Data Parallel C++ Essentials

Data Parallel C++ - SYCL2020 Features

Find out what's new in Data Parallel C++ Language

intel®

DPC++ New Features

- Agenda
  - DPC++ Language Simplification
  - Unified Shared Memory (USM)
  - Sub-Groups
  - Simplified Reduction
- Hands On
  - USM and solving data dependency
  - Sub-group collectives and shuffle operations
  - Simplification with DPC++ Reduction extension

Learning Objectives

Use new DPC++ features like Unified Shared Memory to simplify heterogeneous programming

Understand advantages of using Sub-groups in DPC++

Simplify reductions in heterogenous programming

What is Data Parallel C++?

Data Parallel C++

= C++ and SYCL* standard and extensions

Based on modern C++

- C++ productivity benefits and familiar constructs

Standards-based, cross-architecture

- Incorporates the SYCL standard for data parallelism and heterogeneous programming

DPC++ Extends SYCL* standard

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

DPC++: Fast-moving open collaboration feeding into the SYCL* standard

- Open source implementation with goal of upstream LLVM
- DPC++ extensions aim to become core SYCL*, or Khronos* extensions

DPC++ = C++ + SYCL* + Extensions

Some of DPC++ Extensions:

- Unified Shared Memory (USM)
- Sub-Groups
- Simplified Reduction

Main goals of DPC++ Extensions are to simplify programming and achieve performance by exposing hardware features.

DPC++ Syntax vs SYCL 2020 Syntax

- The syntax of a DPC++ extension to SYCL 1.2.1 and the syntax adopted by SYCL 2020 may differ
- Hands-on materials use DPC++ extension syntax for compatibility with the current DPC++ compiler
- Support for some SYCL 2020 features is already available in the open-source compiler

Language Simplification

DPC++ significantly simplifies SYCL* language by reducing verbosity

DPC++ Language Simplification

Code snippet below shows how SYCL* code can be simplified in DPC++

SYCL

```
buffer<int, 1> buf(data.data(), data.size());
q.submit([&] (handler &h){
  auto A = buf.get_access<access::mode::read_write>(h);
  h.parallel_for<class kernel>(range<1>(N), [=](id<1> i){ A[i] += 1; });
});
```

Lambda name no longer required

Buffer Simplification

Accessor simplification

parallel_for simplification

DPC++

```
buffer buf(data);
q.submit([&] (handler &h){
  auto A = accessor(buf, h);
  h.parallel_for(N, [=](auto i){ A[i] += 1; });
});
```

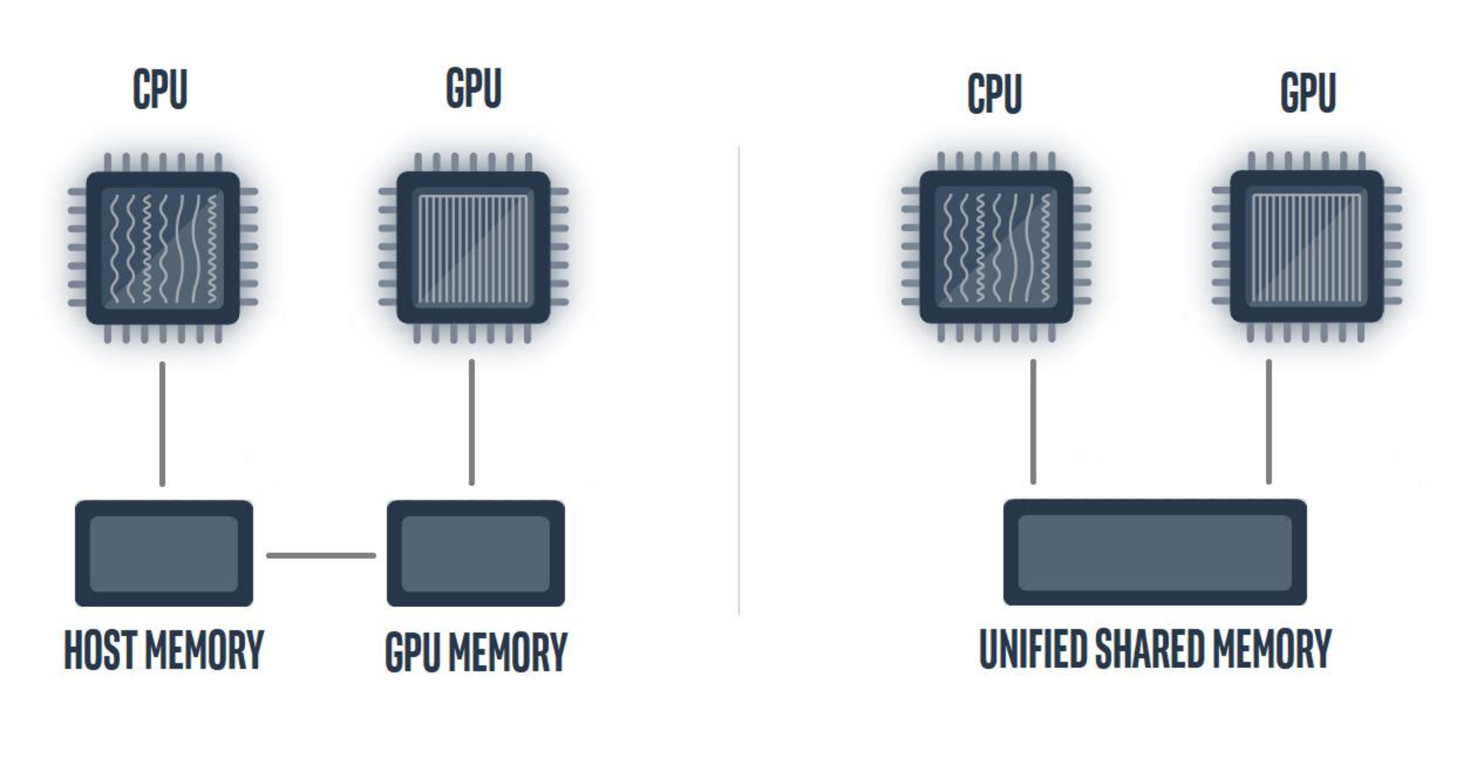
← *Simple and Less Verbose*

Unified Shared Memory (USM)

Unified Shared Memory is pointer-based approach to memory model for heterogeneous programming

Developer View of USM

Developers can reference same memory object in host and device code with Unified Shared Memory



Unified Shared Memory

Unified Shared Memory can be setup as follows:

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

You can also use a more familiar C++/C style malloc:

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

Unified Shared Memory

Unified Shared Memory enables accessing memory on the host and device with same pointer reference

Setup Unified Shared Memory → `auto data = malloc_shared<int>(N, q);`

Host can initialize → `for(int i=0;i<N;i++) data[i] = 10;`

Device can modify → `data[i] += 1;`

Host has output → `for(int i=0;i<N;i++) std::cout << data[i] << " ";`

```
queue q;
auto data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.parallel_for(N, [=](auto i){
        data[i] += 1;
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

SYCL Buffers Method

Same code but using SYCL buffer memory model instead of USM – requires defining buffers and accessors and synchronize as required

Host memory setup

Host can initialize

Create buffer

Create accessor

Device can modify

Buffer destruction

Host has output

```
queue q;
int *data =  static_cast(int*)(malloc(N * sizeof(int), q));
for(int i=0;i<N;i++) data[i] = 10;
{
    buffer<int, 1> buf(data, range<1>(N));
    q.submit([&] (handler &h){
        auto A = buf.get_access<access::mode::read_write>(h);
        h.parallel_for(range<1>(N), [=](id<1> i){
            A[i] += 1;
        });
    });
}
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data);
```

WHY Unified Shared Memory?

The SYCL* standard provides a Buffer memory abstraction

- Powerful and 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
- Complementary to buffers

Unified Shared Memory (USM)

There are three ways to create USM allocations:

Type	Description	Accessible on Host?	Accessible on Device?
`sycl::malloc_device`	Allocations in device memory. Programmer must explicitly transfer data between host and device.	No	Yes
`sycl::malloc_host`	Allocations in host memory. Kernels can access these allocations directly.	Yes	Yes
`sycl::malloc_shared`	Allocations can migrate between host and device memory. Different implementations may provide different guarantees regarding whether allocations can be accessed by host and device concurrently.	Yes	Yes

USM – Explicit Data Transfer

Gives developer full control of moving memory between host and device

malloc_device() will allocate memory on device, Host will not have access

Copy memory explicitly from host to device using q.memcpy()

Make any data modification on device

Copy the memory explicitly from device to host using q.memcpy()

```
queue q;

int data[N];
for (int i = 0; i < N; i++) data[i] = 10;

int *data_device = malloc_device<int>(N, q);

q.memcpy(data_device, data, sizeof(int) * N).wait();

q.parallel_for(N, [=](auto i) { data_device[i] += 1; }).wait();

q.memcpy(data, data_device, sizeof(int) * N).wait();

for (int i = 0; i < N; i++) std::cout << data[i] << std::endl;
free(data_device, q);
```

USM – Implicit Data Transfer

Memory movement between host and device is done implicitly

malloc_shared() will allocate memory that can move between host and device. Host and device will have access

Make any data modification on device

Host has access to the device modified memory

```
queue q;

int *data = malloc_shared<int>(N, q);
for (int i = 0; i < N; i++) data[i] = 10;

q.parallel_for(N, [=](auto i) { data[i] += 1; }).wait();

for (int i = 0; i < N; i++) std::cout << data[i] << std::endl;
free(data, q);
```

Hands-on Coding on Intel DevCloud

USM Implicit and Explicit Data Movement

Unified Shared Memory – When to use it?

SYCL* Buffers are powerful and elegant

- Use if the abstraction applies cleanly in your application, and/or buffers aren't disruptive to your development

USM provides a familiar pointer-based C++ interface

- Useful when porting C++ code to DPC++, by minimizing changes
- Use shared allocations when porting code, to get functional quickly
- Note that shared allocation is not intended to provide peak performance out of box
- Use explicit USM allocations when controlled data movement is needed

USM – Data Dependency in tasks

- When using unified shared memory in multiple kernel tasks, dependences between operations must be specified using events.

- Programmers may either explicitly wait on event objects or use the depends_on method inside a command group to specify a list of events that must complete before a task may begin.

USM – Data Dependency in tasks

Explicit wait() used to ensure data dependency is maintained

*Note that wait() will block execution on host


```
queue q;
int *data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;

q.parallel_for(N, [=](auto i){
    data[i] += 2;
}).wait();

q.parallel_for(N, [=](auto i){
    data[i] += 3;
}).wait();

q.parallel_for(N, [=](auto i){
    data[i] += 5;
}).wait();

for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

USM – Data Dependency in tasks

Use in_order queue property for the queue

* Execution will not overlap even if the tasks have no dependency


```
queue q{property::queue::in_order()};
int *data = malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;

q.parallel_for(N, [=](auto i){
    data[i] += 2;
});

q.parallel_for(N, [=](auto i){
    data[i] += 3;
});

q.parallel_for(N, [=](auto i){
    data[i] += 5;
}).wait();

for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

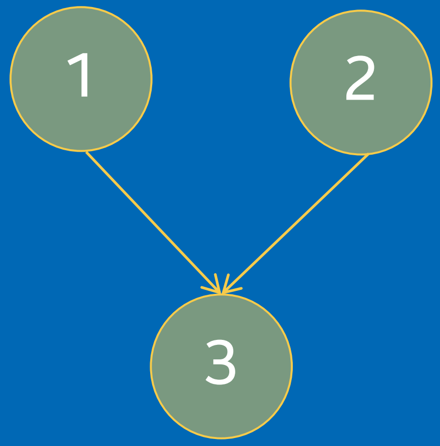
USM – Data Dependency in tasks

Use depends_on() method to let command group handler know that specified event should be complete before specified task can execute.


```
queue q;
int *data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
auto e1 = q.submit([&] (handler &h){
    h.parallel_for(N, [=](auto i){
        data[i] += 2;
    });
});
auto e2 = q.submit([&] (handler &h){
    h.depends_on(e1);
    h.parallel_for(N, [=](auto i){
        data[i] += 3;
    });
});
q.submit([&] (handler &h){
    h.depends_on(e2);
    h.parallel_for(N, [=](auto i){
        data[i] += 5;
    });
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

USM – Data Dependency in tasks

Use depends_on() is also useful to specify dependency for certains and let other tasks overlap if there is no dependency.




```
queue q;
int *data1 =  malloc_shared<int>(N, q);
int *data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.parallel_for(N, [=](auto i){
    data1[i] += 2;
});
auto e2 = q.parallel_for(N, [=](auto i){
    data2[i] += 3;
});
q.submit([&] (handler &h){
    h.depends_on({e1,e2});
    h.parallel_for(N, [=](auto i){
        data1[i] += data2[i];
    });
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data1, q); free(data2, q);
```

Hands-on Coding on Intel DevCloud

Handling Data Dependency when using USM

Unified Shared Memory

- Summary
  - What is Unified Shared Memory (USM)?
  - Implicit and Explicit data movement between host and device
  - Handling data dependency in multiple kernel tasks using wait event, depends_on method and in_order queue property

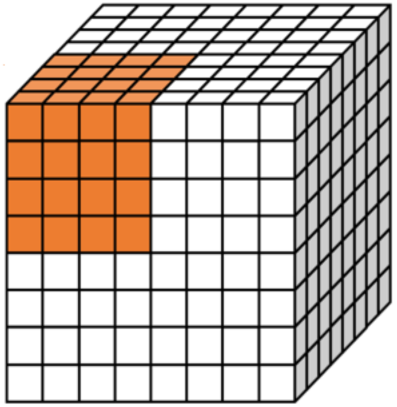
Sub Groups

Sub-groups are subset of the work-items that are executed simultaneously or with additional scheduling guarantees.

Leveraging sub-groups will help to map execution to low-level hardware and may help in achieving higher performance.

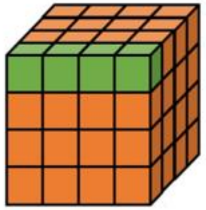
How it maps to Hardware (INTEL GEN11 GRAPHICS)

All work-items in a **work-group** are scheduled on one subslice, which has its own local memory.



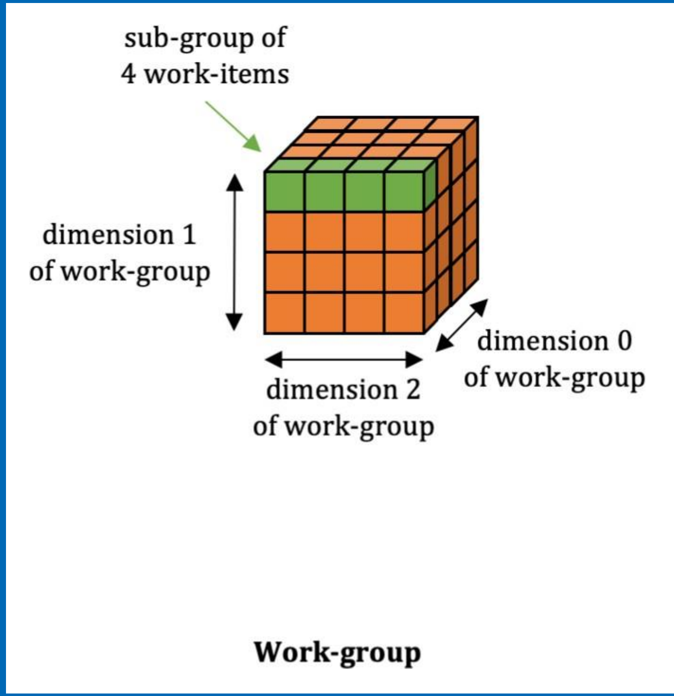
All work-items in a **sub-group** execute on a single EU thread.

Each work-item in a **sub-group** is mapped to a SIMD lane/channel.



Sub Groups

A subset of work-items within a work-group that execute with additional guarantees and often map to SIMD hardware.



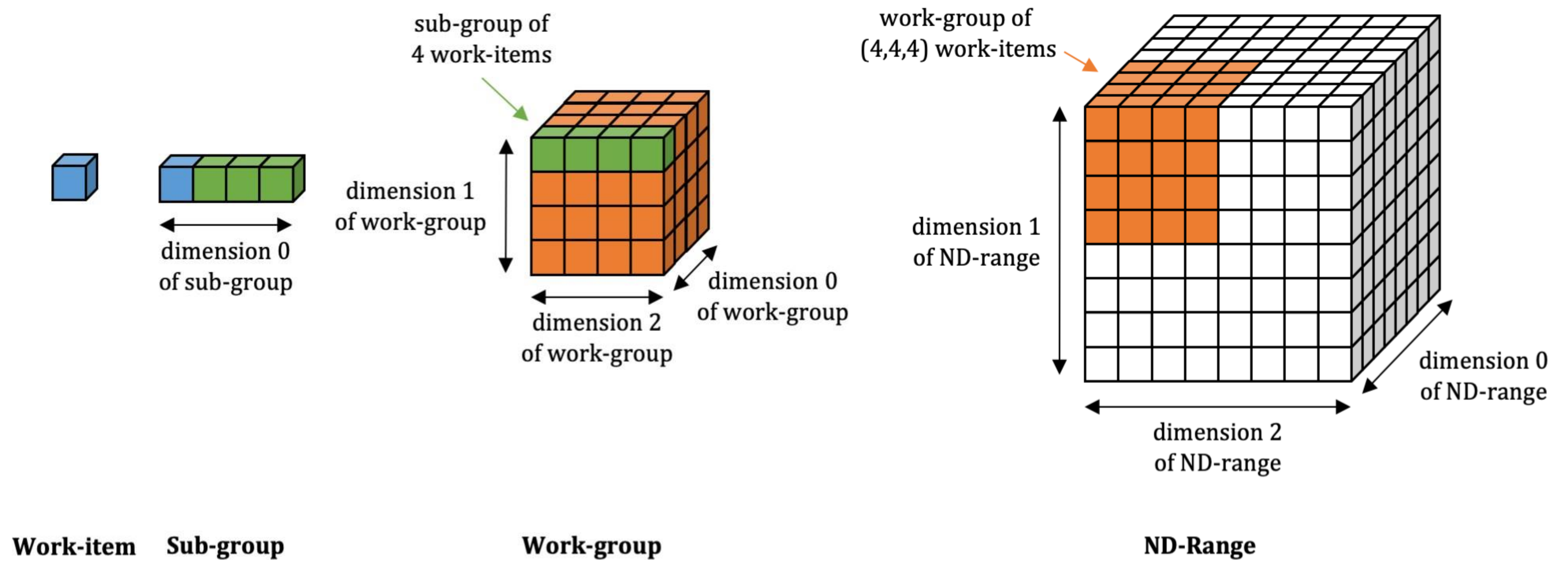
Why use Sub-groups?

- Work-items in a sub-group can communicate directly using shuffle operations, without repeated access to local or global memory, and may provide better performance.
- Work-items in a sub-group have access to sub-group collectives, providing fast implementations of common parallel patterns.

Sub Groups

Sub-Group = subset of work-items within a work-group.

Parallel execution with ND_RANGE Kernel helps to get access to work-group and sub-group



Sub Groups

sub_group class

The sub-group handle can be obtained from the nd_item using the get_sub_group()

Once you have the sub-group handle, you can query for more information about the sub-group, do shuffle operations or use collective functions.

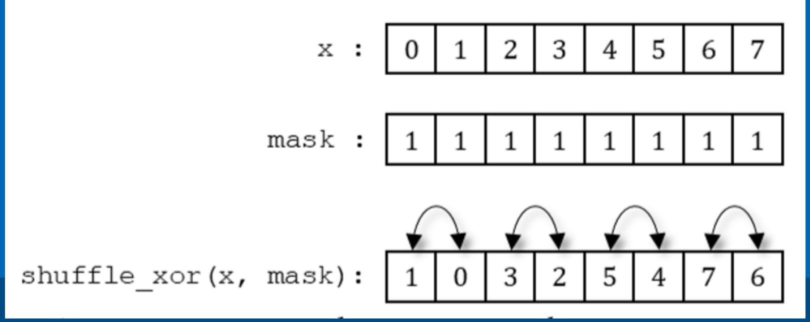
```
q.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
        auto sg = item.get_sub_group();
        // KERNEL CODE
});
```

Sub Groups

Sub-Group Shuffles

- One of the most useful features of sub-groups is the ability to communicate directly between individual work-items without explicit memory operations.
- Shuffle operations enable us to remove work-group local memory usage from our kernels and/or to avoid unnecessary repeated accesses to global memory.

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
        auto sg = item.get_sub_group();
        size_t i = item.get_global_id(0);

        /* Shuffles */
        //data[i] = sg.shuffle(data[i], 2);
        //data[i] = sg.shuffle_up(0, data[i], 1);
        //data[i] = sg.shuffle_down(data[i], 0, 1);
        data[i] = sg.shuffle_xor(data[i], 1);
});
```



Sub Groups

Sub-Group Collectives

- The collective functions provide implementations of closely-related common parallel patterns.
- Providing implementations as library functions increases developer productivity and gives implementations the ability to generate highly optimized code for individual target devices.

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
      auto sg = item.get_sub_group();
      size_t i = item.get_global_id(0);

      /* Collectives */
      data[i] = reduce(sg, data[i], ONEAPI::plus<>());
      //data[i] = reduce(sg, data[i], ONEAPI::maximum<>());
      //data[i] = reduce(sg, data[i], ONEAPI::minimum<>());
});
```

Specifying the Sub-Group Size

The sub-group size can be configured separately for each kernel.
The set of available sub-group sizes is hardware-specific.

```
q.parallel_for(range<1>(N),
               [=](id<1> id) [[intel::reqd_sub_group_size(16)]] {
  // KERNEL CODE
});
```

The sub-group size can be tuned even for kernels that do not use the `sub_group` class (e.g. to tune for SIMD width and register usage).

Sub-groups in SYCL 2020

SYCL 2020 replaces sub-group shuffles from DPC++ with new algorithms

DPC++ extension

```
sycl::ONEAPI::sub_group sg = it.get_sub_group();

auto a = sg.shuffle_down(x, 1);
auto b = sg.shuffle_up(x, 1);
auto c = sg.shuffle(x, id);
auto d = sg.shuffle_xor(x, mask);
```

Shuffles as member functions.

SYCL 2020

```
sycl::sub_group sg = it.get_sub_group();

auto a = sycl::shift_group_left(sg, x, 1);
auto b = sycl::shift_group_right(sg, x, 1);
auto c = sycl::select_from_group(sg, x, id);
auto d = sycl::permute_group_by_xor(sg, x, mask);
```

Shuffles as free functions.
Names aligned with C++.

https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:algorithms

Hands-on Coding on Intel DevCloud

Sub-Group Shuffles and Collectives

Sub Groups

- Summary
  - What are Sub-Groups?
  - Why are they useful?
  - Learned about sub-group shuffle operations and using sub-group collectives

Reductions

A reduction produces a single value by combining multiple values in an unspecified order.

- Parallelizing reductions can be tricky because of the nature of computation and accelerator hardware.
- DPC++ introduces a simplified approach for reductions in heterogenous programming

Simple Reduction

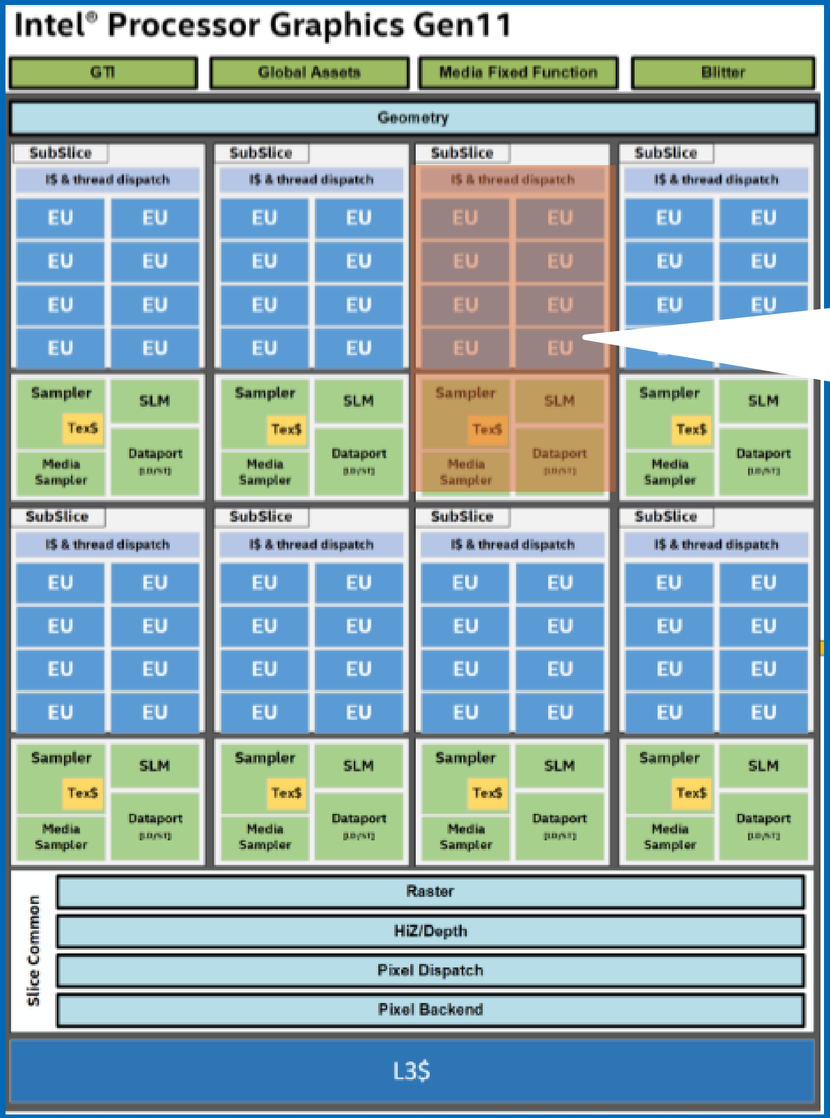
Let's look a simple reduction example: ***Addition of N items***

A simple for-loop in kernel function can accomplish reduction.

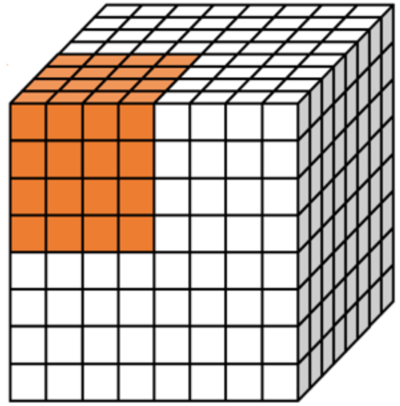
But, for-loop is not efficient and does not take advantage of parallelism in hardware.

```
queue q;
int *data = malloc_shared<int>(N, q);
for (int i = 0; i < N; i++) data[i] = i;

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

std::cout << "Sum = " << data[0] << std::endl;
```

Parallelizing Reductions



work-group executions are mapped to Compute Units on hardware.



Reduction can be parallelized by first reducing items in each work-group using ND-range kernel, multiple work-groups can execute in parallel depending on number of compute units on hardware.

Work-Group Reduction

ND-Range kernel can be used to compute sum of all items in each work-group

ONEAPI::reduce() function will simplify reduction of items in a work-group

A simple for-loop in single_task kernel function can then accomplish final reduction of each work-group sums.

```
q.parallel_for(nd_range<1>(N, B), [=](nd_item<1> item){
  auto wg = item.get_group();
  size_t i = item.get_global_id(0);

  //# Adds all elements in work_group using work_group reduce
  int sum_wg = ONEAPI::reduce(wg, data[i], ONEAPI::plus<>());

  //# write work_group sum to first location for each work_group
  if (item.get_local_id(0) == 0) data[i] = sum_wg;
});
```

```
q.single_task([=](){
  int sum = 0;
  for(int i=0;i<N;i+=B){
      sum += data[i];
  }
  data[0] = sum;
});
```

Some parallelism achieved but code is still complex with 2 kernel functions

Simplified Reduction

DPC++ introduces reduction object in parallel_for

ONEAPI::reduction object in parallel_for encapsulates the reduction variable, an optional operator identity and the reduction operator.

Removes the need for two step approach using two kernel functions.

```
queue q;
auto data = malloc_shared<int>(N, q);
for (int i = 0; i < N; i++) data[i] = i;

auto sum = malloc_shared<int>(1, q);
sum[0] = 0;

q.parallel_for(nd_range<1>{N, B},
                  ONEAPI::reduction(sum, ONEAPI::plus<>()),
                  [=](nd_item<1> it, auto& sum) {
     int i = it.get_global_id(0);
     sum += data[i];
}).wait();

std::cout << "Sum = " << sum[0] << std::endl;
```

SYCL 2020 Reductions

```
myQueue.submit([&](handler& cgh) {

  // Input values to reductions are standard accessors (or USM pointers)
  auto inputValues = accessor(valuesBuf, cgh);

  // Create temporary objects describing variables with reduction semantics
  auto sumReduction = reduction(sumBuf, cgh, plus<>());
  auto maxReduction = reduction(maxBuf, cgh, maximum<>());

  // parallel_for performs two reduction operations
  cgh.parallel_for(range<1>{1024},
    sumReduction, maxReduction,
    [=](id<1> idx, auto& sum, auto& max) {
      sum += inputValues[idx];
      max.combine(inputValues[idx]);
  });
});
```

https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:reduction

Hands-on Coding on Intel DevCloud

Reduction in DPC++

Reductions

- Summary
  - What are Reductions?
  - Parallelizing Reductions in DPC++
  - DPC++ Reduction extension to simplify programming

Summary

DPC++ is a standards-based, cross-architecture language to deliver uncompromised productivity and performance across CPUs and accelerators

- Extends the SYCL standard with new features

New features being developed through a community project

- https://github.com/intel/llvm
- Feel free to open an Issue or submit a PR!

Recap

Learned how to use DPC++ new features like Unified Shared Memory, Sub-Groups and Reduction to simplify programming and achieve performance