On The Compilation Performance of Current SYCL Implementations

Peter Thoman,
Facundo Molina Heredia and Thomas Fahringer

peter.thoman@uibk.ac.at
Background, Motivation & Goals
“SYCL is ideal for accelerating larger C++-based engines and applications with performance portability.”

https://www.khronos.org/sycl
“Larger” also means more source code

SYCL is based on C++, and specifically uses quite a bit of templated code
  • Which gives us all the nice type safety, zero-cost abstractions, etc.
  • “Zero-cost” does not apply to compile time
Motivation

- **Celerity** is a runtime system which implements a SYCL-like API on distributed memory clusters
  - Uses various SYCL implementations as back-ends (which might in turn support multiple target platforms)
  - Large integration and regression testing matrix
Motivation

- **Celerity** is a runtime system which implements a SYCL-like API on distributed memory clusters
  
  ➔ Uses various SYCL implementations as back-ends  
  (which might in turn support multiple target platforms)

  Large integration and regression testing matrix
Goals

- **Quantify** the compile-time performance of various SYCL implementations
- Analyse the impact of **individual SYCL features** on compile time
- Monitor compilation performance over (development) time
- In this work:
  1. A **code generator** for creating parameterized input SYCL code
  2. Open source tooling for **fetching and building** SYCL implementations (at some point in time), and running **statistically sound experiments** on them
  3. A quantitative evaluation on **both generated and real-world** code bases
Sycl Implementations and Code Generation
SYCL Implementations

- Many SYCL implementations available
- How to select for this study?
  - We based our selection on (very imperfect) metrics for use/popularity:
    Age, mention on Khronos SYCL site, Github stars (where applicable)

<table>
<thead>
<tr>
<th></th>
<th>DPCPP</th>
<th>ComputeCpp</th>
<th>hipSYCL</th>
<th>triSYCL</th>
<th>neoSYCL</th>
<th>Sylkan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targets</td>
<td>CPUs, OpenCL+SPIR-V,</td>
<td>SPU, OpenCL+SPIR,</td>
<td>OpenMP, CUDA, ROCm</td>
<td>OpenMP, OpenCL+SPIR*</td>
<td>VEO</td>
<td>Vulkan+SPIR-V (non-compute)</td>
</tr>
<tr>
<td></td>
<td>CUDA+PTX</td>
<td>OpenCL+PTX*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reference</td>
<td>[27]</td>
<td>[29]</td>
<td>[2]</td>
<td>[7]</td>
<td>[20]</td>
<td>[31]</td>
</tr>
<tr>
<td>Github Stars</td>
<td>603</td>
<td>-</td>
<td>496</td>
<td>389</td>
<td>13</td>
<td>12</td>
</tr>
</tbody>
</table>
SYCL Implementations Evaluated

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Notes</th>
<th>Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccpp</td>
<td>ComputeCPP (spir64 BE)</td>
<td>2.4, 2.5, 2.6, 2.7, 2.8</td>
</tr>
<tr>
<td>dpcpp_s</td>
<td>Data-parallel CPP (spir64 BE)</td>
<td>2022-01-13, 2021-11-14, 2021-09-15</td>
</tr>
<tr>
<td>dpcpp_n</td>
<td>Data-parallel CPP (CUDA BE)</td>
<td>2022-01-13, 2021-11-14, 2021-09-15</td>
</tr>
<tr>
<td>dpcpp_ns</td>
<td>Data-parallel CPP (both BEs)</td>
<td>2022-01-13, 2021-11-14, 2021-09-15</td>
</tr>
<tr>
<td>triSYCL</td>
<td>TriSYCL (C++20 BE)</td>
<td>2022-01-12</td>
</tr>
</tbody>
</table>

- Versions *usually* spaced 60 days apart – some exceptions:
  - hipSYCL: 2021-11-14 not stable
  - Dpcpp: build system changes before 2021-09
  - ComputeCPP: use numbered releases
Code Generation - Goals

- **Targeting** of individual SYCL features
- Broad **compatibility** with SYCL implementations
- Guarding against undesired optimization
- Arbitrary **scaling** of parameters
- Composability
Code Generation – Relevant Parameters

- **Buffer Num**: varying how many buffers are accessed
- **Capture Num**: varying the number of variables captured
- **Dimensions**: varying the buffer and work dims operated on from 1 to 3
- **Kernel Num**: varying the total number of kernels
- **Loopnests**: varying the nesting level of loops from 1 up to 6
- **Mix**: testing different instruction mixes within a kernel
  - e.g. **mad:50, cos:50** / **add:25,mad:25,cos:25,sqrt:25**
Code Generation – Sample

```
s::buffer<int, 1> buffer_1{s::range<1>(rt_size)};
s::buffer<int, 1> buffer_2{s::range<1>(rt_size)};
s::buffer<int, 1> buffer_3{s::range<1>(rt_size)};

int capture_1();
int capture_2();

device_queue.submit([&](cl::sycl::handler& cgh) {
    auto buffer_1_acc = buffer_1.get_access<s::access::mode::read_write>(cgh);
    auto buffer_2_acc = buffer_2.get_access<s::access::mode::read_write>(cgh);
    auto buffer_3_acc = buffer_3.get_access<s::access::mode::read_write>(cgh);
    cl::sycl::range<1> ndrange{rt_size};
    cgh.parallel_for<kern1>(ndrange, [=](cl::sycl::id<1> gid) {
        buffer_1_acc[gid] += capture_1 + capture_2;
        buffer_1_acc[gid] += buffer_1_acc[gid] + buffer_2_acc[gid] + buffer_3_acc[gid];
        for(int i0 = 0; i0 < buffer_2_acc[gid]; ++i0) {
            for(int i1 = 0; i1 < buffer_3_acc[gid]; ++i1) {
                buffer_1_acc[gid] = cl::sycl::cos(buffer_2_acc[gid]);
                buffer_1_acc[gid] = buffer_2_acc[gid] * buffer_3_acc[gid] + buffer_2_acc[gid];
            }
        }
    }); // parallel_for
}); // submit
```

compiletime_gen.rb -k 1 -b 3 -c 2 -d 1 -l 2 -t int -m cos:1,mad:1
Evaluation
Experiment Setup

Hardware / Software

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x AMD EPYC 7282 16 Core *</td>
</tr>
<tr>
<td>Memory</td>
<td>256 GB DDR4-3200, 8 channel</td>
</tr>
<tr>
<td>Storage</td>
<td>Samsung NVMe SSD SM981 #</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu Linux 20.04.2 LTS</td>
</tr>
<tr>
<td>Kernel</td>
<td>5.4.0-80-generic</td>
</tr>
<tr>
<td>Base Compiler</td>
<td>g++ 9.3.0</td>
</tr>
</tbody>
</table>

Statistics & Runs

- 48 experiments
- 20 implementations, dbg and rel (2)
- 50 runs per data point

= 96000 measurements

→ Parallelize experiments (verified minimal impact on subset)

* frequency locked at 2.8 GHz all-core
# generated output on ramdisk
Kernel Scaling

![Graph showing kernel scaling with number of kernels vs. time (10^1 to 10^3). Legend includes 'ccpp', 'dpcpp_s', 'dpcpp_n', 'dpcpp_ns', 'hipSYCL', and 'triSYCL'.]
Buffer Scaling

* dpcpp_n(s) and hipSYCL fail
Instruction Mix

![Diagram showing instruction mix for different operations and their corresponding times. The x-axis represents the type and amount of operations (e.g., mix25, add100, cos100, m50c50, mad100, sqrt100), and the y-axis represents time in seconds. Different markers and colors are used to represent various operations and libraries such as ccpp, dpcpp_s, dpcpp_n, dpcpp_ns, hipSYCL, and triSYCL.]
Release and Debug Configurations
Performance Over Time

![Performance Over Time Graph]

- Experiments:
  - mix
  - type
  - loopnest
  - dimensions
  - capture_num
  - kernel_num

- Time (s) on a logarithmic scale:
  - $10^1$
  - $10^2$
  - $10^3$

- Dates:
  - 2021-03-19
  - 2021-05-18
  - 2021-07-17
  - 2021-09-15
  - 2022-01-13
Performance Over Time - Detail

26th of May, 2021

“Accessor variant” feature branch merged
https://github.com/illuhad/hipSYCL/pull/555

→ Far-reaching changes

→ Indicates value of compile-time performance regression testing
Real-world Programs

Moderately sized
(Celerity application)

Larger overall code base
(some compatibility constraints)
Summary & Conclusion
Conclusion

• Compilation time is a non-negligible factor for large-scale SYCL adoption

• Overall, when targeting GPUs:
  • ComputeCPP performs best in terms of compilation times
  • DPCPP and hipSYCL are comparable, though DPCPP is slower with the CUDA BE

• triSYCL offers very fast compilation for CPUs

• Some implementation changes have outsized impact on compile times
  → compiler performance regression testing
Thank you for your attention!

peter.thoman@uibk.ac.at

https://github.com/PeterTh/syclcomp_utils

Contributions to this research were partially funded by the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 956137 (LIGATE project).