Data Management with Buffers, Accessors, and Unified Shared Memory in SYCL

André Cerqueira

November 22, 2024

Este trabalho foi financiado por: Projeto 10110190 – EUROCC2, com apoio financeiro da FCT/MCTES através de fundos nacionais (PIDDAC).

Introduction

- SYCL offers facilities for managing memory in heterogeneous environments.
- Focus on buffer and accessor APIs, and Unified Shared Memory (USM).
- SYCL runtime manages memory, easing development and reducing bugs.

Buffers Overview

- Buffers in SYCL are abstractions for managing memory.
- They represent data that can be accessed on both host and device.
- Buffers simplify memory management by handling data transfers automatically.

Creating Buffers

- Buffers are constructed by:
  - Specifying their **size**.
  - Providing a **view** of the memory they manage.
- The `buffer` class:
  - Is **templated** over the type of the underlying memory.
  - Supports **dimensionality** (1D, 2D, or 3D).
- The size of the buffer is specified using a `range` **object**:
  - `range`s are also used to express parallelism in SYCL.
- Detailed usage of ranges in parallelism is covered in the next:

```
int N = 1024;
std::vector<int> data(N);
buffer<int, 1> buf(data.data(), range<1>(N));
```

Buffer Lifetime

- Buffers manage data movement between host and device.
- SYCL ensures data coherence through buffer destructors.
- Destructor of a buffer blocks until all commands using it are finished.
- This guarantees that all operations on the buffer are complete before destruction.

Buffer Lifetime

```
int main() {
    constexpr size_t N = 1024;

    {
        queue q;

        buffer<int, 1> buf(range<1>{N});

        q.submit([&](handler& cgh) {
            accessor acc(buf, cgh, write_only, no_init);
            cgh.parallel_for(range<1>(N), [=](id<1> i) {
                acc[i] = i[0];
            });
        }).wait();

        host_accessor acc(buf, read_only);
        std::cout << "Buffer[0]: " << acc[0] << std::endl;
    } // Buffer is destroyed here

    // After this point, 'buf' is no longer accessible.
```

Buffer Properties

- Buffers can be read-only, write-only, or read-write.
- Access modes are specified when creating accessors.
- Buffers support different data types and dimensions.
- Efficiently handle data transfers and synchronization.

Using Buffers in Kernels

- Accessors are used to access buffer data in kernels.
- Example: Creating an accessor within a command group.

```
q.submit([&](handler &cgh) {
    accessor<int, 1, access::mode::read> aA(buf, cgh);
    cgh.parallel_for<class simple_kernel>(
        range<1>(N), [=](id<1> i) {
            // Kernel code using aA
        });
});
```

Implicit Data Movement

- Buffers and accessors manage data movement implicitly.
- No explicit memory transfer code is required.
- SYCL handles the synchronization and data transfer between host and device.

Host Accessors

- Host accessors provide a way to access buffer data on the host.
- They synchronize data between device and host when created.
- Example: Using a host accessor to read buffer data on the host.

```
{
    host_accessor h_acc(buf);
    for (int i = 0; i < N; i++) {
        std::cout << h_acc[i] << " ";
    }
}
```

Explicit Data Movement

- While SYCL handles most data movement implicitly, explicit control is possible.
- Buffer objects can be explicitly copied between host and device using command groups.
- Useful for optimizing performance or handling specific synchronization requirements.

Example of Explicit Data Movement

- Example: Explicitly copying data from device to host.

```
q.submit([&](handler &cgh) {
    auto d_acc = buf.get_access<access::mode::read>(cgh);
    cgh.copy(d_acc, host_ptr);
}).wait();
```

Unified Shared Memory (USM)

- USM provides a pointer-based memory management approach.
- Allows direct access to memory from both host and device.
- Simplifies porting existing code to SYCL by using familiar pointer semantics.

USM Memory Allocation

- Three types of USM allocations:
  - **Device Allocations**: Memory physically located on the device.
  - **Host Allocations**: Memory physically located on the host, accessible by both host and device.
  - **Shared Allocations**: Memory in a unified virtual address space, accessible and migratable between host and device.

USM Allocation Examples

- Allocating device memory:

```
void* device_ptr = malloc_device(size_t numBytes, queue syclQueue);
```

- Allocating host memory:

```
void* host_ptr = malloc_host(size_t numBytes, queue syclQueue);
```

- Allocating shared memory:

```
void* shared_ptr = malloc_shared(size_t numBytes, queue syclQueue);
```

USM Typed Allocation Examples

- Typed allocation for device memory:

```
int* device_ptr = malloc_device<int>(size_t count, queue syclQueue);
```

- Typed allocation for host memory:

```
int* host_ptr = malloc_host<int>(size_t count, queue syclQueue);
```

- Typed allocation for shared memory:

```
int* shared_ptr = malloc_shared<int>(size_t count, queue syclQueue);
```

USM Data Management

- USM allows direct manipulation of memory.
- Memory initialization with `memset` and `fill`.
- Example: Initializing USM memory with `fill`.

```
queue Q;
auto x = malloc_device<double>(256, Q);
fill(x, 42.0, 256);
```

USM Data Movement

- USM supports explicit and implicit data movement.
- Explicit data movement with `memcpy` and `copy`.
- Implicit data movement for host and shared allocations.

Explicit Data Movement

- Example: Explicitly copying data from host to device.

```
queue Q;
std::vector<double> x_h(256);
auto x_d = malloc_device<double>(256, Q);

// Explicit data copy
Q.submit([&](handler& cgh) {
    cgh.memcpy(x_d, x_h.data(), 256 * sizeof(double));
}).wait();
```

Implicit Data Movement

- Host and shared allocations benefit from implicit data movement.
- Example: Accessing host and shared memory in a kernel.

```
constexpr auto N = 256;
queue Q;

auto x_h = malloc_host<double>(N, Q);
auto x_s = malloc_shared<double>(N, Q);

for (auto i = 0; i < N; ++i) {
    x_h[i] = static_cast<double>(i);
}

Q.submit([&](handler& cgh) {
    cgh.parallel_for(range<1>(N), [=](id<1> i) {
        x_s[i] = x_h[i] + 1.0;
    });
}).wait();
```

Buffer-Accessor Model vs Unified Shared Memory

- The choice between buffer-accessor model and USM depends on the level of control needed over data transfers.
- **Buffer-Accessor Model:** Managed by the SYCL runtime which automates data transfers and minimizes programming errors.
- **Unified Shared Memory (USM):** Offers direct control over memory, suitable for porting existing codes using pointers, providing a familiar programming approach.
- **Considerations:**
  - Comfort with runtime managing data movement.
  - Compatibility with existing codebases.
- Current SYCL standards do not support interoperability between buffers and USM, which may lead to performance issues.
- Extensions like hipSYCL provide buffer-USM interoperability as an additional feature.

Summary

- Buffers and accessors simplify memory management in SYCL.
- USM provides a pointer-based approach for direct memory access.
- Both supports both explicit and implicit data movement.
- Choosing the right memory model depends on the use case and programmers preference.