SYCL Essentials

# Programming C++ with SYCL

Learn how C++ with SYCL can be used for heterogenous computing

intel®

# C++ with SYCL

- Enables programming for heterogenous hardware from different vendors.
- Single source that has host code and kernel code to offload to CPU, GPU, FPGA or other accelerator devices.
- Based on Open Standards C++ and Khronos* SYCL

# A Complete SYCL Program

Single source

- Host code and heterogeneous accelerator kernels can be mixed in same source files

Familiar C++

- Library constructs add functionality, such as:

| Construct | Purpose |
|---|---|
| queue | Work targeting |
| malloc_shared | Data management |
| parallel_for | Parallelism |

Host code

Accelerator device code

Host code

```
#include <CL/sycl.hpp>
constexpr int N=16;

int main() {
  sycl::queue q;
  int *data = sycl::malloc_shared<int>(N, q);
  q.parallel_for(N, [=](auto i) {
      data[i] = i;
  }).wait();
  for (int i=0; i<N; i++) std::cout << data[i] << "\n";
  sycl::free(data, q);
  return 0;
}
```

# Compiling SYCL Program

To compile for CPUs and GPUs

- Install the Intel oneAPI Base Toolkit which includes the Intel oneAPI C++/DPC++ Compiler
- Setup environment variable once and then use the icpx compiler with –fsycl flag to compile one or multiple C++/SYCL source files as shown below:

```
source /project/p_covidpre/source_libs
icpx -fsycl -fsycl-targets=nvptx64-nvidia-cuda test.cpp
```

# SYCL Classes

Let's learn about some important SYCL classes required to program for offloading computation to devices.

# Queue

- **sycl::queue** is a mechanism where work is submitted to a device.
- A queue **submits command groups** to be executed by the SYCL runtime
- A queue maps to one device

```
sycl::queue q;

q.submit([&](sycl::handler& h) {
    // COMMAND GROUP CODE
});
```

# Device

- The **device** class represents the capabilities of the accelerators.
- The device class contains member functions for querying information about the device, which is useful for SYCL programs where multiple devices are created.
- The function get_info gives information about the device:
  - Name, vendor, and version of the device
  - The local and global work item IDs
  - Width for built in types, clock frequency, cache width and sizes, online or offline

```
sycl::queue q;
sycl::device my_device = q.get_device();
std::cout << "Device: " << my_device.get_info<sycl::info::device::name>() << std::endl;
```

# Choosing Where Device Kernels Run

## Work is submitted to queues

- Each queue is associated with exactly one device (e.g. a specific GPU or FPGA)
- You can:
  - Decide which device a queue is associated with (if you want)
  - Have as many queues as desired for dispatching work in heterogeneous systems

| | |
|---|---|
| Create queue targeting any device (compiler picks best available): | `queue();` |
| Create queue targeting a pre-configured classes of devices: | `queue(cpu_selector{});`<br>`queue(gpu_selector{});`<br>`queue(intel::fpga_selector{});`<br>`queue(accelerator_selector{});` |
| Create queue targeting specific device (custom criteria): | `class custom_selector : public device_selector {`<br>`  int operator()(……   // Any logic you want!`<br>`…`<br>`queue(custom_selector{});` |

# Kernel

- The kernel class encapsulates methods and data for executing code on the device when a command group is instantiated
- Kernel object is not explicitly constructed by the user
- Kernel object is constructed when a kernel dispatch function, such as parallel_for, is called

```
sycl::queue q;
q.submit([&](sycl::handler& h) {
  h.parallel_for(N, [=](auto i) {
    c[i] = a[i] + b[i];
  });
});
```

# SYCL language and runtime

- SYCL language and runtime consists of a set of C++ classes, templates, and libraries
- **Application scope** and **command group scope** :
    - Code that executes on the host
    - The full capabilities of C++ are available at application and command group scope
- **Kernel scope**:
    - Code that executes on the device.
    - At kernel scope there are limitations in accepted C++

# Parallel Kernels

- Parallel Kernel allows multiple instances of an operation to execute in parallel.
- Useful to offload parallel execution of a basic for-loop in which each iteration is completely independent and in any order.
- Parallel kernels are expressed using the parallel_for function

**for**-loop in CPU application

```
for(int i=0; i < 1024; i++){
    c[i] = a[i] + b[i];
});
```

Offload to accelerator using **parallel_for**

```
q.parallel_for(1024, [=](auto i){
    c[i] =  a[i] + b[i];
});
```

# Basic Parallel Kernels

Simplest way to write a kernel to offload execution to device. Mapping execution to hardware resources cannot be controlled and are done by compiler implementation

The functionality of basic parallel kernels is exposed via range, id and item classes

- range class is used to describe the iteration space of parallel execution
- id class is used to index an individual instance of a kernel in a parallel execution
- item class represents an individual instance of a kernel function, exposes additional functions to query properties of the execution range

```
h.parallel_for(range<1>(1024), [=](id<1> idx){
          // CODE THAT RUNS ON DEVICE
});
```

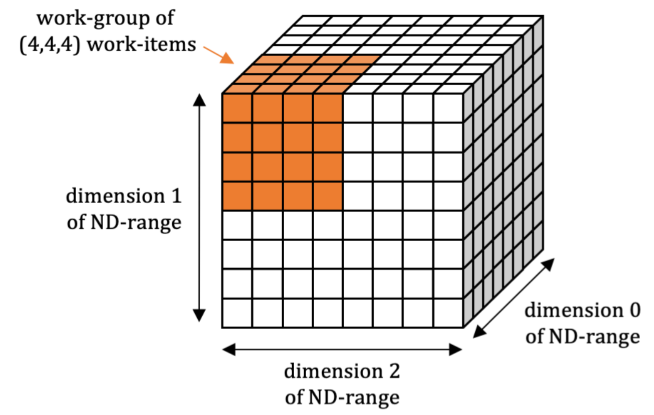
```
h.parallel_for(range<1>(1024), [=](item<1> item){
          auto idx = item.get_id();
          auto R = item.get_range();
          // CODE THAT RUNS ON DEVICE
});
```

# ND-Range Kernels

Basic Parallel Kernels are easy way to parallelize a for-loop but does not allow performance optimization at hardware level.

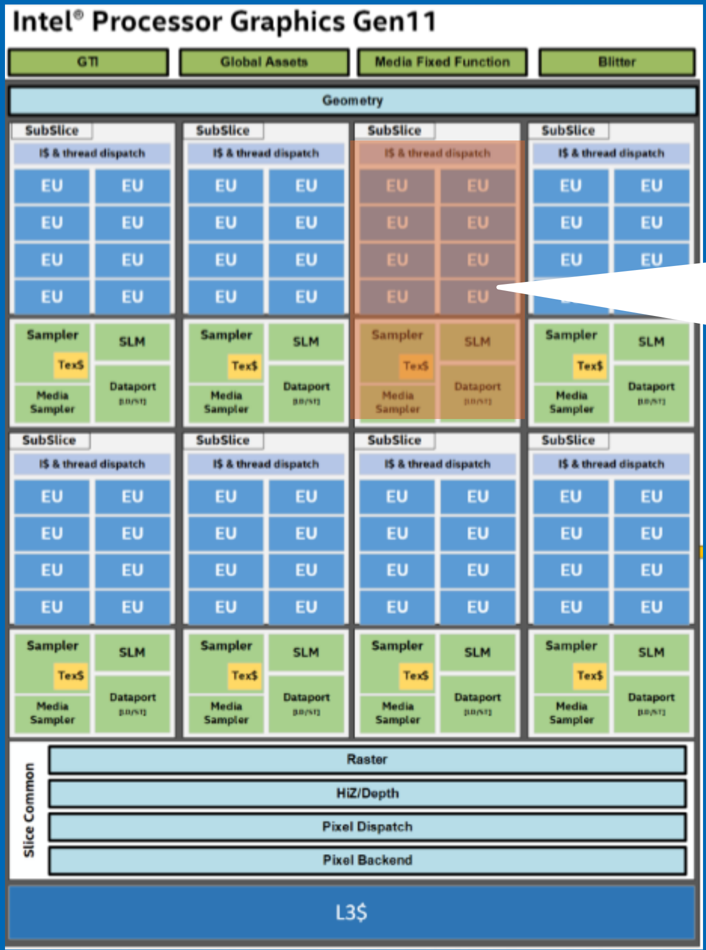
ND-Range kernel is another way to expresses parallelism which enable low level performance tuning by providing access to local memory and mapping executions to compute units on hardware.

- The entire iteration space is divided into smaller groups called work-groups, work-items within a work-group are scheduled on a single compute unit on hardware.
- The grouping of kernel executions into work-groups will allow control of resource usage and load balance work distribution.

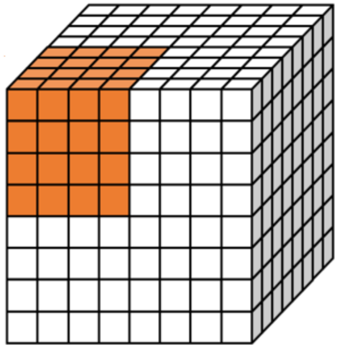



ND-Range

# ND-Range Kernels




**work-group** executions are mapped to Execution Units on hardware.



multiple work-groups can execute concurrently depending on number of execution units on hardware.

# ND-Range Kernels

The functionality of nd_range kernels is exposed via nd_range and nd_item classes

```
h.parallel_for(nd_range<1>(range<1>(1024),range<1>(64)), [=](nd_item<1> item){
    auto idx = item.get_global_id();
    auto local_id = item.get_local_id();
    // CODE THAT RUNS ON DEVICE
});
```

global size

work-group size

- nd_range class represents a grouped execution range using global execution range and the local execution range of each work-group.
- nd_item class  represents an individual instance of a kernel function and allows to query for work-group range, global index, work-group index, group id and more.

# Memory Models

SYCL programs can either use a pointer-based memory model called Unified Shared Memory or can use Buffer-Accessor memory model

- Buffer Memory Model – defines shared array of one, two or three dimensions that can be used by the SYCL kernel and has to be accessed using accessor classes
- Unified Shared Memory (USM) – pointer-based memory model to access data on host and device

# Buffer Memory Model

The buffer class defines a shared array of one, two or three dimensions that can be used by the SYCL kernel and has to be accessed using accessor classes

Buffers: Encapsulate data in a SYCL application

- Across both devices and host!

Accessors: Mechanism to access buffer data

- Creates data dependencies in the SYCL graph that order kernel executions
- Creating host accessor is a blocking call and will only return after all enqueued kernels that modify the same buffer in any queue completes execution and the data is available to the host.

```
sycl::queue q;
std::vector<int> data(N, 10);
sycl::buffer buf(data);
q.submit([&](handler& h) {
  sycl::accessor a(buf, h, sycl::read_write);
  h.parallel_for(N, [=](auto i) {
    a[i] += i;
  });
});
sycl::host_accessor ha(buf, sycl::read_only);
for (int i = 0; i < N; i++) std::cout << data[i] << " ";
```

# Unified Shared Memory

Unified Shared Memory enables shared allocation that uses same pointer reference on host and device, and data is moved implicitly between host and device

- sycl::malloc_shared will allocate memory that can be accessed by both host and device.
- Host can modify this data
- Device can also modify the same data in kernel code
- wait() method is a blocking call that will return after queue completes execution
- The resulting data is available on host

```
sycl::queue q;
auto data = sycl::malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.parallel_for(N, [=](auto i){
        data[i] += 1;
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
sycl::free(data, q);
```

# Unified Shared Memory

Unified Shared Memory also enables device memory allocation and can move data explicitly between host and device

- sycl::malloc_device will allocate memory on device.
- memcpy will copy data from host to device
- Device can modify the data in kernel code
- memcpy will to copy back data from device to host
- The resulting data is available on host
- wait() method is a blocking call that will return after queue completes execution

```
sycl::queue q;
int data[N];
for (int i=0;i<N;i++) data[i] = 10;
auto data_d = sycl::malloc_device<int>(N, q);
q.memcpy(data_device, data, sizeof(int) * N).wait();
q.parallel_for(N, [=](auto i) {
  data_device[i] += 1;
}).wait();
q.memcpy(data, data_device, sizeof(int) * N).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
sycl::free(data_device, q);
```

# Choosing the right Memory Model

## SYCL Buffers are powerful and elegant

- Use if the abstraction applies cleanly in your application, and/or buffers aren't disruptive to your development.
- Kernel dependencies are implicitly handled
- Working with 2/3-dimensional data structures may be easier.

## USM provides a familiar pointer-based C++ interface

- Useful when porting C++/CUDA* code to SYCL, by minimizing changes
- Kernel dependencies must be explicitly handled
- Note that shared allocation is intended to get to functionality quickly, but not intended to provide peak performance out of box, device allocation with explicit data movement is recommended for performance.

# SYCL Code Anatomy

- SYCL programs require the include of CL/sycl.hpp
- It is recommended to employ the namespace statement to save typing repeated references into the sycl namespace

```
#include <CL/sycl.hpp>
using namespace sycl;
```

# SYCL Code Anatomy

```
void sycl_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers allocated in global memory
    accessor A(buf_a, h, read_only);
    accessor B(buf_b, h, read_only);
    accessor C(buf_c, h, write_only);
    //Specify the device kernel body as a lambda function
    h.parallel_for(range<1>(N), [=](auto i){
      C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue
(developer can specify a device type via device selector or use default selector)

Step 2: create buffers
(represent both host and device memory)

Step 3: submit a command for (asynchronous) execution

Step 4: create buffer accessors to access buffer data on the device

Step 5: send a kernel (lambda) for execution

Step 6: write a kernel

Kernel invocations are executed in parallel

Kernel is invoked for each element of the range

Kernel invocation has access to the invocation id

Done!
The results are copied to vector `c` at `buf_c` buffer destruction

# Custom Device Selector

The following code shows derived **device_selector** that employs a device selector heuristic. The selected device prioritizes a GPU device because the integer rating returned is higher than for CPU or other accelerator.

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class my_device_selector : public device_selector {
public:
  int operator()(const device& dev) const override {
    int rating = 0;
    if (dev.is_gpu() & (dev.get_info<info::device::name>().find("Intel") != std::string::npos))
      rating = 3;
    else if (dev.is_gpu()) rating = 2;
    else if (dev.is_cpu()) rating = 1;
    return rating;
  };
};
int main() {
  my_device_selector selector;
  queue q(selector);
  std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
  return 0;
}
```

# Asynchronous Execution

Think of a SYCL application as two parts:

1. Host code
2. The graph of kernel executions

These execute independently, except at synchronizing operations

- The host code submits work to build the graph (and can do compute work itself)
- The graph of kernel executions and data movements executes asynchronously from host code, managed by the SYCL runtime

# Asynchronous Execution

## Host

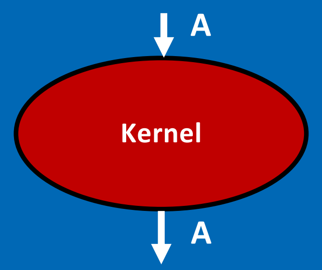
Host code execution

Enqueues kernel to graph, and keeps going

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
  std::vector<int> data(N);
  {
    buffer A(data);
    queue q;
    q.submit([&](handler& h) {
      accessor out(A, h, write_only);
      h.parallel_for(N, [=](auto i) {
        out[i] = i;
      });
    });
  }
  for (int i=0; i<N; ++i) std::cout << data[i];
}
```

## Graph

Graph executes asynchronously to host program



# Kernel Code

```
h.parallel_for(N, [=](auto i){
    // CODE THAT RUNS ON DEVICE FOR ALL WORK-ITEMS
    // C++
});
```

- The kernel code that executes on device can be written entirely using C++ language features.
    - Due to restrictions of the heterogeneous devices where the SYCL kernel will execute, there are certain restrictions on the base C++ language features that can be used inside kernel code.
- SYCL also exposes programming capabilities to access low level hardware which enables tuning for performance.

# Kernel Code

SYCL exposes programming capabilities to access low level hardware and to simplify kernel programming.

| | |
|---|---|
| **Shared Local Memory** | Devices may have dedicated resources for local memory, communicating via local memory will perform better than communicating via global memory |
| **Sub-Groups** | A subset of related work-items within a work-group that execute concurrently. |
| **Group Algorithms** | Group algorithms provide library of optimized algorithms, these can be used on work-groups and sub-groups |
| **Atomic Operations** | Atomic operations enable concurrent access to a memory location without introducing a data race. |
| **Kernel Reductions** | Simplifies reduction in kernels by introducing reduction object in parallel_for. |

# Kernel Reductions

```
q.submit([&](auto &h) {
 sycl::accessor buf_acc(buf, h, sycl::read_only);
 auto sumr = sycl::reduction(sum_buf, h, sycl::plus<>());
   h.parallel_for(sycl::nd_range<1>{data_size, 256}, sumr,
      [=](sycl::nd_item<1> item, auto &sumr_arg) {
        int glob_id = item.get_global_id(0);
        sumr_arg += buf_acc[glob_id];
  }); });
```

- Reduction is a language feature – just create a reduction accessor, pass it to parallel_for and the kernel lambda
- Automatically optimizes parallel reduction

# Exercise

Convert our laplace2d.cpp to use SYCL. Use buffers & reduction

- Don’t forget to re-initialize buffer used for reduction

# Assignment

Convert our LBM code to use SYCL.

Optional: if you submit, it will replace your worst/missing assignment

# Resources

- SYCL Essentials training modules:
  - https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/DPC%2B%2B/Jupyter/oneapi-essentials-training
- Intel GPU Optimization Guide:
  - https://www.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top.html
- SYCL Code Samples:
  - https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/DPC%2B%2B

# Resources

- Download and Install Intel oneAPI Compiler, Libraries and Tools:
  - https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html
- Build open source SYCL compiler:
  - https://github.com/intel/llvm
- SYCL Specification:
  - https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

# Notices & Disclaimers

- Intel technologies may require enabled hardware, software or service activation
- Performance varies by use, configuration and other factors. Learn more on the Performance Index site.
- No product or component can be absolutely secure.
- All product plans and roadmaps are subject to change without notice
- 