

SYCL Essentials

oneAPI VIRTUAL WORKSHOP

Ben Odom
Praveen Kundurthy

ISC 2022

The Intel logo is located in the bottom left corner. It consists of the word "intel" in a white, lowercase, sans-serif font, followed by a registered trademark symbol (®). To the left of the text is a graphic of four squares of varying shades of blue, arranged in a 2x2 grid pattern.

intel®

Abstract: oneAPI SYCL programming for Heterogenous Computing on Intel® Devcloud

- In this workshop we will cover basic to advanced concepts in SYCL programming for heterogenous computing. We will work on the Intel DevCloud to do hands-on coding. At the end of this workshop you will be able to efficiently write SYCL code for heterogenous computing.
- Explain how oneAPI can solve the challenges of programming in a heterogeneous world
- Understand the SYCL language for High Performance Computing Applications and oneAPI programming model
- How to use SYCL Buffers and Accessors for data and memory management between host and device
- Understand the basics of Graphs and Dependences in SYCL.
- Onboard to Intel® DevCloud to test-drive oneAPI tools and libraries.
- We will learn pointer based memory management for heterogenous computing using Unified Shared Memory.
- Understand implicit and explicit way of moving memory using Unified Shared Memory and also handling data dependency between kernel executions.
- Use SYCL reduction to simplify reduction with parallel kernels
- Take advantages reduce function to do reduction at sub_group and work_group level

Introduction to oneAPI

- **Agenda**

- a) Introduction & Overview to oneAPI
- b) Introduction to the Intel® DevCloud
- c) Introduction to Jupyter notebooks used for training
- d) Introduction to SYCL
- e) SYCL Program Structure
- f) Graphs and Dependences
- g) Advanced SYCL Topics (USM, SubGroups, Reductions)

- **Hands On**

- Introduction to SYCL - Simple
- Complex multiplication
- Buffers
- Advanced SYCL Topics
- Unified Shared Memory
- Reductions

Learning Objectives

Explain how oneAPI can solve the **challenges of programming** in a heterogeneous world

Use **oneAPI solutions** to enable your workflows

Experiment with **oneAPI tools and libraries** on the Intel® DevCloud

Understand the SYCL language and programming model

Use **device selection** to **offload kernel workloads**

Build a sample SYCL application through hands-on lab exercises

Cross-Architecture Programming for Accelerated Compute, Freedom of Choice for Hardware

oneAPI: Industry Initiative & Intel Products

One Intel Software & Architecture group
Intel Architecture, Graphics & Software
November 2020



All information provided in this deck is subject to change without notice.
Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Programming Challenges

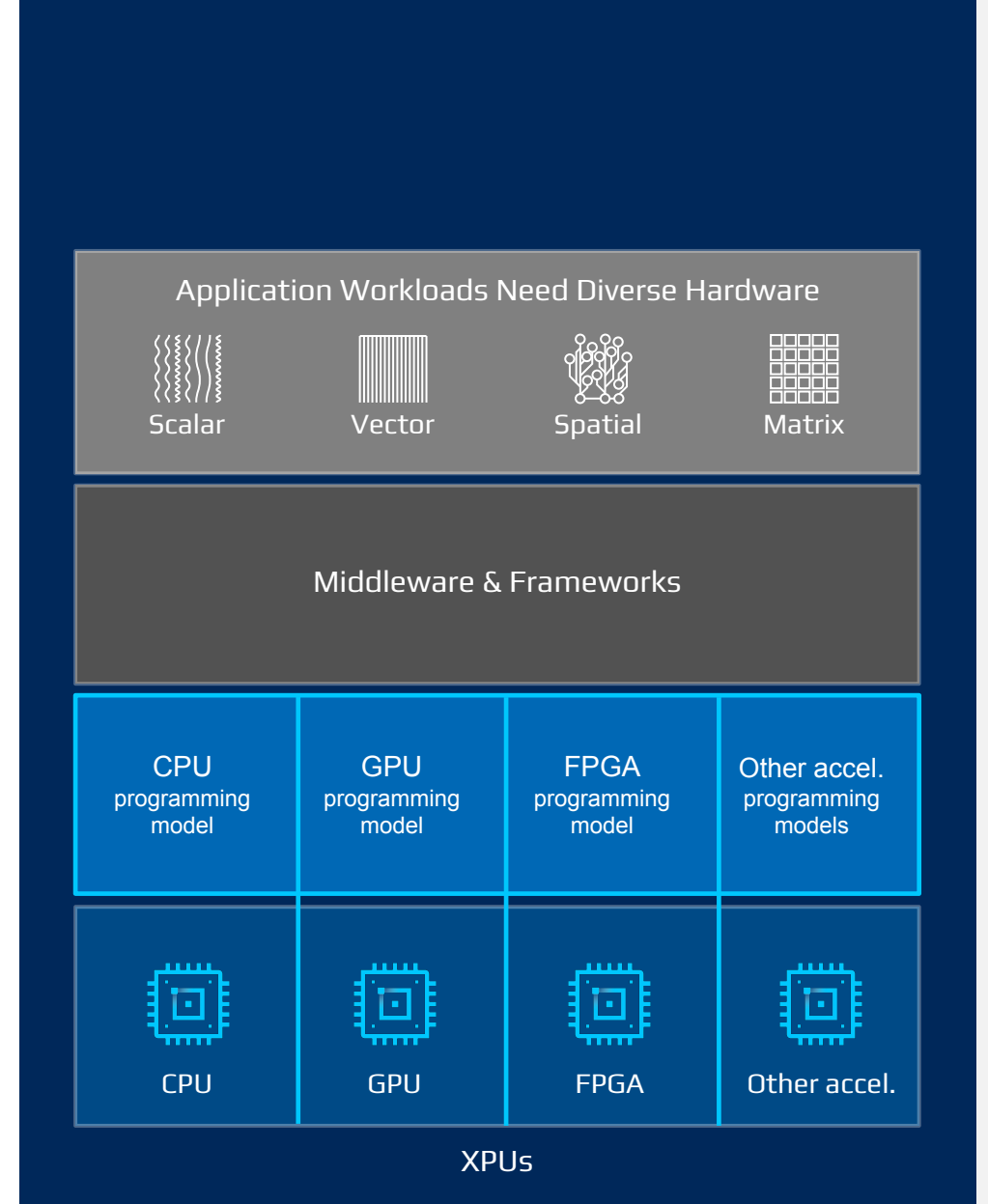
for Multiple Architectures

Growth in specialized workloads

Variety of data-centric hardware required

Separate programming models and toolchains for each architecture are required today

Software development complexity limits freedom of architectural choice



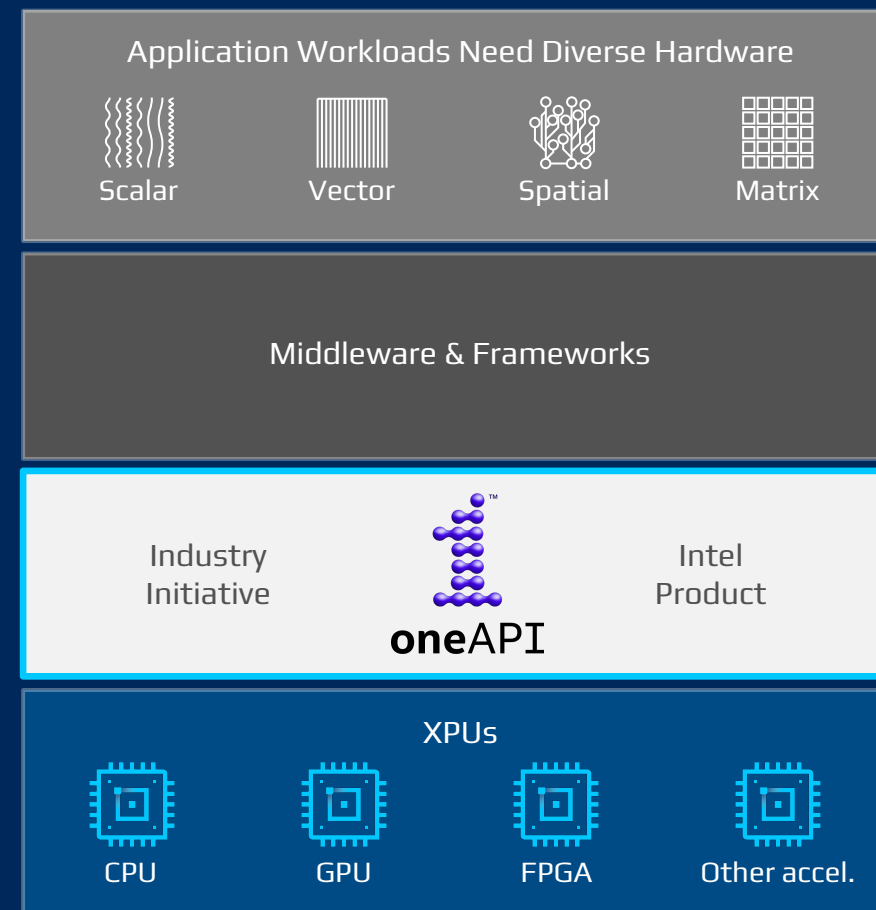
Introducing oneAPI

Cross-architecture programming that delivers freedom to choose the best hardware

Based on industry standards and open specifications

Exposes cutting-edge performance features of latest hardware

Compatible with existing high-performance languages and programming models including C++, OpenMP, Fortran, and MPI



Intel® oneAPI Toolkits

A complete set of proven developer tools expanded from CPU to XPU



Intel® oneAPI Base Toolkit

Native Code Developers



A core set of high-performance tools for building C++, SYCL applications & oneAPI library-based applications

Add-on Domain-specific Toolkits

Specialized Workloads



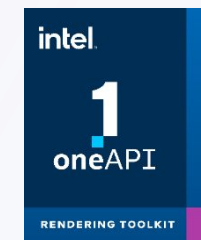
Intel® oneAPI Tools for HPC

Deliver fast Fortran, OpenMP & MPI applications that scale



Intel® oneAPI Tools for IoT

Build efficient, reliable solutions that run at network's edge



Intel® oneAPI Rendering Toolkit

Create performant, high-fidelity visualization applications

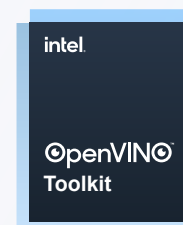
Toolkits powered by oneAPI

Data Scientists & AI Developers



Intel® AI Analytics Toolkit

Accelerate machine learning & data science pipelines with optimized DL frameworks & high-performing Python libraries



Intel® Distribution of OpenVINO™ Toolkit

Deploy high performance inference & applications from edge to cloud

Intel® oneAPI Base Toolkit

Accelerate Data-centric Workloads

A core set of core tools and libraries for developing high-performance applications on Intel® CPUs, GPUs, and FPGAs.

Who Uses It?

- A broad range of developers across industries
- Add-on toolkit users since this is the base for all toolkits

Top Features/Benefits

- Data Parallel C++ compiler, library and analysis tools
- DPC++ Compatibility tool helps migrate existing code written in CUDA
- Python distribution includes accelerated scikit-learn, NumPy, SciPy libraries
- Optimized performance libraries for threading, math, data analytics, deep learning, and video/image/signal processing

Intel® oneAPI Base Toolkit

Direct Programming

Intel® oneAPI DPC++/C++ Compiler

Intel® DPC++ Compatibility Tool

Intel® Distribution for Python

Intel® FPGA Add-on for oneAPI Base Toolkit

API-Based Programming

Intel® oneAPI DPC++ Library oneDPL

Intel® oneAPI Math Kernel Library - oneMKL

Intel® oneAPI Data Analytics Library - oneDAL

Intel® oneAPI Threading Building Blocks - oneTBB

Intel® oneAPI Video Processing Library - oneVPL

Intel® oneAPI Collective Communications Library oneCCL

Intel® oneAPI Deep Neural Network Library - oneDNN

Intel® Integrated Performance Primitives - Intel® IPP

Analysis & debug Tools

Intel® VTune™ Profiler

Intel® Advisor

Intel® Distribution for GDB



Intel® DPC++ Compatibility Tool

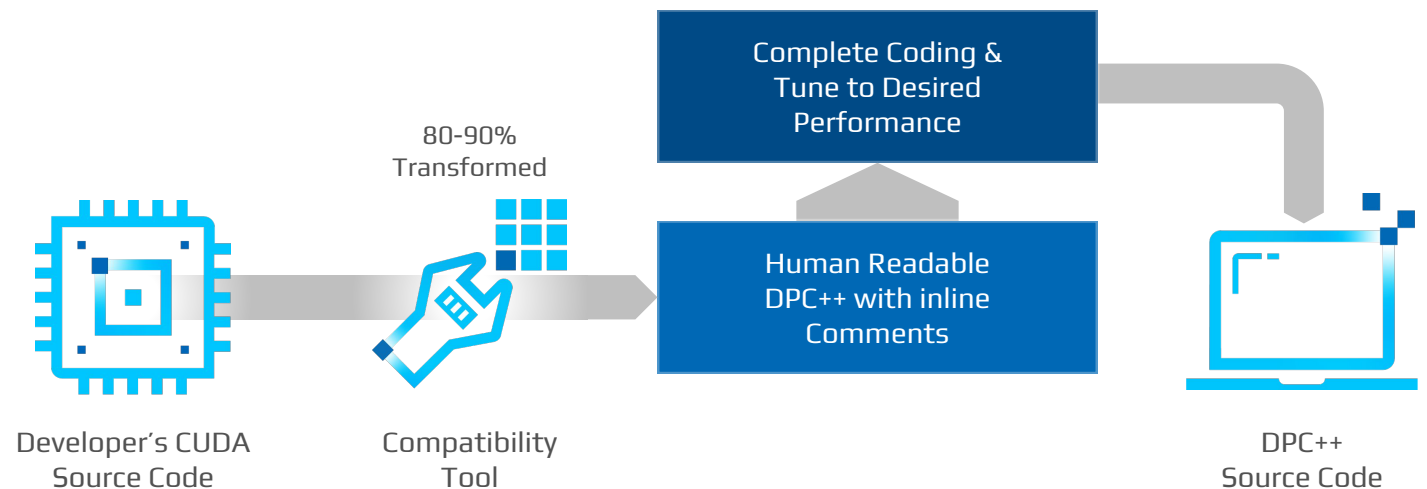
Minimizes Code Migration Time

Assists developers migrating code written in CUDA to SYCL once, generating **human readable** code wherever possible

~80-90% of code typically migrates automatically

Inline comments are provided to help developers finish porting the application

Intel DPC ++ Compatibility Tool Usage Flow



SYCL Profiling-Tune for CPU, GPU & FPGA

See the lines of SYCL that consume the most time

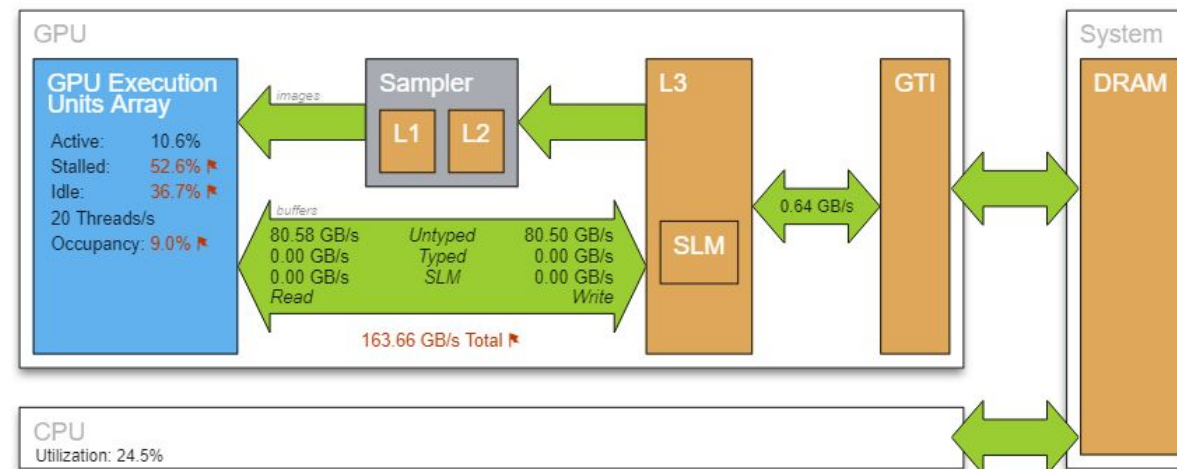
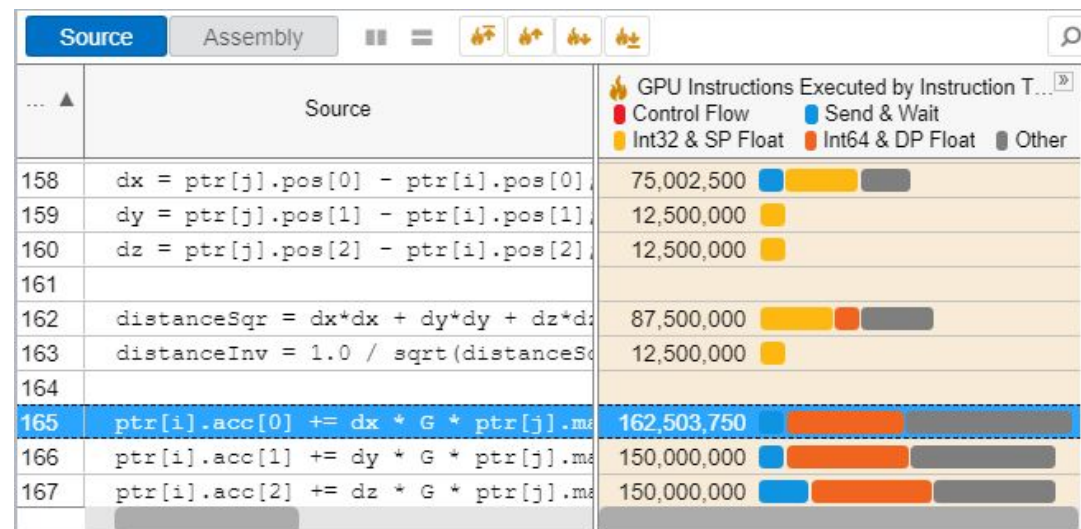
Optimize for any supported hardware accelerator

Tune OpenMP offload performance

CPU, GPU, FPGA, threading, memory, cache, storage...

SYCL, C, C++, Fortran, Python, Go, Java, or a mix

There will still be a need to tune for each architecture.



Intel® Advisor

Design Assistant - Design for Modern Hardware

Offload Advisor

Estimate performance of offloading to an accelerator

Roofline Analysis

Optimize CPU/GPU code for memory and compute

Vectorization Advisor

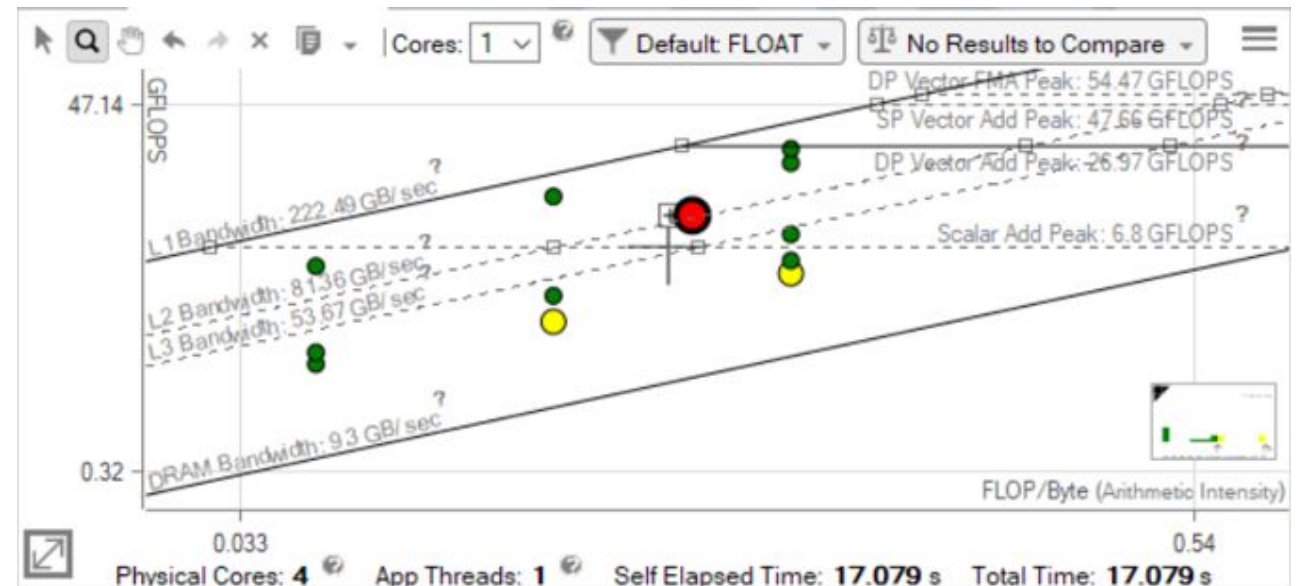
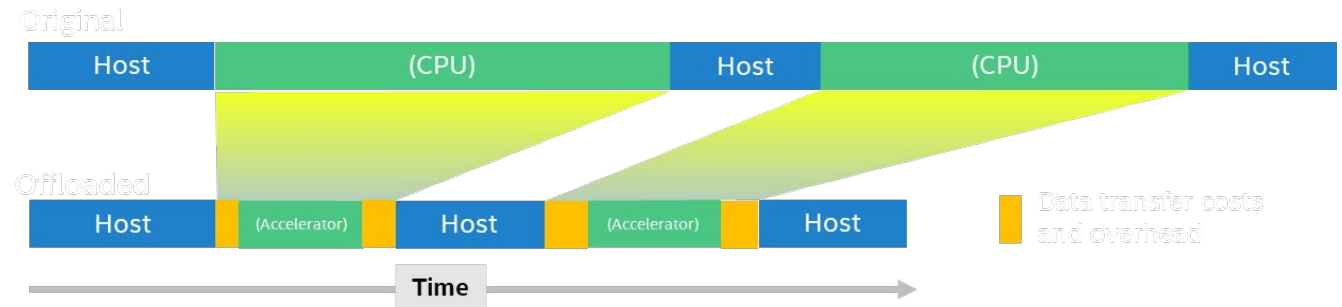
Add and optimize vectorization

Threading Advisor

Add effective threading to unthreaded applications

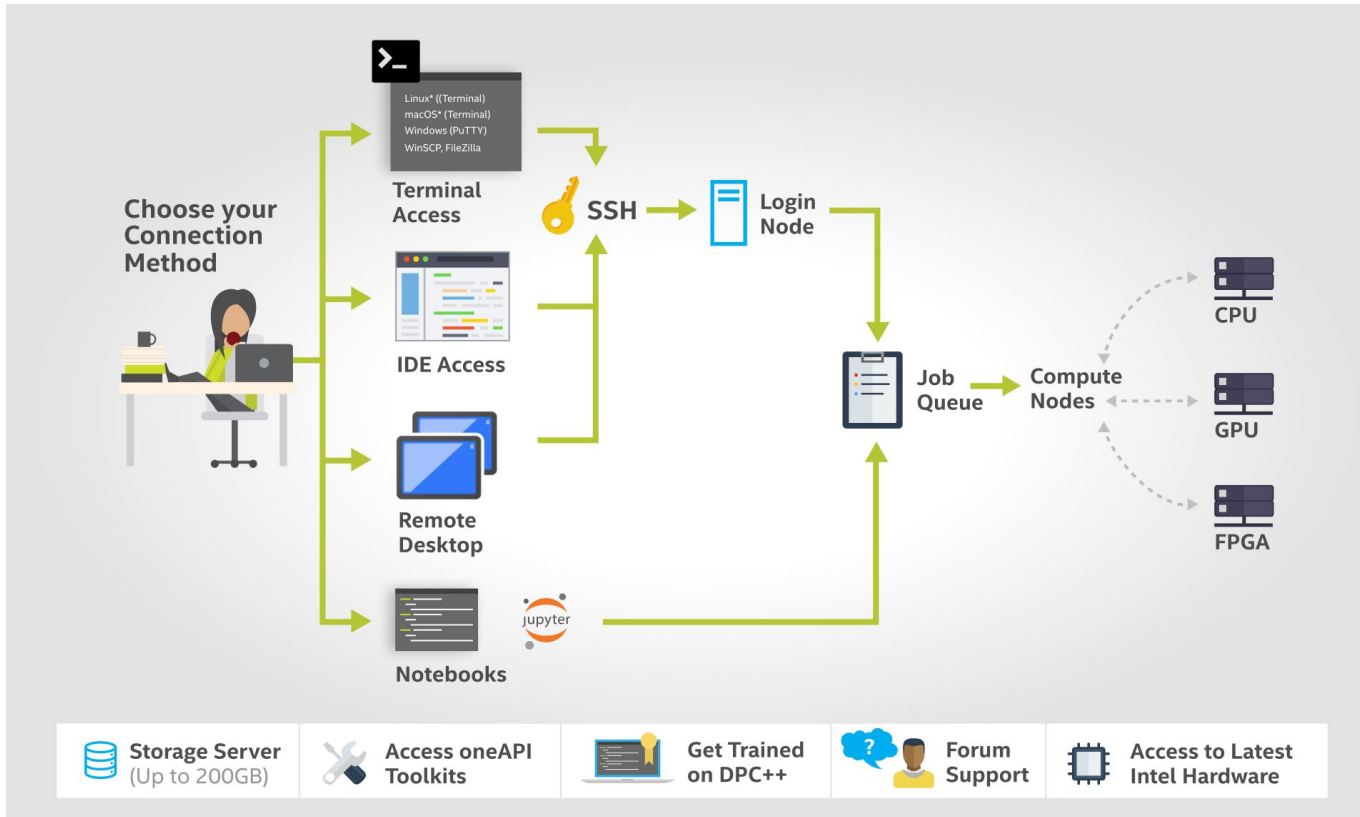
Flow Graph Analyzer

Create and analyze efficient flow graphs



Setup Intel® DevCloud and Jupyter Environment

Intel® DevCloud for oneAPI – How it Works



<https://devcloud.intel.com/oneapi/>

Development Environment

- 220 GB of file storage
- 192 GB of RAM
- Ubuntu 20.04
- Up to 24 hours of continuous workload execution times
- Free 120-day access; account extensions upon request

Quick How-to Resources

- Videos
- Developer guides



Just Minutes to Your oneAPI Project on Intel® DevCloud



1:52

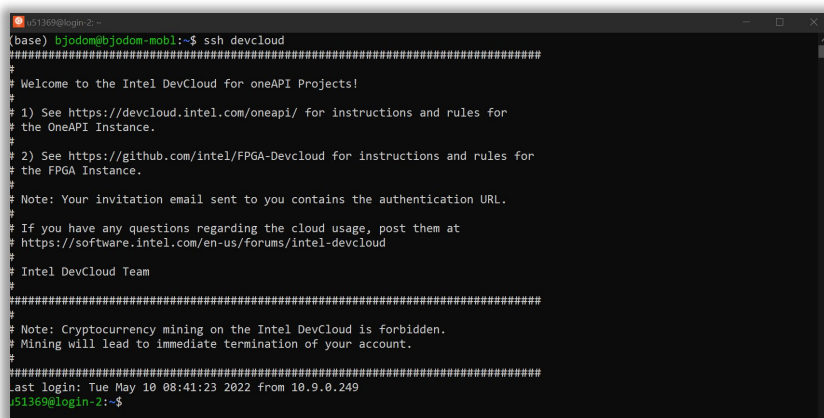
Developer Access/Experience based on their preference:

Power Coders : Use command line/ssh

Data Scientists/Python : [Jupyter Notebook](#) : Modern UI with one click interface, “code-> edit -> run -> see results”

Early Adopter next gen UI: [Visual Code](#) Electron based IDE

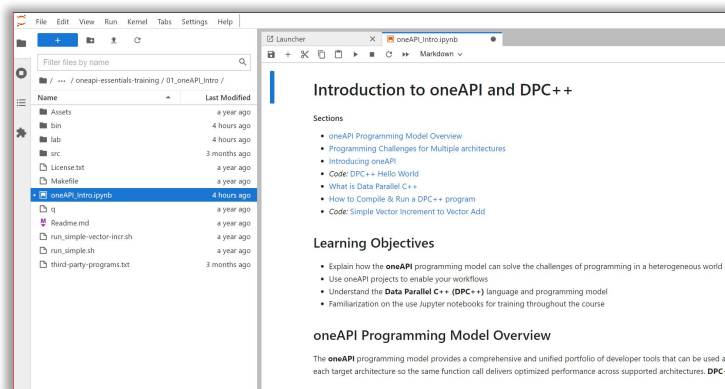
SSH Terminal



```
(base) bjdodm@bjodm-mobl:~$ ssh devcloud
Welcome to the Intel DevCloud for oneAPI Projects!
=====
# 1) See https://devcloud.intel.com/oneapi/ for instructions and rules for
# the OneAPI Instance.
# 2) See https://github.com/intel/FPGA-DevCloud for instructions and rules for
# the FPGA Instance.
# Note: Your invitation email sent to you contains the authentication URL.
# If you have any questions regarding the cloud usage, post them at
# https://software.intel.com/en-us/forums/intel-devcloud
# Intel DevCloud Team
#
# Note: Cryptocurrency mining on the Intel DevCloud is forbidden.
# Mining will lead to immediate termination of your account.
=====
last login: Tue May 10 08:41:23 2022 from 10.9.0.249
$1369@login-2:~$
```

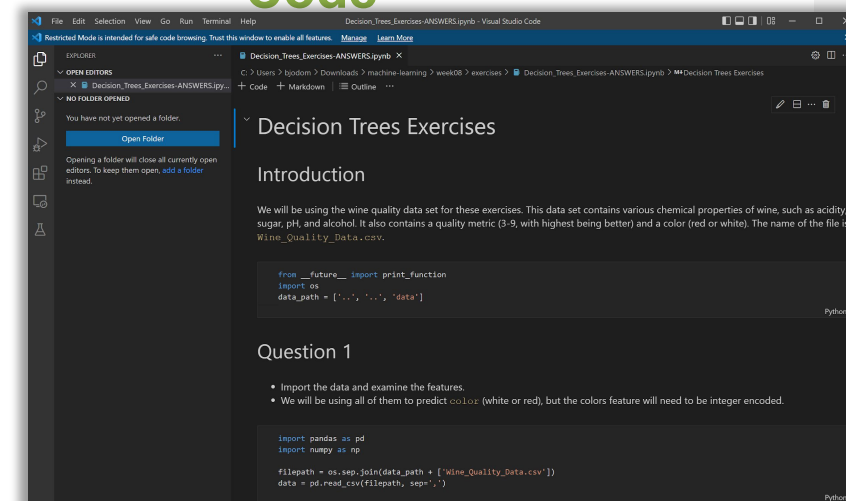
HPC Developers

Jupyter Notebook



HPC Training / AI Developers /
Data Scientists

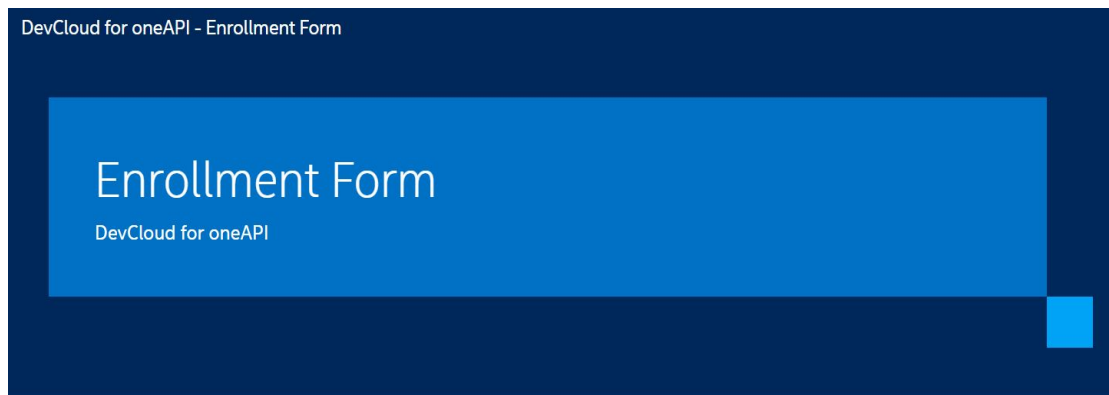
Visual Code



HPC/AI Developers

Register to Intel® Devcloud for oneAPI

- Step 1: Register or Sign into Intel Developer Zone



Step 1: Sign in or Register

To get an Intel® DevCloud account, you must first create a Basic Intel® Account

Sign in

Register

- Step 2: Activate Intel Devcloud Account

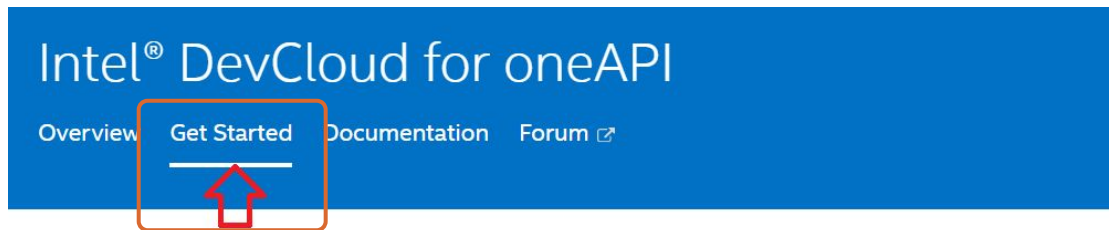
Step 2: Activate Intel® DevCloud for oneAPI

To get free access, tell us a bit more about yourself and how you would like to use the Intel DevCloud.

<https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html>


Get Started with Intel® Devcloud for oneAPI

- Step 3: Click on Get Started button



Explore Intel oneAPI Toolkits in the DevCloud

These toolkits are for performance-driven applications—HPC, IoT, advanced rendering, deep learning—to see what it includes, explore training modules, and go deeper with developer guides.



Intel® oneAPI Base Toolkit

Build and deploy high-performance, data-centric applications across diverse hardware

[Get Started with your first Sample](#) [View Training Modules](#)

The toolkit includes:

- Step 4: Scroll Down to the bottom of the page and click on Launch JupyterLab

Connect with Jupyter* Lab



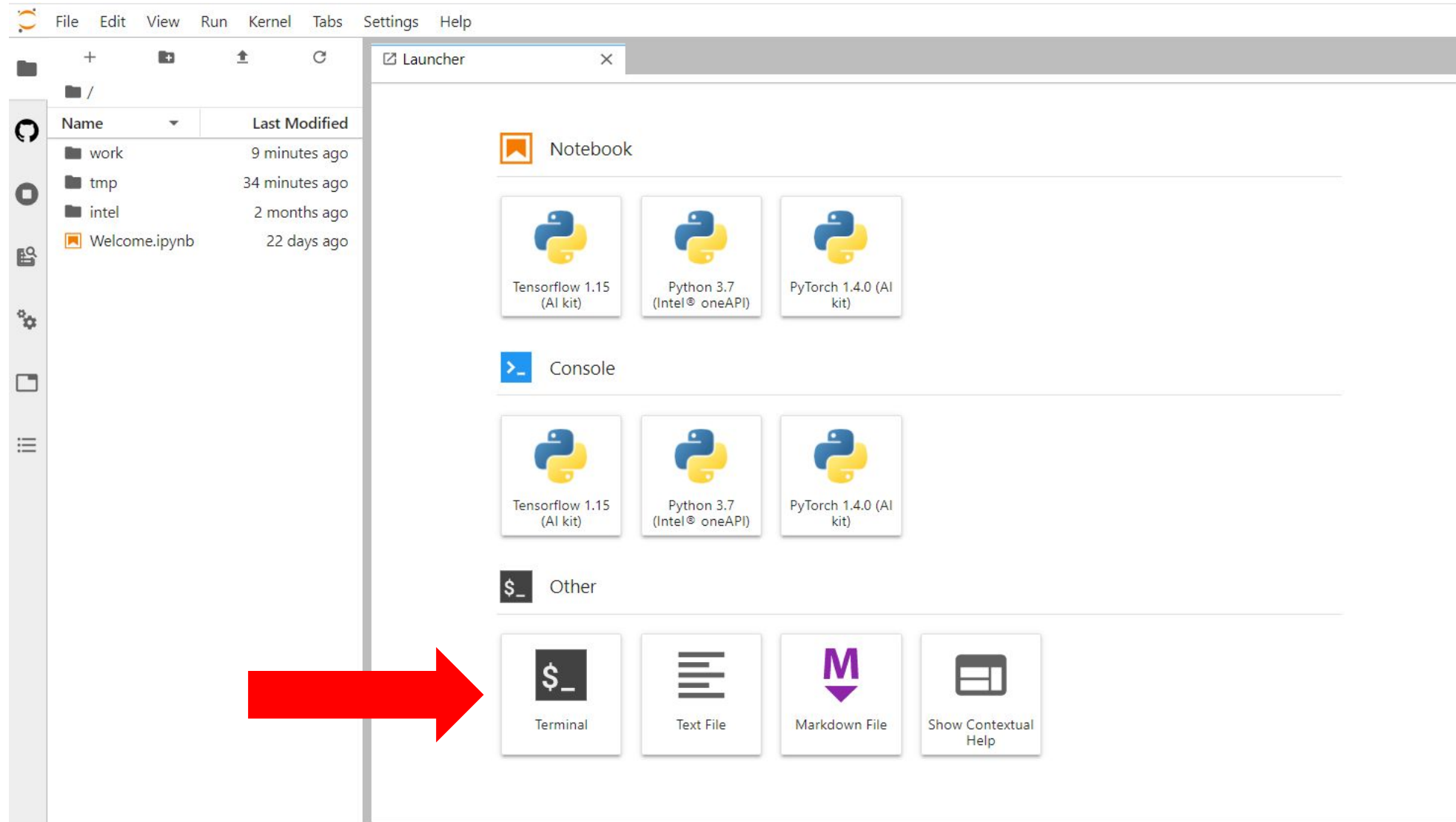
Connect with Jupyter* Notebook

Use Jupyter Notebook to learn about how oneAPI can solve the challenges of programming in a heterogeneous world and understand the Data Parallel C++ (DPC++) language and programming model.



Setup Intel® DevCloud and Jupyter Environment

Launch Jupyter and select Terminal



`git clone https://github.com/oneapi-src/oneAPI-samples.git`

<https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/DPC%2B%2B/Jupyter/oneapi-essentials-training>

SYCL essentials Course



A COMPLETE DPC++ PROGRAM

Single source

- Host code and heterogeneous accelerator kernels can be mixed in same source files

Familiar C++

- Library constructs add functionality, such as:

Construct	Purpose
queue	Work targeting
buffer	Data management
parallel_for	Parallelism

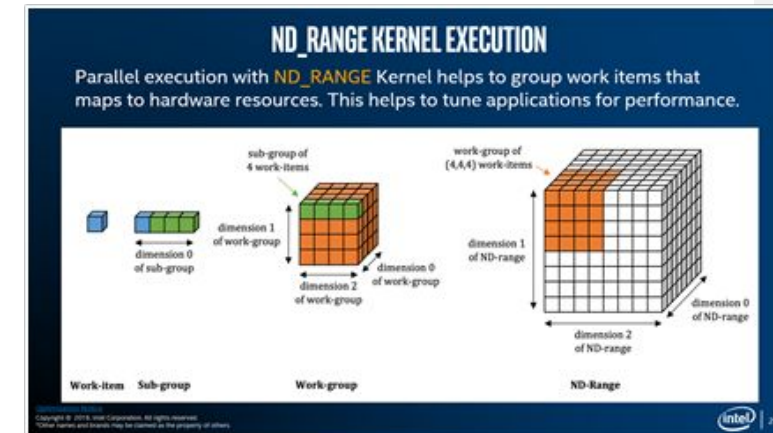
```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

int main() {
    auto R = range<1>{ num };
    buffer<int> A( R );

    queue().submit([&](handler& h) {
        auto out =
        A.get_access<access::mode::write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0]; }); });

    auto result =
    A.get_access<access::mode::read>();
    for (int i=0; i<num; ++i)
        std::cout << result[i] << "\n";

    return 0;
}
```



INTEL OFFLOAD ADVISOR (BETA)

- Starting from a baseline binary (running on CPU):
- Helps defining which sections of the code should run on a given accelerator
- Provides performance projection on accelerators (currently gen9 and gen11)

The screenshot shows the Intel Offload Advisor (Beta) interface. It displays a list of code sections with checkboxes for offloading to different accelerators. It also shows performance metrics and a "Top Offloaders" section.

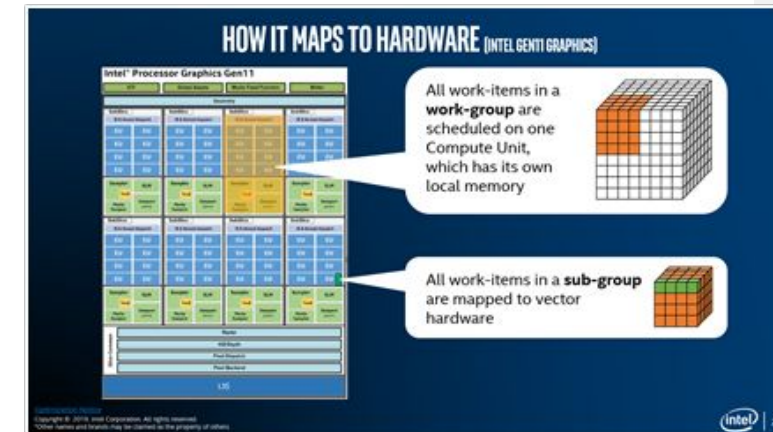
INTEL® VTUNE™ PROFILER: HARDWARE ANALYSIS EXTENDED TO SVMS ARCHITECTURE

DPC++ kernels and their hardware metrics

GPU hardware metrics

- GPU compute Shader
- GPU L3 Cache misses
- GPU Texture Memory

GPU queue and utilization

The screenshot shows the Intel VTune Profiler interface. It displays a table of hardware metrics for DPC++ kernels, including GPU compute Shader, GPU L3 Cache misses, GPU Texture Memory, and GPU queue and utilization.

SYCL Essentials Course Curriculum provides 20 hours of training and exercises using Jupyter Notebooks integrated with Intel® DevCloud

oneAPI's implementation of SYCL

Standards-based, Cross-architecture Language
ISO C++ and Khronos SYCL

Parallelism, productivity and performance for CPUs and Accelerators

- Delivers accelerated computing by exposing hardware features
- Allows code reuse across hardware targets, while permitting custom tuning for specific accelerators
- Provides an open, cross-industry solution to single architecture proprietary lock-in

Based on C++ and SYCL

- Delivers C++ productivity benefits, using common, familiar C and C++ constructs
- Incorporates SYCL from the Khronos Group to support data parallelism and heterogeneous programming

Community Project to drive language enhancements

- Provides extensions to simplify data parallel programming
- Continues evolution through open and cooperative development

Apply your skills to the next innovation, not rewriting software for the next hardware platform

Direct Programming: oneAPI's implementation of SYCL

Community Extensions

Khronos SYCL

ISO C++

What is oneAPI's implementation of SYCL

oneAPI's implementation of SYCL

C++ **and** SYCL* standard **and** extensions

Based on modern C++

- C++ productivity benefits and familiar constructs

Standards-based, cross-architecture

- Incorporates the SYCL standard for data parallelism and heterogeneous programming

A Complete SYCL Program

Single source

- Host code and heterogeneous accelerator kernels can be mixed in same source files

Familiar C++

- Library constructs add functionality, such as:

Construct	Purpose
queue	Work targeting
malloc_shared	Data management
parallel_for	Parallelism

Host code
Accelerator device code

Host code

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
    queue q;
    int *data = malloc_shared<int>(N, q);
    q.parallel_for(N, [=](auto i) {
        data[i] = i;
    }).wait();
    for (int i=0; i<N; i++) std::cout << data[i] << "\n";
    free(data, q);
    return 0;
}
```

Buffer Memory Model

Buffers encapsulate data shared between host and device.

Accessors provide access to data stored in buffers and create data dependences in the graph.

Unified Shared Memory (USM) provides an alternative pointer-based mechanism for managing memory;

```
queue q;  
std::vector<int> v(N, 10);  
{  
    buffer buf(v);  
    q.submit([&](handler& h) {  
        accessor a(buf, h, write_only);  
        h.parallel_for(N, [=](auto i) { a[i] = i; });  
    });  
}  
for (int i = 0; i < N; i++) std::cout << v[i] << " ";
```

Submitting to a Device

- A **device** represents a specific accelerator in the system.
- Work is not submitted to devices directly, but to a **queue** associated with the device.
- Creating a queue for a specific device requires a **device_selector**.

```
default_selector selector;  
// host_selector selector;  
// cpu_selector selector;  
// gpu_selector selector;  
queue q(selector);  
std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
```

Important Classes in SYCL

Class	Functionality
<code>sycl::device</code>	Represents a specific CPU, GPU, FPGA or other device that can execute SYCL kernels.
<code>sycl::queue</code>	Represents a queue to which kernels can be submitted (enqueued). Multiple queues may map to the same <code>sycl::device</code> .
<code>sycl::buffer</code>	Encapsulates an allocation that the runtime can transfer between host and device.
<code>sycl::handler</code>	Used to define a command-group scope that connects buffers to kernels.
<code>sycl::accessor</code>	Used to define the access requirements of specific kernels (e.g. read, write, read-write).
<code>sycl::range</code> , <code>sycl::nd_range</code> <code>sycl::id</code> , <code>sycl::item</code> , <code>sycl::nd_item</code>	Representations of execution ranges and individual execution agents in the range.

Accessor Modes

Access Mode	Description
read_only	Read only Access
write_only	Write-only access. Previous contents not discarded
read_write	Read and Write access

Parallel Kernels

- Parallel Kernel allows multiple instances of an operation to execute in parallel.
- Useful to offload parallel execution of a basic **for-loop** in which each iteration is completely independent and in any order.
- Parallel kernels are expressed using the **parallel_for** function

for-loop in CPU application

```
for(int i=0; i < 1024; i++){  
    a[i] = b[i] + c[i];  
});
```



Offload to accelerator using **parallel_for**

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

Basic Parallel Kernels

The functionality of basic parallel kernels is exposed via range, id and **item** classes

- **range** class is used to describe the iteration space of parallel execution
- **id** class is used to index an individual instance of a kernel in a parallel execution
- **item** class represents an individual instance of a kernel function, exposes additional functions to query properties of the execution range

```
h.parallel_for(range<1>(1024), [=](id<1> idx){  
    // CODE THAT RUNS ON DEVICE  
});
```

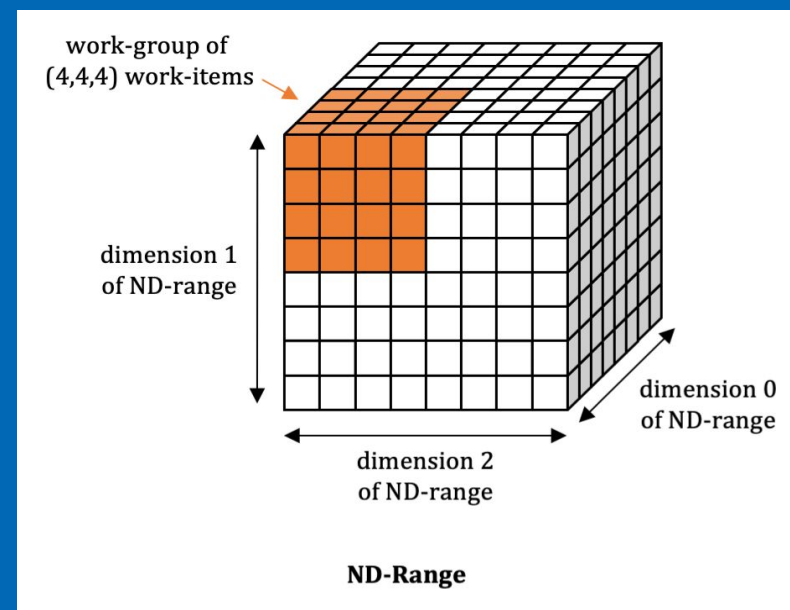
```
h.parallel_for(range<1>(1024), [=](item<1> item){  
    auto idx = item.get_id();  
    auto R = item.get_range();  
    // CODE THAT RUNS ON DEVICE  
});
```

ND-Range Kernels

Basic Parallel Kernels are easy way to parallelize a for-loop but does not allow performance optimization at hardware level.

ND-Range kernel is another way to express parallelism which enable **low level performance tuning** by providing access to local memory and mapping executions to compute units on hardware.


- The entire iteration space is divided into smaller groups called **work-groups**, work-items within a work-group are scheduled on a single compute unit on hardware.
- The grouping of kernel executions into work-groups will allow control of **resource usage** and **load balance** work distribution.



ND-Range Kernels

The functionality of nd_range kernels is exposed via `nd_range` and `nd_item` classes

```
h.parallel_for(nd_range<1>(range<1>(1024), range<1>(64)), [=](nd_item<1> item){  
    auto idx = item.get_global_id();  
    auto local_id = item.get_local_id();  
    // CODE THAT RUNS ON DEVICE  
});
```



- `nd_range` class represents a grouped execution range using global execution range and the local execution range of each work-group.
- `nd_item` class represents an individual instance of a kernel function and allows to query for work-group range and index.

SYCL Code Anatomy

```
void dpcpp_code(int* a, int* b, int* c) {  
    // Setting up a device queue  
    queue q;  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
    //Submit command group function object to the queue  
    q.submit([&](handler &h){  
        //Create device accessors to buffers allocated in global memory  
        accessor A(buf_a, h, read_only);  
        accessor B(buf_b, h, read_only);  
        accessor C(buf_c, h, write_only);  
        //Specify the device kernel body as a lambda function  
        h.parallel_for(range<1>(N), [=](auto i){  
            C[i] = A[i] + B[i];  
        });  
    });  
}
```

Step 1: create a device queue
(developer can specify a device type via device selector or use default selector)

Step 2: create buffers
(represent both host and device memory)

Step 3: submit a command group for (asynchronous) execution

Step 4: create accessors describing how buffer is used on the device

Step 5: specify kernel function and launch parameters (e.g. group size)

Step 6: specify code to run on the device

Kernel invocations
are executed in
parallel

Kernel is invoked
for each element of
the range

Kernel invocation
has access to the
invocation id

Done!
The results are copied to vector c at buf_c buffer destruction

Buffer: sub_buffers

A sub-buffer requires three things, a reference to a parent buffer, a base index, and the range of the sub-buffer.

The main advantage of using the sub-buffers is different kernels can operate on different sub buffers concurrently.

Sub Buffer for one dimensional buffer

Sub buffer for a 2-dimensional buffer

```
buffer B(data, range(N));
```

```
buffer<int> B1(B, 0, range{ N / 2 });
```

```
buffer<int> B2(B, 32, range{ N / 2 });
```

```
buffer<int, 2> b10{range{2, 5}};
```

```
buffer b11{b10, id{0, 0}, range{1, 5}};
```

```
buffer b12{b10, id{1, 0}, range{1, 5}};
```

Sub Buffers

Buffer for Vectors

Create sub buffers B1
and B2

Submit q1 using B1

Submit q2 using B2

Create Host accessors

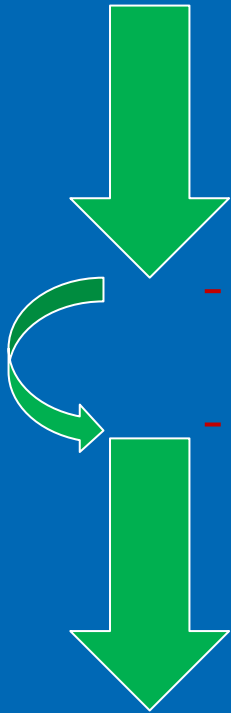
```
int main() {  
    const int N = 64;    const int num1 = 2;    const int num2 = 3;  
    int data[N];  
    for (int i = 0; i < N; i++) data[i] = i;    for (int i = 0; i < N; i++) std::cout << data[i] << " ";  
    buffer B(data, range(N));  
    buffer<int> B1(B, 0, range{ N / 2 });  
    buffer<int> B2(B, 32, range{ N / 2 });  
    queue q1;  
    q1.submit([&](handler& h) {  
        accessor a1(B1, h);  
        h.parallel_for(N/2, [=](auto i) { a1[i] *= num1; });  
    });  
    queue q2;  
    q2.submit([&](handler& h) {  
        accessor a2(B2, h);  
        h.parallel_for(N/2, [=](auto i) { a2[i] *= num2; });  
    });  
    host_accessor b1(B1, read_only);  
    host_accessor b2(B2, read_only);  
    return 0;  
}
```

Asynchronous Execution

Host

Host code execution

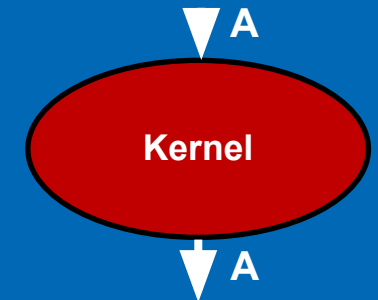
Enqueues kernel to graph, and keeps going



```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
    std::vector<int> data(N);
    {
        buffer A(data);
        queue q;
        q.submit([&](handler& h) {
            accessor out(A, h, write_only);
            h.parallel_for(N, [=](auto i) {
                out[i] = i;
            });
        });
    }
    for (int i=0; i<N; ++i) std::cout << data[i];
}
```

Graph

Graph executes asynchronously to host program



Asynchronous Execution

```
int main() {  
    auto R = range<1>{ num };  
    buffer<int> A{ R }, B{ R };  
    queue q;
```

```
    q.submit([&](handler& h) {  
        accessor out(A, h, write_only);  
        h.parallel_for(R, [=](id<1> i) {  
            out[i] = i; }); });
```

} Kernel 1

```
    q.submit([&](handler& h) {  
        accessor out(A, h, write_only);  
        h.parallel_for(R, [=](id<1> i) {  
            out[i] = i; }); });
```

} Kernel 2

```
    q.submit([&](handler& h) {  
        accessor out(B, h, write_only);  
        h.parallel_for(R, [=](id<1> i) {  
            out[i] = i; }); });
```

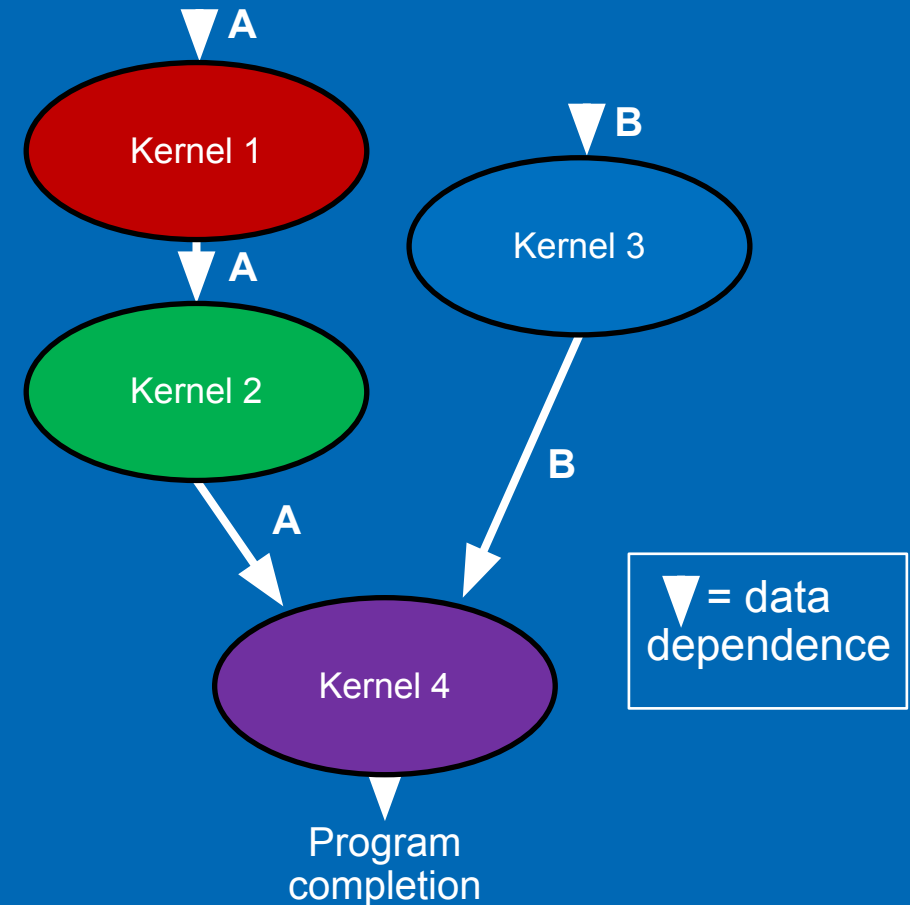
} Kernel 3

```
    q.submit([&](handler& h) {  
        accessor in(A, h, read_only);  
        accessor inout(B, h);  
        h.parallel_for(R, [=](id<1> i) {  
            inout[i] *= in[i]; }); });
```

} Kernel 4

}

Data and control dependences
are resolved by the runtime



Synchronization – Host Accessors

```
#include <CL/sycl.hpp>
using namespace sycl;
constexpr int N = 16;

int main() {
    std::vector<double> v(N, 10);
    queue q;

    buffer buf(v);
    q.submit([&](handler& h) {
        accessor a(buf, h)
        h.parallel_for(N, [=](auto i) {
            a[i] -= 2;
        });
    });

    host_accessor b(buf, read_only);
    for (int i = 0; i < N; i++)
        std::cout << b[i] << "\n";
    return 0;
}
```

Buffer takes **ownership** of the **data** stored in vector.

Creating host accessor is a **blocking call** and will only return after all enqueued kernels that modify the same buffer in any queue completes execution and the **data is available to the host** via this host accessor.

Synchronization – Buffer Destruction

```
#include <CL/sycl.hpp>
using namespace sycl;
constexpr int N=16;

void dpcpp_code(std::vector<double> &v, queue &q){
    buffer buf(v);
    q.submit([&](handler& h) {
        accessor a(buf, h);
        h.parallel_for(N, [=](auto i) {
            a[i] -= 2;
        });
    });
}

int main() {
    std::vector<double> v(N, 10);
    queue q;
    dpcpp_code(v,q);
    for (int i = 0; i < N; i++)
        std::cout << v[i] << "\n";
    return 0;
}
```

Buffer creation happens within a **separate function scope**.

When execution advances beyond this function scope, **buffer destructor is invoked** which relinquishes the ownership of data and **copies back the data to the host** memory.

Custom Device Selector

The following code shows derived **device_selector** that employs a device selector heuristic. The selected device prioritizes a GPU device because the integer rating returned is higher than for CPU or other accelerator.

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class my_device_selector : public device_selector {
public:
    int operator()(const device& dev) const override {
        int rating = 0;
        if (dev.is_gpu() & (dev.get_info<info::device::name>().find("Intel") != std::string::npos))
            rating = 3;
        else if (dev.is_gpu()) rating = 2;
        else if (dev.is_cpu()) rating = 1;
        return rating;
    };
};

int main() {
    my_device_selector selector;
    queue q(selector);
    std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
    return 0;
}
```

Execution Graph Scheduling

Mechanism to achieve proper sequencing of kernels, and data movement in a SYCL application.

- Read-after-Write (RAW) : Occurs when one task needs to read data produced by a different task.
- Write-after-Read (WAR) : Occurs when one task needs to update data after another task has read it.
- Write-after-Write (WAW) : Occurs when two tasks try to write the same data.

```

int main() {
    queue Q;
    //Create Buffers
    buffer A{a};
    buffer B{b};
    buffer C{c};

    Q.submit([&](handler &h) {
        accessor accA(A, h, read_only);
        accessor accB(B, h, write_only);
        h.parallel_for( // computeB
            N, [=](id<1> i) { accB[i] = accA[i] + 1; });
    });

    Q.submit([&](handler &h) {
        accessor accA(A, h, read_only);
        h.parallel_for( // readA
            N, [=](id<1> i) {
                // Useful only as an example
                int data = accA[i];
            });
    });

    Q.submit([&](handler &h) {
        // RAW of buffer B
        accessor accB(B, h, read_only);
        accessor accC(C, h, write_only);
        h.parallel_for( // computeC
            N, [=](id<1> i) { accC[i] = accB[i] + 2; });
    });

    // read C on host
    host_accessor host_accC(C, read_only);
    std::cout << "\n";
    return 0;
}

```

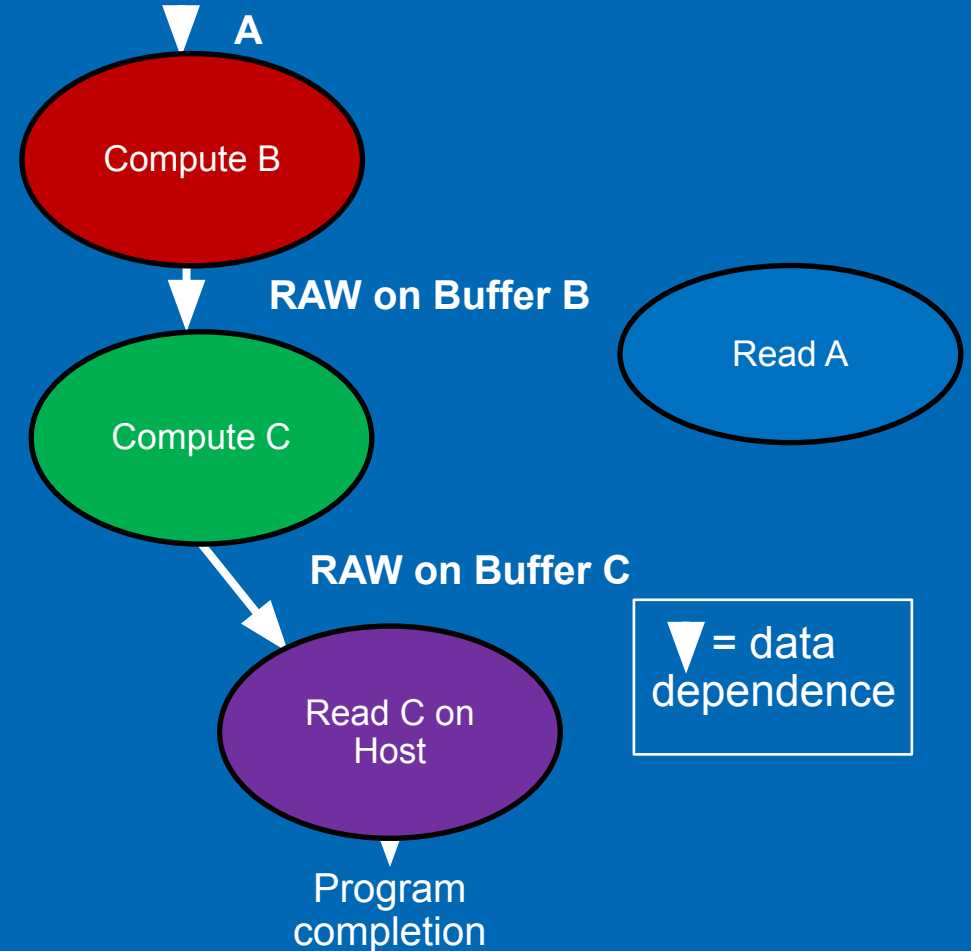
Kernel 1

Kernel 2

Kernel 3

Read after Write (RAW)

Automatic data and control dependence resolution!



Write After Read and Write after Write

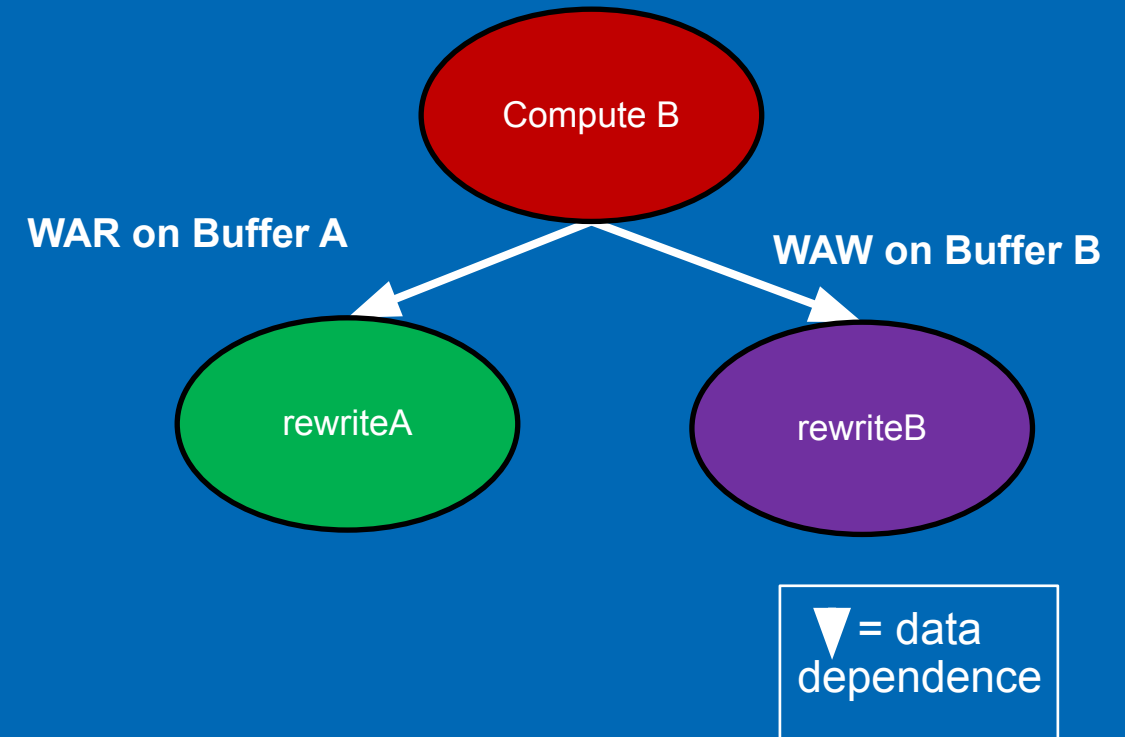
```
queue Q;  
buffer A{a};  
buffer B{b};  
Q.submit([&](handler &h) {  
    accessor accA(A, h, read_only);  
    accessor accB(B, h, write_only);  
    h.parallel_for( // computeB  
        N, [=](id<1> i) {  
            accB[i] = accA[i] + 1;  
        });  
});  
Q.submit([&](handler &h) {  
    // WAR of buffer A  
    accessor accA(A, h, write_only);  
    h.parallel_for( // rewriteA  
        N, [=](id<1> i) {  
            accA[i] = 21 + 21;  
        });  
});  
Q.submit([&](handler &h) {  
    // WAW of buffer B  
    accessor accB(B, h, write_only);  
    h.parallel_for( // rewriteB  
        N, [=](id<1> i) {  
            accB[i] = 30 + 12;  
        });  
});  
host_accessor host_accA(A, read_only);  
host_accessor host_accB(B, read_only);
```

} Kernel 1

} Kernel 2

} Kernel 3

Automatic data and control
dependence resolution!



Linear dependency chain graphs and Y pattern Graphs

- Linear dependence chains where one task executes after another
 - First node represents the initialization of data.
 - Second node presents the reduction operation that will accumulate the data.
- “Y” pattern we independently initialize two different pieces of data.
 - An addition kernel will sum the two vectors together.
 - Finally, the last node in the graph accumulates the result into a single value.

Linear Dependence Using In-order queue

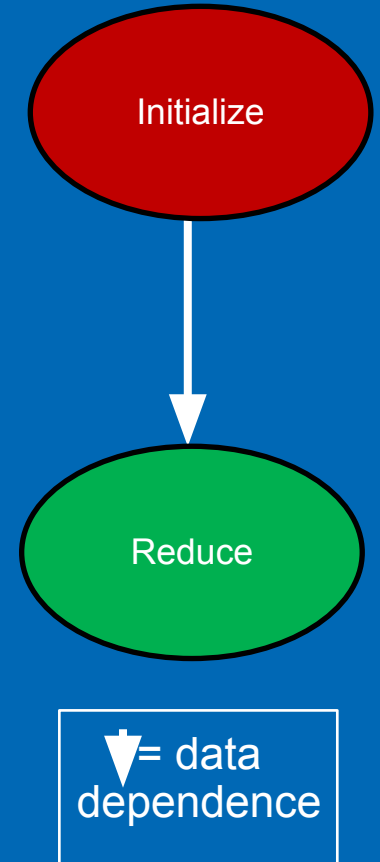
Create In-order
queue

Initialize the data
in Kernel 1

Kernel 2 sums up
the elements

```
constexpr int N = 42;

int main() {
    queue Q{property::queue::in_order()};
    int *data = malloc_shared<int>(N, Q);
    Q.parallel_for(N, [=](id<1> i) { data[i] = 1; });
    Q.single_task([=]() {
        for (int i = 1; i < N; i++)
            data[0] += data[i];
    });
    Q.wait();
    assert(data[0] == N);
    return 0;
}
```



Y Pattern using in-order queues

We can see a “Y” pattern using in-order queues in the below example

In-Order Queue

```
constexpr int N = 42;

int main() {
    queue Q{property::queue::in_order()};

    int *data1 = malloc_shared<int>(N, Q);
    int *data2 = malloc_shared<int>(N, Q);

    Q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });
    Q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });
    Q.parallel_for(N, [=](id<1> i) { data1[i] += data2[i]; });

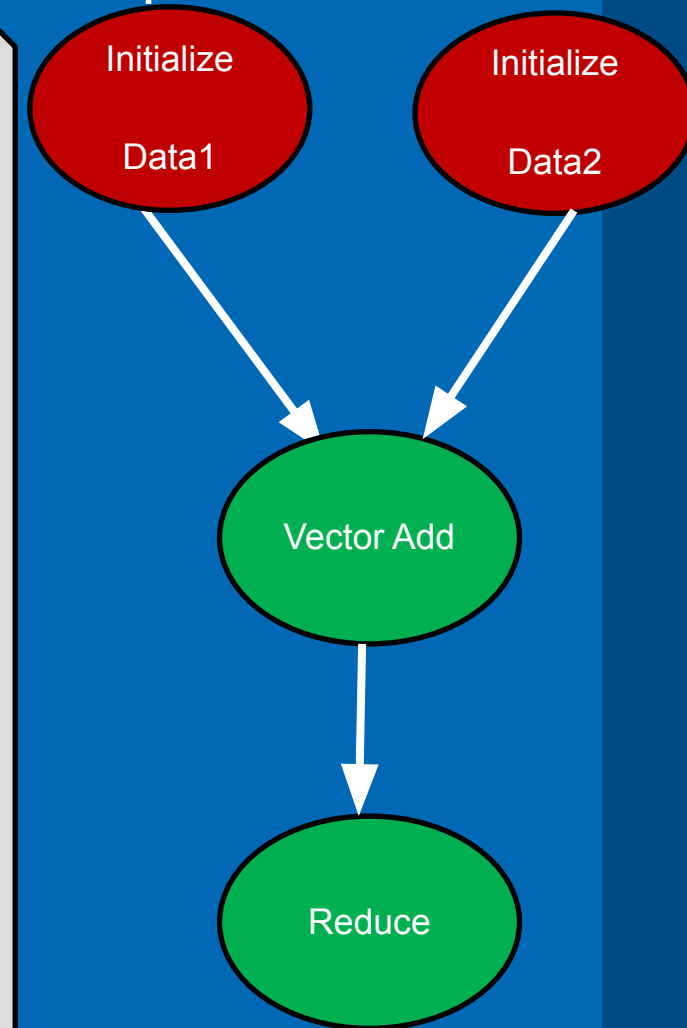
    Q.single_task([=]() {
        for (int i = 1; i < N; i++)
            data1[0] += data1[i];

        data1[0] /= 3;
    });
    Q.wait();

    assert(data1[0] == N);
    return 0;
}
```

Kernel 3 is dependant
on Kernel1 and Kernel2

The final kernel sums
up the elements of the
first array



Advanced SYCL Topics

SYCL 2020 Features

- Agenda

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

- Hands On

- USM and solving data dependency
- Sub-group collectives and shuffle operations
- Simplification with Reduction extension

Learning Objectives

Use SYCL features like **Unified Shared Memory** to simplify heterogeneous programming

Understand advantages of using **Sub-groups** in SYCL

Simplify **reductions** in heterogeneous programming

C++ + SYCL* + Extensions

SYCL 2020 Features:

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

Main goals of new SYCL 2020 Features is to **simplify programming** and **achieve performance** by exposing hardware features.

Language Simplification

Code snippet below shows how SYCL* code can be simplified

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

SYCL

Buffer Simplification

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

Accessor simplification

parallel_for simplification

SYCL 2020

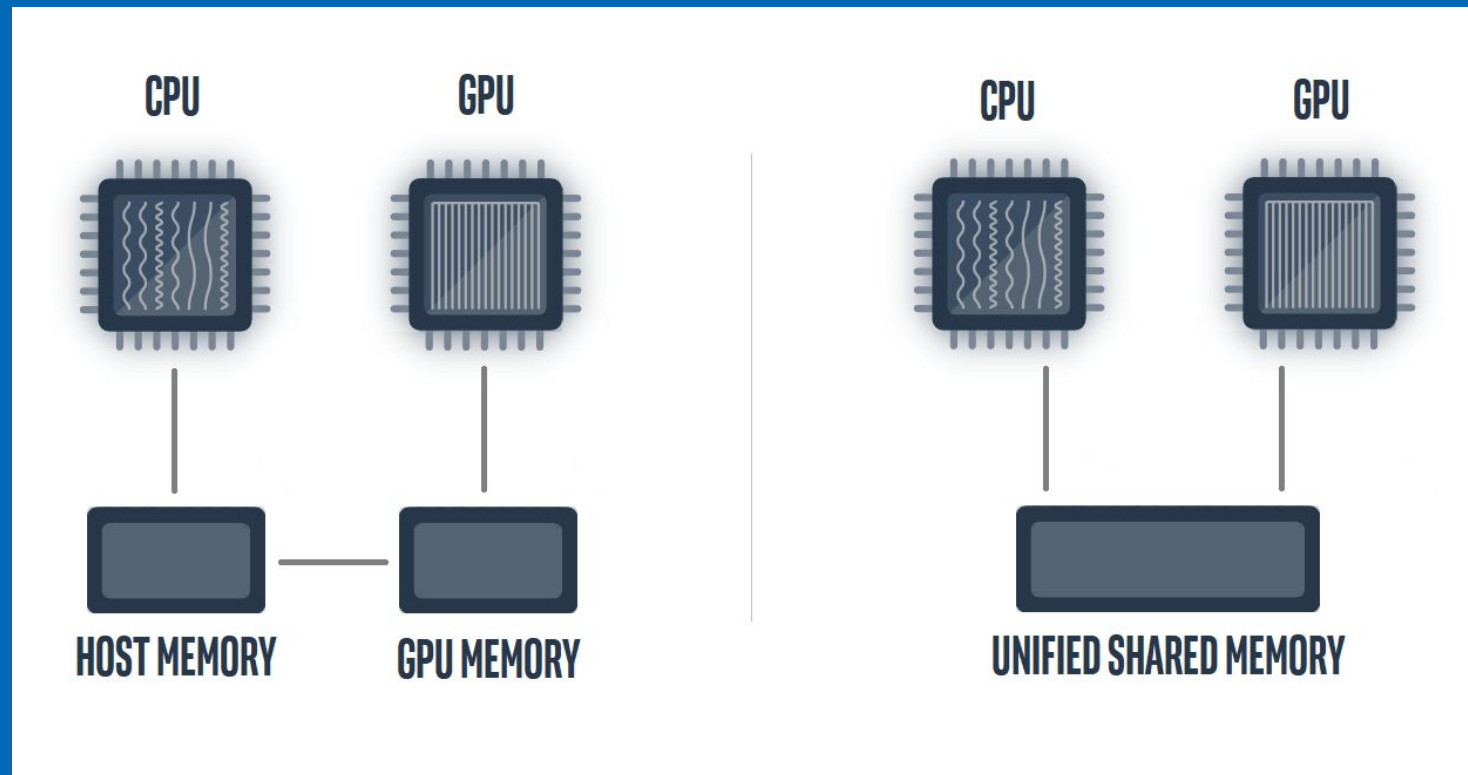
Simple and
Less Verbose

Unified Shared Memory (USM)

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

Developer View of USM

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



Unified Shared Memory

Unified Shared Memory can be setup as follows:

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

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

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

Unified Shared Memory

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

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

Setup Unified Shared Memory →

Host can initialize →

Device can modify →

Host has output →

SYCL Buffers Method

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

```
queue q;

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

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

{

    buffer<int, 1> buf(data, range<1>(N));
    q.submit([& (handler &h){

        auto A = buf.get_access<access::mode::read_write>(h);
        h.parallel_for(range<1>(N), [=](id<1> i){

            A[i] += 1;

        });

    });

}

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

free(data);
```

Host memory setup →

Host can initialize →

Create buffer →

Create accessor →

Device can modify →

Buffer destruction →

Host has output →

WHY Unified Shared Memory?

The SYCL* standard provides a **Buffer memory abstraction**

- Powerful and elegantly expresses data dependences

However...

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

USM provides a pointer-based alternative

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

Unified Shared Memory (USM)

There are three ways to create USM allocations:

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

USM – Explicit Data Transfer

Gives developer full control of moving memory between host and device

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

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

Make any data modification on device

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

```
queue q;

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

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

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

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

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

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

USM – Implicit Data Transfer

Memory movement between host and device is done implicitly

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

Make any data modification on device

Host has access to the device modified memory

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

Hands-on Coding on Intel DevCloud

USM Implicit and Explicit Data Movement

Unified Shared Memory – When to use it?

SYCL* **Buffers are powerful** and elegant

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

USM provides a familiar pointer-based C++ interface

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

USM – Data Dependency in tasks

- When using unified shared memory in multiple kernel tasks, **dependences** between operations must be specified using **events**.
- Programmers may either explicitly wait on **event** objects or use the **depends_on** method inside a command group to specify a list of events that must complete before a task may begin.

USM – Data Dependency in tasks

Explicit `wait()` used to ensure data dependency is maintained

*Note that `wait()` will **block execution** on host



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

USM – Data Dependency in tasks

Use `in_order` queue property for the queue

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



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

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

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

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

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

USM – Data Dependency in tasks

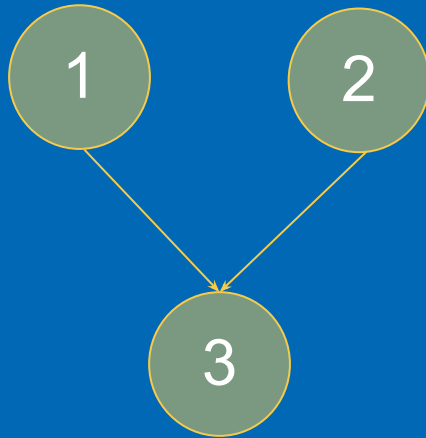
Use `depends_on()` method to let command group handler know that specified event should be complete before specified task can execute.



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

USM – Data Dependency in tasks

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



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

Hands-on Coding on Intel DevCloud

Handling Data Dependency when using USM

Unified Shared Memory

- Summary

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

Sub Groups

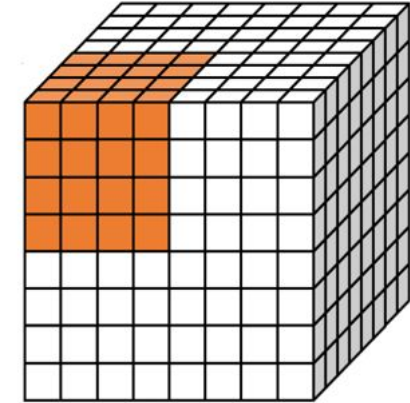
Sub-groups are **subset of the work-items** that are executed simultaneously or with additional scheduling guarantees.

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

How it maps to Hardware (INTEL GEN11 GRAPHICS)

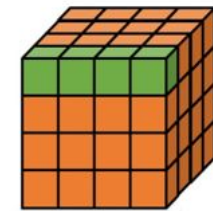


All work-items in a **work-group** are scheduled on one subslice, which has its own local memory.



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

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

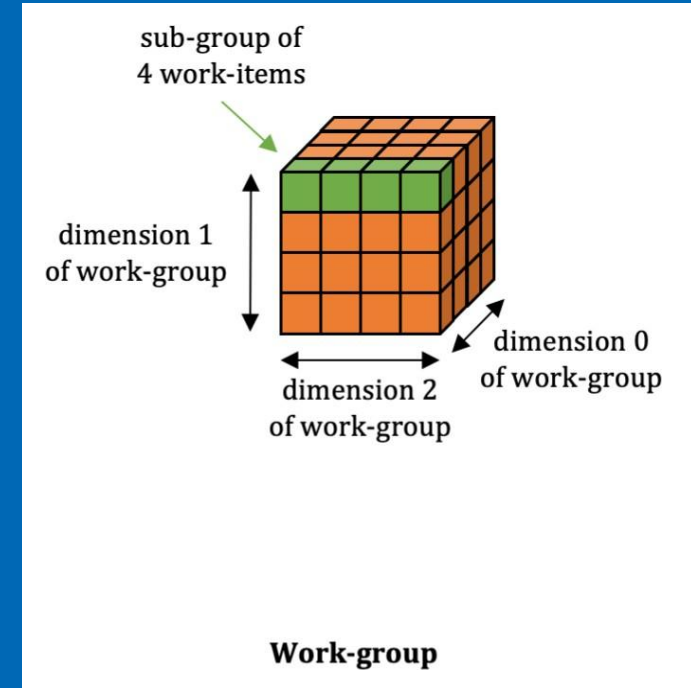


Sub Groups

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

Why use Sub-groups?

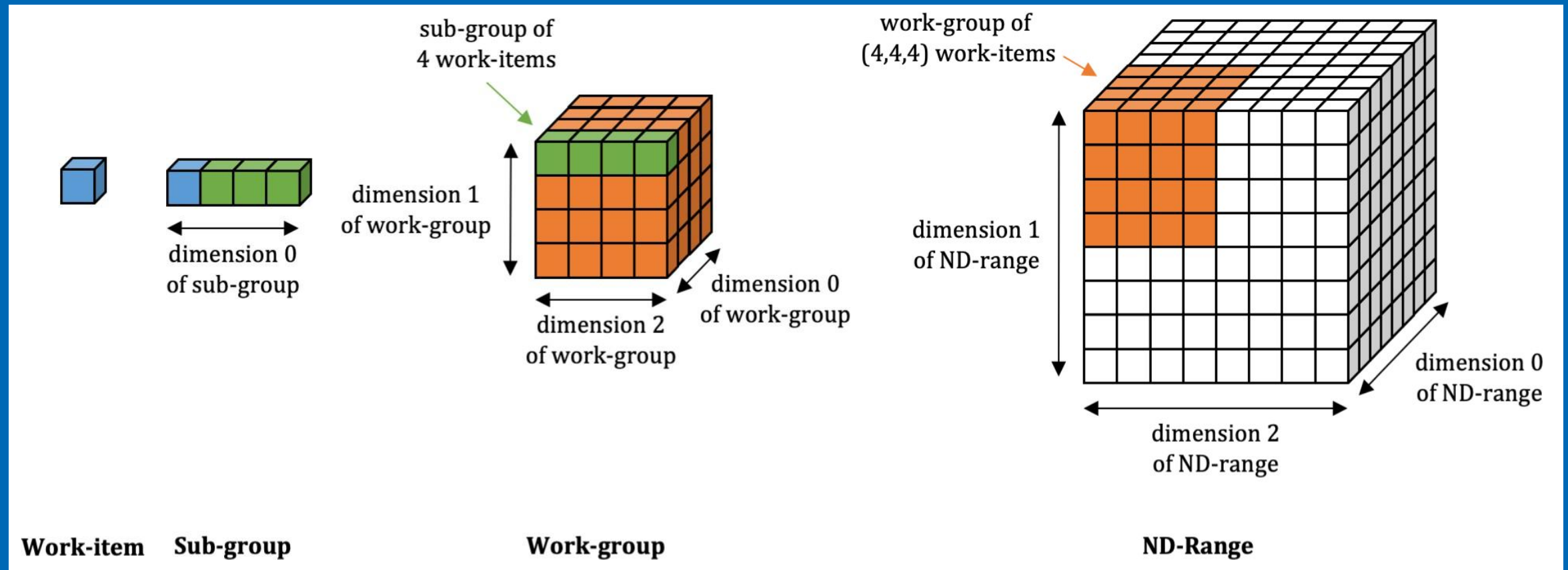
- Work-items in a sub-group can communicate directly using **shuffle operations**, without repeated access to local or global memory, and may provide better performance.
- Work-items in a sub-group have access to **sub-group collectives**, providing fast implementations of common parallel patterns.



Sub Groups

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

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



Sub Groups

sub_group class

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

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

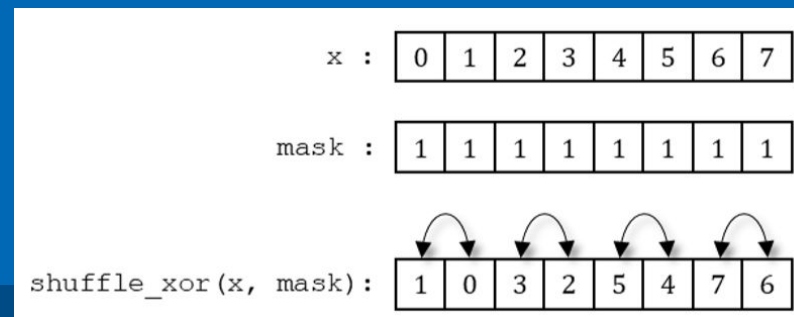
```
q.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){  
  
    auto sg = item.get_sub_group();  
  
    // KERNEL CODE  
  
});
```

Sub Groups

Sub-Group Shuffles

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

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){  
    auto sg = item.get_sub_group();  
    size_t i = item.get_global_id(0);  
  
    /* Shuffles */  
    //data[i] = select_from_group(sg, data[i], 1);  
    //data[i] = shift_group_left(sg, data[i], 1);  
    //data[i] = shift_group_right(sg, data[i], 1);  
    data[i] = permute_group_by_xor(sg, data[i], 1);  
});
```



Sub Groups

Sub-Group Group Algorithms

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

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

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

Specifying the Sub-Group Size

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

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

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

Hands-on Coding on Intel DevCloud

Sub-Group Shuffles and Collectives

Sub Groups

- **Summary**

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

Reductions

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

- **Parallelizing reductions** can be tricky because of the nature of computation and accelerator hardware.
- SYCL 2020 introduces a **simplified** approach for reductions in heterogenous programming

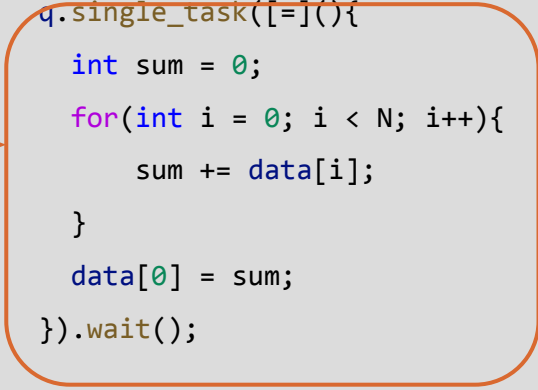
Simple Reduction

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

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

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

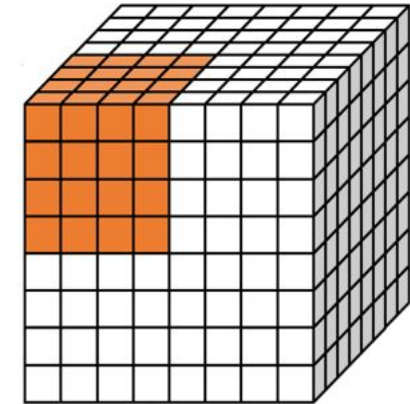
```
queue q;  
int *data = malloc_shared<int>(N, q);  
for (int i = 0; i < N; i++) data[i] = i;  
  
q.single_task([=](){  
    int sum = 0;  
    for(int i = 0; i < N; i++){  
        sum += data[i];  
    }  
    data[0] = sum;  
}).wait();  
  
std::cout << "Sum = " << data[0] << std::endl;
```



Parallelizing Reductions



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



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

Work-Group Reduction

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

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

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

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

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

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

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

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

Simplified Reduction

SYCL 2020 introduces reduction object in `parallel_for` **reduction** object in `parallel_for` encapsulates the reduction variable, an optional operator identity and the reduction operator.

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

```
queue q;  
auto data = malloc_shared<int>(N, q);  
for (int i = 0; i < N; i++) data[i] = i;  
  
auto sum = malloc_shared<int>(1, q);  
sum[0] = 0;  
  
q.parallel_for(nd_range<1>{N, B},  
               reduction(sum, plus<>()), [=](nd_item<1> it, auto& sum) {  
    int i = it.get_global_id(0);  
    sum += data[i];  
}).wait();  
  
std::cout << "Sum = " << sum[0] << std::endl;
```

Multiple Reductions in one kernel

```
myQueue.submit([&](handler& cgh) {  
  
    // Input values to reductions are standard accessors (or USM pointers)  
    auto inputValues = accessor(valuesBuf, cgh);  
  
    // Create temporary objects describing variables with reduction semantics  
    auto sumReduction = reduction(sumBuf, cgh, plus<>());  
    auto maxReduction = reduction(maxBuf, cgh, maximum<>());  
  
    // parallel_for performs two reduction operations  
    cgh.parallel_for(range<1>{1024},  
        sumReduction, maxReduction,  
        [=](id<1> idx, auto& sum, auto& max) {  
            sum += inputValues[idx];  
            max.combine(inputValues[idx]);  
        });  
});
```

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

Optimization Notice
Optimization Notice

Copyright © 2019, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Hands-on Coding on Intel DevCloud

Reduction in SYCL 2020

Reductions

- **Summary**
 - What are Reductions?
 - Parallelizing Reductions
 - Reduction kernel to simplify programming

Recap

- oneAPI solves the challenges of programming in a heterogeneous world
- Take advantage of oneAPI solutions to enable your workflows
- Use the Intel® DevCloud to test-drive oneAPI tools and libraries
- Introduced to SYCL language and programming model
- Important Classes for SYCL application
- Device selection and offloading kernel workloads
- SYCL Buffers, Accessors, Command Group handler, lambda code as kernel
- Utilize different types of data dependences that are important for ensuring execution of graph scheduling
- What is Unified Shared Memory
- What are Sub Groups
- How to take advantage of Reductions



Notices &

- This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.
- The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. No product or component can be absolutely secure. Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.
- INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Copyright ©, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and OpenVINO are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

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