Agenda

Introduction: Heterogeneous Computing
Why SYCL?
SYCL as Portable Programming Model

Execution Model
Memory Model
Best Practices

A few practical case studies
Heterogeneous Computing

What does a machine look like in a heterogeneous world?

- CPU
- GPU
Heterogeneous Computing

What does a machine look like in a heterogeneous world?
Heterogeneous Computing

What does a machine look like in a heterogeneous world?
Heterogeneous Computing

What does a machine look like in a heterogeneous world?

Diversity in devices (capabilities)
Diversity in memory connectivity/coherence
Diversity in how they all connect
Desire to use them all concurrently in parallel
## Pre-Exascale & Exascale Compute Architectures

<table>
<thead>
<tr>
<th>Machine</th>
<th>GPU Models</th>
<th>DOE Facility</th>
<th>GPU</th>
<th>No of GPUs per node</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA, SYCL</td>
<td>ALCF Polaris</td>
<td>Nvidia A100</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CUDA, SYCL</td>
<td>NERSC Perlmutter</td>
<td>Nvidia A100</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>HIP, SYCL</td>
<td>OLCF Frontier</td>
<td>AMD MI 250x</td>
<td>4/8</td>
<td></td>
</tr>
<tr>
<td>SYCL</td>
<td>ALCF Aurora</td>
<td>Intel Max Series</td>
<td>6/12</td>
<td></td>
</tr>
<tr>
<td>HIP, (SYCL)</td>
<td>LLNL El Capitan</td>
<td>AMD MI 300</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What is SYCL?

- SYCL is “not” a programming model but a “language specification”
- Heuristics looks similar to OpenCL-C bindings
- C++ single source (coexists host and device source code)
- Two distinct memory models (USM and/or Buffer)
- Asynchronous programming (overlaps device-compute, copy, host operations)
- Portability (functional and performance)
- Productivity
SYCL – Motivation

oneAPI Implementation of SYCL = C++ and SYCL* standard and extensions

Based on modern C++
✓ C++ productivity features and familiar constructs

Standards-based, cross-architecture
✓ Incorporates the SYCL standard for data parallelism and heterogeneous programming
**SYCL* extensions**

**Productivity**
- Simple things should be simple to express
- Reduce verbosity and programmer burden enhance performance

• Give programmers control over program execution
• Enable hardware-specific features

Fast-moving open collaboration feeding into the SYCL* standard
✓ Open source implementation with goal of upstream LLVM
✓ Extensions aim to become core SYCL*, or Khronos* extensions
SYCL – A Portable Programming Model

A C++-based programming model for intra-node parallelism
- SYCL is a specification and “not” an implementation, currently compliant to C++17 ISO standards
- Cross-platform abstraction layer, heavily backed by industry
- Open-source, vendor agonistic
- Single-source model
SYCL – Compiler Players

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies.

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute.

DPC++
Uses LLVM/Clang
Part of oneAPI

Any CPU
NVIDIA GPUs
AMD GPUs
Intel CPUs
Intel FPGAs

DPC++
Uses LLVM/Clang
Part of oneAPI

Any CPU
NVIDIA GPUs
AMD GPUs
Intel CPUs
Intel FPGAs

computeplay
ComputeCpp
Multiple Backends

Any CPU
NVIDIA GPUs
Intel CPUs
Intel GPUs
Intel FPGAs
AMD GPUs
(depend on driver stack)
Arm Mali
IMG PowerVR
Renesas R-Car

hipSYCL
Multiple Backends

Any CPU
NVIDIA GPUs
AMD GPUs
Intel GPUs

Level Zero

Intel CPUs
Intel GPUs
Intel FPGAs

Level Zero

Intel CPUs
Intel GPUs
Intel FPGAs
NVIDIA GPUs
AMD GPUs
OpenCL
SPIR

Source Code
Queues & Contexts

• “SYCL Queues” provide mechanism to submit work to a device or sub-device
• “SYCL Contexts” is well known to be over-looked

```cpp
sycl::queue Que; // implicitly creates a SYCL context
```

**Context**

• Contexts are used for resources isolation and sharing
• A SYCL context may consist of one or multiple devices
• Both root-devices and sub-devices can be within single context (all from same SYCL platform)
• Memory created can be shared only if their associated queue(s) are created using the same context

**Queue (aka CUDA Stream)**

• SYCL queue is always attached to a single device in a possibly multi-device context
  ✓ Executes “asynchronously” from host code
  ✓ SYCL queue can execute tasks enqueued in either “in-order” or “out-of-order (default)”
  ✓ SYCL queue (in-order) is similar to CUDA stream (FIFO)
**Queues (out-of-order vs in-order)**

(OOO queue) This means commands are allowed to be overlapped and re-ordered or executed concurrently providing dependencies are honoured to ensure consistency.

(In-order) This means commands must execute strictly in the order they were enqueued.

```cpp
auto outOfOrderQueue = sycl::queue{gpu_selector_v};
auto inOrderQueue = sycl::queue{gpu_selector_v, sycl::property::queue::in_order{}};
```
Devices

- Devices are the target for acceleration offload
- **SYCL sub-devices ↔ CUDA Multi-Instance GPU (MIG) mode ↔ OpenCL sub-devices**

- Explicit Scaling: Partitioning of a SYCL root device into multiple sub-devices based on NUMA boundary
  - SYCL queues are further created based on “sub-devices” (better performance)
- Implicit Scaling: SYCL unpartitioned/root device is directly used to create a SYCL queue

```cpp
sycl::queue Que;

// EXPLICIT SCALING (better performance)

sycl::platform platform(sycl::gpu_selector{});
auto const& gpu_devices = platform.get_devices(sycl::info::device_type::gpu);
for (auto const& gpuDev : gpu_devices) {
    if(gpuDev.get_info<sycl::info::device::partition_max_sub_devices>() > 0) {
        auto SubDev = gpuDev.create_sub_devices<sycl::info::partition_property::partition_by_affinity_domain>(sycl::info::partition_affinity_domain::numa);
        for (auto const& tile : SubDev) {
            Que = sycl::queue(tile);
        }
    }
}

// IMPLICIT SCALING

sycl::platform platform(sycl::gpu_selector{});
auto const& gpu_devices = platform.get_devices(sycl::info::device_type::gpu);
for (auto const& gpuDev : gpu_devices) {
    Que = sycl::queue(gpuDev);
}
```
Devices

• Devices are the target for acceleration offload
SYCL sub-devices ↔ CUDA Multi-Instance GPU (MIG) mode ↔ OpenCL sub-devices

• Explicit Scaling: Partitioning of a SYCL root device into multiple sub-devices based on NUMA boundary
  ✓ SYCL queues are further created based on “sub-devices” (better performance)

• Implicit Scaling: SYCL unpartitioned/root device is directly used to create a SYCL queue

```c++
sycl::queue Que;

// EXPLICIT SCALING (better performance)
sycl::platform platform(sycl::gpu_selector{});
auto const& gpu_devices = platform.get_devices(sycl::info::device_type::gpu);
for (auto const& gpuDev : gpu_devices) {
    if (gpuDev.get_info<sycl::info::device::partition_max_sub_devices>() > 0) {
        auto SubDev = gpuDev.create_sub_devices<sycl::info::partition_property::partition_by_affinity_domain>(sycl::info::partition_affinity_domain::numa);
        for (auto const& tile : SubDev) {
            Que = sycl::queue(tile);
        }
    }
}

// IMPLICIT SCALING
sycl::platform platform(sycl::gpu_selector{});
auto const& gpu_devices = platform.get_devices(sycl::info::device_type::gpu);
for (auto const& gpuDev : gpu_devices) {
    Que = sycl::queue(gpuDev);
}
```
SYCL Events for Task-dependencies

- Performs book-keeping of tasks for data-transfer between host-device
- Similar to CUDA/HIP events

- SYCL runtime inherently has a DAG dependency paradigms
- Using SYCL dependency infrastructure provided via “SYCL Event”

Task dependency between a communication-computation tasks

```c
uintptr_t memcpy(void* dest, const void* src, size_t numBytes);
uintptr_t memcpy(void* dest, const void* src, size_t numBytes, event depEvent);
uintptr_t memcpy(void* dest, const void* src, size_t numBytes,
         const std::vector<event>& depEvents);
```

Data-transfer showing “event” returns & dependencies

```c
template <typename KernelName, int Dims, typename... Rest>
uintptr_t parallel_for(nd_range<Dims> executionRange,
         const std::vector<event>& depEvents, Rest&&... rest);
```

Kernel-launch showing “event” returns & dependencies
Porting from CUDA to SYCL
A **grid** is an array of thread blocks launched by a kernel.

An **nd range** has three components:
- global range (total work items)
- local range (work-items per work-group)
- number of work groups (total work groups)
### CUDA - warp (vs) SYCL - sub groups

<table>
<thead>
<tr>
<th><strong>CUDA</strong></th>
<th><strong>SYCL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>work-item</td>
</tr>
<tr>
<td><strong>warp</strong></td>
<td><strong>sub-group</strong></td>
</tr>
<tr>
<td>block</td>
<td>work-group</td>
</tr>
<tr>
<td>grid</td>
<td>nd-range</td>
</tr>
</tbody>
</table>

Sub-groups are subset of the work-items that are executed simultaneously or with additional scheduling guarantees.

Leveraging sub-groups will help to map execution to low-level hardware and may help in achieving higher performance.
Why use SYCL - sub groups?

Sub-Group = subset of work-items within a work-group.

A subset of work-items within a work-group that execute with additional guarantees and often map to SIMD hardware.

- Work-items in a sub-group can communicate directly using shuffle operations, without repeated access to local or global memory, and may provide better performance.
- Work-items in a sub-group have access to sub-group collectives, providing fast implementations of common parallel patterns.
# Memory Model: CUDA vs SYCL

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>CUDA</th>
<th>SYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Register memory</td>
<td>Private memory</td>
</tr>
<tr>
<td>Scope</td>
<td>Thread</td>
<td>Work-item</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Block</td>
<td>Local memory</td>
</tr>
<tr>
<td></td>
<td>Work-group</td>
<td></td>
</tr>
<tr>
<td>Global memory</td>
<td>Grid (all threads)</td>
<td>Global memory</td>
</tr>
<tr>
<td></td>
<td>All work Items</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Allocation Type</th>
<th>Initial Location</th>
<th>Accessible By</th>
<th>Migratable To</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>device</td>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>device</td>
<td>device</td>
<td></td>
<td>device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>device</td>
<td>Another device</td>
<td>Optional (P2P)</td>
<td>Another device</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>host</td>
<td>host</td>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>device</td>
<td></td>
<td>device</td>
<td>No</td>
</tr>
<tr>
<td>shared</td>
<td>Unspecified</td>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>device</td>
<td></td>
<td>device</td>
<td>Yes</td>
</tr>
<tr>
<td>device</td>
<td>Another device</td>
<td>Optional</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Another device</td>
<td></td>
</tr>
</tbody>
</table>

Memory Model: Global Memory

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory Type</strong></td>
<td><strong>Scope</strong></td>
</tr>
<tr>
<td>Register memory</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Block</td>
</tr>
<tr>
<td>Global memory</td>
<td>Grid (all threads)</td>
</tr>
</tbody>
</table>

// allocating device memory

```c
float *A_dev;
cudaMalloc((void **)&A_dev, array_size * sizeof(float));
```

// allocating device memory

```c
sycl::queue q(sycl::gpu_selector{});
float *A_dev = sycl::malloc_device<float>(array_size, q);
```

- SYCL's Global/Device allocated memory is only **valid** on the device
- More importantly not accessible from host
Vector Addition: SYCL Buffer memory model

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

void main() {
    using namespace sycl;
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };
    }
    queue myQueue;
    myQueue.submit([&](handler &cgh) {
        auto accA = bufA.get_access<access::read>(cgh);
        auto accB = bufB.get_access<access::read>(cgh);
        auto accC = bufC.get_access<access::write>(cgh);
        cgh.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
            accC[i] = accA[i] + accB[i];
        }).wait();
    });
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Create SYCL buffers using host pointers.

Create a queue to submit work to a GPU.

Read/write accessors create dependencies if other kernels or host access buffers.

Vector addition device kernel.
#include <sycl/sycl.hpp>
#include <iostream>

void main() {
    float A[1024], B[1024], C[1024]; // initialize A, B, C with values on host
    sycl::queue myQueue;
    float* devA = sycl::malloc_device<float>(1024, myQueue);
    float* devB = sycl::malloc_device<float>(1024, myQueue);
    float* devC = sycl::malloc_device<float>(1024, myQueue);
    myQueue.memcpy(devA, A, 1024 * sizeof(float));
    myQueue.memcpy(devB, B, 1024 * sizeof(float));
    myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
        devC[i] = devA[i] + devB[i];
    });
    myQueue.memcpy(C, devC, 1024 * sizeof(float));
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```cpp
#include <sycl/sycl.hpp>
#include <iostream>

void main() {
    float A[1024], B[1024], C[1024];
    // initialize A, B, C with values on host
    sycl::queue myQueue;
    float* devA = sycl::malloc_device<float>(1024, myQueue);
    float* devB = sycl::malloc_device<float>(1024, myQueue);
    float* devC = sycl::malloc_device<float>(1024, myQueue);
    myQueue.memcpy(devA, A, 1024 * sizeof(float));
    myQueue.memcpy(devB, B, 1024 * sizeof(float));
    myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
        devC[i] = devA[i] + devB[i];
    });
    myQueue.memcpy(C, devC, 1024 * sizeof(float));
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

---

**Vector Addition: SYCL USM memory model**

- **Host Code**
  - MyQueue:queue myQueue;
  - float* devA = sycl::malloc_device<float>(1024, myQueue);
  - float* devB = sycl::malloc_device<float>(1024, myQueue);
  - float* devC = sycl::malloc_device<float>(1024, myQueue);
  - myQueue.memcpy(devA, A, 1024 * sizeof(float));
  - myQueue.memcpy(devB, B, 1024 * sizeof(float));
  - myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
      devC[i] = devA[i] + devB[i];
    });
  - myQueue.memcpy(C, devC, 1024 * sizeof(float));
  - for (int i = 0; i < 1024; i++)
    std::cout << "C[" << i << "] = " << C[i] << std::endl;

- **Device Code**
  - SYCL queue (by-default) is out-of-order. (i.e., the execution starts when possible. Duty of programmer to assure correct dependencies

- **Host Code**
  - myQueue.wait(), wait for H2D to complete before starting the kernel
  - myQueue.wait(), wait for the kernel to finish
  - myQueue.wait(), wait for D2H to complete before printing “C”
Vector Addition: SYCL USM memory model

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

void main() {
    float A[1024], B[1024], C[1024];
    // initialize A, B, C with values on host
    sycl::queue myQueue(sycl::property_list{sycl::property::queue::in_order{}});

    float* devA = sycl::malloc_device<float>(1024, myQueue);
    float* devB = sycl::malloc_device<float>(1024, myQueue);
    float* devC = sycl::malloc_device<float>(1024, myQueue);

    myQueue.memcpy(devA, A, 1024 * sizeof(float));
    myQueue.memcpy(devB, B, 1024 * sizeof(float));

    myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
        devC[i] = devA[i] + devB[i];
    });

    myQueue.memcpy(C, devC, 1024 * sizeof(float));

    for (int i = 0; i < 1024; i++)
    {
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
    }
}
```
Vectors

Objectives:
Learn about scalar and vector instructions
Learn about horizontal and vertical vectorization
Learn how to write explicit vector code

Data parallel devices such as GPUs, SIMD CPUs and other accelerators are vector processors.
• This means they can execute vector instructions.
• Vector instructions are single instructions which perform loads, stores, or operations such as add or multiply on multiple elements at once.

Vectorization is the process of converting scalar code into vectorized code.
• In a SPMD programming model like SYCL vectorization is important.
• Vectorization can be performed in two ways, and it depends on how you write your code and can impact the mapping to hardware.
Vectors

```cpp
template <typename dataT, int numElements>
class vec;
```

The `vec` class template is used to represent explicit vectors in SYCL.
- It has a type which represents the type of elements it stores and a number of elements.
- The valid number of elements are 1, 2, 3, 4, 8, 16.

```cpp
using float4 = sycl::vec<float, 4>;
using double4 = sycl::vec<double, 4>;
using int4 = sycl::vec<int, 4>;
```

- A number of aliases are provided for shorthand with the notation of the type followed by the size, such as `float4`, etc.

Note: These vector data-types ensures vectorization.
**Vectors**

A vec object can be constructed with any combination of scalar and vector values which add up to the correct number of elements.

- A vec object can also be constructed from a single scalar in which case it will initialize ever element to that value.

```cpp
auto f4 = sycl::float4{1.0f, 2.0f, 3.0f, 4.0f}; // {1.0f, 2.0f, 3.0f, 4.0f}
auto f2 = sycl::float4{2.0f, 3.0f}; // {2.0f, 3.0f}
auto f4 = sycl::float4{1.0f, f2, 4.0f}; // {1.0f, 2.0f, 3.0f, 4.0f}
auto f4 = sycl::float4{0.0f}; // {0.0f, 0.0f, 0.0f, 0.0f}
```

- The vec class provides a number of operators such as +, -, *, / and many more, which perform the operation element-wise.

```cpp
auto f4a = sycl::float4{1.0f, 2.0f, 3.0f, 4.0f}; // {1.0f, 2.0f, 3.0f, 4.0f}
auto f4b = sycl::float4{2.0f}; // {2.0f, 2.0f, 2.0f, 2.0f}
auto f4r = f4a * f4b; // {2.0f, 4.0f, 6.0f, 8.0f}
```
SYCL Queue “query” for GPU

• Query helps to know of the status of completion of tasks (computation, communication)
• Prevents explicit/excess synchronization
• Tasks are bound to SYCL queues (cuda/hip/L0 stream)
• Need for light-weight heuristics for task-status query (aka GPU queue query)

• cudaStreamQuery/hipStreamQuery
  • Queries an asynchronous stream for completion status.

• **Portability with SYCL Queues**
  ✓ Such a functionality didn’t yet exist
  ✓ “**sycl::queue::ext_oneapi_empty()**” extension is now ratified by the SYCL specifications

• This extension provides a portable light-weight query feature to check on the task-status on GPU
Portable Math Libraries?

- Open-source implementation of the oneMKL Data Parallel C++ (DPC++) interface
- Works with multiple devices (backends)
- Uses vendor, device-specific libraries underneath

Note: Apart of device-backend, supports host-CPU interface: Intel MKL, NETLIB

<table>
<thead>
<tr>
<th></th>
<th>NVIDIA</th>
<th>AMD</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BLAS</strong></td>
<td>cuBLAS</td>
<td>rocBLAS</td>
<td>oneMKL</td>
</tr>
<tr>
<td><strong>Linear Solvers</strong></td>
<td>cuSOLVER</td>
<td>In-works (rocSOLVER)</td>
<td>oneMKL</td>
</tr>
<tr>
<td><strong>Random Numbers</strong></td>
<td>cuRAND</td>
<td>rocRAND</td>
<td>oneMKL</td>
</tr>
<tr>
<td><strong>FFT</strong></td>
<td>In-works (cuFFT)</td>
<td>In-works (rocFFT)</td>
<td>In-works (onemkl::dft)</td>
</tr>
</tbody>
</table>
How to port existing CUDA to SYCL?

Intel® DPC++ Compatibility Tool
Assist in migrating CUDA® applications to SYCL/DPC++, extending user choices

- Assists developers migrating code written in CUDA® to DPC++
- Target is to migrate up to 80-90% of code automatically
- Inline comments are provided to help developer complete code


Refer to software.intel.com/articles/optimization-notice for more information regarding performance & optimization choices in Intel software.
Porting CUDA projects to SYCL

SYCLomatic: A New CUDA*-to-SYCL* Code Migration Tool
(previously referred to as DPCT, DPC++ Compatibility Tool)

CUDA Source Code
SYCLomatic tool
Human Readable C++ with SYCL Single Source Code with inline comments
Compilers, Libraries, Analyzers, Debuggers
Run on Multiple Devices (Architecture/Vendor Agnostic)

Nvidia CUDA
Migrate
C++ with SYCL
Build
Deploy

CUDA Source Code
SYCLomatic tool
Human Readable C++ with SYCL Single Source Code with inline comments
Compilers, Libraries, Analyzers, Debuggers
Run on Multiple Devices (Architecture/Vendor Agnostic)

https://github.com/oneapi-src/SYCLomatic
Instructions to build SYCLomatic (Optional)

Repository: https://github.com/oneapi-src/SYCLomatic

Setup Environment:

```
export SYCLOMATIC_HOME=~/.workspace
export PATH_TO_C2S_INSTALL_FOLDER=~/.workspace/c2s_install
mkdir $SYCLOMATIC_HOME
cd $SYCLOMATIC_HOME
```

```
git clone https://github.com/oneapi-src/SYCLomatic.git
```

Build Instructions:

```
cd $SYCLOMATIC_HOME
mkdir build
cd build
```

```
cmake -G Ninja -DCMAKE_INSTALL_PREFIX=$PATH_TO_C2S_INSTALL_FOLDER -DCMAKE_BUILD_TYPE=Release
-LLVM_ENABLE_PROJECTS="clang" -DLLVM_TARGETS_TO_BUILD="X86;NVPTX" ../SYCLomatic/llvm
ninja install-c2s
```

Post Installation:

```
export PATH=$PATH_TO_C2S_INSTALL_FOLDER/bin:$PATH
export CPATH=$PATH_TO_C2S_INSTALL_FOLDER/include:$CPATH
```

ALCF Polaris: module load oneapi/upstream already has SYCLomatic
Things to notice when using SYCLomatic

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

1. SYCLomatic is a tool that bridges the cap between CUDA to SYCL

2. “dpct” namespace and headers are used for porting to SYCL

3. “dpct” headers and namespace are not standard SYCL APIs. They are merely wrappers/Helpers

4. (Optional) For a SYCL specification complaint code (or) for production purpose: Consider manually replacing “dpct” with SYCL equivalents
Access to ALCF Polaris

To login:
ssh username@polaris.alcf.anl.gov
Password:

Modules to Load:
module load oneapi/upstream cmake

Repository for hands-on:
git clone https://github.com/argonne-lcf/sycltrain.git
cd sycltrain/9_sycl_of_hell
mkdir -p build

cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_CXX_FLAGS="-fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_80" ..
make -j4

To get a compute node:
qsub -I -A ATPESC2023 -q ATPESC -l select=1:system=polaris -l walltime=60:00 -l filesystems=home:eagle

https://docs.alcf.anl.gov/polaris/getting-started/
Case Study 1: Equivalents for Nvidia Thrust Library?

- Thrust is a C++ template library for CUDA based on the Standard Template Library (STL).
- A rich collection of data parallel primitives such as scan, sort, and reduce, etc.
- Thrust can be utilized in rapid prototyping of CUDA applications where robustness and absolute performance are crucial.

```
thrust::device_vector<int> x = h_vec;
// sort data on the device (This breaks the compile)
thrust::sort(x.begin(), x.end());
```

- oneDPL defines a subset of the C++ standard library which you can use with buffers and data parallel kernels.
- oneDPL extends Parallel STL with execution policies and companion APIs for running algorithms on oneAPI devices.
- Extensions. An additional set of library classes and functions that are known to be useful in practice but are not (yet) included into C++ or SYCL specifications.

```
// sort x!
auto policy = dpstd::execution::make_device_policy<class oneapiSort>( q );
std::sort(policy, x, x+n_points);
q.wait();
```

- Note: oneDPL library is open-source and in-development. Not all features are supported.
- https://github.com/oneapi-src/oneDPL
Experimental Support for CUDA and ROCm devices

Compiling With DPC++ for CUDA GPUs

The following command can be used to compile your code using DPC++ for CUDA backend:

```
clang++ -std=c++17 -fsycl -fsycl-targets=nvptx64-nvidia-cuda-sycldevice -Xsycl-target-backend --cuda-gpu-arch=sm_80 simple-sycl-app.cpp -o simple-sycl-app-cuda
```

Compiling With DPC++ for ROCm GPUs

The following command can be used to compile your code using DPC++ for HIP backend:

```
clang++ -fsycl -fsycl-targets=amdgcn-amd-amdhsa -Xsycl-target-backend --offload-arch=gfx9xx simple-sycl-app.cpp -o simple-sycl-app-rocm
```

*Currently tested for ROCm 4.2.0, gfx906 and gfx908 for MI50 and MI100 GPU targets respectively*
Case Study 1: Transform reduce

Obvious differences in the inclusion of headers

```
#include <thrust/transform_reduce.h>

template<typename InputIterator, typename UnaryFunction, typename OutputType, typename BinaryFunction>
OutputType thrust::transform_reduce(InputIterator first,
                                      InputIterator last,
                                      UnaryFunction unary_op,
                                      OutputType init,
                                      BinaryFunction binary_op)
```

From standard C++ Algorithms library

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>

template< class ExecutionPolicy, class ForwardIt, class T, class BinaryReductionOp, class UnaryTransformOp >
T transform_reduce( ExecutionPolicy&& policy,
                     ForwardIt first,
                     ForwardIt last,
                     T init,
                     BinaryReductionOp reduce,
                     UnaryTransformOp transform );
```
Case Study 1: Transform reduce

```cpp
#include <thrust/transform_reduce.h>

template<typename InputIterator, typename UnaryFunction, typename OutputType, typename BinaryFunction>
OutputType thrust::transform_reduce(InputIterator first, InputIterator last, UnaryFunction unary_op, OutputType init, BinaryFunction binary_op)
```

From standard C++ Algorithms library

```cpp
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>

template< class ExecutionPolicy, class ForwardIt, class T, class BinaryReductionOp, class UnaryTransformOp >
T transform_reduce( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, T init, BinaryReductionOp reduce, UnaryTransformOp transform );
```

`ExecutionPolicy arg, is an additional argument that enables parallelism on CPU or GPU`
# Case Study 1: Transform reduce

```cpp
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>

template< class ExecutionPolicy, class ForwardIt, class T, class BinaryReductionOp, class UnaryTransformOp >
T transform_reduce( ExecutionPolicy&& policy,
               ForwardIt first,
               ForwardIt last,
               T init,
               BinaryReductionOp reduce,
               UnaryTransformOp transform );
```

1. oneDPL adopts uses the standard C++ Algorithms library

2. Most of the thrust algorithms maps to C++ parallel Algorithms library

3. How is oneDPL portable?  
   SYCL kernels are used underneath the oneDPL algorithms, that makes oneDPL portable to other vendors
Case Study 2: Porting cuBLAS API to oneMKL API

```c
void cublasSgemm(cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const float *alpha,
    const float *A, int lda,
    const float *B, int ldb,
    const float *beta,
    float *C, int ldc)
}
```

```cpp
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm(sycl::queue &queue,
        onemkl::transpose transa, onemkl::transpose transb,
        std::int64_t m, std::int64_t n, std::int64_t k,
        Ts alpha,
        const Ta *a, std::int64_t lda,
        const Tb *b, std::int64_t ldb,
        Ts beta,
        Tc *c, std::int64_t ldc,
        const std::vector<sycl::event> &dependencies = {})
}
```

Things to consider when porting existing cu* math libraries to oneMKL
Case Study 2: Porting cuBLAS API to oneMKL API

cuBLAS APIs are column-major by default & oneMKL APIs provide both row-major & column-major

Choose `oneapi::mkl::blas::column_major`

```cpp
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm(sycl::queue &queue,
                    onemkl::transpose transa, onemkl::transpose transb,
                    int m, int n, int k,
                    Ts alpha,
                    const Ta    *a, std::int64_t lda,
                    const Tb    *b, std::int64_t ldb,
                    const float *beta,
                    float *c, std::int64_t ldc)
}
```
Case Study 2: Porting cuBLAS API to oneMKL API

Return types from both the APIs are vastly different.

cuBLAS: Provides info of the status

oneMKL: provides an \texttt{sycl::event}

```cpp
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm(sycl::queue &queue,
                     onemkl::transpose transa, onemkl::transpose transb,
                     std::int64_t m, std::int64_t n, std::int64_t k,
                     Ts alpha,
                     const Ta *a, std::int64_t lda,
                     const Tb *b, std::int64_t ldb,
                     const Tb *beta,
                     float *c, std::int64_t ldc
    )
}
```
Case Study 2: Porting cuBLAS API to oneMKL API

Scalar values (alpha & beta)

cuBLAS API: Requires a pointer

oneMKL API: Requires a scalar

cublasStatus_t cublasSgemm(cublasHandle_t handle,
cublasOperation_t transa, cublasOperation_t transb,
int m, int n, int k,
const float *alpha,
const float *A, int lda,
const float *B, int ldb,
const float *beta,
float *C, int ldc)

namespace oneapi::mkl::blas::column_major {
  sycl::event gemm(sycl::queue &queue,
                   onemkl::transpose transa, onemkl::transpose transb,
                   std::int64_t m, std::int64_t n, std::int64_t k,
                   Ts alpha,
                   const Ta    *a, std::int64_t lda,
                   const Tb    *b, std::int64_t ldb,
                   Ts beta,
                   Tc          *c, std::int64_t ldc,
                   const std::vector<sycl::event> &dependencies = {}}
}
Case Study 2: Porting cuBLAS API to oneMKL API

Subtle details:

1. oneMKL (optional): Additional argument of a `sycl::event` to act as a dependency

2. `sycl::queue` is used for oneMKL APIs. There are no equivalents of a special BLAS handle similar to `cublasHandle_t` in oneMKL

Performance: oneMKL piggy-backs underneath by calling respective vendor APIs (cuBLAS/rocBLAS)
Case Study 3: How to port a large project to SYCL

https://github.com/ccsb-scripps/AutoDock-GPU

Native CUDA kernels (files with extensions .cu)

SYCL (ported files with extensions dp.cpp)
Build Your Own Compiler (~30 mins, plan accordingly)

Get the source code: (takes a while)

```
git clone -b sycl --depth 1 https://github.com/intel/llvm.git
```

Build & Install: (takes a while too)

```
module load cudatoolkit/11.8.0
module list
cd llvm

CUDA_LIB_PATH=/soft/compilers/cudatoolkit/cuda-11.8.0/lib64/stubs python3 $PWD/buildbot/configure.py --cmake-gen "Unix Makefiles" --
cuda --cmake-opt="-DCUDA_TOOLKIT_ROOT_DIR=/soft/compilers/cudatoolkit/cuda-11.8.0"
--cmake-opt="-DSYCL_LIBDEVICE_GCC_TOOLCHAIN=/opt/cray/pe/gcc/11.2.0/snos"
--cmake-opt="-DCMAKE_C_COMPILER=/opt/cray/pe/gcc/11.2.0/snos/bin/gcc"
--cmake-opt="-DCMAKE_CXX_COMPILER=/opt/cray/pe/gcc/11.2.0/snos/bin/g++"
--cmake-opt="-DGCC_INSTALL_PREFIX=/opt/cray/pe/gcc/11.2.0/snos" --llvm-external-projects openmp

python3 $PWD/buildbot/compile.py -j64
```

Where are my SYCL compilers installed?

```
train515@nid001608:~/llvm/build/bin>
```
Useful resources

**oneAPI Beta downloads and documentation:**

**DPC++ Compatibility Tool Getting Started:**

**DPC++ Compatibility Tool User Guide:**

**DevCloud access:**
https://intelsoftwaresites.secure.force.com/devcloud/oneapi

**Codeplay migration docs:**