



ONEAPI: DPC++ FUNDAMENTALS

March 2020

AGENDA

- Introduction
- DPC++ “Hello world”
- DPC++ Software Model
 - Platform Model
 - Execution Model
 - Memory Model
 - Kernel Model
- Tips and tricks



WHAT IS DPC++?

The language is:

C++

+

SYCL

<https://www.khronos.org/sycl/>

+

Additional Features

such as...

ndrange subgroups, USM, ordered queue

<https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/>



WHAT IS DPC++?

The implementation is:

Clang

+

LLVM

+

Runtime

<https://github.com/intel/llvm>



DPC++ implements cross-platform data parallelism support (extends C++).

- ▷ Write `kernels`
- ▷ Control when/where/how they might be accelerated



BEFORE WE START

LAMBDA EXPRESSIONS

- A convenient way of defining an anonymous function object right at the location where it is invoked or passed as an argument to a function
- Lambda functions can be used to define kernels in SYCL
- The kernel lambda MUST use copy for all its captures (i.e., [=])

Capture clause

Parameter list

```
#include <algorithm>
#include <cmath>

void absort(float* x, unsigned n) {
    std::sort(x, x + n,
    // Lambda expression
    [(float a, float b)
    {
        return (std::abs(a) < std::abs(b));
    }
    );
}
```

Lambda body



DPC++ PROGRAM OVERVIEW

```
#include <CL/sycl.hpp>
using namespace sycl;

int main(int argc, char *argv[]) {

    ...

    // define buffers!!!
    queue myQueue{...};

    ...

    myQueue.submit([&] (handler &h) {

        // accessors (for connecting to memory via buffers)
        h.parallel_for( range<3>(1024,1024,1024),
                       [=] (id<3> myID) { // kernel function });
    });
}
```

- ▶ All work requests are done via a queue.
- ▶ A queue uniquely attaches to a single device (e.g., GPU, FPGA, AI, CPU, Host).
- ▶ 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!

Optimization Notice

Copyright © 2020, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



COMMON CODE SNIPPET

```
#include <CL/sycl.hpp>

using namespace sycl;

#define dpc_r access::mode::read
#define dpc_w access::mode::write
#define dpc_rw access::mode::read_write
```

Compile with `-fsycl-unnamed-lambda` option

- default starting from Beta 04



DPC++ “HELLO WORLD”: VECTOR ADDITION

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([& (handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=] (id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

dpcpp test.cpp

[Optimization Notice](#)

Copyright © 2020, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Getting started with oneAPI



ANATOMY OF A DPC++ APPLICATION

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Host code

Host code



ANATOMY OF A DPC++ APPLICATION

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Host code

Accelerator
device code

Host code



DPC++ BASICS

```
int main() {  
    float A[1024], B[1024], C[1024];  
    {  
        buffer<float, 1> bufA { A, range<1> {1024} };  
        buffer<float, 1> bufB { B, range<1> {1024} };  
        buffer<float, 1> bufC { C, range<1> {1024} };  
  
        queue q;  
        q.submit([&](handler& h) {  
            auto A = bufA.get_access<dpc_r>(h);  
            auto B = bufB.get_access<dpc_r>(h);  
            auto C = bufC.get_access<dpc_w>(h);  
  
            h.parallel_for(range<1> {1024}, [=](id<1> i) {  
                C[i] = A[i] + B[i];  
            });  
        });  
    }  
    for (int i = 0; i < 1024; i++)  
        std::cout << "C[" << i << "] = " << C[i] << std::endl;  
}
```

Create buffers using
host pointers



DPC++ BASICS

```
int main() {  
    float A[1024], B[1024], C[1024];  
    {  
        buffer<float, 1> bufA { A, range<1> {1024} };  
        buffer<float, 1> bufB { B, range<1> {1024} };  
        buffer<float, 1> bufC { C, range<1> {1024} };  
  
        queue q;  
        q.submit([&](handler& h) {  
            auto A = bufA.get_access<dpc_r>(h);  
            auto B = bufB.get_access<dpc_r>(h);  
            auto C = bufC.get_access<dpc_w>(h);  
  
            h.parallel_for(range<1> {1024}, [=](id<1> i) {  
                C[i] = A[i] + B[i];  
            });  
        });  
    }  
    for (int i = 0; i < 1024; i++)  
        std::cout << "C[" << i << "] = " << C[i] << std::endl;  
}
```


Create a queue to submit work to a device (including host)



DPC++ BASICS

```
int main() {  
    float A[1024], B[1024], C[1024];  
    {  
        buffer<float, 1> bufA { A, range<1> {1024} };  
        buffer<float, 1> bufB { B, range<1> {1024} };  
        buffer<float, 1> bufC { C, range<1> {1024} };  
  
        queue q;  
        q.submit([&](handler& h) {  
            auto A = bufA.get_access<dpc_r>(h);  
            auto B = bufB.get_access<dpc_r>(h);  
            auto C = bufC.get_access<dpc_w>(h);  
  
            h.parallel_for(range<1> {1024}, [=](id<1> i) {  
                C[i] = A[i] + B[i];  
            });  
        });  
    }  
    for (int i = 0; i < 1024; i++)  
        std::cout << "C[" << i << "] = " << C[i] << std::endl;  
}
```

Read/write accessors create dependencies if other kernels or host access buffers.



DPC++ BASICS

```
int main() {  
    float A[1024], B[1024], C[1024];  
    {  
        buffer<float, 1> bufA { A, range<1> {1024} };  
        buffer<float, 1> bufB { B, range<1> {1024} };  
        buffer<float, 1> bufC { C, range<1> {1024} };  
  
        queue q;  
        q.submit([&](handler& h) {  
            auto A = bufA.get_access<dpc_r>(h);  
            auto B = bufB.get_access<dpc_r>(h);  
            auto C = bufC.get_access<dpc_w>(h);  
  
            h.parallel_for(range<1> {1024}, [=](id<1> i) {  
                C[i] = A[i] + B[i];  
            });  
        });  
    }  
    for (int i = 0; i < 1024; i++)  
        std::cout << "C[" << i << "] = " << C[i] << std::endl;  
}
```

Vector addition kernel enqueues a `parallel_for` task.

Pass a **function object/lambda** to be executed by each work-item



DPC++ CONSTRUCTS THAT DESCRIBE PARALLELISM

```
h.single_task(  
    [=] () {  
        // kernel function is executed EXACTLY once on a SINGLE work-item  
    });  
  
h.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)  
    });  
  
h.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)  
    });  
  
h.parallel_for_work_group(  
    range<2>(1024,1024), // using 2D in this example  
    [=] (group<2> grp) {  
        // 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  
    });
```

Basic data parallel

Explicit ND-Range

Hierarchical parallelism



DPC++ BASICS

```
int main() {  
    float A[1024], B[1024], C[1024];  
    {  
        buffer<float, 1> bufA { A, range<1> {1024} };  
        buffer<float, 1> bufB { B, range<1> {1024} };  
        buffer<float, 1> bufC { C, range<1> {1024} };  
  
        queue q;  
        q.submit([&](handler& h) {  
            auto A = bufA.get_access<dpc_r>(h);  
            auto B = bufB.get_access<dpc_r>(h);  
            auto C = bufC.get_access<dpc_w>(h);  
  
            h.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {  
                C[i] = A[i] + B[i];  
            });  
        });  
    }  
    for (int i = 0; i < 1024; i++)  
        std::cout << "C[" << i << "] = " << C[i] << std::endl;  
}
```

SYCL 1.2.1 requires to name the lambda

-fsycl-unnamed-lambda allows to avoid it



DPC++ BASICS

```
int main() {  
    float A[1024], B[1024], C[1024];  
    {  
        buffer<float, 1> bufA { A, range<1> {1024} };  
        buffer<float, 1> bufB { B, range<1> {1024} };  
        buffer<float, 1> bufC { C, range<1> {1024} };  
  
        queue q;  
        q.submit([&](handler& h) {  
            auto A = bufA.get_access<dpc_r>(h);  
            auto B = bufB.get_access<dpc_r>(h);  
            auto C = bufC.get_access<dpc_w>(h);  
  
            h.parallel_for(range<1> {1024}, [=](id<1> i) {  
                C[i] = A[i] + B[i];  
            });  
        });  
    }  
    for (int i = 0; i < 1024; i++)  
        std::cout << "C[" << i << "] = " << C[i] << std::endl;  
}
```


The **range** defines
the iteration space



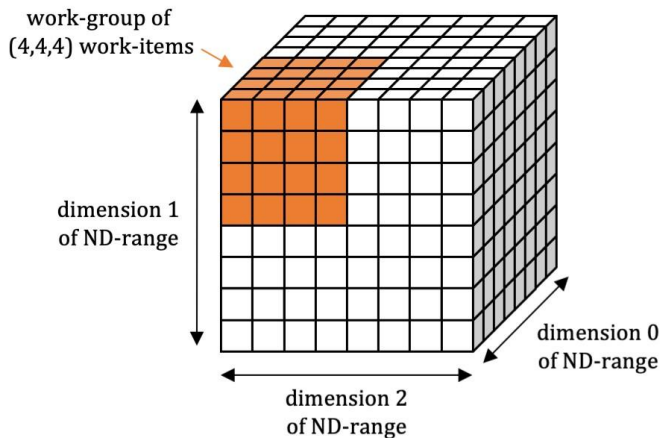
DPC++ BASICS

```
int main() {  
    float A[1024], B[1024], C[1024];  
    {  
        buffer<float, 1> bufA { A, range<1> {1024} };  
        buffer<float, 1> bufB { B, range<1> {1024} };  
        buffer<float, 1> bufC { C, range<1> {1024} };  
  
        queue q;  
        q.submit([&](handler& h) {  
            auto A = bufA.get_access<dpc_r>(h);  
            auto B = bufB.get_access<dpc_r>(h);  
            auto C = bufC.get_access<dpc_w>(h);  
  
            h.parallel_for(range<1> {1024}, [=](id<1> i) {  
                C[i] = A[i] + B[i];  
            });  
        });  
    }  
    for (int i = 0; i < 1024; i++)  
        std::cout << "C[" << i << "] = " << C[i] << std::endl;  
}
```

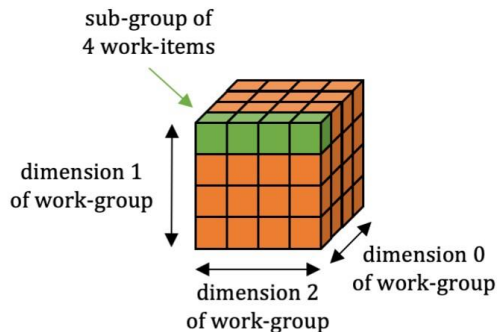
Each iteration
(work-item) will
have a separate
index **id** (i)



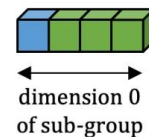
LANGUAGE OF THE HIERARCHY ABSTRACTION (FULL 3D)



ND-Range



Work-group



Sub-group



Work-item

DPC++ vocabulary follows and extends vocabulary of CUDA, OpenCL, SYCL.



DPC++ BASICS

```
int main() {  
    float A[1024], B[1024], C[1024];  
    {  
        buffer<float, 1> bufA { A, range<1> {1024} };  
        buffer<float, 1> bufB { B, range<1> {1024} };  
        buffer<float, 1> bufC { C, range<1> {1024} };  
  
        queue q;  
        q.submit([&](handler& h) {  
            auto A = bufA.get_access<dpc_r>(h);  
            auto B = bufB.get_access<dpc_r>(h);  
            auto C = bufC.get_access<dpc_w>(h);  
  
            h.parallel_for(range<1> {1024}, [=](id<1> i) {  
                C[i] = A[i] + B[i];  
            });  
        });  
    }  
    for (int i = 0; i < 1024; i++)  
        std::cout << "C[" << i << "] = " << C[i] << std::endl;  
}
```



DPC++ BASICS

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Write-buffer is now out-of-scope, so kernel completes and host pointer has consistent view of output.



SIMPLIFIED DPC++ STYLE (BETA 04)

```
std::vector<float> A(1024), B(1024), C(1024);
{
    buffer bufA {A}, bufB {B}, bufC {C};

    queue q;
    q.submit([&](auto &h) {
        auto A = bufA.get_access<dpc_r>(h);
        auto B = bufB.get_access<dpc_r>(h);
        auto C = bufC.get_access<dpc_w>(h);

        h.parallel_for(range(1024), [=](id<1> i) {
            C[i] = A[i] + B[i];
        });
    });
}
```

dpcpp test.cpp -std=c++17



WHERE IS MY “HELLO WORLD” CODE EXECUTED?

DEVICE SELECTOR

Get a device (any device):	<code>queue queue(); // default_selector{}</code>
Get device from class:	<code>queue queue(gpu_selector{}); queue queue(cpu_selector{}); queue queue(host_selector{});</code>

default_selector

- DPC++ runtime scores all the devices and picks one with highest compute power
- Environment variable

```
export SYCL_DEVICE_TYPE=GPU | CPU | HOST
```



DPC++ BASICS

```
int main() {  
    float A[1024], B[1024], C[1024];  
    {  
        buffer<float, 1> bufA { A, range<1> {1024} };  
        buffer<float, 1> bufB { B, range<1> {1024} };  
        buffer<float, 1> bufC { C, range<1> {1024} };  
  
        queue q(gpu_selector{});  
        q.submit([&](handler& h) {  
            auto A = bufA.get_access<dpc_r>(h);  
            auto B = bufB.get_access<dpc_r>(h);  
            auto C = bufC.get_access<dpc_w>(h);  
  
            h.parallel_for(range<1> {1024}, [=](id<1> i) {  
                C[i] = A[i] + B[i];  
            });  
        });  
    }  
    for (int i = 0; i < 1024; i++)  
        std::cout << "C[" << i << "] = " << C[i] << std::endl;  
}
```

Explicit device selector



DPC++ COMPILATION AND EXECUTION

DPC++ COMPILATION FLOW: SINGLE SOURCE CONCEPT

main.cpp:

```
#include <iostream>

int main() {
    const size_t array_size = 16;
    int data[array_size];
    {
        buffer<int, 1> resultBuf{ data, range<1>{array_size} };
        queue q;
        q.submit([&](handler& h) {
            auto resultAcc = resultBuf.get_access<access::mode::write>(h);

            h.parallel_for(range<1>{array_size}, [=](id<1> i) {
                resultAcc[i] = static_cast<int>(i.get(0));
            });
        });
    }
    for (int i = 0; i < array_size; i++) {
        std::cout << "data[" << i << "] = " << data[i] << std::endl;
    }
    return 0;
}
```

dpcpp main.cpp

oneAPI DPC++
Compiler

Standard
Object File
(main.o)

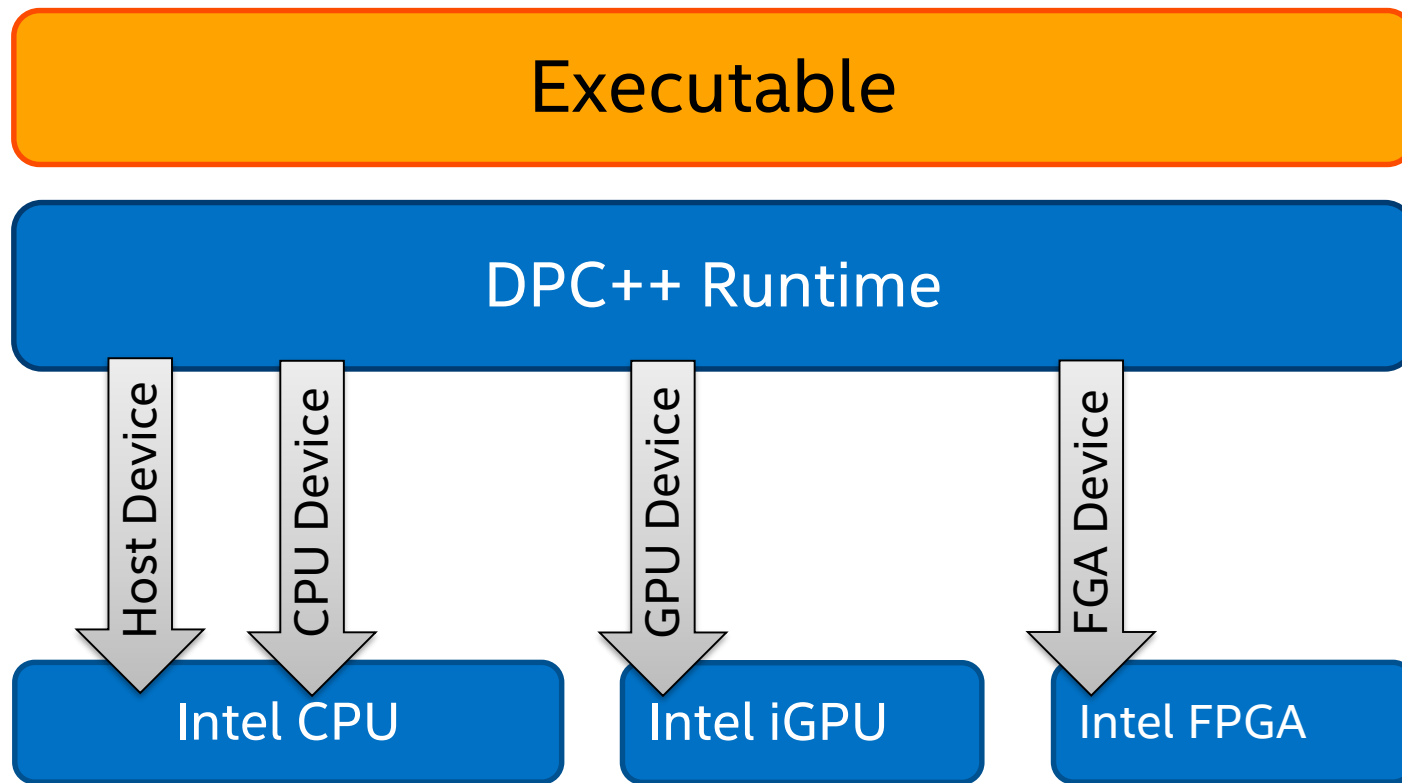
Kernel IR/ISA
(SPIR-V, vISA, ISA)

Standard
Linker

Executable!



DPC++ EXECUTION FLOW



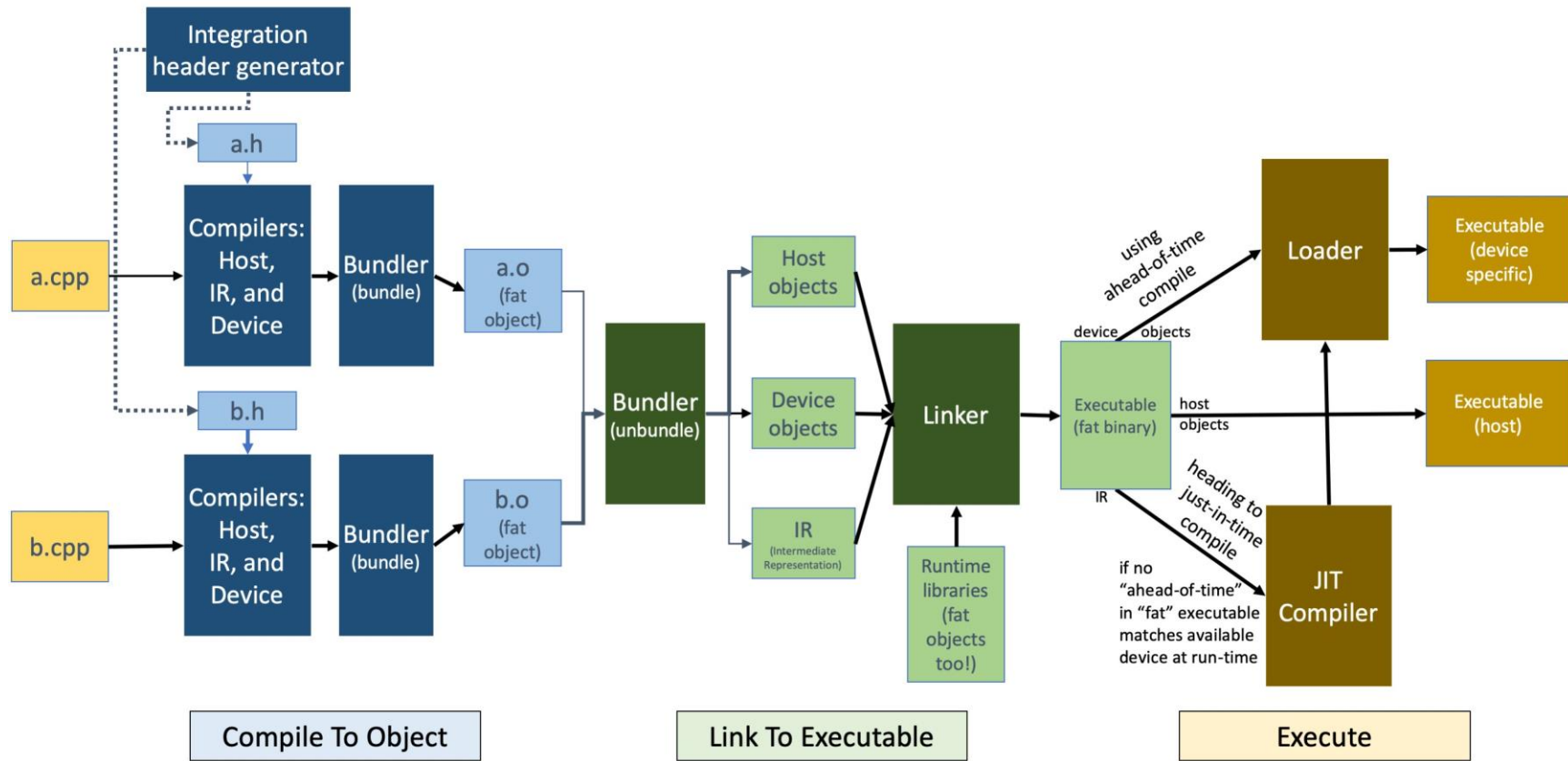
Optimization Notice

Copyright © 2020, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Getting started with oneAPI



DPC++ COMPILATION AND EXECUTION: FULL VERSION



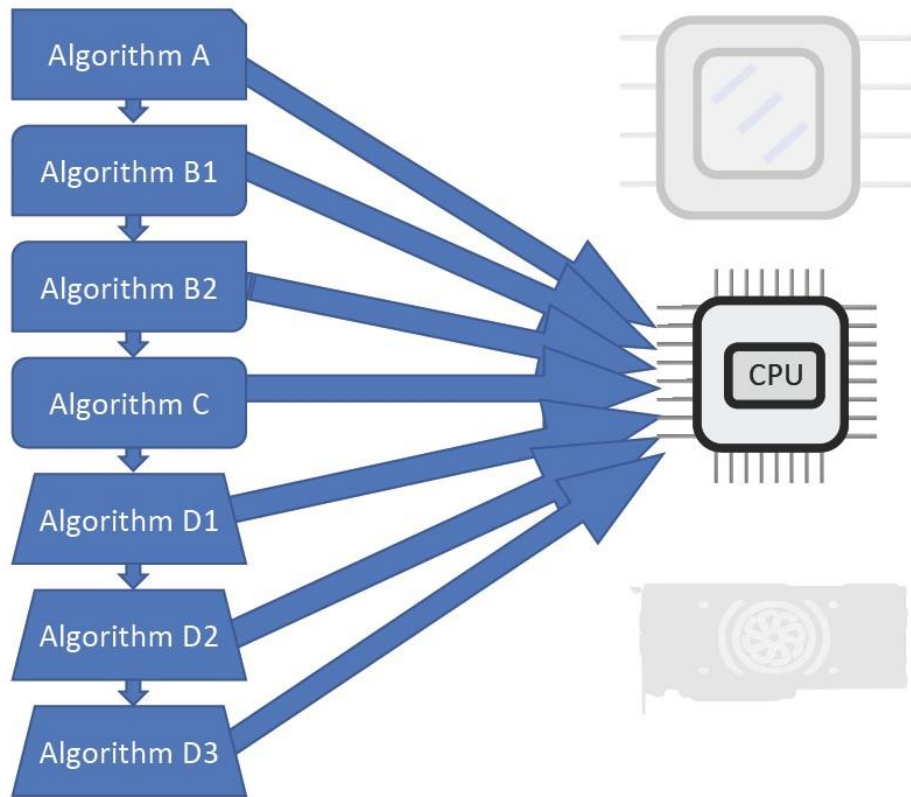
Optimization Notice

Copyright © 2020, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



DPC++ DEVICE SELECTION

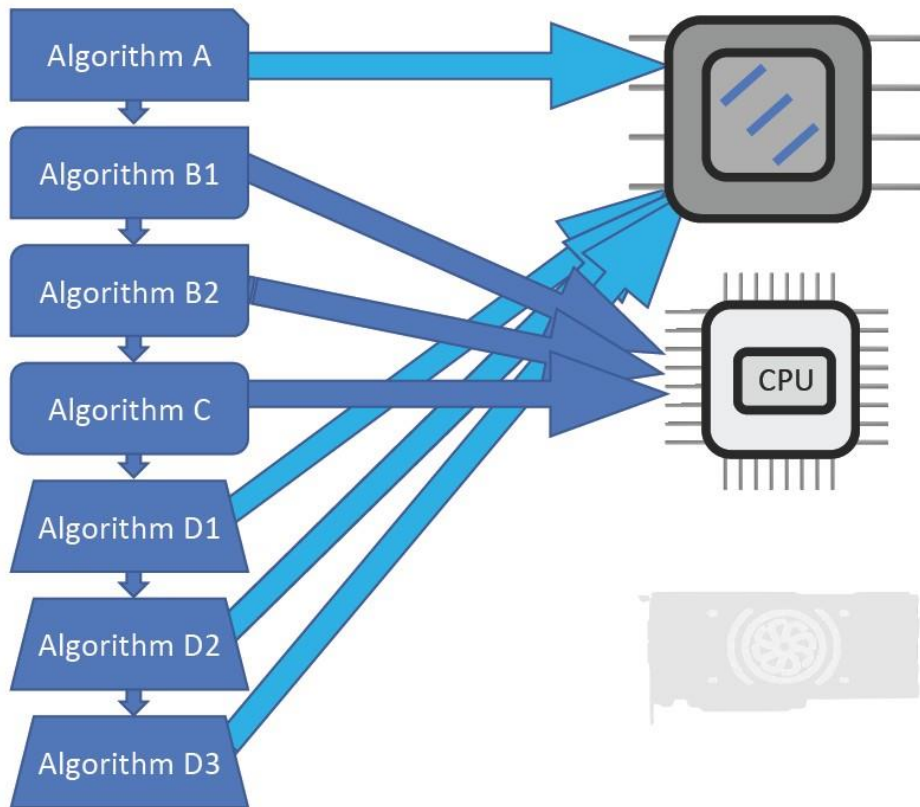
DPC++ DEVICE SELECTION



- ▶ In a system without accelerators, we use the CPU
- ▶ The grayed out devices are not in this system!

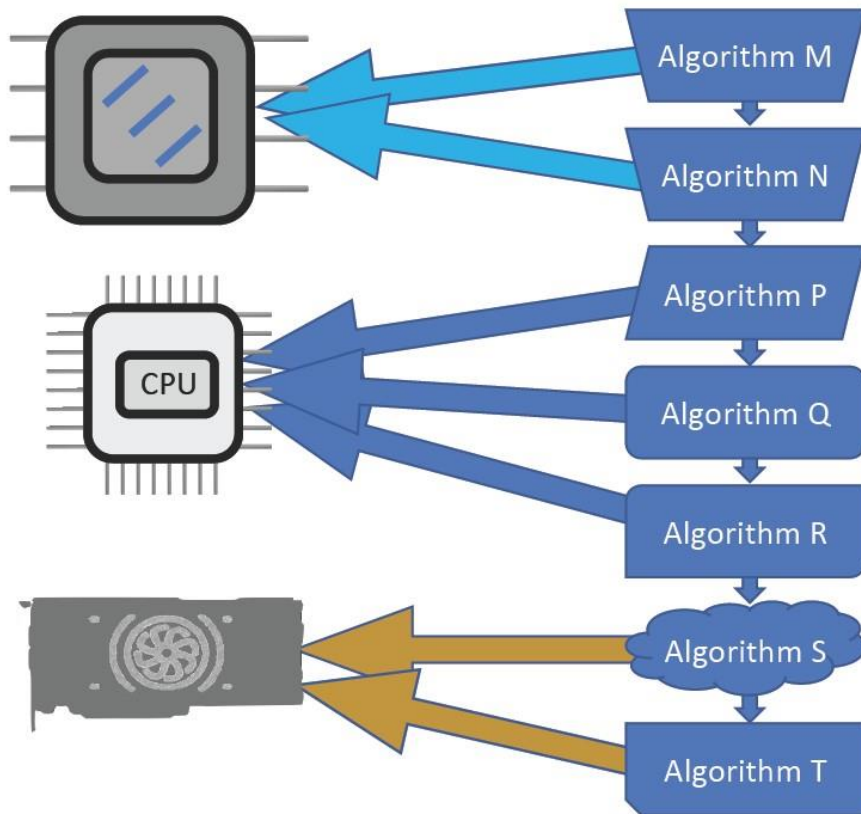


DPC++ DEVICE SELECTION



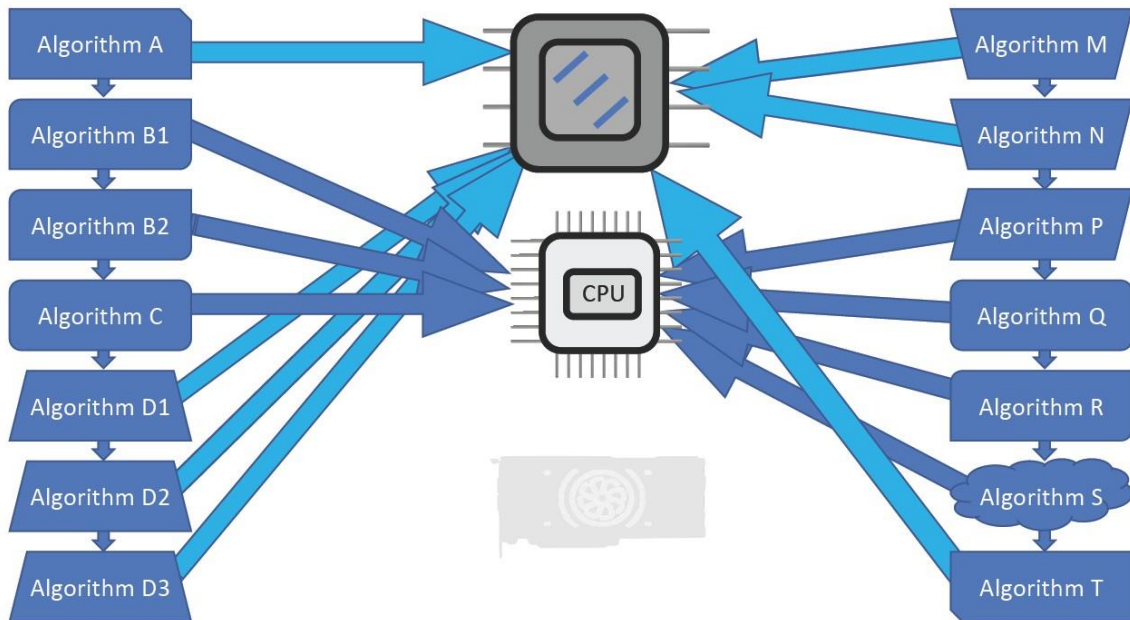
- ▶ We can control devices to use.
- ▶ Some algorithms use the top device.

DPC++ DEVICE SELECTION



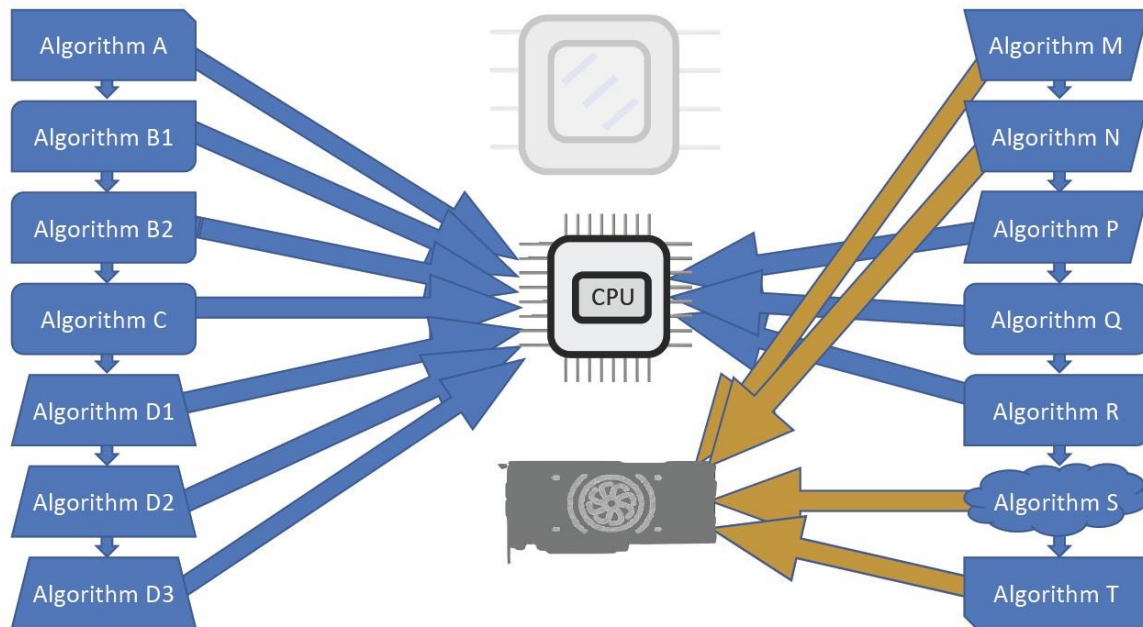
- ▶ A single system might have both accelerators available.
- ▶ We can choose the mapping we want - based on best match, application balance, data movement, etc.

DPC++ DEVICE SELECTION



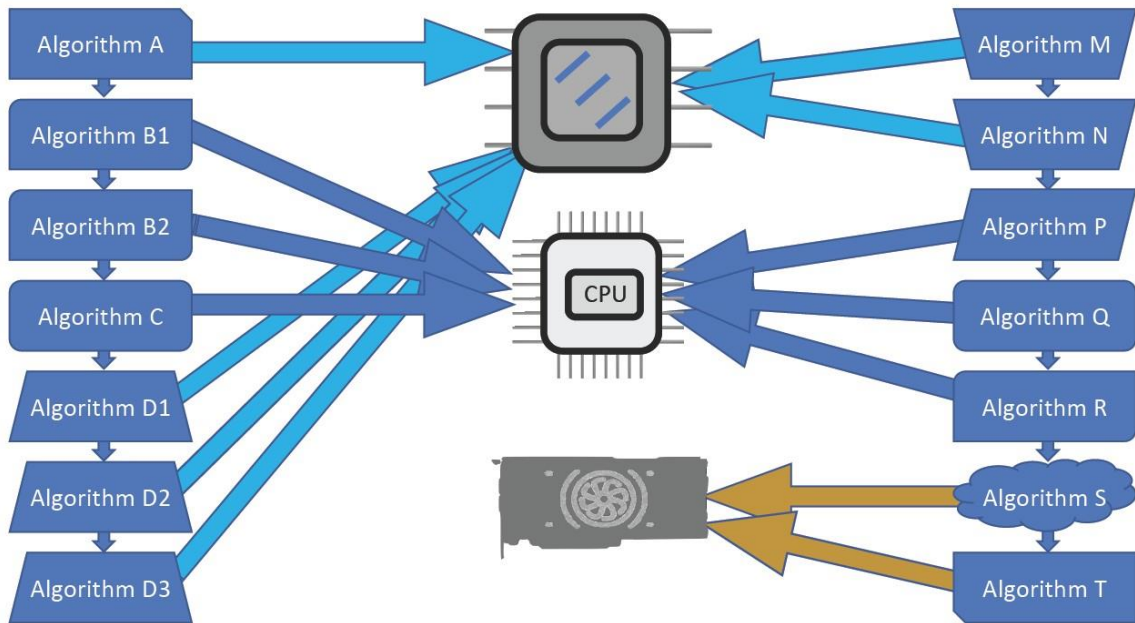
- ▷ In a system with the top device.
- ▷ Both programs can be accelerated.

DPC++ DEVICE SELECTION



- ▶ In a system with the bottom device.
- ▶ Only our second example had uses for it.

DPC++ DEVICE SELECTION



- ▷ In system with BOTH devices.
- ▷ Both programs have options.

WHERE AND HOW TO GET AND USE DPC++?

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.



INTEL® DEVCLLOUD FOR ONEAPI PROJECTS

A development sandbox to develop, test, and run your workloads across a range of Intel®-based CPUs, GPUs, and FPGAs using oneAPI^(Beta) software

[Sign Up for Beta](#)

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>



BUILD FROM OPEN SOURCE, EASY LINUX OR WINDOWS

<https://github.com/intel/llvm>

Branch: `sycl` ▾

`llvm` / `sycl` / `doc` / `GetStartedWithSYCLCompiler.md`

Prerequisites

- `git` - <https://git-scm.com/downloads>
- `cmake` version 3.2 or later - <http://www.cmake.org/download/>
- `python` - <https://www.python.org/downloads/release/python-2716/>
- C++ compiler
 - Linux: `gcc` version 5.1.0 or later (including libstdc++) - <https://gcc.gnu.org/install/>
 - Windows: `Visual Studio` version 15.7 preview 4 or later - <https://visualstudio.microsoft.com/downloads/>



BUILD FROM OPEN SOURCE, LINUX (FOR EXAMPLE)

```
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
```



JUMP ON DEVCLOUD

```
$ ssh devcloud
...login to the devcloud...

$ wget tinyurl.com/oneapimodule?2 -O 2.tz
$ tar xvfz 2.tz
...fetch and unpack code I'll be playing with for module 2...

$ pbsnodes -l free
...list of free nodes...

$ pbsnodes s001-145
...information about node s001-145...

$ pbsnodes | more
...lots more detail...

$ pbsnodes | grep properties
...useful properties list...

$ pbsnodes | grep fpga
...useful fpga oriented list...
```



HELLO QSUB

```
$ mkdir mytst
$ cd mytst

$ cat - > myhello.sh
echo "HELLO, WORLD!"
^D

$ qsub myhello.sh

$ qstat
Job ID Name          ...
-----
3463  myhello.sh ...

$ qstat
```

Optimization Notice

Copyright © 2020, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



HELLO QSUB

```
$ ls -l
total 8
-rw-r--r-- 1 u27938 u27938  21 Oct 14 22:58 myhello.sh
-rw----- 1 u27938 u27938   0 Oct 14 22:58 myhello.sh.e3463
-rw----- 1 u27938 u27938 603 Oct 14 22:58 myhello.sh.o3463

$ cat myhello.sh.o3463

#####
#      Date:           Mon Oct 14 22:58:58 PDT 2019
#      Job ID:         3463.v-qsvr-nda.aidevcloud
#      User:           u27938
# Resources:          neednodes=1:ppn=2,nodes=1:ppn=2,walltime=06:00:00
#####

HELLO, WORLD!

#####
# End of output for job 3463.v-qsvr-nda.aidevcloud
# Date: Mon Oct 14 22:58:59 PDT 2019
#####
```



HELLO QSUB

...use a particular node...

```
$ qsub -lnodes=s001-n155:ppn=2
```

...use a node based on a property...

```
$ qsub -lnodes=1:ppn=2:fpga_compile
```

```
$ qsub -lnodes=1:ppn=2:gpu
```

```
$ qsub -lnodes=1:ppn=2:skl
```

```
$ qsub -lnodes=1:ppn=2:cfl
```

Optimization Notice

Copyright © 2020, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



DPC++ SOFTWARE MODEL

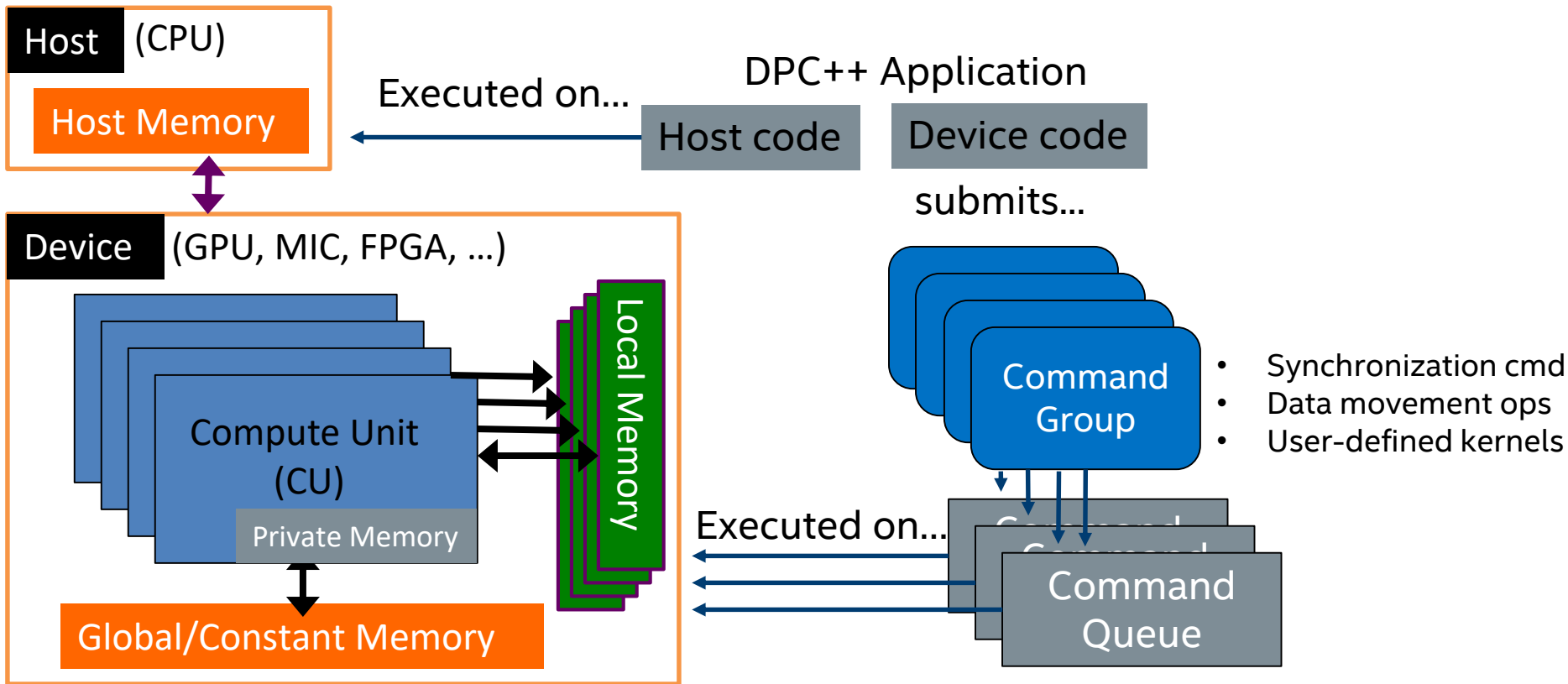
DPC++ SOFTWARE MODEL

- Details four models to employ one or more devices as an accelerator.
- Platform model - what to program (**host** and **devices**).
 - **Host:** A CPU-based system that executes the application scope and command group scope.
 - **Device:** An accelerator or specialized component
 - Examples include CPU, FPGA, GPU.
- Execution model – how to control (command queues)
Queues, Accessors
- Memory model - how to feed the data
Buffers, Images, Unified Shared Memory
- Kernel model – how to program (kernels)
Subset of C++, ND-range, work-item



PLATFORM MODEL

PLATFORM MODEL



Optimization Notice

Copyright © 2020, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Getting started with oneAPI



PLATFORM MODEL

```
auto platforms = platform::get_platforms();

for (auto& platform : platforms) {
    std::cout << "Platform: " << platform.get_info<info::platform::name>();
    std::cout << std::endl;

    auto devices = platform.get_devices();
    for (auto& device : devices) {
        std::cout << "  Device: " << device.get_info<info::device::name>();
        std::cout << std::endl;
    }
}
```

Platform: Intel(R) CPU Runtime for OpenCL(TM) Applications

Device: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz

Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)

Device: Intel(R) FPGA Emulation Device

Platform: Intel(R) OpenCL HD Graphics

Device: Intel(R) Gen9 HD Graphics NEO



PLATFORM MODEL

DEVICE SELECTOR

Get a device (any device):	<code>queue queue(); // default_selector{}</code>
Get device from class:	<code>queue queue(gpu_selector{}); queue queue(accelerator_selector{}); queue queue(cpu_selector{}); queue queue(host_selector{});</code>
Custom selector:	<code>class custom_selector : public device_selector { int operator()(..... ... queue queue(custom_selector{});</code>

default_selector

- DPC++ runtime scores all the devices and picks one with highest compute power
- Environment variable

```
export SYCL_DEVICE_TYPE=GPU | CPU | HOST
```

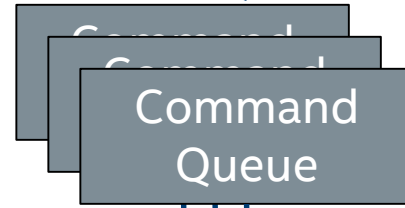


EXECUTION MODEL

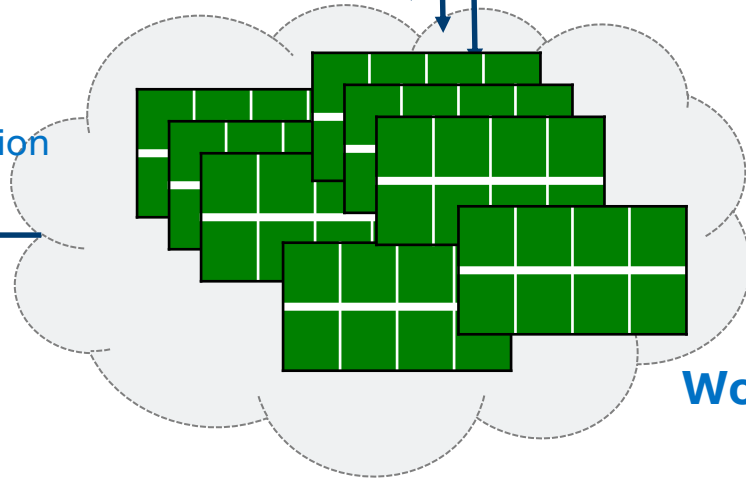
EXECUTION MODEL

EXECUTION OF KERNEL INSTANCES

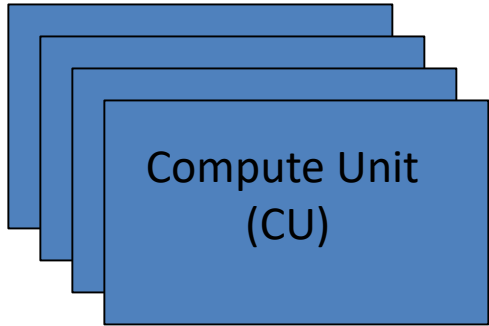
enqueued...



Kernel instance =
Kernel object &
nd_range &
work-group
decomposition



Device (GPU, FPGA, ...)

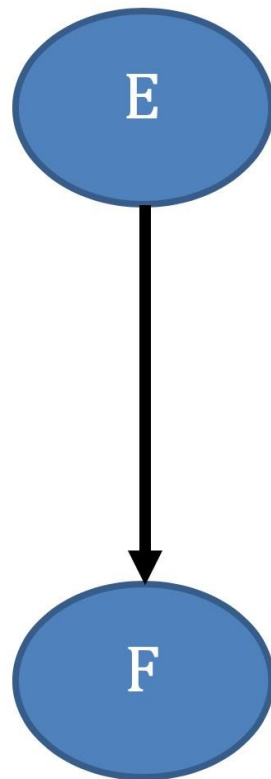
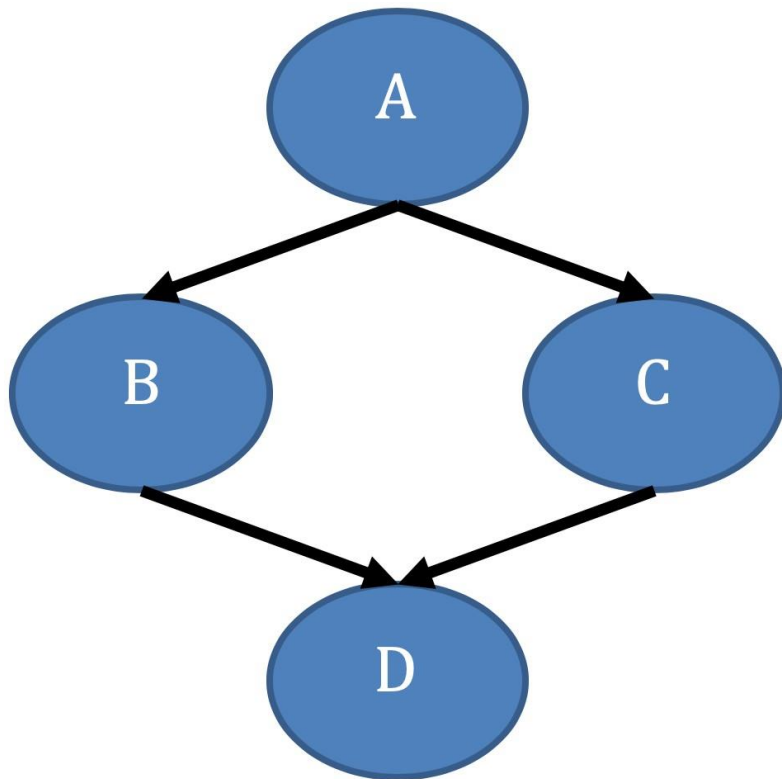


DEPENDENCIES IN A KERNEL-BASED WORLD

- ▶ 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

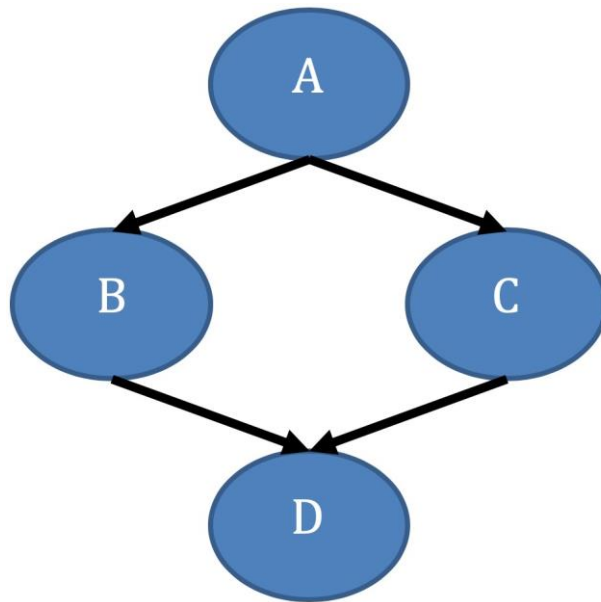
A, B, and C must be done before D

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 &h* 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 explicitly 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)



QUEUES

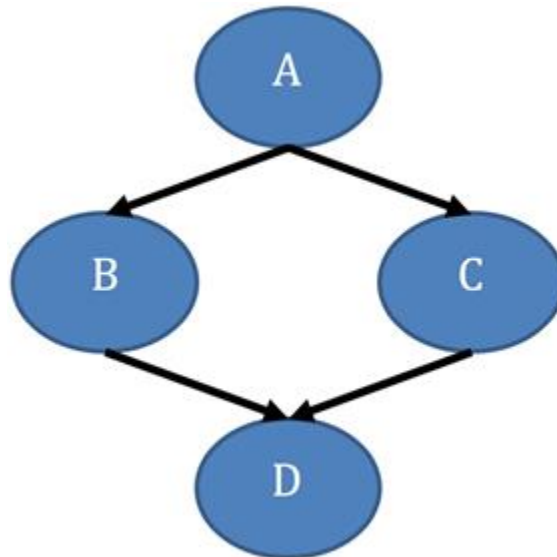
```
queue myQueue;
```

```
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);  
    h.depends_on(myTokC);  
    h.parallel_for<class taskD> (...);  
});
```



EXECUTION MODEL

SYNCHRONIZATION SUMMARY

Synchronization with kernel function

- Barriers for synchronizing work items within a workgroup.
- No synchronization primitives across workgroups

Synchronization between host and device

- Call to wait() member function of device queue
- Buffer destruction will synchronize the data with host memory
- Host accessor constructor is a blocked call and returns only after all enqueued kernels operating on this buffer finishes execution
- DAG construction from command group function objects enqueued into the device queue



C++ EXCEPTION-BASED

DPC++ is based on C++

- Errors in C++ are handled through exceptions
- SYCL uses exceptions, not return codes!

Synchronous exceptions

- Thrown immediately when an API call fails (e.g. can't create buffer)
- Normal C++ exceptions

```
// Synchronous Exception Handler
try {
    device_queue.reset(new queue(device_selector));
}
catch (exception const& e) {
    std::cout << "Caught a synchronous SYCL exception:" <<e.what();
    return;
}
```



C++ EXCEPTION-BASED

Asynchronous exceptions

- Caused by a future failure (e.g. during DAG node execution when data deps met)
 - Host program has already moved on to new things!
- Programmer provides processing function, and says when to process

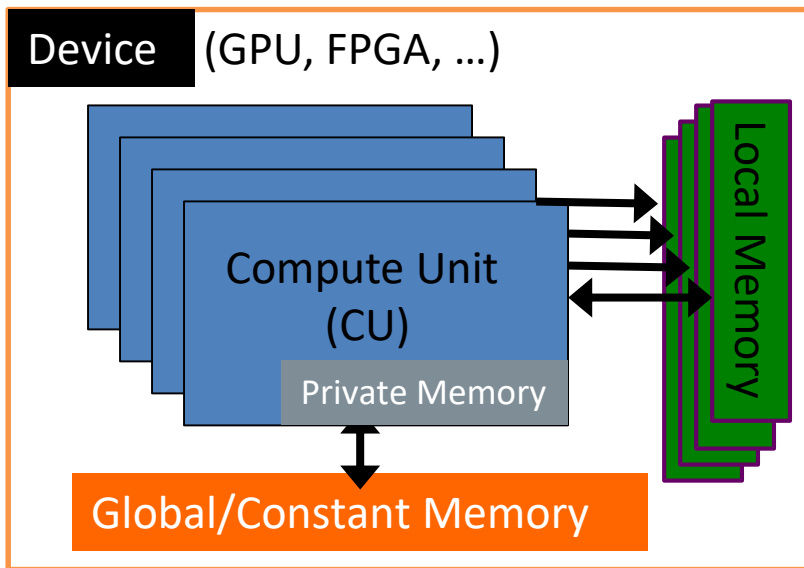
//Asynchronous Exception Handler on Device side

```
auto async_exception_handler = [] (exception_list exceptions) {
    for (std::exception_ptr const& e : exceptions) {
        try {
            std::rethrow_exception(e);
        }
        catch (exception const& e) {
            std::cout << "Caught a Asynchronous SYCL exception" << e.what() << std::endl;
        }
    }
};
```



MEMORY MODEL

MEMORY MODEL



Global Memory:

- Accessible for global operations that work across multiple work-items during the execution of a kernel. Reads and writes may be cached.
- Persistent across kernel invocations



MEMORY MODEL

ACCESS TO MEMORY & MEMORY CONSISTENCY

The application running on the host can use buffer objects to allocate memory in the global address space

To access data in buffers *inside a kernel*, the user must create an **accessor** object



- *Read-only/Write-only/Read-write*
- *Atomic*

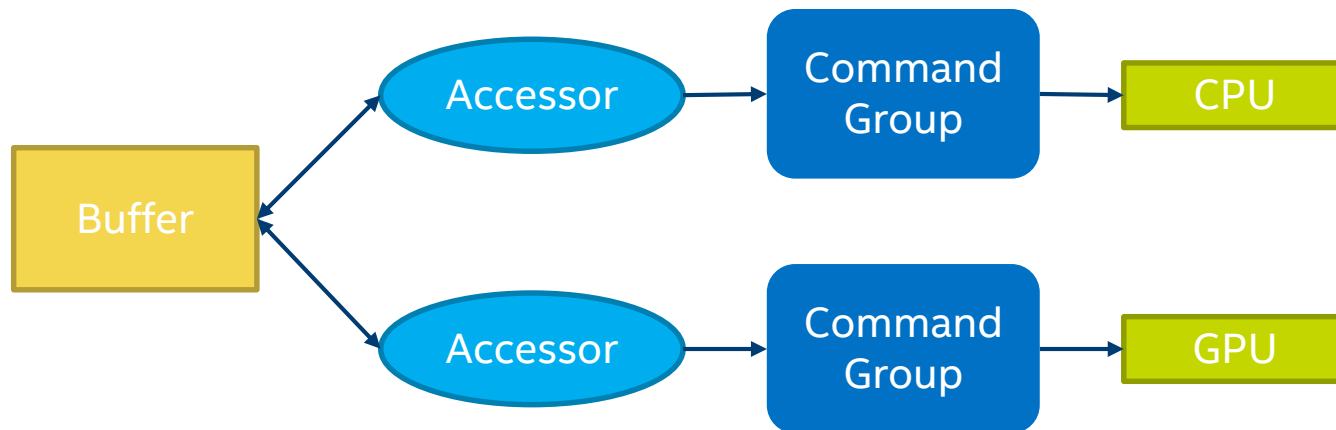
Any variable defined inside a **parallel_for/parallel_for_work_item** scope will be allocated in **private** memory.



MEMORY MODEL

MANAGING MEMORY ACROSS HOST AND DEVICES

Storage and access of memory is separated via buffers and accessors.



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 accessible 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

Basic code samples: <https://github.com/intel/llvm/tree/sycl/sycl/test/usm>



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(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)
free(deviceArray, ctxt);
```

- ▷ 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(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)



EXAMPLE

IMPLEMENTATION WITH BUFFERS & ACCESSORS

```
queue q;
```

Host can
initialize



```
int data[N] = {10,10};
```

Create buffer



```
buffer<int, 1> my_buffer(data, range<1>(N));
```

```
q.submit([&] (handler &h){
```

Instantiate
accessor



```
auto my_accessor = my_buffer.get_access<dpc_rw>(h);
```

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

Device can
modify



```
    size_t index = item.get_linear_id();  
    my_accessor[index] += 1;  
});
```

```
});  
q.wait_and_throw();
```

Update buffer



```
my_buffer.get_access<read>();
```

Host has
output



```
std::cout << "Output : " << data[0] << ", " << data[1] << std::endl;
```



EXAMPLE

IMPLEMENTATION WITH USM

```
queue q;
```

Setup Unified Shared Memory →

```
int* data = (int*)malloc_shared(N * sizeof(int), q.get_device(), q.get_context());
```

Host can initialize →

```
data[0] = 10; data[1] = 10;
```

```
q.submit([&](handler& h) {
```

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

```
    size_t index = item.get_linear_id();
```

```
    data[index] += 1;
```

```
});
```

```
});
```

```
q.wait_and_throw();
```

Device can modify →

```
std::cout << "Output : " << data[0] << ", " << data[1] << std::endl;
```

Host has output →



TASK SCHEDULING WITH USM

Explicit Scheduling

- Work submission returns an event
- Programmers can explicitly wait on these events to order tasks

```
float* a = malloc_shared(...);
float* b = malloc_shared(...);
float* c = malloc_shared(...);

queue Q; auto e = Q.submit([&](handler& h) {
    cgh.parallel_for(range<1> {10}, [=](id<1> i)
    {
        c[i] = a[i] + b[i];
    });
});
e.wait();
```

DAG Scheduling

- Programmers can also build DAGs with these events

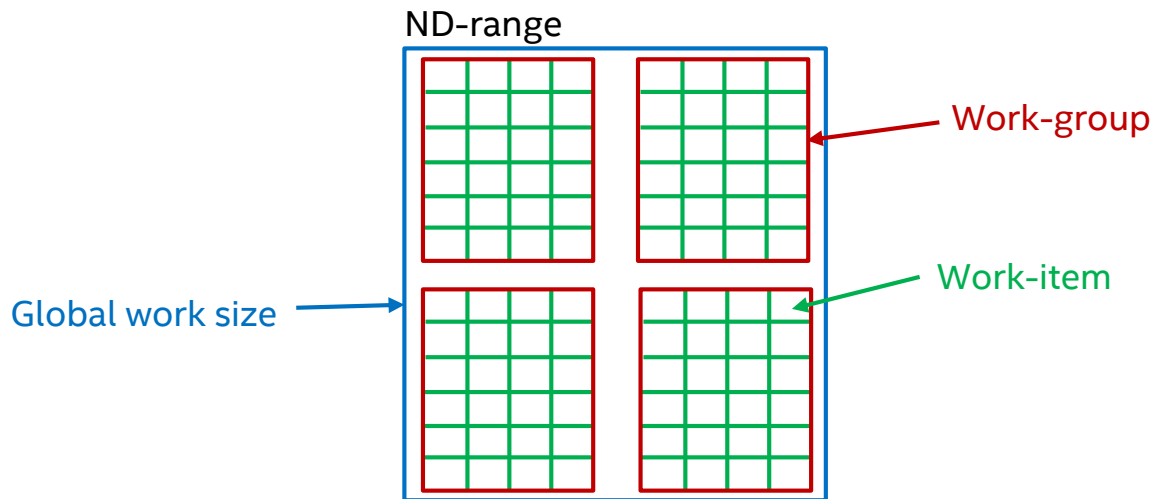
```
class handler {
    ...
public: ... void depends_on(event e);
};
auto e = Q.submit([&](handler& h) {
    ...
});
Q.submit([&](handler& h) {
    h.depends_on(e);
    h.parallel_for(    range<1> {10}, [=](id<1> i)
    {
        c[i] = a[i] + b[i];
    });
});
```



KERNEL MODEL

KERNEL EXECUTION MODEL

- Explicit ND-range for control- similar to programming models such as OpenCL, SYCL, CUDA.



ND_RANGE & ND_ITEM

Example: Process every pixel in a 1920x1080 image

- Each pixel needs processing, **kernel** is executed on each pixel (**work-item**)
- $1920 \times 1080 = 2\text{M}$ pixels = **global size**
- Not all 2M can run in parallel on device, there is hardware resource limits.
- We have to split into smaller groups of pixel blocks = local size (**work-group**)
- Either let the compiler determine work-group size **OR** we can specify the work-group size using `nd_range()`



ND_RANGE & ND_ITEM

Example: Process every pixel in a 1920x1080 image

- Let compiler determine work-group size

```
h.parallel_for(range<2>(1920, 1080), [=] (id<2> item) {  
    // CODE THAT RUNS ON DEVICE  
});
```

- Programmer specifies work-group size

```
h.parallel_for(nd_range<2>(range<2>(1920, 1080), range<2>(8, 8)),  
               [=] (id<2> item) {  
    // CODE THAT RUNS ON DEVICE  
});
```

global
size

local size
(work-group size)



ND_RANGE & ND_ITEM

Example: Process every pixel in a 1920x1080 image

- How do we choose **work-group size**?

GOOD

→ Work-group size of **8x8** divides equally for 1920x1080

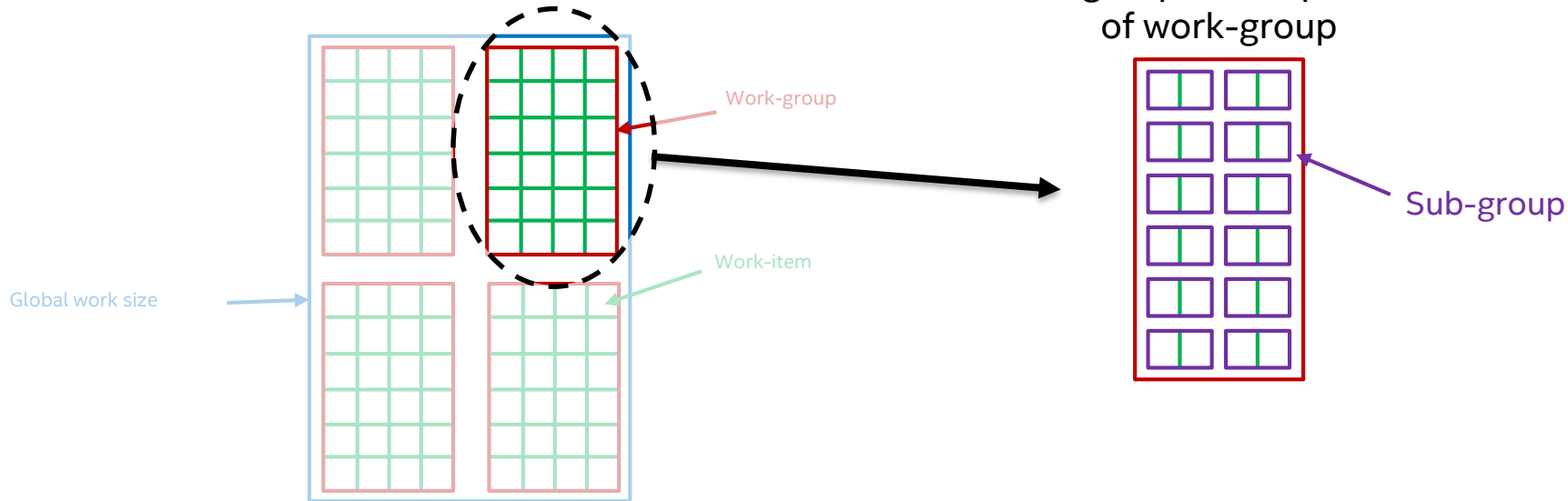
- Work-group size of **9x9** does not divide equally for 1920x1080
 - Compiler will throw error (invalid work group size error)
- Work-group size of **10x10** divides equally for 1920x1080
 - Works, but always better to use multiple of 8 for better resource utilization
- Work-group size of **24x24** divides equally for 1920x1080
 - $24 \times 24 = 576$, will fail compile assuming GPU max work-group size is 256



KERNEL EXECUTION MODEL

SUB-GROUPS

- Additional level of execution model hierarchy
- Think SIMD register or Warp



Optimization Notice

Copyright © 2020, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



DEVICE/KERNEL FEATURES

Supported features

- + templates
- + classes
- + operator overloading
- + static polymorphism
- + lambdas
- + short vector types (2/3/4/8/16-wide)
- + reach library of built-in functions

Unsupported features

- dynamic memory allocation
- dynamic polymorphism
- runtime type information
- exception handling
- function pointers
- pointer structure members
- static variables



KERNEL DEFINITION

STORED LAMBDA

```
buffer<int,1> buf { range<1>{16} };
```

```
auto placeholder_accessor = accessor<int, 1, dpc_w, access::target::global_buffer,  
access::placeholder::true_t>(buf);
```

```
auto lambda = [=](id<1> index) { placeholder_accessor[2] = 5; };
```

```
deviceQueue.submit([&](handler &h) {  
  
    h.require(placeholder_accessor);  
  
    h.parallel_for(range<1>(16), lambda);  
  
});
```



KERNEL DEFINITION

NAMED FUNCTOR

Clear C++ way to define re-usable kernels.

```
class Functor{
public:
    Functor(accessor<int, 1, dpc_w, access::target::global_buffer> ptr) : ptr_ {ptr} {}
    void operator() (item<1> i) { ptr_[i] = 5; } // Kernel body
private:
    accessor<int, 1, dpc_w, access::target::global_buffer> ptr_;
};

int main() {
    queue deviceQueue;
    buffer<int,1> buf { range<1>{16} };
    deviceQueue.submit([&] (handler &h) {
        auto acc = buf.get_access<dpc_w>(h);
        h.parallel_for<>(range<1>(16), Functor( acc ));
    });
}
```



TIPS AND TRICKS

TROUBLESHOOTING RUNTIME ISSUES

<https://github.com/bashbaug/OpenCLPapers/blob/markdown/OpenCLOnLinux.md>

- Step 1: Do you have libOpenCL.so?

```
$ ldd /path/to/your/application | grep OpenCL  
libOpenCL.so.1 => /path/to/your/libOpenCL.so.1 (0x00007f9182d27000)  
locate libOpenCL.so
```
- Step 2: Do You Have An OpenCL Implementation?
 - Is your OpenCL implementation in /etc/OpenCL/vendors?

```
ls -l /etc/OpenCL/vendors/
```
 - Is the file for your OpenCL implementation readable?
 - Are the contents of the file for your OpenCL implementation correct?

```
$ more vendor.icd  
/opt/vendor/opencv-1.2-X.Y.Z.W/lib64/libvndorocl.so
```

NOTE: with oneAPI we set the path using `setvars.sh` and the `icd` file has only the lib name, not a full path
- `clinfo`
 - <https://github.com/Oblomov/clinfo>
- Using `strace` to Troubleshoot

```
strace ./a.out 2>trace.txt
```



BUFFER CREATION



```
int main() {  
    auto Data = new int[10];  
    buffer<int, 1> Buf(Data, {10});  
    Data[4] = 42;  
    delete Data;  
}
```

- The ownership of the memory passed to the buffer constructor belongs to the buffer
- Accessing(and removing) this memory is undefined behavior until the buffer is destructed



BUFFER CREATION



```
int main() {  
    auto Data = new int[10];  
    {  
        buffer<int, 1> Buf(Data, {10});  
    }  
    Data[4] = 42;  
    delete Data;  
}
```

- The ownership of the memory passed to the buffer constructor belongs to the buffer
- The buffer is destructed when we go out of scope
- Operations with the memory is legal



CONTEXT AND PROGRAM COMPILATION



```
int main() {  
    buffer<int, 1> Buf({10});  
    for(int I = 0; I < Buf.get_count(); ++I) {  
        queue Q ();  
        Q.submit([&](handler &h) {  
            auto Acc = Buf.get_access<dpc_w>(h);  
            h.single_task( [= ] () { Acc[0] = 42; });  
        });  
    }  
}
```

- Creation of queue triggers creation of new context object
- Submitting kernel for a first time for a context triggers JIT compilation of device code which is cached for the context
- Exiting loop scope leads to queue -> context -> cached programs destruction
- On the second iteration the process happens again



CONTEXT AND PROGRAM COMPILATION



```
int main() {  
    buffer<int, 1> Buf({10});  
    queue Q ();  
    for(int I = 0; I < Buf.get_count(); ++I) {  
        Q.submit([&](handler &h) {  
            auto Acc = Buf.get_access<dpc_w>(h);  
            h.single_task( [= ] () { Acc[0] = 42; });  
        });  
    }  
}
```

- Creation of queue triggers creation of new context object
- Submitting kernel for a first time for a context triggers JIT compilation of device code which is cached for the context
- Exiting loop scope DOESN'T lead to queue -> context -> cached programs destruction
- On the second a cached program is used



USING COMMAND GROUP



```
int main() {  
    buffer<int, 1> Buf({10});  
    queue Q();  
    Q.submit([&](handler &h) {  
        auto Acc = Buf.get_access<dpc_w>(h);  
        h.single_task( [= ] () { Acc[0] = 42; });  
        h.fill(Acc, 42);  
    });  
    Q.submit([&](handler &h) {  
        auto Acc = Buf.get_access<dpc_w>(h);  
        Acc[0] = 32;  
    });  
}
```

- Only one operation can be “requested” in one command group
 - Command group must contain kernel invocation or explicit memory operation
 - Accessing memory thru device side accessors is legal in kernel scope only



USING COMMAND GROUP



```
int main() {
    buffer<int, 1> Buf({10});
    queue Q();
    Q.submit([&](handler &h) {
        auto Acc = Buf.get_access<dcp_w>(h);
        h.fill(Acc, 42);
    });
    Q.submit([&](handler &h) {
        auto Acc = Buf.get_access<dpc_w>(h);
        Acc[0] = 32;
        h.single_task( [= ] () { Acc[0] = 42; });
    });
}
```

- Only one operation can be “requested” in one command group
 - Command group must contain kernel invocation or explicit memory operation
 - Accessing memory thru device side accessors is legal in kernel scope only



BUFFER CREATION



```
int main() {  
    int Data1[1] = {1};  
    int Data2[1] = {42};  
  
    {  
        buffer<int, 1> Buf1(Data1, {1});  
        buffer<int, 1> Buf2(Data2, {1});  
  
        queue Q();  
        Q.submit([&](handler &h) {  
            auto Acc2 = Buf2.get_access<dpc_r>(h);  
            h.single_task( [= ] () { Acc2[0] *= 42; });  
        });  
  
        Q.submit([&](handler &h) {  
            auto Acc1 = Buf.get_access<dpc_rw>(h);  
            auto Acc2 = Buf.get_access<dpc_rw>(h);  
            h.single_task( [= ] () { Acc1[0] += Acc2[0]; });  
        });  
    }  
  
    return Data1[0];  
}
```

- The pointer passed to the buffer constructor should be aligned to the size of the type buffer points to in order to achieve zero copy
- The memory will be copied to host during buffer destruction while it's not needed



BUFFER CREATION



```
int main() {  
    int Data1[1] __attribute__((aligned(sizeof(int)))) = {1};  
    int Data2[1] = {42};  
    {  
        buffer<int, 1> Buf1(Data1, {1});  
        buffer<int, 1> Buf2(Data2, {1});  
        Buf2.set_write_back(false);  
        queue Q();  
        Q.submit([&](handler &h) {  
            auto Acc2 = Buf2.get_access<dpc_r>(h);  
            h.single_task( [= ] () { Acc2[0] *= 42; });  
        });  
        Q.submit([&](handler &h) {  
            auto Acc1 = Buf.get_access<dpc_rw>(h);  
            auto Acc2 = Buf.get_access<dpc_r>(h);  
            h.single_task( [= ] () { Acc1[0] += Acc2[0]; });  
        });  
    }  
    return Data1[0];  
}
```

- The pointer passed to the buffer constructor should be aligned to the size of the type buffer points to in order to achieve zero copy
- Calling `set_write_back(false)` disables write back on buffer destruction
- Several ways to disable “copy-back” during the buffer construction:
 - `set_write_back(false)`
 - `set_final_data(nullptr)`
 - pass pointer to const data to the buffer constructor
 - no “write accessor” for the buffer



SUMMARY

Data Parallel C++:

- Delivers power and productivity of modern C++
- Leverages SYCL* standard to support parallelism and heterogenous programming

Links:

- Intel oneAPI
 - <https://software.intel.com/en-us/oneapi>
- Open source project on Github
 - <https://github.com/intel/llvm>
- Intel® DevCloud for oneAPI Projects
 - <https://software.intel.com/en-us/devcloud/oneapi>



LEGAL DISCLAIMER & OPTIMIZATION NOTICE

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

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

Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

Intel, the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

Other names and brands may be claimed as the property of others

© Intel Corporation.

