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

![Memory Model Table](https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#table.USM.allocation.characteristics)
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><strong>Global memory</strong></td>
<td><strong>Grid (all threads)</strong></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

Host Code

```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;
}
```

Device Code

- 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;
}

Step 1: Create SYCL queue to create GPU
Step 2: Allocate device memory
Step 3 (H2D): copy inputs “A” & “B” to GPU
Step 4 (Compute): Run the kernel on device
Step 5 (D2H): Copy result “devC” back to host
#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

```c++
#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;
}
```

Device Code

```c++
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));
```

Host Code

```c++
myQueue.wait(), wait for D2H to complete before printing “C”
```
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.

```cpp
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.

```cpp
// sort x!
auto policy = dpl::execution::make_device_policy<class oneapifSort>( 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
Case Study 2: 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.
Case Study 2: Continued... AutoDock-GPU

Native CUDA kernels (files with extensions .cu)

SYCL (ported files with extensions dp.cpp)

https://github.com/ccsb-scripps/AutoDock-GPU
Case Study 3: What are my options for cuBLAS?

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>BLAS</td>
<td>cuBLAS</td>
<td>rocBLAS</td>
<td>oneMKL</td>
</tr>
<tr>
<td>Linear Solvers</td>
<td>cuSOLVER</td>
<td>In-works (rocSOLVER)</td>
<td>oneMKL</td>
</tr>
<tr>
<td>Random Numbers</td>
<td>cuRAND</td>
<td>rocRAND</td>
<td>oneMKL</td>
</tr>
<tr>
<td>FFT</td>
<td>In-works (cuFFT)</td>
<td>In-works (rocFFT)</td>
<td>In-works (onemkl::dft)</td>
</tr>
</tbody>
</table>
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:
https://developer.codeplay.com/products/computecpp/ce(guides/sycl-for-cuda-developers
https://developer.codeplay.com/products/computecpp/ce(guides/sycl-for-cuda-developers/migration

https://github.com/argonne-lcf/sycltrain/tree/master/9_sycl_of_hell