SYCL tutorial

SC25 - St.Louis (MO) - 16th Nov 2025

Thomas Applencourt, Abhishek Bagusetty,
Aurora Perego, Aditya Sadawarte

Argonne NATIONAL LABORATORY

SYCL: Quick Introduction

Thomas Applencourt

November 14, 2025

Overview of SYCL

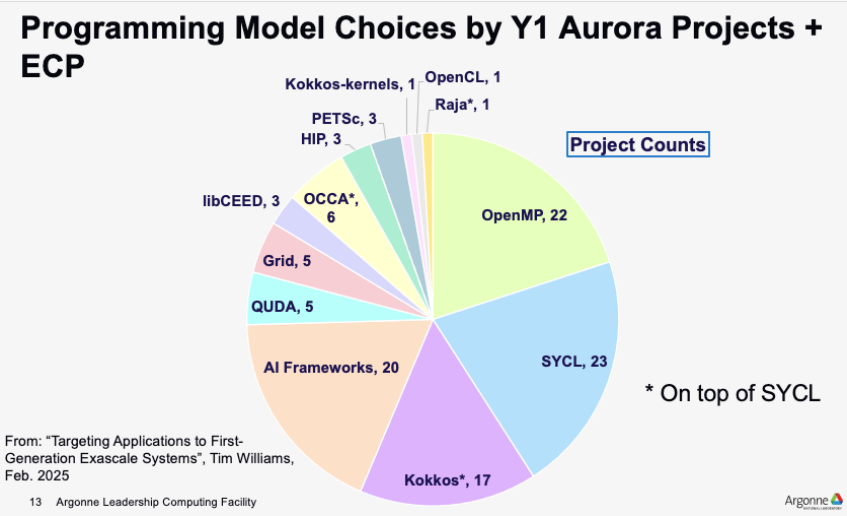
What is SYCL?

- SYCL is 10 years old open Specification by the Kronos Group
- Since recently developed in public at *https://github.com/KhronosGroup/SYCL-Docs*
- 2 Major Implementation: DPCPP (intel), AdaptiveCPP (Heisenberg University) bot other exists

SYCL is a Specification

- ”Portable CUDA”
- Even, if you don’t want to use SYCL implementation, (or are afraid of SYCL futur), you may still want to use the API
- It’s a good API to use in your apps. You can implement it.

Is this thing really used? (At least for Argonne)



What new? What are we doing in the Spec

- Multiple organization are present (Argonne, Heidelberg, Instrubrc, Intel, NVIDIA,...)
- We introduce extension (that we can change before puting stuff in the spec for ever)
- For example, default context, queue flush, more query, etc
- One of those currently brewing is on more ”c / cuda” like API call[1]

[1] for lower submision overhead

Code Example

```
1  #include <sycl/sycl.hpp>
2
3  int main(int argc, char **argv) {
4    sycl::queue Q;
5    std::cout << "Running on "
6              << Q.get_device().get_info<sycl::info::device::name>() << std::endl;
7
8    // Allocate Device Memory
9    int *A = sycl::malloc_device<int>(global_range, Q);
10   // Submit asynchrnoously a kernel who use the memory
11   Q.parallel_for(global_range, [=](auto id) { A[id] = id; });
12   // Syncrhnize
13   Q.wait();
14   // Allocate Host Memory
15   std::vector<int> A_host(global_range);
16   // Copy the device memory to the host
17   Q.copy(A, A_host.data(), global_range).wait();
18   // Memory free
19   sycl::free(A, Q);
20 }
```

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

More than a API

Of course, API are only the tip of the iceberg

- Have access to math function (sin,cos,sqrt)
- Have access to intrinsic (shuffle, popcount, swizzle)
- MKL Library, and lot of other
- Tool to port cuda to SYCL...

How to Compile and Run

How to Compile?

```
$ icpx -fsycl foo.cpp #JIT -> spirv
$ icpx -fsycl-targets=spir64_gen foo.cpp  # AOT -> genISA
```

How To Run?

Use the default queue

```
1 sycl::queue Q;
```

Then Use *gpu_tile_compact*. Each rank will see one tile
mpirun gpu_tile_compact ./a.out. All's right with the world

To be extra fancy

No affinity, you need to understand visibility, hierarchy mode, subdevice, and context.

```
1  int world_rank;
2  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
3  sycl:: device D = sycl::get_devices()[world_rank];
4  sycl::queue Q(D, sycl::context{D});
```

```
mpirun ./a.out
```

Conclusion

One slide

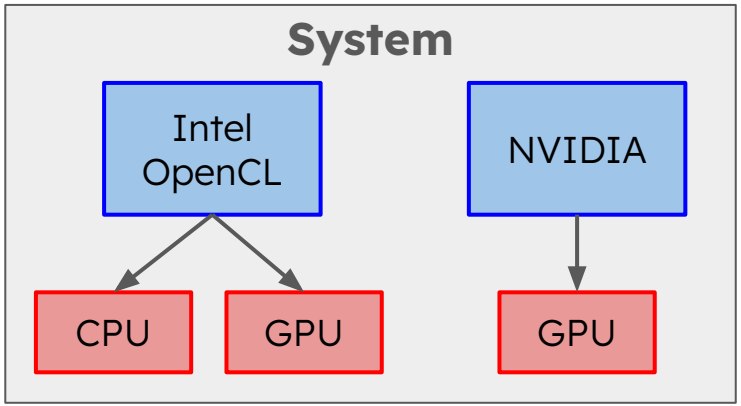
- SYCL is ”just a portable wrapper C++” on top of lower API / C kernel flavor (CUDA, OpenCL, L0, Hip)
- Can use fancy allocator, mdspan, just ping me if interested (example in the repo posted previously)
- Use `"-fsycl"`
- Don’t hesitate to talk to me or Abhi for any question![2]
- Please, if you hate SYCL/ if SYCL is missing stuff, just open an issue on the specification *https://github.com/KhronosGroup/SYCL-Docs/issues* [3]

[2] We can rant about C++, other framework together, python and how we should all go back to Fortran. 77.
[3] The issue will be most likely ignored, but at least you can sleep peacefully knowing you did your part

Part 1: device discovery

SYCL SYSTEM TOPOLOGY

- A SYCL application can execute work across a range of different heterogeneous devices.
- The SYCL runtime will discover a set of **platforms** that are available in the system.
  - Each platform represents a backend implementation such as Intel OpenCL or Nvidia PTX.
- The SYCL runtime will also discover all the **devices** available for each of those platforms.
  - CPU, GPU, FPGA, and other kinds of accelerators.
- Platforms and devices are represented by the `platform` and `device` classes respectively.



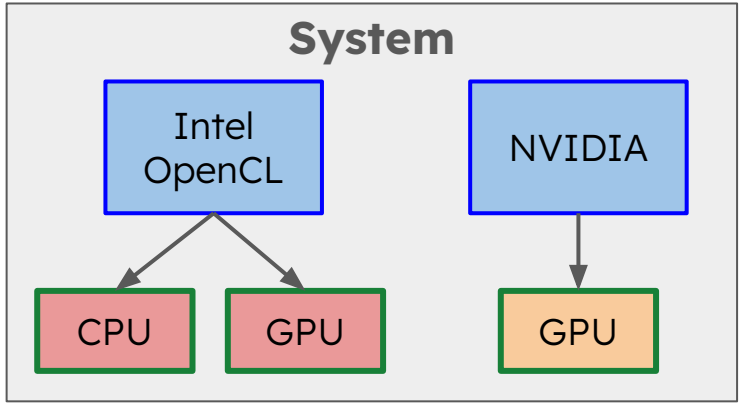
QUERYING THE TOPOLOGY

- In SYCL there are two ways to query a system's topology.
  - The topology can be **manually** queried and iterated over via APIs of the platform and device classes.
  - The topology can be automatically queried and iterated over using a use specified heuristic by a device **selector** object.

QUERYING MANUALLY

- The platform class provides the static function `get_platforms`.
  - It retrieves a vector of all available platforms in the system.

```
auto platforms =
        platform::get_platforms();
```



- The `platform` class provides the member function `get_devices` that returns a vector of all devices associated with that platform.

```
auto intelDevices = intelPlatform.get_devices();
```

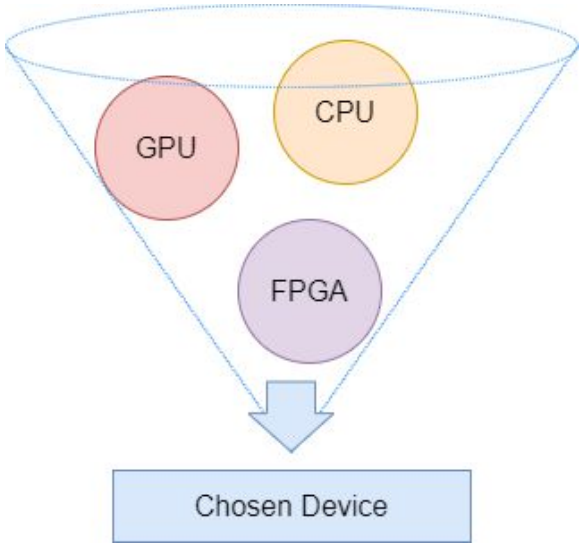
- The device class also provides the static function `get_devices` that returns a vector of all available devices in the system.

```
auto devices = device::get_devices();
```

QUERYING WITH A DEVICE SELECTOR

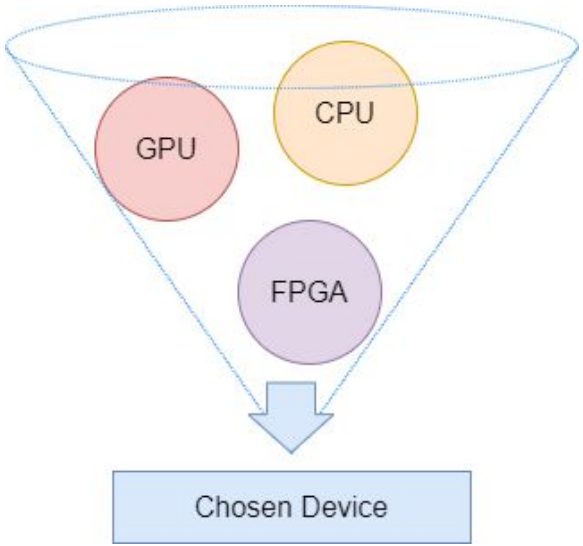
- To simplify the process of traversing the system topology SYCL provides device selectors.
  - A device selector is is a callable C++ object which defines a heuristic for scoring devices.
- A device selector takes a parameter of type `const device&` and gives it a "score".
  - Used to query all devices and return the one with the highest "score".
  - A device with a negative score will never be chosen.

```
auto gpuDevice = device(gpu_selector_v);
```



QUERYING WITH A DEVICE SELECTOR

- SYCL provides a number of standard device selectors, e.g. `default_selector_v`, `gpu_selector_v`, etc.
- Users can also create their own device selectors.
- The default_selector_v is a standard device selector.
  - Chooses a device based on an implementation defined heuristic.
  - A default constructed device or platform will use this selector.




```
auto chosenDevice = device();
auto  chosenDevice = device(default_selector_v);
```

CREATING A CUSTOM DEVICE SELECTOR

- A device selector can be any callable object and must have a function call operator which takes a reference to a device.
- The body of the function call operator defines the heuristic for selecting devices
  - This is where you write the logic for scoring each device
- The device selector can be used to construct a queue.

```
int my_gpu_selector(const device& dev) {
  if (dev.is_gpu()){
    return 1;
  }
  else {
    return -1;
  }
}

int main(int argc, char *argv[]) {
  auto gpuQueue = queue{my_gpu_selector};
}
```

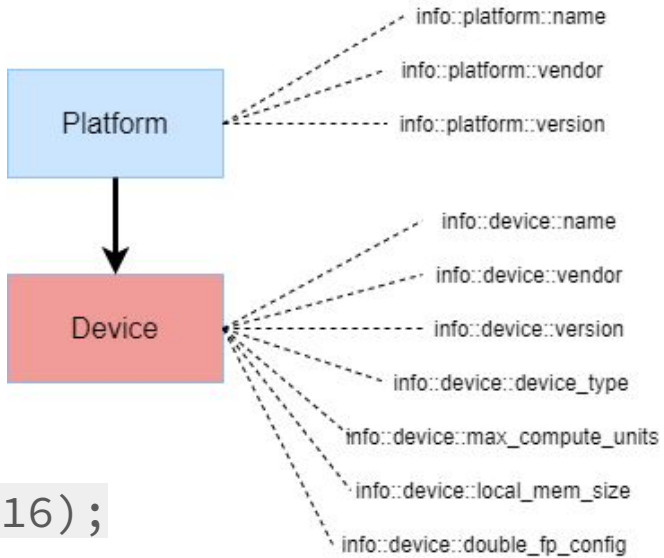
PLATFORM/DEVICE INFO

- Information about platforms and devices can be queried using the template member function `get_info`.
  - The info that you are querying is specified by the template parameter.
- You can also query a device for its associated platform with the `get_platform` member function.

```
auto plt = dev.get_platform();
auto platformName =
  dev.get_info<info::device::name>();
```

- Capabilities of a device or platform are represented by aspects.
  - These can be queried via the `has` member function.

```
bool supportsFp16 = dev.has(aspect::fp16);
```



EXERCISE

https://godbolt.org/z/sxvo9j8n7

- List all the platforms and all the devices for each platform
- Use a device selector (o write your own) to select a device and print some information about it

Useful links:

- Devices — SYCL Reference documentation (Information descriptors)
- SYCL™ 2020 Specification (revision 10)

Part 2: queues

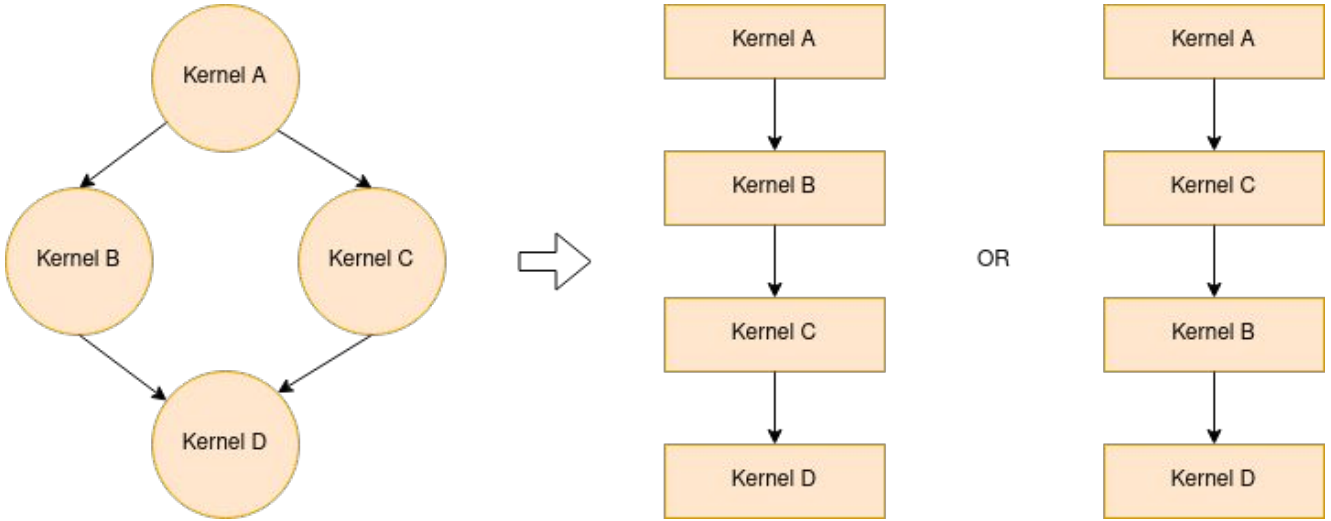
THE QUEUE

- In SYCL all work is submitted via commands to a queue.
- The queue has an associated device that any commands enqueued to it will target.
- There are several different ways to construct a queue.
- The most straight forward is to default construct one.
  - This will have the SYCL runtime choose a device for you.
- As an alternative, it's possible to specify the target device, e.g. using a device selector

```
auto gpuQueue = queue{my_gpu_selector};
```

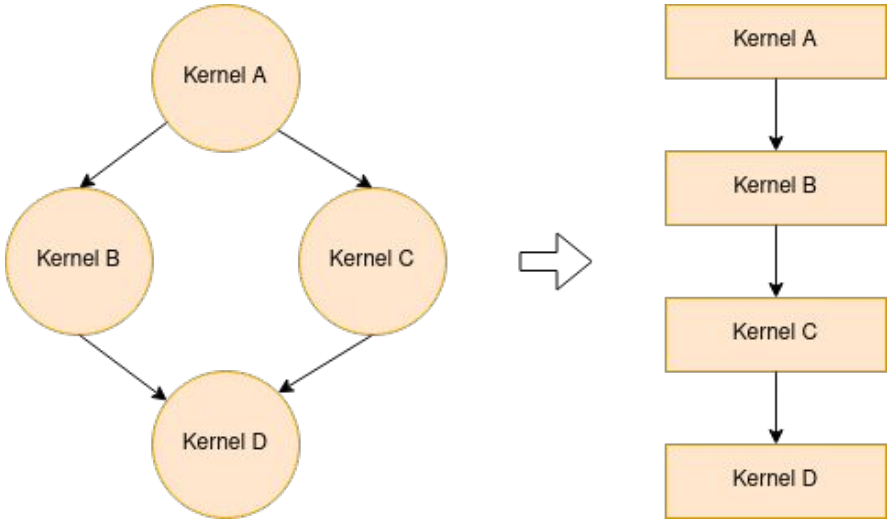
OUT-OF-ORDER EXECUTION

- SYCL queues are by default out-of-order.
- This means commands are allowed to be overlapped and re-ordered
  - they can execute concurrently if hardware allows it
  - dependencies can be provided to ensure consistency.



IN-ORDER EXECUTION

- SYCL queues can be configured to be in-order.
- This mean commands must execute strictly in the order they were enqueued.



OUT-OF-ORDER VS IN-ORDER EXECUTION

out-of-order

- Define manually the order (error prone)
- Manual scheduling mapping tasks to queues
  - Allow concurrency between queues
  - Painful to extract full concurrency
- Dynamic scheduling with dependencies
  - potentially higher latency overhead
  - Scheduling done automatically (maximize concurrency)

```
queue Q;
for (std::function task: tasks)
  Q.submit(task);
Q.wait();
```

in-order

- equivalent of CUDA/HIP stream
- All commands are executed serially
- Ease of programming (no race conditions can occur)
- Potentially lower-latency than out-of-order queues
- Doesn't allow concurrency, potentially suboptimal performance

```
queue Q{property::queue::in_order};
for (std::function task : tasks)
  Q.submit(task);
Q.wait();
```

EXERCISE

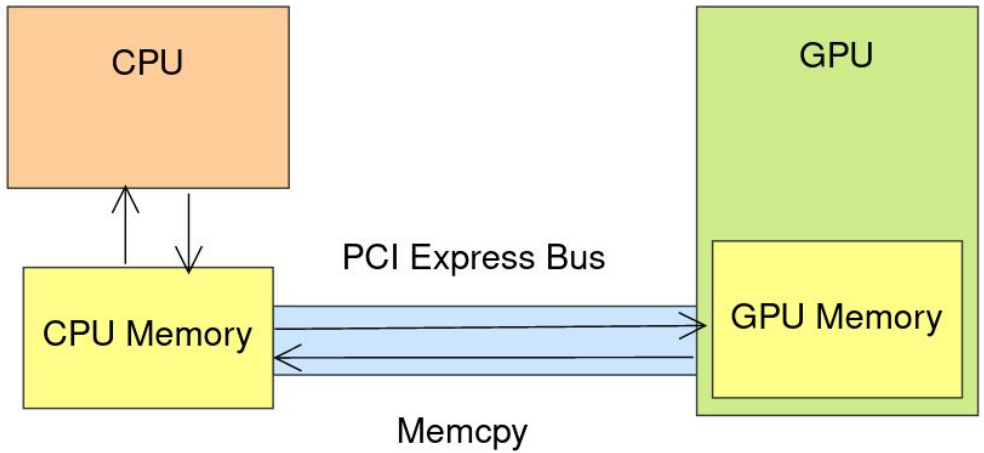
https://godbolt.org/z/Mjq58K1s4

- Switch between in-order and out-of-order queues in the example and check the different behaviours

Part 3: managing data

CPU AND GPU MEMORY

- A discrete GPU has its own memory, separate to CPU memory.
- In order for the GPU to use memory from the CPU, the following actions must take place (either explicitly or implicitly):
  - Memory allocation on the GPU.
  - Data migration from the CPU to the allocation on the GPU.
  - Some computation on the GPU.
  - Migration of the result back to the CPU.
- Memory transfers between CPU and GPU are a bottleneck.
- We want to minimize these transfers, when possible.



MEMORY MODELS

In SYCL there are two models for managing data:

- The buffer/accessor model.
- The USM (unified shared memory) model.

⚠ different meaning wrt CUDA

buffer/accessor model

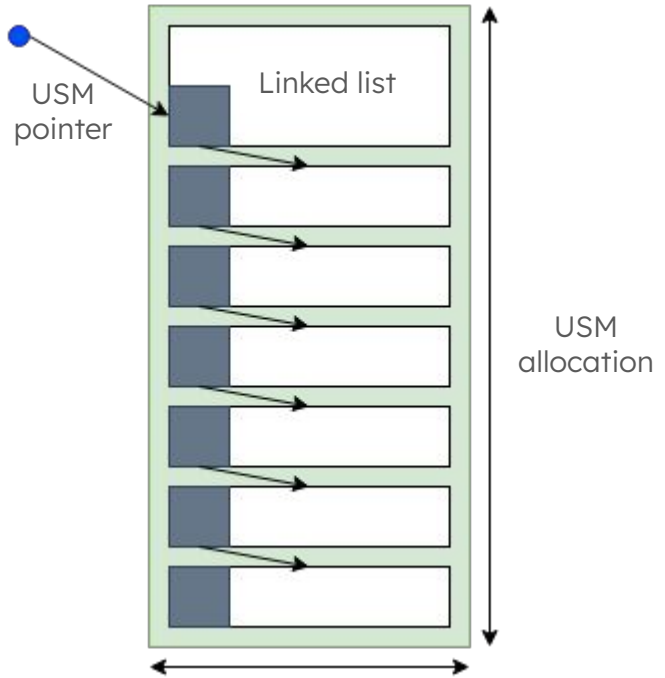
- High-level abstraction
- Takes care of allocations and copies, automatic synchronization
- Buffer manages data across the host and the device(s) and accessors are used to access data
- Automatically manages dependencies ⚠ not covered in details

USM model

- Lower-level pointer-based data management
  - fine grained control
- Explicit memory management
- Unified virtual address space
- Shared memory allocations
- Easier to port existing C or C++ code to use SYCL

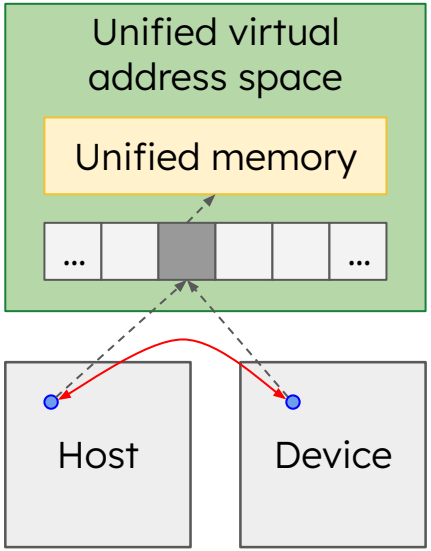
USM: pointer based structures

- USM memory allocations return pointers which are consistent between the host application and kernel functions on a device
- Representing data between the host and device(s) does not require creating accessors
- Data is moved between the host and device(s) in a span of memory in bytes rather than a buffer of a specific type
- Pointers within that region of memory can freely point to any other address in that region



USM allocation types

- USM has three different kinds of memory allocation
  - A **host** allocation is allocated in host memory and is accessible by a device
    `sycl::malloc_host(numBytes, queue);`
    `sycl::malloc_host<T>(numElements, queue);`
  - A **device** allocation is allocated in device memory and is not accessible by the host
    `sycl::malloc_device`
  - A **shared** allocation is allocated in shared memory, is accessible by both and can migrate back and forth based on where it is used (runtime and backends are responsible for the moves)
    `sycl::malloc_shared`

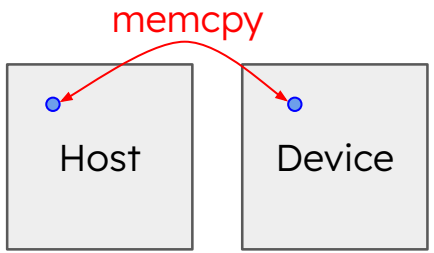


Explicit memory management

- Memory is allocated and data is moved using explicit routines
- Moving data between the host and device(s) does not require accessors or submitting command groups
- The SYCL runtime will not perform any data dependency analysis, dependencies between commands must be managed manually

```
queue.memcpy(dstPtr, srcPtr, numBytes, depEvents);
```

optional vector of dependencies for out-of-order queues
Note: each task submission to a queue returns a `sycl::event` that can be passed here



USING USM

```
// Allocate memory on device
T *device_ptr = sycl::malloc_device<T>(n, myQueue);

// Copy data to device
myQueue.memcpy(device_ptr, cpu_ptr, n * sizeof(T));

// ... Do some computation on device ...

// Copy data back to CPU
myQueue.memcpy(result_ptr, device_ptr, n * sizeof(T)).wait();

// Free allocated data
sycl::free(device_ptr, myQueue);
```

Or use shared memory and avoid copies

- Performance of shared memory accesses may be poor depending on platform

free memory after it has been used to avoid memory leaks

EXERCISE

https://godbolt.org/z/Gf9zeE37Y

- Allocate and fill data on the host
- Create a queue
- Allocate device memory
- Copy data to the device
- Copy back the result and check its correctness
- Free the device memory

SOLUTIONS

- device discovery
  https://godbolt.org/z/41nKPPsKa
- managing data
  https://godbolt.org/z/x6reocM3n

Part 4: enqueuing a kernel

note: say that submit returns an event, useful for next section to wait on it (or say in next section)

Part 5: synchronization

ASYNCHRONOUS EXECUTION

- All command submitted to a queue are done so asynchronously.
- The functions return immediately and the command is run in a background thread.
- This includes individual commands like memcpy and collections of commands derived from a command group.
- This means you have to synchronize with those commands.
- There are a number of reasons why you need to synchronize with commands
  - Await completion of a kernel function.
  - Await the results of a computation.
  - Await error conditions which come from a failure to execute any of the commands.

SYNCHRONIZATION

- There are two ways ways to synchronize with commands.
- Calling `wait` on an event object returned from functions such as memcpy or from enqueuing a kernel function command, either via a command group or a shortcut function.
  - It will wait for that specific command to complete.
  - This effectively creates a blocking operations that will complete in place by immediately synchronizing.

```
auto devicePtr = sycl::malloc_device<int>(n, gpuQueue));
gpuQueue.memcpy(devicePtr, hostData, sizeof(int)).wait();
gpuQueue.parallel_for<kernel_a>(sycl::range{1024},
  [=](sycl::id<1> idx){
  devicePtr[idx] = /* some computation */
}).wait();
```

SYNCHRONIZATION

- There are two ways ways to synchronize with commands.
- Calling `wait` or `wait_and_throw` on the queue itself.
  - It will wait for all commands enqueued to it to complete.
  - Note: don't call `wait` on the queue after every command, but only when results are needed somewhere else (e.g. on the host)

```
auto devicePtr = sycl::malloc_device<int>(n, gpuQueue));
gpuQueue.memcpy(devicePtr, data, sizeof(int));
gpuQueue.wait();
gpuQueue.parallel_for<kernel_a>(sycl::range{1024},
  [=](sycl::id<1> idx){
  devicePtr[idx] = /* some computation */
});
gpuQueue.wait_and_throw();
```

SYNCHRONIZING WITH ERRORS

- Errors are handled by a queue and any asynchronous errors can be produced during any of the synchronization methods we've looked at.
- The best way to ensure all errors are caught is to synchronize by calling `wait` or `wait_and_throw` on the the queue.

Part 1: Local Memory

Motivation for Local Memory

- As we covered earlier global memory is very expensive to access.
- Even with coalesced global memory access if you are accessing the same elements multiple times that can be expensive.
- Instead you want to cache those values in a lower latency memory

CUDA: Shared memory is allocated **per thread block**, so all **threads in the block** have access to the same shared memory

SYCL: Shared memory is allocated **per work-group**, so all **work-items in the group** have access to the same shared memory

Memory Hierarchy & Where “shared” fits

- **Global/device memory**
  - Large, high latency, visible to all work-groups
- **Local memory (a.k.a. shared memory)**
  - On-chip SRAM shared by work-items in a work-group
  - Low latency, limited capacity (e.g., tens of KB)
- **Private memory**
  - Per work-item (registers / spills)
- **SYCL abstraction:**
  - `local_accessor` for buffer model
  - `group_local_memory_for overwrite` for pointer-style shared memory

Shared Memory in CUDA

Dynamic Shared Memory

- Shared memory size not known at “compile-time”
- Approximate size allocated in the kernel launch

```
dynamicReverse<<<1, n, SHM_SIZE>>>(d_d, n);
```

- Inside the kernel body

```
extern __shared__ int s[];
int *integerData = s;
// nI ints
float *floatData = (float*)&integerData[nI];
// nF floats
char *charData = (char*)&floatData[nF];
// nC chars
```

Static Shared Memory

- Shared memory size is known at “compile-time”
- Allocated inside the kernel body

```
__global__ void staticReverse(int *d,
int n)
{
   __shared__ int s[64];
   int t = threadIdx.x;
   int tr = n-t-1;
   s[t] = d[t];
   __syncthreads();
   d[t] = s[tr];
}
```

Shared Memory in SYCL

- Allocates device local memory and provides access to this memory from within a SYCL kernel function
- The local memory that is allocated is shared between all work-items of a work-group
- If multiple work-groups execute simultaneously in an implementation, each work-group receives its own independent copy of the allocated local memory

- `local_accessor` lives in local memory (per work-group)
- Synchronization: Use `barrier()` after writing/reading shared data
- Same pattern as CUDA `__shared__` + `__syncthreads()`

```
constexpr std::size_t WG_SIZE = 256;

q.submit([&](sycl::handler& h) {

  sycl::local_accessor<float, 1> scratch(WG_SIZE, h);

  h.parallel_for(sycl::nd_range<1>(N, WG_SIZE), [=](sycl::nd_item<1> it) {
     auto gid = it.get_global_id(0);
     auto lid = it.get_local_id(0);

     float x = (gid < N) ? in[gid] : 0.0f;
     scratch[lid] = x;

     it.barrier(sycl::access::fence_space::local_space);

     // Modify the shared-memory array
     ……

     if (lid == 0)
       out[it.get_group(0)] = scratch[0];
  });
});
```

Exercise - Local Shared Memory

- TODO 1: Introduce `local_accessor` as the SYCL abstraction of shared memory per work-group.

- TODO 2–4: Connect CUDA-style indexing to SYCL:

```
local_id ≈ threadIdx
global_id ≈ blockIdx*blockDim + threadIdx
group ≈ blockIdx
```

- TODO 5–7: Classic tree reduction pattern using local memory
  - Each work-item loads one element into shared memory
  - Use barriers to ensure all writes are visible

- TODO 8: Only the leader (thread 0) writes the final sum for that block

EXERCISE

https://godbolt.org/z/qYTbooWYG

Solution

- Declare & Allocate local-memory using local_accessor

Useful links:

- Local Accessor — SYCL Reference documentation
- SYCL™ 2020 Specification (revision 11)

Part 2: Reductions

SYCL's built-in reductions

- **`sycl::reduction`** combines:
  - A destination (USM pointer or accessor)
  - A binary operator (`sycl::plus<>`, `sycl::maximum<>`, custom op)
  - An optional identity / initialization property

- **Typical pattern**
  - Kernel signature:
    ```
    parallel_for(range, reduction, [=](item<1> it, auto &acc) {
    acc.combine(x); });
    ```
  - Runtime handles partial sums, local memory, and final combination

- **CUDA comparison**
  - Thrust: `thrust::reduce` over a device range
  - CUB::DeviceReduce: `DeviceReduce::Sum / Max`, etc
  - The difference: in SYCL, the reduction is tied directly to the kernel, rather than a separate library call

SYCL's work-group reductions

- **Group-local operations**
  - `sycl::reduce_over_group(group, value, op)`
  - Also `exclusive_scan_over_group`, `inclusive_scan_over_group`, etc.
  - Operate on all work-items in a work-group or sub-group
- **Usage pattern**
  - Compute a local value 'x' per work-item
  - Call `reduce_over_group` to get the block sum (or min/max, etc.)
  - Use `sycl::leader(group)` or `group.leader()` to let one work-item write the result
- **CUDA comparison**
  - CUB::BlockReduce / BlockScan
  - **cooperative_groups::reduce** over a thread block or a coalesced_group`
  - Manual `__shfl_*` / warp-level reductions
  - Similar idea: block-scoped collective to a single aggregated value

When to use what? (sycl::reduction vs group-algorithms vs manual reductions)

- `sycl::reduction`
  - Simple global reduces: sums, norms, min/max
  - Quickly getting correct & portable code

- Group algorithms (`reduce_over_group`, scans)
  - Per block partial reductions (e.g., one value per cell, tile, or domain chunk)
  - Integrating reductions into richer kernels (stencils, matrix ops, etc.)

- Manual local memory (with `local_accessor` / `group_local_memory_for_overwrite`)
  - Custom data layouts, fused operations, tricky access patterns
  - Extreme fine-tuning (bank conflict avoidance, tiling strategies)

Exercise - Reductions

TODO 1:

```
sycl::reduction(result, sycl::plus<int>{})
```

- `result` = USM pointer for the final scalar
- `sycl::plus<int>` = associative operation

TODO 2:

Kernel lambda signature:

- `[=](sycl::item<1> it, auto& sum)`

- Use `sum.combine(...)` instead of writing to `*result` directly
  Runtime manages local memory, partial sums, and final combination.

EXERCISE

https://godbolt.org/z/xWzjzEEYr

Solution

- Perform reduction

Useful links:

- Reductions — SYCL Reference documentation
- SYCL™ 2020 Specification (revision 11)