SYCL

Abhishek Bagusetty, Thomas Applencourt
Performance Engineering Group
Argonne Leadership Computing Facility

abagusetty@anl.gov

Agenda

Introduction: Heterogeneous Computing
Why SYCL ?
SYCL as Portable Programming Model

Execution Model
Memory Model
Best Practices

A few practical case studies

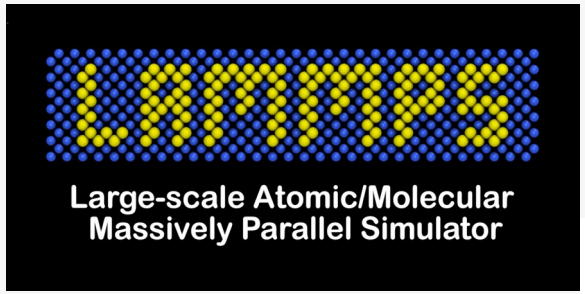
Pre-Exascale & Exascale Compute Architectures

Machine	GPU Models	DOE Facility	GPU	No of GPUs per node
	CUDA, SYCL	ALCF Polaris	Nvidia A100	4
	CUDA, SYCL	NERSC Perlmutter	Nvidia A100	4
	HIP, SYCL	OLCF Frontier	AMD MI 250x	4/8
	SYCL	ALCF Aurora	Intel Max Series	6/12
	HIP, (SYCL)	LLNL El Capitan	AMD MI300A	4

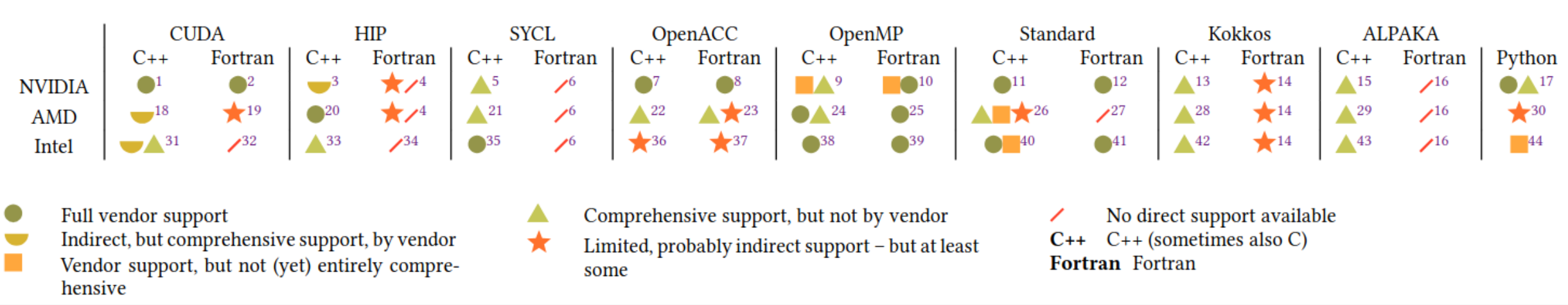
Khronos Group

The Khronos Group consortium develops and maintains some of the key open standards driving today's compute, graphics, and media innovation.

Applications/Libraries using SYCL



Hodgepodge of Programming Models



Many Cores, Many Models: GPU Programming Model vs. VendorCompatibility Overview

How to Join SYCL Resources

SYCL Slack: https://khr.io/slack

SYCL Resources: https://www.khronos.org/sycl/resources

Issues or Questions or Contribute: https://github.com/intel/llvm/issues

SYCL API Reference Documentation: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

What is SYCL ?

- SYCL is “not” a programming model but a “language specification”

- ✓ Heuristics looks similar to OpenCL-C bindings
- ✓ C++ single source (coexists host and device source code)
- ✓ Two distinct memory models (USM and/or Buffer)
- ✓ Asynchronous programming (overlaps device-compute, copy, host operations)
- ✓ Portability (functional and performance)
- ✓ Productivity

SYCL – Motivation

oneAPI Implementation of SYCL = C++ and SYCL* standard and extensions

Based on modern C++

✓ C++ productivity features and familiar constructs

Standards-based, cross-architecture

✓ Incorporates the SYCL standard for data parallelism and heterogeneous programming

SYCL* extensions

Productivity

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

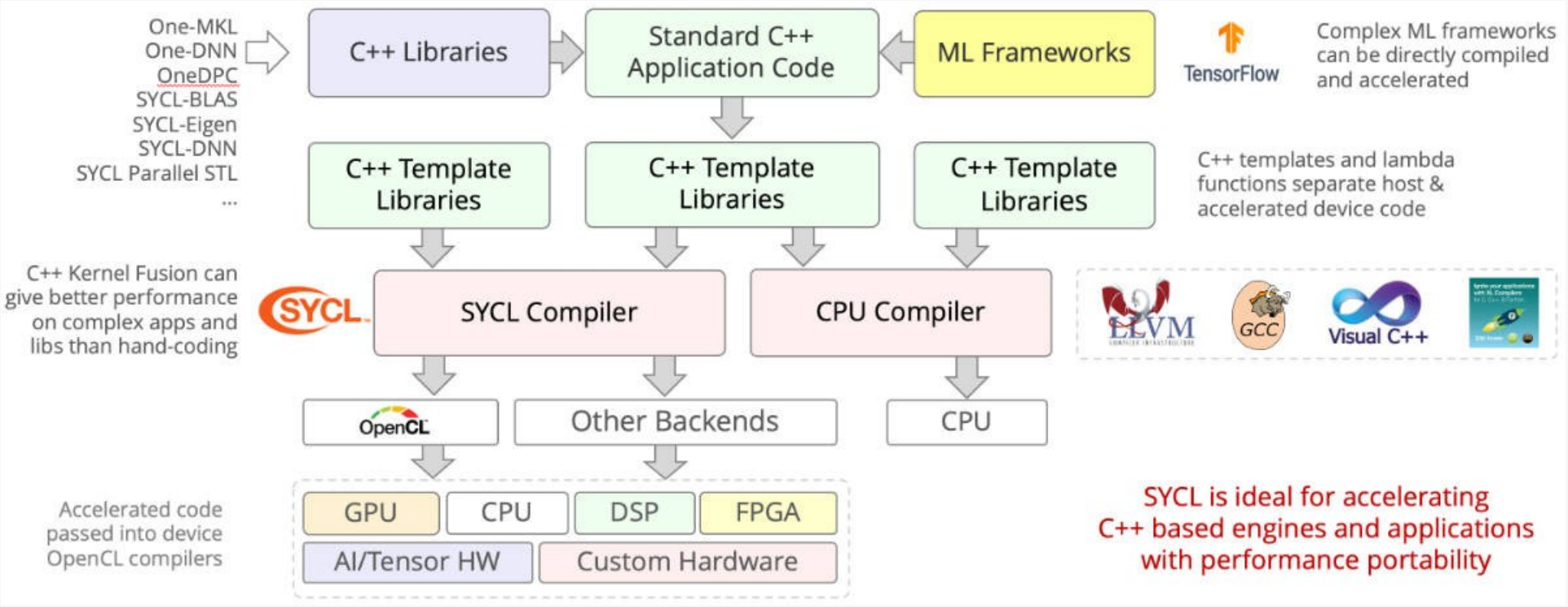
- Give programmers control over program execution
- Enable hardware-specific features

→ 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

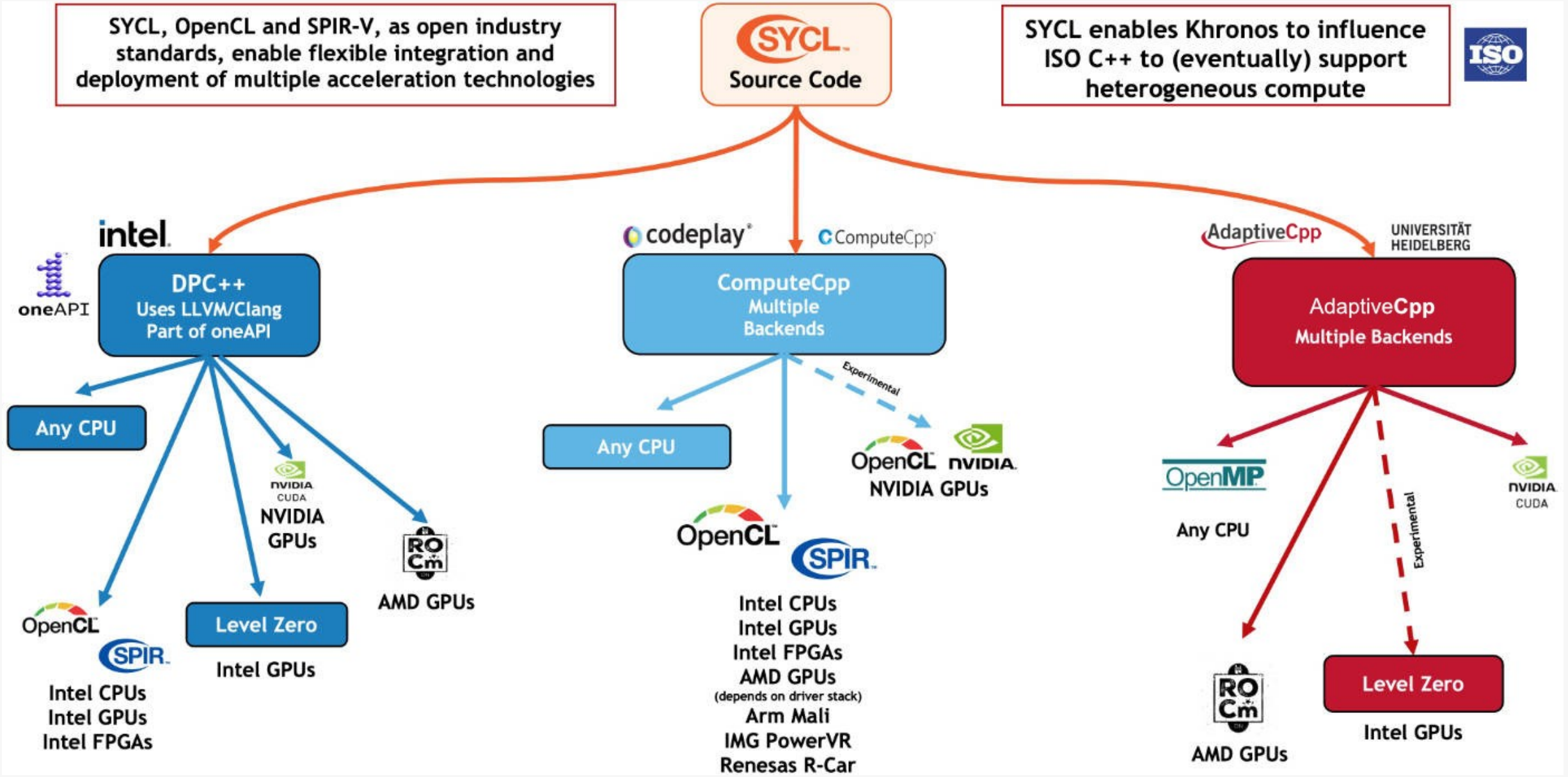
SYCL – A Portable Programming Model

A C++-based programming model for intra-node parallelism

- SYCL is a specification and “not” an implementation, currently compliant to C++17 ISO standards
- Cross-platform abstraction layer, heavily backed by industry
- Open-source, vendor agonistic
- Single-source model



SYCL – Compiler Players



Queues & Contexts

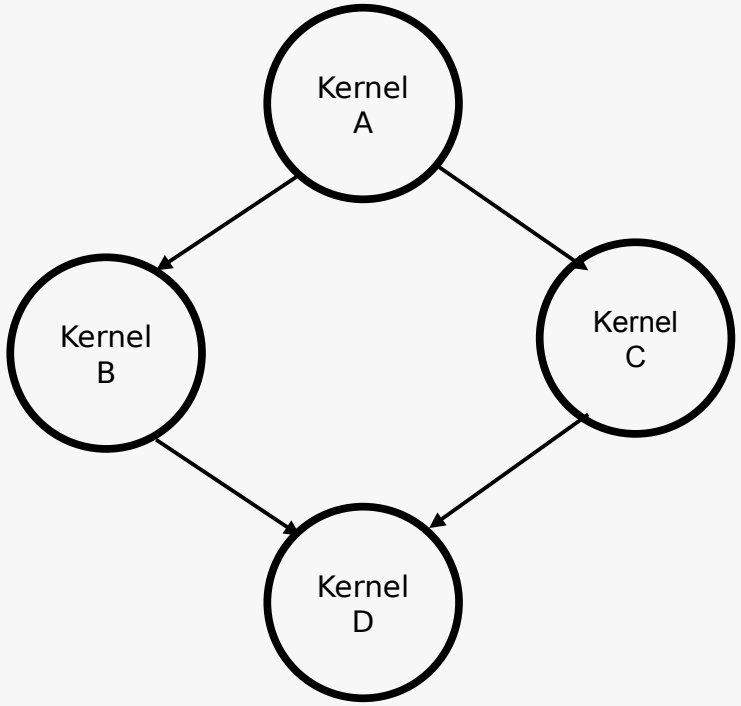
- “SYCL Queues” provide mechanism to **submit** work to a **device** or **sub-device**
- “SYCL Contexts” is well known to be over-looked

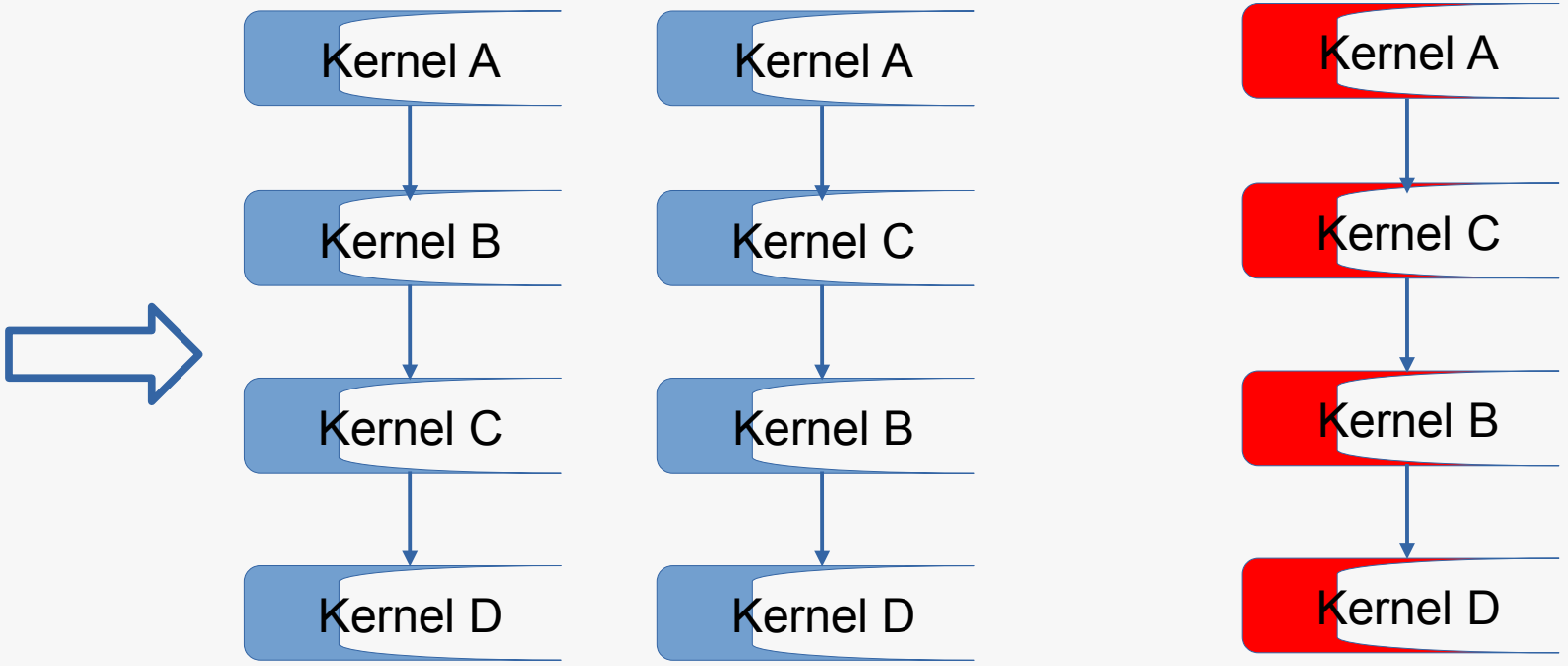
```
sycl::queue myQue;  // implicitly creates a SYCL context
```

- **Context**
  - Contexts are used for resources isolation and sharing
  - A SYCL context may consist of one or multiple devices
  - Both root-devices and sub-devices can be within single context (all from same SYCL platform)
  - Memory created can be shared only if their associated queue(s) are created using the same context

- **Queue (aka CUDA Stream)**
  - SYCL queue is always attached to a single device in a possibly multi-device context
    - ✓ Executes “**asynchronously**” from host code
    - ✓ SYCL queue can execute tasks enqueued in either “**in-order**” or “**out-of-order (default)**”
    - ✓ SYCL queue (in-order) is similar to CUDA stream (FIFO)

Queues (out-of-order vs in-order)





(OOO queue) This means commands are allowed to be overlapped and re-ordered or executed concurrently providing dependencies are honoured to ensure consistency.

(In-order) This mean commands must execute strictly in the order they were enqueued.

```
auto outOfOrderQueue = sycl::queue{gpu_selector_v};
auto inOrderQueue = sycl::queue{gpu_selector_v, sycl::property::queue::in_order{}};
```

Devices

- Devices are the target for acceleration offload

SYCL sub-devices ↔ CUDA Multi-Instance GPU (MIG) mode ↔ OpenCL sub-devices

- Explicit Scaling: Partitioning of a SYCL root device into multiple sub-devices based on NUMA boundary
  - ✓ SYCL queues are further created based on “sub-devices” (better performance)
- Implicit Scaling: SYCL unpartitioned/root device is directly used to create a SYCL queue

```
// EXPLICIT SCALING (better performance)


sycl::platform platform(sycl::gpu_selector{});
auto const& gpu_devices = platform.get_devices(sycl::info::device_type::gpu);
for (auto const& gpuDev : gpu_devices) {
  if(gpu_dev.get_info<sycl::info::device::partition_max_sub_devices>() > 0) {
    auto SubDev =
gpuDev.create_sub_devices<sycl::info::partition_property::partition_by_affinity_domain>(sycl::info::partition_affinity_domain::numa);

    for (auto const& tile : SubDev) {
      Que = sycl::queue(tile);
    }
  }
}

// IMPLICIT SCALING

sycl::platform platform(sycl::gpu_selector{});
auto const& gpu_devices = platform.get_devices(sycl::info::device_type::gpu);
for (auto const& gpuDev : gpu_devices) {
   Que = sycl::queue(gpuDev);
}
```

SYCL Events for Task-dependencies

- Performs book-keeping of tasks for data-transfer between host-device
- Similar to CUDA/HIP events

- SYCL runtime inherently has a DAG dependency paradigms
- Using SYCL dependency infrastructure provided via “SYCL Event”

- Task dependency between a communication-computation tasks

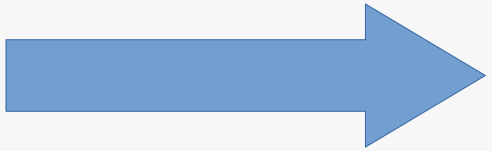
```
event memcpy(void* dest, const void* src, size_t numBytes);
event memcpy(void* dest, const void* src, size_t numBytes, event depEvent);
event memcpy(void* dest, const void* src, size_t numBytes,
             const std::vector<event>& depEvents);
```

Data-transfer showing “event” returns & dependencies

```
template <typename KernelName, int Dims, typename... Rest>
event parallel_for(nd_range<Dims> executionRange,
                   const std::vector<event>& depEvents, Rest&&... rest);
```

kernel-launch showing “event” returns & dependencies

Porting from CUDA to SYCL



Execution Model: CUDA vs SYCL

CUDA	SYCL
thread	work-item
block	work-group
grid	nd-range

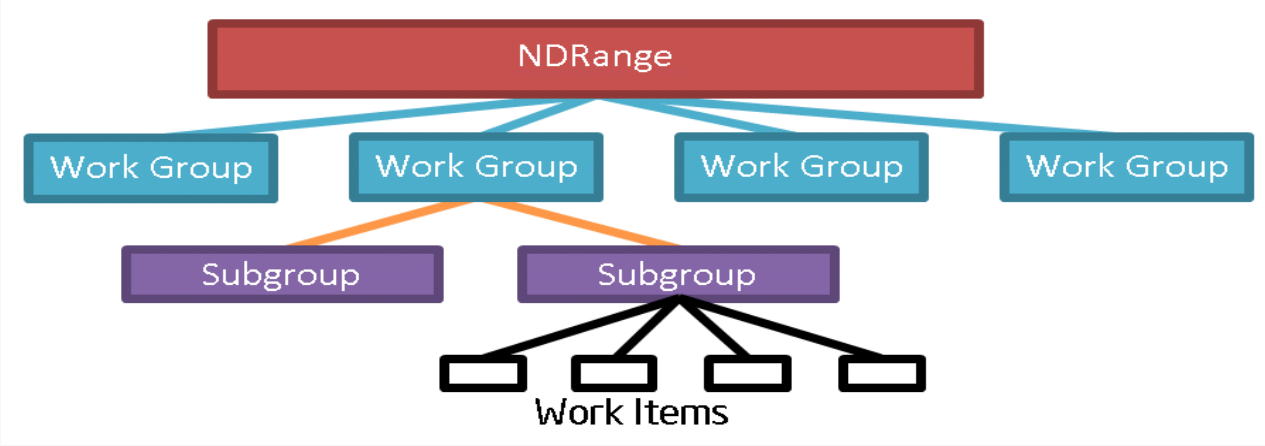
A **grid** is an array of thread blocks launched by a kernel.

An **nd-range** has three components

- global range (total work items)
- local range (work-items per work-group)
- number of work groups (total work groups)

CUDA - warp (vs) SYCL - sub groups

CUDA	SYCL
thread	work-item
warp	**sub-group**
block	work-group
grid	nd-range



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

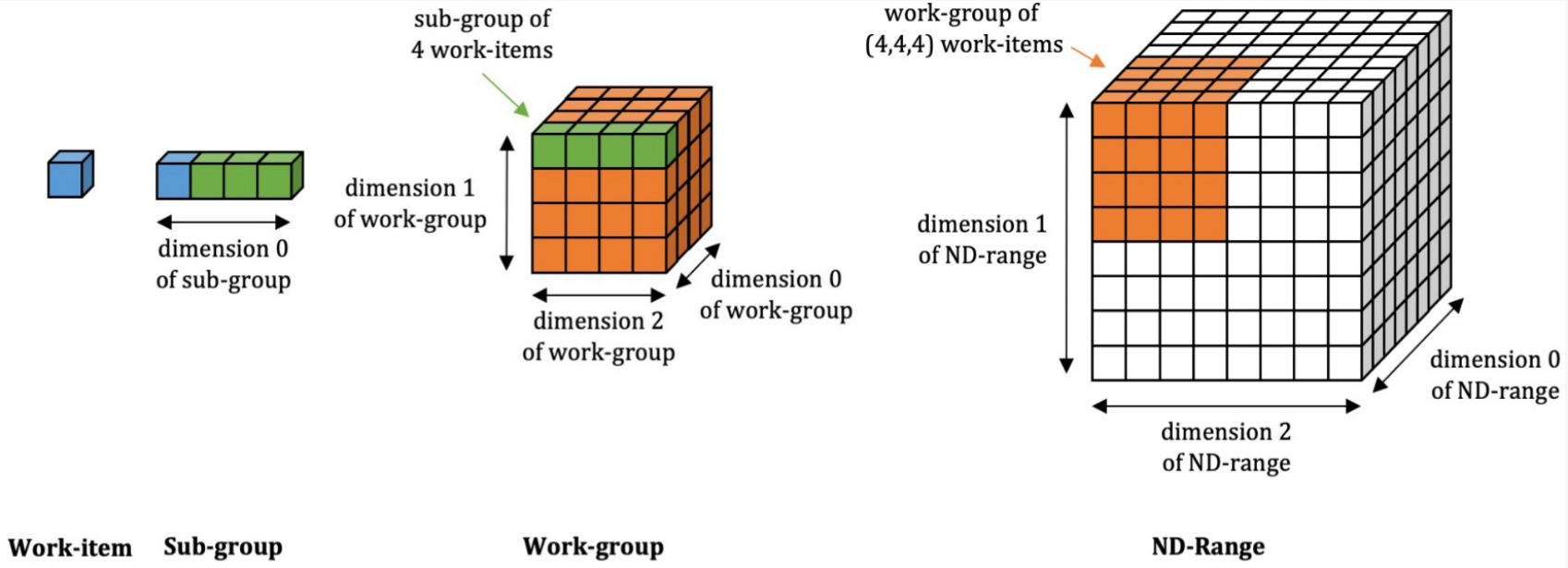
Leveraging sub-groups will help to map execution to low-level hardware and may help in achieving higher performance.

Why use SYCL - sub groups ?

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

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

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



Memory Model: CUDA vs SYCL

CUDA		SYCL	
Memory Type	**Scope**	**Memory Type**	**Scope**
Register memory	Thread	Private memory	Work-item
Shared memory	Block	Local memory	Work-group
Global memory	Grid (all threads)	Global memory	All work Items

Allocation Type	Initial Location	Accessible By		Migratable To	
`device`	`device`	`host`	No	`host`	No
		`device`	Yes	`device`	N/A
		Another `device`	Optional (P2P)	Another `device`	No
`host`	`host`	`host`	Yes	`host`	N/A
		Any `device`	Yes	`device`	No
`shared`	Unspecified	`host`	Yes	`host`	Yes
		`device`	Yes	`device`	Yes
		Another `device`	Optional	Another `device`	Optional

https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#table.USM.allocation.characteristics

Memory Model: Global Memory

CUDA		SYCL	
Memory Type	**Scope**	**Memory Type**	**Scope**
Register memory	Thread	Private memory	Work-item
Shared memory	Block	Local memory	Work-group
Global memory	**Grid (all threads)**	**Global memory**	**All work Items**

```
// allocating device memory

    float *A_dev;
    cudaMalloc((void **)&A_dev, array_size * sizeof(float));
```

```
// allocating device memory

    sycl::queue q(syckl::gpu_selector{});
    float *A_dev = sycl::malloc_device<float>(array_size, q);
```

- SYCL's Global/Device allocated memory is only **valid** on the **device**
- More importantly not accessible from host

Thread Indexing

CUDA	SYCL
gridDim.x/y/z	sycl::nd_item.get_group_range(2/1/0)
blockIdx.x/y/z	sycl::nd_item.get_group(2/1/0)
blockDim.x/y/z	sycl::nd_item.get_local_range().get(2/1/0)
threadIdx.x/y/z	sycl::nd_item.get_local_id(2/1/0)
warpSize	sycl::nd_item.get_sub_group().get_local_range().get(0)

- Always ensure the mapping of CUDA's (x/y/z) dimensions map to (2/1/0) in SYCL

Vector Addition: SYCL Buffer memory model

Host Code

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

void main() {
 using namespace sycl;
 float A[1024], B[1024], C[1024];
 {
  buffer<float, 1> bufA { A, range<1> {1024} };
  buffer<float, 1> bufB { B, range<1> {1024} };
  buffer<float, 1> bufC { C, range<1> {1024} };
```

Create SYCL buffers using host pointers.

Device Code

```
  queue myQueue;
  myQueue.submit([&](handler& cgh) {
   auto accA = bufA.get_access<access::read>(cgh);
   auto accB = bufB.get_access<access::read>(cgh);
   auto accC = bufC.get_access<access::write>(cgh);

   cgh.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
    accC[i] = accA[i] + accB[i];
   });
  }).wait();
 }
```

Create a queue to submit work to a GPU

Read/write accessors create dependencies if other kernels or host access buffers.

Vector addition device kernel

Host Code

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

Vector Addition: SYCL USM memory model

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

void main() {
 float A[1024], B[1024], C[1024];
// initialize A, B, C with values on host

 sycl::queue myQueue;

 float* devA = sycl::malloc_device<float>(1024, myQueue);
 float* devB = sycl::malloc_device<float>(1024, myQueue);
 float* devC = sycl::malloc_device<float>(1024, myQueue);

 myQueue.memcpy(devA, A, 1024 * sizeof(float));
 myQueue.memcpy(devB, B, 1024 * sizeof(float));

 myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
     devC[i] = devA[i] + devB[i];
   });

 myQueue.memcpy(C, devC, 1024 * sizeof(float));

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

Host Code

Device Code

Host Code

Step 1: Create SYCL queue to create GPU

Step 2: Allocate device memory

Step 3 (H2D): copy inputs "A" & "B" to GPU

Step 4 (Compute): Run the kernel on device

Step 5 (D2H): Copy result "devC" back to host

Vector Addition: SYCL USM memory model

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

void main() {
 float A[1024], B[1024], C[1024];
// initialize A, B, C with values on host

 sycl::queue myQueue;

 float* devA = sycl::malloc_device<float>(1024, myQueue);
 float* devB = sycl::malloc_device<float>(1024, myQueue);
 float* devC = sycl::malloc_device<float>(1024, myQueue);

 myQueue.memcpy(devA, A, 1024 * sizeof(float));
 myQueue.memcpy(devB, B, 1024 * sizeof(float));

 myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
     devC[i] = devA[i] + devB[i];
   });

 myQueue.memcpy(C, devC, 1024 * sizeof(float));

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

Host Code

Device Code

Host Code

SYCL queue (by-default) is out-of-order. (i.e., the execution starts when possible. Duty of programmer to assure correct dependencies

myQueue.wait(), wait for H2D to complete before starting the kernel

myQueue.wait(), wait for the kernel to finish

myQueue.wait(), wait for D2H to complete before printing "C"

Vector Addition: SYCL USM memory model

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

void main() {
 float A[1024], B[1024], C[1024];
// initialize A, B, C with values on host

 sycl::queue myQueue(sycl::property_list{sycl::property::queue::in_order{}});

 float* devA = sycl::malloc_device<float>(1024, myQueue);
 float* devB = sycl::malloc_device<float>(1024, myQueue);
 float* devC = sycl::malloc_device<float>(1024, myQueue);

 myQueue.memcpy(devA, A, 1024 * sizeof(float));
 myQueue.memcpy(devB, B, 1024 * sizeof(float));

 myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
     devC[i] = devA[i] + devB[i];
   });

 myQueue.memcpy(C, devC, 1024 * sizeof(float));

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

Host Code

Device Code

Host Code

SYCL queue (in-order) i.e., FIFO like cudaStream_t

myQueue.wait(), wait for D2H to complete before printing "C"

Access to ALCF Polaris

https://docs.alcf.anl.gov/polaris/getting-started/

To login:

```
ssh username@polaris.alcf.anl.gov
Password:
```

Modules to Load:

```
module use /soft/modulefiles/
module load oneapi/upstream cmake
```

To get a compute node:

```
qsub -I -A ATPESC2025 -q ATPESC -l select=1:system=polaris -l walltime=30:00 -l filesystems=home:grand:eagle
```

Access to ALCF Aurora

https://docs.alcf.anl.gov/aurora/getting-started-on-aurora/

To login:

```
ssh username@aurora.alcf.anl.gov
Password:
```

Modules to Load:

```
module restore
```

To get a compute node:

```
qsub -I -A ATPESC2025 -q ATPESC -l select=1,walltime=30:00,place=scatter,filesystems=home:flare
```

Support for CUDA and ROCm devices

Compiling With DPC++ for CUDA GPUs

The following command can be used to compile your code using DPC++ for CUDA backend:

```
clang++ -std=c++17 -fsycl -fsycl-targets=nvptx64-nvidia-cuda-sycldevice -Xsycl-target-backend
--cuda-gpu-arch=sm_80 simple-sycl-app.cpp -o simple-sycl-app-cuda
```

Compiling With DPC++ for ROCm GPUs*

The following command can be used to compile your code using DPC++ for HIP backend:

```
clang++ -fsycl -fsycl-targets=amdgcn-amd-amdhsa -Xsycl-target-backend --offload-arch=gfx9xx
simple-sycl-app.cpp -o simple-sycl-app-rocm
```

*Tested for ROCm 6.4.2, gfx90a for AMD250x on Frontier

oneAPI core elements

oneDPL

oneAPI Data Parallel C++ Library

A companion to the DPC++ Compiler for programming oneAPI devices with APIs from C++ standard library, Parallel STL, and extensions.

oneDNN

oneAPI Deep Neural Network Library

High performance implementations of primitives for deep learning frameworks.

oneCCL

oneAPI Collective Communications Library

Communication primitives for scaling deep learning frameworks across multiple devices.

Level Zero

oneAPI Level Zero

System interface for oneAPI languages and libraries.

oneDAL

oneAPI Data Analytics Library

Algorithms for accelerated data science.

oneTBB

oneAPI Threading Building Blocks

Library for adding thread-based parallelism to complex applications on multiprocessors.

oneMKL

oneAPI Math Kernel Library

High performance math routines for science, engineering, and financial applications.

Porting to SYCL using Math Libraries (oneMKL/oneMath, oneDPL)

oneMKL consists of three parts:

- The oneMKL specification - part of the “oneAPI” specification
  - ✓ oneMKL component provides mathematical routines for HPC applications, etc

- An open-source library implementing the MKL API - “oneMKL Interfaces”
- The UXL (Unified Acceleration) Foundation develops these specifications
- The specification is open-source, available on GitHub

- The original Intel optimized math routines - for clarity, Intel® MKL

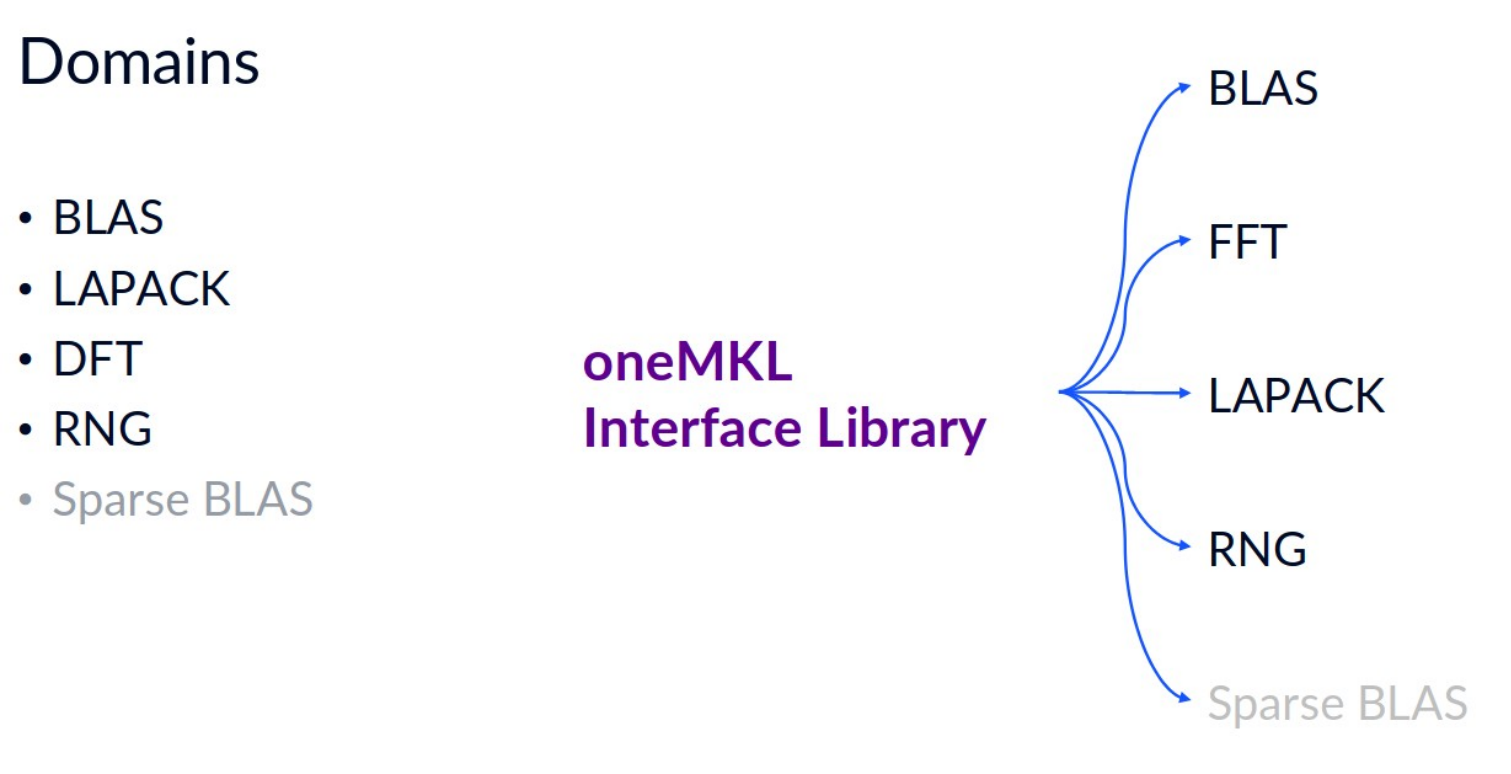
Porting to SYCL using Math Libraries (oneMKL Interface)

Implements the oneMKL specification, dispatching to other libraries underneath

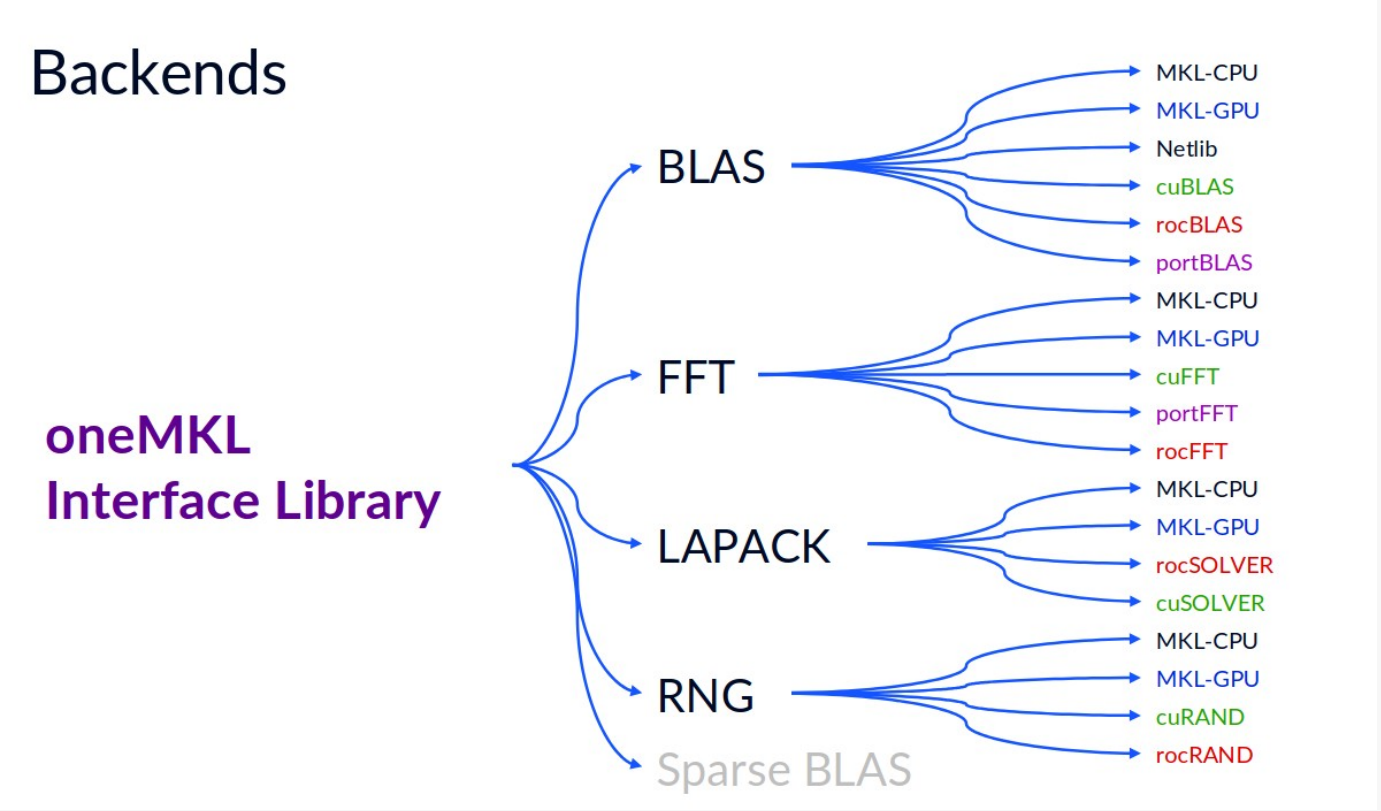
- ✓ Intel (Intel's MKL)
- ✓ Nvidia (cuBLAS, cuRAND, cuFFT etc.)
- ✓ AMD (rocBLAS, rocFFT etc.)
- ✓ And SYCL-supported devices ("generic" SYCL code)

- SYCL compiler implementations (supported): DPC++ and AdaptiveCpp
- (AdaptiveCpp support varies by backend but is being worked on)

Porting to SYCL using Math Libraries (oneMKL domains)



Porting to SYCL using Math Libraries (oneMath backends)



*SparseBLAS is currently a Work-In-Progress
Support for cuSPARSE - https://github.com/oneapi-src/oneMKL/pull/527
Support for rocSPARSE - https://github.com/oneapi-src/oneMKL/pull/544

Porting to SYCL using Math Libraries (oneMKL Usage)

Runtime Dispatch	Static Dispatch
$> clang++ -fsycl -I$ONEMKLROOT/include app.cpp $> clang++ -fsycl app.o -L$ONEMKLROOT/lib -lonemkl	$> clang++ -fsycl -I$ONEMKLROOT/include app.cpp $> clang++ -fsycl app.o -L$ONEMKLROOT/lib -lonemkl_blas_mklcpu -lonemkl_blas_cublas

`#include "oneapi/mkl.hpp"`
 	
`cpu_dev = sycl::device(sycl::cpu_selector());`
`gpu_dev = sycl::device(sycl::gpu_selector());`
 	
`sycl::queue cpu_queue(cpu_dev);`
`sycl::queue gpu_queue(gpu_dev);`
 	
`oneapi::mkl::blas::column_major::gemm(`**`cpu_queue`**`, transA, transB, m, ...);`
`oneapi::mkl::blas::column_major::gemm(`**`gpu_queue`**`, transA, transB, m, ...);`
	`#include "oneapi/mkl.hpp"`
	
	`cpu_dev = sycl::device(sycl::cpu_selector());`
	`gpu_dev = sycl::device(sycl::gpu_selector());`
	
	`sycl::queue cpu_queue(cpu_dev);`
	`sycl::queue gpu_queue(gpu_dev);`
	
	`oneapi::mkl::blas::column_major::gemm(`**`oneapi::mkl::backend_selector<oneapi::mkl::backend::mklcpu> {cpu_queue}`**`, transA, transB, m, ...);`
	`oneapi::mkl::blas::column_major::gemm(`**`oneapi::mkl::backend_selector<oneapi::mkl::backend::cublas> {gpu_queue}`**`, transA, transB, m, ...);`

• oneMKL can build with support for multiple vendors at once • oneMKL can automatically dispatch to the correct backend library • Backends are lazily dlopened (a bit of overhead)	• (a) Uses templated backend selector API, (b) Links with required backend wrapper libraries • Avoids overhead of dispatch tables by linking directly against backend libraries

Porting cuBLAS API to oneMKL API

- ✓ Look out for the “row/column” major data-layout
- ✓ cublas API are column major by-default,
- ✓ oneMKL provides APIs for both row-major and column major layout
- ✓ When porting to oneMKL BLAS API:
  `oneapi::mkl::blas::column_major::gemm<>`

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float                 *C, int ldc)
```

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm(sycl::queue &queue,
                     onemkl::transpose transa, onemkl::transpose transb,
                     std::int64_t m, std::int64_t n, std::int64_t k,
                     Ts alpha,
                     const Ta    *a, std::int64_t lda,
                     const Tb    *b, std::int64_t ldb,
                     Ts beta,
                     Tc          *c, std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Porting cuBLAS API to oneMKL API

- `cublasHandle_t` - opaque structure holding the cuBLAS library context
- Before using cuBLAS API, one must initialize `cublasHandle_t`
- Where as in oneMKL, no such handle exists. SYCL queue is sufficient to dispatch work

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float                 *C, int ldc)
```

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm(sycl::queue &queue,
                     onemkl::transpose transa, onemkl::transpose transb,
                     std::int64_t m, std::int64_t n, std::int64_t k,
                     Ts alpha,
                     const Ta    *a, std::int64_t lda,
                     const Tb    *b, std::int64_t ldb,
                     Ts beta,
                     Tc          *c, std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Porting cuBLAS API to oneMKL API

Tricky things:

- ✓ cuBLAS - “alpha” & “beta” parameters are pointers
- ✓ oneMKL - “alpha” & “beta” parameters are scalars

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float                 *C, int ldc)
```

$$C \leftarrow alpha * op(A) * op(B) + beta * C$$

where:

- op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H
- alpha and beta are scalars
- A, B and C are matrices
- op(A) is m x k matrix
- op(B) is k x n matrix
- C is m x n matrix

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm(sycl::queue &queue,
                     onemkl::transpose transa, onemkl::transpose transb,
                     std::int64_t m, std::int64_t n, std::int64_t k,
                     Ts alpha,
                     const Ta    *a, std::int64_t lda,
                     const Tb    *b, std::int64_t ldb,
                     Ts beta,
                     Tc          *c, std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Porting cuBLAS API to oneMKL API

Return type:

- ✓ cuBLAS – `cublasStatus_t`, Associates with the error-type
- ✓ oneMKL – `sycl::event`, Associates with the asynchronous progress

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float                 *C, int ldc)
```

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm(sycl::queue &queue,
                     onemkl::transpose transa, onemkl::transpose transb,
                     std::int64_t m, std::int64_t n, std::int64_t k,
                     Ts alpha,
                     const Ta    *a, std::int64_t lda,
                     const Tb    *b, std::int64_t ldb,
                     Ts beta,
                     Tc          *c, std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Porting to SYCL using Math Libraries (How to use oneMKL)

BLAS

```
sycl::queue syclQueue;

// Your data needs to be accessible on the GPU.
auto dev_A = sycl::malloc_device<float>(sizeA, syclQueue);

// … allocate memory, give it relevant values.
// Its like the BLAS API, but taking a “SYCL queue” argument. The USM API
returns an event.
gemm_done = oneapi::mkl::blas::column_major::gemm(syclQueue,
                                                  transA, transB, m, n, k,
                                                  alpha, dev_A, ldA, dev_B,
                                                  LdB, beta, dev_C, ldC);


// Wait for the work to finish.
gemm_done.wait_and_throw();
```

Porting to SYCL using Math Libraries (How to use oneMKL)

Random Number Generation

```
sycl::queue syclQueue;


// Your data needs to be accessible on the GPU.
auto dev_A = sycl::malloc_device<float>(sizeA, syclQueue);

// A random number generator is linked to a sycl::queue
rng::default_engine engine(syclQueue, seed);
rng::uniform<float> distribution(low, high);

// Use the state we generated earlier.
auto rng_out = rng::generate(distribution, engine, sizeA, dev_A);

// Wait for the work to finish.
rng_out.wait_and_throw();
```

Porting to SYCL using Math Libraries (How to use oneMKL)

FFT

```
sycl::queue syclQueue;

// Your data needs to be accessible on the GPU.
auto dev_A = sycl::malloc_device<float>(sizeA, syclQueue);

/ A descriptor describes the DFT you want…
dft::descriptor<dft::precision::SINGLE, dft::domain::REAL> desc(N);
desc.set_value(dft::config_param::PLACEMENT, dft::config_value::INPLACE);
// Once set, it is committed on for the chosen queue.
desc.commit(syclQueue);
// Compute the DFTs…
auto fft_out = dft::compute_forward(desc, dev_A);

// Wait for the work to finish.
fft_out.wait_and_throw();
```

Porting to SYCL using Math Libraries (CMake Magic)

oneMKL is already installed, then find oneMKL Dependency via CMake for a project

```
find_packge(oneMKL REQURIED)

// Link via runtime dispatch
target_link_library(mytarget PRIVATE MKL::onemkl)

// (or) Link against specific domain & backend
target_link_library(mytarget PRIVATE
                MKL::onemkl_<domain>_<backend>)
```

oneMKL is “not” available, Build and automatically link via FetchContent in CMake

```
include(FetchContent)
set(BUILD_FUNCTIONAL_TESTS OFF)
set(BUILD_EXAMPLES OFF)
set(ENABLE_<BACKEND_NAME>_BACKEND ON)
FetchContent_Declare(
       onemkl_interface_library
       GIT_REPOSITORY https://github.com/oneapi-src/oneMKL.git
       GIT_TAG develop
)
FetchContent_MakeAvailable(onemkl_interface_library)
target_link_libraries(myTarget PRIVATE onemkl)
// (or) for a specific backend
target_link_libraries(myTarget PRIVATE onemkl_<domain>_<backend>)
```

Porting to SYCL using Math Libraries (Install oneMKL Interfaces)

```
git clone https://github.com/oneapi-src/oneMKL.git
cd oneMKL
mkdir -p build-cuda && cd build-cuda
cmake .. -DCMAKE_CXX_COMPILER=icpx/clang++ \
         -DENABLE_CUBLAS_BACKEND=True \
         -DENABLE_CUSOLVER_BACKEND=True \
         -DENABLE_CURAND_BACKEND=True \
         -DENABLE_MKLCPU_BACKEND=False \
         -DENABLE_MKLGPU_BACKEND=False \
         -DBUILD_FUNCTIONAL_TESTS=False -DBUILD_EXAMPLES=False \
         -DCMAKE_INSTALL_PREFIX=<where-to-install>

make -j; make install
```

It's pretty easy (for instance oneMKL for Nvidia GPUs)

- ✓ Enable the backends you want (CUBLAS, CUSOLVER or CURAND) or (ROCBLAS, ROSOLVER or ROCRAND)
- ✓ Disable building functional-tests and examples

Don`t forget to add the install-directory <where-to-install> to the system path

```
export PATH=<where-to-install>/bin:$PATH
export CPATH=<where-to-install>/include:$CPATH
export LD_LIBRARY_PATH=<where-to-install>/lib:$LD_LIBRARY_PATH
```

Porting to SYCL using Math Libraries (Hands-on)

Objective: Learn oneMKL GEMM API, how to compile and run

https://github.com/codeplaysoftware/syclacademy/tree/main/Code_Exercises/oneMath_gemm

Login to compute node on ALCF Polaris

```
module use /soft/modulefiles/
module load oneapi/release
```

Source

```
git clone -b main --depth=1 https://github.com/codeplaysoftware/syclacademy.git
cd syclacademy/Code_Exercises/oneMath_gemm
```

Compile

```
export MKLROOT=/soft/compilers/oneapi/release/onemath

icpx -std=c++17 -O3 -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_80 -I${MKLROOT}/include -L${MKLROOT}/lib -lonemkl ./solution_onemath_usm_gemm.cpp -o solution_onemath_usm_gemm.out
```

oneDPL: Accelerated C++ on Heterogeneous System

- High productivity and portable performance for heterogeneous computing – CPUs, GPUs, and FPGAs
- APIs based on standards and familiar extensions – C++ STL, SYCL, Boost.Compute
- Optimized C++ standard algorithms implemented on top of SYCL, OpenMP, oneTBB
- Interoperable with DPC++ and other oneAPI libraries
- Integrated with Intel® DPC++ Compatibility Tool to simplify migration of CUDA* applications using Thrust* API to DPC++ code

✓ C++17 is the minimal supported version of the C++ standard
✓ Header names start with oneapi/dpl:

```
#include <oneapi/dpl/algorithm>
```

✓ All functionality is provided in `namespace oneapi::dpl`
short alias: `namespace dpl = oneapi::dpl;`

oneDPL: Accelerated C++ on Heterogeneous System

Obvious differences in the inclusion of headers

```
#include <thrust/transform_reduce.h>

template<typename InputIterator, typename UnaryFunction, typename OutputType, typename
BinaryFunction>
OutputType thrust::transform_reduce(InputIterator first,
                                    InputIterator last,
                                    UnaryFunction unary_op,
                                    OutputType init,
                                    BinaryFunction binary_op)
```

From standard C++ Algorithms library

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>

template< class ExecutionPolicy, class ForwardIt, class T,class BinaryReductionOp, class
UnaryTransformOp >
T transform_reduce( ExecutionPolicy&& policy,
                    ForwardIt first,
                    ForwardIt last,
                    T init,
                    BinaryReductionOp reduce,
                    UnaryTransformOp transform );
```

oneDPL: Accelerated C++ on Heterogeneous System

Obvious differences in the inclusion of headers

```
#include <thrust/transform_reduce.h>

template<typename InputIterator, typename UnaryFunction, typename OutputType, typename
BinaryFunction>
OutputType thrust::transform_reduce(InputIterator first,
                                    InputIterator last,
                                    UnaryFunction unary_op,
                                    OutputType init,
                                    BinaryFunction binary_op)
```

From standard C++ Algorithms library

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>

template< class ExecutionPolicy, class ForwardIt, class T,class BinaryReductionOp, class
UnaryTransformOp >
T transform_reduce( ExecutionPolicy&& policy,
                    ForwardIt first,
                    ForwardIt last,
                    T init,
                    BinaryReductionOp reduce,
                    UnaryTransformOp transform );
```

oneDPL: Transform reduce

Obvious differences in the inclusion of headers

```
#include <thrust/transform_reduce.h>

template<typename InputIterator, typename UnaryFunction, typename OutputType, typename
BinaryFunction>
OutputType thrust::transform_reduce(InputIterator first,
                                    InputIterator last,
                                    UnaryFunction unary_op,
                                    OutputType init,
                                    BinaryFunction binary_op)
```

`ExecutionPolicy` arg, is an additional argument that enables parallelism on CPU or GPU

From standard C++ Algorithms library

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>

template< class ExecutionPolicy, class ForwardIt, class T,class BinaryReductionOp, class
UnaryTransformOp >
T transform_reduce( ExecutionPolicy&& policy,
                    ForwardIt first,
                    ForwardIt last,
                    T init,
                    BinaryReductionOp reduce,
                    UnaryTransformOp transform );
```

oneDPL: Transform reduce

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>

template< class ExecutionPolicy, class ForwardIt, class T,class BinaryReductionOp,
class UnaryTransformOp >
T transform_reduce( ExecutionPolicy&& policy,
                    ForwardIt first,
                    ForwardIt last,
                    T init,
                    BinaryReductionOp reduce,
                    UnaryTransformOp transform );
```

1. oneDPL adopts uses the standard C++ Algorithms library

2. Most of the thrust algorithms maps to C++ parallel Algorithms library

3. How is oneDPL portable ?
   - SYCL kernels are used underneath the oneDPL algorithms, that makes oneDPL portable to other vendors

Portability of oneMKL for Intel, Nvidia, AMD GPUs

Open-source
https://github.com/oneapi-src/oneMKL

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

#include <oneapi/mkl.hpp>  // oneMKL header

////////////////////////////////////////////////////////////////////////////////

int main() {
......

  sycl::queue q(sycl::property_list{sycl::property::queue::in_order{}});
  std::cout << "Device: " << q.get_device().get_info<sycl::info::device::name>() << std::endl << std::endl;

  // Allocate device-arrays
  double* A_dev              = sycl::malloc_device<double>(M*N, q);
  double* B_dev              = sycl::malloc_device<double>(N*P, q);
  double* C_dev_onemkl = sycl::malloc_device<double>(M*P, q);

  // Copy inputs from host to device
  q.memcpy(A_dev, A, (M*N) * sizeof(double));
  q.memcpy(B_dev, B, (N*P) * sizeof(double));

  auto gemm_event = oneapi::mkl::blas::column_major::gemm(q, transB, transA,
                                                          n, m, k,
                                                          alpha,
                                                          B_dev, ldB,
                                                          A_dev, ldA,
                                                          beta,
                                                          C_dev_onemkl, ldC);

  gemm_event.wait();

......
}
```

Standard SYCL and oneMKL headers

Set up SYCL device & print name

Allocate “device” memory for matrices

Copy inputs from host to device

Call to oneMKL GEMM API

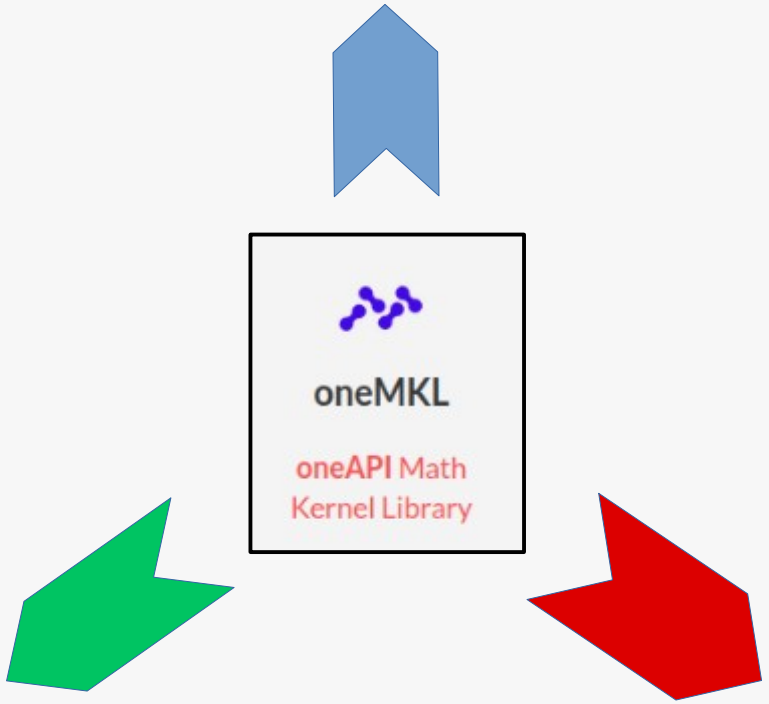
Synchronize for the results to finish

Full source at: https://docs.alcf.anl.gov/polaris/programming-models/sycl-polaris/

Portability of oneMKL for Intel, Nvidia, AMD GPUs

```
clang++ -std=c++17 -O3 -qmkl -L$MKLROOT/lib -lonemkl sycl_onemkl_gemm.cpp -o
sycl_onemkl_gemm.out
```

Compile for Intel PVC on ALCF Aurora



Compile for Nvidia A100 on ALCF Polaris

```
clang++ -std=c++17 -O3 -fsycl -fsycl-targets=nvptx64-
nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_80
-L$MKLROOT/lib -lonemkl sycl_onemkl_gemm.cpp -o
sycl_onemkl_gemm.out
```

Compile for AMD MI250X on OLCF Frontier

```
clang++ -std=c++17 -O3 -fsycl -Xsycl-target-
backend=amdgcn-amd-amdhsa --offload-arch=gfx90a -
L$MKLROOT/lib -lonemkl sycl_onemkl_gemm.cpp -o
sycl_onemkl_gemm.out
```

oneMKL for Nvidia A100 & AMD MI250X

```
abagusetty@x3004c0s31b1n0 ~ $ export CUBLAS_LOGINFO_DBG=1; export CUBLAS_LOGDEST_DBG=stdout
abagusetty@x3004c0s31b1n0 ~ $ ./sycl_onemkl_gemm_v1.out
Problem size: c(600,2400) = a(600,1200) * b(1200,2400)
Device: NVIDIA A100-SXM4-40GB

I! cuBLAS (v11.4) function cublasStatus_t cublasCreate_v2(cublasContext**) called:
i!  handle: type=cublasHandle_t; val=POINTER (IN HEX:0x0x14a709ed7a38)
I!  ....


I! cuBLAS (v11.4) function cublasStatus_t cublasDgemm_v2(cublasHandle_t, cublasOperation_t, cublasOperation_t, int, int, int, const double*, const
double*, int, const double*, int, const double*, double*, int) called:

I!  ....

oneMKL: Test PASSED
```

```
abagusetty@login04 /lustre/orion/gen243/scratch/abagusetty $ ROCBLAS_LAYER=4 ./sycl_onemkl_gemm.out
Problem size: c(600,2400) = a(600,1200) * b(1200,2400)
Device: gfx90a:sramecc+:xnack-

oneMKL: Test PASSED
- { rocblas_function: "rocblas_dgemm", atomics_mode: atomics_allowed, transA: 'N', transB: 'N', M: 2400, N: 600, K: 1200,
alpha: 1.0, lda: 2400, ldb: 1200, beta: 0.0, ldc: 2400, call_count: 1 }
```

- Performance: oneMKL uses cuBLAS & rocBLAS APIs underneath ensuring performance
- Standard rocBLAS/cuBLAS logging tools also work with oneMKL APIs

Full source at: https://docs.alcf.anl.gov/polaris/programming-models/sycl-polaris/

Portability of oneDPL for Intel, Nvidia, AMD GPUs

Open-source https://github.com/oneapi-src/oneDPL

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <sycl/sycl.hpp>
#include <cassert>

int main()
{
  sycl::queue q(sycl::gpu_selector_v);
  std::cout << "Device: " <<
q.get_device().get_info<sycl::info::device::name>() << std::endl <<
std::endl;
  int data_host[4] = {3, 7, 2, 5};
  int* data   = sycl::malloc_device<int>(4, q);       // inputs
  int* data_y = sycl::malloc_device<int>(4, q);       // outputs
  q.memcpy(data, data_host, 4*sizeof(int)).wait();

  // subtracts 10 to all values in data
  std::transform( oneapi::dpl::execution::make_device_policy(q),
                  data, data + 4,       /* first1, last1 */
                  data_y,               /* output */
                  [](const auto& x){return x-10;} );  /* unary op */

  // Results:
  int result_ref[4] = {-7, -3, -8, -5};
  int* result_host = new int[4]; // copy back results to host for
comparing
  q.memcpy(result_host, data_y, 4*sizeof(int));
  for( int i=0; i<4; i++) {
    assert( result_ref[i] == result_host[i] );
  }

  std::cout << “oneDPL: Test PASSED\n”;
  return 0;
}
```

Standard SYCL and oneDPL headers

Set up SYCL device & inputs

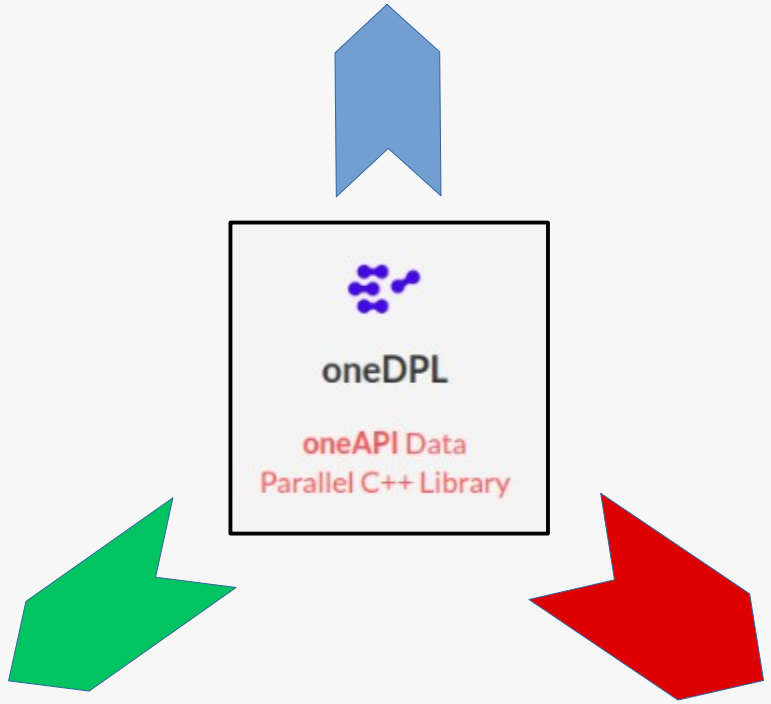
Calling std::transform from ISO C++ for GPU

Synchronize for the results to verify

Portability of oneDPL for Intel, Nvidia, AMD GPUs

```
clang++ -std=c++17 -O3 sycl_onedpl.cpp -o sycl_onedpl.out
```

Compile for Intel PVC on ALCF Aurora



Compile for Nvidia A100 on ALCF Polaris

```
clang++ -std=c++17 -O3 -fsycl -fsycl-targets=nvptx64-
nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_80
sycl_onedpl.cpp -o sycl_onedpl.out
```

Compile for AMD MI250X on OLCF Frontier

```
clang++ -std=c++17 -O3 -fsycl -Xsycl-target-
backend=amdgcn-amd-amdhsa --offload-arch=gfx90a
sycl_onedpl.cpp -o sycl_onedpl.out
```

oneDPL for Nvidia A100 & AMD MI250X

```
abagusetty@x3004c0s31b1n0 ~ $ ./sycl_onedpl.out
Device: NVIDIA A100-SXM4-40GB
oneDPL: Test PASSED
```

```
abagusetty@login04 /lustre/orion/gen243/scratch/abagusetty $ ./sycl_onedpl.out
Device: gfx90a:sramecc+:xnack-
oneDPL: Test PASSED
```

- Performance: oneDPL depends on the underlying portable SYCL kernels for Nvidia & AMD hardware

Full source at: https://docs.alcf.anl.gov/polaris/programming-models/sycl-polaris/

Build Your Own Compiler

- ✓ Build llvm-based SYCL compiler
- ✓ SYCL compiler for Nvidia/AMD hardware
- ✓ Nvidia A100 – NERSC Perlmutter / ALCF Polaris
- ✓ AMD MI250x – OLCF Frontier

Build Your Own Compiler (~60 mins, optional)

Get the source code: (takes a while)

```
git clone -b sycl --depth 1 https://github.com/intel/llvm.git
```

Build & Install: (takes a while too)

```
module load cudatoolkit
module list
cd llvm

CUDA_LIB_PATH=/soft/compilers/cudatoolkit/cuda-11.8.0/lib64/stubs python3 $PWD/buildbot/configure.py --cmake-gen "Unix Makefiles" --
cuda -cmake-opt="-DCUDA_TOOLKIT_ROOT_DIR=/soft/compilers/cudatoolkit/cuda-11.8.0"
--cmake-opt="-DSYCL_LIBDEVICE_GCC_TOOLCHAIN=/opt/cray/pe/gcc/11.2.0/snos"
--cmake-opt="-DCMAKE_C_COMPILER=/opt/cray/pe/gcc/11.2.0/snos/bin/gcc"
--cmake-opt="-DCMAKE_CXX_COMPILER=/opt/cray/pe/gcc/11.2.0/snos/bin/g++"
--cmake-opt="-DGCC_INSTALL_PREFIX=/opt/cray/pe/gcc/11.2.0/snos" --llvm-external-projects openmp

python3 $PWD/buildbot/compile.py -j4
```

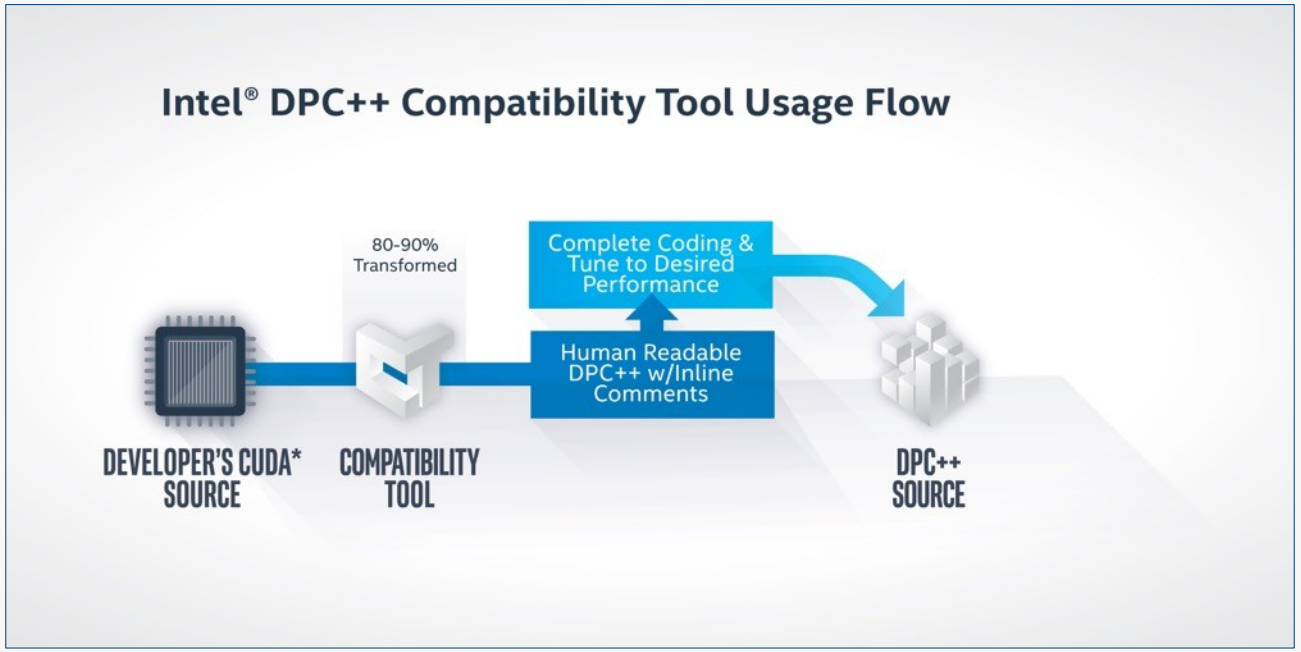
Where are my SYCL compilers installed ?

train515@nid001608:~/llvm/build/bin>

How to port existing CUDA to SYCL ?

Intel® DPC++ Compatibility Tool

Assist in migrating CUDA* applications to SYCL/DPC++, extending user choices

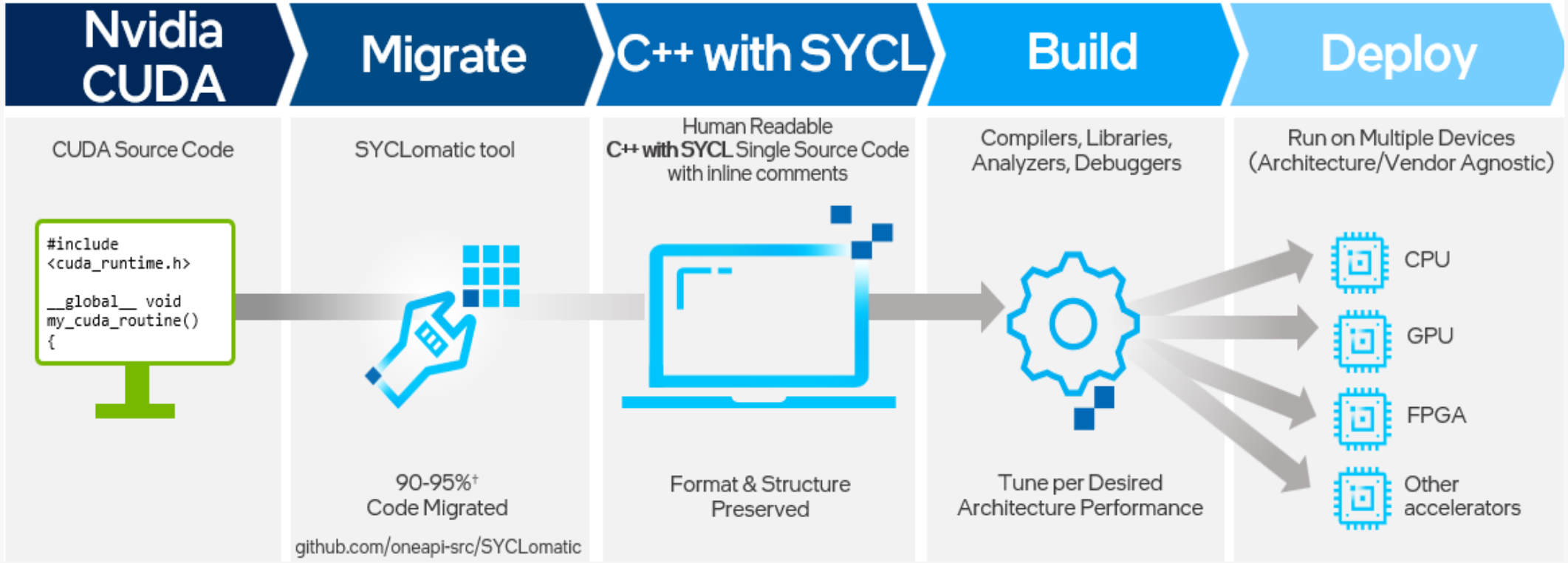


- Assists developers migrating code written in CUDA* to DPC++
- Target is to migrate up to 80-90% of code automatically
- Inline comments are provided to help developer complete code

[Codeplay: CUDA to SYCL for Beginners]
[Intel Compatibility Tool]

Porting CUDA projects to SYCL

SYCLomatic: A New CUDA*-to-SYCL* Code Migration Tool
(previously referred to as DPCT, DPC++ Compatibility Tool)



oneAPI core elements

oneDPL

oneAPI Data Parallel C++ Library

A companion to the DPC++ Compiler for programming oneAPI devices with APIs from C++ standard library, Parallel STL, and extensions.

oneDNN

oneAPI Deep Neural Network Library

High performance implementations of primitives for deep learning frameworks.

oneCCL

oneAPI Collective Communications Library

Communication primitives for scaling deep learning frameworks across multiple devices.

Level Zero

oneAPI Level Zero

System interface for oneAPI languages and libraries.

oneDAL

oneAPI Data Analytics Library

Algorithms for accelerated data science.

oneTBB

oneAPI Threading Building Blocks

Library for adding thread-based parallelism to complex applications on multiprocessors.

oneMKL

oneAPI Math Kernel Library

High performance math routines for science, engineering, and financial applications.

Instructions to build SYCLomatic (Optional)

Repository: https://github.com/oneapi-src/SYCLomatic

Setup Environment:

```
export SYCLOMATIC_HOME=~/workspace
export PATH_TO_C2S_INSTALL_FOLDER=~/workspace/c2s_install
mkdir $SYCLOMATIC_HOME
cd $SYCLOMATIC_HOME

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

Build Instructions:

```
cd $SYCLOMATIC_HOME
mkdir build
cd build
cmake -G Ninja -DCMAKE_INSTALL_PREFIX=$PATH_TO_C2S_INSTALL_FOLDER  -DCMAKE_BUILD_TYPE=Release  -
DLLVM_ENABLE_PROJECTS="clang"  -DLLVM_TARGETS_TO_BUILD="X86;NVPTX" ../SYCLomatic/llvm
ninja install-c2s
```

Post Installation:

```
export PATH=$PATH_TO_C2S_INSTALL_FOLDER/bin:$PATH
export CPATH=$PATH_TO_C2S_INSTALL_FOLDER/include:$CPATH
```

ALCF Polaris: `module load oneapi/upstream` already has SYCLomatic

Things to notice when using SYCLomatic

`https://github.com/oneapi-src/SYCLomatic`

1. SYCLomatic is a tool that bridges the gap between CUDA to SYCL

2. “dpct” namespace and headers are used for porting to SYCL

3. “dpct” headers and namespace are not standard SYCL APIs. They are merely wrappers/helpers

4. (Optional) For a SYCL specification complaint code (or) for production purpose: Consider manually replacing “dpct” with SYCL equivalents

Hands-on Session for Porting CUDA to SYCL

Objective: Learn how to use SYCLomatic to port projects in CUDA

https://github.com/oneapi-src/oneAPI-samples/tree/master/Tools/Migration/rodinia-nw-dpct

Login to compute node on ALCF Polaris

```
module use /soft/modulefiles/
module load oneapi
```

Source

```
git clone -b master --depth=1 https://github.com/oneapi-src/oneAPI-samples.git
cd oneAPI-samples/Tools/Migration/rodinia-nw-dpct
```

To Port

```
make clean
intercept-build make
dpct -p compile_commands.json --in-root=. --out-root=migration_sycl
```

Hands-on Session for Porting CUDA to SYCL

Objective: Learn how to use SYCLomatic to port projects in CUDA

https://github.com/oneapi-src/oneAPI-samples/tree/master/Tools/Migration/rodinia-nw-dpct

Source-code for SYCL

```
cd oneAPI-samples/Tools/Migration/rodinia-nw-dpct/migration_sycl
```

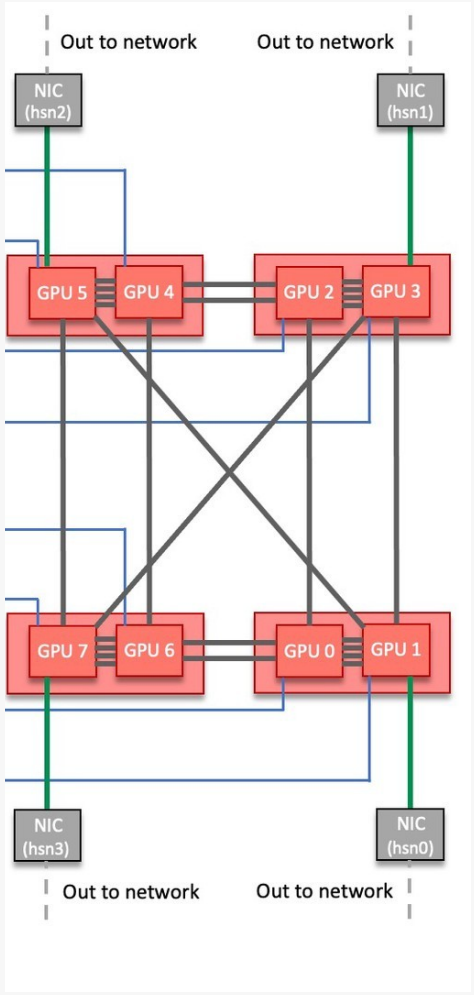
Instructions to cleanup

0. Copy the Makefile from `oneAPI-samples/Tools/Migration/rodinia-nw-dpct/Makefile`
1. Change the Makefile to point to SYCL compiler on Polaris (clang++)
2. Change the CUDA filenames in the Makefile to point to SYCL ported files
3. [Important] Add these SYCL flags to clang++ compiler

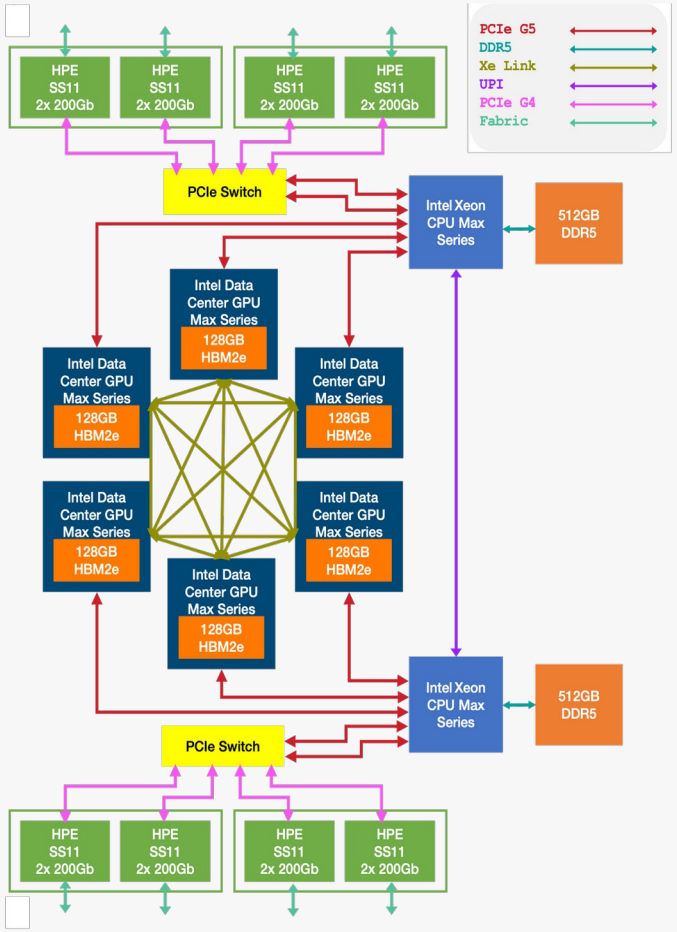
```
CXX=clang++ -std=c++17 -O3 -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend -cuda-gpu-arch=sm_80
```

4. Final step is to replace DPCT functional with standard SYCL API

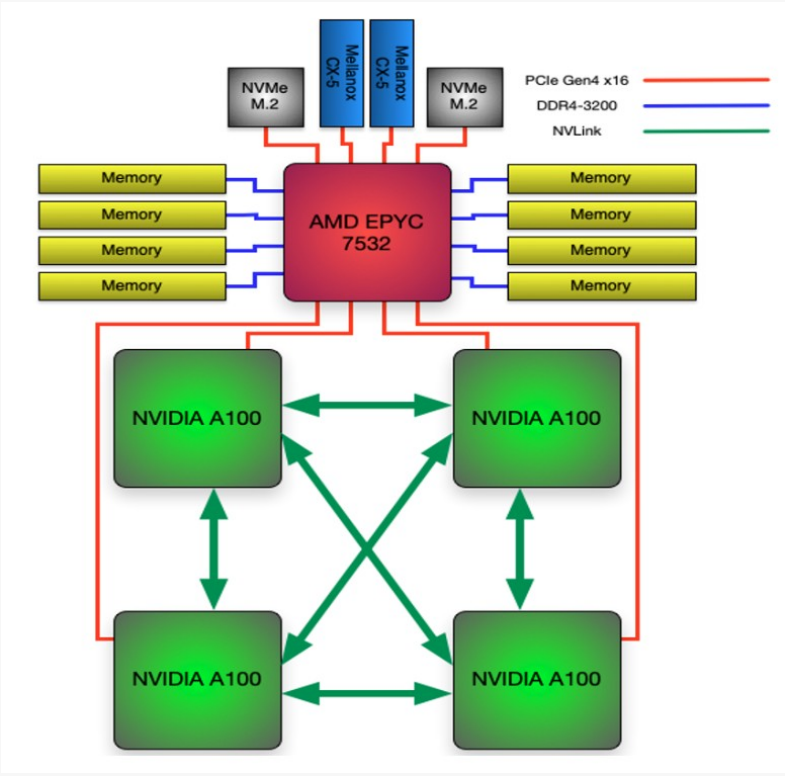
Multi-GPU Programming



OLCF Frontier – 4 MI250x (8 GCDs)



ALCF Aurora – 6 PVC (12 tiles/Xe-Stacks)



ALCF Polaris – 4 A100

Multi-GPU Programming

- SYCL queue provides the handle to submit work on a given device (CPU/GPU/FPGA, etc)
- Single GPU obtained via: `sycl::queue q(sycl::gpu_selector_v);`

- `sycl::gpu_selector_v` provides access to the first GPU available
- Exposing visibility of devices can be altered via setting the environment variables
- `CUDA_VISIBLE_DEVICES, ROCR_VISIBLE_DEVICES, ZE_AFFINITY_MASK`

```
abagusetty@x3011c0s31b0n0 ~ $ CUDA_VISIBLE_DEVICE=3 sycl-ls --verbose
INFO: Output filtered by ONEAPI_DEVICE_SELECTOR environment variable, which is set to cuda:gpu.
To see device ids, use the --ignore-device-selectors CLI option.

[cuda:gpu] NVIDIA CUDA BACKEND, NVIDIA A100-SXM4-40GB 8.0 [CUDA 12.2]
```

Hands-on: SYCL-single-gpu

Multi-GPU Programming – ALCF Polaris

- (Traditional) MPI + X: **Single-GPU per MPI** - Multiple GPUs can be exposed via binding each GPU to a MPI-processes, each MPI-process acts as if there is 1 GPU
- **Multi-GPU per MPI** All GPUs are visible to any given MPI-process

```
auto gpus = sycl::platform(sycl::gpu_selector_v).get_devices();
…
sycl::queue q0(gpus[0]);
sycl::queue q1(gpus[1]);
…
```

```
abagusetty@x3011c0s31b0n0 ~ $ sycl-ls --verbose
INFO: Output filtered by ONEAPI_DEVICE_SELECTOR environment variable, which is set to cuda:gpu.
To see device ids, use the --ignore-device-selectors CLI option.

[cuda:gpu] NVIDIA CUDA BACKEND, NVIDIA A100-SXM4-40GB 8.0 [CUDA 12.2]
[cuda:gpu] NVIDIA CUDA BACKEND, NVIDIA A100-SXM4-40GB 8.0 [CUDA 12.2]
[cuda:gpu] NVIDIA CUDA BACKEND, NVIDIA A100-SXM4-40GB 8.0 [CUDA 12.2]
[cuda:gpu] NVIDIA CUDA BACKEND, NVIDIA A100-SXM4-40GB 8.0 [CUDA 12.2]
```

```
abagusetty@x3011c0s31b0n0 ~ $ nvidia-smi
Thu Aug  1 16:18:57 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.154.05             Driver Version: 535.154.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA A100-SXM4-40GB          On  | 00000000:07:00.0 Off |                    0 |
| N/A   30C    P0              52W / 400W |      0MiB / 40960MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA A100-SXM4-40GB          On  | 00000000:46:00.0 Off |                    0 |
| N/A   30C    P0              54W / 400W |      0MiB / 40960MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   2  NVIDIA A100-SXM4-40GB          On  | 00000000:85:00.0 Off |                    0 |
| N/A   29C    P0              52W / 400W |      0MiB / 40960MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   3  NVIDIA A100-SXM4-40GB          On  | 00000000:C7:00.0 Off |                    0 |
| N/A   32C    P0              55W / 400W |      0MiB / 40960MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Multi-GPU Programming – ALCF Polaris

- (Traditional) MPI + X: **Single-GPU per MPI** - Multiple GPUs can be exposed via binding each GPU to a MPI-processes, each MPI-process acts as if there is 1 GPU

Source code: https://docs.alcf.anl.gov/polaris/programming-models/sycl-polaris/
Look out for a section on “Toggle for SYCL example with OpenMP & MPI for CPU-side”

```
mpiexec -n 4 --ppn 4 --env OMP_NUM_THREADS=1 ./set_affinity_gpu_polaris.sh ./hello_jobstep.out

MPI 000 - OMP 000 - HWT 000 - Node x3200c0s37b0n0 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID
0000:C7:00.0
MPI 001 - OMP 000 - HWT 001 - Node x3200c0s37b0n0 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID
0000:85:00.0
MPI 003 - OMP 000 - HWT 003 - Node x3200c0s37b0n0 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID
0000:07:00.0
MPI 002 - OMP 000 - HWT 002 - Node x3200c0s37b0n0 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID
0000:46:00.0
```

Multi-GPU Programming – (Hands-on, ALCF Polaris)

Objective: Learn how to use expose multi-GPUs via SYCL APIs

https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL/Jupyter/oneapi-essentials-training/14_SYCL_Multi_GPU_Programming/src

Login to compute node on ALCF Polaris

```
module use /soft/modulefiles/
module load oneapi
```

Source

```
git clone -b master --depth=1 https://github.com/oneapi-src/oneAPI-samples.git
cd oneAPI-samples/DirectProgramming/C++SYCL/Jupyter/oneapi-essentials-training/
14_SYCL_Multi_GPU_Programming/src
```

To Compile

```
clang++ -std=c++17 -fsycl -fsycl-targets=nvptx64-nvidia-cuda-sycldevice -Xsycl-
target-backend -cuda-gpu-arch=sm_80 FILENAME.cpp -o FILENAME.out
```

SYCL SC – Safety Critical

- https://www.khronos.org/syclsc

Demand for advanced GPU-accelerated graphics and compute is growing in an increasing number of industries where safety is paramount, such as automotive, autonomy, avionics, medical, industrial, and energy

1990s
Avionics

2010s
Automotive

2020s…
Everywhere

Safety-critical APIs reduce system-level functional safety certification effort and costs

1) Streamlined functionality to reduce documentation and testing
2) Deterministic behavior to simplify system design and testing
3) Unambiguous API with minimized undefined behavior
4) Comprehensive and robust error/fault handling

Industry safety-critical standards include
RTCA DO-178C (avionics) | ISO 26262 ASIL D (automotive)
IEC 61508 (industrial) | IEC 62304 (medical)

SYCL SC – Safety Critical – Design Philosophy

- https://www.khronos.org/syclsc

SYCL SC will use the same definition and scope of "safety critical" as Vulkan SC to reduce system-level functional safety certification effort and costs

Deterministic
Predictable execution times and results simplify design and test

Robust
Comprehensive error handling, remove ambiguity, eliminate undefined behavior

Simplified
Streamline functionality to reduce documentation and testing

Minimize Divergence
Between SYCL and SYCL SC with SYCL SC aiming to be a subset of SYCL wherever possible

Flexible Development
Enable SYCL SC to use off-the-shelf C++ compiler, for debugging, emulation, simulation and testing

No C++ language extensions
Add features via C++ libraries

SYCL SC – Safety Critical – Getting Involved

Join the Khronos SYCL SC Working Group

KHRONOS GROUP

The SYCL SC Working Group is open to any Khronos member,
and Khronos membership is open to any company.

https://www.khronos.org/members/ or email memberservices@khronosgroup.org

SYCL SC Working Group meetings starting in late March 2023
https://www.khronos.org/syclsc

Participate to influence Khronos' growing family of open, royalty-free standards
for embedded and safety-critical markets