Software

DATA PARALLEL C++

Extending SYCL Through Extensions for Productivity and Performance

James Brodman, Michael Kinsner, Ben Ashbaugh, Jeff Hammond, Alexey Bader, John Pennycook, Jason Sewall, Roland Schulz

SYCLcon 2020

Outline

- Intro
- DPC++ Extensions
  - Unified Shared Memory
  - Unnamed Kernel Lambda
  - In-order Queues
  - Sub-groups
  - Reductions
  - Simplifications
- Summary

DPC++ Extends SYCL* 1.2.1

DPC++ = modern C++ and SYCL and Extensions

Enhance Productivity

- Simple things should be simple to express
- Reduce verbosity and programmer burden

Enhance Performance

- Give programmers control over program execution
- Enable hardware-specific features

Unified Shared Memory (USM)

SYCL 1.2.1 provides the Buffer abstraction for memory

- Very powerful, elegantly expresses data dependences

However…

- Replacing all pointers and arrays with buffers in a C++ program can be a burden to programmers

USM provides a pointer-based alternative in DPC++

- Simplifies porting to an accelerator
- Gives programmers the desired level of control

What is USM?

Allocation Types

Type	Description
device	Allocations in device memory
host	Allocations in host memory accessible by the device
shared	Allocations accessible by both host and device that may migrate between them

APIs

```
void* sycl::malloc_device(size_t size, …)
void* sycl::malloc_host(size_t size, …)
void* sycl::malloc_shared(size_t size, …)
T* sycl::malloc_shared<T>(size_t count, …)
…

sycl::free(void *ptr, …)

void queue::memcpy(void* dest,
          const void* src, size_t count)
```

https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/USM/USM.adoc

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
```

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
```

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

Task Scheduling with USM

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) {…});
});
```

Why Unified Shared Memory?

USM makes it easier to get applications running on an accelerator

- Easier integration into C++ apps
- Shared allocations handle data movement for the programmer
  - Faster time to working program, fewer errors

Check out the IWOCL presentation from Michal Mrozek on USM in OpenCL:

- “Taking memory management to the next level – Unified Shared Memory in action”
- Learn how USM differs from OpenCL SVM

Unnamed Kernel Lambda

SYCL 1.2.1 requires all kernels to have a unique name:

- Functor class type
- Template typename for Lambdas

DPC++ removes this requirement for Lambdas

- Must use DPC++ compiler for both host and device code
- Enabled via compiler switch or dpcpp executable

```
q.submit([&] (handler& h) {
  auto R = range{N};

  h.parallel_for<class VAdd>(
    R, [=](id<1> ID) {
      C[ID] = A[ID] + B[iD];
  });
});
```

Unnamed Kernel Lambda

SYCL 1.2.1 requires all kernels to have a unique name:

- Functor class type
- Template typename for Lambdas

DPC++ removes this requirement for Lambdas

- Must use DPC++ compiler for both host and device code
- Enabled via compiler switch or dpcpp executable

```
q.submit([&] (handler& h) {
  auto R = range{N};

  h.parallel_for(
    R, [=](id<1> ID) {
      C[ID] = A[ID] + B[ID];
  });
});
```

In-order Queue

DPC++ Queues are out-of-order

- Allows expressing complex DAGs

Linear task chains are common

- DAGs are overkill here and add verbosity

Simple things should be simple to express

- In-order semantics express the linear task pattern easily

```
// Without in-order Queues
queue q;
auto R = range{N};

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

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

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

In-order Queue

DPC++ Queues are out-of-order

- Allows expressing complex DAGs

Linear task chains are common

- DAGs are overkill here and add verbosity

Simple things should be simple to express

- In-order semantics express the linear task pattern easily

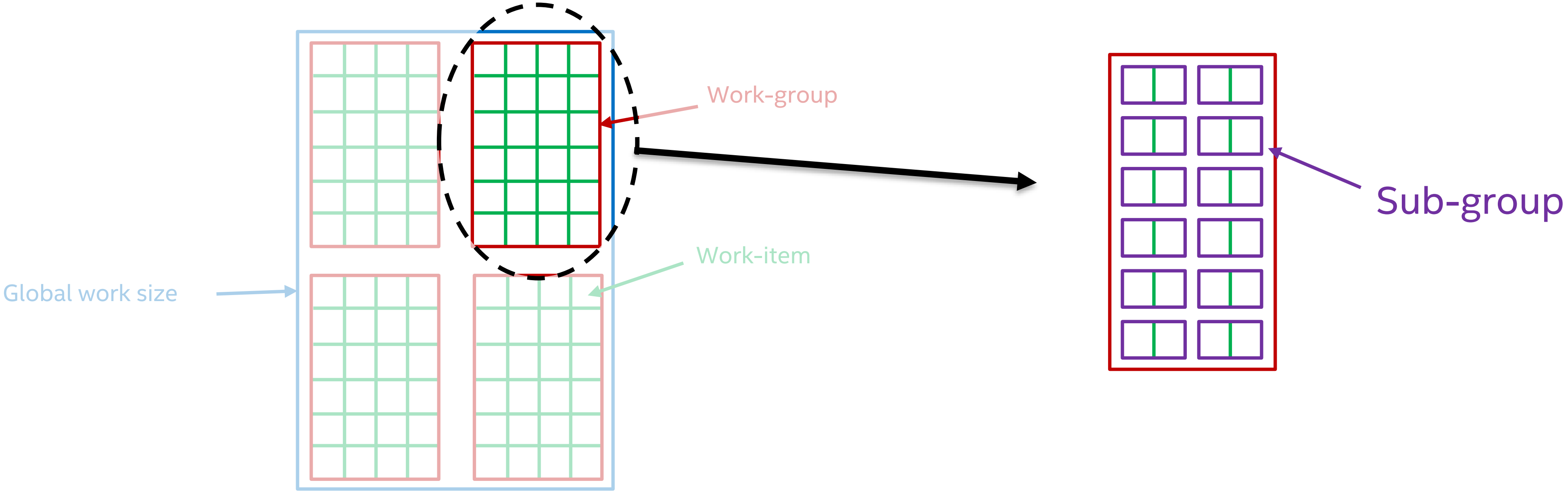
```
// With in-order Queues
queue q{property::queue::in_order()};
auto R = range{N};

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

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

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

Sub-groups in DPC++

Implementation-defined subset of work-items in a work-group

Work-items in a sub-group execute “together”

- e.g. SIMD instructions, NVIDIA* warps, AMD* wavefronts, fibers/coroutines



Example: Sub-groups in DPC++

```
q.parallel_for(R, [=](nd_item<1> it)
 [[intel::reqd_sub_group_size(8)]] /* Request specific sub-group size */ {

 // Get handle to the sub-group this item belongs to
 sub_group sg = it.get_sub_group();
 ...
 // Optimized code when all work-items in the sub-group take the same branch
 bool condition = ...;
 if (all_of(sg, condition)) {
  ...
  int sum = reduce(sg, x, plus<>()); // Accumulate partial results from all work-items
  ...
 }
 // Otherwise, fall back to less efficient path
 else {
  ...
 }
});
```

Reductions in DPC++

Reduction kernels combining multiple values to produce a single output appear frequently across applications from multiple domains

Reductions have simple semantics…

- The input values can be combined in any order
- Only the final result is meaningful

… but implementing high-performance reductions is non-trivial:

- How many input values are there?
- How much parallelism is there?
- What features does the hardware have? (e.g. atomic instructions, scratchpads)

DPC++ shifts implementation burden from developers to compiler/runtime

Example: Reductions in DPC++

```
// Compute dot-product by reducing all values using standard plus functor
q.parallel_for(R, reduction(sum, 0, plus<float>()), [=](nd_item<1> it, auto& partial_sum) {
  int i = it.get_global_id(0);
  partial_sum += (a[i] * b[i]);
}).wait();
```

1. A reduction operation is described by:
   - A reduction variable (e.g. `sum`)
   - An (optional) identity variable (e.g. `0`)
   - A combination operation (e.g. `plus<float>()`)

2. The kernel lambda accepts a reference to a reducer per work-item
   - Restricts interface to prevent updates incompatible with the combination operation

3. Implementation combines reducers and updates reduction variable before kernel completes

Language and API Simplifications

Simple things should be simple to express!

- Class Template Argument Deduction (CTAD)
  - `buffer<int, 2> b(ptr, range<2>(5, 5))` → `buffer b(ptr, range(5, 5))`, etc.
- Queue shortcuts
  - Useful when combined with USM
  - `q.submit([&] (handler& h) { h.parallel_for(…); }` → `q.parallel_for(…);`
- More planned

Summary

DPC++ builds upon the strong foundation of SYCL

- Builds upon SYCL 1.2.1 with new features that:
  - Make simple things simple to express
  - Provide access to hardware-specific features
- We hope many of these extensions appear in a future version of SYCL

New features being developed through a community project

- https://github.com/intel/llvm
- Specifications for the extensions found there or at https://www.oneapi.com/

intel®

Software