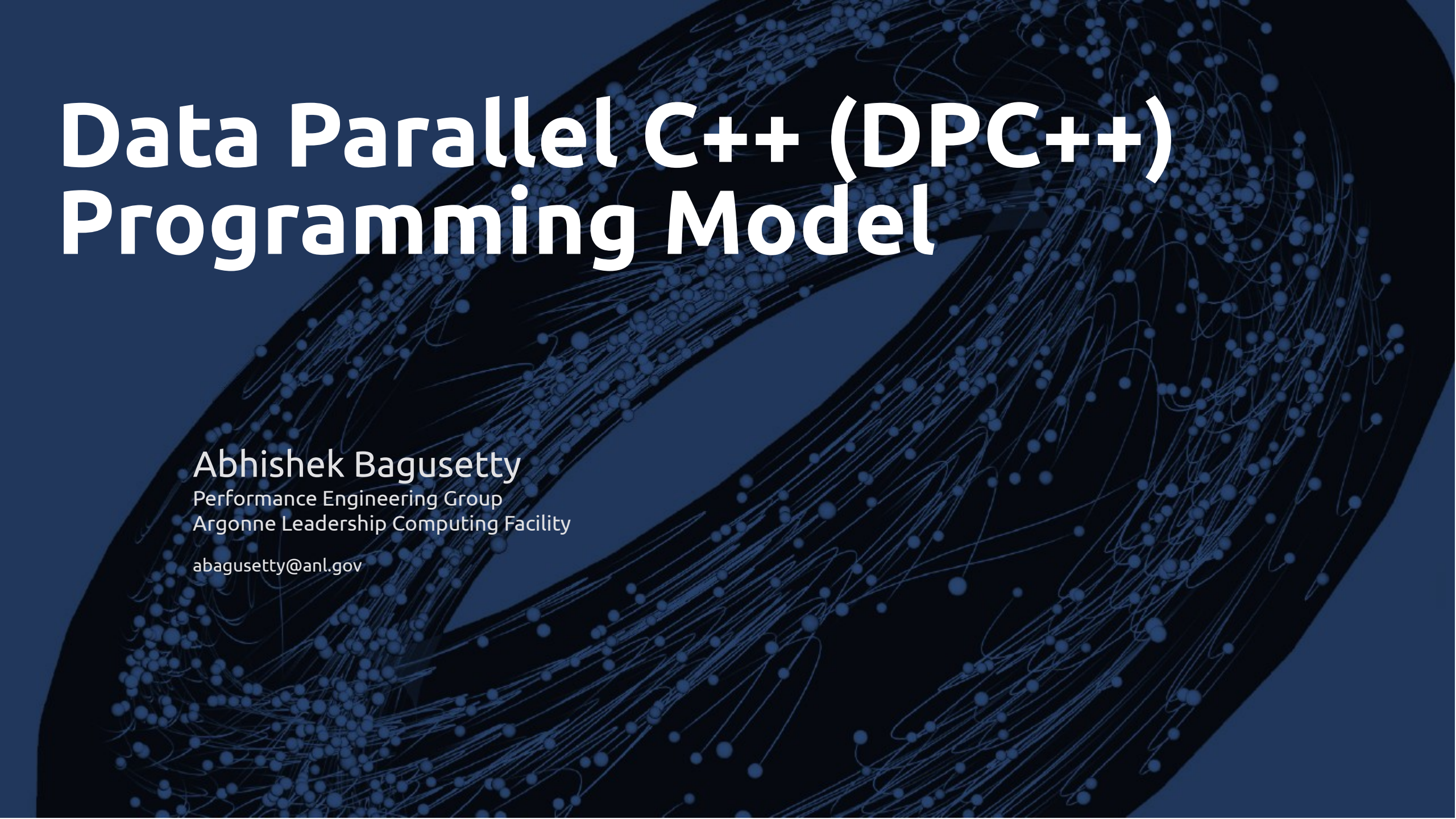


Data Parallel C++ (DPC++) Programming Model

Abhishek Bagusetty
Performance Engineering Group
Argonne Leadership Computing Facility

abagusetty@anl.gov

SYCL – Specification

KHRONOS GROUP

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

– Heuristics looks similar to OpenCL-C bindings
– C++ single source (co-exists 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

Data Parallel C++ (DPC++)

Intel's 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

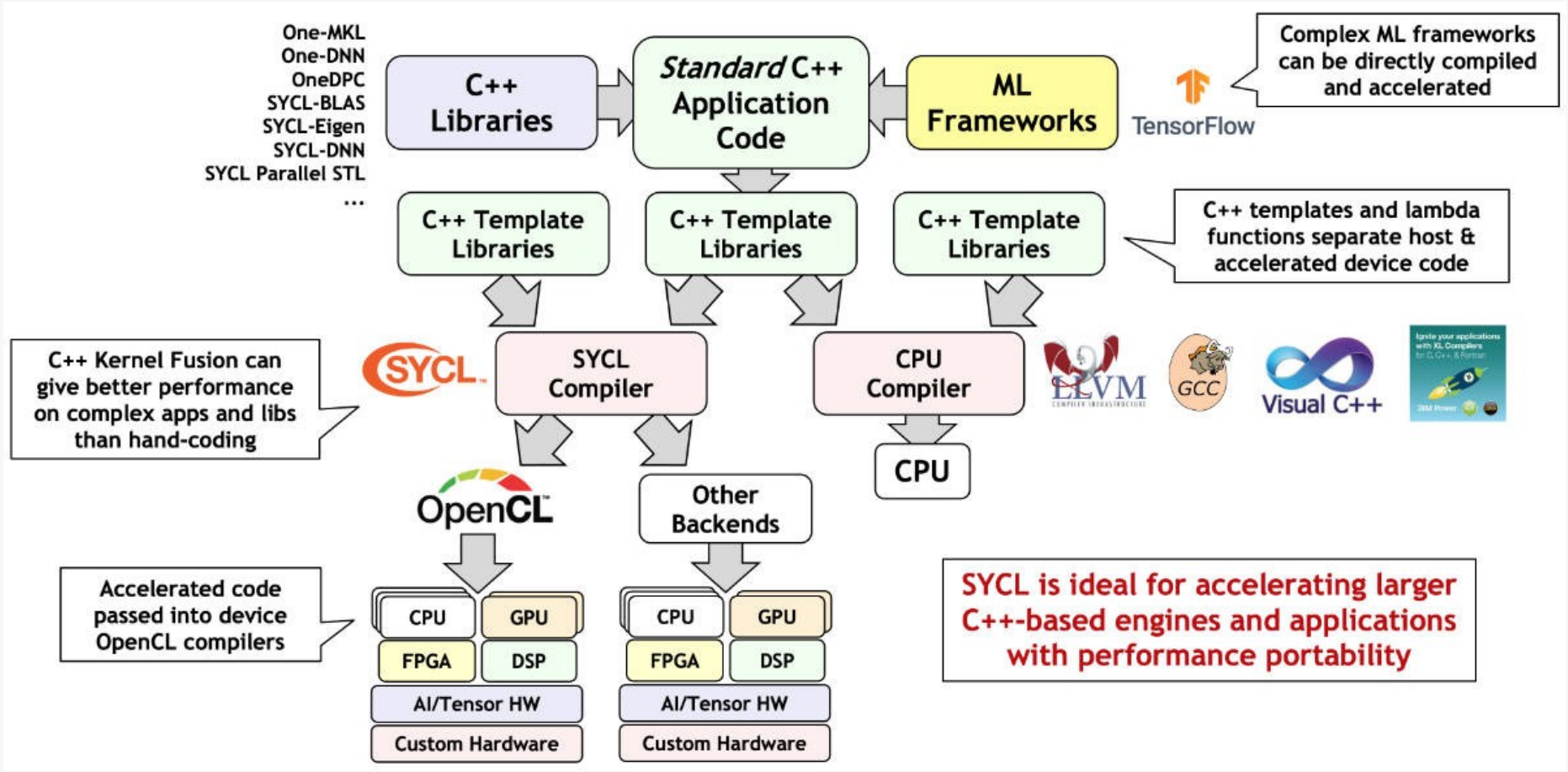
Fast-moving open collaboration feeding into the SYCL* standard

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

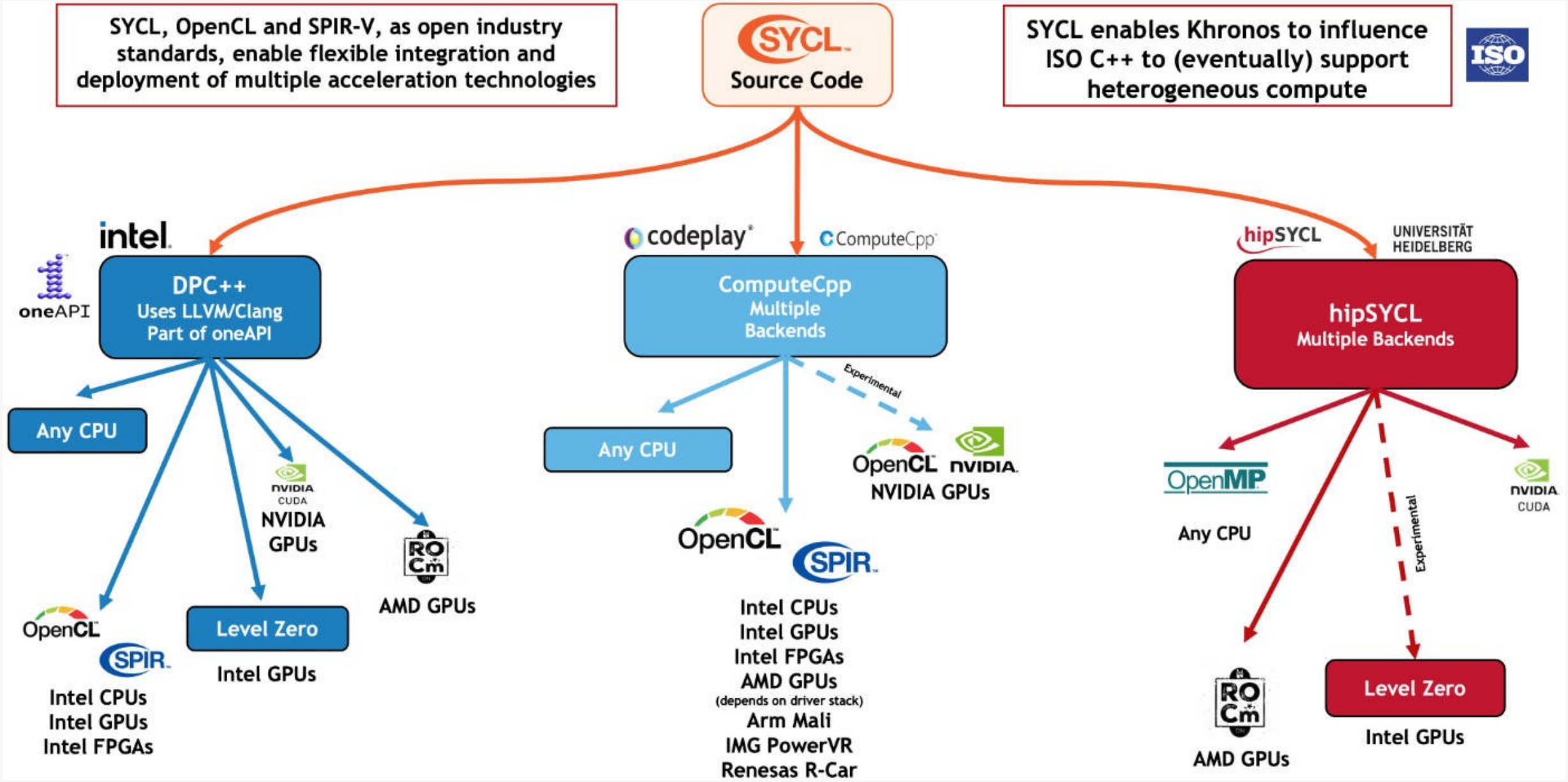
SYCL – A Portable Programming Model

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

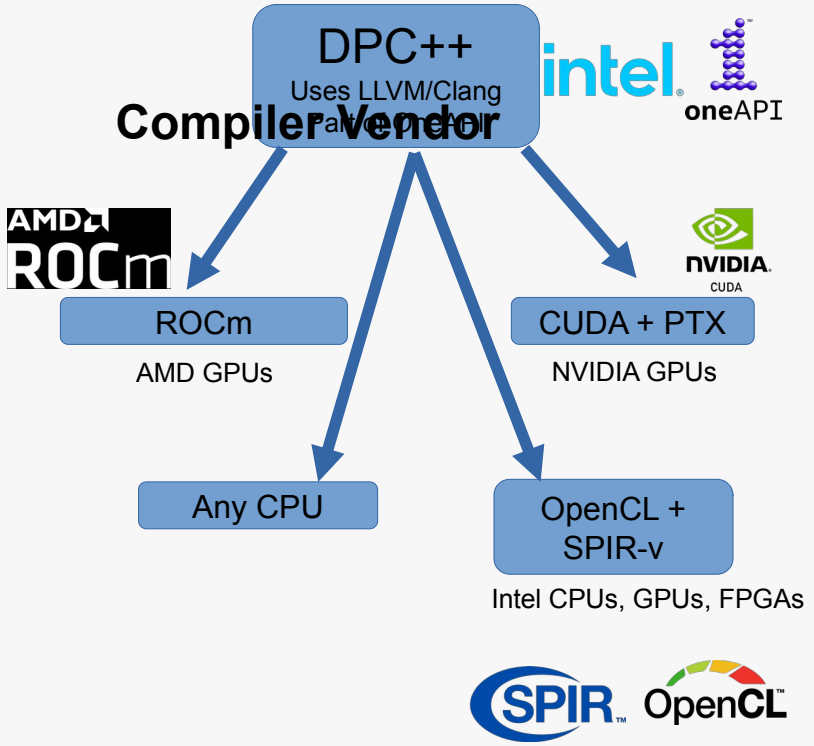


SYCL @ NERSC

- Collaboration between ALCF, NERSC and Codeplay to enable support for NVIDIA A100 GPUs in LLVM DPC++/SYCL2020

- Initial scope of work complete
  - support for tensor cores, USM, atomics, and more available

- Current focus on performance, upstreaming features to LLVM, tracking library support (e.g. FFT, oneMKL)



PrgEnv-llvm for CPE

NERSC has developed an additional PrgEnv which adds to the Cray Programming Environment (CPE) that HPE provides.

- LLVM compiler with support for OpenMP offload, SYCL

Argonne NATIONAL LABORATORY

NERSC National Energy Research Scientific Computing Center

codeplay

https://docs.nersc.gov/development/programming-models/sycl/

Credits: Brandon Cook (NERSC)

SYCL @ NERSC


NERSC
Powering Scientific Discovery Since 1974
My NERSC | A-Z Index | Share | Follow
search...
HOME ABOUT SCIENCE SYSTEMS FOR USERS NEWS R & D EVENTS LIVE STATUS
FOR USERS
Getting Help
NERSC Code of Conduct
Live Status
Getting Started
Accounts & Allocations
Documentation
Policies
My NERSC
Job Logs & Statistics
Training & Tutorials
Training Events
Migrating from Cori to Perlmutter Training, Dec 1, 2022
Migrating from Cori to Perlmutter Office Hours, Nov 2022 to Jan 2023
NERSC GPU Hackathon, Nov-Dec 2022
SpinUp Workshop: Nov-Dec 2022
Data Day 2022, October 26-27
GPUs for Science day 2022, October 25th
Quantum for Science day 2022, October 24th
Home » For Users » Training & Tutorials » Training Events » An Introduction to Programming with SYCL on Perlmutter and Beyond, March 1, 2022
AN INTRODUCTION TO PROGRAMMING WITH SYCL ON PERLMUTTER AND BEYOND, MARCH 1, 2022
Introduction
SYCL is an open standard programming model that allows developers to use standard C++ code to program for a range of GPUs and other accelerator processors. This means that it is possible to develop using modern C++ code and target Nvidia, AMD and Intel GPUs from a single code base. To enable SYCL on the latest supercomputers, Codeplay has been working in partnership with different National Laboratories to bring SYCL support to Perlmutter, Polaris and Frontier.
Join engineers from Codeplay for a half day hands-on workshop that will walk through the fundamentals of SYCL programming using practical examples and exercises to help reinforce the learning. Attendees will also learn how to compile their SYCL code using the DPC++ compiler to target Nvidia GPUs including those on the Perlmutter supercomputer. Lastly, we'll talk about some of the things you need to know to achieve good performance, including best practices for memory management, with free time for questions and discussions.
ALCF and OLCF users are welcome to this training. NERSC training accounts will be provided if needed.
Workshop Leader: Hugh Delaney, Software Engineer, Codeplay Software
Course Outline
Introduction

Queues & Contexts

- "SYCL Queues" provide mechanism to **submit** work to a **device**
- "SYCL Contexts" is well known to be over-looked

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

- **Context** (aka cuContext)
  - Contexts are used for resources isolation and sharing
  - A SYCL context may consist of one or multiple devices
  - Memory created can be shared only if their associated queue(s) are created using the same context

- **Queue** (aka CUDA Stream)
  - ✓ 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)

Bring You Own Compiler – Perlmutter

(~30 mins, plan accordingly)

Download the compiler:

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

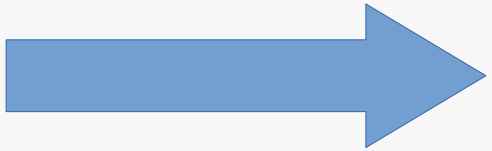
Build & Install: (takes a while)

```
module load cudatoolkit/11.5
export DPCPP_HOME=$HOME

cd llvm

export CUDA_LIB_PATH=/opt/nvidia/hpc_sdk/Linux_x86_64/21.11/cuda/lib64/stubs
CC=`which gcc` CXX=`which g++` python $DPCPP_HOME/llvm/buildbot/configure.py --cuda --cmake-gen="Unix Makefiles" --cmake-opt="-
DCUDA_TOOLKIT_ROOT_DIR=/opt/nvidia/hpc_sdk/Linux_x86_64/21.11/cuda/11.5"

python $DPCPP_HOME/llvm/buildbot/compile.py
```

Where are my SYCL compilers installed ?

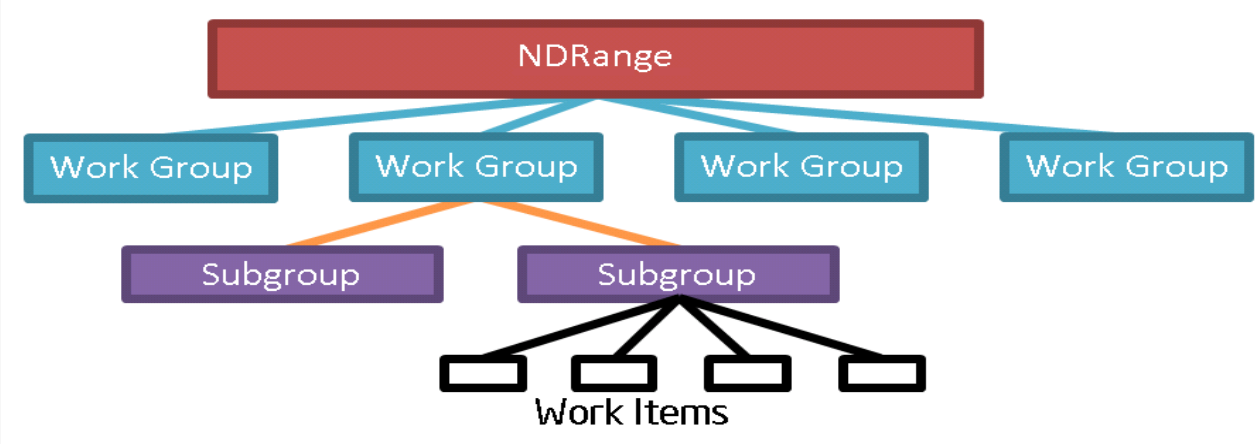
train515@nid001608:~/llvm/build/bin>

Porting from CUDA to SYCL



Execution Model: CUDA vs SYCL

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.

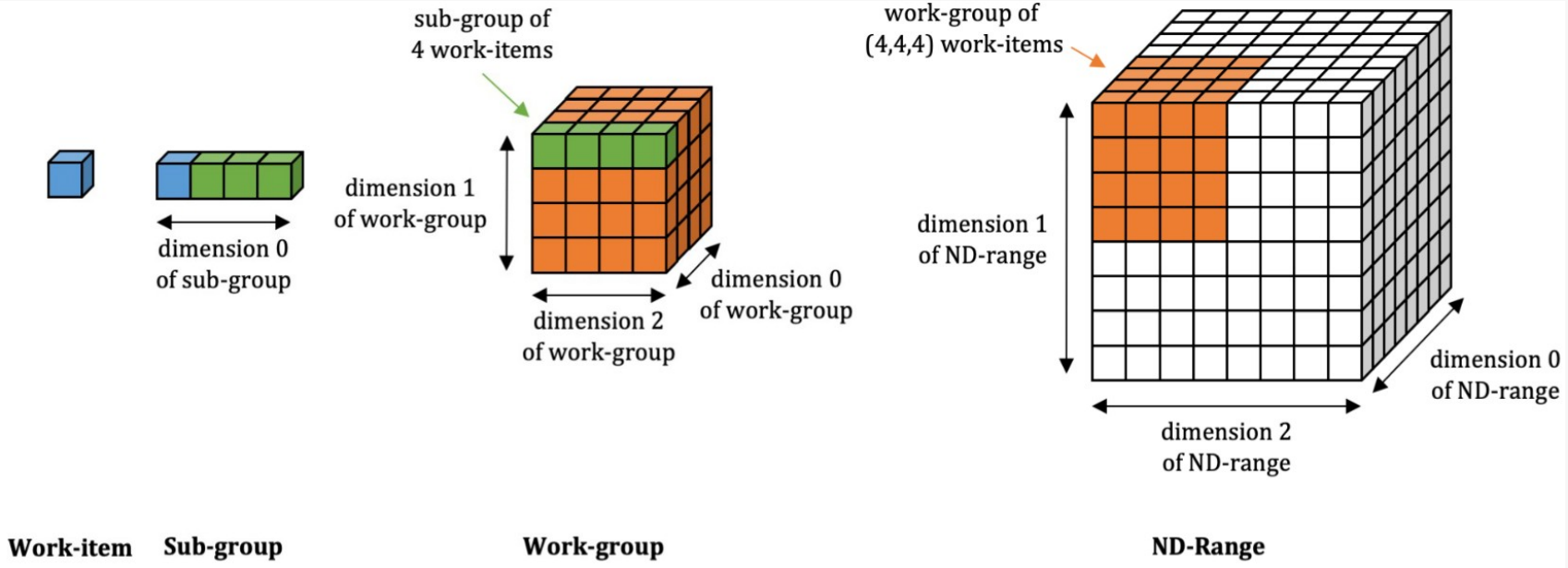
Leveraging sub-groups will help to map execution to low-level hardware and may help in achieving higher performance.

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

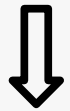
Allocation Type	Initial Location	Accessible By		Migratable To	
`device`	`device`	`host`	No	`host`	No
		`device`	Yes	`device`	N/A
		Another `device`	Optional (P2P)	Another `device`	No
`host`	`host`	`host`	Yes	`host`	N/A
		Any `device`	Yes	`device`	No
`shared`	Unspecified	`host`	Yes	`host`	Yes
		`device`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

Vector Addition: SYCL Buffer memory model

```
#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} };

  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();
 }

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

Host Code

Device Code

Host Code

Create SYCL buffers using host pointers.

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

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

Host Code

```
#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));
```

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

Device Code

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

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

Host Code

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

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

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

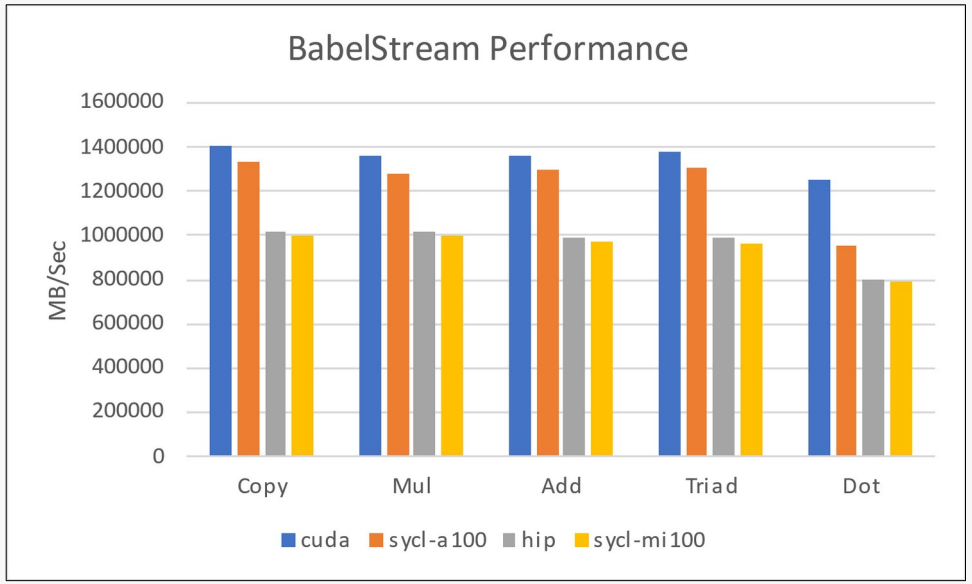
Device Code

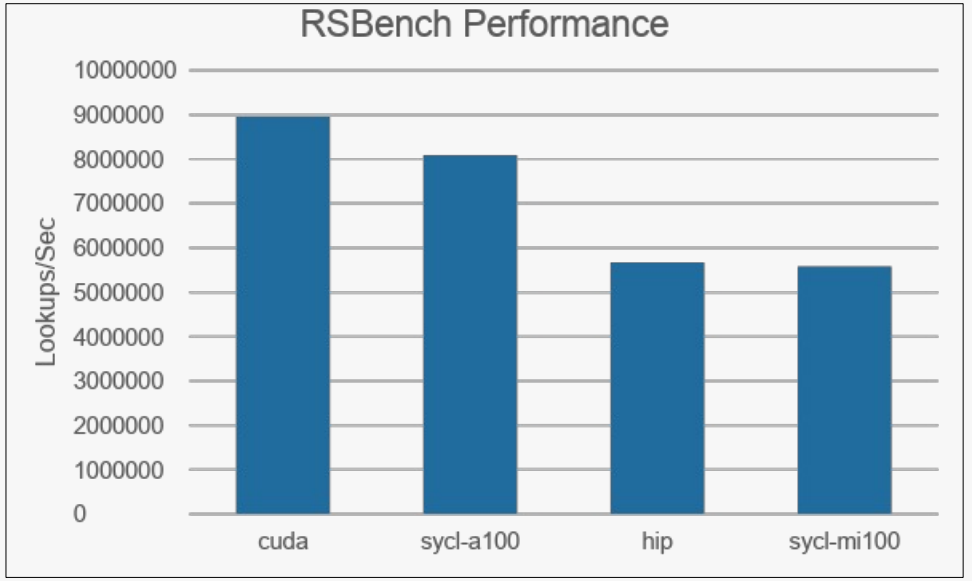
Host Code

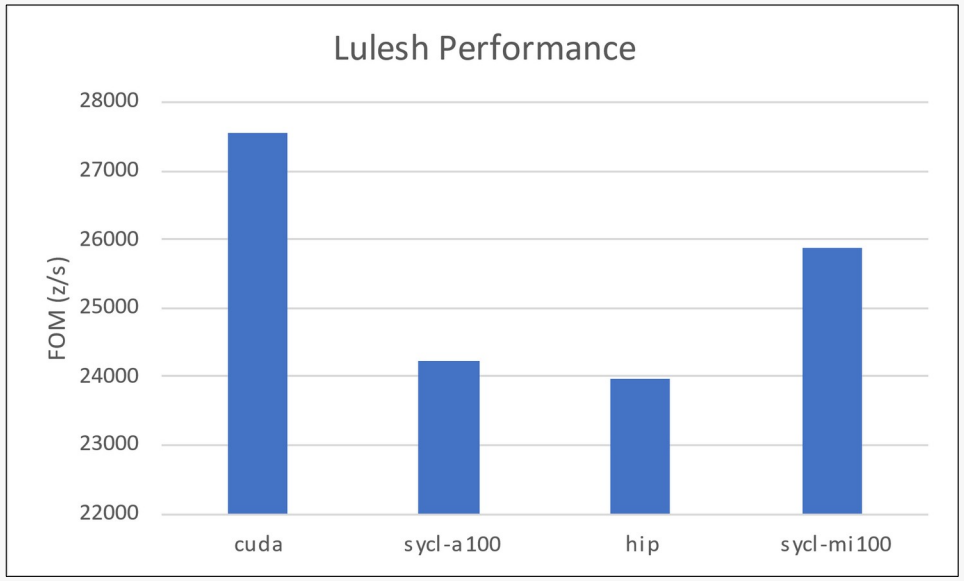
SYCL queue (in-order) i.e., FIFO like cudaStream_t

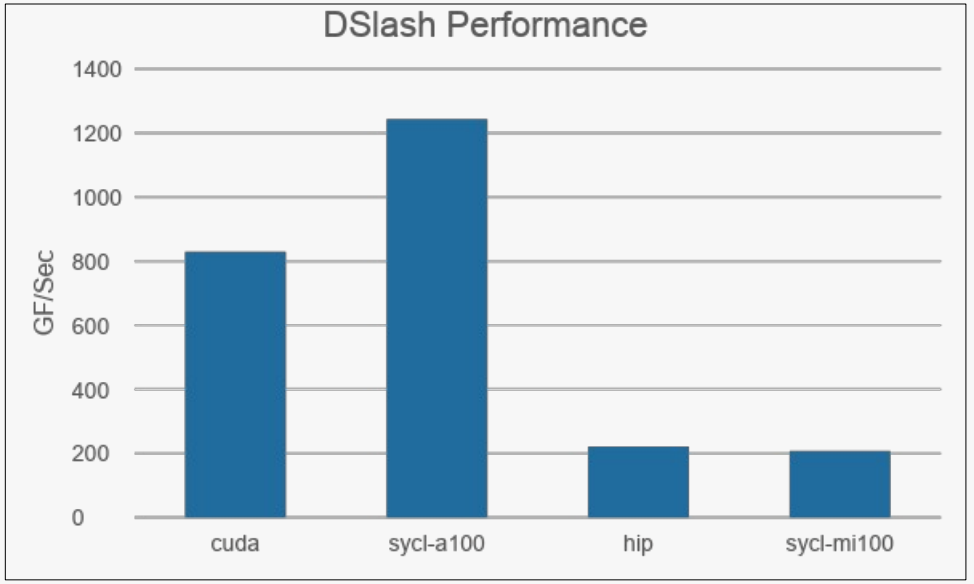
myQueue.wait(), wait for D2H to complete before printing "C"

Performance Benchmarks



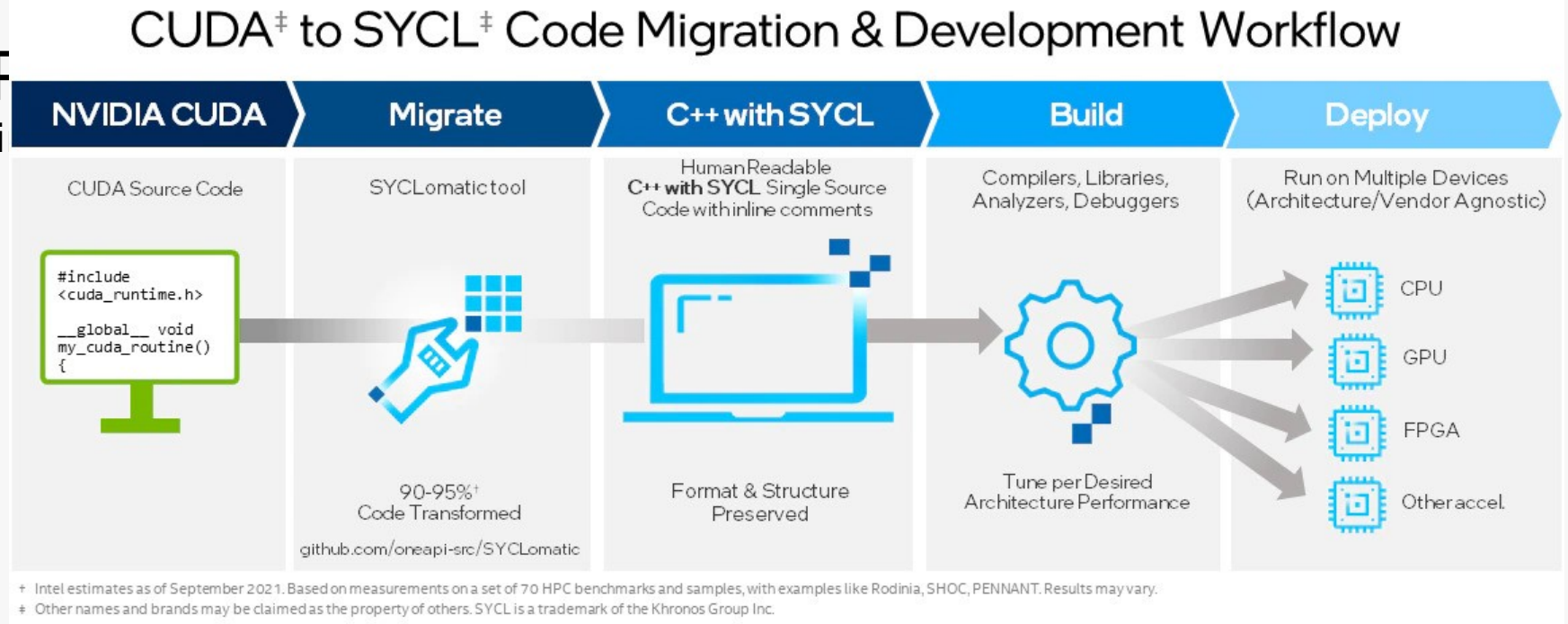






Tools : How to port existing CUDA to SYCL ?

Intel® DP
Assist in migrati



SYCLomatic: A "open-source" New CUDA*-to-SYCL* Code Migration Tool

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

Additional Resources:
https://developer.codeplay.com/products/computecpp/ce/guides/sycl-for-cuda-developers/cuda-to-sycl-examples

Math Libraries : What are my options for cublas,cu* ?

- open-source implementation of the oneMKL Data Parallel C++ (DPC++) interface
- works with multiple devices (backends) uses vendor device-specific libraries underneath

Note: Apart of device-backend, supports host-CPU interface: Intel MKL, NETLIB

	NVIDIA	AMD	Intel
BLAS	cuBLAS	rocBLAS	oneMKL
Linear Solvers	cuSOLVER	(rocSOLVER)	oneMKL
Random Numbers	cuRAND	rocRAND	oneMKL
FFT	(cuFFT)	(rocFFT)	(oneMKL)

(work-in-progress)

Questions

https://developer.codeplay.com/products/computecpp/ce/guides/sycl-for-cuda-developers/cuda-to-sycl-examples
https://www.intel.com/content/www/us/en/developer/articles/training/intel-dpcpp-compatibility-tool-training.html