Data Parallel C++ - New Features

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

(Presentation Starts at 14:15 CST, 10/15/2020)

LEARNING OBJECTIVES

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

Understand advantages of using Sub-groups in DPC++

Understand advantages of using Data Parallel C++ Library for heterogeneous computing.

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 1.2.1

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* + NEW FEATURES

DPC++ New Features:

- Unified Shared Memory (USM)
- Sub-Groups
- And more…

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

UNIFIED SHARED MEMORY (USM)

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

WHY UNIFIED SHARED MEMORY (USM)

The SYCL 1.2.1 standard provides a Buffer memory abstraction

- Powerful and elegantly expresses data dependences

However…

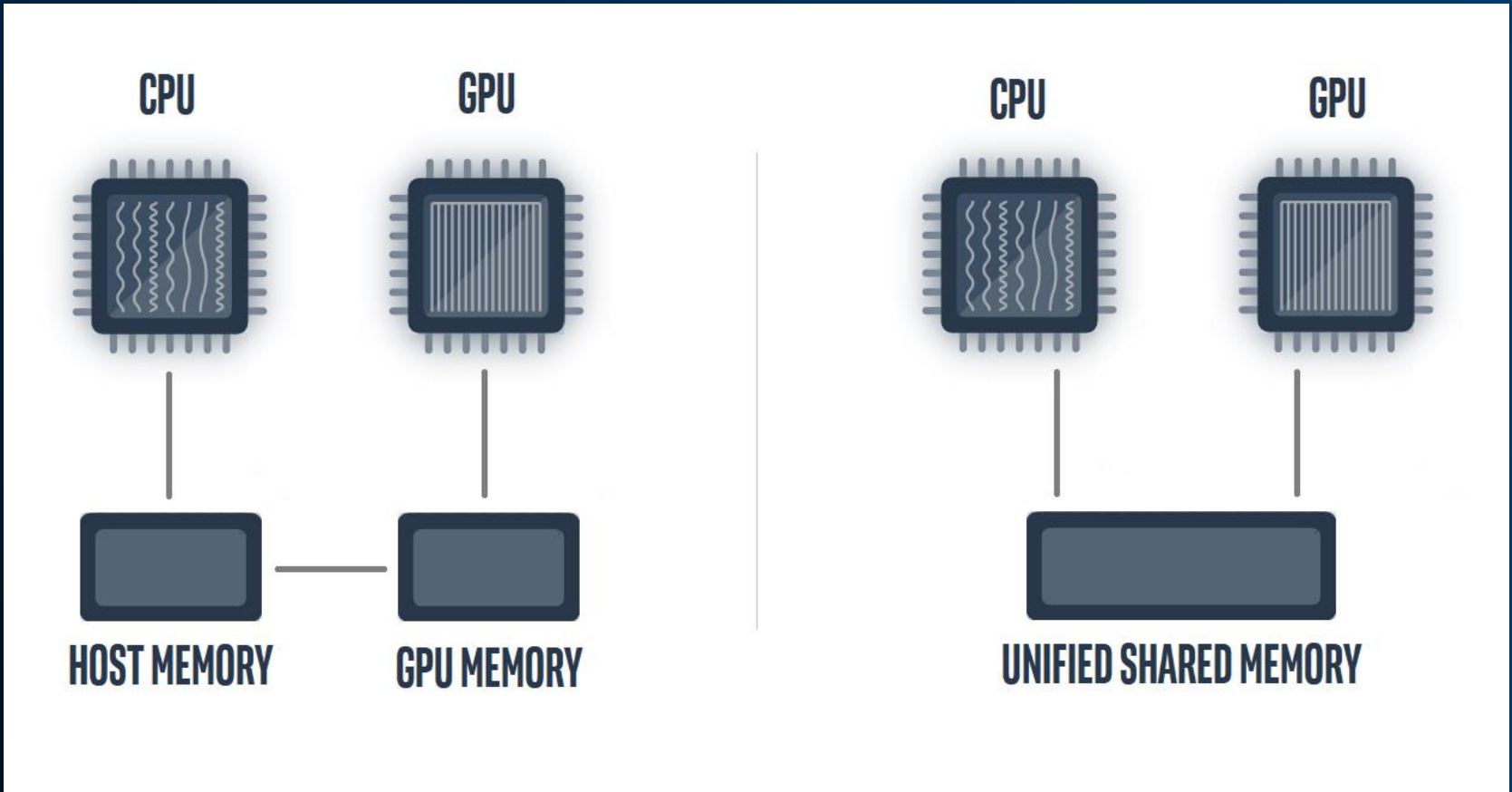
- Replacing all pointers and arrays with buffers in a C++ program can be a burden to programmers

USM provides a pointer-based alternative in DPC++

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

DEVELOPER VIEW OF USM

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



DPC++ UNIFIED SHARED MEMORY

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

Setup Unified Shared Memory

Host can initialize

Device can modify

Host has output

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

SYCL BUFFERS AND ACCESSORS

Memory Model with Buffers & Accessors – 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> my_buffer(data, range<1>(N));
    q.submit([&] (handler &h){
        auto my_accessor = my_buffer.get_access<access::mode::read_write>(h);
        h.parallel_for(range<1>(N), [=](id<1> idx){
            my_accessor[idx] += 1;
        });
    });
}
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data);
```

DPC++ 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));
```

DPC++ UNIFIED SHARED MEMORY

Unified shared memory provides both explicit and implicit models for managing memory.

Allocation Type	Description	Accessible on HOST	Accessible on DEVICE
device	Allocations in device memory (**explicit**)	NO	YES
host	Allocations in host memory (**implicit**)	YES	YES
shared	Allocations can migrate between host and device memory (**implicit**)	YES	YES

Automatic data accessibility and explicit data movement supported

USM – EXPLICIT DATA TRANSFER

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 = static_cast<int*>(malloc(N * sizeof(int)));
int *data_device = static_cast<int*>(malloc_device(N * sizeof(int), q));
for(int i=0;i<N;i++) {data[i] = 10;}

auto e1 = q.memcpy(data_device, data, sizeof(int)*N);
auto e2 = q.submit([&] (handler &h){
    h.depends_on(e1);
    h.parallel_for(range<1>(N), [=](id<1> i){
        data_device[i] *= 2;
    });
});
q.submit([&] (handler &h){
    h.depends_on(e2);
    h.memcpy(data, data_device, sizeof(int)*N);
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data); free(data_device, q);
```

USM – IMPLICIT DATA TRANSFER

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(range<1>(N), [=](id<1> i){
        data[i] += 1;
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
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

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

On many modern hardware platforms, a subset of the work-items in a work-group are executed simultaneously or with additional scheduling guarantees.

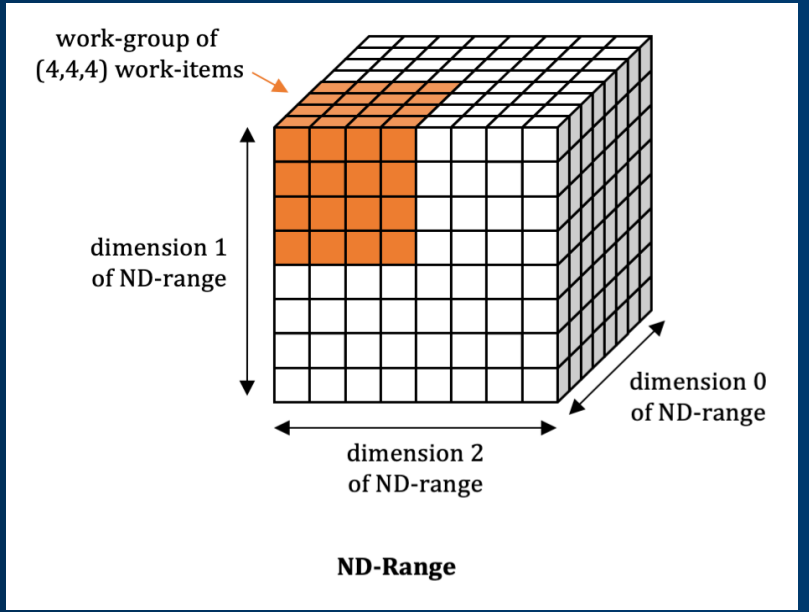
These subset of work-items are called sub-groups, leveraging sub-groups will help to map execution to low-level hardware and may help in achieving higher performance.

ND_RANGE KERNELS

ND-Range kernel is a way to express parallelism which enable mapping executions to compute units on hardware.

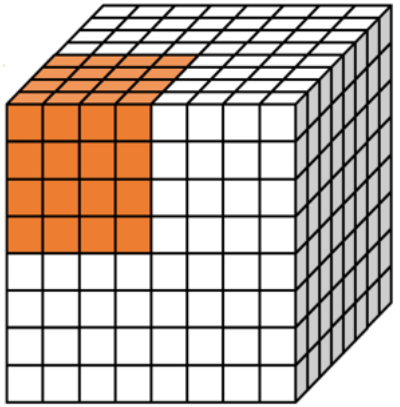
```
range<3> N(8, 8, 8);
range<3> B(4, 4, 4);

h.parallel_for(nd_range<3>(N, B), [=](nd_item<1> item){
    // CODE THAT RUNS ON DEVICE
});
```



HOW IT MAPS TO HARDWARE (INTEL GEN11 GRAPHICS)

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



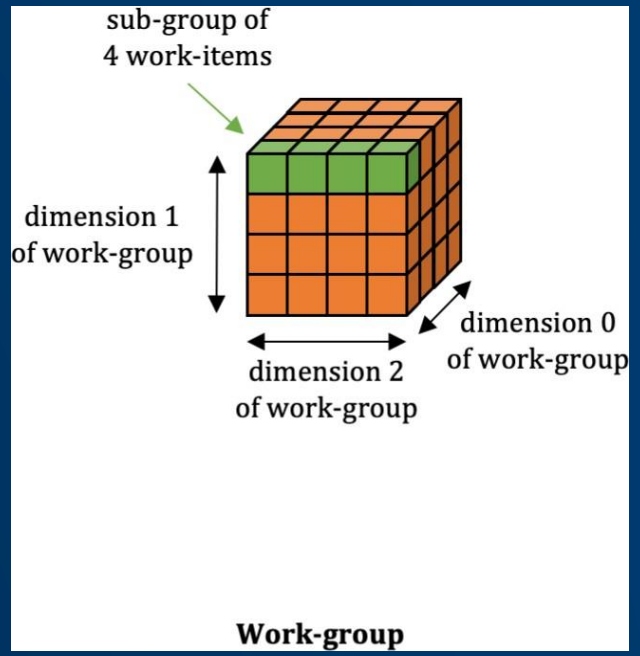
A subset of work-groups called **sub-groups** are mapped to vector hardware

SUB-GROUPS

A subset of work-items within a work-group that may map to vector hardware.

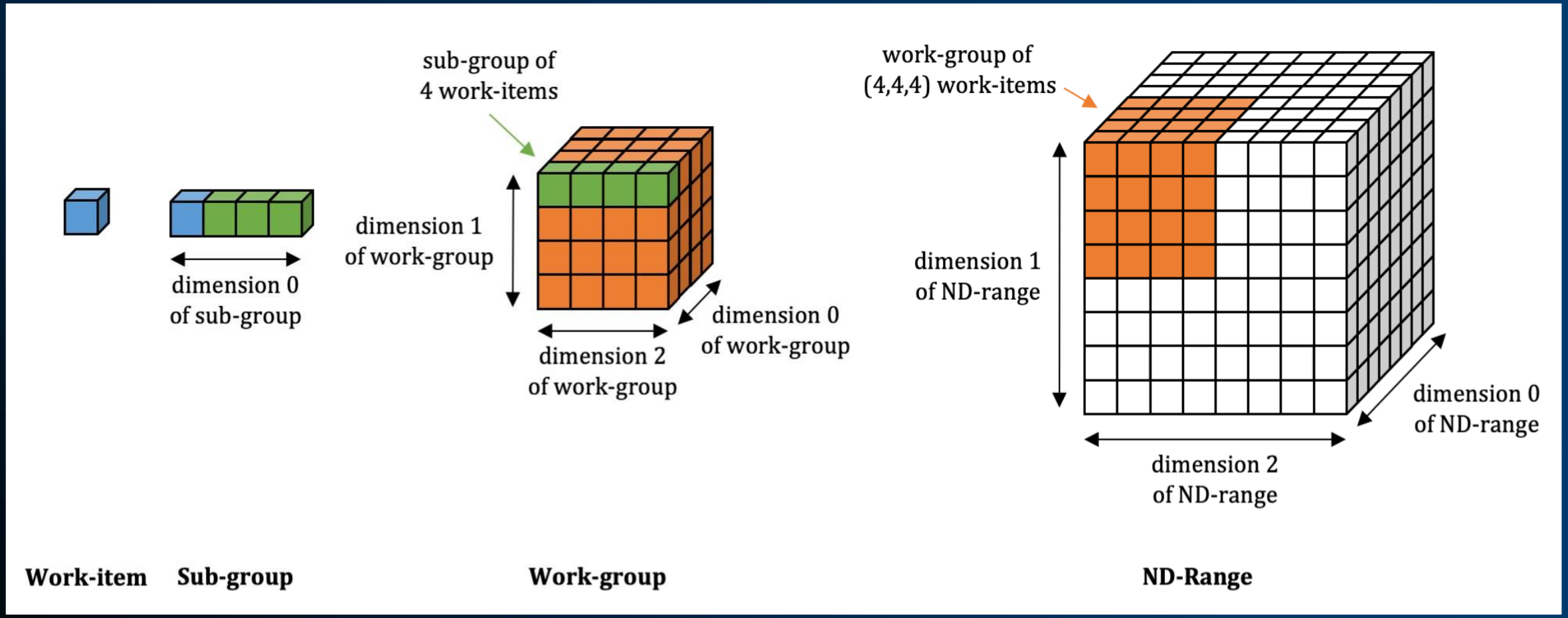
Why use Sub-groups?

- Work-items in a sub-group can communicate directly using shuffle operations, without explicit memory operations.
- Work-items in a sub-group can synchronize using sub-group barriers and guarantee memory consistency using sub-group memory fences.
- Work-items in a sub-group have access to sub-group collectives, providing fast implementations of common parallel patterns.



ND_RANGE KERNEL EXECUTION

Parallel execution with ND_RANGE Kernel helps to group work items that maps to hardware resources. This helps to tune applications for performance.



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.

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){

        intel::sub_group sg = item.get_sub_group();

        // KERNEL CODE
});
```

SUB-GROUPS

The sub-group handle can be queried to get other information:

- **get_local_id()** returns the index of the work-item within its sub-group
- **get_local_range()** returns the size of sub_group
- **get_group_id()** returns the index of the sub-group
- **get_group_range()** returns the number of sub-groups within the parent work-group

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
        intel::sub_group sg = item.get_sub_group();

        if(sg.get_local_id() == 0){
            out << "sub_group id: " << sg.get_group_id()[0]
                << " of " << sg.get_group_range()
                << ", size=" << sg.get_local_range()[0]
                 << endl;
        }
});
```

```
sub_group id: 1 of 4, size=16
sub_group id: 3 of 4, size=16
sub_group id: 2 of 4, size=16
sub_group id: 0 of 4, size=16
```

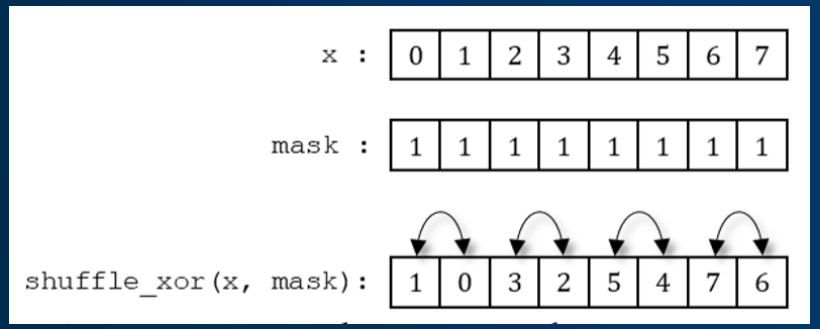
SUB-GROUPS

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){
        intel::sub_group 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 these 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){
        intel::sub_group sg = item.get_sub_group();
        size_t i = item.get_global_id(0);

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

Hands-on Coding on Intel DevCloud

Sub-Group Shuffle and Collectives

SUB-GROUPS

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

WHAT IS DPC++ LIBRARY?

The Intel® oneAPI Data Parallel C++ Library ***(oneDPL)*** is a companion to the Intel® oneAPI DPC++ Compiler and provides an alternative for C++ developers who create heterogeneous applications and solutions.

Its APIs are based on familiar standards and maximizes productivity and performance across CPUs, GPUs, and FPGAs.

WHAT IS DPC++ LIBRARY?

DPC++ Library consists of the following components:

- **Standard C++ APIs** – C++ standard APIs have been tested and function well within DPC++ kernels.
- **Parallel STL** – algorithms which offers efficient support for both parallel and vectorized execution of algorithms for Intel® processors is extended with support for DPC++ compliant devices by introducing special DPC++ execution policies.
- **Extensions APIs** – additional set of algorithm, classes and iterators.

WHY USE DPC++ LIBRARY ?

The Intel oneAPI DPC++ Library helps to **maximize productivity** and **performance** across CPUs, GPUs, and FPGAs.

Compute on host

```
std::sort(v.begin(), v.end());
```

Compute on GPU with oneDPL

```
queue q(gpu_selector{});

std::sort(oneapi::dpl::execution::make_device_policy(q), v.begin(), v.end()
);
```

Execution policy tells where the library function is executed

WHY USE DPC++ LIBRARY ?

Lets look at a simple DPC++ code example and see how DPC++ Library can be used to simplify programming.

DPC++ Kernel Code can be accomplished with one line of oneDPL code

```
queue q;
std::vector<int> v(N);
{
    buffer<int> buf(v.data(),v.size());
    q.submit([&](handler &h){
        auto V = buf.get_access<access::mode::read_write>(h);
        h.parallel_for(range<1>(N),[=] (id<1> i){ V[i] = 20; });
    });
}
for(int i = 0; i < v.size(); i++) std::cout << v[i] << std::endl;
```

WHY USE DPC++ LIBRARY ?

The DPC++ library function used here is Parallel STL std::fill, which executes the functionality on the device and handles all the memory transfers.

DPC++ Library function →

```
queue q;
std::vector<int> v(N);

std::fill(oneapi::dpl::execution::make_device_policy(q), v.begin(), v.end(),
 20);

for(int i = 0; i < v.size(); i++) std::cout << v[i] << std::endl;
```

DPC++ LIBRARY EXAMPLE

Familiar Parallel STL standard algorithm with a execution policy that executes on heterogeneous device and is optimized for data parallelism.

DPC++ Library header files

DPC++ Library function

```
#include<CL/sycl.hpp>
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
constexpr int N = 4;

int main() {
    sycl::queue q;
    std::vector<int> v(N);

    std::fill(oneapi::dpl::execution::make_device_policy(q), v.begin(), v.end(), 20);

    for(int i = 0; i < v.size(); i++) std::cout << v[i] << std::endl;
}
```

Execution policy tells where the library function is executed

HOW DPC++ LIBRARY WORKS?

- Parallel STL algorithms can be called with ordinary iterators
- A temporary SYCL buffer is created and the data is copied to this buffer.
- After processing of the temporary buffer on a device is complete, the data is copied back to the host.

```
std::fill(oneapi::dpl::execution::make_device_policy(q), v.begin(), v.end(), 20);
```

MULTIPLE DPC++ LIBRARY ALGORITHMS

Lets look at a simple DPC++ code example that uses multiple oneDPL algorithms

Works **but** memory is copied back to host after each library function

```
queue q;
std::vector<int> v{2,3,1,4};

std::for_each(make_device_policy(q), v.begin(), v.end(), [](int &a){ a *= 2; });

std::sort(make_device_policy(q), v.begin(), v.end());

for(int i = 0; i < v.size(); i++) std::cout << v[i] << std::endl;
```

To minimize copies and retain memory on device, we use “Buffer Iterators”

DPC++ LIBRARY – BUFFER ITERATORS

Lets look at a how we can minimize memory copies by using buffer iterators

Create sycl buffer

Memory copied back to host on buffer destruction

Buffer Iterators

```
queue q;
std::vector<int> v{2,3,1,4};

{
  buffer buf(v);

auto buf_begin = oneapi::dpl::begin(buf);
auto buf_end   = oneapi::dpl::end(buf);
  std::for_each(make_device_policy(q), buf_begin, buf_end, [](int &a){ a *= 2; });
  std::sort(make_device_policy(q), buf_begin, buf_end);
}
for(int i = 0; i < v.size(); i++) std::cout << v[i] << std::endl;
```

DPC++ LIBRARY – USM POINTERS

Lets look at a simple DPC++ code example and see how DPC++ Library can be used with Unified Shared Memory (USM) pointers.

USM Shared Allocation

Wait for completion

Iterate Pointers

```
queue q;

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

std::fill(make_device_policy(q), data, data + N, 20);
q.wait();

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

Hands-on Coding on Intel DevCloud

oneAPI Data Parallel C++ Library

SUMMARY

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

- Extends the SYCL 1.2.1 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!

NOTICES & DISCLAIMERS

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. No product or component can be absolutely secure. Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804