SYCLomatic compatibility library: Making Migration to SYCL Easier

Andy Huang, Software Engineer
April 2023
Notices & Disclaimers

All product plans and roadmaps are subject to change without notice.

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

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

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

Results have been estimated or simulated.

No product or component can be absolutely secure.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

© Intel Corporation. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

Other names and brands may be claimed as the property of others.

SYCL is a trademark of the Khronos Group Inc.
Agenda

• The Background of SYCLomatic
• Design Philosophy
• Addressing Semantic Difference
  • Accessibility of sycl::queue
  • Pointer-like memory operations for targets, which don’t support USM
  • Interface to fetch image
• Compatible APIs
  • Atomic operations
  • Utility function for memory allocation
  • Utility function for 2D/3D memory operation
  • Compatible APIs to popular CUDA libraries
• Summary / Call to Action
Background of SYCLomatic

- Collect compilation options of the Developer’s CUDA* source from project build scripts, eg. Makefile, vcxproj file

- **Assist** developers migrating code written in CUDA to SYCL* by generating SYCL code wherever possible

- Typically, 90%-95%+ of CUDA code automatically migrates to SYCL code

- Inline comments are provided to help developer complete and tune the code

* Other names and brands may be claimed as the property of others.

+ 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.
Design Philosophy

• Assisting the migration of SYCLomatic through addressing
  • Difference in language API design
  • Difference in runtime/library API design

• Friendly interface for developers
  • Can be used as a standalone library without SYCLomatic

• Performance Aspirations
  • To minimize the performance impact caused by the compatibility library APIs
  • To leverage the performance benefit of SYCL runtime and SYCL library

• Maintainability
  • Keeping backward compatibility
  • Targeting reusable class/API design
Addressing Semantic Difference – sycl::queue

Difference

- Missing context to record the device selection in the current thread
  - Programmer needs to select the device every time before getting a queue

- No default sycl::queue is available in sycl::device
  - Programmer needs to passing the created queue around the host functions

- No single API call to synchronize all queues on a device

Solution

- Singleton class dev_mgr
  - Keeping a map to record the thread’s tid and the selected device

- A class device_ext for each device
  - The “default queue”
  - Recording all the created queue in the device
Addressing Semantic Difference – sycl::queue (example)

```c
__global__ void kernel_foo() {} 

int foo() {
    kernel_foo<<<1,1,0>>>();
}

int foo2() {
    cudaSetDevice(1);
}

void kernel_foo() {}

int foo() {
    dpct::get_default_queue() parallel_for(
        sycl::nd_range<3>(sycl::range<3>(1, 1, 1),
        sycl::range<3>(1, 1, 1),
        [=](sycl::nd_item<3> item_ct1) {
            kernel_foo();
        });
}

int foo2() {
    dpct::select_device(1);
}
```
Addressing Semantic Difference –
Pointer-like memory operations for targets, which don’t support USM

Difference
• Pointer-like operations are used by CUDA programmers

Solution
• Singleton class `mem_mgr`
  • Creating a “virtual” pointer for each device memory allocation
  • Providing a function to retrieve the accessor from a “virtual pointer”

```c
int foo() {
    float *h_A = (float *)malloc(size);
    float *d_A = NULL;

    cudaMalloc((void **)&d_A, 100);
    cudaMemcpyAsync(d_A, h_A, size, cudaMemcpyHostToDevice);
}
```

```c
int foo() {
    float *h_A = (float *)malloc(size);
    float *d_A = NULL;

    d_A = (float *)dpct::dpct_malloc(100);
    dpct::async_dpct_memcpy(d_A, h_A, size,
                            dpct::host_to_device);
}
```
Addressing Semantic Difference – Flexible interface to fetch Image data

Difference

- CUDA workflow:
  - Allocating device memory
  - Creating texture
  - Binding a texture to the memory

- SYCL image workflow:
  - The memory is allocated when `sycl::image` is constructed
  - The format, dimension and pitch of the image cannot be changed

Solution

- When migrating `cudaBindTexture()`
  - Recording the device pointer, dimension and channel info into an `image_wrapper`

- Lazy constructing the `sycl::image` base on the info in the `image_wrapper` when needed
Addressing Semantic Difference – Flexible interface to fetch Image data (example)

```cpp
static texture<float4, 2> tex42;
__global__ void kernel() {
  float4 f42 = tex2D(tex42, 1.0f, 1.0f);
}

int foo() {
  float4 *d_data42;
  auto tex42_ptr = &tex42;
  cudaMalloc(&d_data42, sizeof(float4) * 32 * 32);

  cudaBindTexture2D(0, tex42_ptr, d_data42, sizeof(float4) * 32 * 32);

  kernel<<<1, 1>>>();
}
```

```cpp
dpct::image_wrapper<sycl::float4, 2> tex42;
void kernel(dpct::image_accessor_ext<sycl::float4, 2> tex42) {
  sycl::float4 f42 = tex42.read(1.0f, 1.0f);
}

int foo() {
  dpct::device_ext &dev_ct1 = dpct::get_current_device();
  sycl::queue &q_ct1 = dev_ct1.default_queue();
  sycl::float4 *d_data42;
  auto tex42_ptr = &tex42;
  d_data42 = (sycl::float4 *)sycl::malloc_device(sizeof(sycl::float4) * 32 * 32, q_ct1);
  tex42_ptr->attach(d_data42, 32 * sizeof(sycl::float4), 32, 32 * sizeof(sycl::float4), tex42.get_channel());

  q_ct1.submit([&](sycl::handler &cgh) {
    auto tex42_acc = tex42.get_access(cgh);
    auto tex42_smpl = tex42.get_sampler();
    cgh.parallel_for(
      sycl::nd_range<3>(sycl::range<3>(1, 1, 1), sycl::range<3>(1, 1, 1)),
      [=](sycl::nd_item<3> item_ct1) {
        kernel(
          dpct::image_accessor_ext<sycl::float4, 2>(tex42_smpl, tex42_acc));
      });
  });
```
Compatible APIs – Free functions for atomic operation

Difference

- In SYCL 2020, atomic operations require 2 steps:
  - Constructing an atomic_ref
  - Performing the required operation on the created atomic_ref

Solution

- Free functions to wrap the 2 steps in a single function call

```c
_device__ void addByte(unsigned int *s_WarpHist, 
                       unsigned int data) {
    atomicAdd(s_WarpHist + data, 1);
}

/device__ void addByte(unsigned int *s_WarpHist, 
                       unsigned int data) {
    dpct::atomic_fetch_add<sycl::access::address_space::generic_space>(
        s_WarpHist + data, 1);
}
```
Compatible APIs – Utility Classes to simplify device memory allocation

Difference

- SYCL does not provide features to declare static/global variable for device

Solution

- Class `constant_memory` to recording the dimension/default value
  - Allocate device memory only when needed
  - Create accessor only when needed

```cpp
__constant__ int t1;
__constant__ float t2[4][5];
__global__ void kernel() {
    int a = t1;
}
int foo() {
    kernel<<<1, 1>>>();
}

int foo() {
    dpct::get_default_queue().submit([&](sycl::handler &cgh) {
        t1.init();
        auto t1_ptr_ct1 = t1.get_ptr();
        cgh.parallel_for(
            sycl::nd_range<3>(sycl::range<3>(1, 1, 1), sycl::range<3>(1, 1, 1)),
            [=](sycl::nd_item<3> item_ct1) {
                kernel(*t1_ptr_ct1);
            });
    });
}
Compatible APIs – 2D and 3D Memory Operations

Difference
• SYCL does not provide function to allocate/copy/set 2D or 3D memory
• Cannot copy to certain range like cudaMemcpy2DAsync()

Solution
• Adding free functions to
  • Handling pitch size during allocation
  • Recording pitch information
  • Provide copy to range feature

```c
int foo() {

    int size = 10 * sizeof(float);
    int pitch_des = size, pitch_src = size;
    int width = size, height = size;
    float *h_A = (float *)malloc(size);
    float *d_A = NULL;

    cudaMalloc((void **)&d_A, size);
    cudaMemcpy2DAsync(d_A, pitch_des, h_A, pitch_src, width, height, cudaMemcpyHostToDevice, cudaStreamDefault);
}
```

```c
int foo() {

dpct::device_ext &dev_ct1 = dpct::get_current_device();
sycl::queue &q_ct1 = dev_ct1.default_queue();

    int size = 10 * sizeof(float);
    int pitch_des = size, pitch_src = size;
    int width = size, height = size;
    float *h_A = (float *)malloc(size);
    float *d_A = NULL;

d_A = (float *)sycl::malloc_device(size, q_ct1);
    dpct::async_dpct_memcpy(d_A, pitch_des, h_A, pitch_src, width, height, dpct::host_to_device);
}
```
Compatible APIs –
Compatible APIs for popular CUDA libraries

Difference

• Libraries which provide similar feature may have quite different API design concept

• For example
  • curand(CUDA) workflow: curandGenerator_t can set generator type dynamically after been constructed
  • oneapi::mkl::rng(Intel® oneAPI ) workflow: The type of generator cannot be changed after construction

Solution

• Adding utility class/functions for different cases

• In the case of curand,
  • Adding template class which take generator type as a pointer and the class is derived from a non-template base class
  • Using the base class to migrate curandGenerator_t
  • Creating new mkl generator and updating the pointer when changing the generator

• Current supported libraries:
  • BLAS, CCL, DNN, STL algorithm, FFT, Rand
Summary / Call-to-Action

• The compatibility library simplify the auto migration process of SYCLomatic

• The friendly API design can help developers to create SYCL-based projects with less effort

• Future work
  • Trying to promote some APIs to SYCL spec/extension
  • Improving coverage of popular libraries
  • Providing more detailed spec of the compatibility library

• Call for contribution: SYCLomatic & SYCLomatic test
More Resources

- **SYCLomatic Project** on GitHub: [GetStartedGuide.md](#), [Contributing.md](#) guide
- Get started developing
  - *Book*: Mastering Programming of Heterogeneous Systems using C++ & SYCL
  - *Essentials of SYCL training*
  - *The oneAPI samples* on Github
- **oneAPI specification** and **SYCL** specification
- **Intel® oneAPI Toolkits**
- **Intel® DevCloud** - A free environment to access Intel® oneAPI Tools and develop and test code across a variety of Intel® architectures (CPU, GPU, FPGA)
- CodeProject: Using oneAPI to convert CUDA code to SYCL