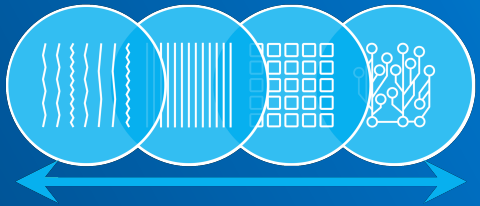
# ONEAPI

SINGLE PROGRAMMING MODEL TO DELIVER CROSS-ARCHITECTURE PERFORMANCE



## MODULE 4
## DPC++ FUNDAMENTALS, PART 2 OF 2

COLFAX RESEARCH

intel

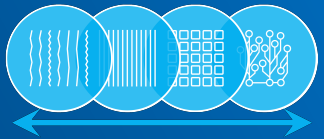
# ONEAPI TRAINING SERIES

- Module 1: Getting Started with oneAPI
- Module 2: Introduction to DPC++
- Module 3: Fundamentals of DPC++, part 1 of 2
- Module 4: Fundamentals of DPC++, part 2 of 2
- Modules 5+: Deeper dives into specific DPC++ features, oneAPI libraries and tools

https://oneapi.com
https://software.intel.com/en-us/oneapi
https://tinyurl.com/book-dpcpp
http://tinyurl.com/oneapimodule?4

1. Hierarchical Parallelism
2. Data Management: USM, Buffers
3. Data Management: Synchronization, DAGs
4. Lab exercise: VTune
5. Module 4 draws to a close

# §1. HIERARCHICAL PARALLELISM



1. Hierarchical Parallelism
2. Data Management: USM, Buffers
3. Data Management: Synchronization, DAGs
4. Lab exercise: VTune
5. Module 4 draws to a close

# HIERARCHICAL PARALLELISM

The hierarchical parallel kernel execution interface provides the same functionality as is available from the nd_range interface, but exposed differently.

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {
  ...
  {
    // define buffers!!!
    queue myQueue{...};
    ...
    myQueue.submit([&](handler &cgh) {

      // Kernels can be invoked as single tasks,
      // basic data-parallel kernels,
      // nd_range in work-groups, or
      // SYCL hierarchical parallelism.

    });
  } // destroy buffers - synchronizes us!
}
```

```
cgh.single_task(
  [=]() {
    // kernel function is executed EXACTLY once on a SINGLE work-item
});

cgh.parallel_for(
  range<3>(1024,1024,1024), // using 3D in this example
  [=](id<3> myID) {
    // kernel function is executed on an n-dimensional range (NDrange)
});

cgh.parallel_for(
  nd_range<3>({1024,1024,1024},{16,16,16}), // using 3D in this example
  [=](nd_item<3> myID) {
    // kernel function is executed on an n-dimensional range (NDrange)
});

cgh.parallel_for_work_group(
  range<2>(1024,1024),      // using 2D in this example
  [=](group<2> myGroup) {
    // kernel function is executed once per work-group
});

grp.parallel_for_work_item(
  range<1>(1024),           // using 1D in this example
  [=](h_item<1> myItem) {
    // kernel function is executed once per work-item
});
```

# HIERARCHICAL PARALLELISM

- *top-down* programming style (vs. the *bottom-up* style of the other kernel forms
- Very compelling, very C++
- Relatively new to SYCL

Subgroups currently limited to nd_range (not in hierarchical parallelism)

# HIERARCHICAL PARALLELISM

- a way of specifying data parallel kernels that execute within work-groups via a different syntax which highlights the hierarchical nature of the parallelism
- allows us to write code that looks like there is an inner work-item loop inside an outer work-group loop, which closely matches the effect of the execution model
- purely a compiler feature and does not change the execution model of the kernel

# HIERARCHICAL PARALLELISM

- a way of specifying data parallel kernels that execute within work-groups via a different syntax which highlights the hierarchical nature of the parallelism
- allows us to write code that looks like there is an inner work-item loop inside an outer work-group loop, which closely matches the effect of the execution model
- purely a compiler feature and does not change the execution model of the kernel

# HIERARCHICAL PARALLELISM

- a way of specifying data parallel kernels that execute within work-groups via a different syntax which highlights the hierarchical nature of the parallelism
- allows us to write code that looks like there is an inner work-item loop inside an outer work-group loop, which closely matches the effect of the execution model
- purely a compiler feature and does not change the execution model of the kernel

# HIERARCHICAL PARALLELISM

- instead of calling cl::sycl::parallel_for, call cl::sycl::parallel_for_work_group with:
  - a cl::sycl::range value = number of work-groups to launch
  - optional 2nd cl::sycl::range = size of each work-group for performance tuning
- Code within the parallel_for_work_group scope effectively executes once per work-group
- Within parallel_for_work_group scope, it is possible to call parallel_for_work_item
  - creates a new scope in which all work-items within the current work-group execute
- variables declared inside the parallel_for_work_group scope are allocated in workgroup local memory
  - vs. all variables declared inside the parallel_for_work_item scope are declared in private memory
- parallel_for_work_item calls within a given parallel_for_work_group execution must have the same dimensions

# HIERARCHICAL PARALLELISM

- instead of calling cl::sycl::parallel_for, call cl::sycl::parallel_for_work_group with:
  - a cl::sycl::range value = number of work-groups to launch
  - optional 2nd cl::sycl::range = size of each work-group for performance tuning
- Code within the parallel_for_work_group scope effectively executes once per work-group
- Within parallel_for_work_group scope, it is possible to call parallel_for_work_item
  - creates a new scope in which all work-items within the current work-group execute
- variables declared inside the parallel_for_work_group scope are allocated in workgroup local memory
  - vs. all variables declared inside the parallel_for_work_item scope are declared in private memory
- parallel_for_work_item calls within a given parallel_for_work_group execution must have the same dimensions

# EXAMPLE: HIERARCHICAL PARALLELISM (HPAR.CPP - 1 OF 2)

```
theQueue.submit([&](handler & cgh) {
  cl::sycl::stream kernelout(108*64+128, 64, cgh);

  // Issue 27 work-groups of 4 work-items each
  cgh.parallel_for_work_group<class example_kernel>(
    range<3>(3, 3, 3),
    range<3>(2, 1, 2),
    [=](group<3> theGroup) {
      // *** workgroup code...
      kernelout << "WG+" << theGroup.get_id()
      << cl::sycl::endl;

      // >>>> cgh.parallel_for_work_group <<<<
      // >>>>      SEE next slide        <<<<

      // *** more workgroup code...
    });
  });
```

- Per item work assigned on next slide...

# EXAMPLE: HIERARCHICAL PARALLELISM (HPAR.CPP - 2 OF 2)

```
// Issue parallel work-items.  The number issued per
// work-group is determined by the work-group size
// range of parallel_for_work_group.  In this case,
// 4 work-items will execute the parallel_for_work_item
// body for each of the 27 work-groups, resulting in
// 108 executions globally/total.
theGroup.parallel_for_work_item([&](h_item<3> theItem) {
  kernelout << "WI1." << theItem.get_global_id()
  << cl::sycl::endl;
  });
// Implicit work-group barrier
// We can have as many parallel_for_work_items as we want
theGroup.parallel_for_work_item([&](h_item<3> theItem) {
  // *** work-item code ***
  kernelout << "WI2." << theItem.get_global_id()
  << cl::sycl::endl;
  });
```

- placed in workgroup (per prior slide)

# OUTPUT OF HPAR.CPP

```
WG+{0, 0, 0}
WG+{0, 0, 1}
WG+{0, 0, 2}
WG+{0, 1, 0}
WG+{0, 1, 1}
WG+{0, 1, 2}
WG+{0, 2, 0}
WG+{0, 2, 1}
WG+{0, 2, 2}
WG+{1, 0, 0}
WG+{1, 0, 1}
WG+{1, 0, 2}
WG+{1, 1, 0}
WG+{1, 1, 1}
WG+{1, 1, 2}
WG+{1, 2, 0}
WG+{1, 2, 1}
WG+{1, 2, 2}
WG+{2, 0, 0}
WG+{2, 0, 1}
WG+{2, 0, 2}
WG+{2, 1, 0}
WG+{2, 1, 1}
WG+{2, 1, 2}
WG+{2, 2, 0}
WG+{2, 2, 1}
WG+{2, 2, 2}
```

```
WI1.{0, 0, 0}
WI1.{0, 0, 1}
WI1.{0, 0, 2}
WI1.{0, 0, 3}
WI1.{0, 0, 4}
WI1.{0, 0, 5}
WI1.{0, 1, 0}
WI1.{0, 1, 1}
WI1.{0, 1, 2}
...
WI1.{5, 0, 1}
WI1.{5, 0, 2}
WI1.{5, 0, 3}
WI1.{5, 0, 4}
WI1.{5, 0, 5}
WI1.{5, 1, 0}
WI1.{5, 1, 1}
WI1.{5, 1, 2}
WI1.{5, 1, 3}
WI1.{5, 1, 4}
WI1.{5, 1, 5}
WI1.{5, 2, 0}
WI1.{5, 2, 1}
WI1.{5, 2, 2}
WI1.{5, 2, 3}
WI1.{5, 2, 4}
WI1.{5, 2, 5}
```

```
WI2.{0, 0, 0}
WI2.{0, 0, 1}
WI2.{0, 0, 2}
WI2.{0, 0, 3}
WI2.{0, 0, 4}
WI2.{0, 0, 5}
WI2.{0, 1, 0}
WI2.{0, 1, 1}
WI2.{0, 1, 2}
...
WI2.{5, 0, 1}
WI2.{5, 0, 2}
WI2.{5, 0, 3}
WI2.{5, 0, 4}
WI2.{5, 0, 5}
WI2.{5, 1, 0}
WI2.{5, 1, 1}
WI2.{5, 1, 2}
WI2.{5, 1, 3}
WI2.{5, 1, 4}
WI2.{5, 1, 5}
WI2.{5, 2, 0}
WI2.{5, 2, 1}
WI2.{5, 2, 2}
WI2.{5, 2, 3}
WI2.{5, 2, 4}
WI2.{5, 2, 5}
```

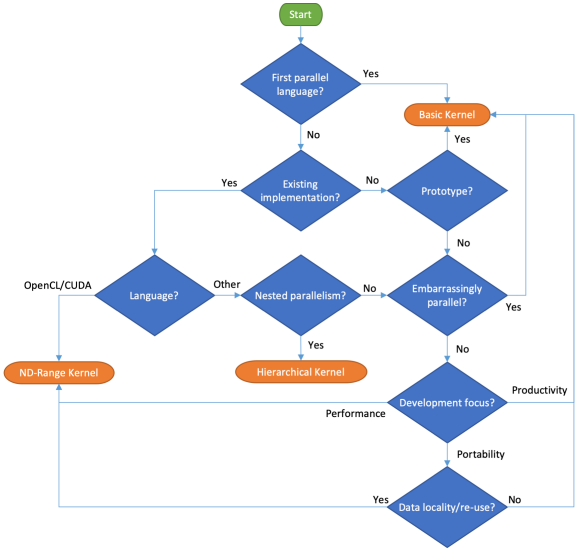
- hpar.cpp
- sorted output
- 1st column: 27 WG+ items
- 2nd column: 108 WI1. items
- 3rd column: 108 WI2. items

# H_ITEM

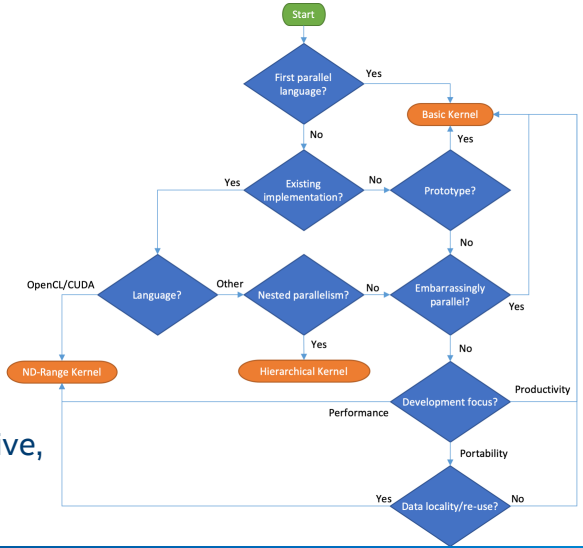
```
// Issue parallel work-items.  The number issued per
// work-group is determined by the work-group size
// range of parallel_for_work_group.  In this case,
// 4 work-items will execute the parallel_for_work_item
// body for each of the 27 work-groups, resulting in
// 108 executions globally/total.
theGroup.parallel_for_work_item([&](h_item<3> theItem) {
  kernelout << "WI1." << theItem.get_global_id()
  << cl::sycl::endl;
  });
// Implicit work-group barrier
// We can have as many parallel_for_work_items as we want
theGroup.parallel_for_work_item([&](h_item<3> theItem) {
  // *** work-item code ***
  kernelout << "WI2." << theItem.get_global_id()
  << cl::sycl::endl;
  });
```

- provides access to local ids and ranges that reflect both kernel and parallel_for_work_item invocation ranges
  - encapsulates physical global and localids and ranges
  - also contains a logical/flexible localid and range defined by hierarchical parallelism

# HIERARCHICAL - SELECT FOR PERFORMANCE OR FAMILIARITY

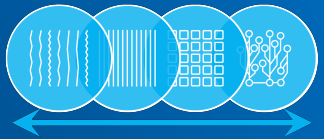


# HIERARCHICAL - SELECT FOR PERFORMANCE OR FAMILIARITY




Hierarchical is both more descriptive, and more prescriptive, than ND range!

# §2. DATA MANAGEMENT: USM, BUFFERS



1. Hierarchical Parallelism
2. Data Management: USM, Buffers
3. Data Management: Synchronization, DAGs
4. Lab exercise: VTune
5. Module 4 draws to a close

# WHY MEMORY OBJECTS?

Memory objects, along with accessors, allow us to inform the implementation how to move data. Together they inform dependency information that allow the runtime to know when kernels can be launched.

# WHY MEMORY OBJECTS?

- Traditional CPU-based systems have a single memory.
- Accelerator devices often have their own distinct (attached) memories.
- The resulting coherence model - requires us to understand what coherence we want (or need), and how to ask for it.
- Memory objects, along with accessors, allow us to inform the implementation how to move data.

# ACCESS LOCALLY!

- Accessing data local to a device is more efficient (faster) than accesing remote data.
- Therefore, co-locating code (the kernels) with the data they access, is very important.

# IMPLICIT OR EXPLICIT MOVEMENT?

- SYCL provides mechanisms to for us to describe the connections to allow the compiler/runtime to manage movement (descriptive provides for implicit data movement), and yet help us control the movement when we want (prescriptive allows for explicit data movement).

# USM VS. BUFFERS VS. IMAGES

- Unified Shared Memory: pointer-based approach that is familiar for C++ programmers
- Buffers: abstract view of memory that can be local to the host or a device, and is accessible only via accessors.
- Images: a special type of buffer that has special extra functionality specific to image processing.

# UNIFIED SHARED MEMORY (USM)

- DPC++ only (not part of the SYCL 1.2.1 specification)
- Requires hardware support for a unified virtual address space (this allows a pointer value to be the same on a device and the host).
- All memory is allocated by the host, however USM supports three allocation types:
  - device - located on the device, not accesible by the host
  - host - located on the host, accessible by host or device
  - shared - accessible by host or device, location can migrate back and forth

# USM MEMCPY FOR EXPLICIT DATA MOVEMENT

```
queue myQueue;
auto dev = myQueue.get_device();
auto ctxt = myQueue.get_context();
int hostArray[42];
int *deviceArray = (int*) malloc_device(42 * sizeof(int), dev, ctxt);
for (int i = 0; i < 42; i++) hostArray[i] = 42;
myQueue.submit([&](handler& h) {
  // copy hostArray to deviceArray
  h.memcpy(deviceArray, &hostArray[0], 42 * sizeof(int));
  });
myQueue.wait(); // needed for now (we learn how to ditch soon)
myQueue.submit([&](handler& h) {
  h.parallel_for<class foo>(range<1>{42}, [=](id<1> ID) {
    int i = ID[0];
    deviceArray[i]++;
  });  });
myQueue.wait(); // needed for now (we learn how to ditch soon)
myQueue.submit([&](handler& h) {
  // copy deviceArray back to hostArray
  h.memcpy(&hostArray[0], deviceArray, 42 * sizeof(int));
});
myQueue.wait(); // needed for now (we learn how to ditch soon)
```

- device memory can be accessed by the host via a memcpy operation

# USM IMPLICIT DATA MOVEMENT

```
queue myQueue;
auto dev = myQueue.get_device();
auto ctxt = myQueue.get_context();
int *hostArray = (int*) malloc_host(42 * sizeof(int), ctxt);
int *sharedArray = (int*) malloc_shared(42 * sizeof(int), dev, ctxt);
for (int i = 0; i < 42; i++) hostArray[i] = 1234;
myQueue.submit([&](handler& h) {
  h.parallel_for<class theKernel>(range<1>{42}, [=](id<1> myID) {
    int i = myID[0];
    // access sharedArray and hostArray on device
    sharedArray[i] = hostArray[i] + 1;
  });
});
myQueue.wait();
for (int i = 0; i < 42; i++) hostArray[i] = sharedArray[i];
free(sharedArray, ctxt);
free(hostArray, ctxt);
```

- no memcpy needed for host or shared types, but coherence needs to be understood (hence the calls to wait)

# BUFFERS

- abstractions that represent one or more objects of a C++ type
- once data is represented by a buffer, that data must be accessed by an accessor for the l ife of the buffer
- destruction of the buffer will move data back if that was requested by the type of buffer
- creation can be empty (no initialization) or created from data that exists on the host

# BUFFER EXAMPLE - MODULE03/EXPLORE1.CPP

```
1  // Set up SYCL device and queue.
2  queue q = queue();
3
4  // ultimately we will be 4 x 4 x 4 in 3D
5  const uint32_t D = 4;
6
7  std::vector<int> x(D*D*D);
8  std::vector<int> y(D*D*D);
9  std::vector<int> z(D*D*D);
10 std::fill(x.begin(), x.end(), 7);
11 std::fill(y.begin(), y.end(), 8);
12 std::fill(z.begin(), z.end(), 9);
13
14 {
15   // buffers for device access to x[], y[], and z[]
16   buffer<int,1> x_buf(x.data(), range<1>(D*D*D));
17   buffer<int,1> y_buf(y.data(), range<1>(D*D*D));
18   buffer<int,1> z_buf(z.data(), range<1>(D*D*D));
19
20   q.submit([&](handler& cgh) {
21
22         // accessors are way for device to touch shared data
23         auto xx = x_buf.get_access<access::mode::read_write>(cgh);
24         auto yy = y_buf.get_access<access::mode::read_write>(cgh);
25         auto zz = z_buf.get_access<access::mode::read_write>(cgh);
```

- explore1.cpp
- more observations

# BUFFER EXAMPLE - MODULE03/EXPLORE1.CPP

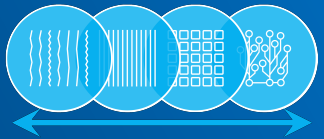
```
1  cgh.parallel_for<class foo>(range<1>{D*D*D}, [=](id<1> item) {
2    xx[ 3 ] = 3;
3    yy[ 4 ] = 4;
4    zz[ 5 ] = 5;
5  });
6
7
8     0         1         2         3         4         5         6
9     .123456789.123456789.123456789.123456789.123456789.123456789.123
10 x: 777377777777777777777777777777777777777777777777777777777777777
11 y: 888848888888888888888888888888888888888888888888888888888888888
12 z: 999995999999999999999999999999999999999999999999999999999999999
```

- explore1.cpp
- sample output (output should not vary)

# BUFFER ACCESS MODES

- read: read-only access
- write: write-only access (original contents preserved if not overwritten)
- discard_write: write-only access (value non-deterministic if not written)
- read_write: Read and write access
- discard_read_write: Read and write access (value non-deterministic if not written)
- atomic: Read and write with atomic access on an element-by-element basis.

# §3. DATA MANAGEMENT: SYNCHRONIZATION, DAGS



1. Hierarchical Parallelism
2. Data Management: USM, Buffers
3. Data Management: Synchronization, DAGs
4. Lab exercise: VTune
5. Module 4 draws to a close

# Root of all Parallel Programming Evil: Shared Mutable State

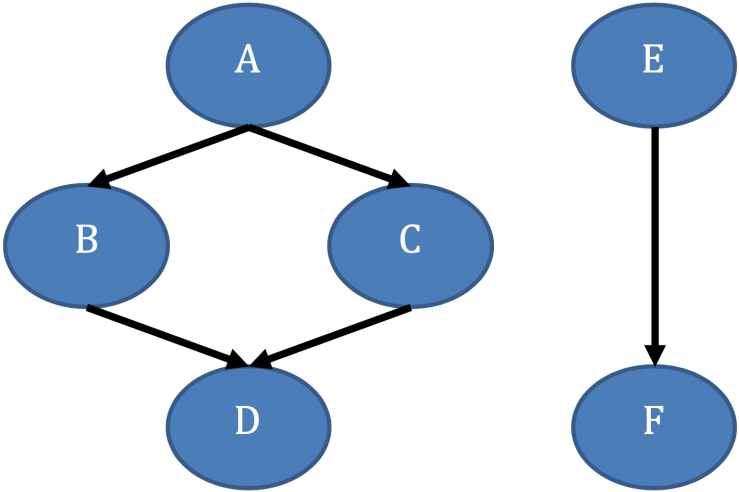
# ORDERING USE OF DATA

- If any shared data value is changed (written), then the timing of another read or write may matter (to the correctness of the program).
- RAW (read-after-write)
- WAW (write-after-write)
- WAR (write-after-read)

# DEPENDENCIES IN A KERNEL-BASED WORLD

- The order of kernels may matter - think of kernels as tasks with potential dependencies.
- Data transfers to/from the host create dependencies also - think of data accesses on the hosts as defining tasks with potential dependencies as well.

# TASK GRAPH WITH DISJOINT DEPENDENCIES



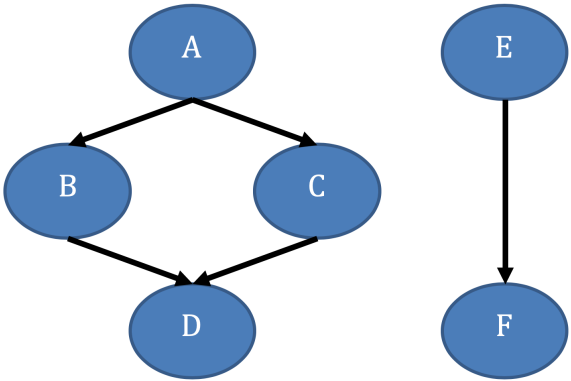

E must be done before F

A, B, and C must be done before D

A must be done before B or C

# IN-ORDER QUEUES

```
ordered_queue myQueue;
myQueue.submit([&](handler& h) {
  h.parallel_for<class taskA>(…);
});
myQueue.submit([&](handler& h) {
  h.parallel_for<class taskB>(…);
});
myQueue.submit([&](handler& h) {
  h.parallel_for<class taskC>(…);
});
myQueue.submit([&](handler& h) {
  h.parallel_for<class taskD>(…);
});
myQueue.submit([&](handler& h) {
  h.parallel_for<class taskE>(…);
});
myQueue.submit([&](handler& h) {
  h.parallel_for<class taskF>(…);
});
```



# DEPENDENCIES VIA COMMAND GROUPS

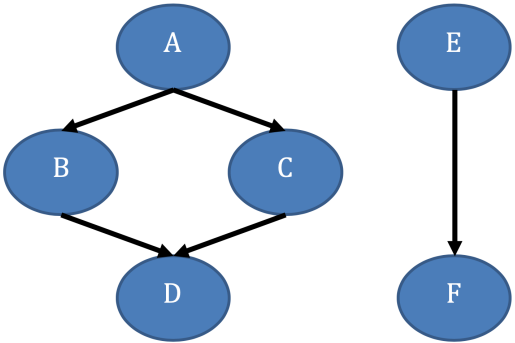
- We've been using *handler &cgh* in examples already!
- A command group handler object can only be constructed by the SYCL runtime.
- The runtime determines the dependencies, because:
  - all of the accessors defined in command group scope take as a parameter an instance of the command group handler, and
  - all the kernel invocation functions are member functions of this class.
- An instance of a command group handler may not be moved or copied.

# EXPLICIT DEPENDENCIES

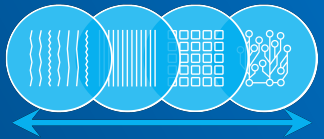
- Tasks can be explictly ordered.
- myQueue.wait() - waits for everything submitted to a queue to finish.
- auto myTokenX = myQueue.submit...
  - myToken.wait() - waits for a particular submission to finish.
  - cgh.depends_on(myToken) - waits for another submission (DPC++ only)

# IN-ORDER QUEUES

```
queue myQueue;
auto myTokA = myQueue.submit([&](handler& h) {
  h.parallel_for<class taskA>(...);
});
auto myTokB = myQueue.submit([&](handler& h) {
  h.depends_on(myTokA);
  h.parallel_for<class taskB>(...);
});
auto myTokC = myQueue.submit([&](handler& h) {
  h.depends_on(myTokA);
  h.parallel_for<class taskC>(...);
});
auto myTokD = myQueue.submit([&](handler& h) {
  h.depends_on(myTokB); // these two could be combined
  h.depends_on(myTokC); // using a vector form of the call
  h.parallel_for<class taskD>(...);
});
auto myTokE = myQueue.submit([&](handler& h) {
  h.parallel_for<class taskE>(...);
});
auto myTokF = myQueue.submit([&](handler& h) {
  h.depends_on(myTokE);
  h.parallel_for<class taskF>(...);
});
```



# §4. LAB EXERCISE: VTUNE

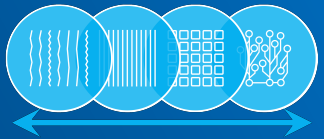


1. Hierarchical Parallelism
2. Data Management: USM, Buffers
3. Data Management: Synchronization, DAGs
4. Lab exercise: VTune
5. Module 4 draws to a close

# LAB EXERCISE: VTUNE

- Follow the directions in the Lab-VTune subdirectory of the module04 directory.
- The instructions will teach you to use the remote desktop connection to DevCloud to utilize VTune to analyze a program.

# §5. MODULE 4 DRAWS TO A CLOSE

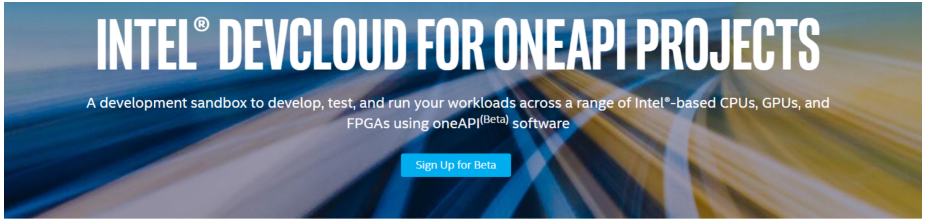


1. Hierarchical Parallelism
2. Data Management: USM, Buffers
3. Data Management: Synchronization, DAGs
4. Lab exercise: VTune
5. Module 4 draws to a close

# ONEAPI TRAINING SERIES

- Module 1: Getting Started with oneAPI
- Module 2: Introduction to DPC++
- Module 3: Fundamentals of DPC++, part 1 of 2
- Module 4: Fundamentals of DPC++, part 2 of 2
- Modules 5+: Deeper dives into specific DPC++ features, oneAPI libraries and tools

https://oneapi.com
https://software.intel.com/en-us/oneapi
https://tinyurl.com/book-dpcpp
http://tinyurl.com/oneapimodule?4




What You Can Do

- Learn Data Parallel C++
- Learn about Intel® oneAPI Toolkits
- Evaluate Workloads
- Prototype Your Project
- Build Heterogeneous Applications

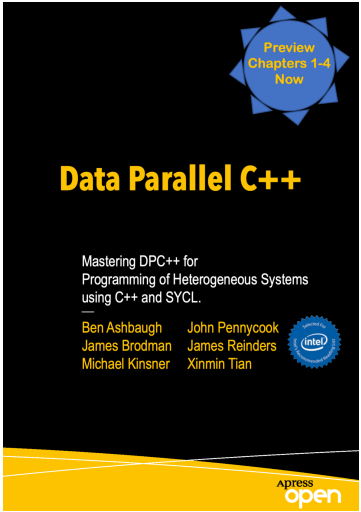
https://software.intel.com/en-us/devcloud/oneapi

# RESOURCES

- Book (Chapters 1-4 Preview)
- oneAPI Toolkit(s)
- Training, Support, Forums, Example Code

All available
Free

https://software.intel.com/en-us/oneapi




https://tinyurl.com/book-dpcpp
http://tinyurl.com/oneapimodule?4

# NEXT FOR YOU

- Watch prior modules if you skipped them!
- Do the three labs to experience more (two in Module 3, one in Module 4)
- Write code - explore - enjoy!
- Explore the oneAPI toolkits
- Read the DPC++ book - preprint/preview of Chapters 1-4
- Watch for more availability of Training Modules 5+