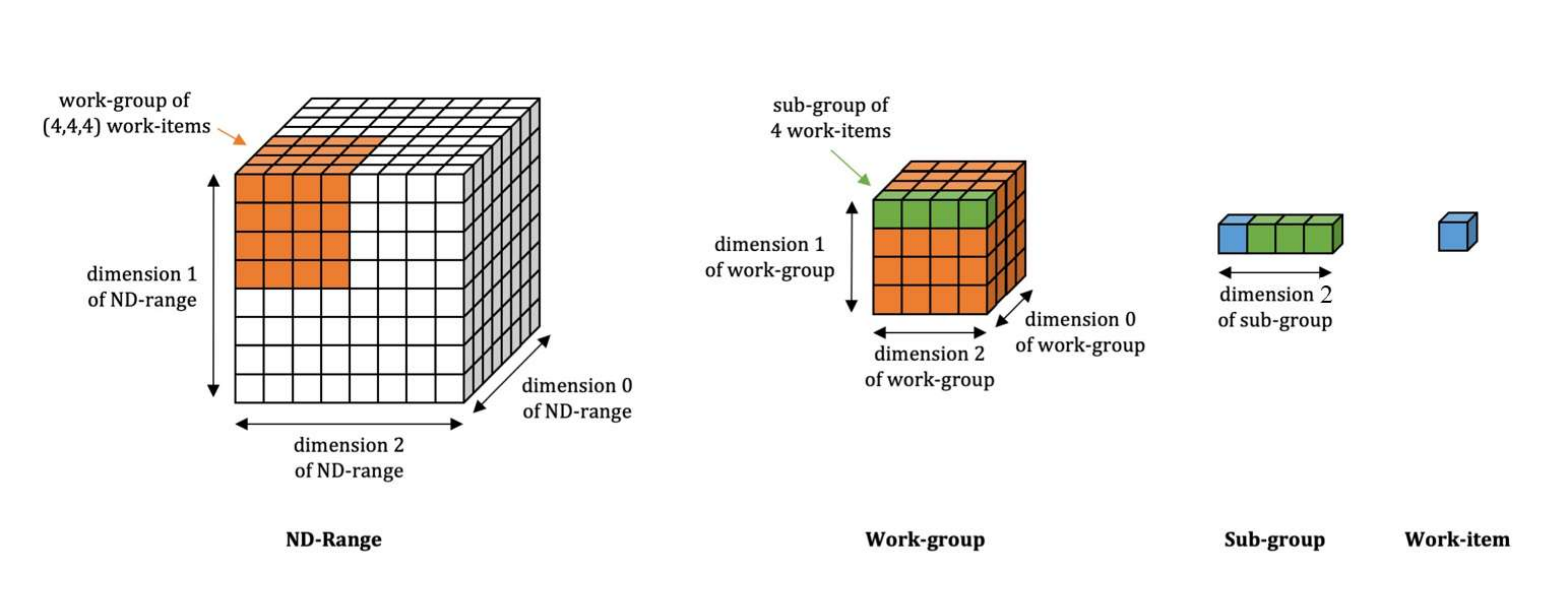
A kernel is a function defined for a ranges of values in an index space

Instances of the function body are executed in parallel for each point in this space

While each instance, or work-item, executes the same code, the data operated on can vary

The mapping of work-items to hardware processing elements is implementation specific

Thread Hierarchy



Defining Kernels Using Lambdas

- Kernels can be defined using C++ lambdas
- Basic kernels take `sycl::id` or `sycl::item` argument
- ND-range kernels take a `sycl::nd_item` argument
- Key difference: ND-range kernels can use shared local memory and synchronize work-items within the same work-group

```
[=](sycl::id<1> i) {
  a[i] = b[i] + s * c[i];
}
```

Basic Kernel

```
[=](sycl::nd_item<2> work_item) {
  //...
}
```

ND-Range Kernel

Launching Kernels & Data Dependencies

```
sycl_queue.parallel_for({vector_length},
  [=](sycl::id<1> i) {
    a[i] = b[i] + s * c[i];
});
```

Basic Kernel

```
sycl::range<2> local_range(block_size, block_size);
sycl::range<2> global_range(N, M);
sycl::nd_range<2> kernel_range(global_range, local_range);

sycl::event gemm_kernel = sycl_queue.parallel_for(
  kernel_range, {copy_a, copy_b},
  [=](sycl::nd_item<2> work_item) {
  //...
});
```

ND-range Kernel

Device Functions

- Since SYLC uses single-source C++ code, functions may be compiled for the host, the device, or both
- Extra restrictions are placed on device functions compared to regular C++ code
  - Data types must be trivially copyable
  - No memory allocation
  - No virtual functions
  - No RTTI
  - No variadic templates
  - No exception handling
- See the SYCL spec for a complete list: https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:language.restrictions.kernels

```
void f(handler& cgh) {
  // Function "f" is not compiled for device

  cgh.single_task([=] {
    // This code is compiled for device
    g(); // This line forces "g" to be compiled for device
  });
}

void g() {
  // Called from kernel, so "g" is compiled for device
}

void h() {
  // Not called from a device function, so not compiled for device
}
```

Parallel Patterns (Examples Demo)

Porting to SYCL

Porting an Existing C++ Application

- Modular, testable code is easier to port
  - Worth spending time improving test coverage, refactoring before porting
- Profile performance under realistic workloads relevant to your target science problem
- Identify parts of application that are most likely to benefit from using GPUs
  - E.g., Use Intel Advisor Offload Modelling https://www.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/model-offloading-to-a-gpu.html
- Extract proxy/mini-apps which exercise a critical part of your code in isolation
- Port mini-apps, then incorporate ported code back into main application

Porting an Existing CUDA Application

- Using USM approach manual code porting should be straightforward
- Start with in-order queues and ensure program correctness before relaxing to out-of-order
- Port code in stages using interoperability interfaces
- Automated porting tools exist—like Intel's DPC++ Compatibility Tool https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html
- Still requires intervention by the developer

Next Steps

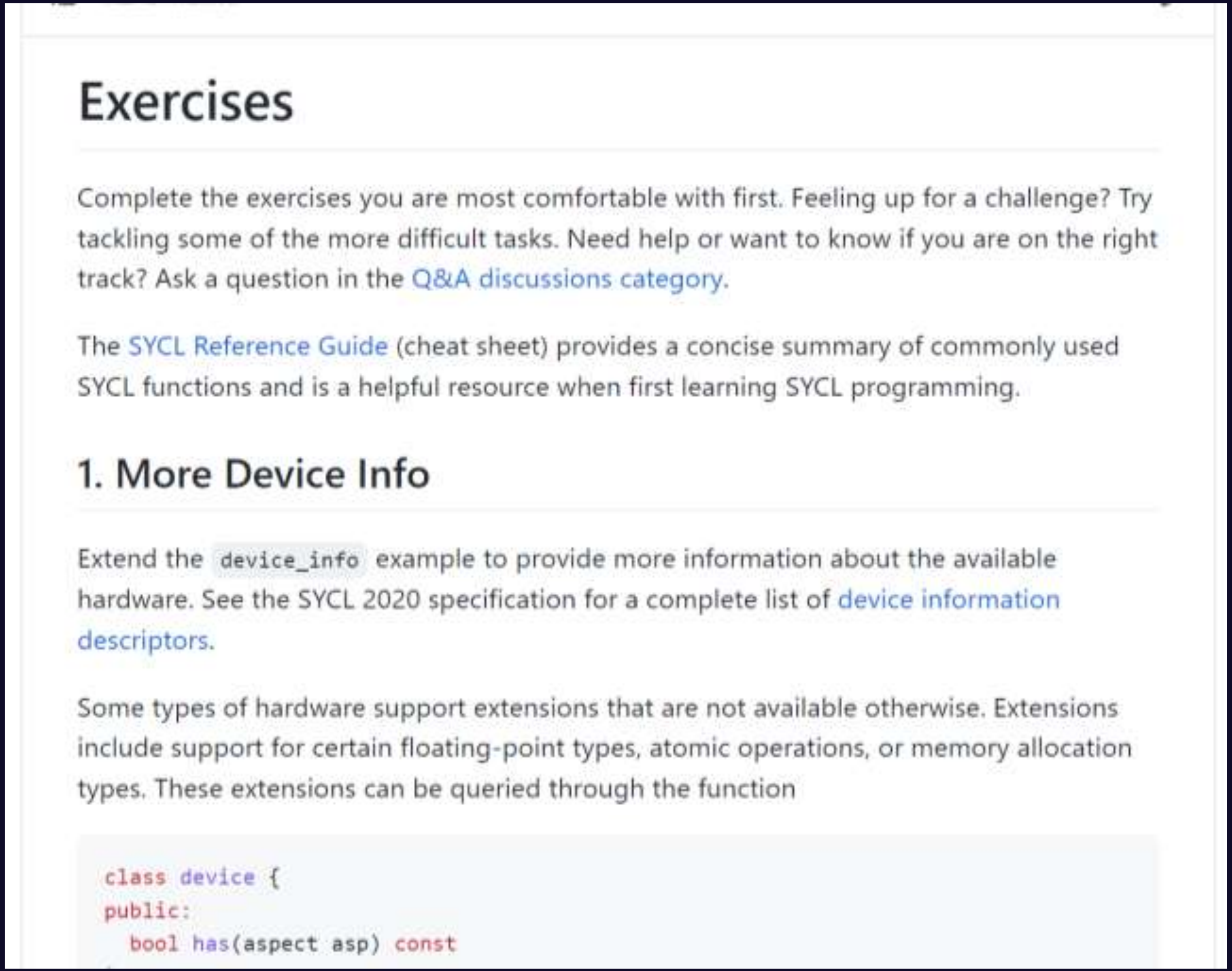
Exercises

Several exercises are outlined on GitHub

Stubs or significant source code is provided for most of the exercises

Focus is on using SYCL concepts

https://github.com/kris-rowe/coss-2022-sycl-tutorial/tree/main/exercises#exercises



Exercises

Complete the exercises you are most comfortable with first. Feeling up for a challenge? Try tackling some of the more difficult tasks. Need help or want to know if you are on the right track? Ask a question in the Q&A discussions category.

The SYCL Reference Guide (cheat sheet) provides a concise summary of commonly used SYCL functions and is a helpful resource when first learning SYCL programming.

1. More Device Info

Extend the `device_info` example to provide more information about the available hardware. See the SYCL 2020 specification for a complete list of device information descriptors.

Some types of hardware support extensions that are not available otherwise. Extensions include support for certain floating-point types, atomic operations, or memory allocation types. These extensions can be queried through the function

```
class device {
public:
  bool has(aspect asp) const
```

Portable Libraries

oneMKL Interfaces

https://github.com/oneapi-src/oneMKL

Checkout other libraries defined as part of the oneAPI spec

https://www.oneapi.io/spec/

oneAPI Math Kernel Library (oneMKL) Interfaces

oneAPI

oneMKL Interfaces is an open-source implementation of the oneMKL Data Parallel C++ (DPC++) interface according to the oneMKL specification. It works with multiple devices (backends) using device-specific libraries underneath.

oneMKL is part of oneAPI.

User Application	oneMKL Layer	Third-Party Library	Hardware Backend
oneMKL interface	oneMKL selector	Intel(R) oneAPI Math Kernel Library for x86 CPU	x86 CPU
		Intel(R) oneAPI Math Kernel Library for Intel GPU	Intel GPU
		NVIDIA cuBLAS for NVIDIA GPU	NVIDIA GPU
		NVIDIA cuSOLVER for NVIDIA GPU	NVIDIA GPU
		NVIDIA cuRAND for NVIDIA GPU	NVIDIA GPU
		NETLIB LAPACK for x86 CPU	x86 CPU
		AMD rocBLAS for AMD GPU	AMD GPU
		AMD rocRAND for AMD GPU	AMD GPU

Resources

Intel

- DPC++ Essentials Training
  - https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/dpc-essentials.html
- GPU Optimization Guide
  - https://www.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top.html
- oneAPI samples
  - https://github.com/oneapi-src/oneAPI-samples
- Developer Conferences
  - https://www.oneapi.io/events/
- DevMesh
  - https://devmesh.intel.com/

ALCF

- Events
  - https://www.alcf.anl.gov/events
- Training
  - https://www.alcf.anl.gov/support-center/training-assets
- ATPESC
  - https://extremecomputingtraining.anl.gov/

References

- SYCL 2020 Specification
  - https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html
- SYCL Reference Guide (cheat sheet)
  - https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf
- oneAPI DPC++ (Intel LLVM) Compiler Documentation
  - https://intel.github.io/llvm-docs/

This work was supported by Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.