CUDA to SYCL Migration

Learn how to migrate CUDA source to C++ SYCL source using SYCLomatic Tool

rakshith.krishnappa@intel.com

Why Migrate to SYCL?

What is SYCL?

SYCL (pronounced ‘sickle’) is a royalty-free, cross-platform abstraction layer that:

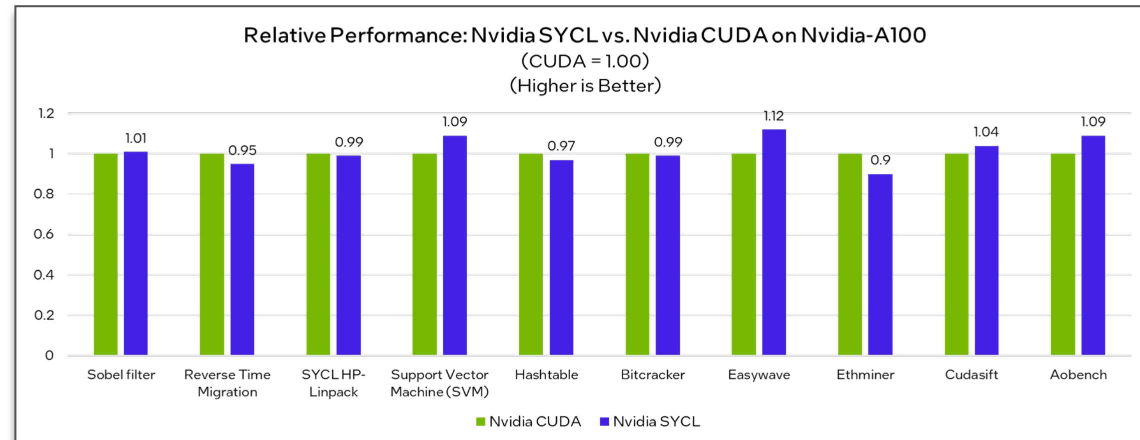
- Enables code for heterogeneous and offload processors to be written using modern ISO C++ (at least C++ 17).
- Provides APIs and abstractions to find devices (e.g. CPUs, GPUs, FPGAs) on which code can be executed, and to manage data resources and code execution on those devices.

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

Accelerating Choice with SYCL

Khronos Group Standard

- Open, standards-based
- Multiarchitecture performance
- Freedom from vendor lock-in
- Comparable performance to native CUDA on Nvidia GPUs
- Extension of widely used C++ language
- Speed code migration via open source SYCLomatic or Intel® DPC++ Compatibility Tool



Testing Date: Performance results are **based on testing by Intel as of August 15, 2022** and may not reflect all publicly available updates.

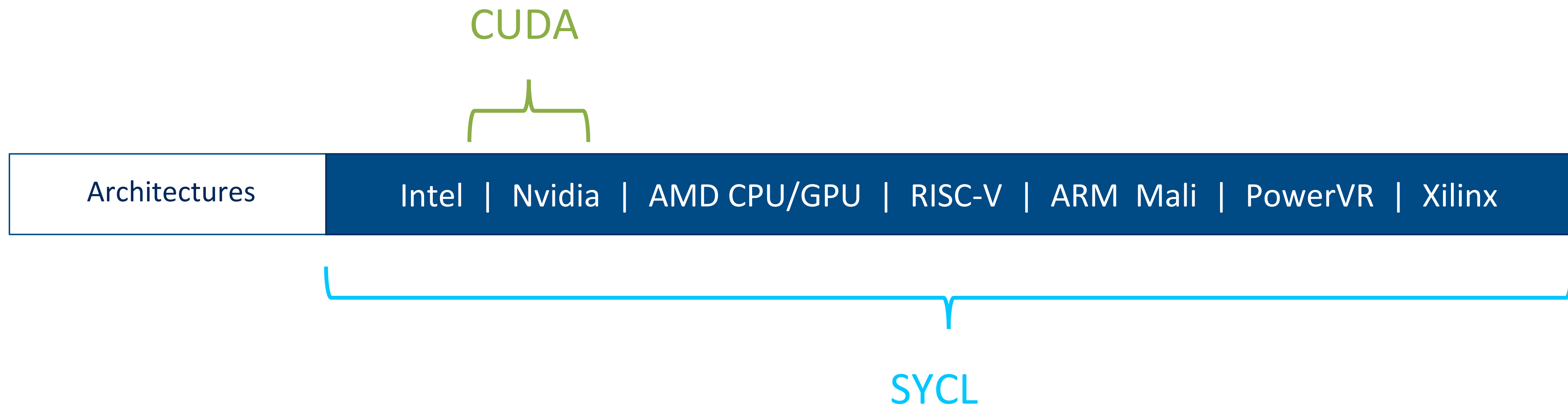
Configuration Details and Workload Setup: Intel® Xeon® Platinum 8360Y CPU @ 2.4GHz, 2 socket, Hyper Thread On, Turbo On, 256GB Hynix DDR4-3200, ucode 0x000363. GPU: Nvidia A100 PCIe 80GB GPU memory. Software: SYCL open source/CLANG 15.0.0, CUDA SDK 11.7 with NVIDIA-NVCC 11.7.64, cuMath 11.7, cuDNN 11.7, Ubuntu 22.04.1. SYCL open source/CLANG compiler switches: -fscycl-targets=nvptx64-nvidia-cuda, NVIDIA NVCC compiler switches: -O3 –gencode arch=compute_80, code=sm_80. Represented workloads with Intel optimizations.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex. Your costs and results may vary.

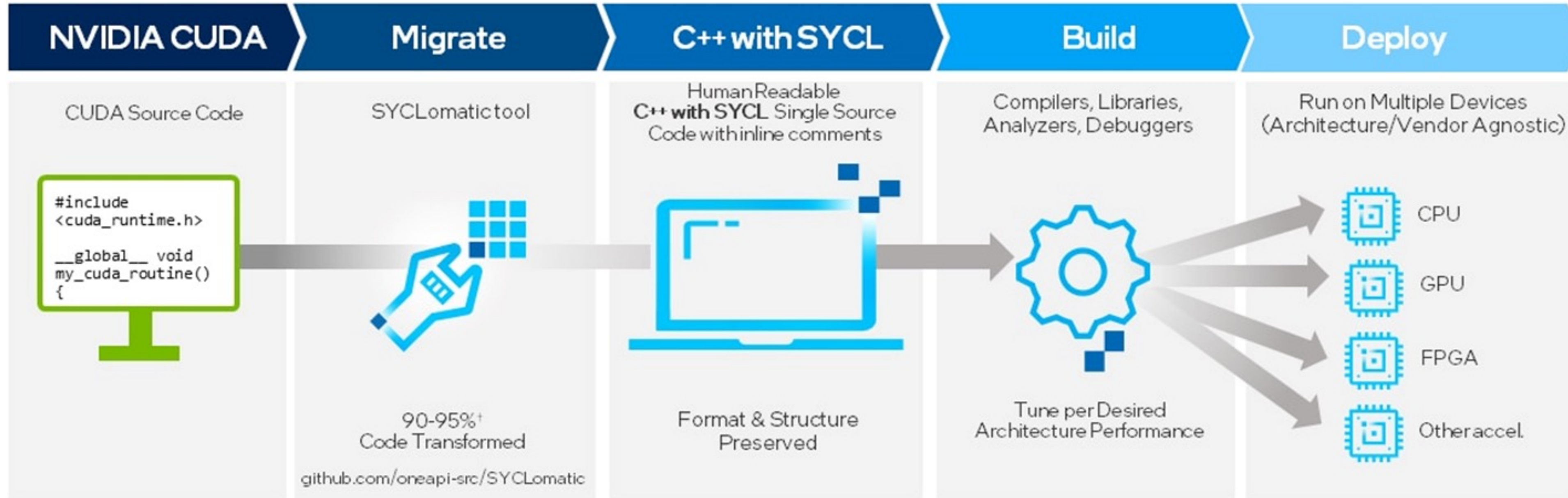
Architectures	Intel | Nvidia | AMD CPU/GPU | RISC-V | ARM Mali | PowerVR | Xilinx

Architecture Support – CUDA vs SYCL



CUDA to SYCL Migration Made Easy

Open Source SYCLomatic Tool Reduces Code Migration Time



Assists developers migrating code written in CUDA to C++ with SYCL, generating **human readable** code wherever possible

~90-95% of code typically migrates automatically[1]

Inline comments are provided to help developers finish porting the application

Intel® DPC++/C++ Compatibility Tool is Intel's implementation, available in the Base Toolkit

[1]Intel estimates as of September 2021. Based on measurements on a set of 70 HPC benchmarks and samples, with examples like Rodinia, SHOC, PENNANT. Results may vary.

intel.

Productive and Performant SYCL Compiler

Intel® oneAPI DPC++/C++ Compiler

Uncompromised parallel programming productivity and performance across CPUs and accelerators

- Allows code reuse across hardware targets, while permitting custom tuning for a specific accelerator
- Open, cross-industry alternative to single architecture proprietary language

Khronos SYCL Standard

- Delivers C++ productivity benefits, using common and familiar C and C++ constructs
- Created by Khronos Group to support data parallelism and heterogeneous programming

Builds upon Intel's decades of experience in architecture and high-performance compilers

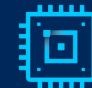
Learn More & Download

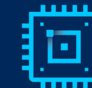
oneAPI DPC++/C++ Compiler and Runtime

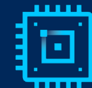
C++ with SYCL Source Code

Clang/LLVM

C++ SYCL Runtime







There will still be a need to tune for each architecture.

Codeplay Compiler Plug-ins for Nvidia and AMD GPUs

Adding support for NVIDIA and AMD GPUs to the Intel® oneAPI Base Toolkit

oneAPI for NVIDIA & AMD GPUs

- Free Codeplay download of latest binary plugins to the Intel DPC++/C++ compiler:
  - Nvidia GPU
  - AMD Beta GPU
- Availability at the same time as the Intel oneAPI Base Toolkit
- Plug-ins updated quarterly in-sync with oneAPI

Priority Support

- Sold by Intel and Codeplay and our channel
- Requires Intel Priority support for Intel DPC++ /C++ compiler
- Intel takes first call and Codeplay delivers backend support
- Codeplay access to older versions of plugins

Nvidia GPU plug-in | AMD GPU plug-in | Codeplay blog | Codeplay press release

CUDA to SYCL Migrated Code Comparison

Header file · Kernel function · Device Memory Allocation · Copy host to device · Submit Kernel Task

```
#include <cuda.h>
#include <iostream>
#define N 2048

// Computation Offloaded to device
__global__ void VectorAddKernel(float* A, float* B, float* C)
{
        int id = blockDim.x * blockIdx.x + threadIdx.x;
        C[id] = A[id] + B[id];
}

int main()
{
        // Initialize data on host
        float A[N], B[N], C[N];
        for (int i = 0; i < N; i++){
                A[i] = 1;
                B[i] = 2;
                C[i] = 0;
        }

        // Allocate memory on device
        float *d_A, *d_B, *d_C;
        cudaMalloc(&d_A, N*sizeof(float));
        cudaMalloc(&d_B, N*sizeof(float));
        cudaMalloc(&d_C, N*sizeof(float));

        // Copy data from  host to device
        cudaMemcpy(d_A, A, N*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, B, N*sizeof(float), cudaMemcpyHostToDevice);

        // Offload computation to device
        int nThreads = 256;
        int nBlocks = N / nThreads;
        VectorAddKernel<<<nBlocks, nThreads>>>(d_A, d_B, d_C);

        // Copy result data from device to host
        cudaMemcpy(C, d_C, N*sizeof(float), cudaMemcpyDeviceToHost);

        // Print output on host
        for (int i = 0; i < N; i++) std::cout<< C[i] << " ";
        std::cout << "\n";

        // Free device memory
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);
        return 0;
}
```

```
#include <CL/sycl.hpp>
#include <dpct/dpct.hpp>
#include <iostream>
#define N 2048

// Computation Offloaded to device
void VectorAddKernel(float* A, float* B, float* C, sycl::nd_item<3> item_ct1)
{
        int id = item_ct1.get_local_range(2) * item_ct1.get_group(2) +
                 item_ct1.get_local_id(2);
        C[id] = A[id] + B[id];
}
int main()
{
 dpct::device_ext &dev_ct1 = dpct::get_current_device();
 sycl::queue &q_ct1 = dev_ct1.default_queue();
        // Initialize data on host
        float A[N], B[N], C[N];
        for (int i = 0; i < N; i++){
                A[i] = 1;
                B[i] = 2;
                C[i] = 0;
        }

        // Allocate memory on device
        float *d_A, *d_B, *d_C;
        d_A = sycl::malloc_device<float>(N, q_ct1);
        d_B = sycl::malloc_device<float>(N, q_ct1);
        d_C = sycl::malloc_device<float>(N, q_ct1);
        // Copy data from  host to device
        q_ct1.memcpy(d_A, A, N * sizeof(float));
        q_ct1.memcpy(d_B, B, N * sizeof(float)).wait();
        // Offload computation to device
        int nThreads = 256;
        int nBlocks = N / nThreads;
        /*
        DPCT1049:0: The work-group size passed to the SYCL kernel may exceed the
        limit. To get the device limit, query info::device::max_work_group_size.
        Adjust the work-group size if needed.
        */
        q_ct1.parallel_for(sycl::nd_range<3>(sycl::range<3>(1, 1, nBlocks) *
                                             sycl::range<3>(1, 1, nThreads),
                                         sycl::range<3>(1, 1, nThreads)),
                       [=](sycl::nd_item<3> item_ct1) {
                             VectorAddKernel(d_A, d_B, d_C, item_ct1);
                       });
        // Copy result data from device to host
        q_ct1.memcpy(C, d_C, N * sizeof(float)).wait();

        // Print output on host
        for (int i = 0; i < N; i++) std::cout<< C[i] << " ";
        std::cout << "\n";

        // Free device memory
        sycl::free(d_A, q_ct1);
        sycl::free(d_B, q_ct1);
        sycl::free(d_C, q_ct1);
        return 0;
}
```

CUDA to SYCL Migration

	CUDA	SYCL
Header File	#include <cuda_runtime.h>	#include <sycl/sycl.hpp>
Create a CUDA stream	cudaStream_t stream1;	sycl::queue q;
Allocate memory on device	cudaMalloc()	sycl::malloc_device()
Memset on	cudaMemsetAsync()	q.memset()
Memcpy from host to device	cudaMemcpyAsync()	q.memcpy()

Submit kernel to device	kernel_function<<<NUM_BLOCKS, N>>>();	q.parallel_for(sycl::nd_range<1>(N, WG_SIZE) [=](sycl::nd_item<1> item){
		  kernel_function();
		});

Free device allocation	cudaFree()	sycl::free()
Synchronize host and device	cudaStreamSynchronize(stream)	q.wait()

CUDA to SYCL Migration

	CUDA	SYCL
Shared Local memory	__shared float local_data[N]	sycl::local_accessor<float, 1> local_data(N, h)
Memory synchronization / fence	cg::sync(cta);	sycl::group_barrier(group)
Atomic Add	atomicAdd()	auto a = sycl::atomic_ref() a.fetch_add()
Thread Block / Work-Group	cg::thread_block cta = cg::this_thread_block();	auto group = item.get_group();
Tile / warp / Sub-Group	cg::thread_block_tile<32> tile = cg::tiled_partition<32>(cta);	sycl::sub_group tile = item.get_sub_group();
Warp/Group Shuffle	tile.shfl_down()	sycl::shift_group_left()
Block Thread Index /Work-Group Local ID	threadIdx.x	item.get_local_id()
gem Library	cublasSgemm	oneapi::mkl::blas::column_major::gemm

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.
Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

CUDA to SYCL Migration Process

Pre-requisites

- **CUDA development machine:** Have system ready with CUDA SDK installed, you should be able to compile/run a simple CUDA sample code.

- Sign up for **Intel Developer Cloud** account at devcloud.intel.com/oneapi

CUDA to SYCL Migration Process

STEP 1

CUDA Development Machine

- Copy/Clone the CUDA source
- Install SYCLomatic
- Use SYCLomatic Tool to migrate CUDA source to SYCL source

Optional:

- *Install oneAPI Base Toolkit + NVIDIA Plugin for oneAPI*
- *Compile SYCL Source for NVIDIA GPU*

STEP 2

Intel Developer Cloud

- Login to Intel Developer Cloud
- Transfer the migrated SYCL source from CUDA Development machine to Intel Developer Cloud
- Compile SYCL source
- Fix any errors
- Compile & Execute SYCL application on Intel GPUs and CPUs, verify functional correctness.
- Optimize SYCL source for performance

CUDA to SYCL Migration Training Material

CUDA to SYCL Migration - Workshop Examples

CUDA Code	CUDA to SYCL Migration Details
VectorAdd	Shows migrating a simple single source CUDA code • Device Memory allocation and copy, kernel submission
SortingNetworks	Shows migrating a CUDA project with multiple CUDA source files • Device Memory allocation and copy, kernel submission
Jacobi Iterative Algorithm	Shows migrating a CUDA project that uses CUDA APIs in kernel function to access GPU low-level hardware • Device Memory allocation and copy, kernel submission • Local Memory access, Atomic operation, CUDA Cooperative groups/warps to SYCL Sub-Groups, CUDA warp primitives to SYCL group algorithms
cuBlas Matrix Multiplication	Shows migrating a CUDA project that uses a CUDA library for computation on GPU • Device Memory allocation and copy • CUDA library migration (cuBlas to oneMKL blas)

Accessing Training Content on GitHub

- Github link to the training: CUDA to SYCL Migration
  - Multiple modules in folders
  - Each Module folder has a Jupyter Notebook file (*.ipynb) which can be viewed in Github to follow instructions
  - The module folder has sub-folder with SYCL code:
    - dpct_output – SYCLomatic output (may or may not compile)
    - sycl_migrated – SYCL code that gets functional correctness
    - sycl_migrated_optimized – SYCL code that is optimized for performance

Accessing Training Content on Intel DevCloud

- Login to Intel DevCloud: devcloud.intel.com/oneapi
  - Click "Get Started", scroll down and click on "Launch JupyterLab"

- Copy latest CUDA_To_SYCL_Migration training content:
  - In JupyterLab, File menu -> New -> Terminal
  - `cp -r /data/oneapi_workshop/CUDA_To_SYCL_Migration/ .`

- Access the training content:
  - In Jupyter, on left panel, open `CUDA_To_SYCL_Migration`
  - Open `Welcome.ipynb`, access the different modules and follow the steps

Reference Material

CUDA Development Machine – Sanity Check

- Run nvidia-smi in terminal
  - nvidia-smi
  - check that driver version is displayed, Nvidia card is detected and CUDA version is displayed

```
rkris11@holmes:~$ nvidia-smi
Mon Mar  6 19:52:26 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.86.01    Driver Version: 515.86.01    CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| 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  Tesla P100-PCIE...  Off  | 00000000:02:00.0 Off |                    0 |
| N/A   29C    P0    24W / 250W |      0MiB / 12288MiB |      3%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

- Check CUDA headers in install path
  - Default installation path: ls /usr/local/cuda/include
  - If CUDA is installed in non-default path, make a note of path, you will need it later
- Check that you are able to compile a simple CUDA code
  - nvcc test.cu

Install SYCLomatic Tool

- Go to https://github.com/oneapi-src/SYCLomatic/releases
  - Under Assets
  - Right-click and copy web link to linux_release.tgz
- On you CUDA Development machine:
  - In home directory or anywhere: mkdir syclomatic; cd syclomatic
  - wget <link to linux_release.tgz>
  - tar –xvf linux_release.tgz
  - export PATH="/home/$USER/syclomatic/bin:$PATH"
  - c2s --version

SYCLomatic Tool Usage (c2s --help)

- Migrate a single CUDA source file:
  - `c2s test.cu`
- Migrate a single CUDA source file and copy all syclomatic helper header files:
  - `c2s test.cu --use-custom-helper=all/file/api`
- Migrate a single CUDA source to a specific directory name
  - `c2s test.cu --out-root sycl_code`
- Migrate a single CUDA source with source root tree
  - `c2s test.cu --in-root ../`
- Migrate a single CUDA source with custom CUDA installation
  - `c2s test.cu --cuda-include-path /tmp/cuda/include`
- Migrate a CUDA project with makefile:
  - `intercept-build make`
  - `c2s -p compile_command.json`

test.cu

Compiling SYCL code for Intel and Nvidia targets

- Install oneAPI C++/DPC++ Compiler or Intel oneAPI Base Toolkit
- Install CUDA Plugin for oneAPI from CodePlay
- Link to Installation Instructions

- Set environment variable for using the Compiler
  - `source /opt/intel/oneapi/setvars.sh --include-intel-llvm`

- Compile SYCL for Intel CPUs/GPUs
  - `icpx –fsycl test.cpp`

- Compile SYCL for Nvidia GPUs
  - `clang++ -fsycl -fsycl-targets=nvptx64-nvidia-cuda test.cpp`

Learn Basics of SYCL Programming

- SYCL 2020 Specification
- Data Parallel C++ Book
- SYCL Academy from CodePlay
- Guided learning path with code samples (Jupyter Notebooks):
  - SYCL Essentials
  - SYCL Performance Portability
- C++ SYCL code samples

Optimizing SYCL code for Intel GPUs

- Refer to [Intel GPU Optimization Guide]
  - Detailed guide explaining how to optimize SYCL code:
    - Thread Mapping and GPU Occupancy Calculation.
    - Memory allocation and transfer optimization when using Buffers or Unified Shared memory.
    - Kernel code optimization – Local memory, Sub-Groups, Atomics, Reduction and more.
    - Using libraries for offload
    - Debugging and Profiling

CUDA to SYCL Migration Portal

- One Stop Portal for CUDA to SYCL Migration
  - Industry Examples of CUDA Migration to SYCL
  - Learn How to Migrate Your Code
  - Guided flow of migrating from CUDA to SYCL
  - Get all the tools and resources necessary for migrating from CUDA to SYCL

CUDA to SYCL Migration – Lab Exercise

Migrate histogram from CUDA Samples to SYCL

- CUDA source
  - https://github.com/NVIDIA/cuda-samples/Samples/2_Concepts_and_Techniques/histogram

1. Install SYCLomatic
2. Migrate CUDA source to SYCL source using SYCLomatic tool
3. Compile and Run the migrated SYCL source
4. Run SYCL code on CPUs and GPUs

CUDA to SYCL Migration

STEP 1 - CUDA Development Machine

- Clone the CUDA source
  - git clone https://github.com/NVIDIA/cuda-samples.git
  - cd cuda-samples/Samples/2_Concepts_and_Techniques/histogram
- Install SYCLomatic
- Use SYCLomatic Tool to migrate CUDA source to SYCL source
  - make clean
  - intercept-build make
  - c2s -p compile_commands.json --in-root ../../.. --use-custom-helper=all

CUDA to SYCL Migration

STEP 2 - Intel Developer Cloud

- Copy dpct_output directory from CUDA Machine to Intel DevCloud
- Compile and Run SYCL source
  - icpx -fsycl -I ../../../Common -I ../../../include *.cpp
  - ./a.out
- Access GPUs on Intel DevCloud
  - qsub -I -l nodes=1:gen9:ppn=2
  - qsub -I -l nodes=1:gen11:ppn=2