DPC++ Essentials: Advanced Concepts

oneAPI – 가속 컴퓨팅을 개발하기 위한 스마트한 방식

2022. 09. 23.

MOASYS

oneAPI Smart Development Series (2021)

1. Introduction to Intel oneAPI for HPC and AI-DL
   - https://www.allshowtv.com/detail.html?idx=474

2. Benchmarking the Performance of oneAPI on Heterogeneous Computing Platforms
   - https://www.allshowtv.com/detail.html?idx=660

3. Optimization and GPU Offloading Workflow with Intel oneAPI
   - https://www.allshowtv.com/detail.html?idx=826

4. Leveraging Intel® oneDNN for AI Workload
   - https://www.allshowtv.com/detail.html?idx=909

oneAPI Smart Development Series (2022)

1. FPGA Development Flow with Intel® oneAPI Base Toolkit
   - https://www.allshowtv.com/detail.html?idx=995

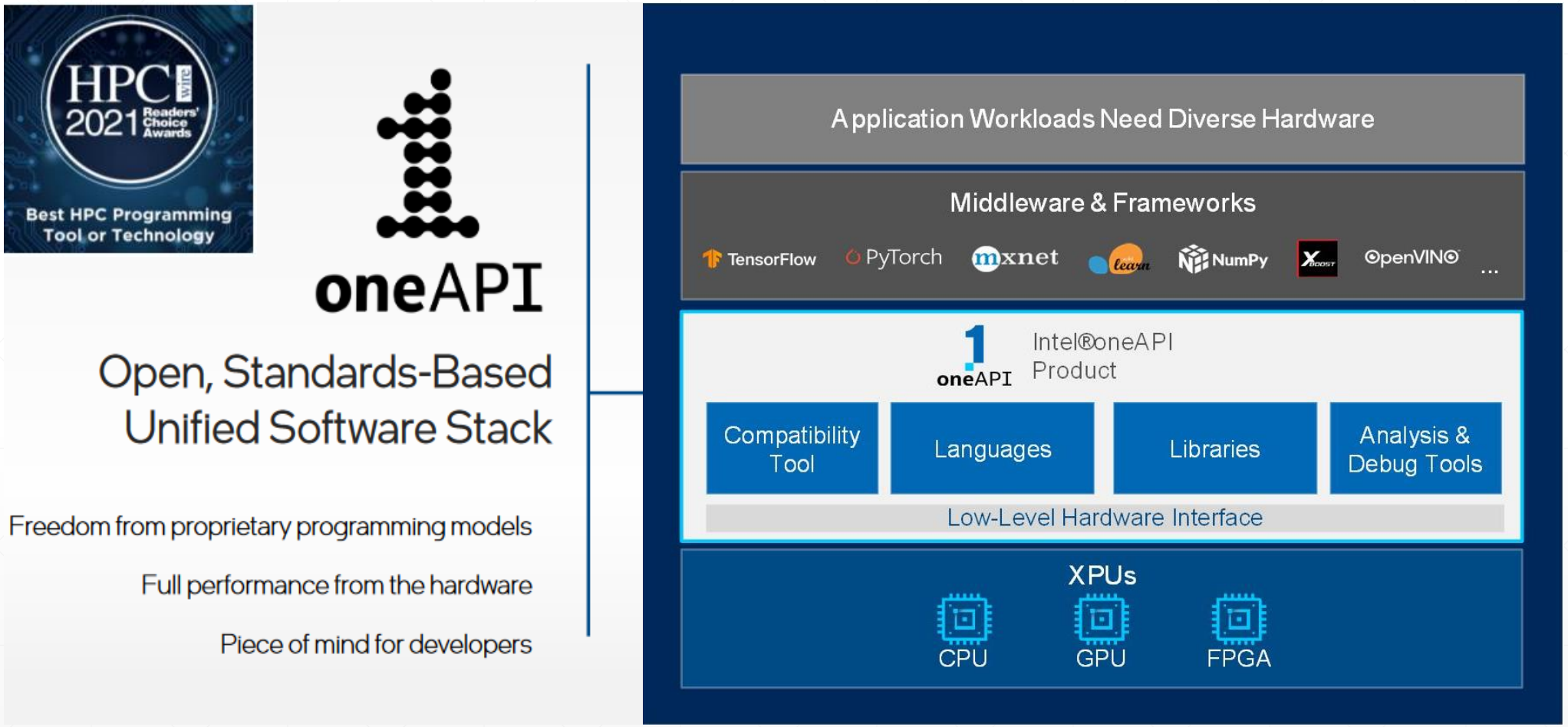
2. OpenMP Offload with Intel® oneAPI HPC Toolkit
   - https://www.allshowtv.com/detail.html?idx=1041

3. DPC++ Essentials: Advanced Concepts
   - https://www.allshowtv.com/detail.html?idx=1112

4. DPC++ Essentials: GPU Optimizations Techniques
   - TBA

Contents

Overview of oneAPI for Heterogenous Computing



- Support diverse accelerator devices (XPU) such as CPU, GPU and FPGA
- Continuously evolving specifications for high performance computing and machine learning

Comparison of Heterogenous Programming Models

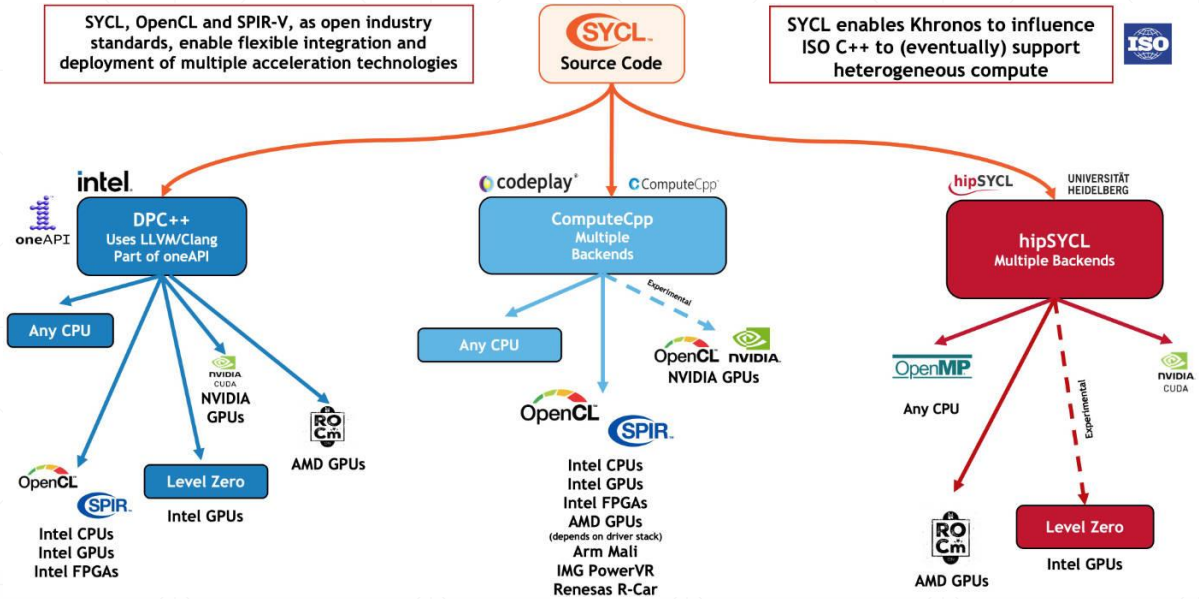
	CUDA	HIP	OpenACC	OpenMP	SYCL/DPC++
Languages	C/C++/Fortran	C/C++/Fortran	C/C++/Fortran	C/C++/Fortran	C++
Abstraction	Low	Low	High	High	Medium
Coding	-	-	Directive-based	Directive-based	C++ lambda
Parallelism	SIMT	SIMT	Fork-join SIMD	Fork-join SIMD	OpenCL
Offload	GPU (NVIDIA)	GPU (NVIDIA/AMD)	GPU (NVIDIA)	CPU/GPU (NVIDIA/AMD/Intel)	CPU/GPU/FPGA (NVIDIA/AMD/Intel)
Compiler	Proprietary	LLVM	PGI/CCE/GCC	PGI/CCE/GCC/LLVM/XL/Intel	LLVM
License	Proprietary	Open-source	Open-source	Open-source	Open-source

- Write once, run everywhere with SYCL/DPC++
  - ISO C++17 and SYCL standards and extensions
  - Single-source style framework
  - Asynchronous execution model
  - Open, cross-architecture and cross-vendor

Top500: June 2022

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1 AMD GPU	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2 ARM	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3 AMD GPU	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942

https://www.top500.org/lists/top500/2022/06/



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

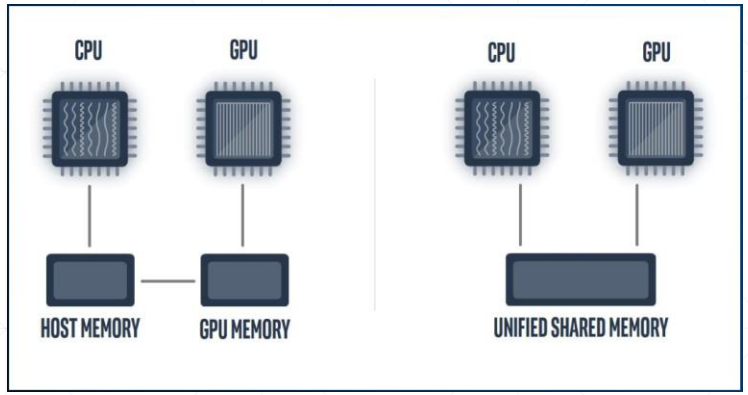
- Intel DPC++ Compiler: oneAPI Base Toolkit
  - OpenCL backend: optimized for Intel CPUs, GPUs (Gen9, 11, Xe) and FPGA (Stratix, Aria)
  - Level Zero backend: low-level offloading API currently supporting only Intel GPUs
- Intel LLVM Compiler: open source project
  - CUDA backend: experimental support for NVIDIA GPUs
  - HIP backend: experimental support for AMD GPUs via ROCm 4.x

SYCL 2020: Basic Concepts

```
#include <iostream>
#include <vector>
#include <sycl/sycl.hpp>
#define N 1000

using namespace sycl;

int main() {
    std::vector<int> v(N)

    queue q{default_selector()};

    { // scope opens
        buffer buf{v};

        q.submit([&](handler& h) {
            accessor av{buf, h, read_write};

            h.parallel_for(N, [=](auto i) {
                av[i] = i;
            });
        });
    } // scope closes

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

    return 0;
}
```

- sycl::queue:
  - Offload code submitted to device via queue
  - One queue maps to exactly one device to avoid runtime ambiguity
  - Multiple queues can use same device
- sycl::buffer:
  - Initialized from already allocated memory
  - SYCL 1.2.1: **`buffer<int,1> buf{v.data(),range<1>{N}};`**
  - SYCL 2020: simplified, and less verbose thanks to CTAD (C++17)
  - Buffer destruction via scope closing is a blocking call
- sycl::accessor:
  - How memory is accessed: *read_only, write_only, read_write*
  - Where memory is accessed: *global_memory, local_memory*
  - SYCL 1.2.1 style: **`buf.get_access<access::mode::read_write>(h)`**
- sycl::handler:
  - Constructed at runtime, as argument to lambda function of *submit*
  - How code is submitted to queue: *single_task (serial), paralle_for (SIMT)*
  - Kernel code as callable lambda functions
  - No dynamic memory allocation

SYCL 2020: Unified Shared Memory (USM)



Allocation	Host accessible ?	Device accessible ?	Memory Space
malloc_host	Yes	Yes	Host
malloc_device	No	Yes	Device
malloc_shared	Yes	Yes	UVA

- A pointer-based approach to memory allocation
  - Simplify porting to accelerators with minimal changes
- sycl::malloc::host:
  - Return a pointer to memory physically located on the host
  - Device can access host allocated memory via hardware interface, *e.g.* PCIe buses
- sycl::malloc::device:
  - Return a pointer to memory physically located on the device
- sycl::malloc::shared:
  - Return a pointer to the *unified virtual address* (UVA) space
  - SYCL runtimes automatically handle data movements between host/device, but incurring additional latency

USM: API

C allocation

```
void* malloc_device(size_t numBytes,
                    const queue& syclQueue,
                    const property_list& prop_list = {});

void* malloc_host(size_t numBytes,
                  const queue& syclQueue,
                  const property_list& prop_list = {});

void* malloc_shared(size_t numBytes,
                    const queue& syclQueue,
                    const property_list& prop_list = {});
```

C++ allocation

```
template <typename T>
T* malloc_device(size_t count,
                 const queue& syclQueue,
                 const property_list& prop_list = {});

template <typename T>
T* malloc_host(size_t count,
               const queue& syclQueue,
               const property_list& prop_list = {});

template <typename T>
T* malloc_shared(size_t count,
                 const queue& syclQueue,
                 const property_list& prop_list = {});
```

USM: Explicit Data Movement

Initialization on host →

Default queue creation →

*USM allocation on device via **malloc::device()*** →

*Memory copy from host to device via **memcpy()*** →

Data modification on device →

*Memory copy from device back to host via **memcpy()*** →

*Free memory with **sycl::free()** to avoid memory leak* →

```
#include <iostream>
#include <sycl/sycl.hpp>
#include <vector>

#define N 1000

using namespace sycl;

int main() {
    int v[N];
    for (int i = 0; i < N; i++) v[i] = i;

    queue myQueue;

    auto *v_device = malloc_device<int>(N, q)

    q.memcpy(v_device, v, sizeof(int)*N).wait()

    q.parallel_for(N, [=](auto i) {
            v_device[i] = i;
    });

    q.memcpy(v, v_device, sizeof(int)*N).wait()

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

    free(v_device, q)

    return 0;
}
```

USM: Implicit Data Movement

Default queue creation →

*USM memory allocation via **malloc_shared()*** →

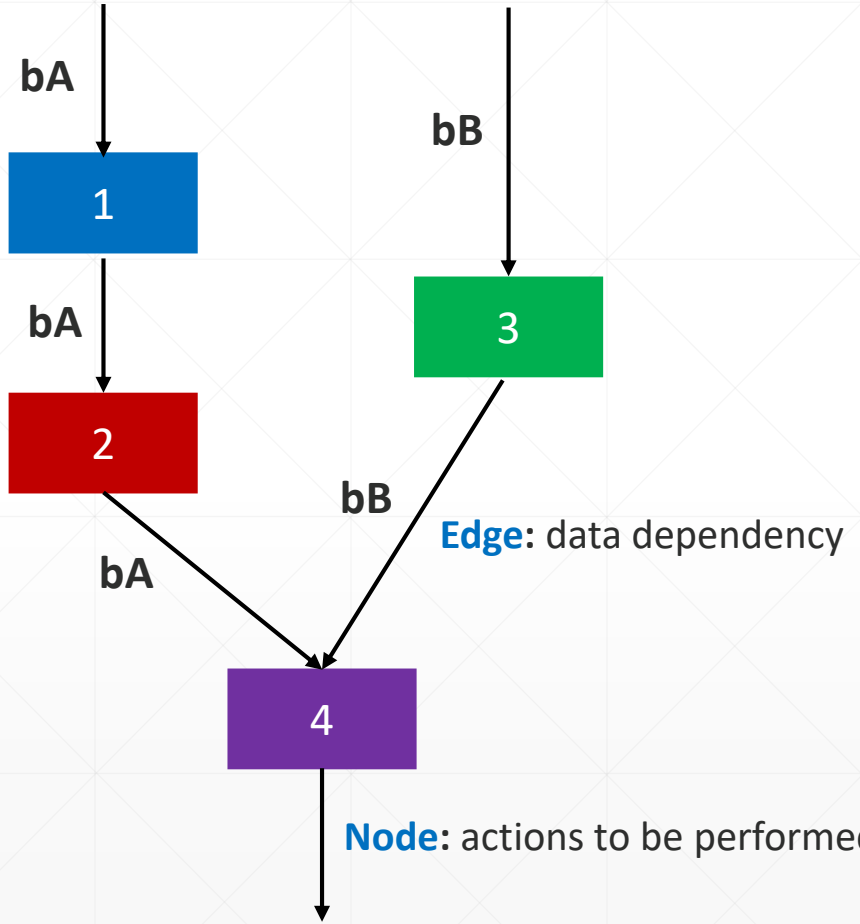
Data modification on device →

*Free memory with **sycl::free()** to avoid memory leak* →

```
#include <iostream>
#include <sycl/sycl.hpp>
#include <vector>

#define N 1000

using namespace sycl;

int main() {
    queue q;

    auto *v_shared = malloc_shared<int>(N, q);

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

    q.parallel_for(N, [=](auto i) {
        v_shared[i] = i;
    });

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

    free(v_shared, q)

    return 0;
}
```

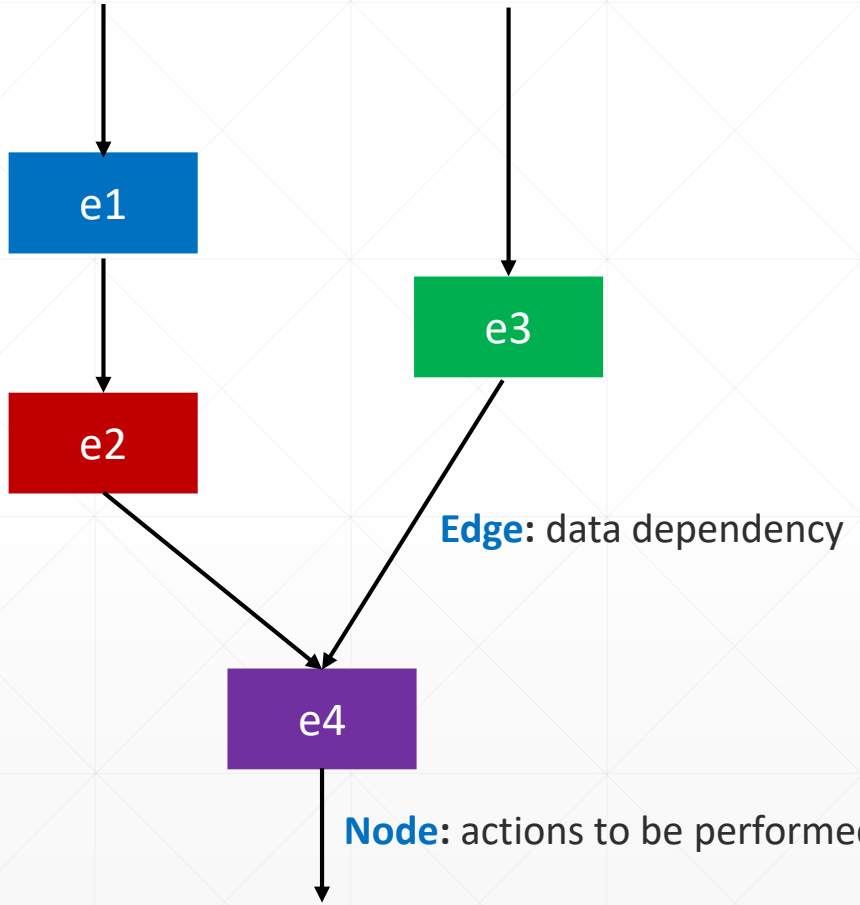
- Shared allocation is convenient but not intended for providing peak performance out of the box.

SYCL Task Graph: Buffer




```
int main() {
    queue q;

    std::vector<int> A(N), B(N);

    buffer buf_A{A}, buf_B{B};

    q.submit([&](handler& h) {
        accessor out{buf_A, h, write_only};
        h.parallel_for(N, [=](auto i) { out[i] = i; });
    });

    q.submit([&](handler& h) {
        accessor out{buf_A, h, write_only};
        h.parallel_for(N, [=](auto i) { out[i] = 2*i; });
    });

    q.submit([&](handler& h) {
        accessor out{buf_B, h, write_only};
        h.parallel_for(N, [=](auto i) { out[i] = i; });
    });

    q.submit([&](handler& h) {
        accessor out{buf_A, h, read_only};
        accessor inout{buf_B, h, read_write};
        h.parallel_for(N, [=](auto i) { inout[i] *= in[i]; });
    });

    return 0;
}
```

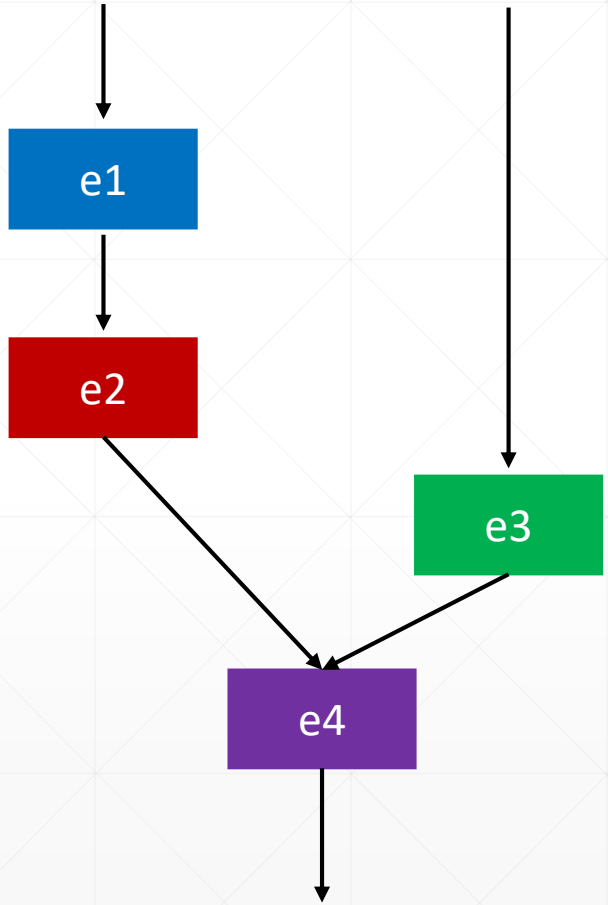
SYCL Task Graph: USM with depend_on()




```
int main() {
    queue q;

    std::vector<int> A(N), B(N);
    int *usm_a = malloc_shared<int>(N, q)
    int *usm_b = malloc_shared<int>(N, q)

    auto e1 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_a[i] = i; });
    });

    auto e2 = q.submit([&](handler& h) {
        h.depends_on(e1);
        h.parallel_for(N, [=](auto i) { usm_a[i] = 2*i; });
    });

    auto e3 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_b[i] = i; });
    });

    auto e4 = q.submit([&](handler& h) {
        h.depends_on(e2, e3);
        h.parallel_for(N, [=](auto i) { usm_b[i] *= usm_a[i]; });
    });

    return 0;
}
```

- depends_on() allows some computation tasks to overlap when there is no dependency

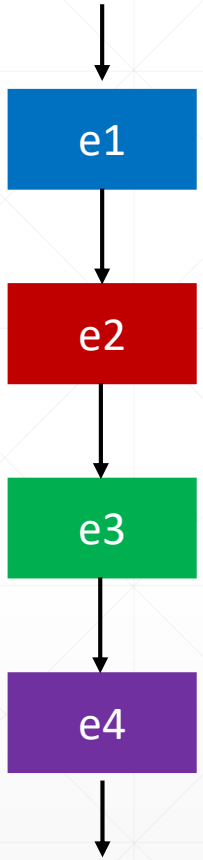
SYCL Task Graph: USM with wait()




```
int main() {
    queue q;

    std::vector<int> A(N), B(N);
    int *usm_a = malloc_shared<int>(N, q)
    int *usm_b = malloc_shared<int>(N, q)

    auto e1 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_a[i] = i; });
    });

    auto e2 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_a[i] = 2*i; });
    }).wait();

    auto e3 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_b[i] = i; });
    }).wait();

    auto e4 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_b[i] *= usm_a[i]; });
    });

    return 0;
}
```

- wait() also block execution on host, making it less ideal for large-scale application

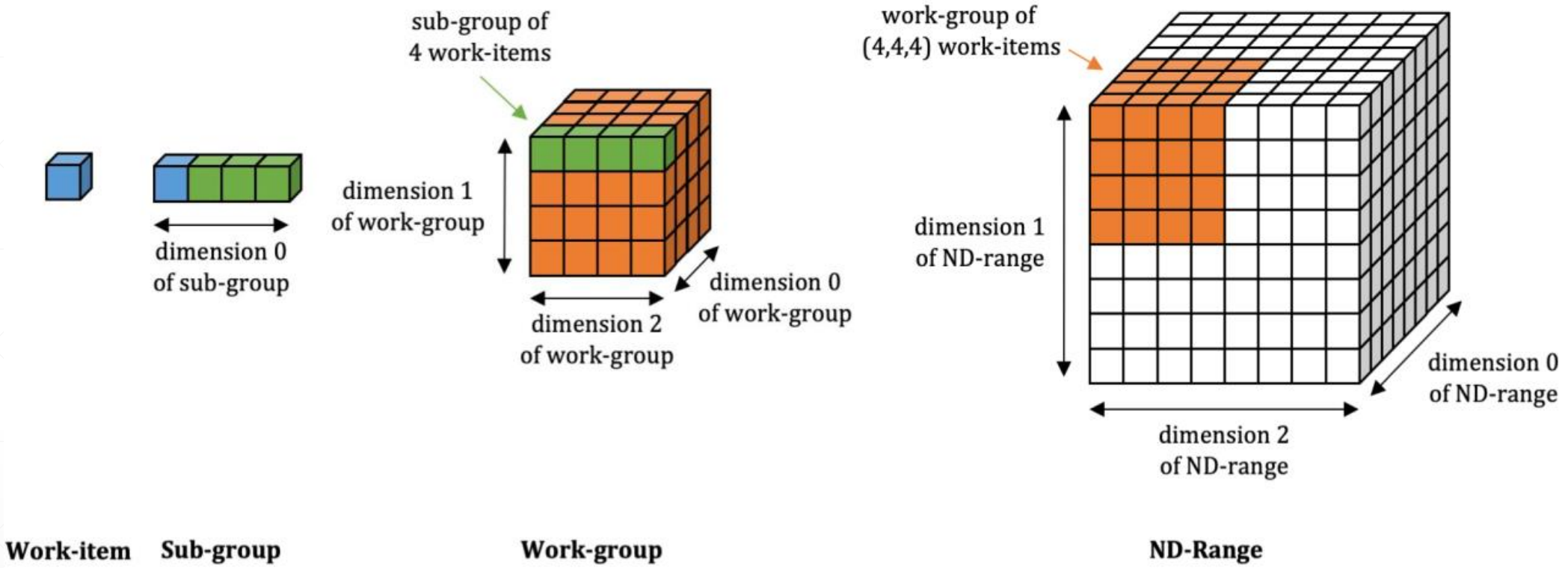
SYCL Task Graph: USM with In-order Queue




```
int main() {
    queue q{properties::queue::in_order()};

    std::vector<int> A(N), B(N);
    int *usm_a = malloc_shared<int>(N, q)
    int *usm_b = malloc_shared<int>(N, q)

    auto e1 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_a[i] = i; });
    });

    auto e2 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_a[i] = 2*i; });
    });

    auto e3 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_b[i] = i; });
    });

    auto e4 = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto i) { usm_b[i] *= usm_a[i]; });
    });

    return 0;
}
```

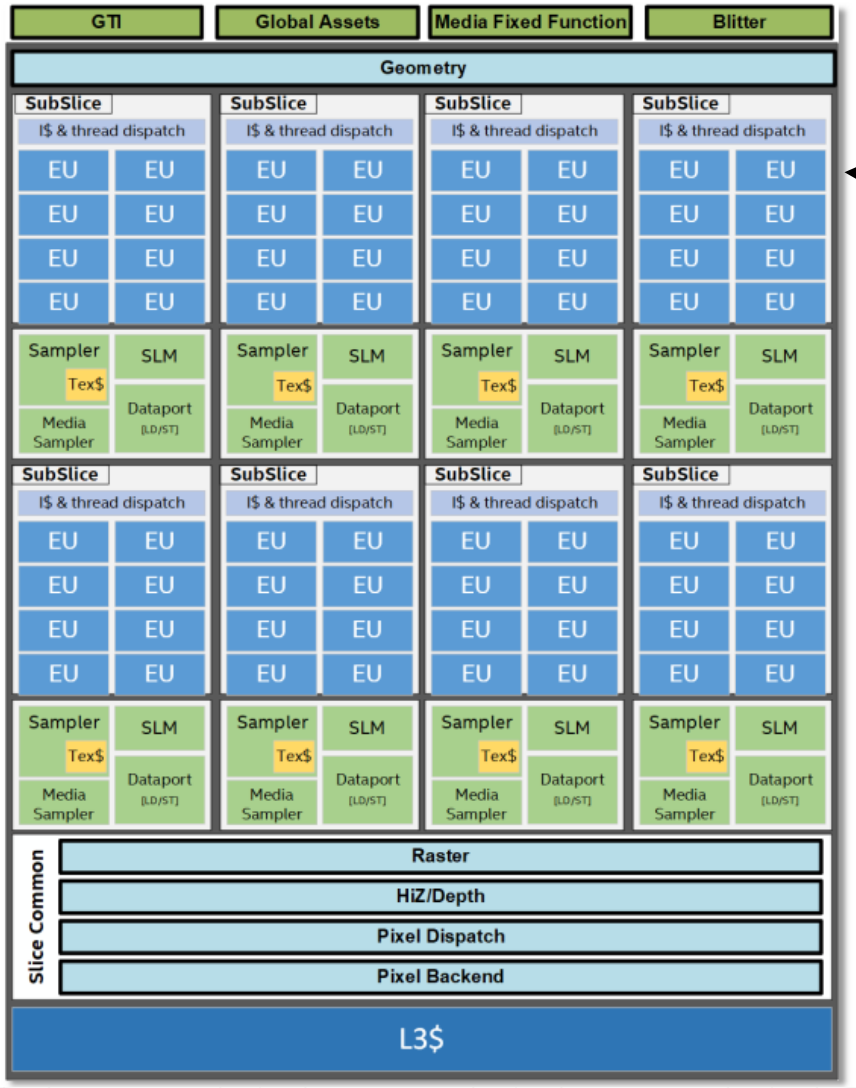
- SYCL/DPC++ offers two kinds of queue:
  - In-order queue: kernels are executed in the order they were submitted to the queue
  - Out-of-order queue: kernels can be executed in an arbitrary order determined by SYCL runtime (default)

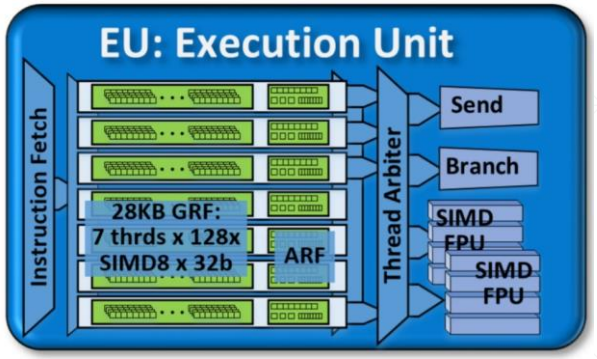
SYCL Hierarchical Parallelism



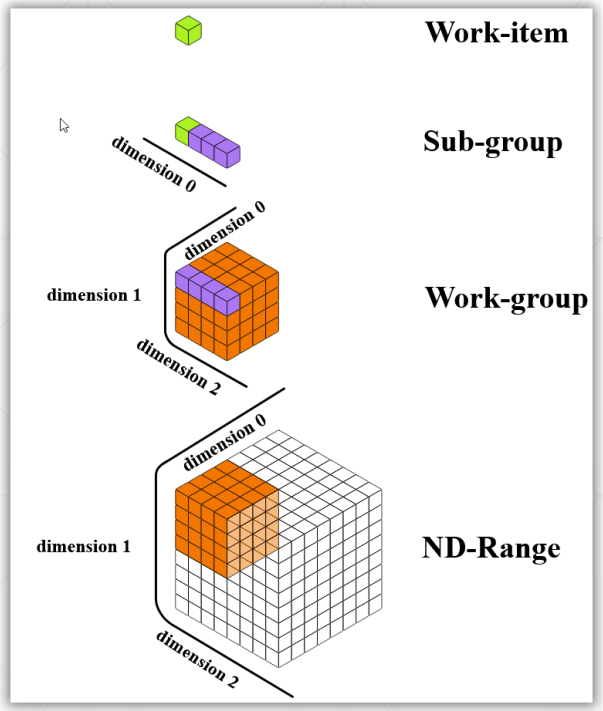
- Sub-group:
  - A subset of work-items within a work-group that executed simultaneously
  - Often mapped to SIMD lane/channel
  - Direct communication between work-item without access to local and global memories
  - Access to sub-group collectives providing faster implementation of common parallel patterns

Mapping of Sub Group to Hardware on Intel Gen 11





- 1 x slice
- 8 x subslices
- 8 x EUs per subslice
- 7 x threads per EU
- 128 x SIMD8 lanes per thread



SIMD8 lane

EU's thread

Executing Unit (EU)

Sub Slice

Sub-group Shuffle: Swap Adjacent Items

Truth table

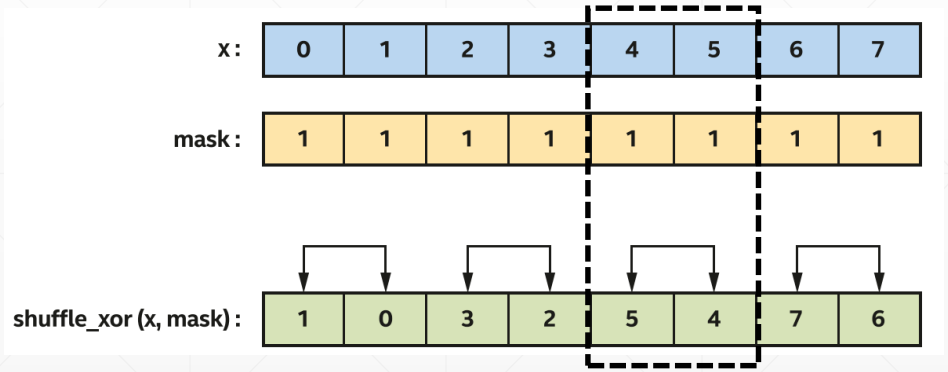
x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

(4) 100
(1) 001
(5) 101

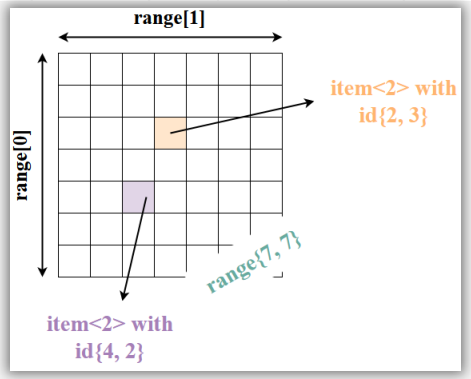
4 → 5

(5) 101
(1) 001
(4) 100

5 → 4




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

using namespace sycl;

static const size_t N = 256;
static const size_t B = 64;

int main() {
    queue q;

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

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

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

        data[i] = sg.shuffle_xor(data[i], 1);
    }).wait();

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

    free(data, q);

    return 0;
}
```



- Sub group handler can be obtained with *get_sub_group()*

Sub-group Shuffle: Reverse Order of Items

Truth table

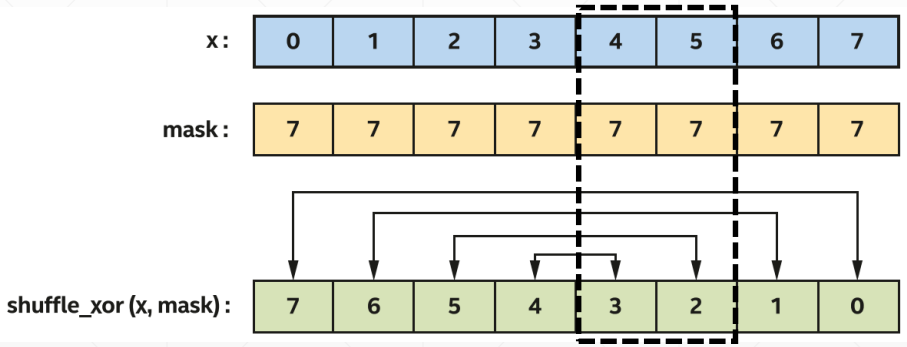
x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

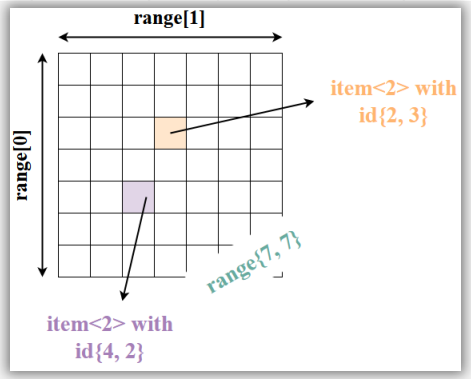
(4) 100
(7) 111
(3) 011

4 → 3

(5) 101
(7) 111
(2) 010

5 → 2






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

using namespace sycl;

static const size_t N = 256;
static const size_t B = 64;

int main() {
    queue q;

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

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

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

        data[i] = sg.shuffle_xor(data[i], sg.get_max_local_range()-1);
    }).wait();

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

    free(data, q);

    return 0;
}
```

- Sub group handler can be obtained with *get_sub_group()*

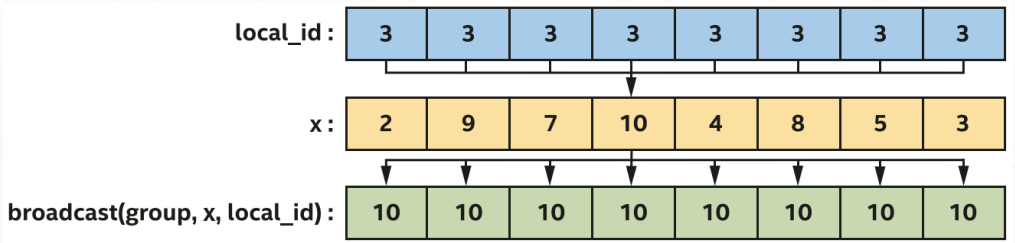
Sub Group Collectives: Broadcast

Default queue creation →

*USM allocation via **malloc::shared*** →

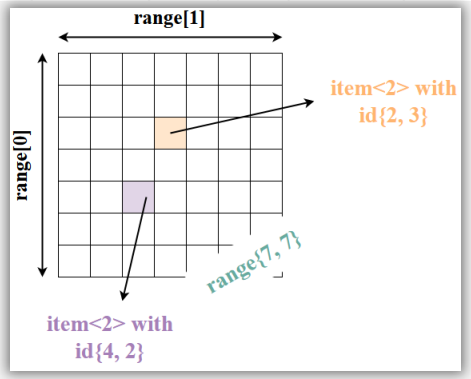
Data initialization on host →

*Sub-group handle obtained from **get_sub_group()*** →



→

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

using namespace sycl;

static const size_t N = 256;
static const size_t B = 64;

int main() {
    queue q;

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

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

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

        data[i] = group_broadcast(sg, data[i], 3);
    }).wait();

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

    free(data, q);

    return 0;
}
```



- *broadcast()* takes a value from one work item and communicate it to others in same work group

Sub Group Reduction

Default queue creation →

*USM allocation via **malloc::shared*** →

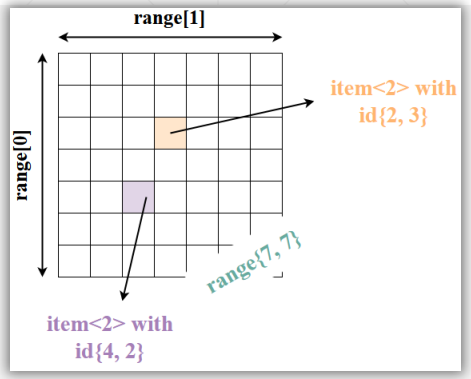
Data initialization on host →

*Sub-group handle obtained from **get_sub_group()*** →

Adds all elements in using sub_group collectives →

Write sum to first location for each sub_group →

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

using namespace sycl;

static const size_t N = 256;
static const size_t B = 64;

int main() {
    queue q;

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

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

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

        int sum = reduce(sg, data[i], plus<>());

        if (sg.get_local_id()[0] == 0) data[i] = sum;
    }).wait();

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

    free(data, q);

    return 0;
}
```



Specifying Sub Group Size

Default queue creation →

USM allocation via ***malloc::shared*** →

Data initialization on host →

Sub group size configuration →

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

using namespace sycl;

static const size_t N = 256;
static const size_t B = 64;
static const sizt_t S = 16;

int main() {
    queue q;

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

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

    q.parallel_for(nd_range<1>{N, B}, [=](nd_item<1> item)[[intel::reqd_sub_group_size(S)]] {
        sub_group sg = item.get_sub_group();
        auto i = item.get_global_id(0);

        data[i] = broadcast(sg, data[i], 3);
    }).wait();

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

    free(data, q);

    return 0;
}
```

Sub Group Size of Intel Gen 9 (DevCloud)

Platform Name	**Intel(R) OpenCL HD Graphics**
Number of devices	1
Device Name	Intel(R) UHD Graphics [0x9a60]
Device Vendor	Intel(R) Corporation
Device Vendor ID	0x8086
Device Version	OpenCL 3.0 NEO
Driver Version	22.23.23405
Device OpenCL C Version	OpenCL C 1.2
Device Type	GPU
Device Profile	FULL_PROFILE
Device Available	Yes
Compiler Available	Yes
Linker Available	Yes
Max compute units	32
Max clock frequency	1450MHz
Device Partition	(core)
Max number of sub-devices	0
Supported partition types	None
Supported affinity domains	(n/a)
Max work item dimensions	**3**
Max work item sizes	**512x512x512**
Max work group size	**512**
Preferred work group size multiple	**64**
Max sub-groups per work group	**64**
Sub-group sizes (Intel)	**8, 16, 32**

- The set of available sub group size is hardware dependency

SYCL 2020 Reduction: Summation of All Work Items

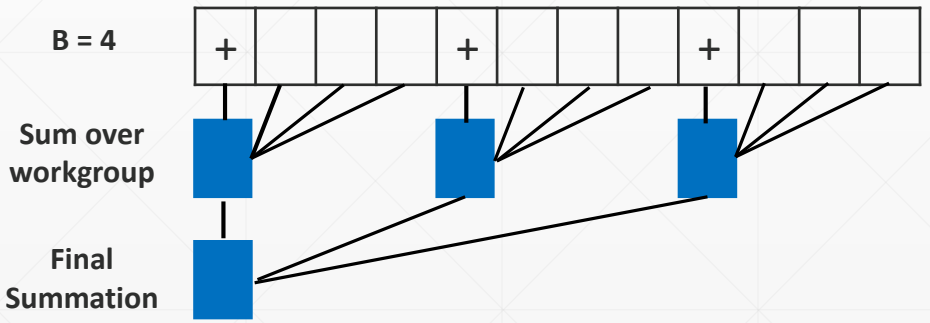
Default queue creation →

*USM allocation via **malloc::shared*** →

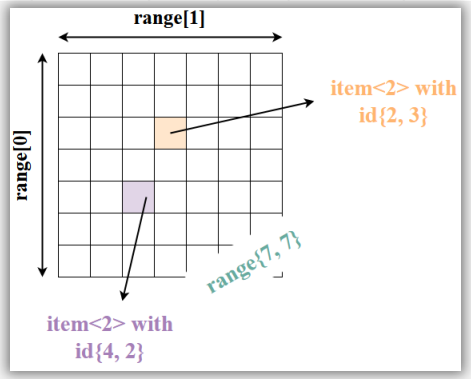
Calculate sum of all items for each work group →

Loop over work group items using global index →

Final reduction of workgroup sums →




```
#include <iostream>
#include <CL/sycl.hpp>
using namespace sycl;
static const size_t N = 1024;
static const size_t B = 64;

int main() {
    queue q;

    int *data = malloc_shared<int>(N, q);
    for (int i = 0; i < N; i++) data[i] = i;

    q.parallel_for(nd_range<1>(N, B), [=](nd_item<1> item){
        auto index = item.get_global_id(0);
        if(item.get_local_id(0) == 0 ){
            int sum_wg = 0;
            for(int i=index; i<index+B; i++) sum_wg += data[i];
            data[index] = sum_wg;
        }
    }).wait();

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

    free(data, q);

    return 0;
}
```



SYCL 2020 Reduction: Using Work Group Reduction

Default queue creation →

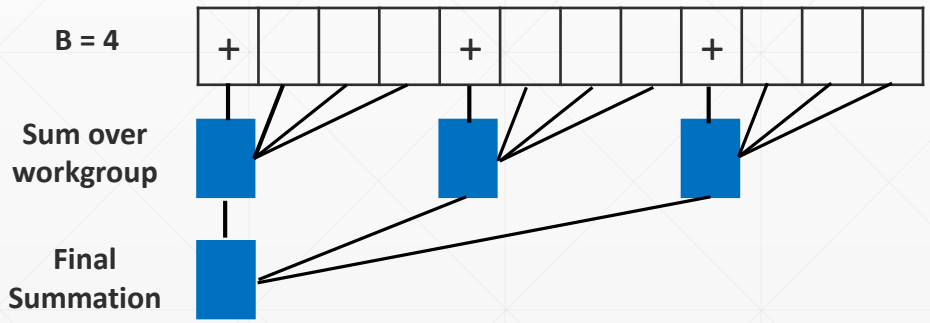
USM allocation via ***malloc::shared*** →

Calculate sum of items in work groups →

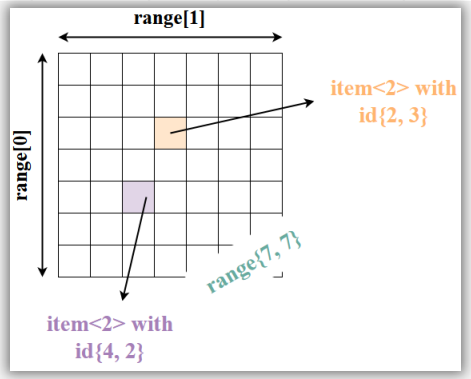
Use work group ***reduce()*** →

Write work group sum to first location →

Final reduction of work group sums →




```
#include <iostream>
#include <CL/sycl.hpp>
using namespace sycl;
static const size_t N = 1024;
static const size_t B = 64;

int main() {
    queue q;

    int *data = malloc_shared<int>(N, q);
    for (int i = 0; i < N; i++) data[i] = i;

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

        int sum_wg = reduce(wg, data[i], plus<>());

        if (item.get_local_id()[0] == 0) data[index] = sum_wg;

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

    free(data, q);

    return 0;
}
```



SYCL 2020 Reduction: Using Sub Group Reduction

Default queue creation →

USM allocation via ***malloc::shared*** →

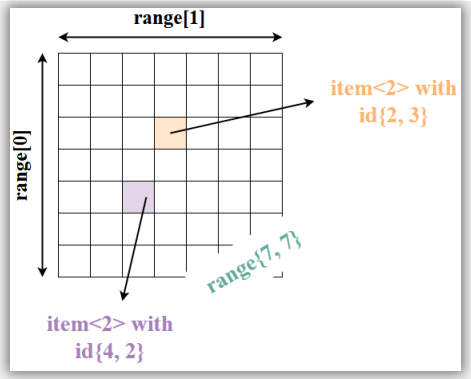
Calculate sum of items in sub groups →

Use sub group ***reduce()*** →

Write sub group sum to first location →

Final reduction of subgroup sums →

```
#include <iostream>
#include <CL/sycl.hpp>
using namespace sycl;
static const size_t N = 1024;
static const size_t B = 64;
static const size_t S = 16;
int main() {
    queue q;

    int *data = malloc_shared<int>(N, q);
    for (int i = 0; i < N; i++) data[i] = i;

    q.parallel_for(nd_range<1>(N, B), [=](nd_item<1> item)[[intel::reqd_sub_group_size(S)]]{
        auto sg    = item.get_sub_group();
        auto index = item.get_global_id(0);

        int sum_sg = reduce(sg, data[i], plus<>());

        if (sg.get_local_id()[0] == 0) data[index] = sum_sg;

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

    free(data, q);

    return 0;
}
```



SYCL 2020 Reduction: Simplified Reduction

Default queue creation →

USM allocation via ***malloc::shared*** →

USM allocation of sum variable →

reduction() returns a reducer object →
Pass reference of reducer as parameter of the lambda

+= corresponds to plus<> →

```
#include <iostream>
#include <CL/sycl.hpp>
using namespace sycl;
static const size_t N = 1024;
static const size_t B = 64;

int main() {
    queue q;

    int *data = malloc_shared<int>(N, q);
    for (int i = 0; i < N; i++) data[i] = i;

    int *usum = malloc_shared<int>(1, q);
    *usum = 0;

    q.parallel_for(nd_range<1>{N, B}, reduction{usum, plus<>()},
                   [=](nd_item<1> item, auto& sum){
        auto index = item.get_global_id(0);

        sum += data[index];
     }).wait();

    free(data, q);
    free(usum, q);

    return 0;
}
```

- Reduction object within ***parallel_for*** contruct simplifies reduction semantics within SYCL kerne

SYCL 2020 Reduction: Multiple Reduction

Default queue creation →

*USM allocation via **malloc::shared*** →

USM allocation of sum variable →

USM allocation of max variable →

max<> has no shorthand notation →
*Updated via **combine()***

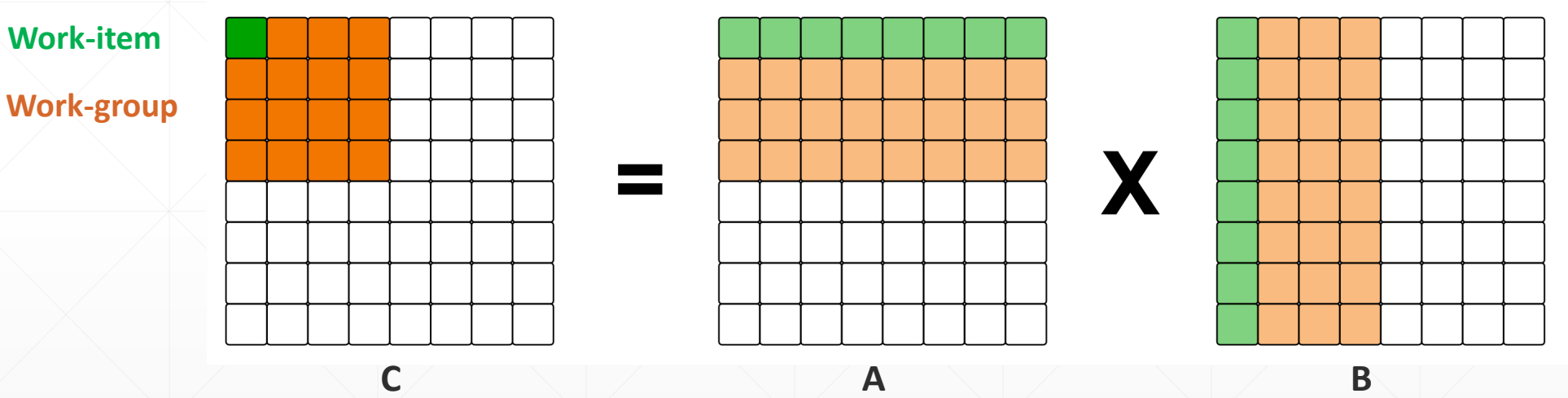
```
#include <iostream>
#include <CL/sycl.hpp>
using namespace sycl;
static const size_t N = 1024;
static const size_t B = 64;

int main() {
    queue q;

    int *data = malloc_shared<int>(N, q);
    for (int i = 0; i < N; i++) data[i] = i;

    int *usum = malloc_shared<int>(1, q);
    *usum = 0;
    int *umax = malloc_shared<int>(1, q);
    *umax = 0;

    q.parallel_for(nd_range<1>{N, B},
                   reduction{usum, plus<>()}, reduction{umax, max<>()},
                   [=](nd_item<1> item, auto& sum, auto& max){
        auto index = item.get_global_id(0);

        sum += data[index];
        max.combine(data[index]);
    }).wait();

    free(data, q);
    free(usum, q);
    free(umax, q);
    return 0;
}
```

Conclusion

- Unified Shared Memory (USM):
  - Pointer-based approach to data management
  - Implicit vs. explicit data movement
  - Handling data dependencies with event wait(), depend_on() and in-order queue
- Sub group:
  - Direct communication between work-items to avoid repeated access to global and local memories
  - Fast implementation of common parallel pattern by using sub group collectives
- Reduction:
  - Reduction object to simplify parallel reduction
- Contact: sales@moasys.com
  - GPU and FPGA migration
  - Code optimization and parallelization consultant
  - Specialized HPC education

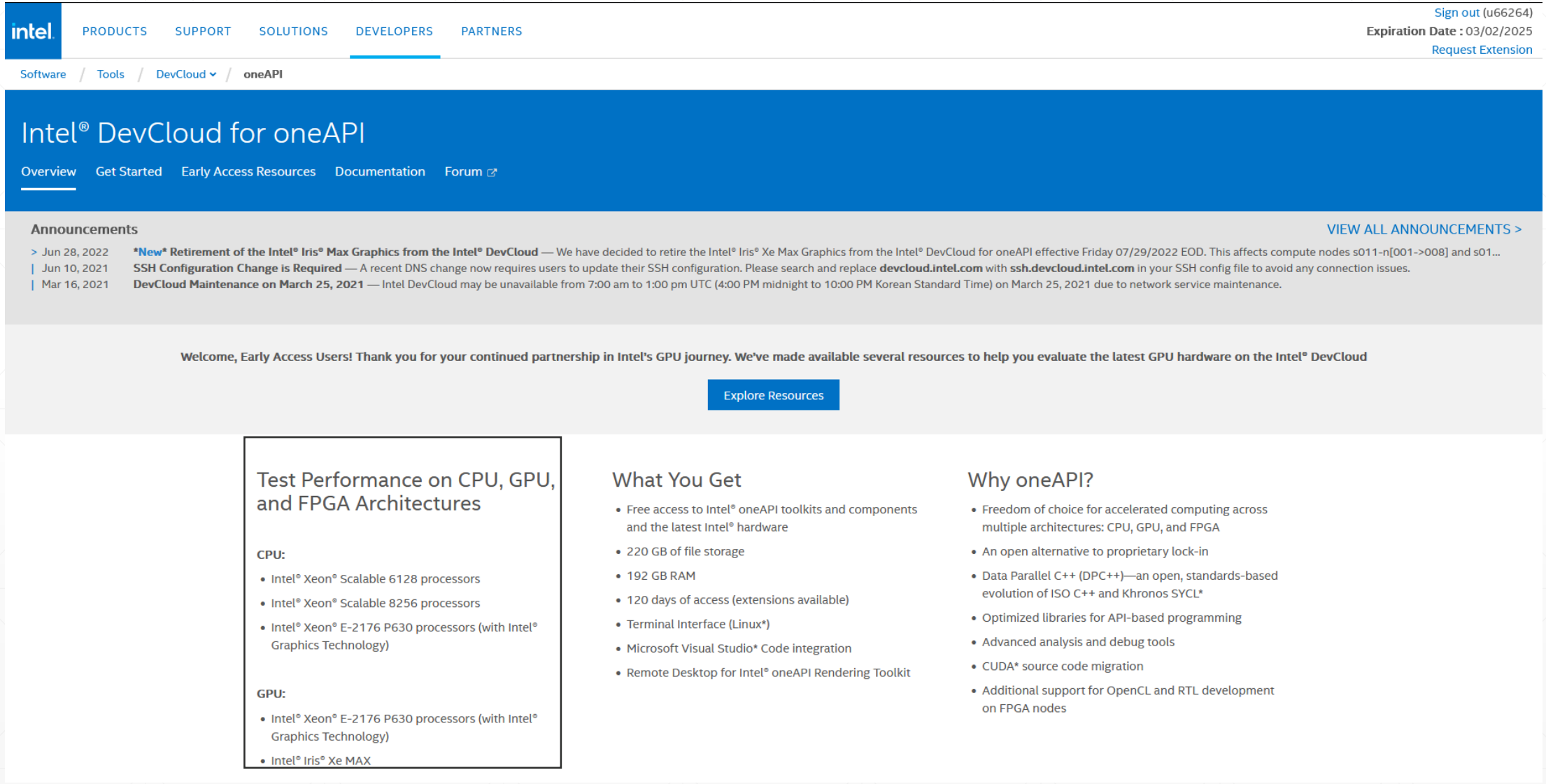
Hand on: Matrix Multiplication on DevCloud

- Basic usage of Intel DevCloud
- Device discovery and query
- Implementation of matrix multiplication using ND-range kernel: buffer, USM

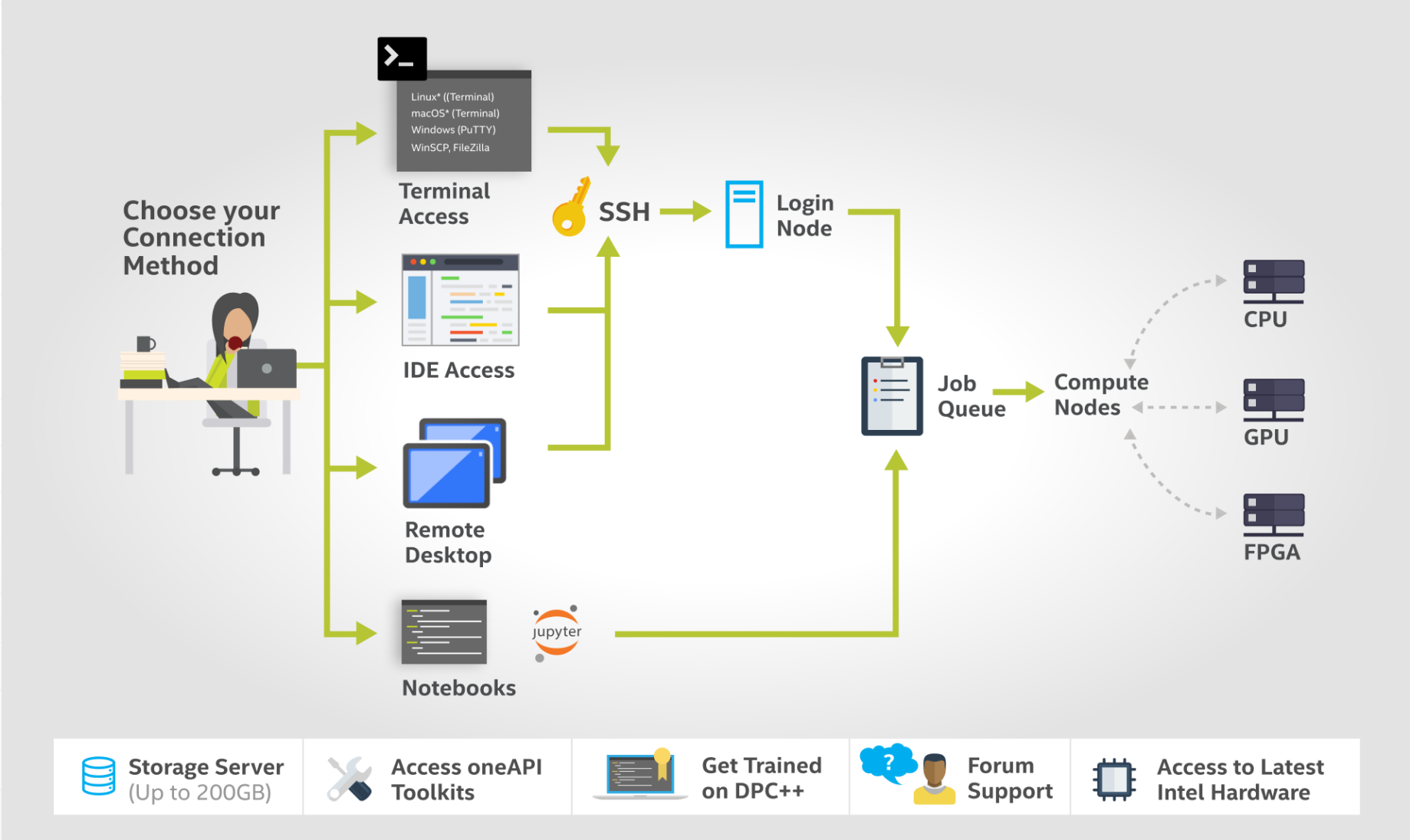


- Optimization #1: Usage of work group local memory
- Optimization #2: Usage of sub group BCAST

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



Connection Methods



https://devcloud.intel.com/oneapi/get_started/

OpenCL for FPGA development

Intel® FPGA SDK for OpenCL™ software technology1 is a development environment that enables software developers to accelerate their applications by targeting heterogeneous platforms with Intel CPUs and FPGAs.

Get Started with your first Sample

- Microsoft* Visual Studio or Eclipse*-based Intel® Code Builder for OpenCL™ API now with FPGA support
- Fast FPGA emulation based on Intel's compiler technology
- Create OpenCL™ project jump-start wizard
- Development Environment for both host (CPU) and accelerator (FPGA)
- Syntax highlighting and code auto-completion features
- FPGA resource and performance analysis
- Fast and incremental FPGA compile

RTL Acceleration Functional Unit

The revolutionary Intel® Quartus® Prime Design Software includes everything you need to design for Intel® FPGAs, SoCs, and complex programmable logic device (CPLD) from design entry and synthesis to optimization, verification, and simulation. Dramatically increased capabilities on devices with multi-million logic elements are providing designers with the ideal platform to meet next-generation design opportunities.

Build and design using standard logic gates. Great for visualization and education.

Get Started with your first Sample

Connect with JupyterLab*

Connect with JupyterLab*

Use JupyterLab* 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.

Launch JupyterLab*

Training Resources

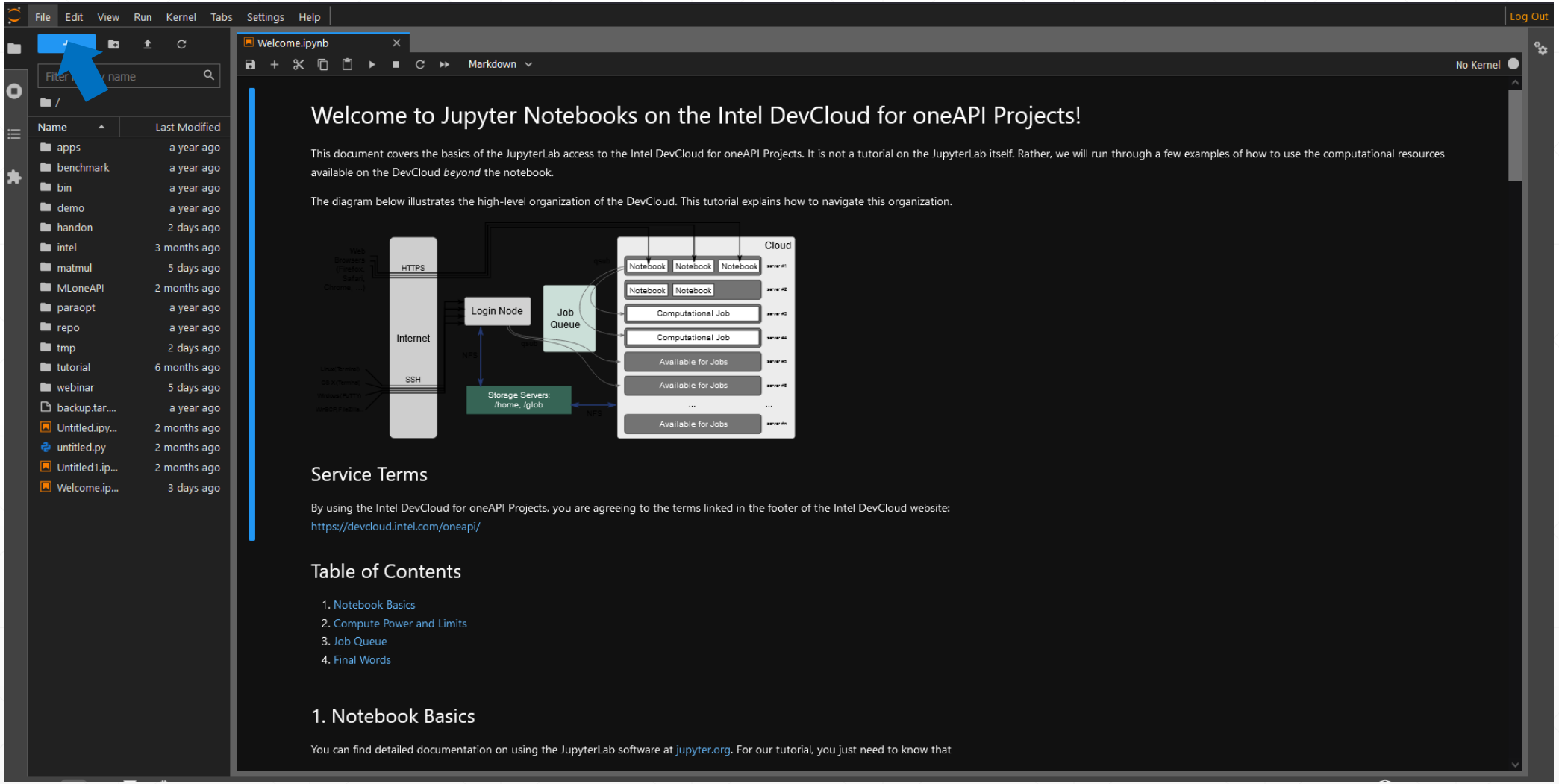
DevCloud Commands

Learn about the features of the compute nodes, data management, and how to submit, query, and delete your jobs.

Introduction to oneAPI and Essentials of Data Parallel C++

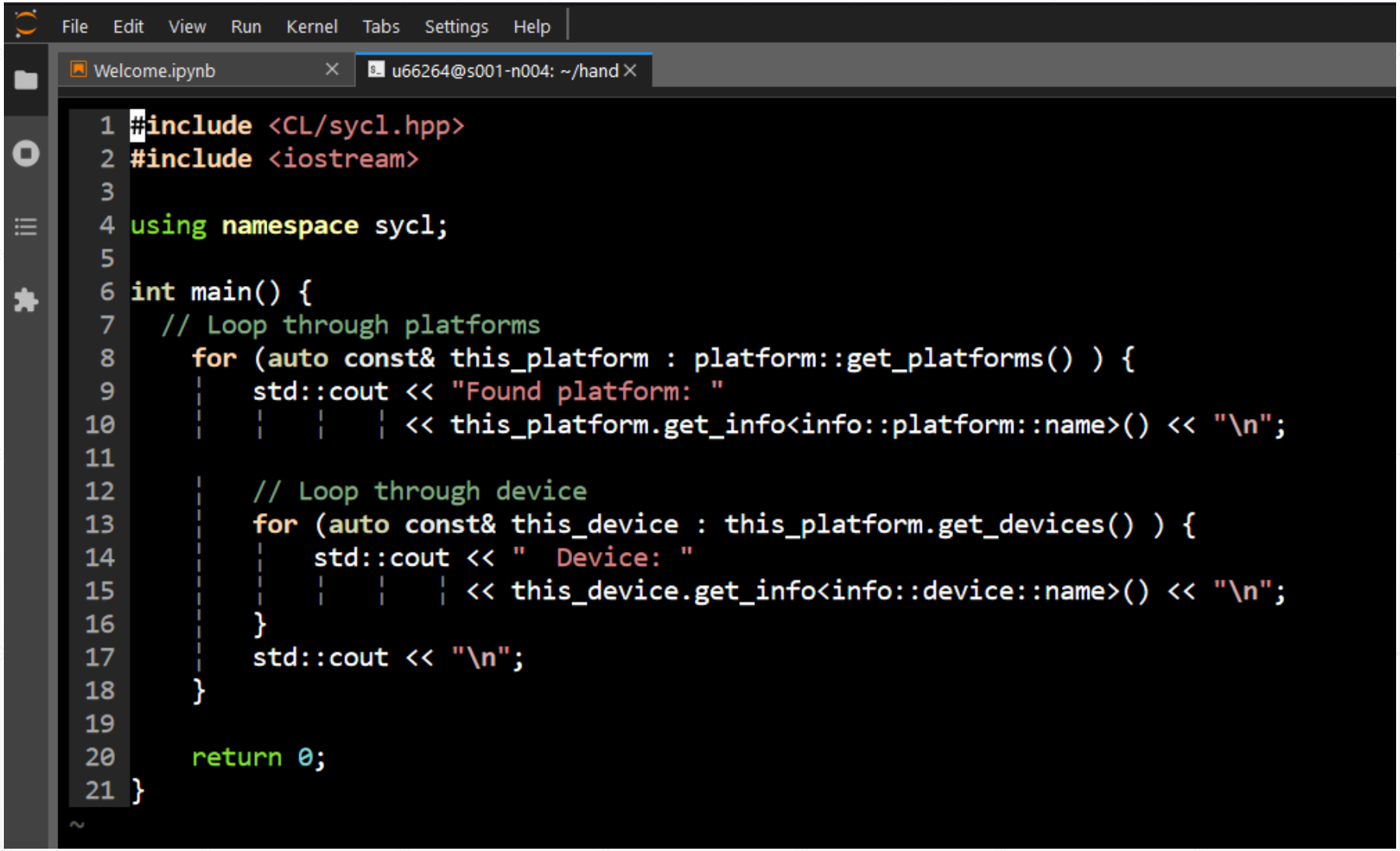
Use JupyterLab* to learn about how oneAPI can solve the challenges of programming in a heterogeneous world and understand the Data Parallel C++ (DPC++) language and programming model.

Jupyter Hub Interface

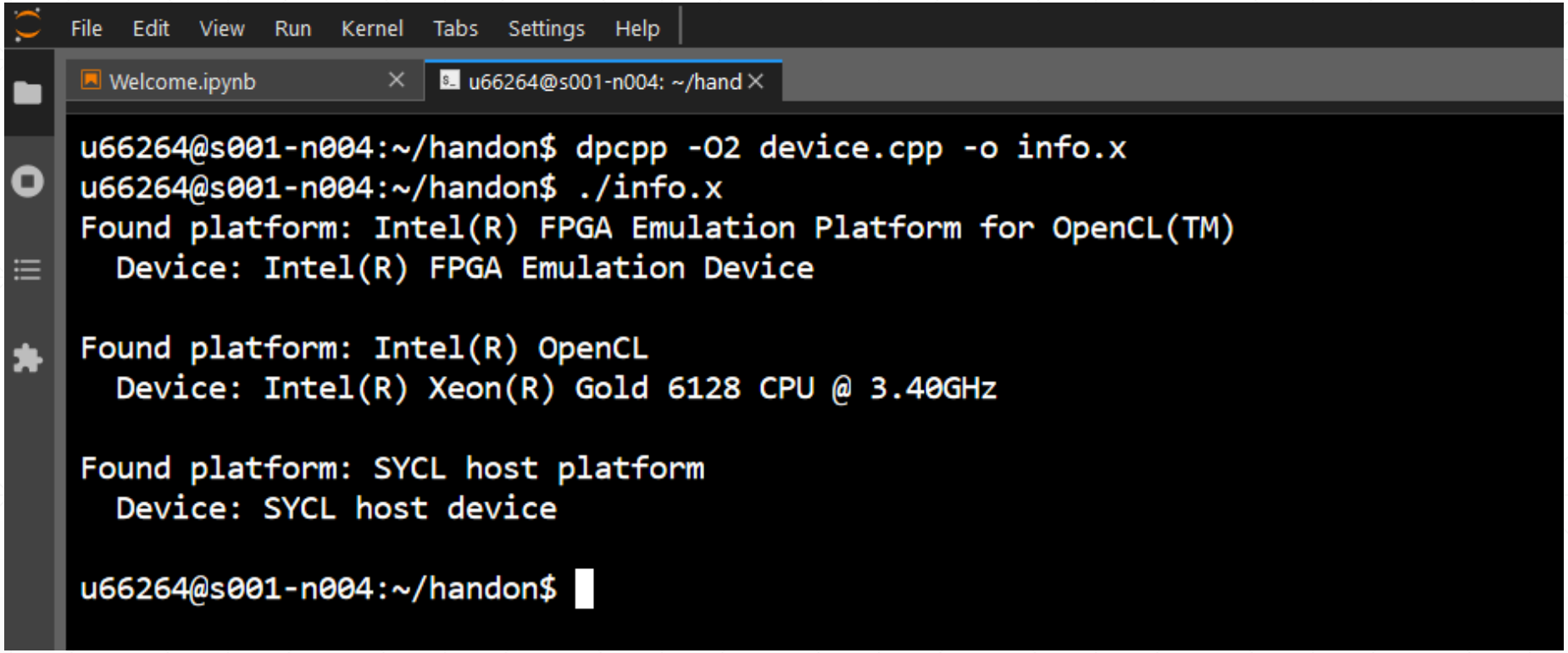


- New Launcher -> Terminal

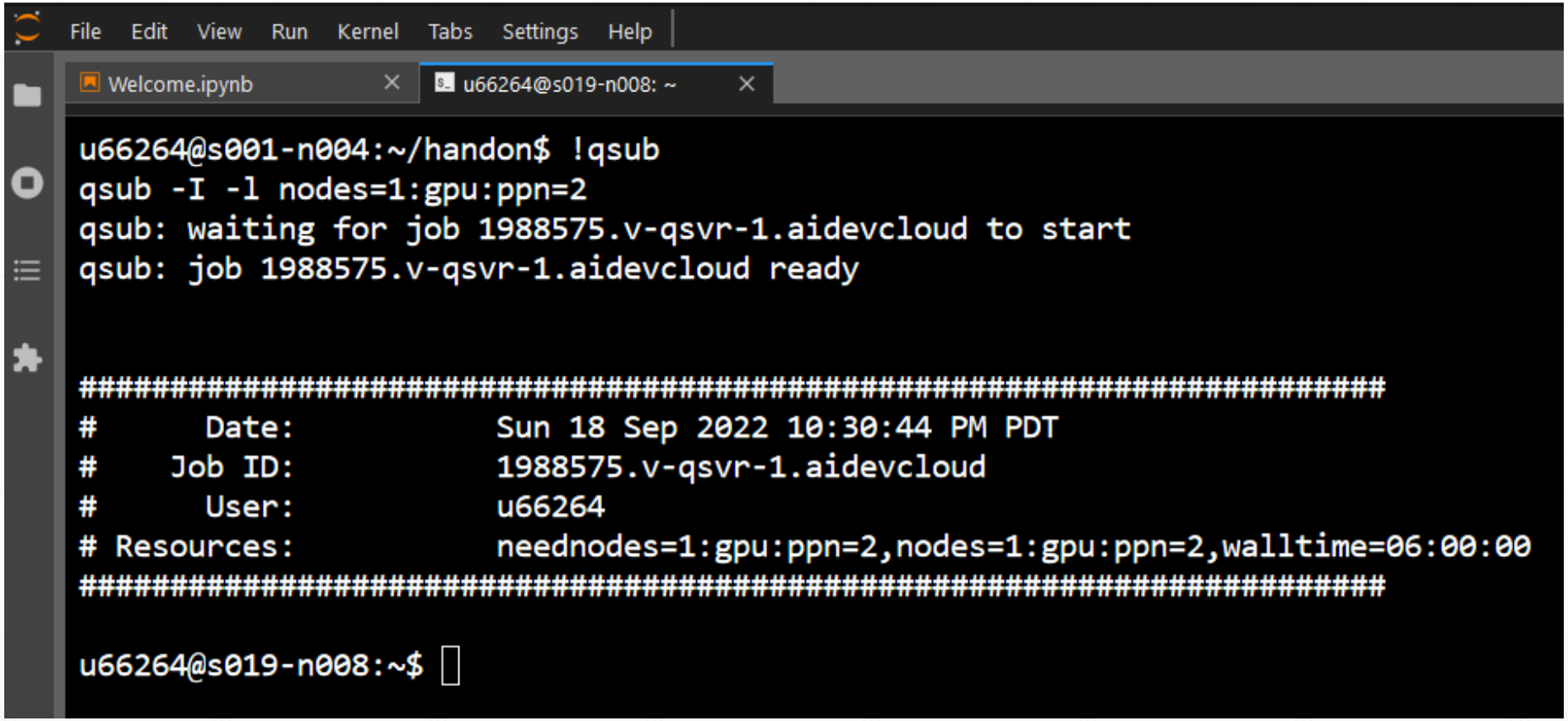
Device and Platform Discovery



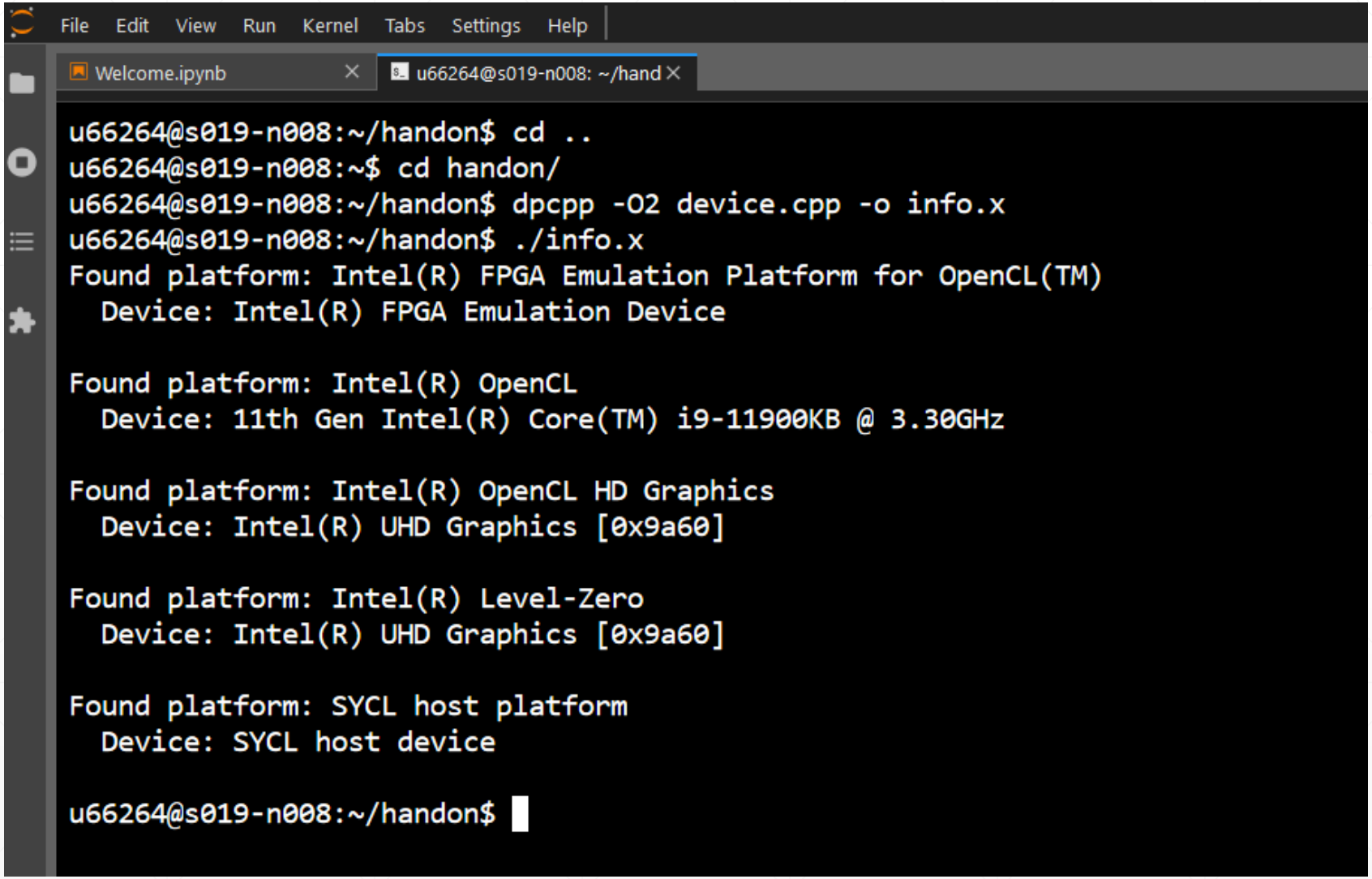
Device and Platform Discovery



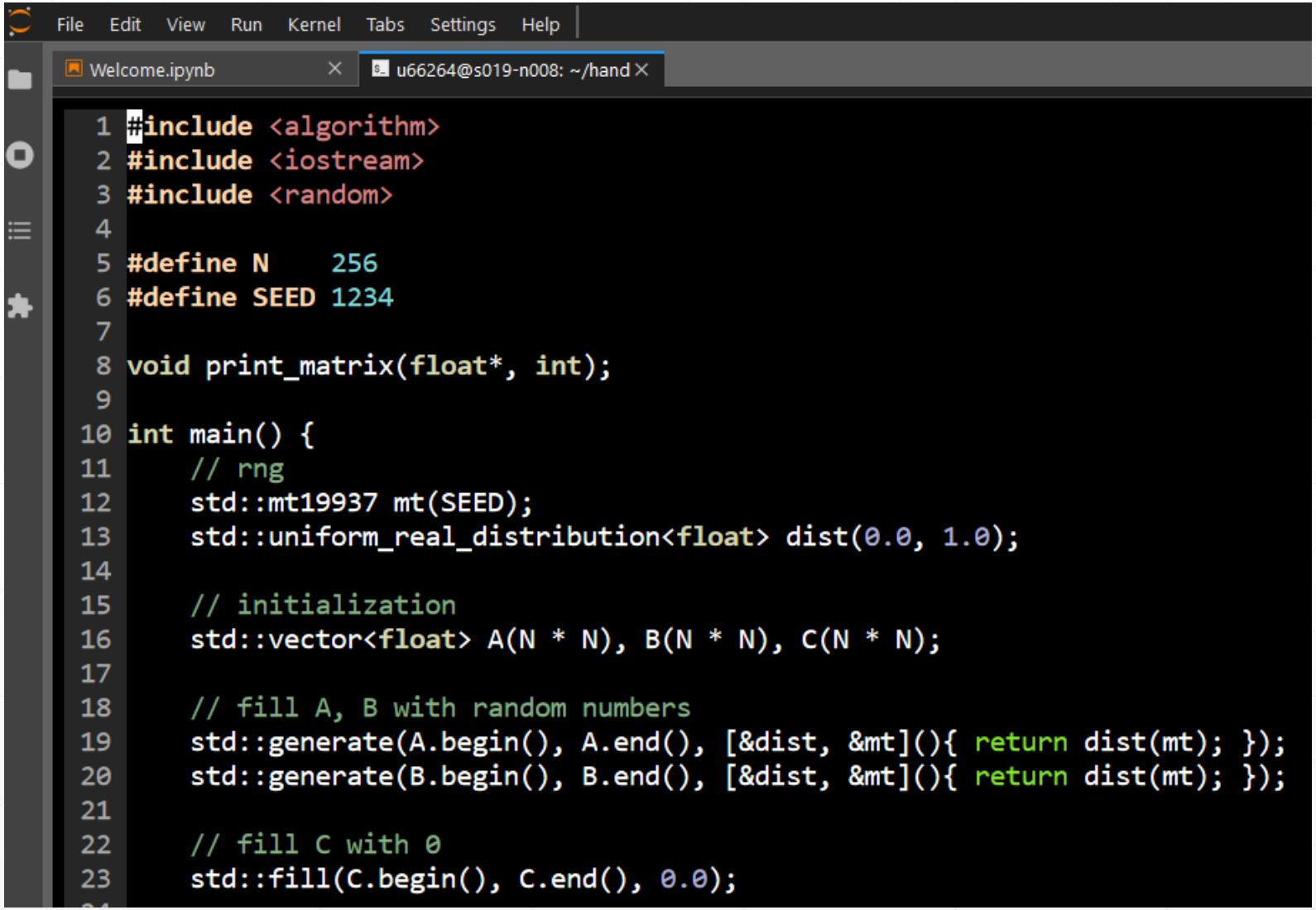
Device and Platform Discovery (GPU queue)



Device and Platform Discovery (GPU queue)



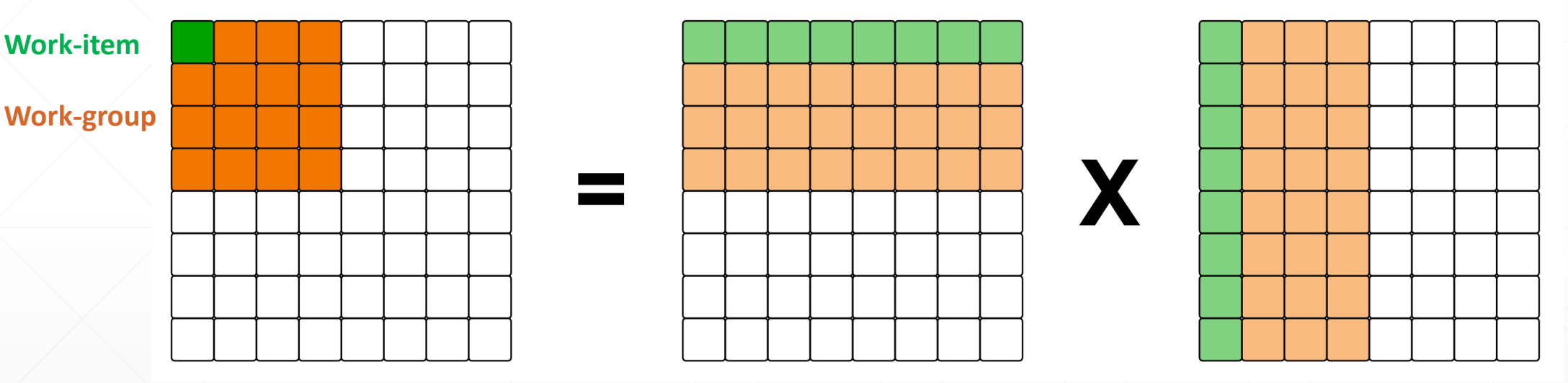
Matrix Multiplication: CPU version



Matrix Multiplication: CPU version (matmul.cpp)

```
25      // naive matmul
26      for (int i = 0; i < N; i++)
27          for (int j = 0; j < N; j++)
28              for (int k = 0; k < N; k++)
29                  C[i*N+j] += A[i*N+k] * B[k*N+j];
30
31      // print top left 4x4 block of C
32      print_matrix(C.data(), N);
33
34      return 0;
35  }
36
37  void print_matrix(float *matrix, int n) {
38      for (int i = 0; i < fmin(N,4); i++) {
39          for (int j = 0; j < fmin(N,4); j++) {
40              printf ("%12.5f", matrix[i*n+j]);
41          }
42          printf ("\n");
43      }
44  }
```

Matrix Multiplication on GPUs:

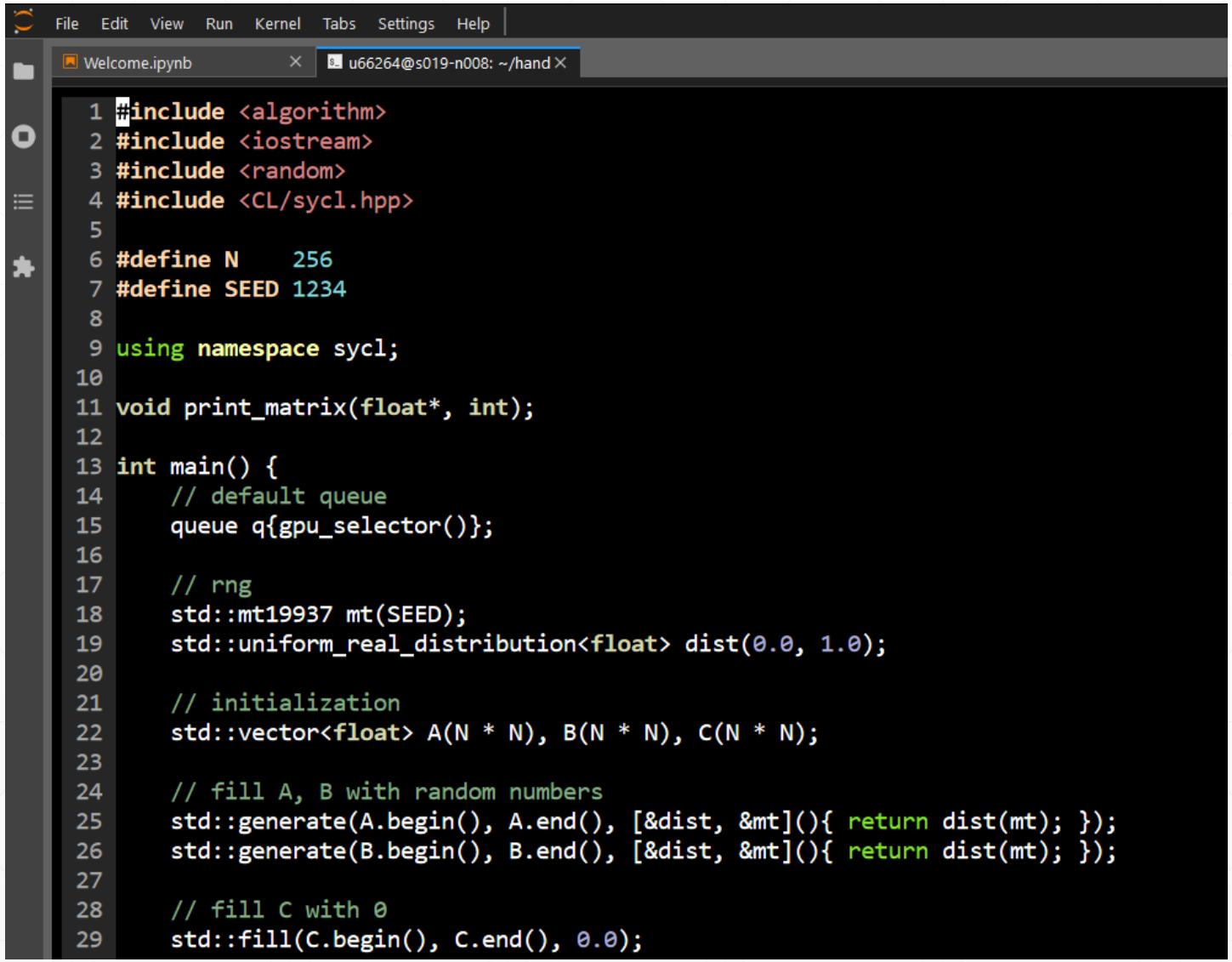


Matrix Multiplication: Buffer Version (matmul_buf.cpp)

SYCL header

SYCL namespace

Default queue



Matrix Multiplication: Buffer Version (matmul_buf.cpp)

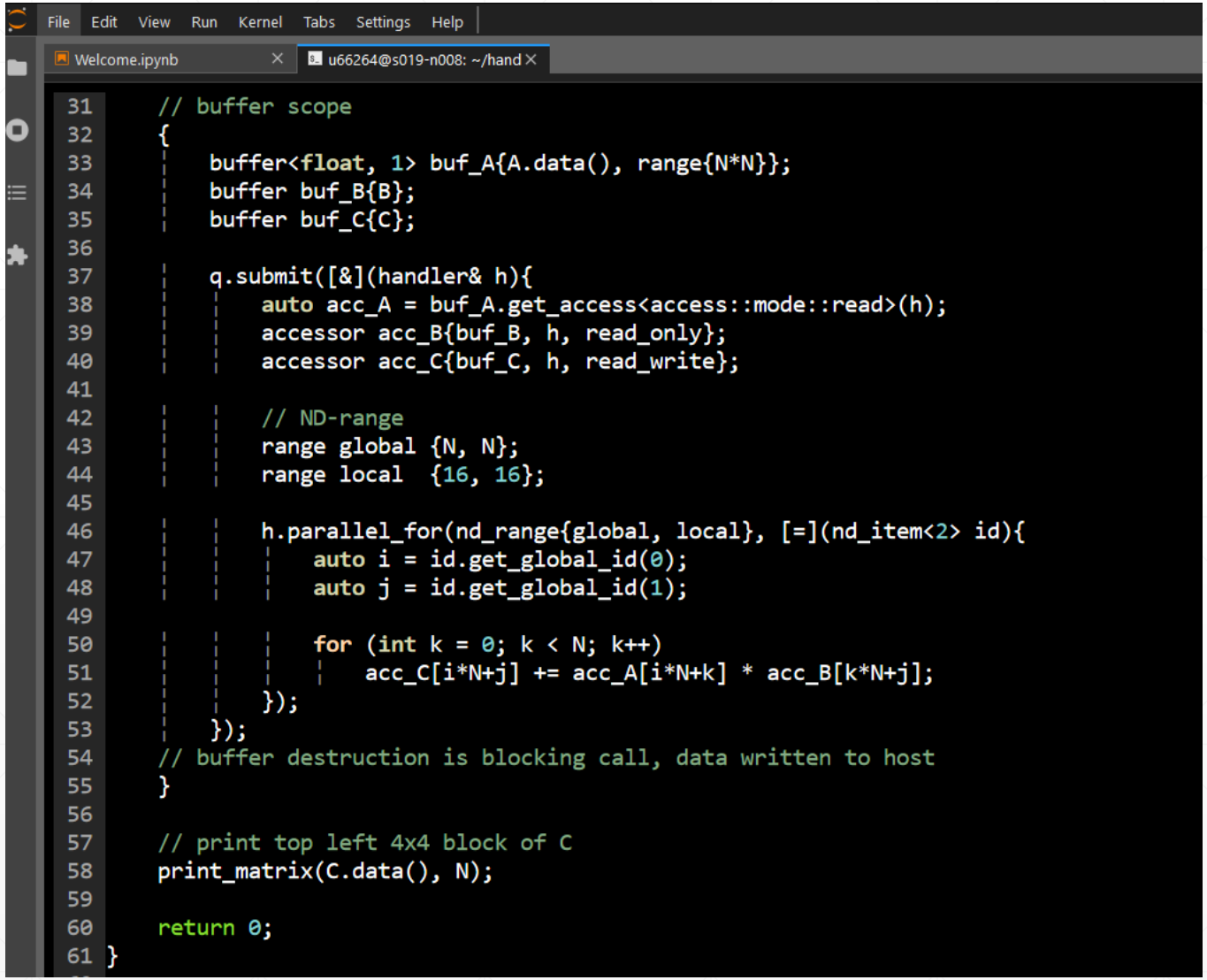
Buffer creation

Accessor creation

ND-range

Parallel for

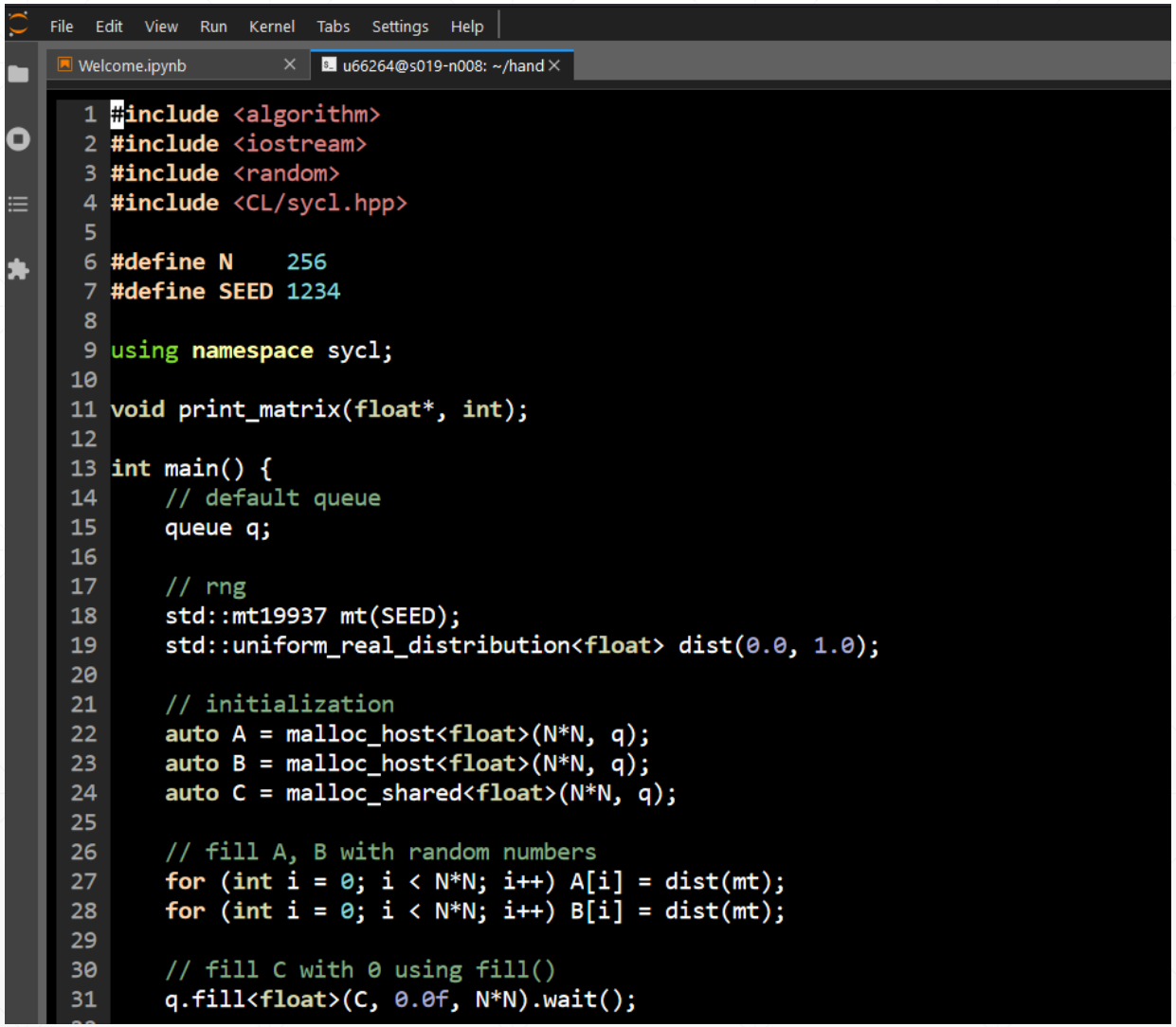
Data written to host




```
31      // buffer scope
32      {
33          buffer<float, 1> buf_A{A.data(), range{N*N}};
34          buffer buf_B{B};
35          buffer buf_C{C};
36
37          q.submit([&](handler& h){
38              auto acc_A = buf_A.get_access<access::mode::read>(h);
39              accessor acc_B{buf_B, h, read_only};
40              accessor acc_C{buf_C, h, read_write};
41
42              // ND-range
43              range global {N, N};
44              range local  {16, 16};
45
46              h.parallel_for(nd_range{global, local}, [=](nd_item<2> id){
47                  auto i = id.get_global_id(0);
48                  auto j = id.get_global_id(1);
49
50                  for (int k = 0; k < N; k++)
51                      acc_C[i*N+j] += acc_A[i*N+k] * acc_B[k*N+j];
52              });
53          });
54      // buffer destruction is blocking call, data written to host
55      }
56
57      // print top left 4x4 block of C
58      print_matrix(C.data(), N);
59
60      return 0;
61  }
```

Matrix Multiplication: Buffer Version (matmul_usm.cpp)

USM pointer allocation

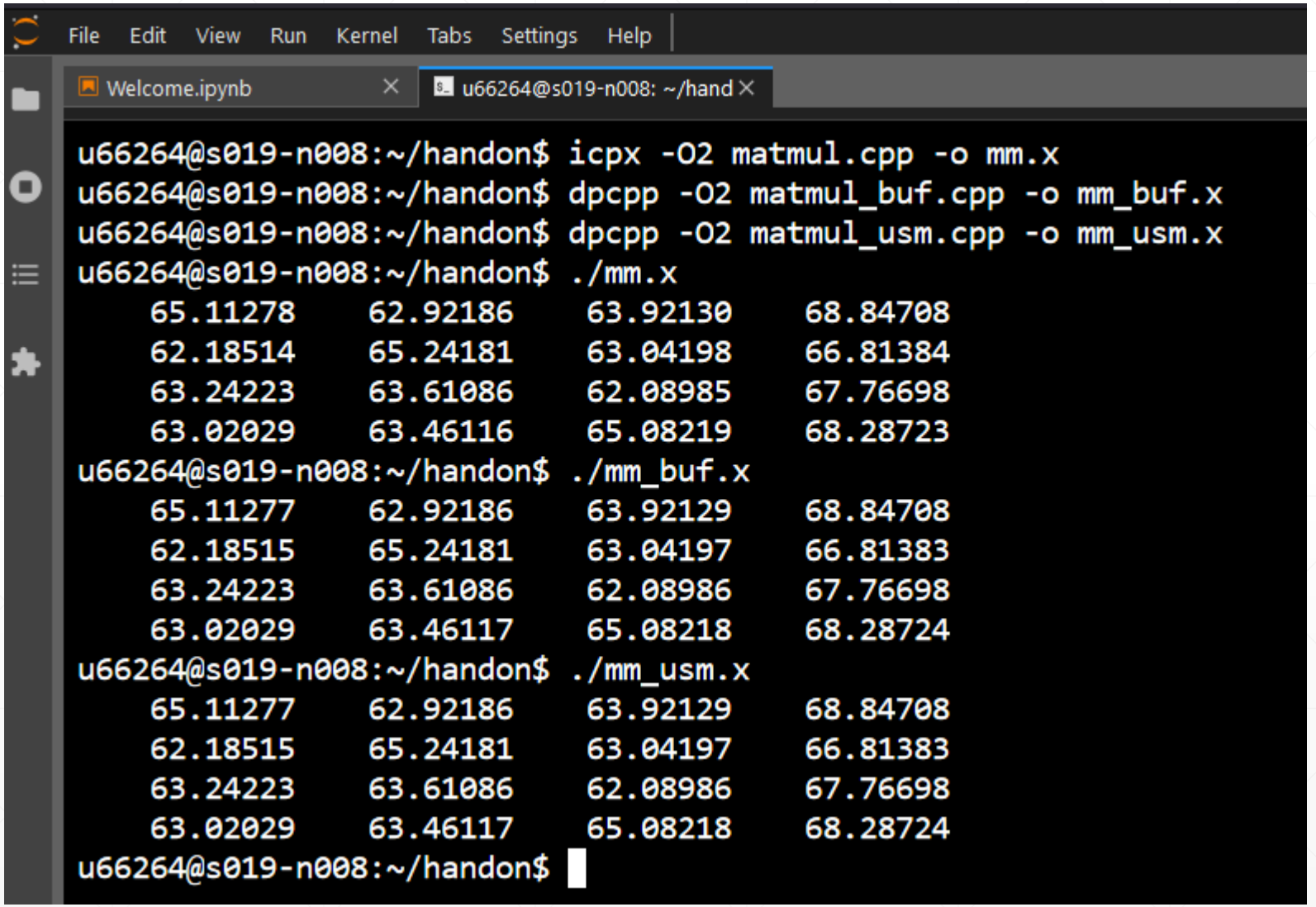


Matrix Multiplication: Buffer Version (matmul_usm.cpp)

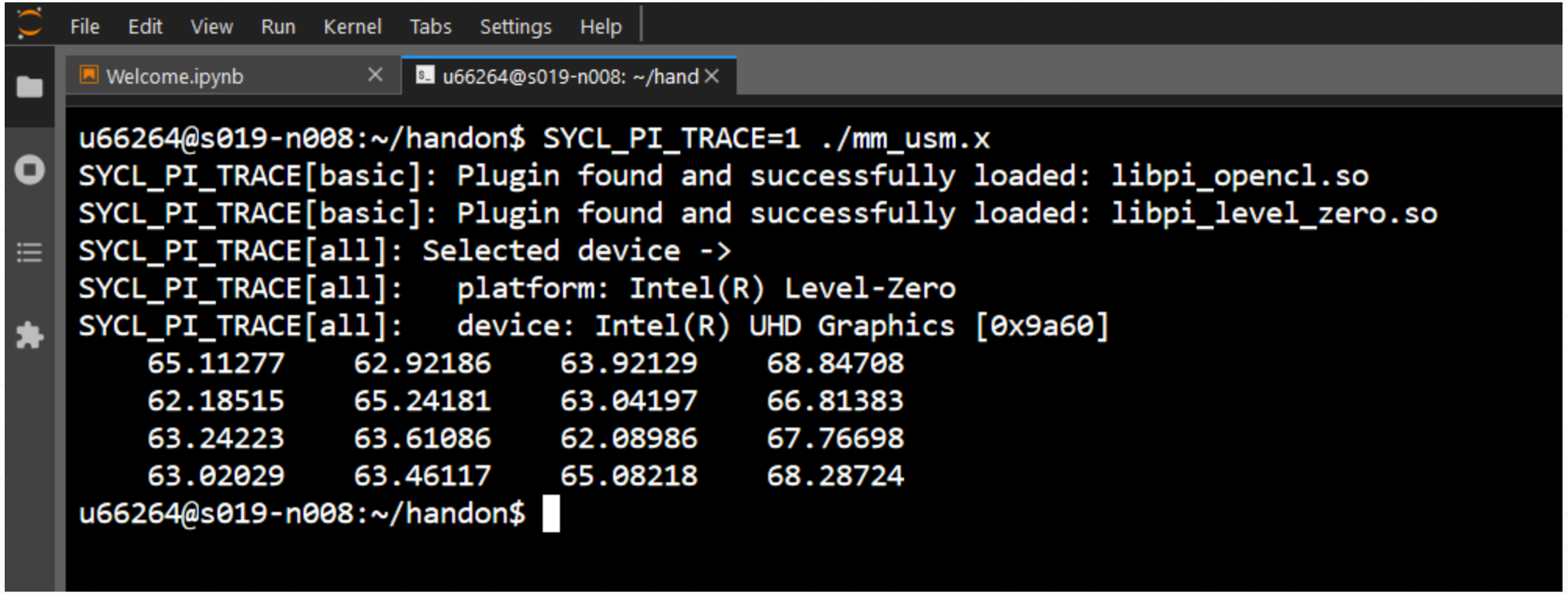
```
33      // ND-range
34      range global {N, N};
35      range local  {16, 16};
36
37      // queue shortcut
38      q.parallel_for(nd_range{global, local}, [=](nd_item<2> id){
39      ¦   auto i = id.get_global_id(0);
40      ¦   auto j = id.get_global_id(1);
41
42      ¦   for (int k = 0; k < N; k++)
43      ¦   ¦   C[i*N+j] += A[i*N+k] * B[k*N+j];
44      }).wait();
45
46      // print top left 4x4 block of C
47      print_matrix(C, N);
48
49      return 0;
50  }
```

- Advantages of USM:
  - Convenience for migration from C++ to SYCL with minimal changes
  - Less verbosity than buffer

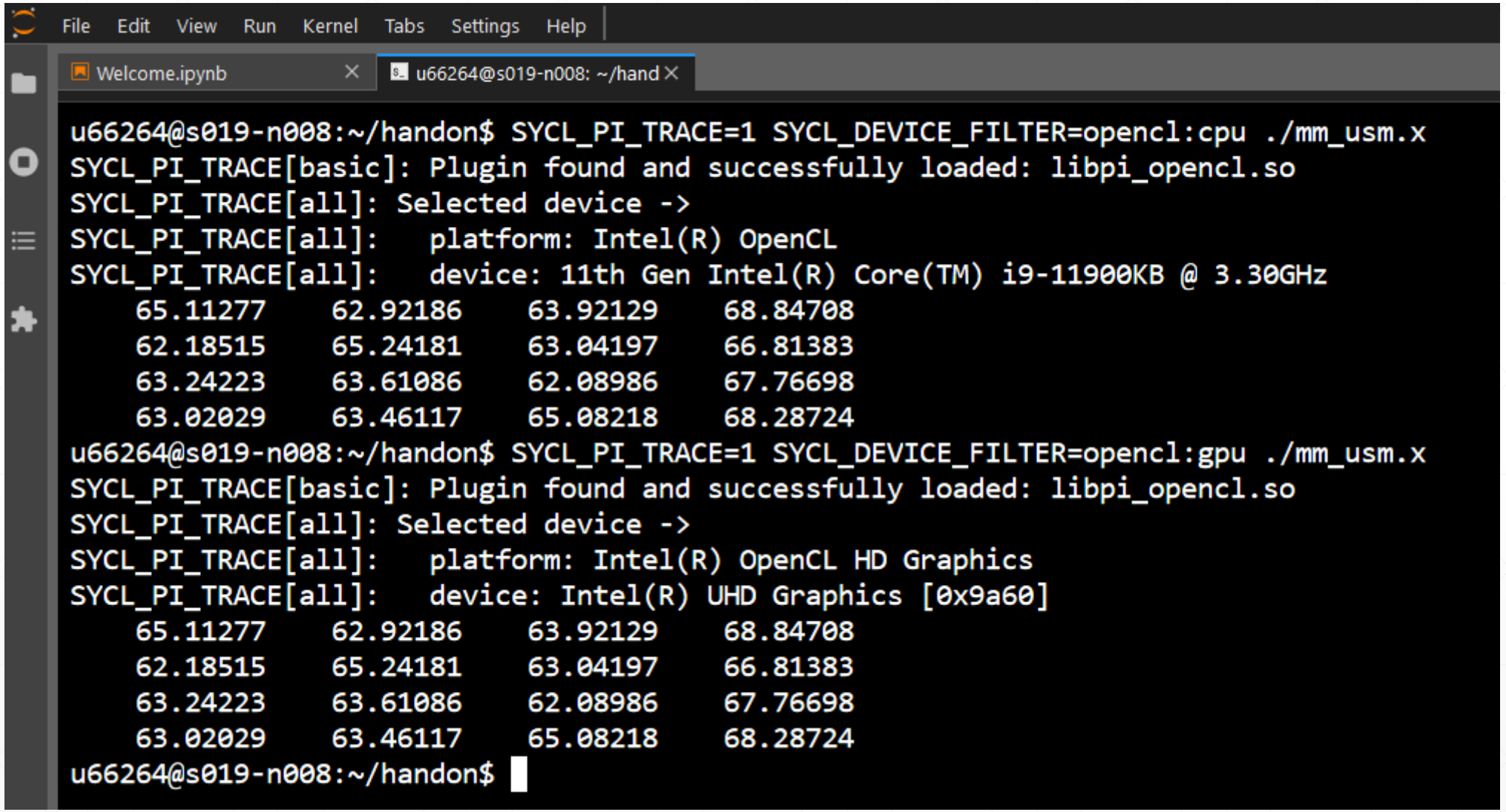
Results


File Edit View Run Kernel Tabs Settings Help
Welcome.ipynb
u66264@s019-n008: ~/hand
u66264@s019-n008:~/handon$ icpx -O2 matmul.cpp -o mm.x
u66264@s019-n008:~/handon$ dpcpp -O2 matmul_buf.cpp -o mm_buf.x
u66264@s019-n008:~/handon$ dpcpp -O2 matmul_usm.cpp -o mm_usm.x
u66264@s019-n008:~/handon$ ./mm.x
65.11278 62.92186 63.92130 68.84708
62.18514 65.24181 63.04198 66.81384
63.24223 63.61086 62.08985 67.76698
63.02029 63.46116 65.08219 68.28723
u66264@s019-n008:~/handon$ ./mm_buf.x
65.11277 62.92186 63.92129 68.84708
62.18515 65.24181 63.04197 66.81383
63.24223 63.61086 62.08986 67.76698
63.02029 63.46117 65.08218 68.28724
u66264@s019-n008:~/handon$ ./mm_usm.x
65.11277 62.92186 63.92129 68.84708
62.18515 65.24181 63.04197 66.81383
63.24223 63.61086 62.08986 67.76698
63.02029 63.46117 65.08218 68.28724
u66264@s019-n008:~/handon$

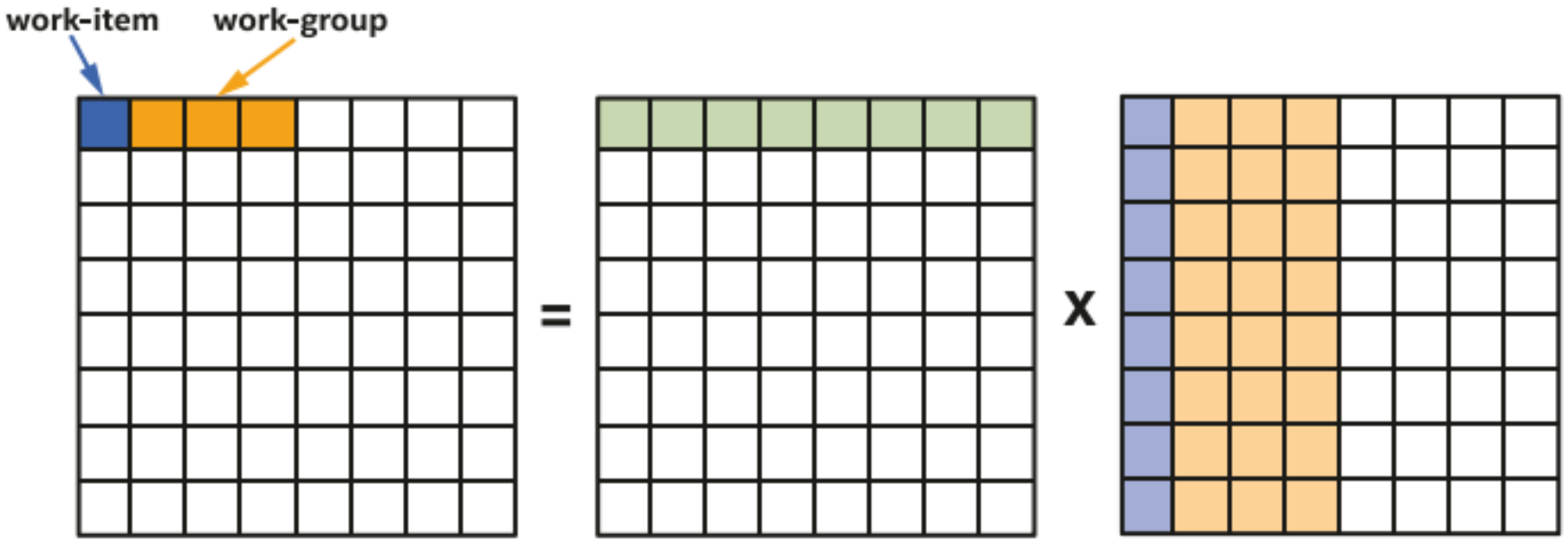
SYCL Debug with SYCL_PI_TRACE



Device Selection with SYCL_DEVICE_SELECTOR

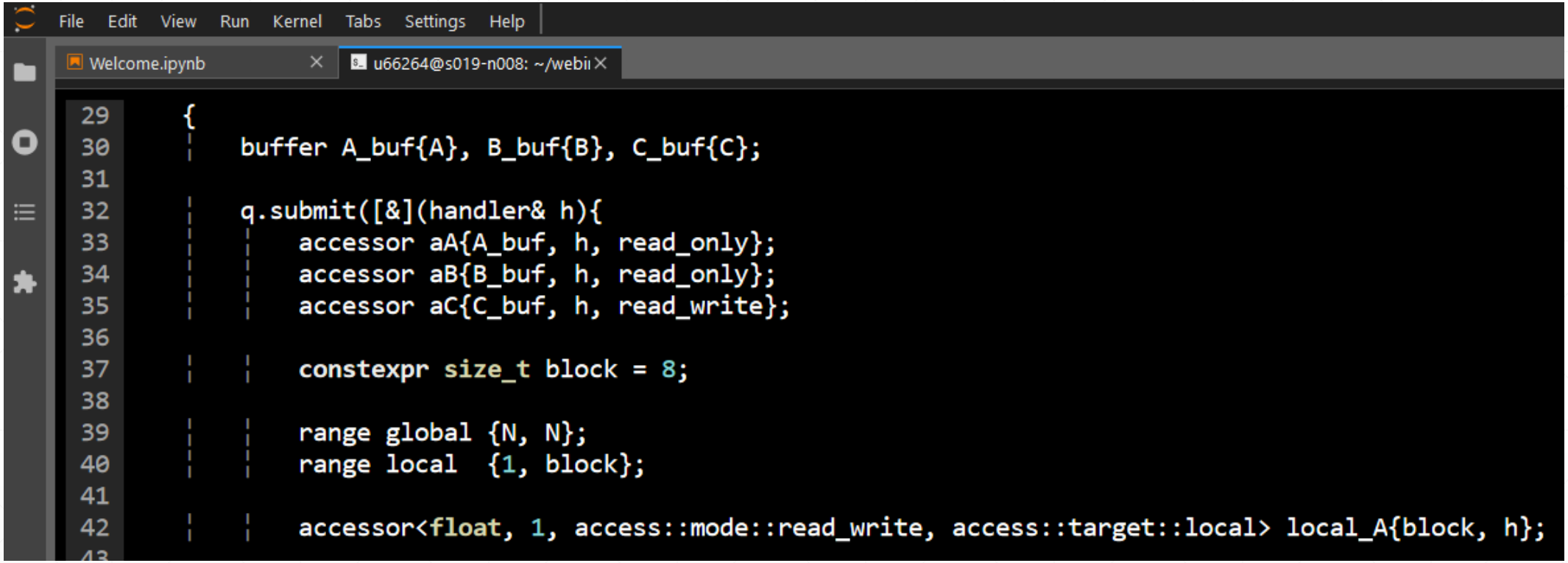


Optimization 1: Workgroup Local Memory



- In C++, matrix is stored in row order
  - A[i*N+k] is repeatedly accessed by member of workgroup
- Local memory optimization:
  - 2D {B, B} workgroup -> pseudo-1D {1, B} workgroup (B is tiling parameter)
  - Load each tile into memory and synchronize with barrier()

Optimization 1: Local Accessor



- Create local accessor per work group with *access::target::local* parameter

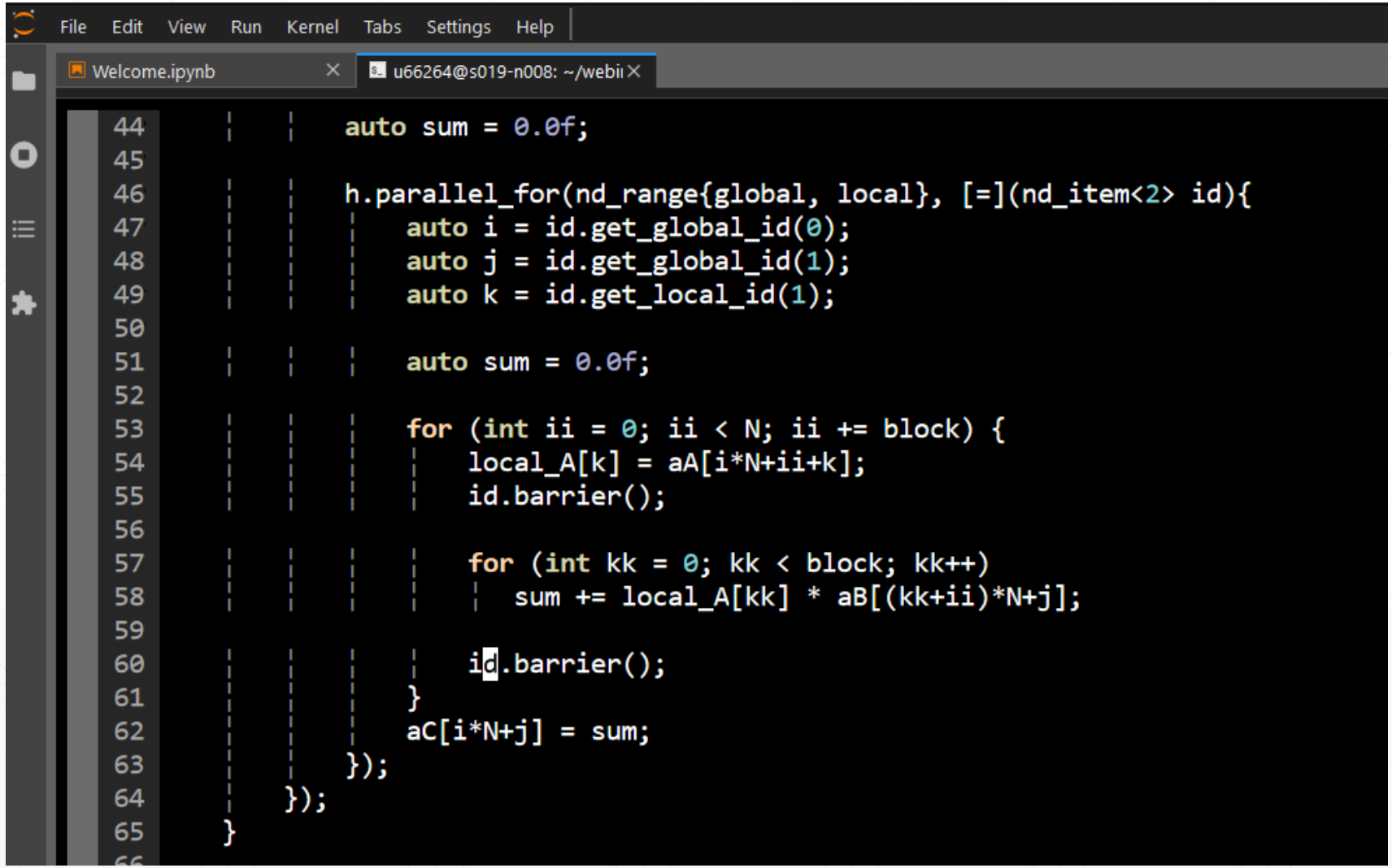
Optimization 1: Synchronization

Local workgroup id

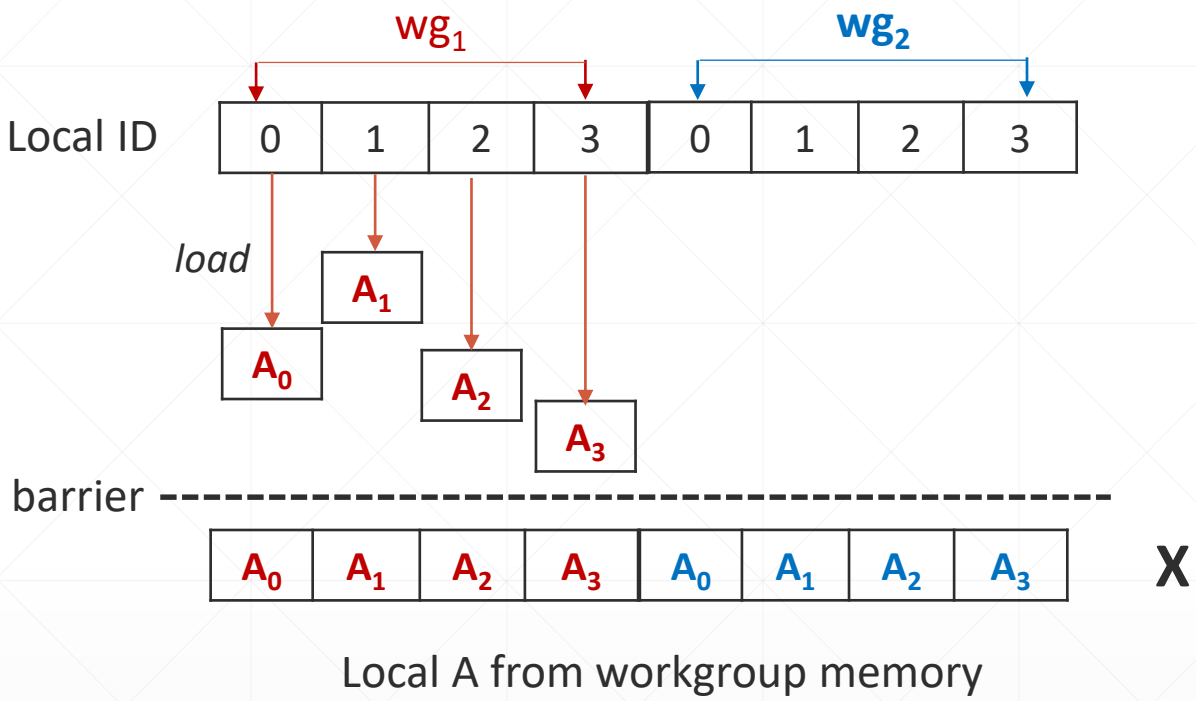
Step through each tile
Each k-thread load data
Synchronize

Tiling matmul

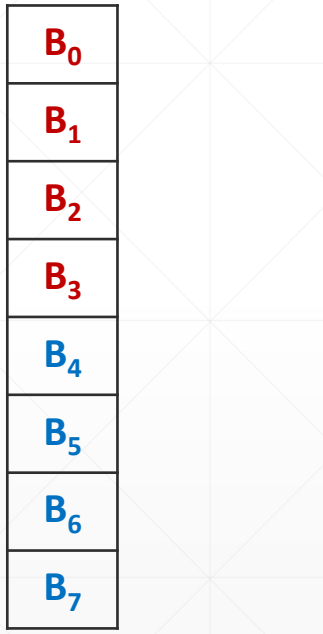
Final accumulation



Optimization 1: Synchronization



X



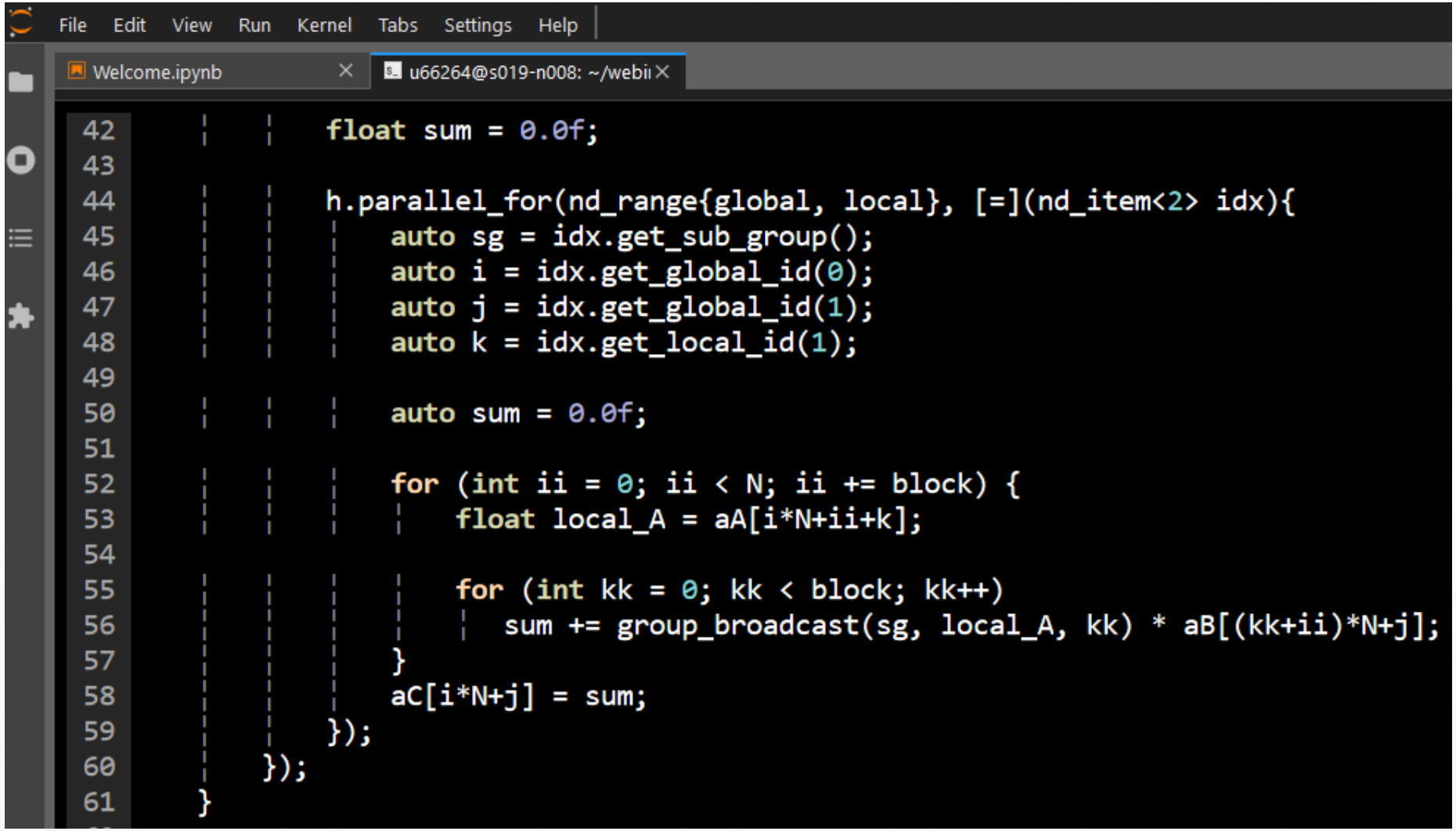
- Work item $C_0 = A_0*B_0 + A_1*B_1 + A_2*B_2 + A_3*B_3 + A_0*B_4 + A_1*B_5 + A_2*B_6 + A_3*B_7$

Optimization 2: Sub Group Broadcast

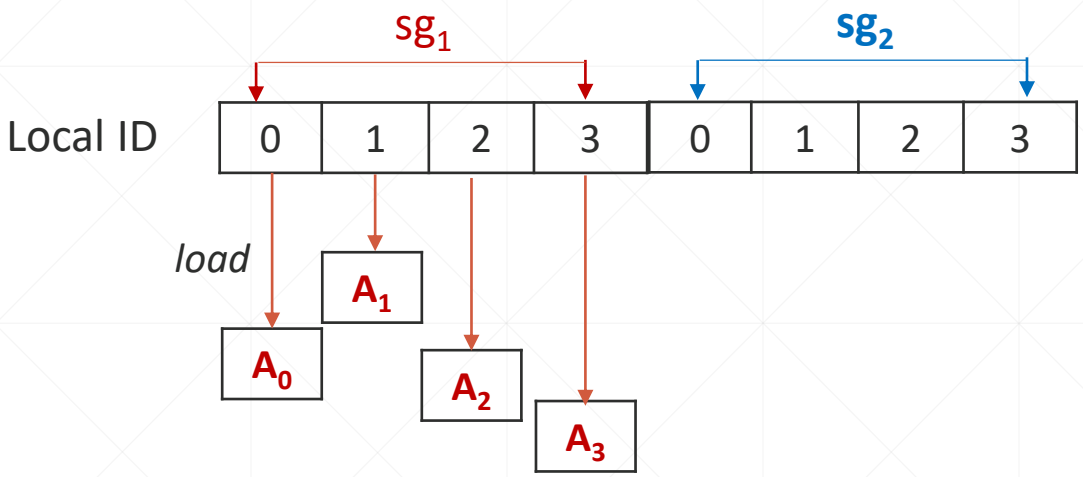
Sub group hander

Each thread load scalar local_A

Each thread broadcasts local_A



Optimization 2: Broadcast



local A in work-item's private memory

X

B_0	B_8
B_1	B_9
B_2	B_{10}
B_3	B_{11}
B_4	B_{12}
B_5	B_{13}
B_6	B_{14}
B_7	B_{15}

B from global memory

- Work item $C_0 = A_0*B_0 + A_1*B_1 + A_2*B_2 + A_3*B_3 + A_0*B_4 + A_1*B_5 + A_2*B_6 + A_3*B_7$
- Work item $C_1 = A_0*B_8 + A_1*B_9 + A_2*B_{10} + A_4*B_{11} + A_0*B_{12} + A_1*B_{13} + A_2*B_{14} + A_3*B_{15}$
- Each work item C_i uses exactly same (A_0,A_1,A_2,A_3) broadcasted from corresponding i-thread in sub group
- Access to local work group memory and explicit barrier is no longer required

Results

```
u66264@s019-n008:~/webinar/advanced_concepts/matmul$ icpx -O2 matmul.cpp -o mm.x
u66264@s019-n008:~/webinar/advanced_concepts/matmul$ dpcpp -O2 matmul_local.cpp -o mm_local.x
u66264@s019-n008:~/webinar/advanced_concepts/matmul$ dpcpp -O2 matmul_sg.cpp -o mm_sg.x
u66264@s019-n008:~/webinar/advanced_concepts/matmul$ ./mm.x
    65.11278    62.92186    63.92130    68.84708
    62.18514    65.24181    63.04198    66.81384
    63.24223    63.61086    62.08985    67.76698
    63.02029    63.46116    65.08219    68.28723
u66264@s019-n008:~/webinar/advanced_concepts/matmul$ ./mm_local.x
    65.11277    62.92186    63.92129    68.84708
    62.18515    65.24181    63.04197    66.81383
    63.24223    63.61086    62.08986    67.76698
    63.02029    63.46117    65.08218    68.28724
u66264@s019-n008:~/webinar/advanced_concepts/matmul$ ./mm_sg.x
    65.11277    62.92186    63.92129    68.84708
    62.18515    65.24181    63.04197    66.81383
    63.24223    63.61086    62.08986    67.76698
    63.02029    63.46117    65.08218    68.28724
u66264@s019-n008:~/webinar/advanced_concepts/matmul$
```