Droidcon MEC Hackathon 2020

oneAPI essentials

FPGA Development with Intel® FPGA Add-on for oneAPI Base Toolkit

Speakers

- Benjamin J Odom (INTEL)
- Adonay Berhe (INTEL)

Agenda

- Agenda:
  a) Introduction & Overview to oneAPI
  b) Introduction to the Intel® DevCloud
  c) Introduction to Jupyter notebooks used for training
  d) Introduction to Data Parallel C++
  e) Introduction to USM
  f) Complex Number multiplication example
- Hands On: Hough Transform using FPGA Add-on for oneAPI

Why do we care about Heterogeneous computing?

The term refers to “systems that use more than one kind of processor or cores.” Wikipedia

This gives developers gains in:

- ✓ Performance
- ✓ Power consumption
- ✓ Latency
- ✓ IO Flexibility
- ✓ Memory bandwidth
- ✓ Off-load functionalities

Programming Challenges
for Multiple Architectures

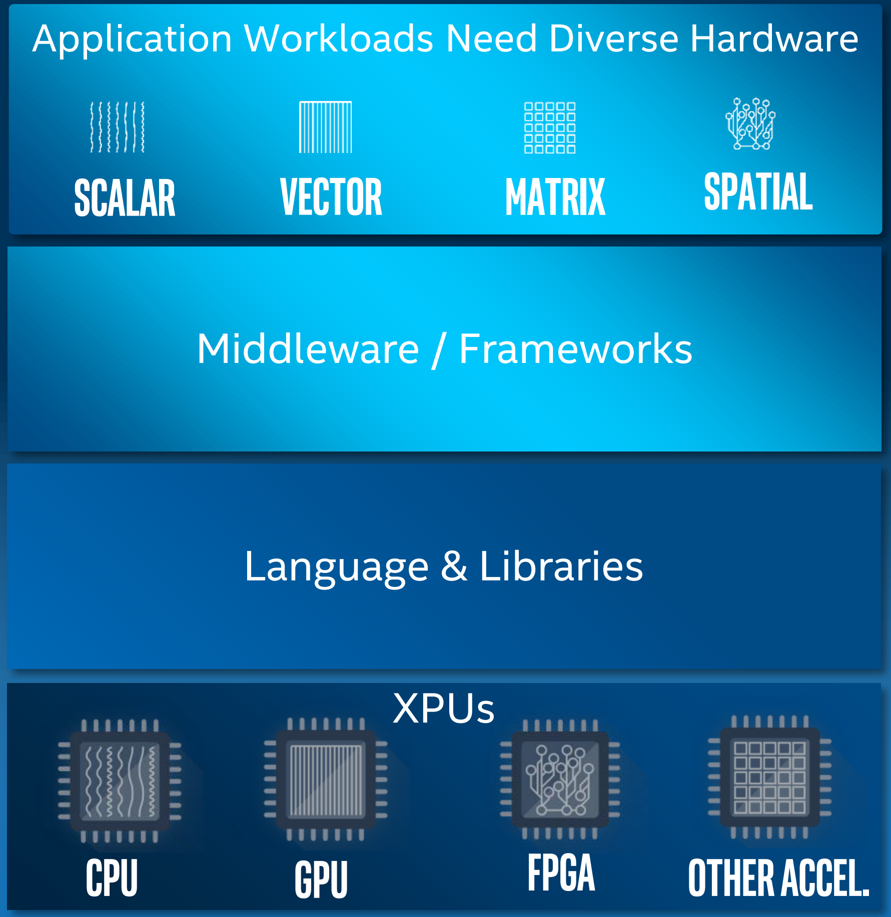
Growth in specialized workloads

No common programming language or APIs

Inconsistent tool support across platforms

Each platform requires unique software investment

Diverse set of data-centric hardware required



Introducing
oneAPI

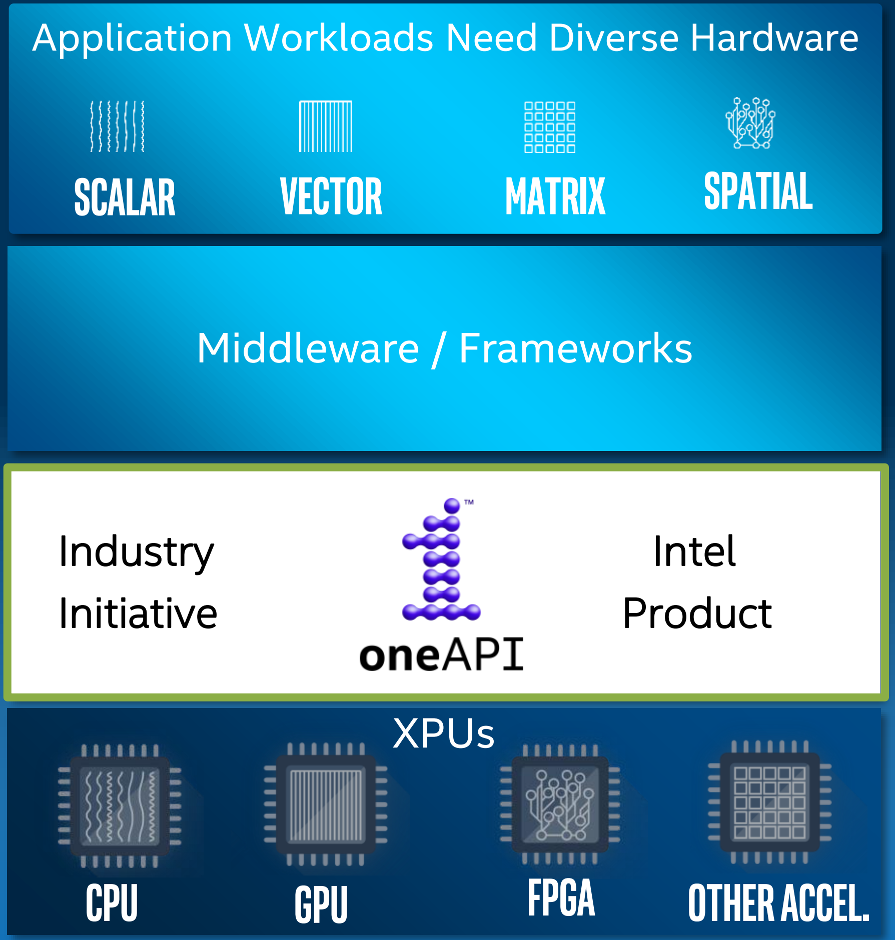
Unified programming model to simplify development across diverse architectures

Unified and simplified language and libraries for expressing parallelism

Uncompromised native high-level language performance

Based on industry standards and open specifications

Interoperable with existing HPC programming models



OneAPI Industry Initiative

Alternative to Single-Vendor Solution

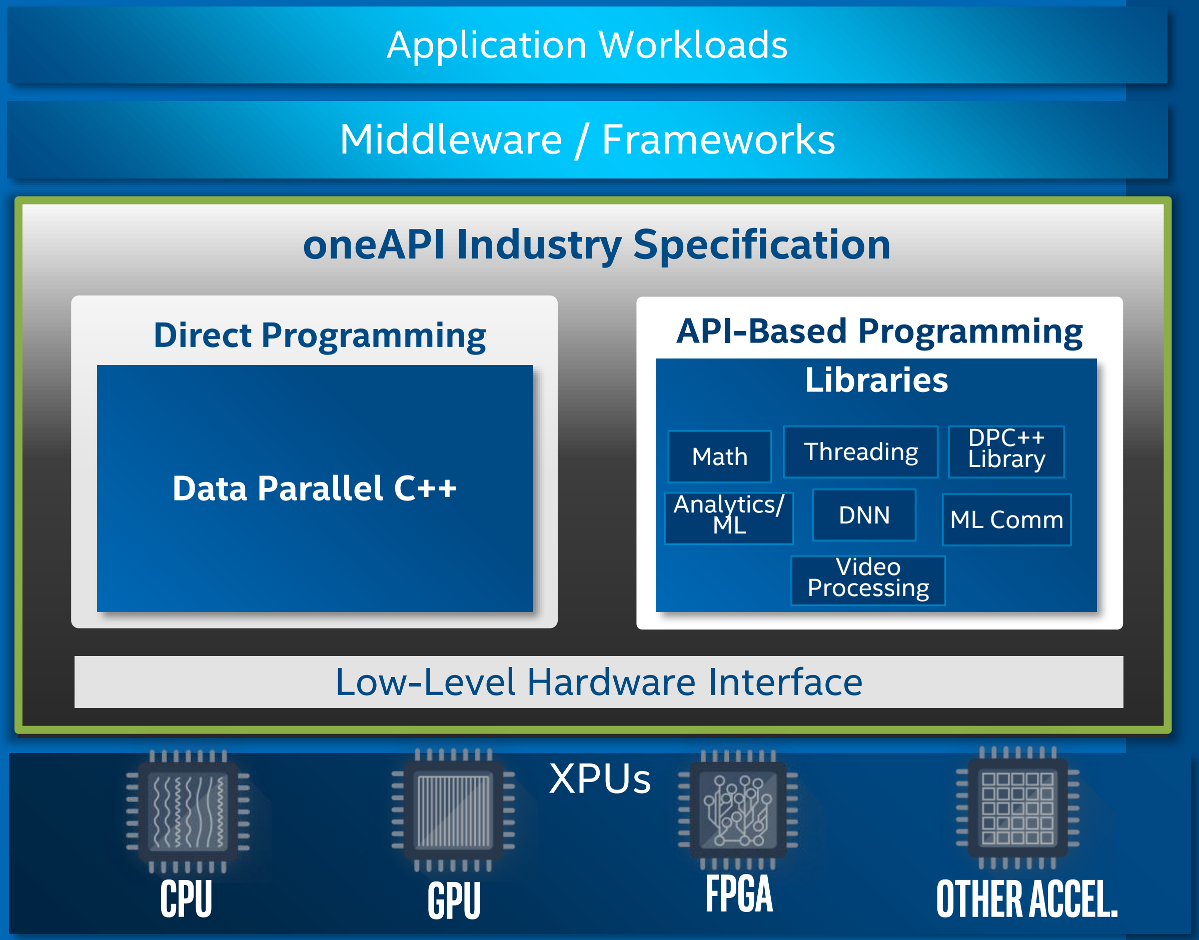
A standards based cross-architecture language, DPC++, based on C++ and SYCL

Powerful APIs designed for acceleration of key domain-specific functions

Low-level hardware interface to provide a hardware abstraction layer to vendors

Open standard to promote community and industry support

Enables code reuse across architectures and vendors



Application Workloads

Middleware / Frameworks

oneAPI Industry Specification

Direct Programming

Data Parallel C++

API-Based Programming

Libraries

Math
Threading
DPC++ Library
Analytics/ ML
DNN
ML Comm
Video Processing

Low-Level Hardware Interface

XPUs

Visit for more details

Some capabilities may differ per architecture and custom-tuning will still be required.

Data Parallel C++

Standards-based, Cross-architecture Language

Get functional quickly. Then analyze and tune.

Parallelism, productivity and performance for CPUs and Accelerators

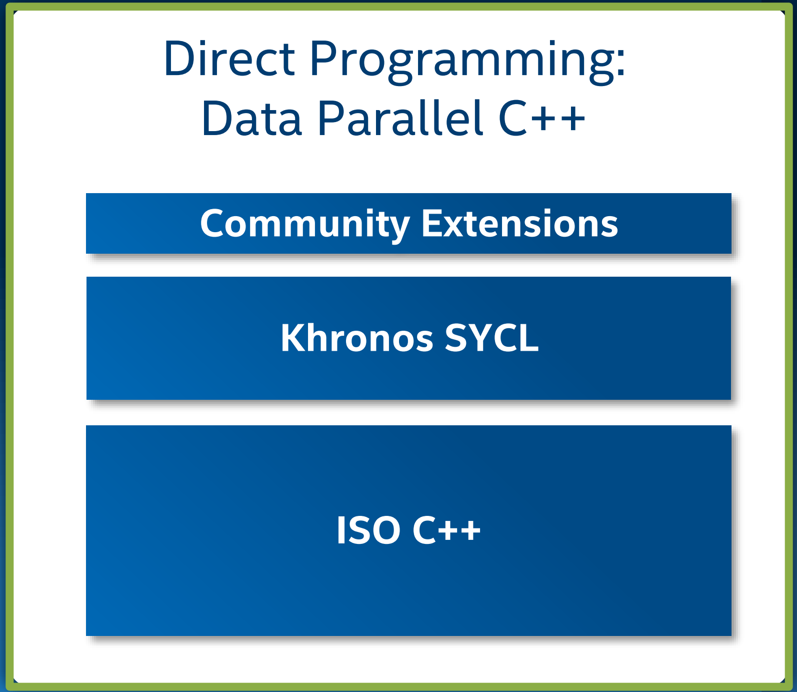
- Allows code reuse across hardware targets, while permitting custom tuning for a specific accelerator
- Open, cross-industry alternative to single architecture proprietary language

Based on ISO C++ and Khronos SYCL

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

Community Project to drive language enhancements

- Extensions to simplify data parallel programming
- Open and cooperative development for continued evolution


Direct Programming:
Data Parallel C++

Community Extensions

Khronos SYCL

ISO C++

The open source and Intel beta DPC++ compiler currently supports hardware including Intel CPUs, GPUs, and FPGAs.
Codeplay announced a .

What is Data Parallel C++?

Data Parallel C++

= C++ and SYCL* standard and extensions

Based on modern C++

- C++ productivity benefits and familiar constructs

Standards-based, cross-architecture

- Incorporates the SYCL standard for data parallelism and heterogeneous programming

DPC++ Extends SYCL 1.2.1

Enhance Productivity

- Simple things should be simple to express
- Reduce verbosity and programmer burden

Enhance Performance

- Give programmers control over program execution
- Enable hardware-specific features

DPC++: Fast-moving open collaboration feeding into the SYCL* standard

- Open source implementation with goal of upstream LLVM
- DPC++ extensions aim to become core SYCL*, or Khronos* extensions

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

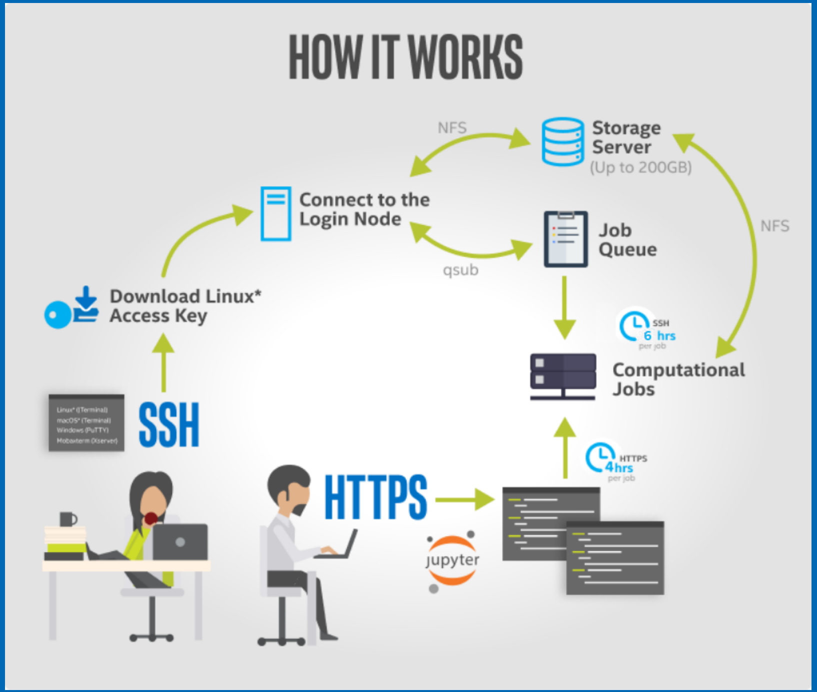
Setup Intel® DevCloud and Jupyter Environment

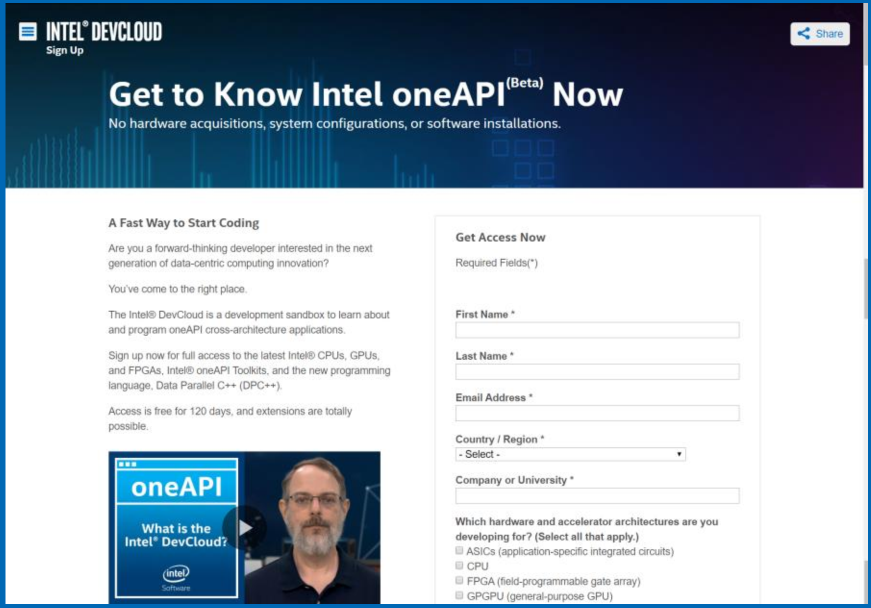
Intel® DevCloud for oneAPI

Overview

- A development sandbox to develop, test and run workloads across a range of Intel CPUs, GPUs, and FPGAs using Intel® oneAPI beta software
- A fast way to start coding
- Try the oneAPI toolkits, compilers, performance libraries, and tools
- Get 120 days of free access to the latest Intel® hardware and oneAPI software
- No downloads; No hardware acquisition; No installation

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





Get to Know Intel oneAPI$^{(Beta)}$ Now

No hardware acquisitions, system configurations, or software installations.

A Fast Way to Start Coding

Are you a forward-thinking developer interested in the next generation of data-centric computing innovation?

You've come to the right place.

The Intel® DevCloud is a development sandbox to learn about and program oneAPI cross-architecture applications.

Sign up now for full access to the latest Intel® CPUs, GPUs, and FPGAs, Intel® oneAPI Toolkits, and the new programming language, Data Parallel C++ (DPC++).

Access is free for 120 days, and extensions are totally possible.

Get Access Now

Required Fields(*)

First Name *

Last Name *

Email Address *

Country / Region *
- Select -

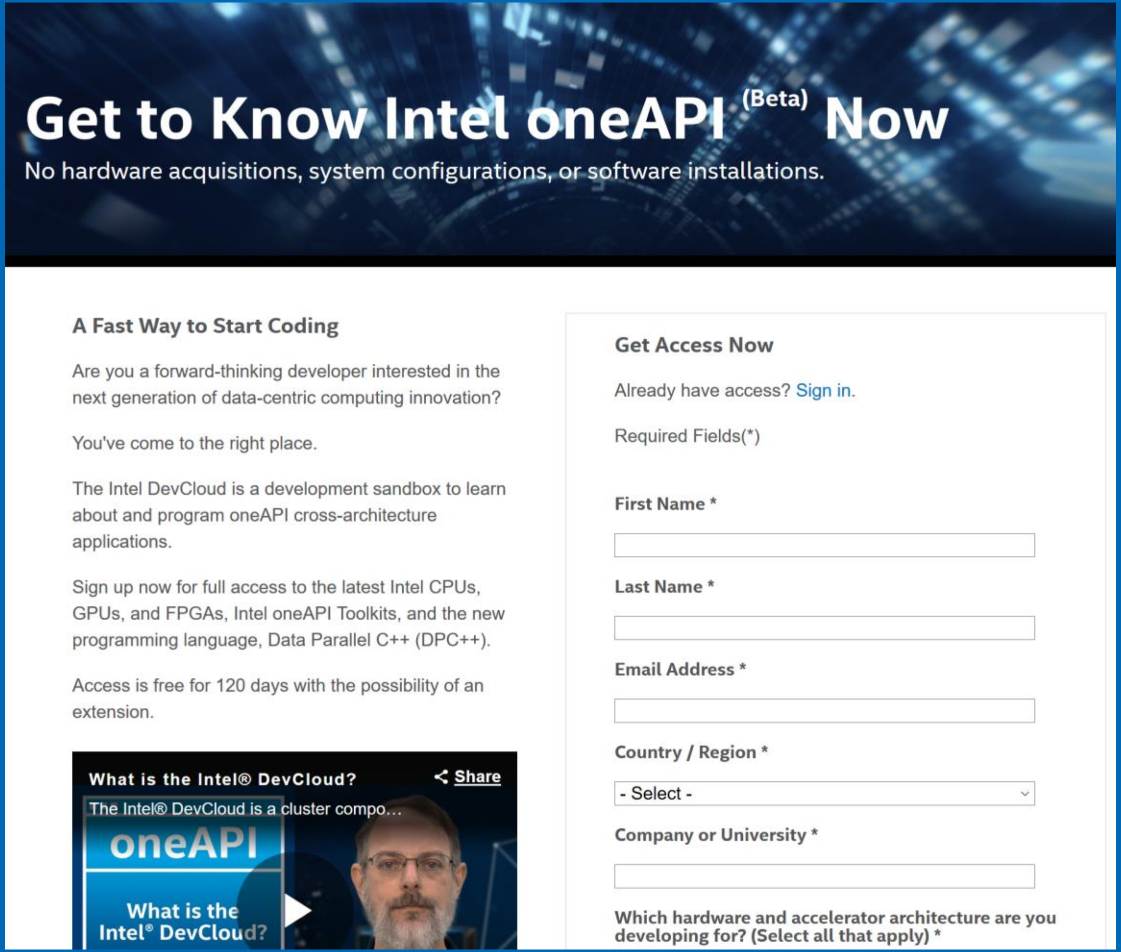
Company or University *

Which hardware and accelerator architectures are you developing for? (Select all that apply.)
☐ ASICs (application-specific integrated circuits)
☐ CPU
☐ FPGA (field-programmable gate array)
☐ GPGPU (general-purpose GPU)

Steps

- Sign up for a DevCloud for oneAPI account here: https://intelsoftwaresites.secure.force.com/devcloud/oneapi

- Open-up a Jupyter lab notebook
- Common recipes on the DevCloud



Get to Know Intel oneAPI (Beta) Now

No hardware acquisitions, system configurations, or software installations.

A Fast Way to Start Coding

Are you a forward-thinking developer interested in the next generation of data-centric computing innovation?

You've come to the right place.

The Intel DevCloud is a development sandbox to learn about and program oneAPI cross-architecture applications.

Sign up now for full access to the latest Intel CPUs, GPUs, and FPGAs, Intel oneAPI Toolkits, and the new programming language, Data Parallel C++ (DPC++).

Access is free for 120 days with the possibility of an extension.

Get Access Now

Already have access? Sign in.

Required Fields(*)

First Name *

Last Name *

Email Address *

Country / Region *

- Select -

Company or University *

Which hardware and accelerator architecture are you developing for? (Select all that apply) *

What is the Intel® DevCloud?
Share
The Intel® DevCloud is a cluster compo…
oneAPI
What is the Intel® DevCloud?

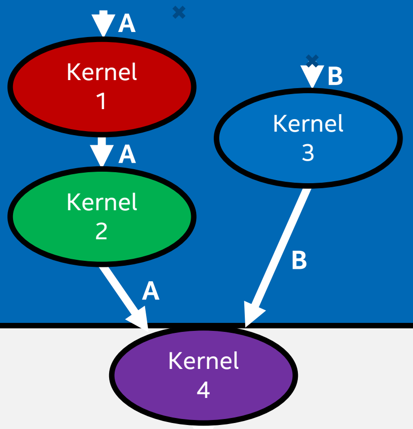
The Buffer Model

Buffers: Encapsulate data in a SYCL application

- Across both devices and host!

Accessors: Mechanism to access buffer data

- Create data dependencies in the SYCL graph that order kernel executions



Buffer

Accessor to buffer

```
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, [=](auto idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h) {
    accessor out(A, h, write_only);
    h.parallel_for(R, [=](auto idx) {
      out[idx] = idx[0]; }); });
…
```

DPC++ code anatomy

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ 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];
    });
  });
}
```

Kernel invocations are executed in parallel

Kernel is invoked for each element of the range

Kernel invocation has access to the invocation id

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 for (asynchronous) execution

Step 4: create buffer accessors to access buffer data on the device

Step 5: send a kernel (lambda) for execution

Step 6: write a kernel

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

Transition to Jupyter Notebook – Complex number multiplication

Welcome.ipynb

Select link **DPC++ Program Structure**

DPC++ = C++ + SYCL* + New Features

DPC++ New Features:

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

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

DPC++ Unified Shared Memory

Unified Shared Memory enables the sharing of memory between the host and device without explicit accessors in the source code

```
queue q;
int *data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.parallel_for(range<1>(N), [=](id<1> idx){
        data[idx[0]] += 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

DPC++ Unified Shared Memory

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

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

Automatic data accessibility and explicit data movement supported

USM – Explicit DATA TRANSFER

1. malloc_device() will allocate memory on device, Host will not have access
2. Copy memory explicitly from host to device using q.memcpy()
3. Make any data modification on device
4. Copy the memory explicitly from device to host using q.memcpy()

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

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

USM – Implicit DATA TRANSFER

1. malloc_shared() will allocate memory that can move between host and device. Host and device will have access
2. Make any data modification on device
3. Host has access to the device modified memory

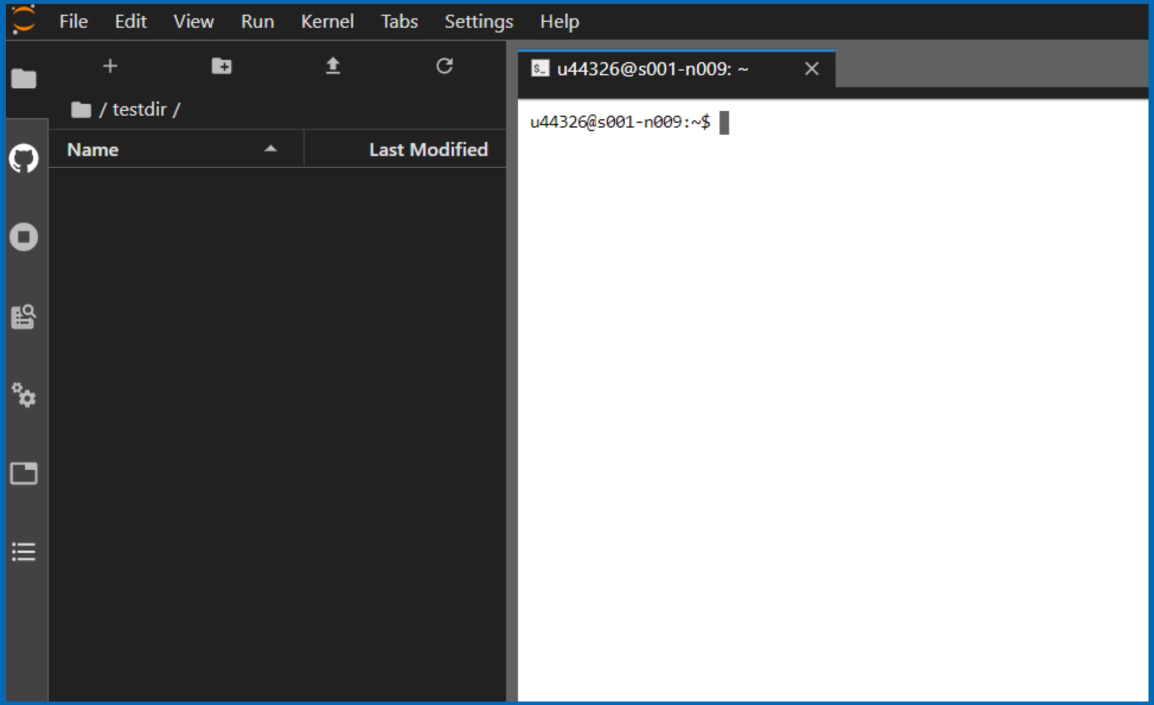
```
queue q;
int *data = malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.parallel_for(range<1>(N), [=](id<1> i){
        data[i] += 1;
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

Hough Transform Code Walk-through on Intel® DevCloud

Add Notebooks to Your Accounts

```
cp /data/oneapi_workshop/hough_transform_oneapi_notebooks-master.zip .
```

```
unzip hough_transform_oneapi_notebooks-master.zip
```



```
u44326@s001-n009:~$
```

Notices & Disclaimers

Optimization Notice