Buffers and Accessors in Depth

Find out how to use Buffers and Accessors in Depth
Buffers and Accessors

• Agenda
  • Buffers and Accessors In Depth
  • Buffer Creation methods
  • Buffer Properties: use_host_ptr, set_final_data, set_write_back
  • Sub buffers
  • Accessors and its Properties

• Hands On
  • Buffer Properties
  • Sub Buffers
Learning Objectives

Utilize Buffers and Accessors to apply control over data movement.

Determine appropriate usage of the following buffer properties: `use_host_ptr`, `set_final_data` and `set_write_data`.

Split buffer into two sub buffers create kernels concurrently.

Explain host accessors and the different use cases of host accessors.
Buffers encapsulate data shared between host and device.

Accessors provide access to data stored in buffers and create data dependences in the graph.

Unified Shared Memory (USM) provides an alternative pointer-based mechanism for managing memory.

```cpp
queue q;
std::vector<int> v(N, 10);
{
    buffer buf(v);
    q.submit([&](handler& h) {
        accessor a(buf, h, write_only);
        h.parallel_for(N, [=](auto i) { a[i] = i; });
    });
}
for (int i = 0; i < N; i++) std::cout << v[i] << " ";
```
## Accessor Modes

<table>
<thead>
<tr>
<th>Access Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_only</td>
<td>Read only Access</td>
</tr>
<tr>
<td>write_only</td>
<td>Write-only access. Previous contents not discarded</td>
</tr>
<tr>
<td>read_write</td>
<td>Read and Write access</td>
</tr>
</tbody>
</table>
DPC++ Code Anatomy

```cpp
void dpcpp_code(int* a, int* b, int* c) {
    // Setting up a 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 group for (asynchronous) execution

**Step 4:** create accessors
describing how buffer is used on the device

**Step 5:** specify kernel function and launch parameters (e.g. group size)

**Step 6:** specify code to run on the device

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
Buffer Creation

Buffer Class: Template class with three arguments

- Type of the Object
- Dimensionality of the Buffer
- Optional C++ Allocator

The choice of buffer creation depends on how the buffer needs to be used as well as programmer's coding preferences.
Let's look at a simple DPC++ code example and see different ways of buffer creation:

```cpp
{
    // Create a buffer of ints from an input iterator
    std::vector<int> myVec;
    buffer b1{myVec};
    buffer b2{myVec.begin(), myVec.end()};

    // Create a buffer of ints from std::array
    std::array<int, 42> my_data;
    buffer b3{my_data};

    // Create a buffer of 4 doubles and initialize it from a host pointer
    double myDoubles[4] = {1.1, 2.2, 3.3, 4.4};
    buffer b4{myDoubles, range{4}};
}
```
Use _host_ptr requires the buffer to not allocate any memory on the host.

Buffer should use the memory pointed to by a host pointer that is passed to the constructor.

This option can be useful when the program wants full control over all host memory allocations.

```cpp
int main() {
    queue q;
    int myInts[42];
    // create a buffer of 42 ints, initialize
    // with a host pointer,
    // and add the use_host_pointer property
    buffer b1(myInts, range(42), property::use_host_ptr{});
}
```
Buffer Properties: use_host_ptr

This property requires the buffer to not allocate any memory on the host, instead, the buffer should use the memory pointed to by a host pointer that is passed to the constructor.

Initialize vector a and b

Use property::use_host_ptr()

Submit the work

```cpp
#include <dpcppbuffer.h>

int main() {
    queue q;
    std::vector<float> a(N, 10.0f);
    std::vector<float> b(N, 20.0f);

    {  
        buffer buf_a(a, {property::buffer::use_host_ptr()});
        buffer buf_b(b, {property::buffer::use_host_ptr()});

        q.submit([&](handler& h) {
            //create Accessors for a and b
            accessor A(buf_a, h);
            accessor B(buf_b, h, read_only);

            h.parallel_for(R, [=](auto i) {
                A[i] += B[1];
            });
        });
    }

    return 0;
}
```
The `set_final_data` method of a buffer is the way to update host memory however the buffer was created.

When the buffer is destroyed, data will be written to the host using the supplied location.

Call the set_final_data to the created shared ptr where the values will be written back when the buffer gets destructed.

```cpp
{
    queue q;
    buffer my_buffer(my_data);
    my_buffer.set_final_data(nullptr);
    q.submit([&](handler &h) {
        accessor my_accessor(my_buffer, h);
        h.parallel_for(N, [=](id<1> i) {
            my_accessor[i]*=2;
        });
    });
}
```
Buffer: set_write_back

We can control whether writeback occurs from the device to the host by calling the `set_write_back` method.

Call the `set_write_back` method to control the data to be written back to the host from the device.

Setting it to false will not update the host with the updated values.

```cpp
buffer my_buffer(my_data);
my_buffer.set_write_back(false);
q.submit([&](handler &h) {
    accessor my_accessor(my_buffer, h);
    h.parallel_for(N, [=](id<1> i) {
        my_accessor[i] *= 2;
    });
});
```
A sub-buffer requires three things, a reference to a parent buffer, a base index, and the range of the sub-buffer.

The main advantage of using the sub-buffers is different kernels can operate on different sub buffers concurrently.

Sub Buffer for one dimensional buffer

Sub buffer for a 2-dimensional buffer
int main() {
    const int N = 64;  const int num1 = 2;  const int num2 = 3;
    int data[N];
    for (int i = 0; i < N; i++) data[i] = i;  for (int i = 0; i < N; i++) std::cout << data[i] << " ";
    buffer B(data, range(N));

    buffer<int> B1(B, 0, range{ N / 2 });
    buffer<int> B2(B, 32, range{ N / 2 });

    queue q1;
    q1.submit([&](handler& h) {
        accessor a1(B1, h);
        h.parallel_for(N/2, [=](auto i) { a1[i] *= num1; });
    });

    queue q2;
    q2.submit([&](handler& h) {
        accessor a2(B2, h);
        h.parallel_for(N/2, [=](auto i) { a2[i] *= num2; });
    });

    host_accessor b1(B1, read_only);
    host_accessor b2(B2, read_only);
    return 0;
}
Asynchronous Execution

Host

Host code execution

Enqueues kernel to graph, and keeps going

Graph

Graph executes asynchronously to host program

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
    std::vector<int> data(N);
    {        // Enqueues
        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];
    }
```
int main() {
    auto R = range<1>{ num }; 
    buffer<int> A{ R }, B{ R }; 
    queue q;

    q.submit([&](handler& h) {
        accessor out(A, h, write_only);
        h.parallel_for(R, [=](id<1> i) {
            out[i] = i; }); });

    q.submit([&](handler& h) {
        accessor out(A, h, write_only);
        h.parallel_for(R, [=](id<1> i) {
            out[i] = i; }); });

    q.submit([&](handler& h) {
        accessor out(B, h, write_only);
        h.parallel_for(R, [=](id<1> i) {
            out[i] = i; }); });

    q.submit([&](handler& h) {
        accessor in(A, h, read_only);
        accessor inout(B, h);
        h.parallel_for(R, [=](id<1> i) {
            inout[i] *= in[i]; }); });
}

Kernel 1
Kernel 2
Kernel 3
Kernel 4

Automatic data and control dependence resolution!
Synchronization – Host Accessors

Buffer takes ownership of the data stored in vector.

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 via this host accessor.
Hands-on Coding on Intel DevCloud

Buffers and Accessors
Summary

In this module you learned:
Buffers and Accessors in Depth
Buffers properties and when to use _host_ptr, set_final_data and set_write_data
Sub buffers and how to create and use Sub buffers
How to create Accessors, host accessors and initialize buffer data using host accessors