Lec16: SYCL

Single source C++ heterogeneous parallelism

What is SYCL?

- Single source C++ (no separate shader language, language extension, it's just C++17)
- Open standard by Khronos group (like OpenCL, OpenGL, Vulkan, etc)
- High-level (slightly higher than CUDA/OpenCL)
- Cross-platform (a few compilers, notable targets NVIDIA, AMD, Intel GPUs, Intel FPGAs, and many CPUs)
- Could be considered very loosely the evolution of OpenCL

Try it out on the course server

- Intel DPC++ compiler is installed
- Set up environments:
  $ source /opt/intel/oneapi/setvars.sh
- Get Sample code:
  $ cp –r /opt/2025s_cosc4397/sycl .
- Build and run
  $ make
  $ ./vadd 100001

Unified Shared Memory (USM)

- USM is pointer based memory management, more familiar traditional C++ programmer
- C: malloc(), free()
  C++: new, delete
  USM: malloc_xxx(), free().
- All types of malloc_() are accessible from device!!
  - malloc_device()
  - malloc_host()
  - malloc_shared()

Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	×	✓	device
host	Allocations in host memory	✓	✓	host
shared	Allocations shared between host and device	✓	✓	Can migrate between host and device

USM Memory Allocation

- Device allocation
  - Physically allocates memory on device
  - Cannot be accessed on host
- Host allocation
  - Physically allocates memory on host
  - Accessible on device! (via PCIe remote access, could be slow)
- Shared allocation
  - Not fixed physical allocation (host? device? It’s fluid)
  - Migrate between host and device automatically
  - Accessible from both host and device

Two strategies of data movement: Explicit

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q;

  std::array<int, N> host_array;
  int* device_array = malloc_device<int>(N, q);
  for (int i = 0; i < N; i++) host_array[i] = N;

  q.submit([&](handler& h) {
    // copy host_array to device_array
    h.memcpy(device_array, &host_array[0], N * sizeof(int));
  });
  q.wait();  // needed for now (we learn a better way later)

  q.submit([&](handler& h) {
    h.parallel_for(N, [=](id<1> i) { device_array[i]++; });
  });
  q.wait();  // needed for now (we learn a better way later)

  q.submit([&](handler& h) {
    // copy device_array back to host_array
    h.memcpy(&host_array[0], device_array, N * sizeof(int));
  });
  q.wait();  // needed for now (we learn a better way later)

  free(device_array, q);
  return 0;
}
```

Figure 6-6 USM explicit data movement example

Implicit Data Movement

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q;

  int* host_array = malloc_host<int>(N, q);
  int* shared_array = malloc_shared<int>(N, q);
  for (int i = 0; i < N; i++) host_array[i] = i;

  q.submit([&](handler& h) {
    h.parallel_for(N, [=](id<1> i) {
      // access shared_array and host_array on device
      shared_array[i] = host_array[i] + 1;
    });
  });
  q.wait();

  free(shared_array, q);
  free(host_array, q);
  return 0;
}
```

Figure 6-7 USM implicit data movement example

Fine-grained control

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

// Appropriate values depend on your HW
constexpr int BLOCK_SIZE = 42;
constexpr int NUM_BLOCKS = 2500;
constexpr int N = NUM_BLOCKS * BLOCK_SIZE;

int main() {
  queue q;
  int *data = malloc_shared<int>(N, q);
  int *read_only_data = malloc_shared<int>(BLOCK_SIZE, q);

  for (int i = 0; i < N; i++) {
    data[i] = -i;
  }

  // Never updated after initialization
  for (int i = 0; i < BLOCK_SIZE; i++) {
    read_only_data[i] = i;
  }

  // Mark this data as "read only" so the runtime can copy
  // it to the device instead of migrating it from the host.
  // Real values will be documented by your backend.
  int HW_SPECIFIC_ADVICE_RO = 0;
  q.mem_advise(read_only_data, BLOCK_SIZE,
               HW_SPECIFIC_ADVICE_RO);
  event e = q.prefetch(data, BLOCK_SIZE * sizeof(int));

  for (int b = 0; b < NUM_BLOCKS; b++) {
    q.parallel_for(range{BLOCK_SIZE}, e, [=](id<1> i) {
      data[b * BLOCK_SIZE + i] += read_only_data[i];
    });
    if ((b + 1) < NUM_BLOCKS) {
      // Prefetch next block
      e = q.prefetch(data + (b + 1) * BLOCK_SIZE,
                     BLOCK_SIZE * sizeof(int));
    }
  }
  q.wait();

  free(data, q);
  free(read_only_data, q);
  return 0;
}
```

Scheduling Kernels and Data Movement

- Task graph: sequencing task
- Two kind of tasks:
  - Kernel launches
  - Data movement (memcpy)
- Using dependencies to order taks
- How task graph is built at runtime
- Synchronization

Graph scheduling

- Graph node: task (kernel, memcpy)
- Graph edge: Dependencies
- Dependencies are based on data access of kernel
- Three dependencies:
  - “Read-after-Write (RAW)”
  - Write-after-Read (WAR)
  - Write-after-Write (WAR)

Command group

- A command group can contain three different things
  - Action (task)
  - Its dependencies
  - Misc host code
- Action:
  - parallel_for(), single_task()
  - memcpy(), memset(), fill(), copy(), update_host()
- Declaring dependencies:
  - In-order queues: sequential dependencies between command groups
  - Event based: explicit set dependent event in command group
  - Accessors: (only applicable to buffer/accessor pattern)

Chain dependencies:

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q{property::queue::in_order()};

  int *data = malloc_shared<int>(N, q);

  q.parallel_for(N, [=](id<1> i) { data[i] = 1; });

  q.single_task([=]() {
    for (int i = 1; i < N; i++) data[0] += data[i];
  });
  q.wait();

  assert(data[0] == N);
  return 0;
}
```

Figure 8-3 Linear dependence chain with in-order queues

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q;

  int *data = malloc_shared<int>(N, q);

  auto e = q.parallel_for(N, [=](id<1> i) { data[i] = 1; });

  q.submit([&](handler &h) {
    h.depends_on(e);
    h.single_task([=]() {
      for (int i = 1; i < N; i++) data[0] += data[i];
    });
  });
  q.wait();

  assert(data[0] == N);
  return 0;
}
```

Figure 8-4 Linear dependence chain with events

```cpp
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q;

  buffer<int> data{range{N}};

  q.submit([&](handler &h) {
    accessor a{data, h};
    h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
  });

  q.submit([&](handler &h) {
    accessor a{data, h};
    h.single_task([=]() {
      for (int i = 1; i < N; i++) a[0] += a[i];
    });
  });

  host_accessor h_a{data};
  assert(h_a[0] == N);
  return 0;
}
```

Figure 8-5 Linear dependence chain with buffers and accessors

Y-shape dependencies

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q{property::queue::in_order()};

  int *data1 = malloc_shared<int>(N, q);
  int *data2 = malloc_shared<int>(N, q);

  q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });

  q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });

  q.parallel_for(N, [=](id<1> i) { data1[i] += data2[i]; });

  q.single_task([=]() {
    for (int i = 1; i < N; i++) data1[0] += data1[i];

    data1[0] /= 3;
  });
  q.wait();

  assert(data1[0] == N);
  return 0;
}
```

Figure 8-6 "Y" pattern with in-order queues

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q;

  int *data1 = malloc_shared<int>(N, q);
  int *data2 = malloc_shared<int>(N, q);

  auto e1 =
      q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });

  auto e2 =
      q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });

  auto e3 = q.parallel_for(
      range{N}, {e1, e2},
      [=](id<1> i) { data1[i] += data2[i]; });

  q.single_task(e3, [=]() {
    for (int i = 1; i < N; i++) data1[0] += data1[i];

    data1[0] /= 3;
  });
  q.wait();

  assert(data1[0] == N);
  return 0;
}
```

Figure 8-7 "Y" pattern with events

```cpp
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q;

  buffer<int> data1{range{N}};
  buffer<int> data2{range{N}};

  q.submit([&](handler &h) {
    accessor a{data1, h};
    h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
  });

  q.submit([&](handler &h) {
    accessor b{data2, h};
    h.parallel_for(N, [=](id<1> i) { b[i] = 2; });
  });

  q.submit([&](handler &h) {
    accessor a{data1, h};
    accessor b{data2, h, read_only};
    h.parallel_for(N, [=](id<1> i) { a[i] += b[i]; });
  });

  q.submit([&](handler &h) {
    accessor a{data1, h};
    h.single_task([=]() {
      for (int i = 1; i < N; i++) a[0] += a[i];

      a[0] /= 3;
    });
  });

  host_accessor h_a{data1};
  assert(h_a[0] == N);
  return 0;
}
```

Communication and Synchronization

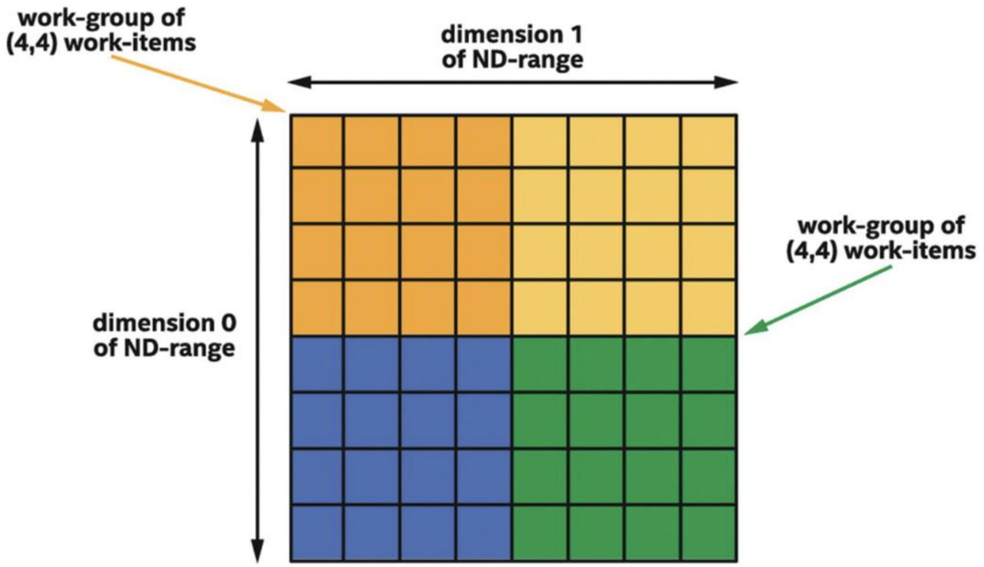

Figure 9-1 Two-dimensional ND-range of size (8, 8) divided into four work-groups of size (4,4)

- Communication:
  - Between work-items in NDRange: Global memory
  - Between work-items in a work-group: Local memory (eq to CUDA shared memory)
  - Between work-items in a sub-group (eq CUDA warp): sub-group collective functions

Local memory (shared memory in CUDA)

```
/ This is a typical global accessor.
accessor dataAcc{dataBuf, h};

// This is a 1D local accessor consisting of 16 ints:
auto localIntAcc = local_accessor<int, 1>(16, h);

// This is a 2D local accessor consisting of 4 x 4
// floats:
auto localFloatAcc =
    local_accessor<float, 2>({4, 4}, h);

h.parallel_for(
    nd_range<1>{{size}, {16}}, [=](nd_item<1> item) {
      auto index = item.get_global_id();
      auto local_index = item.get_local_id();

      // Within a kernel, a local accessor may be read
      // from and written to like any other accessor.
      localIntAcc[local_index] = dataAcc[index] + 1;
      dataAcc[index] = localIntAcc[local_index];
    });
```

Figure 9-7 Declaring and using local accessors

Sub-group Data Exchange

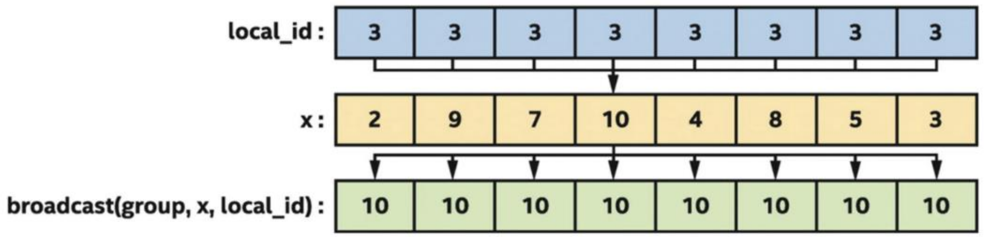

Figure 9-10 Processing by the `broadcast` function

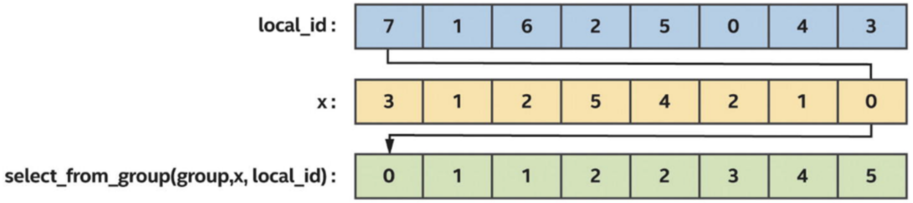

Figure 9-14 Using a generic `select_from_group` to sort values based on precomputed indices

- Group broadcast
- Group vote: any_of_group, all_of_group, none_of_group
- Shuffles: select_from_group, shift_xxx

References:

- Book: Data Parallel C++ (2023, open access at https://link.springer.com/book/10.1007/978-1-4842-9691-2)
- Tutorial: Heterogeneous programming with SYCL https://enccs.github.io/sycl-workshop/
- Official standard reference: https://github.khronos.org/SYCL_Reference/index.html
- Playground: https://sycl.tech/playground