Migrating from CUDA-only to Multi-Platform DPC++

Steffen Christgau

Supercomputing Department
Zuse Institute Berlin

IXPUG Technical Webinar
February 18, 2021

Agenda

Motivation

A Look on easyWave

Migrating from CUDA to oneAPI... and being more SYCLesque

Using Different Architectures

Conclusion

Table of Contents

Heterogeneous Platforms

- continuing trend for **heterogeneous architectures**
  - energy effiency + computational power
  - special purpose architectures
  - examples: GPUs, ASICs (like TPUs), Vector Processors, FPGAs

- challenge: **programming environment** for heterogeneous platforms
  - proprietary languages and/or standards bound to vendor (CUDA, OpenACC)
  - "exotic" ways of programming (VHDL/Verilog)
  - different code for supporting different devices (of the same class)

CUDA code

```
myKernel<<<blocks, threads>>>(...)
```

1:1 →

image source: https://www.nvidia.com/en-us/data-center/v100/

oneAPI

- **oneAPI** – "common developer experience across accelerator architectures"
  - **Data Parallel C++** as programming language of choice → **based on SYCL**
  - expectation: single code for different platforms, like CPUs and accelerators

- **challenge:** Migrate existing codes targeting single hardware platforms.

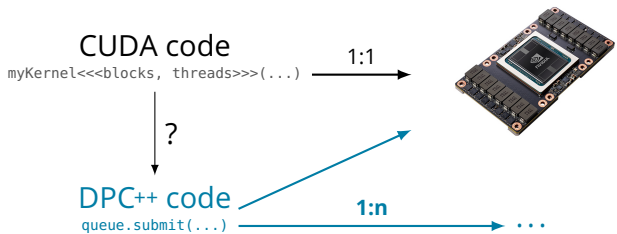
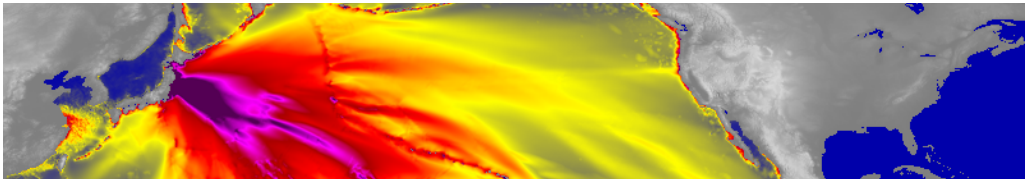


Table of Contents

Case Study: easyWave

- German Research Center for GeoSciences (GFZ) and University of Potsdam
  - tsunami simulation: arrival times and wave heights in case of seismic event
  - open source: `https://gitext.gfz-potsdam.de/id2/geoperil/easyWave`
- originally written in C++: 4470 LoC
  - OpenMP and CUDA support
  - classes for programming model abstraction $\rightarrow$ different code paths
- memory bound **stencil kernels** on dynamically **growing compute domain**



easyWave: A (simplified) Closer Look – Host Side

```
KernelData data;

 /* memory allocation */
data.h = cudaMallocPitch(...);
data.cR6 = cudaMalloc(...);

 /* prepare execution environment */
dim3 threads = (32, 16);
dim3 blocks = ( NX / threads.x, NY / threads.y );

 /* start kernel and measure execution time */
cudaEventRecord( evtStart, 0 );
runWaveUpdateKernel<<<blocks,threads>>>( data );
cudaEventRecord( evtEnd, 0 );

 /* launch other kernels */

cudaDeviceSynchronize();
cudaEventElapsedTime(time, evtStart, evtEnd);
```

1D and 2D memory allocation

thread/block-oriented kernel execution

event-based time measurement

function-call-like kernel launch

easyWave: A (simplified) Closer Look – Device Side

```
__global__ void runWaveUpdateKernel( KernelData data ) {

  Params& dp = data.params;
  int i = blockIdx.y * blockDim.y + threadIdx.y + dp.iMin;
  int j = blockIdx.x * blockDim.x + threadIdx.x + dp.jMin;
  int ij = data.idx(i,j);

  data.h[ij] = data.h[ij] - data.cR1[ij] * ( data.fM[ij] - data.fM[data.le(ij)] +
               data.fN[ij] * data.cR6[j] - data.fN[data.dn(ij)]*data.cR6[j-1] );
```

pointer-based data access

```
  if( data.h[ij] > data.hMax[ij] ) data.hMax[ij] = data.h[ij];

  /* ... */
}
```

Table of Contents

Bringing CUDA code to DPC++/oneAPI

- two possible strategies
  1. manually adjust code to SYCL/DPC++
  2. tool-supported migration → Intel DPC++**Compatibility Tool** (dpct)

- dpct provides **convenient migration assistance**
  - included in oneAPI Base Toolkit
  - input: CUDA-flavoured project code
  - output: migrated code → **still readable + maintainable**
  - only CUDA-related parts touched
  - no manual and tedious boilerplate/syntax changes
  - migrated code good starting point for further development + optimizations

Migration Process

- `intercept-built` *`make`* to generate *compilation database*
  - records compiler commands, files and flags → `compile_commands.json`
  - requires CUDA headers to be present in your system (check supported version)

- compilation database = input for actual migration:
  - ```
    dpct -p path/to/compile_commands.json --in-root=cuda_code \
    --out-root=migrated_code
    ```

- migration also works for individual files (no compilation database required)

- next slides: detailed view on differences between migrated and CUDA code (based on easyWave migration)
    ```

What's different: Device Handling

- CUDA: implicit current GPU (and only GPU)
  - use `cudaSetDevice` to change device
  - not employed in easyWave (and most other codes)

- SYCL/DPC++: explicit usage of devices, usually by means of a *queue*
  - select queue of a specific **device (of arbitrary type)**
  - memory management and kernel launch almost always done using queue
  - code generated by Compatibility Tool

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

  - control (default) device with `SYCL_DEVICE_FILTER` environment variable or SYCL device selector

What's different: Memory Handling

- CUDA: explicit allocation and transfer (if not using *CUDA Unified Memory*)
  - pointers for device data
  - allocate from host code, pass returned pointer to kernels

```
cudaMalloc( &device_ptr, data_count * sizeof(float) );
```

- SYCL/DPC++: **Unified Shared Memory** (USM) – new in SYCL 2020!
  - pointers and explicit memory transfers – associated with *queue*
  - allocate from host code, pass returned pointer to kernels (shared addresses)

```
device_ptr = sycl::malloc_device<float>(data_count, queue);
```

  - convenient for CUDA/C/C++-familiar developers → **pointers in kernels**
  - different from OpenCL-inherited and tedius-to-use *buffers+accessors* in SYCL

What's different: Kernel Invocation and Execution

- CUDA: call implicit data-parallel kernel function + execution environment
  - asynchronous execution w.r.t. to host in a **stream** (0 by default)

```
runWaveUpdateKernel<<<blocks,threads>>>( device_ptr );
```

- SYCL/DPC++: submit **command group** to device **queue**
  - group contains (parallel) kernel invocation → asychronous kernel execution

```
queue.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(
        sycl::nd_range<N_DIMS>(...),
        [=](sycl::nd_item<N_DIMS> item) {
            runWaveUpdateKernel(kernel_data, item);
        });
    });
```

What's different: Order of Kernel Execution

- CUDA: kernels execute **in order of submission** inside same stream
  - can use events for dependencies

- SYCL/DPC++: queues are **out of order by default**
  - "old" SYCL 1.2 $\rightarrow$ Accessors enable automatic depedency detection
  - SYCL 2020 with USM: need other way
  - Compatibility Tool keeps execution order with `wait` call on queue
  - dependency specification with events possible (see later)

What's different: How many Kernel instances?

- Runtime/hardware needs information about how many resources are needed.
- CUDA: hardware-oriented → `my_kernel<<<blocks, threads_per_block>>>`
  - `threads_per_block` – number of concurrent kernel invocations in a block/SM
  - `blocks` – how many instances of `threads_per_block`
  - problem size = `threads_per_block` × `blocks`
- SYCL/DPC++: domain-oriented → `cgh.parallel_for(sycl::range<2>,..)`
  - `range` equals total problem size = number of total **work items** (CUDA's threads)
  - **work group** = group of work items (CUDA's block size)
  - `nd_range` = number of work items + work group size → more control over HW
- Be aware of hard-coded and CUDA device-dependent values (1024, e.g.).

Other Differences: Error Handling and Timing

Error Handling

- CUDA: C-fashioned check for **return values** – often coupled with macros
- SYCL/DPC++: **exception**-based (including asynchronous exceptions)
- Compatibility Tool migrates between both approaches

Timing

- CUDA best-practise: events for measuring (asynchronous) kernel execution
- SYCL/DPC++: Compatibility Tool issues code for
  1. synchronization → `wait`
  2. time measurement with C++ `std::chrono` clock

Being more SYCLesque

- Compatibility Tool is generic tool.
- For easyWave: almost no additional changes required to let code compile & run

- Migrated code still has room for improvements…
  1. memory management → switch to SYCL buffers and accessors if needed
     also enables **implicit dependencies** between kernels/command groups
  2. for USM: use events for **explicit dependencies**
  3. use events for **timekeeping**
  4. make kernel invocation less verbose
  5. query device properties for work group size
  6. replace functions from dpct headers, e.g. for 2D memory allocation, with SYCL pendant (if possible) …

Being more SYCLesque: Example

Starting Point: Sequence of depending kernels

```
queue.submit(...).wait();

auto ts_start = std::chrono::high_resolution_clock::now();

queue.submit([&](sycl::handler &cgh) {

    cgh.parallel_for(
    sycl::nd_range<2>(problem_size, workgroup_size),
        [=](sycl::nd_item<2> work_item) {
            runWaveUpdateKernel(kernel_data, work_item);
        });
}).wait();

auto ts_end = std::chrono::high_resolution_clock::now();
```

Being more SYCLesque: Example

Refactoring 1: Use returned event to define dependencies.

```
1  auto kernel_dependency = queue.submit(...);
2
3  auto ts_start = std::chrono::high_resolution_clock::now();
4
5  auto event = queue.submit([&](sycl::handler &cgh) {
6      cgh.depends_on(kernel_dependency);
7      cgh.parallel_for(
8      sycl::nd_range<2>(problem_size, workgroup_size),
9          [=](sycl::nd_item<2> work_item) {
10             runWaveUpdateKernel(kernel_data, work_item);
11         });
12 });
13 event.wait(); /* wait for kernel to finish */
14 auto ts_end = std::chrono::high_resolution_clock::now();
```

Being more SYCLesque: Example

Refactoring 2: Use event for timekeeping.

```
1  auto kernel_dependency = queue.submit(...);
2
3  auto event = queue.submit([&](sycl::handler &cgh) {
4      cgh.depends_on(kernel_dependency);
5      cgh.parallel_for(
6      sycl::nd_range<2>(problem_size, workgroup_size),
7          [=](sycl::nd_item<2> work_item) {
8              runWaveUpdateKernel(kernel_data, work_item);
9          });
10 });
11 event.wait(); /* wait for kernel to finish */
12 auto dt = event.get_profiling_info<cl::sycl::info::event_profiling::command_end
```

Hint: A queue with enabled profiling infos must be used here.

Being more SYCLesque: Example

Shorten kernel invocation

```
auto kernel_dependency = queue.submit(...);

auto event = queue.parallel_for(
    sycl::nd_range<2>(problem_size, workgroup_size), kernel_dependency,
    [=](sycl::nd_item<2> work_item) {
        runWaveUpdateKernel(kernel_data, work_item);
    });
});
event.wait(); /* wait for kernel to finish */
auto dt = event.get_profiling_info<cl::sycl::info::event_profiling::command_end
```

Table of Contents

Using Intel Processors and Graphics Devices

- possible targets:
  - Intel CPUs (Core, Xeon)
  - GPUs (Gen9, Gen12/XeLP/DG1)
- compilation as usual: just use `dpcpp`
- For execution: **use same binary**, use `SYCL_{BE|DEVICE_TYPE}` to switch device[1]

```
$ export SYCL_BE=PI_OPENCL

$ SYCL_DEVICE_TYPE=GPU ./easyWave ...
using Intel(R) Graphics [0x5916]

$ SYCL_DEVICE_TYPE=CPU ./easyWave ...
using Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
$
```

[1] use `SYCL_DEVICE_FILTER` in future releases

Using Intel FPGAs

- Compilation: use `dpcpp` with some flags
  - Compilation: `-fintelfpga [-fsycllink]`
  - Link: `-fintelfpga -Xshardware -Xsboard=intel_s10sx_pac:pac_s10`
  - …have time

- For execution: basically just run the binary → no need for bitstream handling, but limits advanced usage currently
- Board Support Package for OpenCL/oneAPI required

- Save time with the FPGA emulation
  - runs on CPU, at CPU speed (likely to be native SYCL execution)
  - be aware: emulation might differ from real FPGA → data types, atomics, …
  - refer to documentation for details

- **great opportunity** for bringing FPGAs to the masses, but needs code adaption

Reusing your CUDA hardware

- remember: DPC++ = SYCL + extensions
- migrating to new platfrom "only" requires SYCL implementation

- CodePlay contributed to Open Source Intel LLVM Compiler
  - download and build Intel Open Source LLVM compiler from Github
  - pass flags to compiler, including:
    - `-fsycl`
    - `-fsycl-unnamed-lambda`
    - `-fsycl-targets=nvptx64-nvidia-cuda-sycldevice`

- run the binary with `SYCL_BE=PI_CUDA` or set `SYCL_DEVICE_FILTER`

- can still use Nvidia Tools, like profilers, debugger etc.

Table of Contents

Summary

- Compatibility Tool supports (tedios) migration from CUDA to DPC++
- much similarities between both languages, subtle differences
- manual tweaking and tuning still required for optimized/elegant code
- usage of SYCL/DPC++ pays off with gained multi-platform support

Thanks for your attention!

christgau@zib.de

Further Reading and Resources

- Compatibility Tool documentation: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compatibility-tool.html
- A Code Walk-Through for Data Parallel C++ (DPC++) Foundations: https://software.intel.com/content/www/us/en/develop/articles/dpcpp-foundations-code-sample.html
- Migrating from CUDA to SYCL: https://developer.codeplay.com/products/computecpp/ce/guides/sycl-for-cuda-developers/migration
- SYCL Standard: https://www.khronos.org/sycl/
- Intel Open Source LLVM Compiler: https://github.com/intel/llvm
- oneAPI website https://www.oneapi.com/