

CUDA to SYCL Migration Introduction

Benefits of migrating CUDA source to C++ SYCL source



Learning Objectives

- Explain the benefits of migrating CUDA codebase to SYCL
- Explain the advantages of programming using C++ with SYCL
- Explain what SYCLomatic project is about
- Install SYCLomatic tool on CUDA development machine
- Execute the most used command-line options of SYCLomatic tool

What is CUDA?

What is CUDA?

CUDA is a parallel computing platform and programming model created by NVIDIA.

CUDA helps developers speed up their applications by harnessing the power of NVIDIA GPU accelerators.

<https://blogs.nvidia.com/>

Why Migrate to SYCL?

What is SYCL?

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

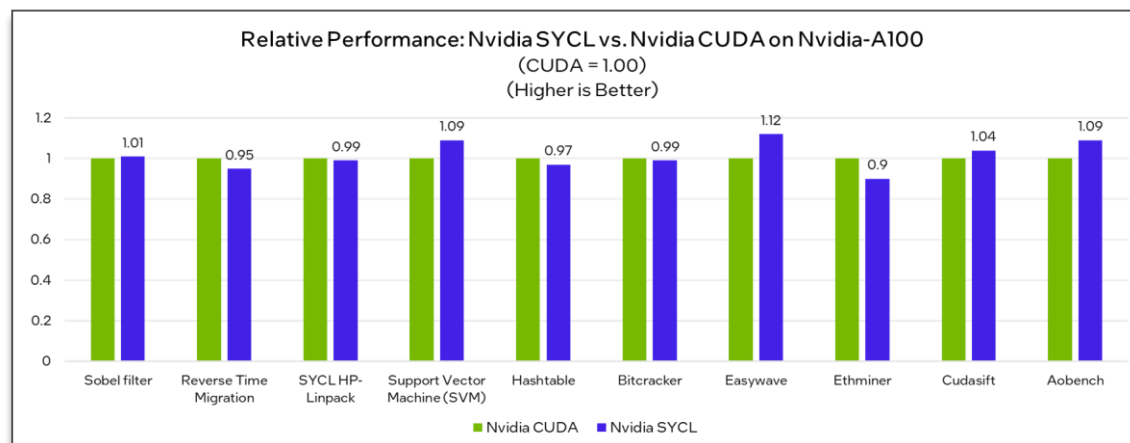
- Enables code for **heterogeneous** and offload processors to be written using modern ISO C++ (at least C++ 17).
- Provides APIs and abstractions to find devices (e.g. CPUs, GPUs, FPGAs) on which code can be executed, and to manage data resources and code execution on those devices.

<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: -fsycl-targets=nvptx64-nvidia-cuda, NVIDIA NVCC compiler switches: -O3 -gencode arch=compute_80,code=sm_80. Represented workloads with Intel optimizations.

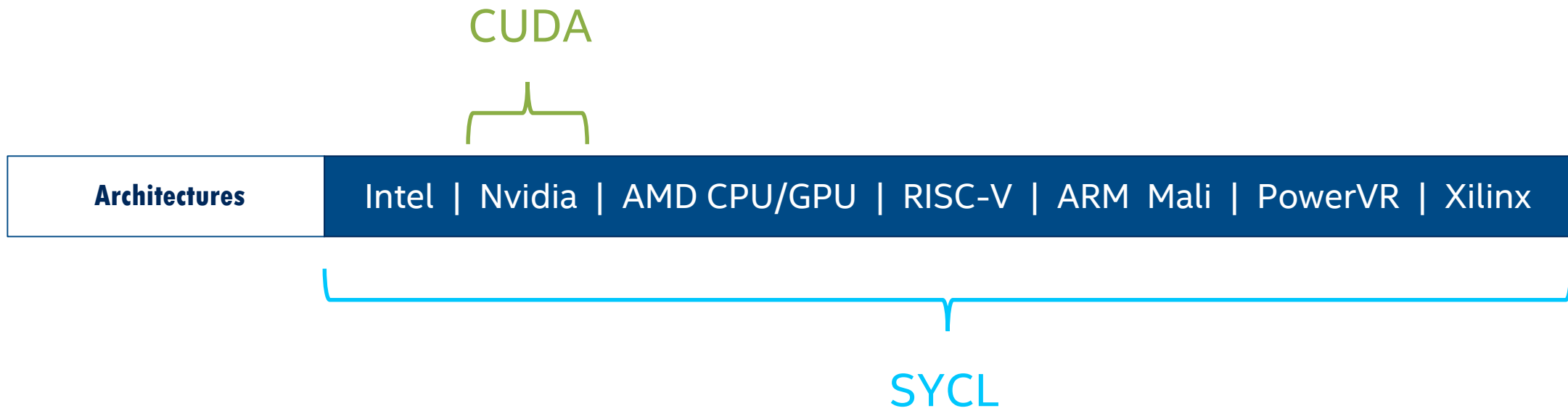
Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

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



SYCLomatic for migrating CUDA to SYCL

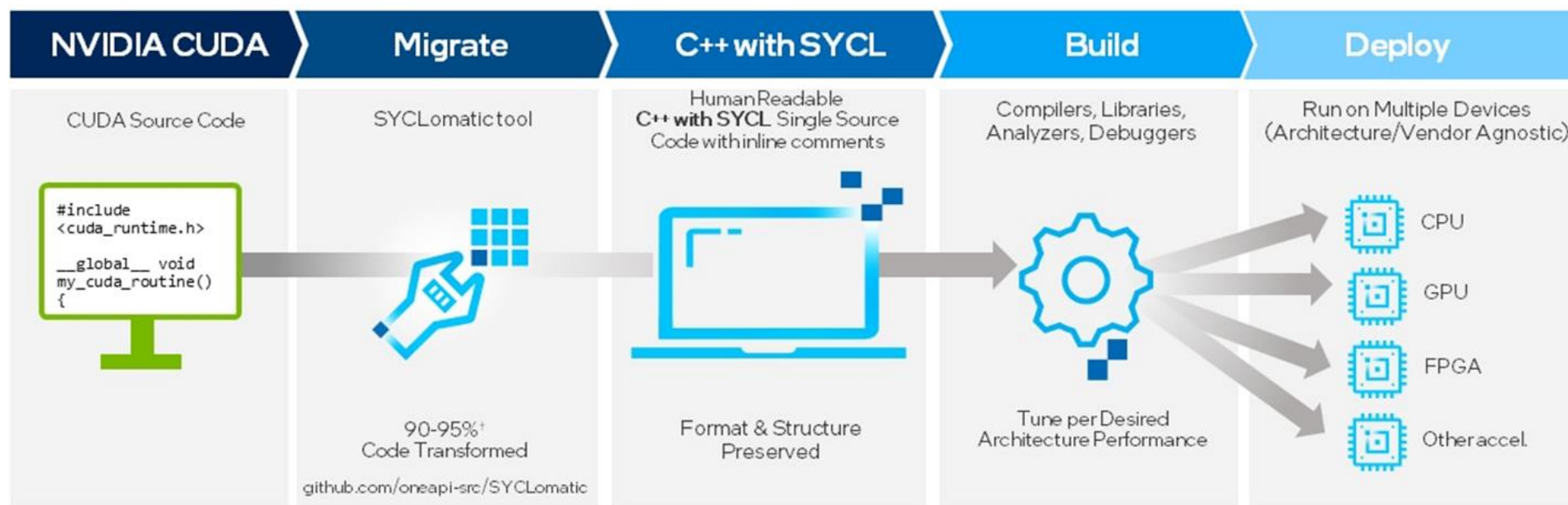
SYCLomatic

- **SYCLomatic** is an open-source project to assist developers in migrating their existing code written in different programming languages to the SYCL* C++ heterogeneous programming model.
- **SYCLomatic** tool will migrate existing CUDA codebases to SYCL, which is an industry standard. Once code is migrated to SYCL, it can be compiled and executed by any compiler that implements the SYCL specification.

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

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 automatically¹

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

¹Intel 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.

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

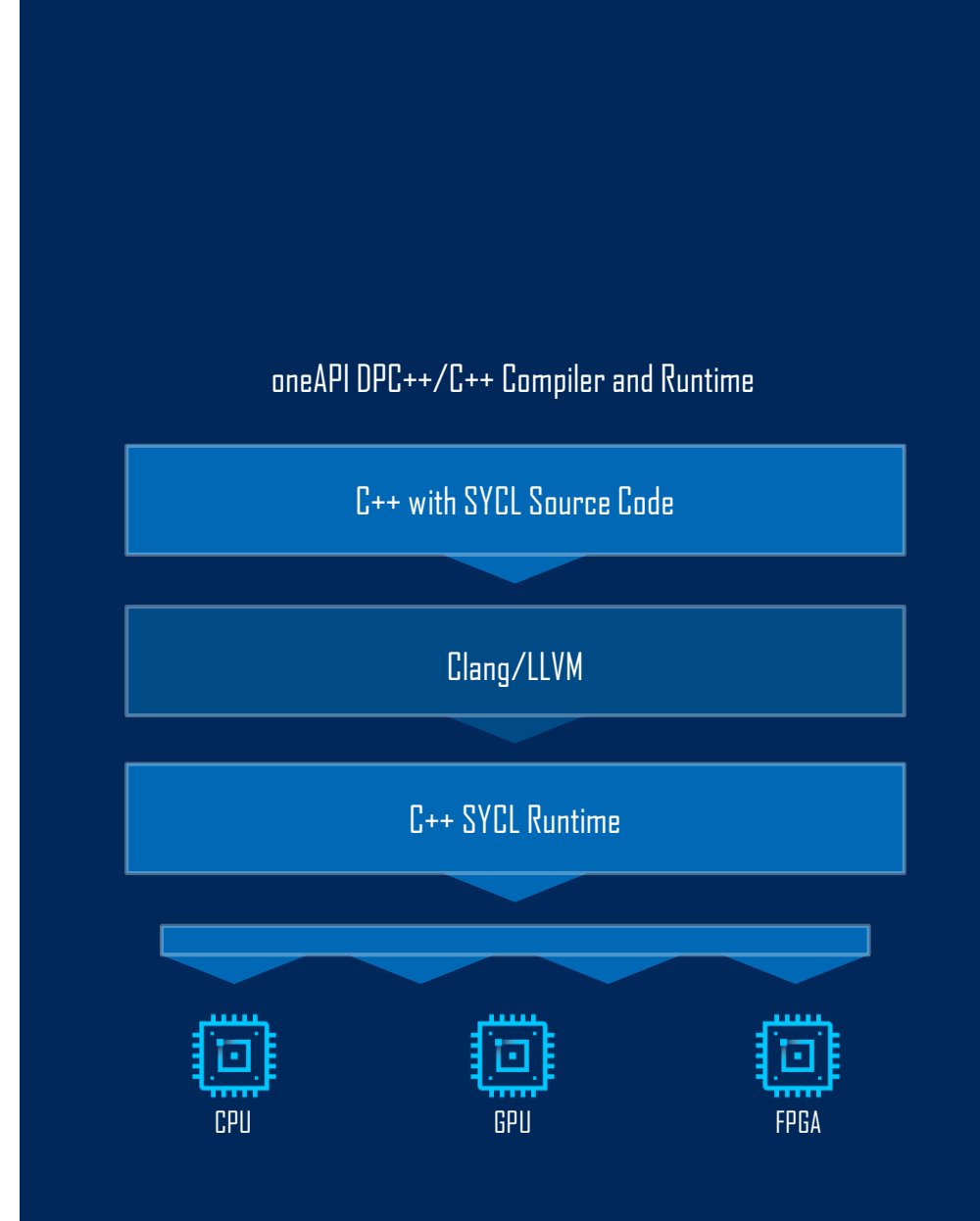
Khronos SYCL Standard

- 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](#)

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 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 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. |
|=====+=====+
| 0 Tesla P100-PCIE...    Off          | 00000000:02:00.0 Off |                    0 |
| N/A   29C    P0           24W / 250W   | 0MiB / 12288MiB |      3%    Default  |
|=====+=====+
|                                  |          MIG M.     |
+-----+-----+

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`
- Migrate a single CUDA source to a specific directory name
 - `c2s test.cu --out-root sycl_code`
- 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` OR `c2s -p .`
- Migrate a CUDA project with Makefile and generate Makefile for building SYCL code :
 - `c2s -p . --gen-build-script`
- Migrate a single CUDA source with source root tree (include dependency path)
 - `c2s -p . --in-root ../..`

Compiling SYCL code for Intel, NVIDIA, AMD

- Install oneAPI C++/DPC++ Compiler or Intel oneAPI Base Toolkit
- Install oneAPI Plugin for NVIDIA GPU support
- [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
 - `icpx -fsycl -fsycl-targets=nvptx64-nvidia-cuda test.cpp`
- Compile SYCL for AMD GPUs
 - `icpx -fsycl -fsycl-targets=amdgc-n-amd-amdhsa test.cpp`

CUDA to SYCL Migrated Code Comparison

```
#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;
}
```

Header file

Kernel function

Device Memory Allocation

Copy host to device

Submit Kernel Task

```
#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
<i>Header File</i>	<code>#include <cuda_runtime.h></code>	<code>#include <sycl/sycl.hpp></code>
<i>Create a CUDA stream</i>	<code>cudaStream_t stream1;</code>	<code>sycl::queue q;</code>
<i>Allocate memory on device</i>	<code>cudaMalloc()</code>	<code>sycl::malloc_device()</code>
<i>Memset on</i>	<code>cudaMemsetAsync()</code>	<code>q.memset()</code>
<i>Memcpy from host to device</i>	<code>cudaMemcpyAsync()</code>	<code>q.memcpy()</code>
<i>Submit kernel to device</i>	<code>kernel_function<<<NUM_BLOCKS, N>>>();</code>	<code>q.parallel_for(sycl::nd_range<1>(N, WG_SIZE) [=](sycl::nd_item<1> item){ kernel_function(); });</code>
<i>Free device allocation</i>	<code>cudaFree()</code>	<code>sycl::free()</code>
<i>Synchronize host and device</i>	<code>cudaStreamSynchronize(stream)</code>	<code>q.wait()</code>

CUDA to SYCL Migration

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

CUDA to SYCL Migration Training Material

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

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 - Workshop Examples

CUDA Code	CUDA to SYCL Migration Details
VectorAdd	<p>Shows migrating a simple single source CUDA code</p> <ul style="list-style-type: none">• Device Memory allocation and copy, kernel submission
SortingNetworks	<p>Shows migrating a CUDA project with multiple CUDA source files</p> <ul style="list-style-type: none">• Device Memory allocation and copy, kernel submission
Jacobi Iterative Algorithm	<p>Shows migrating a CUDA project that uses CUDA APIs in kernel function to access GPU low-level hardware</p> <ul style="list-style-type: none">• 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	<p>Shows migrating a CUDA project that uses a CUDA library for computation on GPU</p> <ul style="list-style-type: none">• Device Memory allocation and copy• CUDA library migration (cuBlas to oneMKL blas)

Summary

- Why migrate from CUDA to SYCL?
 - Open, standards-based
 - Multiarchitecture performance portability
 - Freedom from vendor lock-in
 - Extension of widely used C++ language
- SYCLomatic:
 - Is open-source tool to assist with migration from CUDA to SYCL
 - Must be installed on a CUDA Development machine
 - Will migrate CUDA codebase to SYCL

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.