Unified Shared Memory

March 2020

What is Unified Shared Memory

Unified Shared Memory (USM) is:

- A pointer-based approach for data management in DPC++
- Complements buffers, not a replacement

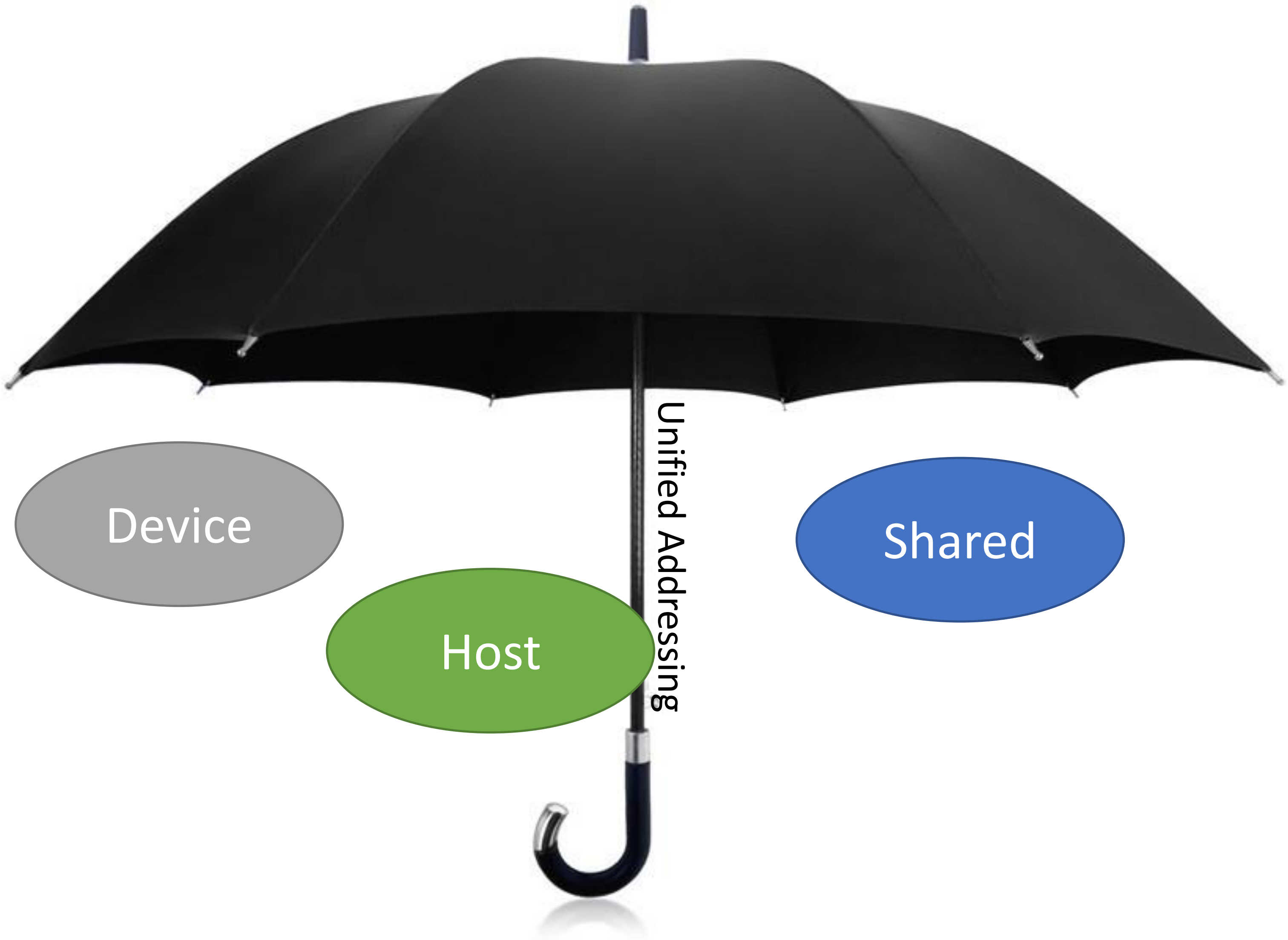
Why USM?

- Simplify porting existing C++ codes to DPC++
- Give desired level of control over data movement

Why is it called USM?

- OpenMP calls it that – don't reinvent the wheel

USM in a Picture



Allocation Types

Three types of allocations:

- Device
  - Accessible on device, not on host
  - “Give me a pointer to my GPU’s DRAM”
- Host
  - Accessible on host and device
  - Allocated in pinned Host memory, does not migrate to device DRAM
- Shared
  - Accessible on host and device
  - Can migrate between host and device DRAM

Buffer Example

Declare C++ Arrays

```
  auto A = (int *) malloc(N * sizeof(int));
  auto B = (int *) malloc(N * sizeof(int));
  auto C = (int *) malloc(N * sizeof(int));

  for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
  }

{
  buffer<int, 1> Ab(A, range<1>{N});
  buffer<int, 1> Bb(B, range<1>{N});
  buffer<int, 1> Cb(C, range<1>{N});

  q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
      aC[i] = aA[i] + aB[i];
    });
  });
  q.wait();
} // A,B,C updated
```

oneAPI

Declare C++ Arrays

Initialize C++ Arrays

```
  auto A = (int *) malloc(N * sizeof(int));
  auto B = (int *) malloc(N * sizeof(int));
  auto C = (int *) malloc(N * sizeof(int));

  for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
  }

{
  buffer<int, 1> Ab(A, range<1>{N});
  buffer<int, 1> Bb(B, range<1>{N});
  buffer<int, 1> Cb(C, range<1>{N});

  q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
      aC[i] = aA[i] + aB[i];
    });
  });
  q.wait();
} // A,B,C updated
```

Declare C++ Arrays

Initialize C++ Arrays

Declare Buffers

```
  auto A = (int *) malloc(N * sizeof(int));
  auto B = (int *) malloc(N * sizeof(int));
  auto C = (int *) malloc(N * sizeof(int));

  for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
  }

{
  buffer<int, 1> Ab(A, range<1>{N});
  buffer<int, 1> Bb(B, range<1>{N});
  buffer<int, 1> Cb(C, range<1>{N});

  q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
      aC[i] = aA[i] + aB[i];
    });
  });
  q.wait();
} // A,B,C updated
```

oneAPI

Declare C++ Arrays

Initialize C++ Arrays

Declare Buffers

Declare Accessors

```
  auto A = (int *) malloc(N * sizeof(int));
  auto B = (int *) malloc(N * sizeof(int));
  auto C = (int *) malloc(N * sizeof(int));

  for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
  }

{
  buffer<int, 1> Ab(A, range<1>{N});
  buffer<int, 1> Bb(B, range<1>{N});
  buffer<int, 1> Cb(C, range<1>{N});

  q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
      aC[i] = aA[i] + aB[i];
    });
  });
  q.wait();
} // A,B,C updated
```

Declare C++ Arrays

Initialize C++ Arrays

Declare Buffers

Declare Accessors

Use Accessors in Kernel

```
  auto A = (int *) malloc(N * sizeof(int));
  auto B = (int *) malloc(N * sizeof(int));
  auto C = (int *) malloc(N * sizeof(int));

  for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
  }

{
  buffer<int, 1> Ab(A, range<1>{N});
  buffer<int, 1> Bb(B, range<1>{N});
  buffer<int, 1> Cb(C, range<1>{N});

  q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
      aC[i] = aA[i] + aB[i];
    });
  });
  q.wait();
} // A,B,C updated
```

Declare C++ Arrays

Initialize C++ Arrays

Declare Buffers

Declare Accessors

Use Accessors in Kernel

C++ Arrays Updated

```
  auto A = (int *) malloc(N * sizeof(int));
  auto B = (int *) malloc(N * sizeof(int));
  auto C = (int *) malloc(N * sizeof(int));

  for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
  }

{
  buffer<int, 1> Ab(A, range<1>{N});
  buffer<int, 1> Bb(B, range<1>{N});
  buffer<int, 1> Cb(C, range<1>{N});

  q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
      aC[i] = aA[i] + aB[i];
    });
  });
  q.wait();
} // A,B,C updated
```

USM Example

Declare USM Arrays

```
int *A = malloc_shared<int>(N, q);
int *B = malloc_shared<int>(N, q);
int *C = malloc_shared<int>(N, q);

for (int i = 0; i < N; i++) {
  A[i] = i; B[i] = 2*i;
}

q.submit([&] (handler& h) {
  auto R = range<1>{N};
  h.parallel_for(R, [=] (id<1> ID) {
    auto i = ID[0];
    C[i] = A[i] + B[i];
  });
});
q.wait();
// A,B,C updated and ready to use
```

Declare USM Arrays

Initialize USM Arrays

```
int *A = malloc_shared<int>(N, q);
int *B = malloc_shared<int>(N, q);
int *C = malloc_shared<int>(N, q);

for (int i = 0; i < N; i++) {
  A[i] = i; B[i] = 2*i;
}

q.submit([&] (handler& h) {
  auto R = range<1>{N};
  h.parallel_for(R, [=] (id<1> ID) {
    auto i = ID[0];
    C[i] = A[i] + B[i];
  });
});
q.wait();
// A,B,C updated and ready to use
```

Declare USM Arrays

Initialize USM Arrays

Read/Write USM Arrays

```
int *A = malloc_shared<int>(N, q);
int *B = malloc_shared<int>(N, q);
int *C = malloc_shared<int>(N, q);

for (int i = 0; i < N; i++) {
  A[i] = i; B[i] = 2*i;
}

q.submit([&] (handler& h) {
  auto R = range<1>{N};
  h.parallel_for(R, [=] (id<1> ID) {
    auto i = ID[0];
    C[i] = A[i] + B[i];
  });
});
q.wait();
// A,B,C updated and ready to use
```

Declare USM Arrays

Initialize USM Arrays

Read/Write USM Arrays

USM Arrays Updated

```
int *A = malloc_shared<int>(N, q);
int *B = malloc_shared<int>(N, q);
int *C = malloc_shared<int>(N, q);

for (int i = 0; i < N; i++) {
  A[i] = i; B[i] = 2*i;
}

q.submit([&] (handler& h) {
  auto R = range<1>{N};
  h.parallel_for(R, [=] (id<1> ID) {
    auto i = ID[0];
    C[i] = A[i] + B[i];
  });
});
q.wait();
// A,B,C updated and ready to use
```

Allocation Routines

- Many flavors defined:
  - void* returning malloc_shared, malloc_device, malloc_host
  - T returning malloc_shared<T>, malloc_device<T>, malloc_host<T>
  - void* returning malloc(…, kind)
  - T returning malloc<T>(…, kind)
  - C++ allocator (useful with containers)
- Aligned versions exist as well

- Allocations must take a context (and sometimes a device)
  - Can also just pass a queue and use its context/device
  - Device and shared allocations need to allocate against a specific device
  - Allocations are not guaranteed to be usable across different contexts
- Free takes a context or queue.

Explicit Data Movement

Device allocations cannot be directly accessed on the host

- Programmers must manually copy data between host and device
- memcpy(…) on queue or handler classes
  - Currently async by default
- memset(…) currently, fill(…) in future
  - byte vs T

Trades off extra complexity for full control over data movement

Host allocations have no data movement.

Implicit Data Movement

Shared allocations do not require programmers to manually transfer data between host and device.

- Data movement will be automatically handled by some combination of HW, drivers, and low-level runtimes.

Trades off control (and performance) for simplicity and productivity

However:

- Programmers can give automatic systems additional information to change their behavior
- Prefetch, Memadvise, etc.

Shared Allocations: Restricted or not?

USM defines two capability levels for shared allocations:

- Restricted
- Concurrent

Restricted:

- Shared allocs are basically device allocs visible on host
- Limited to device memory, but still migrates between host and device
- Should not concurrently access allocations on host and device

Concurrent:

- Shared allocs are just allocs in the “shared” address space
- Migrates freely and different parts of the allocation can be concurrently accessed on host and device (Think page granularity)
- Not limited to device memory

Task Scheduling

Explicit Scheduling

- Submitting a kernel returns an Event
- Wait on Events to order tasks

```
auto E = q.submit([&] (handler& h) {
  auto R = range<1>{N};
  h.parallel_for(R, [=] (id<1> ID) {
    auto i = ID[0];
    C[i] = A[i] + B[i];
  });
});
E.wait();
```

DPC++ Graph Scheduling

- Build Task Graphs from Events

```
auto R = range<1>{N};

auto E = q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});

q.submit([&] (handler& h) {
  h.depends_on(E);
  h.parallel_for(R, [=] (id<1> ID) {…});
});
```

Device Queries

- Are device allocations supported?
- Are host allocations supported?
- Are shared allocations supported?
- Are shared allocations restricted?
- Is the system allocator supported?

Pointer Queries

Two flavors:

- Given a ptr, what type of USM allocation is it, if any?
- Given a ptr, what device was it allocated against, if it’s a shared/device allocation?

History: Why not OpenCL SVM?

OpenCL defined 3 flavors of Shared Virtual Memory (SVM):

- Coarse-grain buffer – Too hard to use
- Fine-grain buffer – Too hard to use
- System – Too hard to implement

We have also proposed USM for OpenCL

Summary

USM is for pointer-based data management in DPC++

- Multiple types of allocations
- Explicit and Implicit data movement
- Event-based task scheduling

Enables other API simplifications to reduce DPC++ verbosity

- q.parallel_for, etc.