ONEAPI

SINGLE PROGRAMMING MODEL TO DELIVER CROSS-ARCHITECTURE PERFORMANCE

MODULE 3
DPC++ FUNDAMENTALS, PART 1 OF 2
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?3
DPC++ Programs
Execution Model
Where and how to get and use DPC++, etc.
id, item, nd_item
Lab exercise: VADD on Various Devices
Host/Accelerator Model
Lab exercise: Stencil
Module 3 draws to a close
§1. DPC++ PROGRAMS
DPC++ Programs

Execution Model

Where and how to get and use DPC++, etc.

id, item, nd_item

Lab exercise: VADD on Various Devices

Host/Accelerator Model

Lab exercise: Stencil

Module 3 draws to a close
Training Module 2 covers What and Why of DPC++
Training Module 1 covers What and Why of oneAPI
WHAT IS DPC++?

Data Parallel C++ is

- Open source project built on
  - C++
  - with SYCL for data parallelism
  - with a few extensions to smooth things out

Note: Extensions are not Intel specific - but Intel implemented, with an eventually hope of influencing SYCL. This includes subgroups (covered in training module 3) and USM (covered in module 4).
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {

▷ Include the SYCL header file.
  • defines the runtime API
  • To use Intel's FPGA selector, also
    #include <CL/sycl/intel/fpga_extensions.hpp>

▷ Most of us will also add a using for namespace
  cl::sycl to make our coding more readable.
```cpp
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {
    ...

    queue myQueue{...};
}
```

- All work requests are done via a queue.
- A queue uniquely attaches to a single device (e.g., GPU, FPGA, AI, CPU, Host).
```cpp
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {
    ...

    queue myQueue{...};
    ...

    myQueue.submit([&](handler &cgh) {
        // accessors (for connecting to memory via buffers)
        // kernel defined here (with lambda -
        // by value captures only)
    });
}
```

- Queue accepts work requests as submissions.
- Highlighted lines are the command group scope.
- Submissions finish asynchronously.
- Only one kernel (work described in a lambda) per submit!
```cpp
#include <CL/sycl.hpp>
using namespace cl::sycl;

int main(int argc, char *argv[]) {
    ...
    {
        // define buffers!!!
        queue myQueue{...};
        ...
        myQueue.submit([&](handler &cgh) {
            // accessors (for connecting to memory
            // via buffers) choose one of four ways
            // to express parallelism; only one per
            // submit; use by-value lambda capture

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

OneAPI module 3: DPC++ Fundamentals, Part 1 of 2
```cpp
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {
    ...

    {
        // define buffers!!!
        queue myQueue{...};
        ...
        myQueue.submit([&](handler &cgh) {
            // accessors (for connecting to memory via buffers)
            // kernel defined here (with lambda -
            // by value captures only)
        });
        } // destroy buffers - synchronizes us!
    }

    // destroy buffers synchronizes us!
}
```

- Control the scope of buffers, to control synchronization with host programs.
- This is a convention - follow it!
We should *always* wrap SYCL code in a try-catch-block!

```cpp
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {
    ...
    try {
        // define buffers!!!
        queue myQueue{...};
        ...
        myQueue.submit([&](handler &cgh) {
            // accessors (for connecting to memory via buffers)
            // kernel defined here (with lambda -
            // by value captures only)
            });
        myQueue.wait_and_throw();
    } catch (...) { /* error handling */ }
    // destruction of buffers synchronizes us too
}
```

- **Two categories of errors exist:**
  - runtime error due to scheduling errors that may happen during execution
  - device error due to the execution errors on a SYCL device

- Some are thrown asynchronously at the use of a SYCL API (caught by try/catch in code to left).
- Some are thrown synchronously, they are stored by the runtime and passed to and async_handler (myAsyncHandler in code to left).
- We will come back to error handling in a *future* training module, so I will completely ignore error handling for all examples today. 😊
§2. EXECUTION MODEL
DPC++ Programs

Execution Model

Where and how to get and use DPC++, etc.

id, item, nd_item

Lab exercise: VADD on Various Devices

Host/Accelerator Model

Lab exercise: Stencil

Module 3 draws to a close
DPC++ vocabulary follows and extends vocabulary of CUDA, OpenCL, SYCL.
Kernel functions are executed as work-items.

- Like a thread, yet very different from a C++ thread
- A work-item cannot synchronize with another work-item (achieve by kernel must end, submit another kernel invocation). Wait - sub-groups and work-groups offer synchronizations.
  - could be a OS thread
  - but it could be done on a GPU element
  - or it could be processed in an FPGA
  - or it could be processed in a DSP
  - or it could be processed in an AI accelerator

Such flexibility for target, brings some restrictions and responsibilities.
RESTRICTIONS ON KERNEL CODE

Supported include:

▷ lambdas
▷ operator overloading
▷ templates
▷ classes
▷ static polymorphism
▷ share data with host via accessors
▷ read-only values of host variables subject via lambda captures

Not supported:

▷ dynamic polymorphism
▷ dynamic memory allocations
▷ static variables
▷ function pointers
▷ pointer structure members
▷ runtime type information
▷ exception handling
We collect work-items into (work) sub-groups, and work-groups.

- There are sub-group and group barriers, which force all work-items in a particular sub-group or group to reach a certain point. There are also group and sub-group memory fences to manage memory consistency.
- SYCL (currently) only provides work-groups.
- work-items are single items (1D)
- work-groups can be 3D (or 2D)
- work-subgroup (DPC++ specific) give us a 2D option (useful when working in 3D spaces)
- When optimizing for performance, the choice of sizes for work-groups, and for sub-groups, have a connection to the capabilities of the device(s) being targeted. This is an advanced topic we will discuss in a future training module, and not important for making functional DPC++ code.
Kernel functions are invoked in an ND-Range.

▷ An ND-Range consists of work-groups, which consist of sub-groups, which consist of work-items.

▷ Work-groups in a range, all have the same size (number of sub-groups, total number of work-items).

▷ Work sub-groups in a range, all have the same size (number of work-items).
While an ND Range can be 3D, it can also be 2D, or 1D.

- Dimension support is for programmer convenience, the mapping to a contiguous linear block of a memory always sits underneath in a predictable reliable fashion - allowing us to reason about locality (for optimization).

- There is no built-in support for more than three dimensions.
An ND-Range has a global and local components.

- **global-range** describes the total number of work-items in each dimension
- **local-range** describes number of work-items in a work-group in a dimension
Each invocation of a kernel is based on a particular work-item.

- This is in the 'execution space' which can be different that we think about the data. More on that in an example later in this training module.
Information on the work-item is available to the kernel via queries.

<table>
<thead>
<tr>
<th>Global</th>
<th>Range</th>
<th>ID</th>
<th>Linear ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>{12, 8, 12}</td>
<td>{6, 1, 0}</td>
<td>588</td>
</tr>
<tr>
<td>Group</td>
<td>{12, 8, 1}</td>
<td>{6, 1, 0}</td>
<td>49</td>
</tr>
<tr>
<td>Local</td>
<td>{1, 1, 12}</td>
<td>{0, 0, 0}</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subgroup</th>
<th>Group Range</th>
<th>ID</th>
<th>Local Range</th>
<th>Linear ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subgroup</td>
<td>1</td>
<td>{0}</td>
<td>{12}</td>
<td>0</td>
</tr>
</tbody>
</table>

Output from module03/explore12 sample program.
§3. WHERE AND HOW TO GET AND USE DPC++, ETC.
DPC++ Programs

Execution Model

Where and how to get and use DPC++, etc.

id, item, nd_item

Lab exercise: VADD on Various Devices

Host/Accelerator Model

Lab exercise: Stencil

Module 3 draws to a close
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?3
EASIEST - USE THE PREBUILT DPC++ WITH COMPLETE ONEAPI TOOLKITS

- DevCloud
- Download Toolkits

You'll want oneAPI toolkits, even if you build your own DPC++ compiler.
BUILD FROM OPEN SOURCE, EASY LINUX OR WINDOWS

https://tinyurl.com/openSYCL

Branch: sycl → llvm / sycl / doc / Get Started With SYCL Compiler.md

Prerequisites

- **git** - [https://git-scm.com/downloads](https://git-scm.com/downloads)
- **cmake** version 3.2 or later - [http://www.cmake.org/download/](http://www.cmake.org/download/)
- **C++ compiler**
  - Linux: **gcc** version 5.1.0 or later (including libstdc++) - [https://gcc.gnu.org/install/](https://gcc.gnu.org/install/)
  - Windows: **Visual Studio** version 15.7 preview 4 or later - [https://visualstudio.microsoft.com/downloads/](https://visualstudio.microsoft.com/downloads/)
```bash
export SYCL_HOME=/export/home/sycl_workspace
mkdir $SYCL_HOME

cd $SYCL_HOME
git clone https://github.com/intel/llvm -b sycl
mkdir $SYCL_HOME/build

cd $SYCL_HOME/build

cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD="X86" \ 
-DLLVM_EXTERNAL_PROJECTS="llvm-spirv;sycl" \ 
-DLLVM_ENABLE_PROJECTS="clang;llvm-spirv;sycl" \ 
-DLLVM_EXTERNAL_SYCL_SOURCE_DIR=$SYCL_HOME/llvm/sycl \ 
-DLLVM_EXTERNAL_LLVM_SPIRV_SOURCE_DIR=$SYCL_HOME/llvm/llvm-spirv \ 
$SYCL_HOME/llvm/llvm

make -j`nproc` sycl-toolchain

export PATH=$SYCL_HOME/build/bin:$PATH
export LD_LIBRARY_PATH=$SYCL_HOME/build/lib:$LD_LIBRARY_PATH

clang++ -fsycl foo.cpp
```

https://tinyurl.com/openSYCL
DPC++ implements cross-platform data parallelism support (extends C++).

- Write `kernels`
- Control when/where/how they might be accelerated
§4. ID, ITEM, ND_ITEM
1. DPC++ Programs
2. Execution Model
3. Where and how to get and use DPC++, etc.
4. id, item, nd_item
5. Lab exercise: VADD on Various Devices
6. Host/Accelerator Model
7. Lab exercise: Stencil
8. Module 3 draws to a close
code for this module: http://tinyurl.com/oneapimodule?3
$ ssh devcloud
...login to the devcloud...

$ wget tinyurl.com/oneapimodule?3 -O 3.tz
... note the -O option does use a capital letter O...
$ tar xvfz 3.tz
...fetch and unpack code I'll be playing with for module 3...

$ pbsnodes -l free
...see if many are free...
...if only a handful are free... please BE POLITE and use batch...
...training module 2 (please WATCH) explains all this...

$ qsub -I -l nodes=1:ppn=2
... I am using this because lots of nodes are free and available...
// Set up SYCL device and queue.
queue q = queue();

// ultimately we will be 4 x 4 x 4 in 3D
const uint32_t D = 4;

std::vector<int> x(D*D*D);
std::vector<int> y(D*D*D);
std::vector<int> z(D*D*D);

std::fill(x.begin(), x.end(), 7);
std::fill(y.begin(), y.end(), 8);
std::fill(z.begin(), z.end(), 9);

{  
    // buffers for device access to x[], y[], and z[]
    buffer<int,1> x_buf(x.data(), range<1>(D*D*D));
    buffer<int,1> y_buf(y.data(), range<1>(D*D*D));
    buffer<int,1> z_buf(z.data(), range<1>(D*D*D));

    q.submit([&](handler& cgh) {  
        auto xx = x_buf.get_access<access::mode::read_write>(cgh);
        auto yy = y_buf.get_access<access::mode::read_write>(cgh);
        auto zz = z_buf.get_access<access::mode::read_write>(cgh);
    });
}
```cpp
ct.parallel_for<class foo>(range<1>{D*D*D}, [=](id<1> item) {
    xx[ 3 ] = 3;
    yy[ 4 ] = 4;
    zz[ 5 ] = 5;
});
```

**explore1.cpp**

**sample output (output should not vary)**

```
x: 7777777777777777777777777777777777777777777777777777777777777777
y: 8888888888888888888888888888888888888888888888888888888888888888
z: 9999999999999999999999999999999999999999999999999999999999999999
```

```cpp
cgh.parallel_for<class foo>(range<1>{D*D*D}, [=](id<1> item) {
  xx[ item[0] - 1 ] = 3;
  yy[ item[0] ] = item[0] % 10;
  zz[ item[0] + 1 ] = 5;
});
```

- `explore2.cpp`
- `sample output (output may vary, if any (due to error))`
- `NOTE: simply compiling with -g helps get errors explained (therefore, Makefile has -g)`
```cpp
cgh.parallel_for<class foo1D>(range<1>{D*D*D}, [=](id<1> item) {
   if (item[0] > 0)
      xx[ item[0] - 1 ] = 3;
   yy[ item[0] ] = flag ? -item[0] : item[0] % 10;
   if (item[0] < (D*D*D+1))
      zz[ item[0] + 1 ] = 5;
});

$./explore3 (no arguments)

0 1 2 3 4 5 6
.123456789.123456789.123456789.123456789.123456789.123456789.123
x: 3333333333333333333333333333333333333333333333333333333333333333
y: 01234567890123456789012345678901234567890123456789012345678901234567890123
z: 9555555555555555555555555555555555555555555555555555555555555555

$./explore3 x

0 1 2 3 4 5 6
.123456789.123456789.123456789.123456789.123456789.123456789.123
x: 3333333333333333333333333333333333333333333333333333333333333333
z: 9555555555555555555555555555555555555555555555555555555555555555
```

- **explore3.cpp**
- **sample outputs (output should not vary)**
if (flag & 1) {
    cgh.parallel_for<class foo1D>(range<1>{D*D*D}, [=](id<1> item) {
        xx[ item[0] ] = 1;
    });
}
if (flag & 2) {
    cgh.parallel_for<class foo2D>(range<2>{D,D*D}, [=](id<2> item) {
        yy[ item[0] + D*item[1] ] = 2;
    });
}
if (flag & 4) {
    cgh.parallel_for<class foo3D>(range<3>{D,D,D}, [=](id<3> item) {
        zz[ item[0] + D*(item[1]+item[2]*D) ] = 3;
    });
}

$ ./explore4 7

0  1  2  3  4  5  6
.123456789.123456789.123456789.123456789.123456789.123456789.123
x: 777777777777777777777777777777777777777777777777777777777
y: 888888888888888888888888888888888888888888888888888888888
z: 333333333333333333333333333333333333333333333333333333333

why this output?
explore4.cpp

sample outputs (output may vary based on implementation)
```cpp
if (flag & 1) {
    cgh.parallel_for<class foo1D>(range<1>{D*D*D}, [=](id<1> item) {
        xx[ item[0] ] = 1;
    });
}

if (flag & 2) {
    cgh.parallel_for<class foo2D>(range<2>{D,D*D}, [=](id<2> item) {
        yy[ item[0] + D*item[1] ] = 2;
    });
}

if (flag & 4) {
    cgh.parallel_for<class foo3D>(range<3>{D,D,D}, [=](id<3> item) {
        zz[ item[0] + D*(item[1]+item[2]*D) ] = 3;
    });
}
```

```
$ ./explore4 7
```

```
x: 7777777777777777777777777777777777777777777777777777777777777777
y: 8888888888888888888888888888888888888888888888888888888888888888
z: 3333333333333333333333333333333333333333333333333333333333333333
```
```
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {

  ...

  queue myQueue{...};
  ...

  myQueue.submit([&](handler &cgh) {
    // accessors (for connecting to memory via buffers)
    // kernel defined here (with lambda -
    // by value captures only)
  });
}
```

- Queue accepts work requests as submissions.
- Highlighted lines are the command group scope.
- Submissions finish asynchronously.
- Only one kernel (work described in a lambda) per submit!
explore4.cpp

if we choose only one kernel, everything works

if we choose more, that's illegal - and not defined at all
The first code was a failed attempt to loop a bunch of times for checking performance. Oooops. 😊
Dimensions are numbered 0, 1, 2. \( x[0] \ y[0][1] \ z[0][1][2] \)
```cpp
q.submit([&](handler& cgh) {
    ...
    cgh.parallel_for<class foo1D>(range<1>{D*D}, [=](id<1> item) {
        xx[ item[0] ] = 1;
    });

    printit;

    q.submit([&](handler& cgh) {
        ...
        cgh.parallel_for<class foo2D>(range<2>{D,D*D}, [=](id<2> item) {
            xx[ item[0] + D*item[1] ] = 2;
        });

        printit;

    q.submit([&](handler& cgh) {
        ...
        cgh.parallel_for<class foo3D>(range<3>{D,D,D}, [=](id<3> item) {
            xx[ item[0] + D*(item[1]+item[2]*D) ] = 3;
        });

    })

    printit;
});
```

**explore5.cpp**

**Will this work?**
Note: actual output is a little more verbose, condensed here for viewing!

```
$ ./explore5
x: 1111111111111111111111111111111111111111111111111111111111111111
x: 1111111111111111111111111111111111111111111111111111111111111111
x: 3333333333333333333333333333333333333333333333333333333333333333
$ ./explore5
x: 1111111111111111111111111111111111111111111111111111111111111111
x: 2222222222222222222222222222222222222222222222222222222222222222
x: 3333333333333333333333333333333333333333333333333333333333333333
$ ./explore5
x: 7777777777777777777777777777777777777777777777777777777777777777
x: 7777777777777777777777777777777777777777777777777777777777777777
x: 3333333333333333333333333333333333333333333333333333333333333333
Above runs on default device (GPU when I ran these) were inconsistent.

$ ./explore5 h
x: 1111111111111111111111111111111111111111111111111111111111111111
x: 2222222222222222222222222222222222222222222222222222222222222222
x: 3333333333333333333333333333333333333333333333333333333333333333
$ ./explore5 h
x: 1111111111111111111111111111111111111111111111111111111111111111
x: 2222222222222222222222222222222222222222222222222222222222222222
x: 3333333333333333333333333333333333333333333333333333333333333333
When run on the host device ('h' option above)... output were always correct (I tried many many times!).
```
{ // buffers...
q.submit([&](handler& cgh) {
  ...
});
}
printf
{
  // buffers...
q.submit([&](handler& cgh) {
    ...
});
}
printf
{
  // buffers...
q.submit([&](handler& cgh) {
    ...
});
}
printf

- explore6.cpp
- Forced synchronization is one possible fix.
explores6.cpp

output is deterministic

Yeah!
std::cout will not work, and is not permitted, in device code.
Fortunately, SYCL provides a stream class which is useful for
sending messages to standard output from device code.
This can be very useful for debugging!
SYCL 1.2.1r5 section 4.12 "Stream class"

```cpp
q.submit([&](handler &cgh) {
    cl::sycl::stream kernelout(512*D*D*D*1024, 512, cgh);
    ...
    cgh.parallel_for<class foo1D>(range<1>{D*D*D},
        [=](id<1> item) {
            xx[item[0]] = 1;
            kernelout << "Hello:" << item[0] << cl::sycl::endl;
        });
});
```

- explore7.cpp
- printing from a kernel is useful
- stream is implemented to hold lines together
- lines are delivered asynchronously, which will interleave (non-deterministic ordering)
- explicit use of cl::sycl:: avoids confusion with std::
Hello: 0
Hello: 1
Hello: 2
Hello: 3
Hello: 4
Hello: 5
Hello: 6
Hello: 7
Hello: 8
Hello: 9
Hello: 10
Hello: 11
Hello: 12
Hello: 13
Hello: 14
Hello: 15
Hello: 16
Hello: 17
Hello: 18
Hello: 19
Hello: 20
Hello: 21
Hello: 22
Hello: 23
Hello: 24
Hello: 25
Hello: 26
Hello: 27
Hello: 28
Hello: 29
Hello: 30
Hello: 31
Hello: 32
Hello: 33
Hello: 34
Hello: 35
Hello: 36
Hello: 37
Hello: 38
Hello: 39
Hello: 40
Hello: 41
Hello: 42
Hello: 43
Hello: 44
Hello: 45
Hello: 46
Hello: 47
Hello: 48
Hello: 49
Hello: 50
Hello: 51
Hello: 52
Hello: 53
Hello: 54
Hello: 55
Hello: 56
Hello: 57
Hello: 58
Hello: 59
Hello: 60
Hello: 61
Hello: 62
Hello: 63

Sample output (output may vary in ordering)
q.submit([&](handler& cgh) {
    ...
    cgh.parallel_for<class foo1D>(range<1>{D*D*D},
    [=](id<1> item) { xx[ item[0] ] = 1;
        kernelout << "1D(" ...; }); });
printit
q.submit([&](handler& cgh) {
    ...
    cgh.parallel_for<class foo2D>(range<2>{D,D*D},
    [=](id<2> item) { xx[ item[0] + D*item[1] ] = 2;
        kernelout << "2D(" ...; }); });
printit
q.submit([&](handler& cgh) {
    ...
    cgh.parallel_for<class foo3D>(range<3>{D,D,D},
    [=](id<3> item) {
        xx[ item[0] + D*(item[1]+item[2]*D) ] = 3;
        kernelout << "3D(" ...; }); });
printit

▷ explore8.cpp
▷ more observations

oneAPI module 3: DPC++ Fundamentals, Part 1 of 2
explore8.cpp

sample output
(output may vary - ordering and assignments)
explore8.cpp

sample output (output may vary - ordering and assignments)
explore8.cpp

sample output
(output may vary - ordering and assignments)
q.submit([&](handler& cgh) {
...

cgh.parallel_for<class foo3Did>(range<3>{D,D,D},
[
]=)(id<3> item) {
...
q.submit([&](handler& cgh) {
...

cgh.parallel_for<class foo3Ditem>(range<3>{D,D,D},
[
]=)(item<3> item) {
...
q.submit([&](handler& cgh) {
...

cgh.parallel_for<class foo3Dnd_item>(nd_range<3>({D,D,D},{2,2,2}),
[
]=)(nd_item<3> item) {
...
 explores9.cpp
id<3>
sample output
(output may vary - ordering and assignments)
<table>
<thead>
<tr>
<th>3D(0,0,0)</th>
<th>3D(2,2,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D(1,2,0)</td>
<td>3D(2,1,1)</td>
</tr>
<tr>
<td>3D(1,2,1)</td>
<td>3D(2,2,0)</td>
</tr>
<tr>
<td>3D(1,1,3)</td>
<td>3D(2,2,1)</td>
</tr>
<tr>
<td>3D(0,2,2)</td>
<td>3D(2,2,0)</td>
</tr>
<tr>
<td>3D(0,2,3)</td>
<td>3D(2,3,0)</td>
</tr>
</tbody>
</table>

```
3D(0,0,0)
3D(0,0,1)
3D(0,0,2)
3D(0,0,3)
3D(0,1,0)
3D(0,1,1)
3D(0,1,2)
3D(0,1,3)
3D(2,0,0)
3D(1,0,0)
3D(2,0,1)
3D(1,0,1)
3D(1,0,2)
3D(1,0,3)
3D(1,1,0)
3D(1,1,1)
3D(1,1,2)
3D(1,1,3)
3D(2,0,2)
3D(2,0,3)
3D(2,1,0)
3D(2,1,1)
3D(2,1,2)
3D(2,1,3)
3D(1,2,0)
3D(1,2,1)
3D(1,2,2)
3D(1,2,3)
3D(1,3,0)
3D(1,3,1)
3D(1,3,2)
3D(1,3,3)
3D(2,2,0)
3D(2,2,1)
3D(2,2,2)
3D(3,0,0)
3D(3,0,1)
```

**explore9.cpp**

**item<3>**

**sample output**

(output may vary - ordering and assignments)
explore9.cpp

nd_item<3>

sample output (output may vary - ordering and assignments)
cgh.parallel_for<class foo3Dnd_item>(nd_range<3>({D,D,D},{2,2,2}), [=](nd_item<3> item) {
xx[ item.get_global_id()[0] +
D*(item.get_global_id()[1]+item.get_global_id()[2]*D) ] = 5;
kernou << "3D" << item.get_local_id()[0] << "," << item.get_local_id()[1] << " R<" << item.get_local_range()[0] << "," << item.get_local_range()[1] << "," << item.get_local_range()[2] << "> LLid:
<< item.get_global_id()[0] << "," << item.get_global_id()[1] << ","
<< item.get_global_id()[2] << " ) GR<" << item.get_global_range()[0] << "," << item.get_global_range()[1] << "," << item.get_global_range()[2] << "> Glid:
<< item.get_global_linear_id() << cl::sycl::endl;
});
printit

cgh.parallel_for<class foo2Dnd_item>(nd_range<2>({D,D},{2,2}), [=](nd_item<2> item) {
xx[ item.get_global_id()[0] + D*item.get_global_id()[1] ] = 8;
printit

cgh.parallel_for<class foo1Dnd_item>(nd_range<1>({D*D+D},{2}), [=](nd_item<1> item) {
xx[ item.get_global_id()[0] ] = 2;

explore10.cpp
3D, 2D, or 1D... it's the same data underneath
oneAPI module 3: DPC++ Fundamentals, Part 1 of 2

... 3D(0,1,0) R<2,2,2> Llid:2 Global(0,3,2) GR<4,4,4> GLid:14 group:(0,1,1) R<2,2,2> Grpid:3
3D(1,1,0) R<2,2,2> Llid:6 Global(1,3,2) GR<4,4,4> GLid:30 group:(0,1,1) R<2,2,2> Grpid:3
3D(0,0,1) R<2,2,2> Llid:1 Global(0,2,3) GR<4,4,4> GLid:11 group:(0,1,1) R<2,2,2> Grpid:3
3D(1,0,1) R<2,2,2> Llid:5 Global(1,2,3) GR<4,4,4> GLid:27 group:(0,1,1) R<2,2,2> Grpid:3
3D(0,1,1) R<2,2,2> Llid:3 Global(0,3,3) GR<4,4,4> GLid:15 group:(0,1,1) R<2,2,2> Grpid:3
3D(1,1,1) R<2,2,2> Llid:7 Global(1,3,3) GR<4,4,4> GLid:31 group:(0,1,1) R<2,2,2> Grpid:3
3D(0,0,0) R<2,2,2> Llid:0 Global(2,2,2) GR<4,4,4> GLid:42 group:(1,0,1) R<2,2,2> Grpid:3
3D(1,0,0) R<2,2,2> Llid:4 Global(3,2,2) GR<4,4,4> GLid:58 group:(1,0,1) R<2,2,2> Grpid:3
3D(0,1,0) R<2,2,2> Llid:2 Global(2,3,2) GR<4,4,4> GLid:46 group:(1,0,1) R<2,2,2> Grpid:3
3D(1,1,0) R<2,2,2> Llid:6 Global(3,3,2) GR<4,4,4> GLid:62 group:(1,0,1) R<2,2,2> Grpid:3
3D(0,0,1) R<2,2,2> Llid:1 Global(2,2,3) GR<4,4,4> GLid:43 group:(1,0,1) R<2,2,2> Grpid:3
3D(1,0,1) R<2,2,2> Llid:5 Global(3,2,3) GR<4,4,4> GLid:59 group:(1,0,1) R<2,2,2> Grpid:3
3D(0,1,1) R<2,2,2> Llid:3 Global(2,3,3) GR<4,4,4> GLid:47 group:(1,0,1) R<2,2,2> Grpid:3
3D(1,1,1) R<2,2,2> Llid:7 Global(3,3,3) GR<4,4,4> GLid:63 group:(1,0,1) R<2,2,2> Grpid:3

×: 5555555555555555555555555555555555555555555555555555555555555555
0 1 2 3 4 5 6

explore10.cpp

sample output (output may vary - ordering and assignments)
...
...  

1D(0)  R<2>  Llid:0  Global(54)  GR<64>  GLid:54  group:(27)  R<32>  Grpid:27  
1D(1)  R<2>  Llid:1  Global(55)  GR<64>  GLid:55  group:(27)  R<32>  Grpid:27  
1D(0)  R<2>  Llid:0  Global(56)  GR<64>  GLid:56  group:(28)  R<32>  Grpid:28  
1D(1)  R<2>  Llid:1  Global(57)  GR<64>  GLid:57  group:(28)  R<32>  Grpid:28  
1D(0)  R<2>  Llid:0  Global(58)  GR<64>  GLid:58  group:(29)  R<32>  Grpid:29  
1D(1)  R<2>  Llid:1  Global(59)  GR<64>  GLid:59  group:(29)  R<32>  Grpid:29  
1D(0)  R<2>  Llid:0  Global(60)  GR<64>  GLid:60  group:(30)  R<32>  Grpid:30  
1D(1)  R<2>  Llid:1  Global(61)  GR<64>  GLid:61  group:(30)  R<32>  Grpid:30  
1D(0)  R<2>  Llid:0  Global(62)  GR<64>  GLid:62  group:(31)  R<32>  Grpid:31  
1D(1)  R<2>  Llid:1  Global(63)  GR<64>  GLid:63  group:(31)  R<32>  Grpid:31

0  1  2  3  4  5  6
.123456789.123456789.123456789.123456789.123456789.123456789.123456789.123
x: 22222222222222222222222222222222222222222222222222222222222222222222

explore10.cpp
sample output (output may vary - ordering and assignments)
q.submit([&](handler& cgh) {
...

// 2 dimensional execution shape used to
// give out a row at a time from a
// three dimensional data shape

cgh.parallel_for<class foo2Dnd_item>(nd_range<2>({D,D},{2,2}),
    [=](nd_item<2> item) {
    kernelout << "x: ";
    for (int i = 0; i < 64; ++i) kernelout << xx[i];
    kernelout << cl::sycl::endl;
    xx[ item.get_global_linear_id()*D+0 ] = 1;
    xx[ item.get_global_linear_id()*D+1 ] = 2;
    xx[ item.get_global_linear_id()*D+2 ] = 3;
    xx[ item.get_global_linear_id()*D+3 ] = 4;
    kernelout << "2D( " << ... << cl::sycl::endl;
    });
});
printit

▶ explore11.cpp
▶ the execution shape need not match the data shape
explore11.cpp

in this example, we purposefully handle a row at a time ourselves

there are multiple ways to do this

the KEY really is to think about the mappings we want
ND RANGE - SELECT FOR PERFORMANCE OR FAMILIARITY
```cpp
#include <cgh.

parallel_for<
class foo3Dnd_item
>(nd_range<3>{D1,D2,D3}, [=](nd_item<3> item) {
    if ((item.get_global_id()[0] == 6) &&
        (item.get_global_id()[1] == 1) &&
        (item.get_global_id()[2] == 0)) {
        intel::sub_group sg = item.get_sub_group();
        kernelout << "global range "
            << item.get_global_range() << cl::sycl::endl;
        kernelout << "global id "
            << item.get_global_id() << cl::sycl::endl;
        kernelout << "global linear id "
            << item.get_global_linear_id() << cl::sycl::endl;
        kernelout << "group range "
            << item.get_group_range() << cl::sycl::endl;
        kernelout << "group id "
            << item.get_group().get_id() << cl::sycl::endl;
        kernelout << "group linear id "
            << item.get_group_linear_id() << cl::sycl::endl;
        kernelout << "local range "
            << item.get_local_range() << cl::sycl::endl;
        kernelout << "local id "
            << item.get_local_id() << cl::sycl::endl;
        kernelout << "local linear id "
            << item.get_local_linear_id() << cl::sycl::endl;
        kernelout << "subgroup group range "
            << sg.get_group_range() << cl::sycl::endl;
        kernelout << "subgroup group id "
            << sg.get_group_id() << cl::sycl::endl;
        kernelout << "subgroup local range "
            << sg.get_local_range() << cl::sycl::endl;
```
$ ./explore12 1
nd_range<3>({12,8,16}{6,2,4})
global range {12, 8, 16}
global id {6, 1, 0}
global linear id 784
group range {2, 4, 4}
group {1, 0, 0}
group linear id 16
local range {6, 2, 4}
local id {0, 1, 0}
local linear id 4

subgroup group range 3
subgroup group id {0}
subgroup local range {16}
subgroup local id {6}
subgroup uniform group range 3
subgroup max local range {16}
```c
$ ./explore12 2
nd_range<3>({12, 8, 16}{3, 2, 4})
global range   {12, 8, 16}
global id       {6, 1, 0}
global linear id 784

group range {4, 4, 4}
group {2, 0, 0}
group linear id 32

local range {3, 2, 4}
local id {0, 1, 0}
local linear id 4

subgroup group range 2
subgroup group id {0}
subgroup local range {16}
subgroup local id {3}
subgroup uniform group range 2
subgroup max local range {16}
```
OneAPI module 3: DPC++ Fundamentals, Part 1 of 2

ND_RANGE - EXPLORE123

$ ./explore12 3
nd_range<3>({12,8,16},{4,4,4})
global range {12, 8, 16}
global id {6, 1, 0}
global linear id 784
group range {3, 2, 4}
group {1, 0, 0}
group linear id 8
local range {4, 4, 4}
local id {2, 1, 0}
local linear id 36

subgroup group range 4
subgroup group id {0}
subgroup local range {16}
subgroup local id {6}
subgroup uniform group range 4
subgroup max local range {16}
The group range multiplied by the local range gives the global range.

```c
$ ./explore12 3
nd_range<3>({12,8,16},{4,4,4})
global range  {12, 8, 16}
global id     {6, 1, 0}
global linear id  784
group range  {3, 2, 4}
group       {1, 0, 0}
group linear id  8
local range  {4, 4, 4}
local id     {2, 1, 0}
local linear id  36

subgroup group range  4
subgroup group id     {0}
subgroup local range  {16}
subgroup local id     {6}
subgroup uniform group range  4
subgroup max local range  {16}
```
$ ./explore12 4
nd_range<3>(\{12,8,16\}\{3,4,8\})
global range \{12, 8, 16\}
global id \{6, 1, 0\}
global linear id 784
group range \{4, 2, 2\}
group \{2, 0, 0\}
group linear id 8
local range \{3, 4, 8\}
local id \{0, 1, 0\}
local linear id 8

subgroup group range 6
subgroup group id \{0\}
subgroup local range \{16\}
subgroup local id \{3\}
subgroup uniform group range 6
subgroup max local range \{16\}
ND_RANGE - EXECUTION, DATA, THINKING

![Diagram showing ND-Range, Work-group, Sub-group, and Work-item with dimensions and work-items](image.png)
§5. LAB EXERCISE: VADD ON VARIOUS DEVICES
1. DPC++ Programs
2. Execution Model
3. Where and how to get and use DPC++, etc.
4. id, item, nd_item
5. **Lab exercise: VADD on Various Devices**
6. Host/Accelerator Model
7. Lab exercise: Stencil
8. Module 3 draws to a close
Follow the directions in the Lab-VADD subdirectory of the module03 directory.

The instructions will help you use a Jupyter Notebook interface to DevCloud to learn from the VADD example.
DPC++ implements cross-platform data parallelism support (extends C++).

- Write `kernels`
- Control when/where/how they might be accelerated
§6. HOST/ACCELERATOR MODEL
DPC++ Programs
Execution Model
Where and how to get and use DPC++, etc.
id, item, nd_item
Lab exercise: VADD on Various Devices
Host/Accelerator Model
Lab exercise: Stencil
Module 3 draws to a close
Kernel functions are executed as work-items, on devices.

- Like a thread, yet very different from a C++ thread
- A work-item cannot synchronize with another work-item (achieve by kernel must end, submit another kernel invocation). Wait - sub-groups and work-groups offer synchronizations.
  - could be a OS thread
  - but it could be done on a GPU element
  - or it could be processed in an FPGA
  - or it could be processed in a DSP
  - or it could be processed in an AI accelerator

Such flexibility for target, brings some restrictions and responsibilities.
RESTRICTIONS ON KERNEL CODE

Supported include:

▷ lambdas
▷ operator overloading
▷ templates
▷ classes
▷ static polymorphism
▷ share data with host via accessors
▷ read-only values of host variables subject via lambda captures

Not supported:

▷ dynamic polymorphism
▷ dynamic memory allocations
▷ static variables
▷ function pointers
▷ pointer structure members
▷ runtime type information
▷ exception handling
printf("Value at start: myArray[42] is %d.\n", myArray[42]);
{
    queue myQ; /* use defaults today */
    /* (queue parameters possible – future topic) */
    range<1> mySize{SIZE};
    buffer<int, 1> bufferA(myArray.data(), mySize);

    myQ.submit([&](handler &myHandle) {
        auto deviceAccessorA = bufferA.get_access<access::mode::read_write>(myHandle);
        myHandle.parallel_for<class uniqueID>(mySize,
            [=](id<1> index)
            {
                deviceAccessorA[index] *= 2;
            }
        );
    });

    printf("Value at finish: myArray[42] is %d.\n", myArray[42]);

▷ All code runs on the host, except kernel scope code
▷ kernel scope lines 11-16
Single source code

- Regular CPU code
- Device kernels

Runs natively on CPU

Submitted to `sycl::queue`

SYCL devices (e.g. GPU, FPGA, CPU)
FIVE METHODS TO STEER WORK TO A DEVICE

- Method#1: Just run on any device
- Method#2: host device for dev & dbg
- Method#3: use accelerator (e.g., GPU)
- Method#4: multiple devices
- Method#5: very specific (custom)

Debugging using Method #2 is an attractive option before moving on to Methods 3+. 
METHOD#1: JUST RUN ON ANY DEVICE

```c
queue myQueue;

// equivalent to:
queue myQueue( default_selector );

// either one lets the implementation choose
// that will be the host if there is no
// accelerator available when the program runs
// if there is an accelerator, the implementation will pick
// there is no standard on what it will pick
```

Method#1
- Running device code but not caring which device runs the code.
- Potentially a first step in application development, because it requires the least code.
Method#2

- Explicitly run device code on the host device.
- Typical used for debugging and is guaranteed to be always available on any system.
- `explore5.cpp` showed how this can mask memory move issues, a pro and a con.
**METHOD#3: USE ACCELERATOR (E.G., GPU)**

```cpp
queue myQueue( default_selector );
queue myQueue( host_selector );
queue myQueue( cpu_selector );
queue myQueue( gpu_selector );
queue myQueue( accelerator_selector );
```

// DPC++ only...
#include "CL/sycl/intel/fpga_extensions.hpp"
queue myQueue( intel::fpga_selector );

// When a queue is created, if no device matching the // requested type exists, then the selector throws // a sycl::runtime_error exception.
//
// Yes, I said we'd want try/catch in our real programs.
// We'll get to that in a future training module!

**Method#3**
Explicitly dispatching device code to:

- default (Method #1)
- host (Method #2)
- CPU (a device that identifies itself as a CPU)
- GPU (a device that identifies itself as a GPU)
- accelerator (a device that identifies itself as an accelerator, includes FPGAs)
- DPC++ has an extension to specifically request an FPGA
// available with: wget tinyurl.com/oneapimodule?2 -O 2.tz
#include <CL/sycl.hpp>

int main() {
    unsigned number = 0;
    auto MyPlatforms = cl::sycl::platform::get_platforms();

    /* Loop through the platforms SYCL can find
       there is always ONE */
    for (auto &OnePlatform : MyPlatforms) {
        std::cout << ++number << " found..."
                  << std::endl
                  << "Platform: "
                  << OnePlatform.get_info<cl::sycl::info::platform::name>()
                  << std::endl;
    }

    /* Loop through the devices SYCL can find
       there is always ONE */
    auto MyDevices = OnePlatform.get_devices();
    for (auto &OneDevice : MyDevices ) {
        std::cout << " Device: "
                  << OneDevice.get_info<cl::sycl::info::device::name>()
                  << std::endl;
    }
    std::cout << std::endl;
}
$ make verycurious
dpcpp verycurious.cpp -o verycurious

$ ./verycurious
1 found...
Platform:
  cl::sycl::info::platform::profile is 'EMBEDDED_PROFILE'
  cl::sycl::info::platform::version is 'OpenCL 1.0 Intel(R) FPGA SDK for OpenCL(TM), Version 19.2'
  cl::sycl::info::platform::name is 'Intel(R) FPGA Emulation Platform for OpenCL(TM)'
  cl::sycl::info::platform::vendor is 'Intel(R) Corporation'
  cl::sycl::info::platform::extensions is :
    1) cl_khr_icd
    2) cl_khr_byte_addressable_store
    3) cl_intel_fpga_host_pipe
    4) cles_khr_int64
    5) cl_khr_il_program
Device: Intel(R) FPGA Emulation Device
  is_host() = No
  is_cpu() = No
  is_gpu() = No
2 found...

Platform:

- `cl::sycl::info::platform::profile` is 'FULL_PROFILE'
- `cl::sycl::info::platform::version` is 'OpenCL 2.1 '
- `cl::sycl::info::platform::name` is 'Intel(R) OpenCL HD Graphics'
- `cl::sycl::info::platform::vendor` is 'Intel(R) Corporation'
- `cl::sycl::info::platform::extensions` is:
  1) `cl_khr_3d_image_writes`
  2) `cl_khr_byte_addressable_store`
  3) `cl_khr_fp16`
  4) `cl_khr_depth_images`
  5) `cl_khr_global_int32_base_atomics`
  ...  
  37) `cl_intel_advanced_motion_estimation`
  38) `cl_intel_va_api_media_sharing`

Device: Intel(R) Gen9 HD Graphics NEO
- `is_host()` = No
- `is_cpu()` = No
- `is_gpu()` = Yes
- `is_accelerator()` = No
3 found...

Platform:
- cl::sycl::info::platform::profile is 'FULL_PROFILE'
- cl::sycl::info::platform::version is 'OpenCL 2.1 LINUX'
- cl::sycl::info::platform::name is 'Intel(R) OpenCL'
- cl::sycl::info::platform::vendor is 'Intel(R) Corporation'
- cl::sycl::info::platform::extensions is:
  1) cl_khr_icd
  2) cl_khr_global_int32_base_atomics
  3) cl_khr_global_int32_extended_atomics
  4) cl_khr_local_int32_base_atomics
  5) cl_khr_local_int32_extended_atomics
  ...
  16) cl_khr_fp64
  17) cl_khr_image2d_from_buffer

Device: Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz
- is_host() = No
- is_cpu() = Yes
- is_gpu() = No
- is_accelerator() = No
4 found...
Platform:
  cl::sycl::info::platform::profile is 'FULL PROFILE'
  cl::sycl::info::platform::version is '1.2'
  cl::sycl::info::platform::name is 'SYCL host platform'
  cl::sycl::info::platform::vendor is ''
  cl::sycl::info::platform::extensions is : NO extensions
Device: SYCL host device
  is_host() = Yes
  is_cpu() = No
  is_gpu() = No
  is_accelerator() = No
  cl::sycl::info::device::vendor_id is '32902'
  cl::sycl::info::device::max_compute_units is '12'
  cl::sycl::info::device::max_work_item_dimensions is '3'
  ...
  cl::sycl::info::device::name is 'SYCL host device'
  cl::sycl::info::device::vendor is ''
  cl::sycl::info::device::driver_version is '1.2'
Method#4: MULTIPLE DEVICES

1. `queue myGpuQueue( gpu_selector{} );`
2. `queue myFpgaQueue( intel::fpga_selector{} );`

Method#4

- Dispatching device code to a heterogeneous set of devices, such as a GPU and an FPGA.
- Each device needs its own queue.
- A single queue can never dispatch to multiple devices.
// Defining operator in a class derived from
// sycl::device_selector is all that is required
// to define any complexity of device selection logic.
//
// Here is an example from Chapter 2 in the DPC++ book...
// demonstrating creating arria_selector to select
// an Intel Arria FPGA.

class arria_selector : public device_selector {
  public:
    virtual int operator()(const device &dev) const {
      if (dev.get_info<info::device::name>().find("Arria")
          != std::string::npos &&
          dev.get_info<info::device::vendor>().find("Intel")
          != std::string::npos) {
        return 1;
      } else {
        return -1;
      }
    }
};

Method#5

- Selecting specific devices from a more general class of devices, such as a specific type of FPGA from a collection of available FPGA devices.
§7. LAB EXERCISE: STENCIL
DPC++ Programs
Execution Model
Where and how to get and use DPC++, etc.
id, item, nd_item
Lab exercise: VADD on Various Devices
Host/Accelerator Model
Lab exercise: Stencil
Module 3 draws to a close

Lab exercise: Stencil
Follow the directions in the Lab-Stencil subdirectory of the module03 directory.

We have a small stencil example from Intel's oneAPI example codes, as a real-world application from which to learn.

The instructions will help you use a Jupyter Notebook interface to DevCloud to learn from the stencil example.
§8. MODULE 3 DRAWS TO A CLOSE
Module 3 draws to a close
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?3
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?3
- Do the LAB exercises - cool learning
- Module 4 will discuss:
  - Hierarchical Parallelism
  - Data Management (buffers, USM, synchronization, DAGs)
  - Launching Kernels with dependencies (DAGs, queues, etc.)
- Watch prior modules if you skipped them!