SYCL I

Soner Steiner

Intel certified oneAPI Instructor
sonersteiner at gmail dot com

Overview

- Introduction
- Remainder of the lambda functions
- Compilation and run
- Queues and device selectors
- Dispatching mechanism
- Basic parallel kernels

What is SYCL?

What is oneAPI Implementation of SYCL?

oneAPI 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

Extends SYCL* standard

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

Fast-moving open collaboration feeding into the SYCL* standard

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

Why not CUDA?

- Unlike CUDA, SYCL supports data parallelism in C++ for all vendors and all types of architectures (not just GPUs).
- CUDA is focused on NVIDIA GPU support only, and efforts (such as HIP/ROCm) to reuse it for GPUs by other vendors have limited ability.
- With the explosion of accelerator architectures, only SYCL offers the support we need for harnessing this diversity and offering a multivendor/multiarchitecture approach to help with portability that CUDA does not offer.

new-golden-age-for-computer-architecture

Why Standard C++ with SYCL?

- Every program using SYCL is first and foremost a C++ program.
- SYCL takes C++ programming places it cannot go without SYCL.
- We don't believe the C++ standard will evolve to displace the need for SYCL anytime soon!?

Getting a C++ compiler with SYCL support?

- Compilers supporting SYCL: khoronos-sycl

- By using LLVM, the DPC++ compiler project has backends for numerous devices.

- This has already resulted in support for Intel, Nvidia and AMD GPUs, numerous CPUs and Intel FPGAs.

- oneAPI Tools, including the libraries, debuggers, DPC++ compiler and other tools, which are freely available.

Data Parallel C++

Standards-based, Cross-architecture Language

DPC++

=

ISO C++

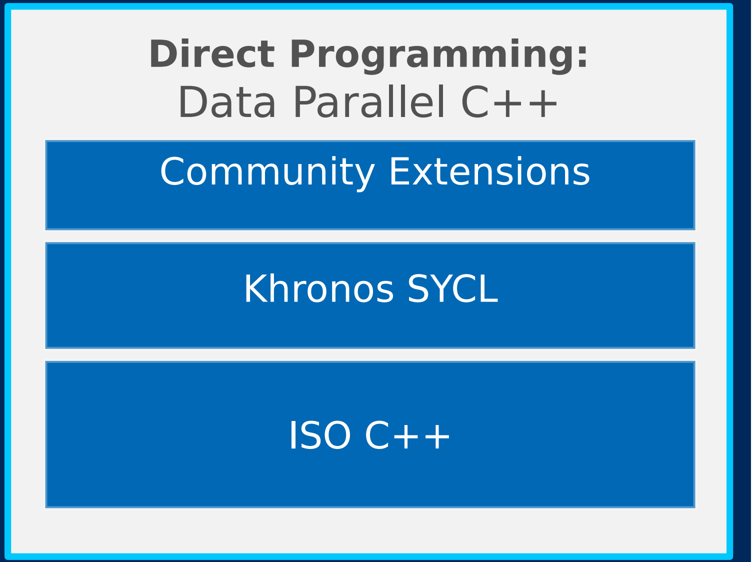
+

Khronos SYCL

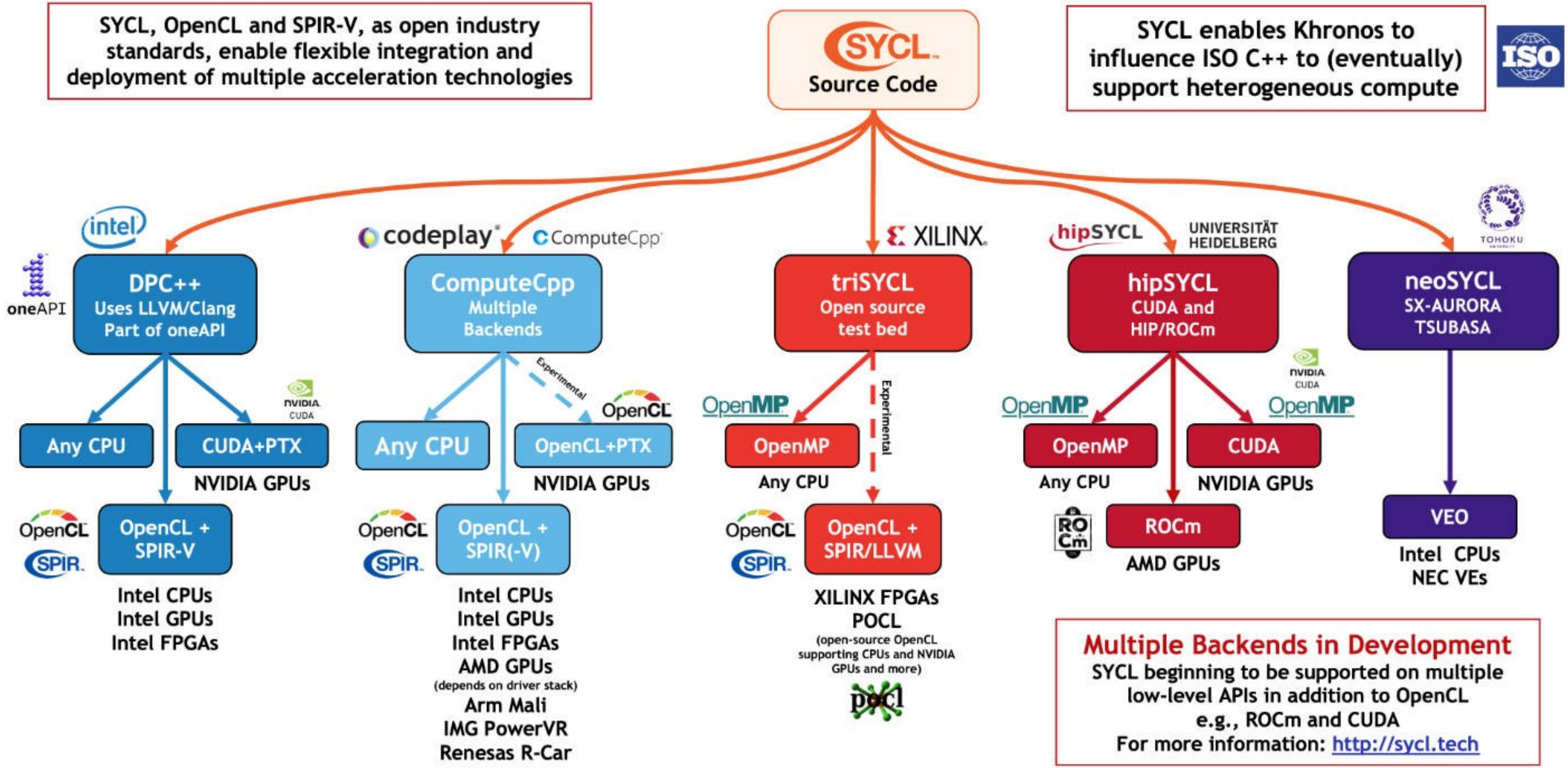
+

Community Extensions

tinyurl.com/sycl2020-support-in-dpcpp



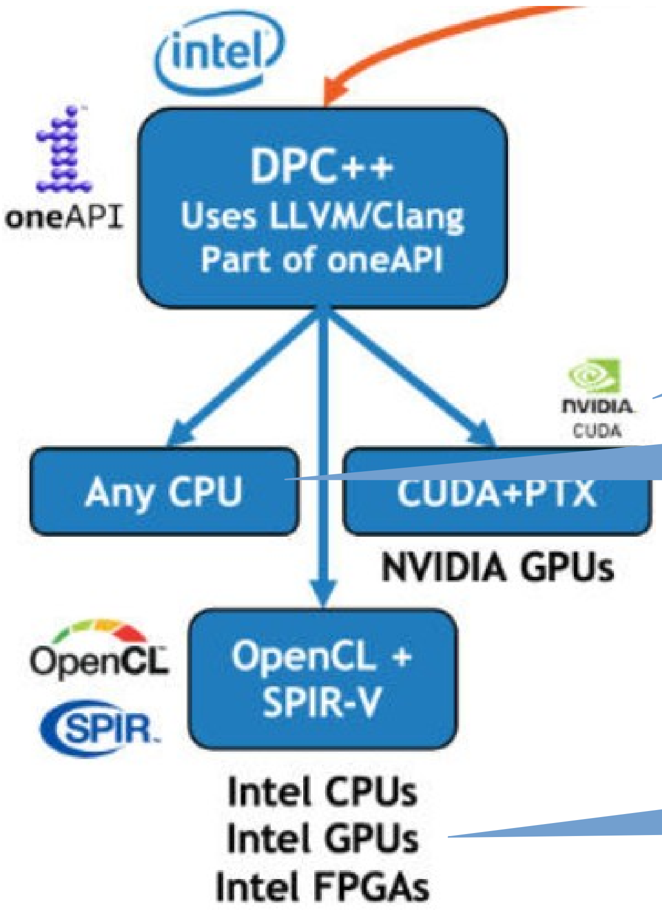
Intel DC++ in the SYCL ecosystem?



https://www.khronos.org/blog/sycl-2020-what-do-you-need-to-know

*This slide is prepared by The Khronos Group Inc

Intel DC++ in the SYCL ecosystem?



Supports NVIDIA GPUs

Supports Any [x86] GPUs

Supports Intel CPUs, GPUs and FPGAs

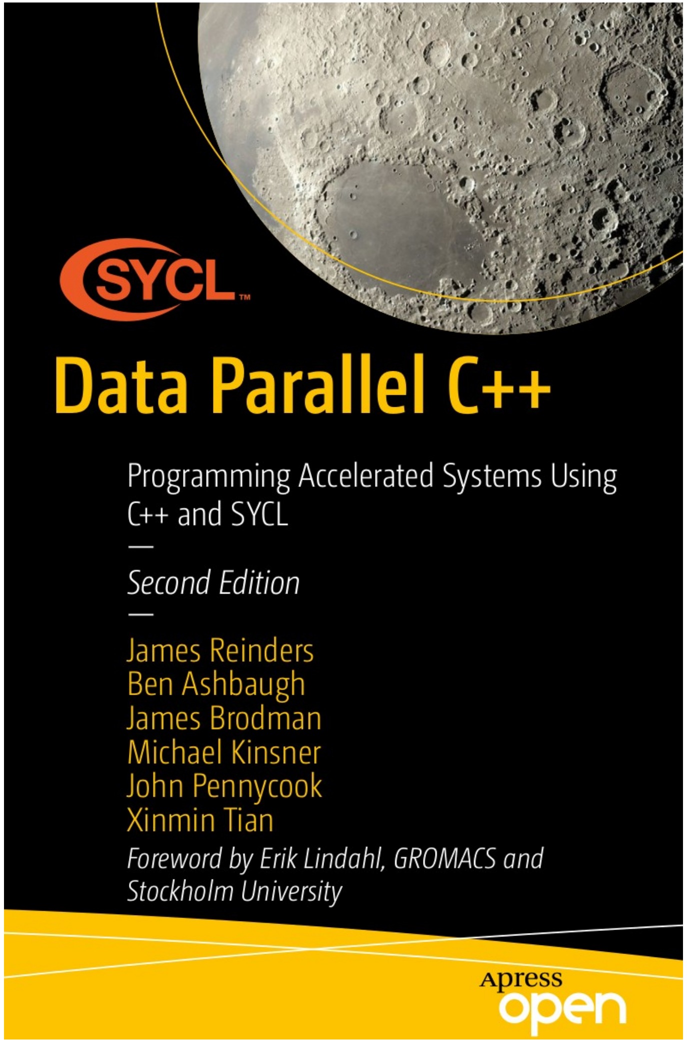
https://www.khronos.org/blog/sycl-2020-what-do-you-need-to-know

Many of the source examples are from Book:

- Source code accessible from
  $ oneapi-cli

```
(1) Create a project
(2) View oneAPI docs in browser
(q) Quit
```

```
(1) cpp
(2) python
(3) fortran
(b) Back
(q) Quit
```



data-parallel-c++-free-book-2ed

```
Samples
├─Compiler Infrastructure
│ ├─Intrinsics
│ ├─Matrix Multiply
│ └─OpenMP Offload
├─Graph Traversal
│ └─MergeSort OMP
├─Parallel Patterns
│ └─OpenMP* Reduction
├─Structured Grids
│ └─ISO3DFD OMP Offload
└─Tutorials Jupyter Notebooks
  └─OpenMP Offload C++ Tutorials
─C++SYCL
├─Combinational Logic
│ ├─Mandelbrot
│ └─Sepia Filter
├─Concurrent Kernels
├─convolutionSeparable
├─Dense Linear Algebra
│ ├─Base: Vector Add
│ ├─Complex Mult
│ ├─Jacobi Cuda Graphs
│ ├─Jacobi Iterative Solver
│ └─Matrix Multiply
```

```
Description
This sample Multiplies two large Matrices in
parallel using SYCL and OpenMP* (OMP)

The following tools are needed to build this sample
but are not locally installed: (icc)
You may continue and view the sample without the
prerequisites. To install the missing prerequisites,
visit:
https://www.intel.
com/content/www/us/en/developer/tools/oneapi/overvie
whpc-kit
or https://www.intel.
com/content/www/us/en/developer/tools/oneapi/overvie
wiot-kit
```

```
Press Backspace to return to previous screen!
```

Compilation and Run

prompt

$ source /opt/intel/oneapi/setvars.sh

$ dpcpp -O2 -g -std=c++17 -o 00Hello.out 00Hello.cpp

!

NOW

$ icpx -fsycl -O2 -g -std=c++17 -o 00-Hello.x 00-Hello.cpp

List SYCL Devices available

$ sycl-ls [--verbose]

Where Code Executes

```
$sycl-ls
[opencl:acc:0] Intel(R) FPGA Emulation Platform for OpenCL(TM), Intel(R) FPGA
Emulation Device 1.2 [2022.15.12.0.01_081451]
[opencl:cpu:1] Intel(R) OpenCL, Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz 3.0
[2022.15.12.0.01_081451]
[opencl:gpu:2] Intel(R) OpenCL HD Graphics, Intel(R) UHD Graphics 630 [0x9bc5]
 3.0 [21.36.20889]
```

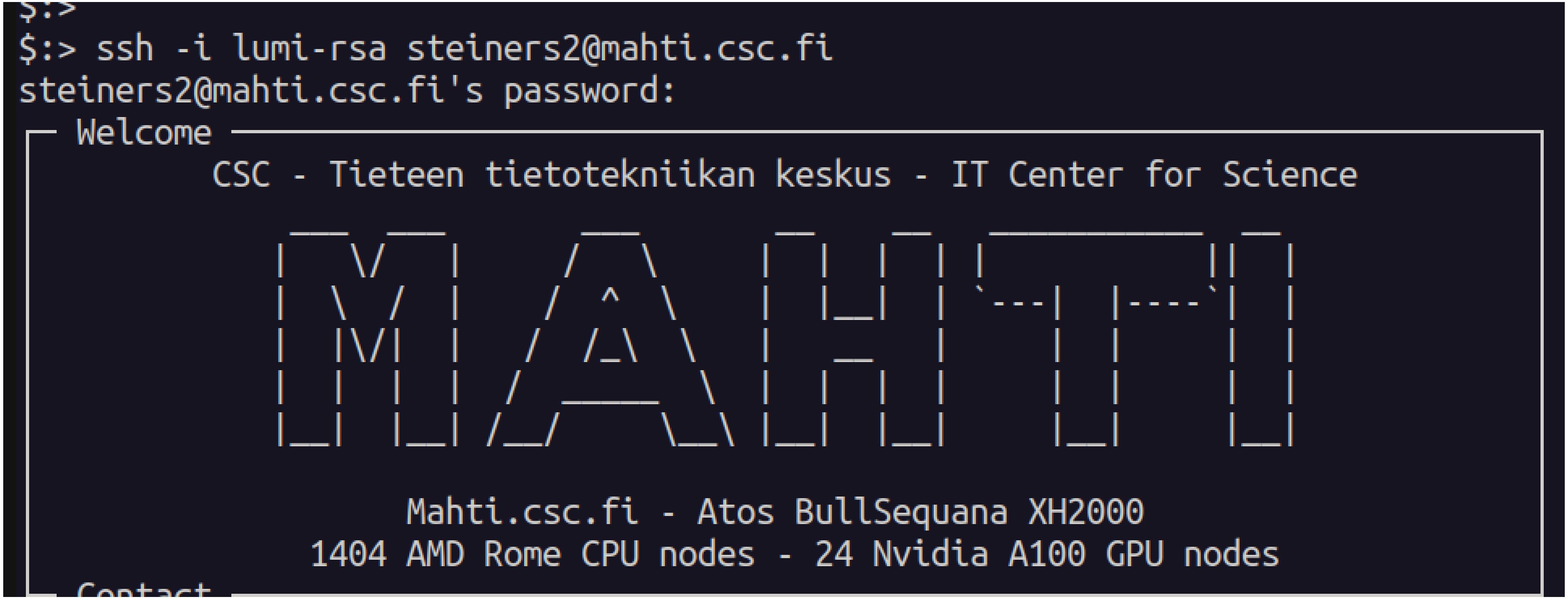
three devices on this computer

Where Code Executes

```
$:> sycl-ls
[opencl:cpu:0] Intel(R) OpenCL, 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz OpenCL 3.0 (Build 0) [2023.16.10.0.17_1
60000]
[opencl:acc:1] Intel(R) FPGA Emulation Platform for OpenCL(TM), Intel(R) FPGA Emulation Device OpenCL 1.2  [2023.16.6.
0.22_223734]
[opencl:cpu:2] Intel(R) OpenCL, 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz OpenCL 3.0 (Build 0) [2023.16.6.0.22_22
3734]
[ext_oneapi_cuda:gpu:0] NVIDIA CUDA BACKEND, NVIDIA GeForce RTX 3070 Laptop GPU 8.6 [CUDA 11.4]
```

four devices on my computer

Where Code Executes



Where Code Executes

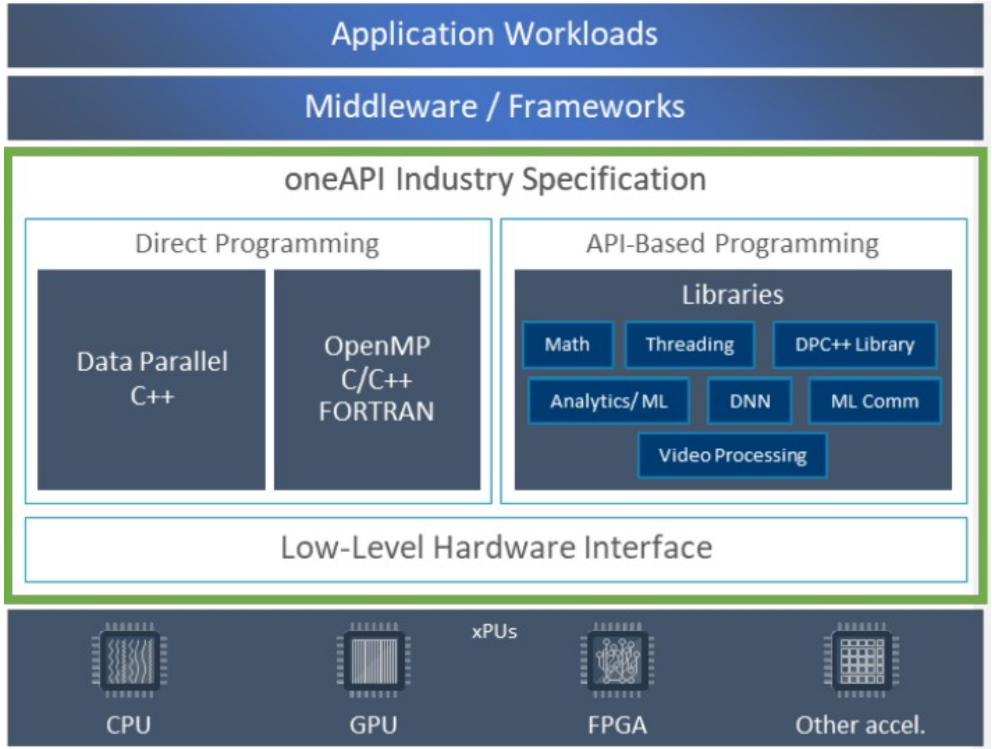
```
[steiners2@mahti-login15 ~]$ . /scratch/project_2008874/cristian/intel/oneapi/setvars.sh --include-intel-llvm

:: initializing oneAPI environment ...
   -bash: BASH_VERSION = 4.4.20(1)-release
   args: Using "$@" for setvars.sh arguments: --include-intel-llvm
:: advisor -- latest
:: ccl -- latest
:: compiler -- latest
:: dal -- latest
:: debugger -- latest
:: dev-utilities -- latest
:: dnnl -- latest
```

```
[steiners2@mahti-login15 ~]$ module load cuda
[steiners2@mahti-login15 ~]$ sycl-ls
[opencl:acc:0] Intel(R) FPGA Emulation Platform for OpenCL(TM), Intel(R) FPGA Emulation Device OpenCL 1.2  [2023.16.12.0.12_195853.xmain-hotfix]
[opencl:cpu:1] Intel(R) OpenCL, AMD EPYC 7402 24-Core Processor                OpenCL 3.0 (Build 0) [2023.16.12.0.12_195853.xmain-hotfix]
```

Where Code Executes

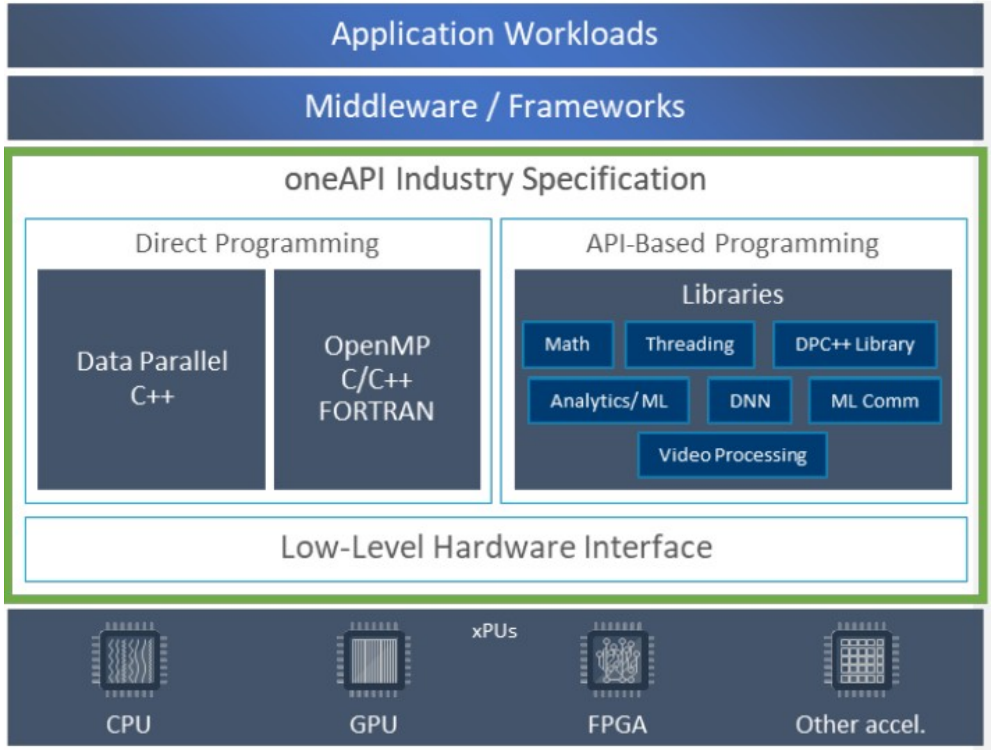
Programmers’ perspective: Three things to consider

1. Offload the code to device
2. Manage the transfer of Data
3. Implement Parallelism



Programmers' perspective: Three things to consider

1. Offload the code to device
2. Manage the transfer of Data
3. Implement Parallelism



```cpp
1 #include<iostream>
2 #include<sycl/sycl.hpp>
3 using namespace sycl;
4
5 const std::string secret
6 {
7 "Ifmmp-!xpsme\"\012J(n!tpssz-!Ebwf/!"
8 "J(n!bgsbje!J!dbo(u!ep!uibu/!.!IBM\01"
9 };
10
11 const auto sz=secret.size();
12
13 int main()
14 {
15     queue Q;
16     char* result = malloc_shared<char>(sz, Q);
17     std::memcpy(result, secret.data(), sz);
18
19     Q.parallel_for(sz, [=] (auto& i)
20     {
21         result[i] -= 1;
22     }).wait();
23
24     std::cout << result << "\n" ;
25     free(result, Q);
26     return 0;
27 }
```

SYCLs Hello World

1: Access to all SYCL constructs

3: Avoid having to write sycl::

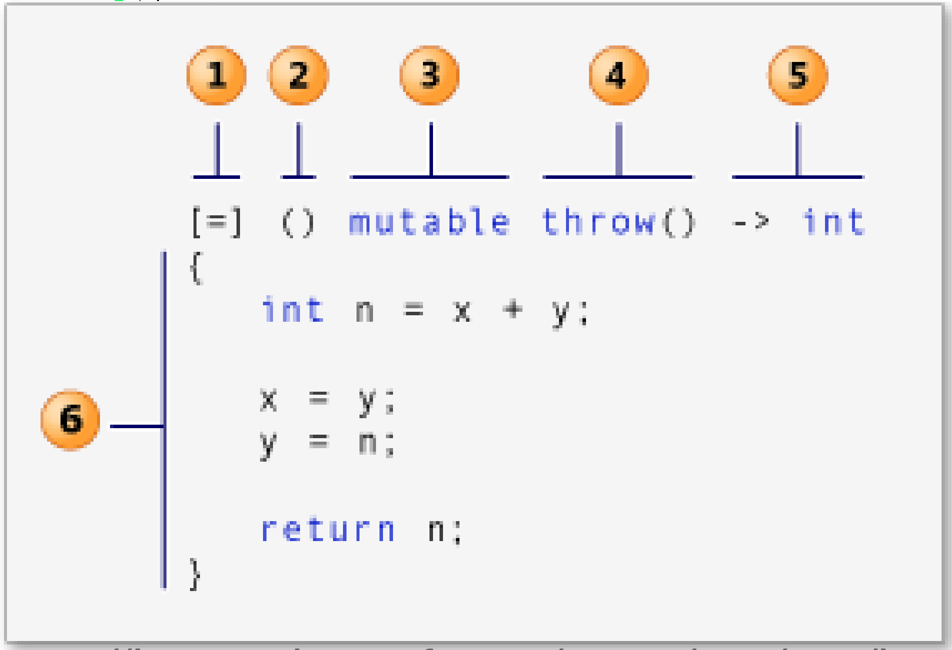
15: Establish queue for work requests to a particular device
16: create shared data

19: Enqueue work to the device

21: Only line that runs on the device

Lambda-functions … Lambdas

```
q.parallel_for(N, [=](auto i)
{
    a[i] -= 2;
});
```



1. capture clause
2. parameter list optional
3. mutable specification optional
4. exception-specification optional
5. trailing-return-type optional
6. lambda body

- [=] : capture by value
- [&] : capture by reference

https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp

Kernel Code

!

- **Run Asynchronously**
- **Limitation on what kind of C++ code**

```
q.parallel_for(N, [=](auto i)
{
	a[i] -= 2;
});
```

Kernel Code
Cannot use
these features

- Dynamic Polymorphism
- Dynamic memory allocations
- Static variables
- Function pointers
- Runtime Type Informatoion (RTTI)
- Exception Handling
- Recursion

SYCL fundamentals

Explain the SYCL fundamental classes

Use device selection to offload kernel workloads

Decide when to use basic parallel kernels and ND-Range kernels

Understand various ways to synchronize data between host and device with using buffer memory model

Write a complete SYCL program that offload computation to accelerator device

C++ with SYCL

- Enables programming for heterogenous hardware from different vendors.
- Single source that has host code and kernel code to offload to CPU, GPU, FPGA or other accelerator devices.
- Based on Open Standards C++ and Khronos* SYCL

Anatomy of a SYCL Application

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

int main() {
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), C(1024);
    {
      buffer bufA {A}, bufB {B}, bufC {C};
      queue q;
      q.submit([&](handler &h) {
          auto A = bufA.get_access(h, read_only);
          auto B = bufB.get_access(h, read_only);
          auto C = bufC.get_access(h, write_only);
          h.parallel_for(1024, [=](auto i){
              C[i] = A[i] + B[i];
          });
      });
    }
for (int i = 0; i < 1024; i++)
       std::cout << "C[" << i << "] = " << C[i] << std::endl;

}
```

Host code

Accelerator device code

Host code

Anatomy of a SYCL Application

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

int main() {
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), C(1024);
    {
      buffer bufA {A}, bufB {B}, bufC {C};
      queue q;
      q.submit([&](handler &h) {
          auto A = bufA.get_access(h, read_only);
          auto B = bufB.get_access(h, read_only);
          auto C = bufC.get_access(h, write_only);
          h.parallel_for(1024, [=](auto i){
              C[i] = A[i] + B[i];
          });
      });
    }
for (int i = 0; i < 1024; i++)
       std::cout << "C[" << i << "] = " << C[i] << std::endl;

}
```

Application scope

Command group scope

Device scope

Application scope

SYCL Basics

```
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), C(1024);
 {    buffer bufA {A}, bufB {B}, bufC {C};
      queue q;
      q.submit([&](handler &h)
      {
           auto A = bufA.get_access(h, read_only);
           auto B = bufB.get_access(h, read_only);
           auto C = bufC.get_access(h, write_only);
           h.parallel_for(1024, [=](auto i)
           {
                C[i] = A[i] + B[i];
           });
      });

  for (int i = 0; i < 1024; i++)
      std::cout << "C[" << i << "] = " << C[i] << std::endl;

 }
```

Buffers creation via host vectors/pointers

Buffers encapsulate data in a SYCL application

- Across both devices and host!

SYCL Basics

```
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), C(1024);
 {      buffer  bufA {A}, bufB {B}, bufC {C};
        queue  q;
        q.submit([&](handler  &h)
        {
             auto A = bufA.get_access(h, read_only);
             auto B = bufB.get_access(h, read_only);
             auto C = bufC.get_access(h, write_only);
             h.parallel_for(1024, [=](auto  i)
             {
                  C[i] = A[i] + B[i];
             });
        });


  for  (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;

  }
```

- A queue submits command groups to be executed by the SYCL runtime
- Queue is a mechanism where work is submitted to a device.

SYCL CLASSES

Where Code Executes

- Queues
- Device Selectors

QUEUES CONNECT US TO DEVICES

- We submit actions into queues to request computational work and data movement.

- Actions happen ASYNCHRONOUSLY

Device

- The **device** class represents the capabilities of the accelerators in a oneAPI system.
- The device class contains member functions for querying information about the device, which is useful for DPC++ programs where multiple devices are created.
- The function get_info gives information about the device:
  - Name, vendor, and version of the device
  - The local and global work item IDs
  - Width for built in types, clock frequency, cache width and sizes, online or offline

```
queue q;
device my_device = q.get_device();
std::cout << "Device: " << my_device.get_info<info::device::name>() << std::endl;
```

Device Selector

- The **device_selector** class enables the runtime selection of a particular device to execute kernels based upon user-provided heuristics.
- The following code sample shows use of the standard device selectors (default_selector, cpu_selector, gpu_selector…) and a derived **device_selector**

```
default_selector_v selector;
// host_selector_v selector;
// cpu_selector_v selector;
// gpu_selector_v selector;
queue q(selector);
std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
```

Queue

- A queue **submits command groups** to be executed by the SYCL runtime
- Queue is a mechanism where work is submitted to a device.
- A Queue map to one device and multiple queues can be mapped to the same device.

```
queue q;


q.submit( [&] (handler& h)
{
    // COMMAND GROUP CODE
});
```

The queue class

Actions are submitted to a queue for execution on a single device

- Always bound to a single device
  - Q1 → GPU1
  - Q2 → CPU
  - Q3 → FPGA
  - Q4 → GPU

- Several queues can point to the same device
  - Q1 → GPU1
  - Q2 → GPU1
  - Q3 → GPU1
  - Q4 → CPU

Choosing Where Device Kernels Run

Work is submitted to queues

- Each queue is associated with exactly one device (e.g. a specific GPU or FPGA)
- You can:
  - Decide which device a queue is associated with (if you want)
  - Have as many queues as desired for dispatching work in heterogeneous systems

Create queue targeting any device:	queue();

Create queue targeting a pre-configured classes of devices:	queue(cpu_selector_v);
	queue(gpu_selector_v);
	queue(ext::intel::fpga_selector_v);
	queue(accelerator_selector_v);
	queue(host_selector_v);

Create queue targeting specific device (custom criteria):	class custom_selector : public device_selector {
	  int operator()(…… **// Any logic you want!**
	…
	queue(custom_selector);

Always available

The queue class – Binding done at construction

```
class queue {
 public:
  // Create a queue associated with a default
  // (implementation chosen) device.
  queue(const property_list & = {});

  queue(const async_handler &, const property_list & = {});

  // Create a queue using a DeviceSelector.
  // A DeviceSelector is a callable that ranks
  // devices numerically. There are a few SYCL-defined
  // device selectors available such as
  // cpu_selector_v and gpu_selector_v.
  template <typename DeviceSelector>
  explicit queue(const DeviceSelector &deviceSelector,
                 const property_list &propList = {});

  // Create a queue associated with an explicit device to
  // which the program already holds a reference.
  queue(const device &, const property_list & = {});

  // Create a queue associated with a device in a specific
  // SYCL context. A device selector may be used in place
  // of a device.
  queue(const context &, const device &,
        const property_list & = {});
};
```

Default Device used here

The queue class – key member functions

a queue is bound to a single devices

```
class queue {
 public:
  // Submit a command group to this queue.
  // The command group may be a lambda expression or
  // function object. Returns an event reflecting the status
  // of the action performed in the command group.
  template <typename T>
  event submit(T);

  // Wait for all previously submitted actions to finish
  // executing.
  void wait();

  // Wait for all previously submitted actions to finish
  // executing. Pass asynchronous exceptions to an
  // async_handler function.
  void wait_and_throw();
};
```

Device selectors

Selector
cpu_selector_v
gpu_selector_v
ext::intel::fpga_selector_v
accelerator_selector_v
default_selector_v
or write a custom selector

Choosing Devices: Five use cases:

#	**Methods**	**Comments**
1	Anywhere (don’t care where)	Runtime chooses
2	Always on Host	Good for debugging
3	GPU or Accelerator	
4	Heterogeneous set of devices	
5	Specific Class of device	e.g. FPGA

Method #1, Binding a Queue to a Device When Any Device Will Do

Default Device used here Decided by the runtime

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

int main()
{
    // create queue on whatever default device that the
    // implementation chooses.
    queue Q;

    std::cout << "Selected device: " <<
    Q.get_device().get_info<info::device::name>() << "\n";

    return 0;
}
```

Method #1, Binding a Queue to a Device When Any Device Will Do

```
Sample Outputs (one line per run depending on system):
Selected device: NVIDIA GeForce RTX 3060
Selected device: AMD Radeon RX 5700 XT
Selected device: Intel(R) Data Center GPU Max 1100
Selected device: Intel(R) FPGA Emulation Device
Selected device: AMD Ryzen 5 3600 6-Core Processor
Selected device: Intel(R) UHD Graphics 770
Selected device: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
Selected device: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
many more possible… these are only examples
```

Method #2 Using Host Device, Development, Debugging and Deployment

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

int main()
{
    queue Q{ cpu_selector{} };

    std::cout << "Selected Device: " <<
    Q.get_device().get_info<info::device::name>() << "\n";
    std::cout << " ---->>>>>  Device Vendor: " <<
    Q.get_device().get_info<info::device::vendor>() << "\n";

    return 0;
}
```

Old notation
SYCL 2020
easier

Method #3 -Using a GPU or Accelerator (just change the selector to gpu_selector or accelerator_selector)

Method #2 Using Host Device, Development, Debugging and Deployment

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

int main()
{
    queue Q{ cpu_selector_v};
//    queue Q2{ host_selector_v};
    queue Q3{ default_selector_v};
    queue Q4{ gpu_selector_v};

    std::cout << "Selected Device: " <<
    Q.get_device().get_info<info::device::name>() << "\n";
    std::cout << " ---->>>>>  Device Vendor: " <<
    Q.get_device().get_info<info::device::vendor>() << "\n";

    std::cout << "Selected Device: " <<
    Q3.get_device().get_info<info::device::name>() << "\n";
    std::cout << " ---->>>>>  Device Vendor: " <<
    Q3.get_device().get_info<info::device::vendor>() << "\n";

    std::cout << "Selected Device: " <<
    Q4.get_device().get_info<info::device::name>() << "\n";
    std::cout << " ---->>>>>  Device Vendor: " <<
    Q4.get_device().get_info<info::device::vendor>() << "\n";
```

SYCL 2020 easier

Method #3 - Using a GPU or Accelerator just change the selector to gpu_selector_v or accelerator_selector_v

Method #4 Using Multiple Devices

```
#include <sycl/sycl.hpp>
#include <sycl//ext/intel/fpga_extensions.hpp>
#include <iostream>
using namespace sycl;

int main()
{
    queue gpu_q( gpu_selector_v );
    queue cpu_q( cpu_selector_v );
    queue fpga_q( ext::intel::fpga_selector_v );

    std::cout << "Selected Device1: " <<
    cpu_q.get_device().get_info<info::device::name>() << "\n";
    std::cout << "Selected Device2: " <<
    gpu_q.get_device().get_info<info::device::name>() << "\n";
    std::cout << "Selected Device3: " <<
    fpga_q.get_device().get_info<info::device::name>() << "\n";

    return 0;
}
```

Three Queues
Three Devices

Method #5: Custom (Very Specific) Device Selection → Skip

Control Device Selection via SYCL_DEVICE_FILTER

- Limits the choice of devices available to the runtime
- Syntax: SYCL_DEVICE_FILTER=backend:device_type:device_num, …
  - Backend: host, opencl, level_zero, cuda, hip, *
  - Device_type: host, cpu, gpu, acc, *
  - Device_num: unsigned integer
    - Enumeration index of devices from the sycl-ls utility
  - Each field is *optional*, so missing entry is regarded as ‘*’.
    - E.g., SYCL_DEVICE_FILTER=gpu SYCL_DEVICE_FILTER=*:gpu:*
  - Multiple triples can be specified separated by commas.

Control Device Selection via SYCL_DEVICE_FILTER

- Dual purposes
  - Users can specify their desired devices with the given triple(s).
  - SYCL only loads relevant plugins into runtime.

Control Device Selection via SYCL_DEVICE_FILTER

$ icpx -fsycl 02-Default-selector.cpp -o 02-Default-selector.x

$ SYCL_PI_TRACE=1 ./02-Default-selector.x

```
$ SYCL_PI_TRACE=1 ./02-Default-selector.x
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_opencl.so [ PluginVersion: 14.37.1 ]
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_cuda.so [ PluginVersion: 14.38.1 ]
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_unified_runtime.so [ PluginVersion: 14.37.1 ]
SYCL_PI_TRACE[all]: Requested device_type: info::device_type::automatic
SYCL_PI_TRACE[all]: Requested device_type: info::device_type::automatic
SYCL_PI_TRACE[all]: Requested device_type: info::device_type::automatic
SYCL_PI_TRACE[all]: Selected device: -> final score = 500
SYCL_PI_TRACE[all]:   platform: NVIDIA CUDA BACKEND
SYCL_PI_TRACE[all]:   device: NVIDIA GeForce RTX 3070 Laptop GPU
Selected device: NVIDIA GeForce RTX 3070 Laptop GPU
$
```

Control Device Selection via SYCL_DEVICE_FILTER

$ icpx -fsycl 02-Default-selector.cpp -o 02-Default-selector.x

$ SYCL_PI_TRACE=1 SYCL_DEVICE_FILTER=*:cpu ./02-Default-selector.x

```
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_opencl.so [ PluginVersion: 14.37.1 ]
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_cuda.so [ PluginVersion: 14.38.1 ]
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_unified_runtime.so [ PluginVersion: 14.37.1 ]
SYCL_PI_TRACE[all]: Requested device_type: info::device_type::automatic
SYCL_PI_TRACE[all]: Selected device: -> final score = 300
SYCL_PI_TRACE[all]:   platform: Intel(R) OpenCL
SYCL_PI_TRACE[all]:   device: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
Selected device: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
```

Examples ONEAPI_DEVICE_SELECTOR

ONEAPI_DEVICE_SELECTOR=

*opencl:**	Only the OpenCL devices are available
level_zero:gpu	Only GPU devices on the Level Zero platform are available.
"opencl:gpu;level_zero:gpu"	GPU devices from both Level Zero and OpenCL are available. Note that escaping (like quotation marks) will likely be needed when using semi- colon separated entries.
opencl:gpu,cpu	Only CPU and GPU devices on the OpenCL platform are available.
opencl:0	Only the device with index 0 on the OpenCL backend is available.
hip:0,2	Only devices with indices of 0 and 2 from the HIP backend are available.

Dispatching mechanism

Dispatching Code – Device Dispatch Mechanism

- So far. We’ve used

- queue::parallel_for()
- queue::single_task()

```
q.parallel_for(N, [=](auto i)
{
    a[i] -= 2;
});
```

- handler::single_task()
- handler::parallel_for()
- handler::parallel_for_work_group()

Kernel

- The kernel class encapsulates methods and data for executing code on the device when a command group is instantiated
- Kernel object is not explicitly constructed by the user
- Kernel object is constructed when a kernel dispatch function, such as parallel_for, is called

```
q.submit( [&] (handler& h)
{
  h.parallel_for(N, [=](auto i)
  {
    A[i] = B[i] + C[i]);
  });
});
```

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; });
});
```

Lambda name no longer required

SYCL

Buffer Simplification

Accessor simplification

parallel_for simplification

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

← *Simple and Less Verbose*

SYCL 2020

intel.

DPC++ language and runtime

- DPC++ language and runtime consists of a set of C++ classes, templates, and libraries
- **Application scope** and **command group scope** :
  - Code that executes on the host
  - The full capabilities of C++ are available at application and command group scope
- **Kernel scope**:
  - Code that executes on the device.
  - At kernel scope there are limitations in accepted C++

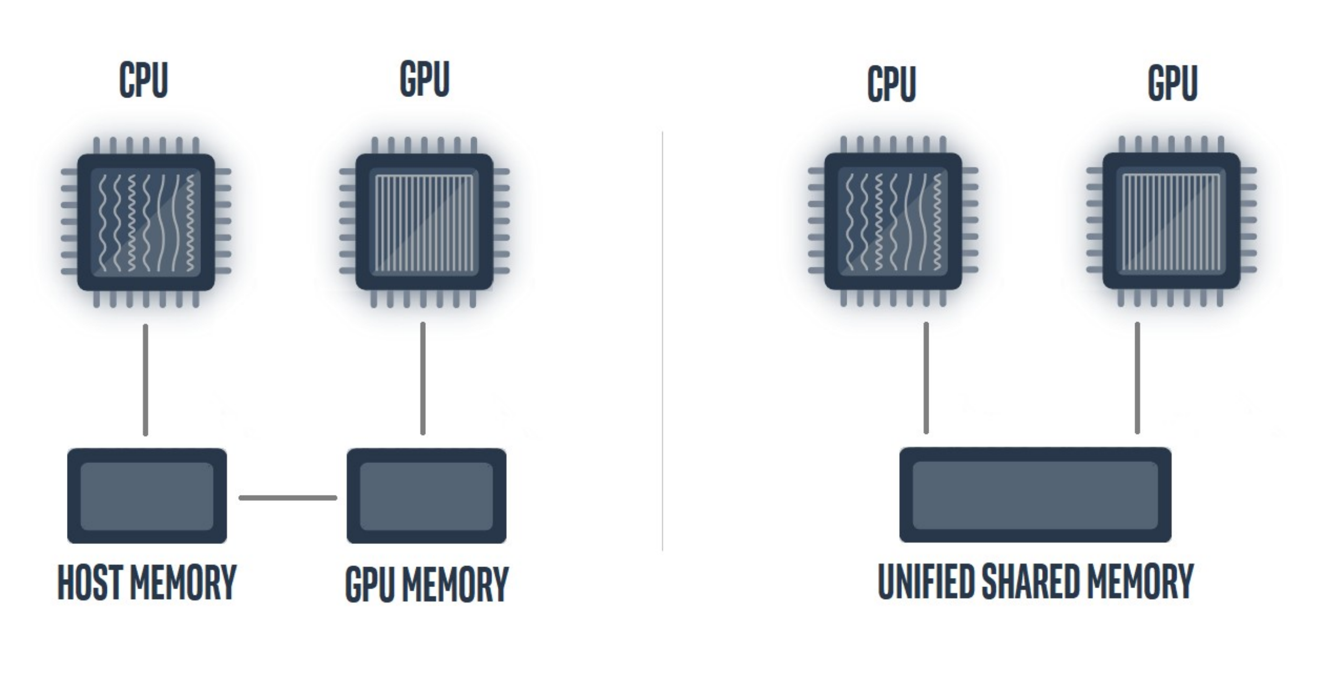
Actions

Work Type	Actions (handler class methods)	Summary
Device code execution	single_task	Execute a single instance of a device function.
	parallel_for	Multiple forms are available to launch device code with different combinations of work sizes.
	parallel_for_work_group	Launch a kernel using hierarchical parallelism, described in Chapter 4.
Explicit memory operation	copy	Copy data between locations specified by accessor, pointer, and/or shared_ptr. The copy occurs as part of the DAG, including dependence tracking.
	update_host	Trigger update of host data backing of a buffer object.
	fill	Initialize data in a buffer to a specified value.

J. Reinders et al., Data Parallel C++, download link

Developer View of USM

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



Overview

- Introduction
- Remainder of the lambda functions
- Compilation and run
- Queues and device selectors
- Dispatching mechnism
- Basic parallel kernels

Unified Shared Memory (USM)

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

Setup Unified Shared Memory

Host can initialize

Device can modify

Host has output

```
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);
```

Exercises

- SYCL Program Structure
- Read the instructions carefully, it is about compiling and env variables.
- The sycl I exercises can be done on the Intel Dev Cloud (IDC) and or on LUMI.

Getting Started on DevCloud

- qsub -I -l nodes=1:gpu:ppn=2 -d .
- sycl-ls (control devices via SYCL_DEVICE_FILTER)
- Compile and run a simple vecAdd code
- export SYCL_PI_TRACE=1
- export ONEAPI_DEVICE_SELECTOR=opencl:gpu

Useful Links

Open source projects

oneAPI Data Parallel C++ compiler: github.com/intel/llvm

Graphics Compute Runtime: Graphics Compiler: github.com/intel/compute-runtime github.com/intel/intel-graphics-compiler

SYCL 2020: tinyurl.com/sycl2020-spec

DPC++ Extensions: tinyurl.com/dpcpp-ext

Environment Variables: tinyurl.com/dpcpp-env-vars

DPC++ book: tinyurl.com/dpcpp-book

SYCL Academy github.com/codeplaysoftware/syclacademy/tree/main

Code samples:

github.com/intel/llvm/tree/sycl/sycl/test
github.com/intel/llvm/tree/sycl/sycl/test-e2e
github.com/oneapi-src/oneAPI-samples

Notices and Disclaimers