Queues, Command Groups, and Kernels

Heterogeneous Programming with SYCL

André Cerqueira

November 18, 2024

Funded by Project 10110190 – EUROCC2, with financial support from FCT/MCTES through national funds (PIDDAC).

Session Objectives

- Understand how work is organized in SYCL applications.
- Learn about queues and their role in execution.
- Explore command groups and task graphs.
- Queue actions: single task, parallel for, and parallel for work group.
- Learn kernel programming: lambdas and function objects.
- Analyze memory operations: `copy`, `update_host`, `fill`.

Queues in SYCL

- A queue connects the host to a single device.
- All actions (kernels, data transfers, etc.) are submitted to a queue.
- Queues support:
  - Asynchronous execution.
  - Task graph generation.

Creating Queues

```
sycl::queue defaultQueue;

// Selecting a GPU if available
sycl::queue gpuQueue(sycl::gpu_selector{});

// Default device selection
sycl::queue deviceQueue(sycl::default_selector{});
```

Work Submission and Command Groups in SYCL

- **How is work submitted in SYCL?**
  - Work is submitted to a **queue**, which connects the host program to the device.
  - We can submit code directly with actions as **Q.parallel_for()**, or
  - We can use a more personalized submission, that defines a **command group**, allowing us to manually specify the task and its dependencies (**Q.submit()**).
- **What is a Command Group?**
  - Encapsulates work submitted to the queue.
  - Defined with a `sycl::handler`, which:
    - Describes dependencies using accessors or events.
    - Contains one action (e.g., kernel execution).
- **Why are Command Groups important?**
  - Define dependencies, contributing to the task graph construction.

Actions in SYCL: `parallel_for`

- **Actions**: Operations submitted to the queue for execution.
- `parallel_for`:
  - Executes a kernel across a specified range of work-items.
  - A single instruction, multiple thread (SIMT) abstraction.
  - Takes two primary arguments:
    - **Execution Range**: Defines the total number of work-items.
    - **Kernel Function**: A callable, often a lambda function, executed by each work-item.
- **When to use?**
  - For data-parallel tasks where the same operation is applied to many elements.

What is a Command Group?

- A **command group** is a unit of work submitted to the queue.
- Defined using a `sycl::handler`.
- **What does a Command Group contain?**
  - **Host Code**: Defines dependencies, such as buffer access or event-based dependencies.
  - **One Action**: Examples include:
    - `parallel_for`: A data-parallel kernel execution.
    - Memory operations, such as `fill`, `copy`, or `update_host`.
- **Why is it used?**
  - Enqueues work asynchronously to the queue.
  - Contributes to task graph construction and scheduling.

Kernels in SYCL

- Kernels represent work executed on a device.
- Two main types of writing kernels:
  - **Lambdas**: Concise, easy to write.
  - **Function objects**: More verbose, but reusable.

Lambda Kernel Example

```
Q.submit([&](handler& cgh) {
    cgh.parallel_for<>(
        range<1>{1024}, [=](id<1> idx) { buffer_acc[idx] += 1; });
});
```

Function Object Kernel Example

```
class PlusOne {
    accessor<int> acc;
    public:
        PlusOne(accessor<int> a) : acc(a) {}
        void operator()(id<1> idx) { acc[idx] += 1; }
};

Q.submit([&](sycl::handler& cgh) {
    accessor acc{buffer, cgh};
    cgh.parallel_for<PlusOne>(range<1>{1024}, PlusOne(acc));
});
```

Kernel Restrictions

- Must return `void`.
- Cannot use RTTI (Run-Time Type Information) or dynamic memory allocation.

SYCL Execution Model

- SYCL organizes work as a **task graph**.
- **Task Graph** (DAG: Directed acyclic graph):
  - **Nodes**: Actions to be performed (e.g., kernel execution or data transfer).
  - **Edges**: Dependencies between actions (e.g., data prerequisites).
- Runtime manages the task graph and executes asynchronously.

Benefits of Task Graphs

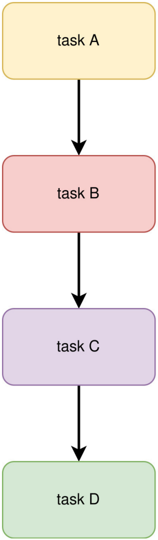
1. Automatic dependency resolution.
2. Optimized task scheduling.

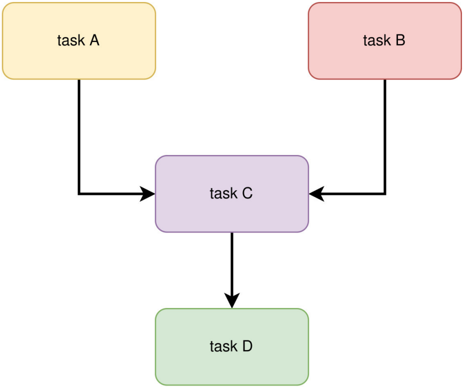
In-Order and Out-of-Order Queues

- **Out-of-Order Queues** (default):
  - The runtime decides task ordering based on data dependencies.
  - Allows maximum flexibility for scheduling.
- **In-Order Queues**:
  - Tasks execute sequentially in the order submitted.
  - Limits scheduling flexibility but simplifies reasoning about task order.
- Example:

```
sycl::queue ioQueue(sycl::property::queue::in_order());
```

Task Graph Visualization (Diagram)





How are Dependencies created

- **Memory:**
  - The runtime infers dependencies based on buffer access modes.
- **Event-Based:**
  - Use `depends_on` to define dependencies explicitly.

Event Dependency Example

```
auto e1 = Q.parallel_for(range{N}, [=](id<1> id) {
    a[id] = 1.0; // Task A
});
auto e2 = Q.parallel_for(range{N}, [=](id<1> id) {
    b[id] = 2.0; // Task B
});
auto e3 = Q.parallel_for(range{N}, {e1, e2}, [=](id<1> id) {
    a[id] += b[id]; // Task C depends on A and B
});
Q.single_task(e3, [=]() {
    for (int i = 1; i < N; i++) a[0] += a[i]; // Task D
});
```

Memory Operations Overview

- Common operations:
  - `copy`: Transfer data between buffers.
  - `fill`: Initialize buffer with a value.
  - `update_host`: Synchronize device memory to host.
- You invoke them as methods on the queue class directly or on the handler class.

Introducing SYCL Streams

- **What are SYCL Streams?**
  - A mechanism for printing from device kernels to the host console.
  - Similar to C++ standard streams (`std::cout`).
- **Why use SYCL Streams?**
  - Useful for debugging and logging within kernels.
  - Provides insights into the behavior of device code.
- **How do SYCL Streams work?**
  - Streams are created using `sycl::stream`.
  - A stream object requires:
    - Buffer size for storing output.
    - Maximum size of a single message.
    - A `sycl::handler`.

Stream Example

```
Q.submit([&](sycl::handler& cgh) {
    sycl::stream out(1024, 256, cgh);

    cgh.single_task<>([=]() {
        out << "Hello, SYCL!" << sycl::endl;
    });
});
```

Summary

- Queues organize and manage tasks.
- Task graph tracks dependencies and optimizes execution.
- Kernels can be defined using lambdas or function objects.
- Memory operations are essential for data movement.
- Synchronization ensures correct execution order.