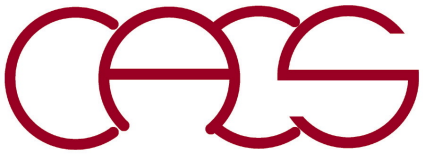
SYCL for Heterogeneous Architectures

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Quantitative & Computational Biology
University of Southern California

Email: anakano@usc.edu

Goal: Unified low-level programming of both CPU & various accelerators including GPU



Open Programming Models

- **OpenCL (Open Computing Language)**

  Open standard for programming heterogeneous devices

  https://www.khronos.org/opencl/

- **SYCL**

  High-level programming standard (or abstraction layer) for single-source C++ based language on heterogeneous computer architectures

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

  See SYCL 101 (Intel, 2023)

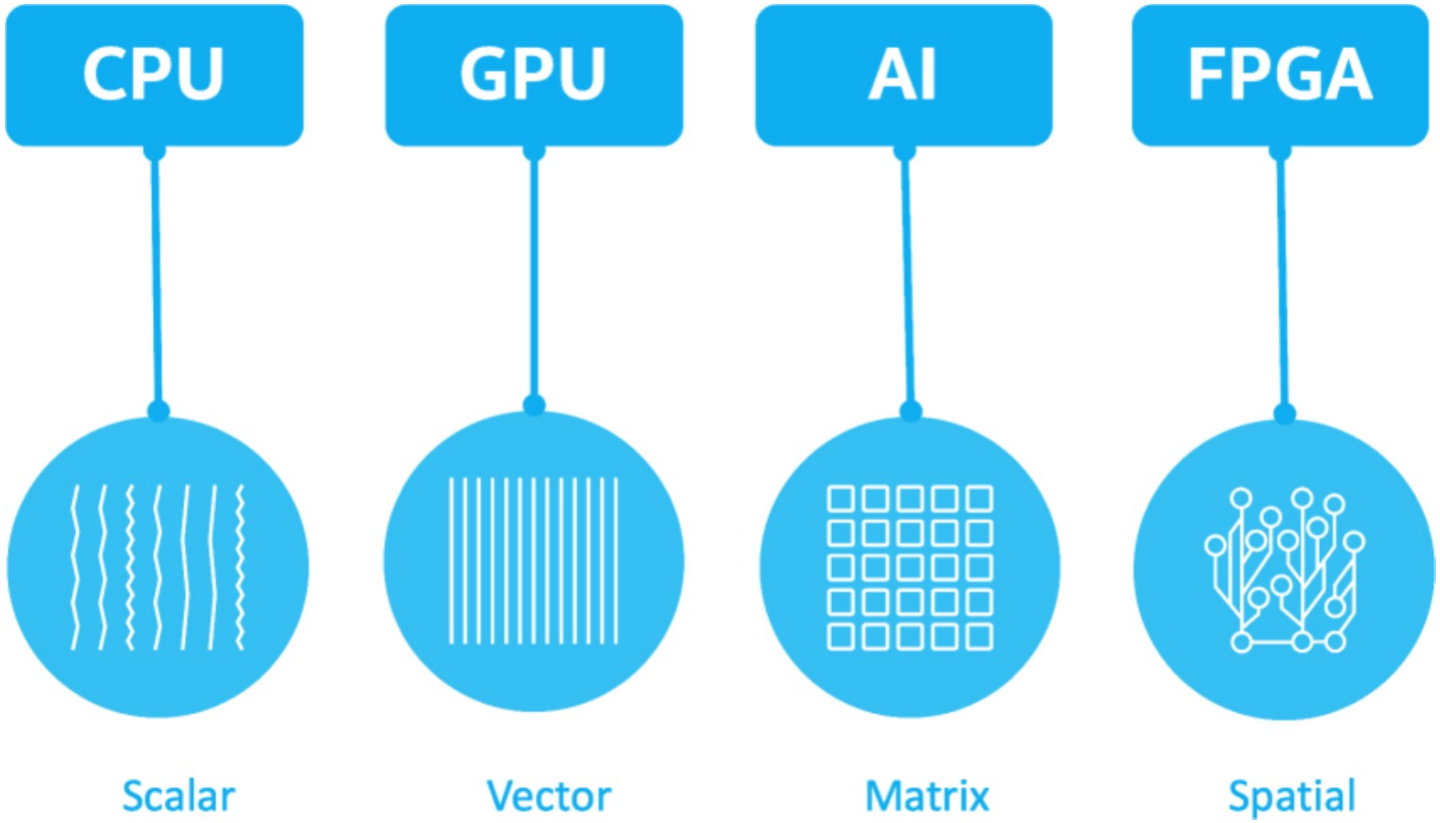
- **Data parallel C++ (DPC++)**

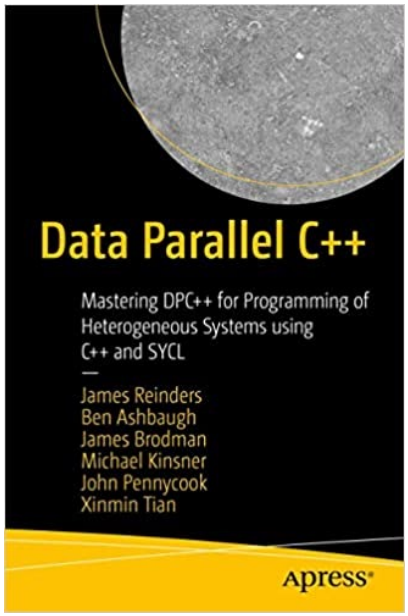
  Extension of C++ programming language, incorporating SYCL & other features, initially created by Intel; an open-source compiler is available on GitHub

  https://intel.github.io/llvm-docs/index.html

Platform Model

- **DPC++ unifies programming of central processing unit (CPU, scalar computation), graphics processing unit (GPU, vector computation), artificial-intelligence accelerator (AI, matrix or tensor) and field-programmable gate array (FPGA, spatial computation)**

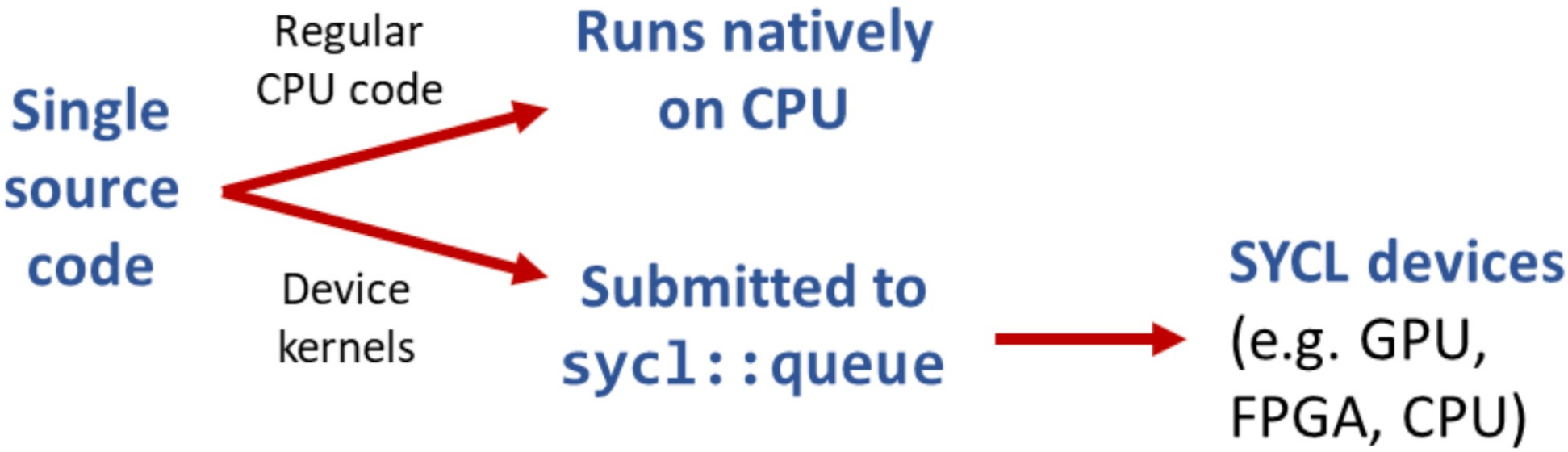




Data Parallel C++, B. Ashbaugh *et al*. (Apress, 2021);
sample codes at https://github.com/Apress/data-parallel-CPP

Host & Device Codes

- **Various accelerators (*e.g.*, GPU & FPGA) are referred to as devices**
- **DPC++ program can be a single source, *i.e.*, the same file contains both the host code to run on CPU and device kernels that run on devices**



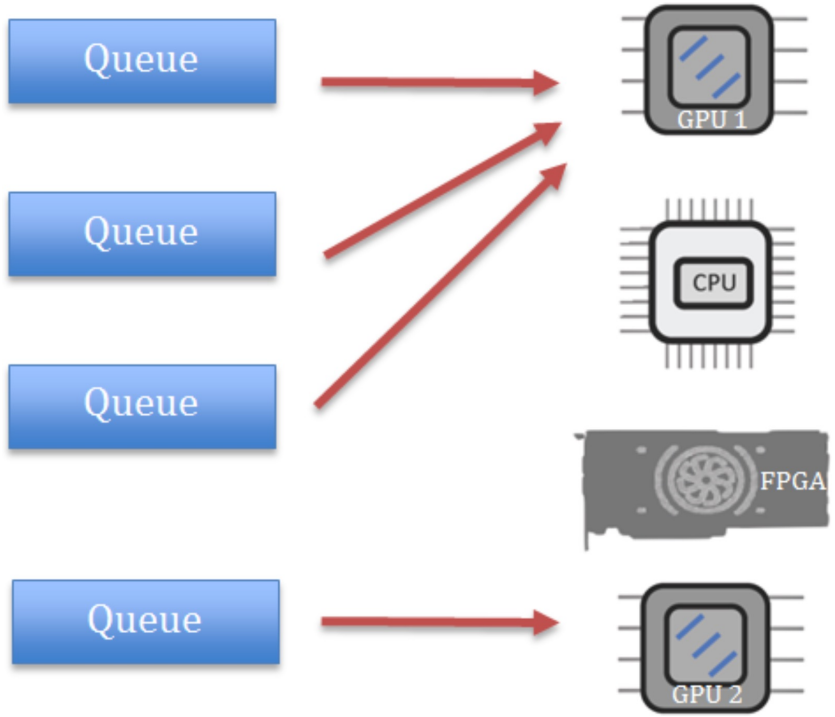
C++ in a nutshell

- **Class: User-defined data type that contains both member variables & member functions to work on them**
- **Object: Instance of a class**

Many C++ tutorials like: http://www.cplusplus.com/doc/tutorial

Queue

- **Queue: Abstraction to which work is submitted for execution on a single device (defined in SYCL as `sycl::queue` class)**
- **A queue is bound to a device upon construction of the queue object**
- **Selection of a device is achieved using `sycl::device_selector` class**



Built-in selectors:

`cpu_selector`	**CPU as device (useful for debugging)**
`gpu_selector`	**GPU**
`Intel::fpga_selector`	**FPGA**

Binding a Queue to a Device

get_device.cpp

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;
int main() {
  queue q( gpu_selector{} );
  std::cout << "Device: "
            << q.get_device().get_info<info::device::name>()
            << std::endl;
  return 0;
}
```

Header that defines sycl constructs

C++ I/O stream (*i.e.*, sequence of data elements for I/O)

Allows the use of sycl-defined constructs w/o sycl:: prefix

Initializer of a gpu_selector object

Construct a queue object

get_info() returns information of the device object, which in turn was returned by *get_device*() function of the queue

newline character in standard namespace

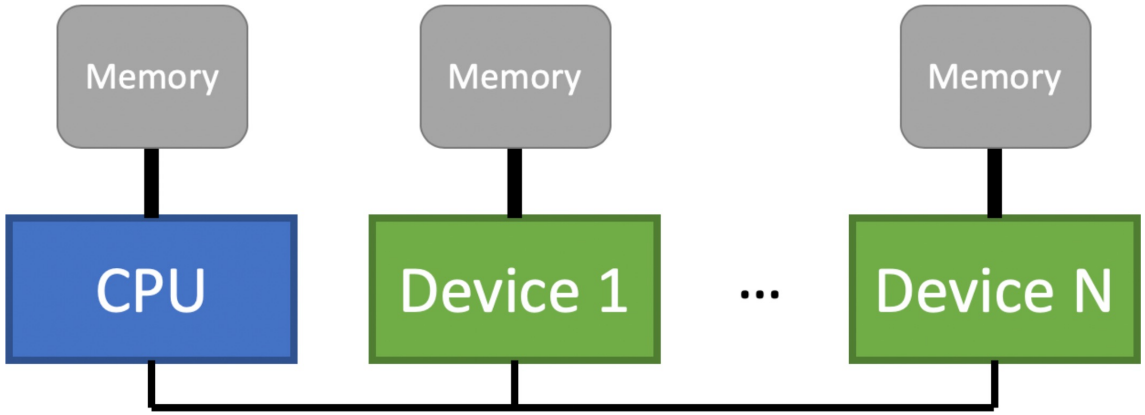
How to compile & run on Intel DevCloud*:

```
$ dpcpp -o get_device get_device.cpp
$ qsub -I -l nodes=1:gpu:ppn=2
$ ./get_device
Device: Intel(R) Gen9 HD Graphics NEO
```

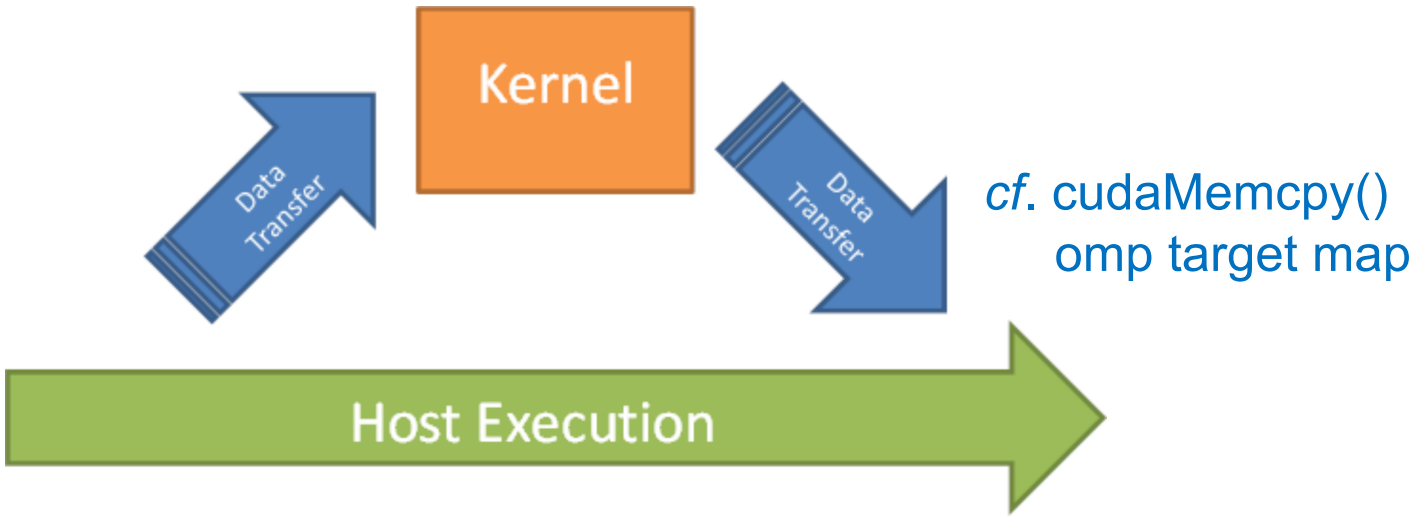
*Now use Intel Tiber AI Cloud instead

Host & Device Memory

- **Host & device have separate memories**



- **Data needed by a device kernel must be transferred from host memory to device memory prior to kernel execution, and results of kernel computation must be transferred back from device memory to host memory upon termination of kernel execution**



Data Management: Buffer

- **Buffer class: Abstraction of data object (not specific memory addresses)**
- **A buffer object can be created from existing data on the host; data is copied during buffer construction from the existing host allocation into the buffer object**
- **Range class: Represents one-, two- or three-dimensional range**

```
#define NTRD 512                    # of threads to be spawned on GPU
std::array<float, NTRD> sum;        Array of NTRD float elements
for (int i=0; i<NTRD; ++i) sum[i] = 0.0f;
range<1> sizeBuf{NTRD};             1-dim. range object initialized to NTRD
buffer<float, 1> sumBuf(sum.data(), sizeBuf);
```

Construct a 1-dim. float buffer object named sumBuf

Copy host data: data() returns the address of the first element of an array object

# of elements

Data Management: Accessor

- **Accessor class: Abstraction of reading & writing operations on buffer objects; usually created by `get_access()` method in the `buffer` class**

Type is automatically deduced from the initializer

```
auto sumAccessor =
  sumBuf.get_access<access::mode::read_write>(h);
```

get_access() method of a buffer object creates an accessor object, with which the buffer can be accessed with a specified access mode

Command-group handler (see next slide) that will access the buffer

Access mode	Description
read	Read-only access by device code
write	Device code will write into it
read_write	Read & write access

Device Code

- **Device code (*cf.* CUDA kernel) is submitted to a queue using `submit()` function of a `queue` object**
- **Argument to `submit()` is a command group function object in the form of lambda expression (*i.e.*, function with no name):**

  [*access mode to caller's variables*] (*argument list*) { *function body* }
- **The argument of the passed function is a handler to access the command group, which will be created by a runtime system and passed to the user through the argument**

```
queue q(gpu_selector{});

q.submit([&](handler &h){ Command group });
```

Access by address

Parallelization Construct

cf. omp parallel for

- **Device code can be parallelized using `parallel_for()` function, which takes a range of a loop index and a function as arguments**
- **Argument of the function is a loop index, which is of `id` class (index in a one-, two or three-dimensional range)**
- **Loop indices are distributed among multiple threads on device for parallel execution**

```
#define NTRD 512

range<1> sizeBuf{NTRD};

h.parallel_for(sizeBuf, [=](id<1> index) { Code for each index });
```

Access by value

Index in one-dim. range

Example: Computing the Value of π

- **Numerical integration**

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`

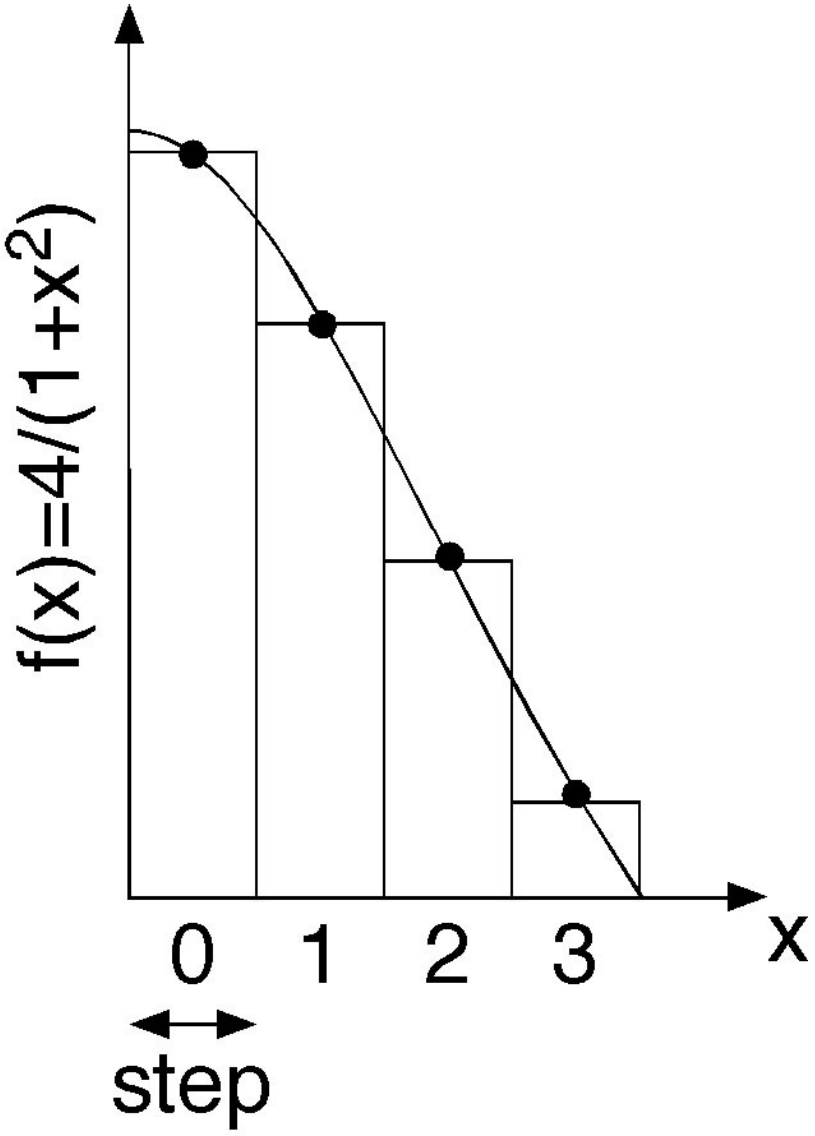
$x_i = (i+0.5)\Delta$ $(i = 0,\ldots,N\text{-}1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\,\Delta \cong \pi$$

```
#define NBIN 1000000

float sum = 0.0f;
float step = 1.0f/NBIN;
for (int i=0; i<NBIN; i++) {
  float x = (i+0.5f)*step;
  sum += 4.0f/(1.0f+x*x);
}
float pi = sum*step;
```

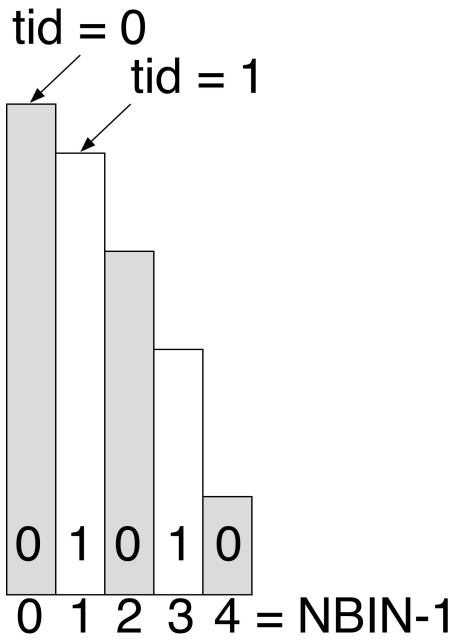
Area under the curve ≅ sum of N rectangular areas



Multithreading & Data Privatization

- **Multithreading:** Interleaved assignment of bins *i* among *NTHRD* threads, where thread ID $tid \in [0, NTHRD - 1]$
- **Data privatization:** Provide each thread a dedicated accumulator to avoid a race condition (*i.e.*, nondeterministic result depending on the timing of read & write operations on a shared variable by multiple threads)

```
for (int i=tid; i<NBIN; i+=NTHRD) {
  float x = (i+0.5)*step;
  sum[tid] += 4.0/(1.0+x*x);
}
```



- **Interthread reduction:** After all partial summations have been executed by multiple threads, the total sum must be computed by a single thread

```
float pi = 0.0f
for (int i=0; i<NTHRD; i++)
  pi += sum[i];
Pi *= step;
```

Computing π on a Device

From pi.cpp

```
q.submit([&](handler &h){
  auto sumAccessor =
  sumBuf.get_access<access::mode::read_write>(h);
  h.parallel_for(sizeBuf, [=](id<1> tid) {
    for (int i=tid; i<NBIN; i+=NTRD) {
      float x = (i+0.5f)*step;
      sumAccessor[tid] += 4.0f/(1.0f+x*x);
    }
  });  // End parallel_for
});  // End queue submit
```

Compile & run on DevCloud*

Interactive job on one GPU-accelerated computing node

```
u49162@login-2:~$ dpcpp -o pi pi.cpp
u49162@login-2:~$ qsub -I -l nodes=1:gpu:ppn=2
```

GPU-accelerated node has been allocated, and automatically logged in

```
u49162@s001-n181:~$ ./pi
Running on: Intel(R) Gen9 HD Graphics NEO
Pi = 3.14159
```

*Now use Intel Tiber AI Cloud instead

Synchronization

- **Synchronization between host & device can be achieved by buffer destruction**

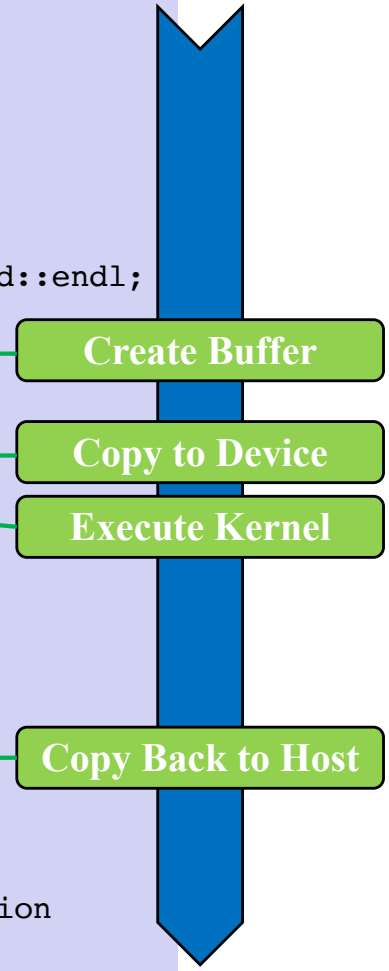
```
std::array<float, NTRD> sum;
{    Buffer is created in a separate scope
  queue q(gpu_selector{});
  range<1> sizeBuf{NTRD};
  buffer<float,1> sumBuf(sum.data(),sizeBuf);
  q.submit([&](handler &h){
    auto sumAccessor =
    sumBuf.get_access<access::mode::read_write>(h);
    h.parallel_for(sizeBuf, [=](id<1> tid) {
      for (int i=tid; i<NBIN; i+=NTRD) {
        float x = (i+0.5f)*step;
        sumAccessor[tid] += 4.0f/(1.0f+x*x);
      }
    });  // End parallel for
  });  // End queue submit
}    Buffer destructor is invoked when exiting from the scope
float pi=0.0f;
for (int i=0; i<NTRD; i++)
  pi += sum[i];
pi *= step;
std::cout << "Pi = " << pi << std::endl;
```

Buffer now takes ownership of sum array

Buffer relinquishes the ownership of data & copies its contents back to host memory

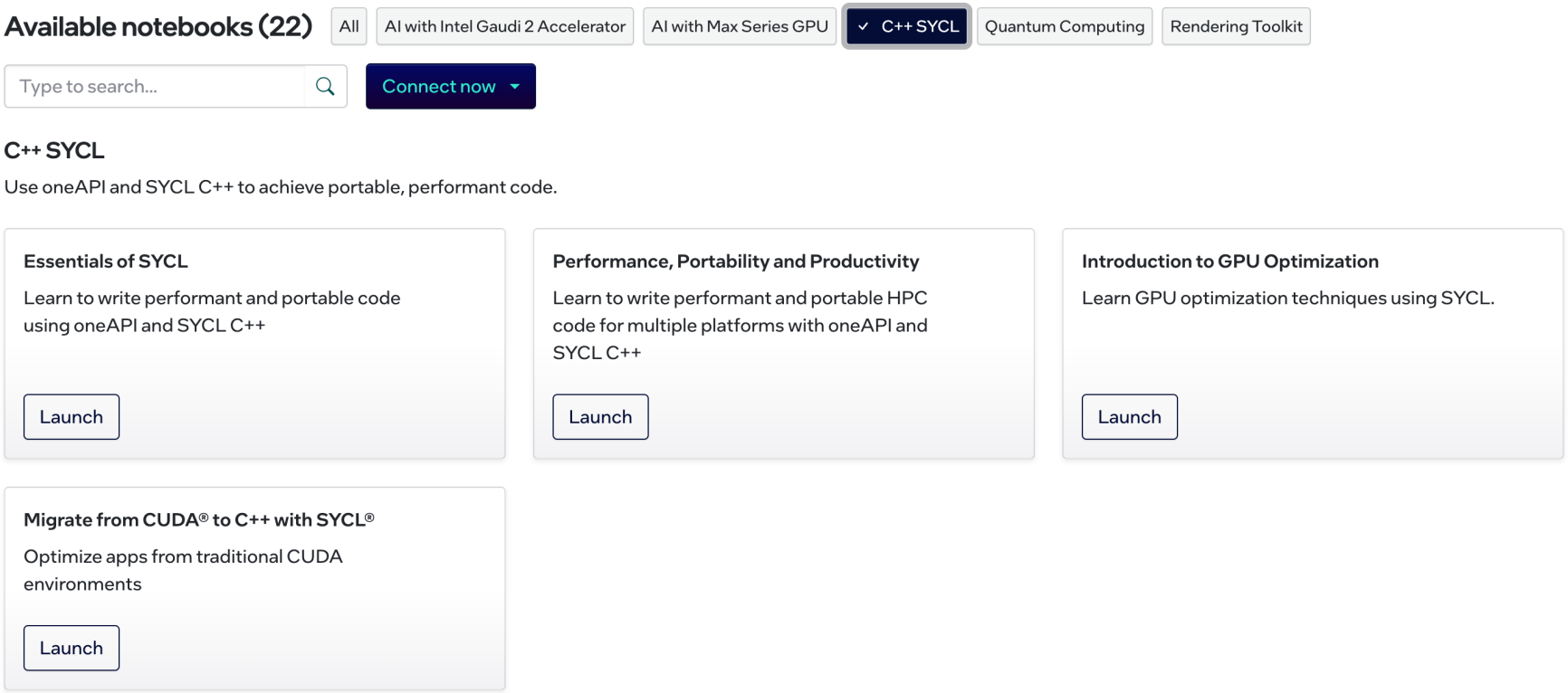
SYCL Program Pattern

```
#include <CL/sycl.hpp>
#include <iostream>
#include <array>
using namespace cl::sycl;
#define NBIN 1000000  // # of bins for quadrature
#define NTRD 512      // # of threads
int main() {
  float step = 1.0f/NBIN;
  std::array<float, NTRD> sum;
  for (int i=0; i<NTRD; ++i) sum[i] = 0.0f;
  {
    queue q(gpu_selector{});
    std::cout << "Running on: " <<
      q.get_device().get_info<info::device::name>() << std::endl;
    range<1> sizeBuf{NTRD};
    buffer<float, 1> sumBuf(sum.data(), sizeBuf);
    q.submit([&](handler &h){
      auto sumAccessor =
      sumBuf.get_access<access::mode::read_write>(h);
      h.parallel_for(sizeBuf, [=](id<1> tid) {
        for (int i=tid; i<NBIN; i+=NTRD) {
          float x = (i+0.5f)*step;
          sumAccessor[tid] += 4.0f/(1.0f+x*x);
        }
      });  // End parallel for
    });  // End queue submit
  }
  float pi=0.0f;
  for (int i=0; i<NTRD; i++)  // Thread reduction
    pi += sum[i];
  pi *= step;  // Multiply bin width to complete integration
  std::cout << "Pi = " << pi << std::endl;
  return 0;
}
```



Where to Go from Here

Sign up to Intel Tiber AI Cloud and go through SYCL tutorials



See also the SYCL 101 book:

https://www.intel.com/content/www/us/en/docs/sycl/introduction/latest/overview.html