Exploring the possibility of a hipSYCL-based implementation of oneAPI

Aksel Alpay Bálint Soproni Holger Wünsche Vincent Heuveline
Heidelberg University

Speaker: Aksel Alpay

IWOCL '22

Introduction

oneAPI

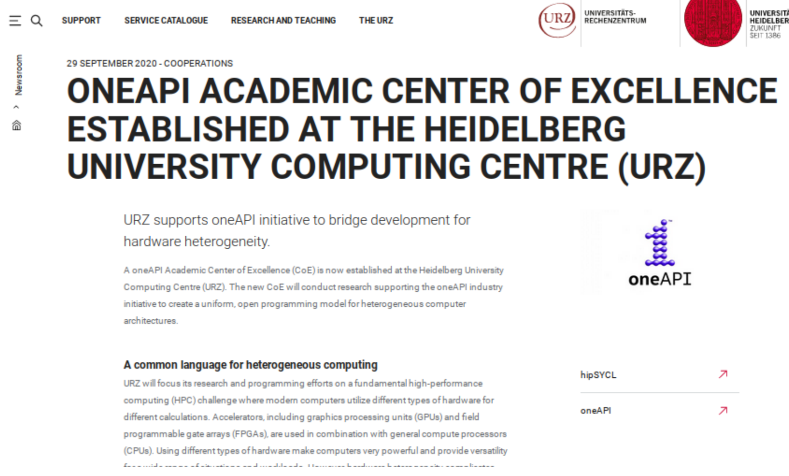
Open specification for platform built around:

- SYCL 2020
- Accelerated libraries (e.g. oneMKL, oneDPL, oneDNN, ...)
- Low-level building blocks (Level Zero)

Implementations:

- Reference implementation led by Intel using DPC++ SYCL implementation
- (Partial) port to NVIDIA led by Codeplay using DPC++ CUDA backend
- (Partial) port to AMD led by Codeplay using DPC++ HIP backend (new)

Idea: Leverage hipSYCL to run oneAPI code on AMD, NVIDIA, CPUs



https://www.urz.uni-heidelberg.de/en/newsroom/oneapi-academic-center-of-excellence-established-at-the-heidelberg-university-computing-centre-urz

Motivation

Being able to use multiple, independent compilers brings benefits:

- Users can test code with multiple compilers
- Can reveal bugs in user code and implementations
- Can reveal ambiguities in the specification

Can oneAPI be implemented with a compiler that is not derived from DPC++?

- Attempt proof-of-concept implementation with hipSYCL
- First attempt to implement oneAPI independently from DPC++

Goals

When can we conclude that oneAPI can indeed be implemented with hipSYCL?

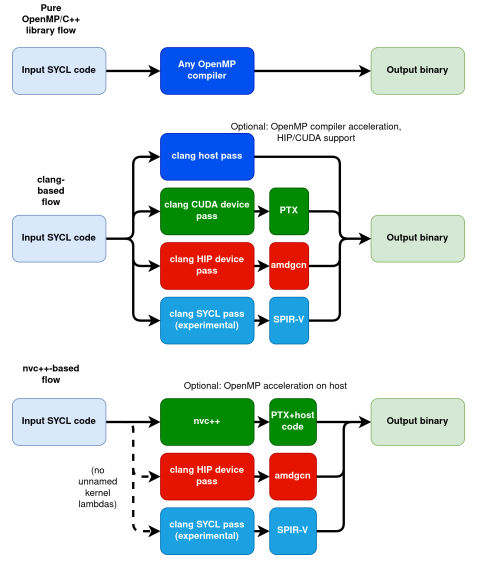
oneAPI component	Goal
Programming model	SYCL 2020 must be reasonably well supported:
	→ Common code must compile
	→ At least 80% of native performance
Libraries	oneAPI libraries must be implementable with hipSYCL.
	→ Demonstrate using oneMKL BLAS domain.
Building blocks	Level Zero must be supported
	→ Add Level Zero runtime backend

Introduction to hipSYCL

hipSYCL architecture

- Multi-backend open-source[1] SYCL implementation
- Can integrate and ride on top of various already existing toolchains
- Consists of three main components:
  1. Compiler component (`syclcc` driver + clang/LLVM components)
  2. Runtime (device management, task graph management, scheduling, ...)
  3. libkernel: Multi-backend header libarary that can be used inside kernels (math builtins, group algorithms, ...)

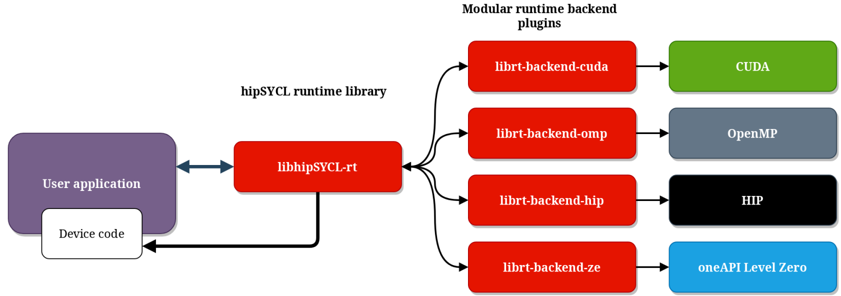
[1] `https://github.com/illuhad/hipSYCL`



Multiple kinds of compilation flows:

- Library-only (OpenMP, nvc++)
- Single-pass with compiler acceleration for CPU
- Integrated multipass
- Explicit multipass

Runtime design



- User code constructs a kernel launcher callable for each backend
- Runtime invokes kernel launcher $\rightarrow$ generic runtime can invoke kernels using language extensions (e.g. CUDA's `kernel<<<>>>()`)

Experimental Setup

Following software and hardware was used:

- hipSYCL 0.9.2 (258dc87) with clang 13
- DPC++ 2022.0.1
- ROCm 4.5
- CUDA 11.6

This work focused on NVIDIA and AMD hardware:

- AMD Radeon Pro VII
- GeForce GTX 1080Ti

Building on Level Zero

Level Zero backend

- Straight-forward to integrate in hipSYCL's runtime backend model
- Some parts of kernel library still unimplemented (group algorithms, reductions, ...)

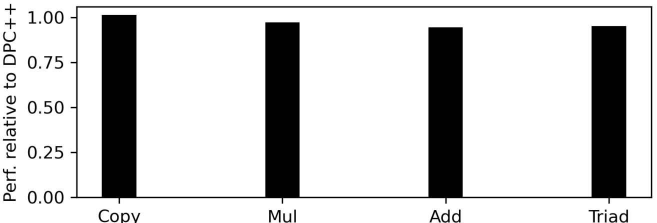


Figure: hipSYCL Level Zero BabelStream performance on Intel UHD 620 relative to DPC++ performance

Deakin T, Price J, Martineau M, McIntosh-Smith S. Evaluating attainable memory bandwidth of parallel programming models via BabelStream

Programming model: SYCL 2020

SYCL 2020 in hipSYCL

Feature	Status
Optional lambda naming	✔ (PR)
Subgroups	✔ (PR)
In-order queues	✔ (PR)
Explicit dependencies (`depends_on()`)	✔ (PR)
Backend interop API	✔ (PR)
Reductions	✔ (PR)
Group algorithms	✔ (PR)
New device selector API	✔ (PR)
Aspect API	✔ (PR)
Deduction guides	✔ (PR)
`atomic_ref`	✔ (PR)

https://github.com/hipSYCL/featuresupport

hipSYCL supports many SYCL 2020 features!

In this work, we take a closer look at key features:

- Unified Shared Memory (USM)
- Group Algorithms and sub-groups
- Optional lambda kernel naming

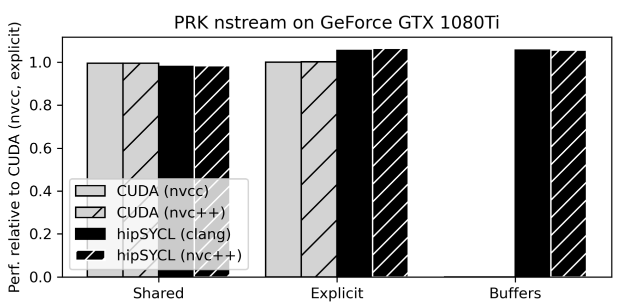
Programming model: SYCL 2020
I. Unified Shared Memory

Terminology:

- buffer-accessor model: Traditional way of managing memory in SYCL
- Explicit USM: Pointer-based, explicit data transfers necessary
- Shared USM: Data migrates automatically (see e.g. CUDA unified memory)

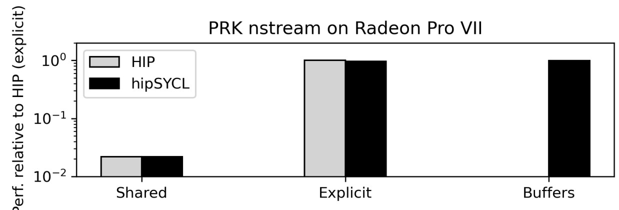
Parallel Research Kernels
USM performance

The Parallel Research Kernels[2]: Memory benchmarks for multiple models



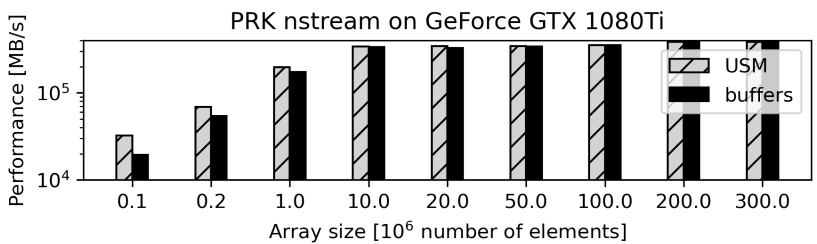
[2]Rob F. Van der Wijngaart and Timothy G. Mattson. 2014. The Parallel Research Kernels.

Parallel Research Kernels
USM on AMD



- ROCm performance with shared allocations leaves a lot to be desired (not hipSYCL problem)

USM vs buffers



- Buffers may have slight additional overhead (only noticable for short-running problems)
  - hipSYCL buffers internally use (explicit) USM pointers anyway
  - Runtime does automatic dependency analysis, data migration with buffers
  - Don't conclude that USM is always better!

Programming model: SYCL 2020
II. Sub-groups and group algorithms

Terminology:

- Sub-groups: Groups below granularity of work groups. In hipSYCL on GPU, mapped to warps/wavefronts.
- Group algorithms: Algorithmic primitives at sub-group and work group level (reductions, scans, ...)

We contribute a benchmark suite[3] to measure group algorithm perf (various algorithms, data types, supports native libraries e.g. CUB).

- Runtime of a kernel of 10^5 works groups, each with 512 invocations of work group algorithms

[3] https://github.com/DieGoldeneEnte/sycl-bench/tree/groupFunctions

Group inclusive scans (NVIDIA)

Work group size 128

Work group size 256

Runtime [s]

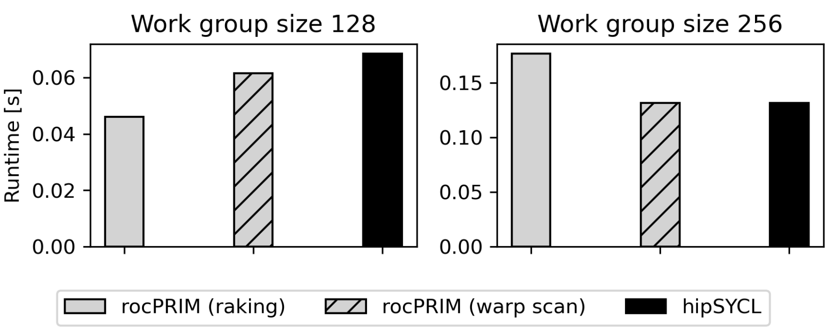
0.08
0.06
0.04
0.02
0.00

0.15
0.10
0.05
0.00

clang
nvc++
clang
nvc++

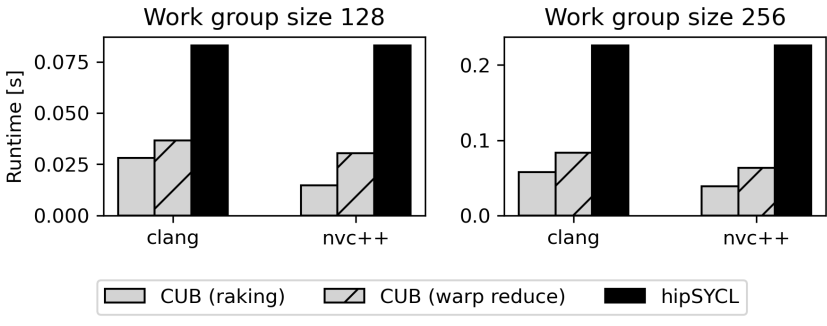
CUB (raking)
CUB (warp scan)
hipSYCL

Group inclusive scans (AMD)

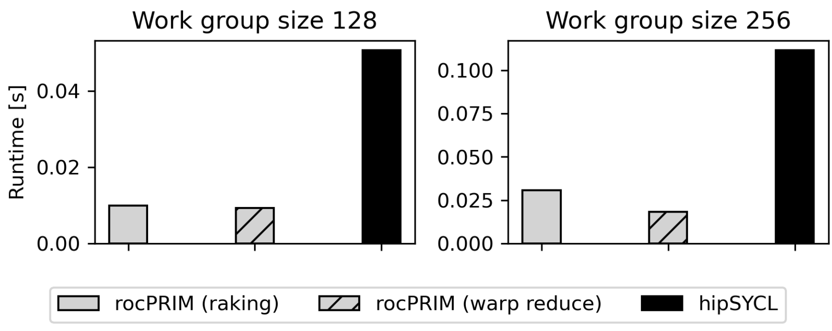


- Competitive performance for both AMD and NVIDIA

Group reductions (NVIDIA)



Group reductions (AMD)



Group reductions

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

- Reductions are significantly slower on AMD and NVIDIA
- Instruction overhead due to group sizes not being known at compile time (SYCL problem ☹)
  - JIT compiling does not work well with hipSYCL because it relies heavily on AOT compilation
  - Multi-versioning is difficult in library-only compilation flows
  - Attributes (`reqd_work_group_size`) not implementable in library-only flows

Solution: Programming model like hipSYCL's scoped parallelism which allows implementation to instantiate kernel with different group types → "multi-versioning" with pure C++ template semantics

- How much are real-world applications even dominated by group algorithm performance?

Programming model: SYCL 2020
III. Optional lambda kernel naming

Before:

```
class UniqueKernelName;
q.submit([&](sycl::handler& cgh){
  cgh.parallel_for<UniqueKernelName>(...);
});
```

After:

```
q.submit([&](sycl::handler& cgh){
  cgh.parallel_for(...);
});
```

- **Massive convenience improvement!**
- Highly non-trivial in multipass scenarios (C++ does not define unique names for lambdas)
- Just works in integrated multipass; in explicit multipass relies on clang HIP/CUDA `__builtin_get_device_side_mangled_name()`
- Cannot work in nvc++-flow for non-CUDA/non-CPU targets

Practical impact:
Challenges when transitioning oneAPI code from DPC++ to hipSYCL

- HeCBench[4]: Many benchmarks (>280) gathered from various sources
- Multiple programming models, including HIP, CUDA, SYCL
- SYCL ports originally developed for DPC++ and oneAPI
- ⇒ Can investigate performance compared to native models, and portability issues between DPC++ and hipSYCL

[4] https://github.com/zjin-lcf/HeCBench

Porting HeCBench to hipSYCL

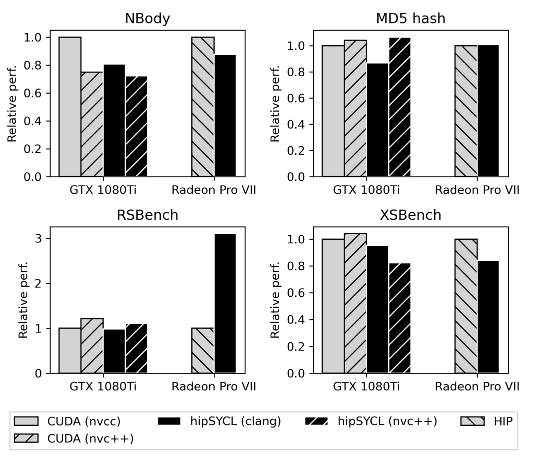
- 209 benchmarks work with DPC++, out-of-the box 91 work with hipSYCL
- Recurring problems prevent most of the remaining from compiling:
  - HeCBench uses `CL/sycl.hpp` and the `::sycl` namespace. The SYCL specification is ambiguous; in hipSYCL `CL/sycl.hpp` only exposes `::cl::sycl`
    - fixing this increases the number of compiling benchmarks to 114.
  - Some benchmarks use SYCL 2020 functionality `sycl::ext::oneapi` namespace, even though it should be in `::sycl` (e.g. `atomic_ref`).
    - fixing increases number of compiling benchmarks to 122.
  - In hipSYCL (or DPC++ in CUDA interop scenarios), vector aliases (e.g. `sycl::float2`) collide with CUDA types (`::float2`) if `using namespace sycl`
  - Some benchmarks are both `using namespace sycl` and `using namespace std` which causes collisions (e.g. `std::queue`, `sycl::queue`)
  - Some non-standard APIs are used (e.g. `sub_group::shuffle()`)
  - Subtle differences in implicit type conversion behavior, e.g. in `vec` constructor

Addressing portability issues

- How to improve SYCL namespace usage? → Best practice guide & spec clarifications
- Non-standard APIs → should become better as implementations move fully to SYCL 2020
- Implicit conversion behavior → spec clarifications?

- **The good:** Most issues are not due to lack of functionality in hipSYCL, but due to recurring, simple issues
- **The bad:** The fact that the basics like header and namespace usage already cause problems is worrying for the SYCL ecosystem as a whole!

HeCBench performance



- hipSYCL within 20% of native model
- hipSYCL outperforms HIP by 3x for RSBench; this indicates suboptimal performance with the HIP version.

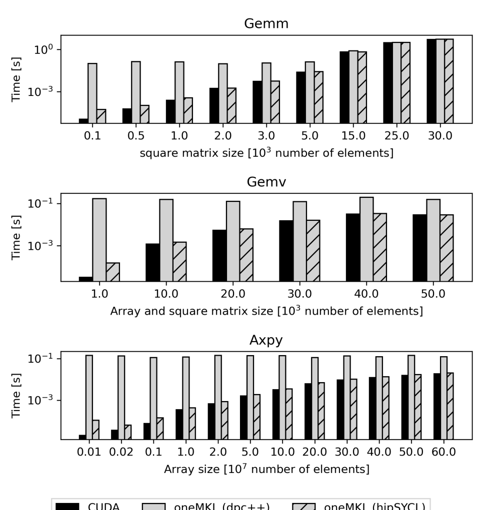
Libraries: oneMKL

oneMKL hipSYCL support

- We have upstreamed support for oneMKL BLAS with hipSYCL on NVIDIA, CPU
- We have upstreamed new rocBLAS backend for hipSYCL
- rocRAND backend PR open
- oneMKL mainly requires backend interoperability; stresses ability of SYCL implementation to expose backend objects
- Leverage hipSYCL `enqueue_custom_operation` extension which has been shown to outperform SYCL 2020 `host_task` for this purpose[5]

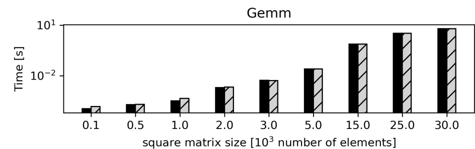
[5] `https://github.com/illuhad/hipSYCL/blob/develop/doc/enqueue-custom-operation.md`

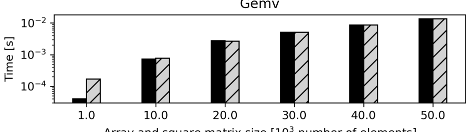
oneMKL on GTX 1080 Ti

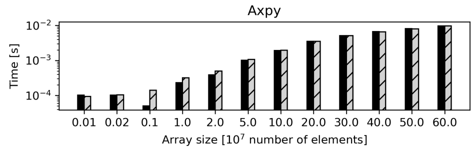


- DPC++ affected by performance bug (only fixed recently after submission deadline)
- oneMKL with hipSYCL competitive with native cuBLAS until very small problem sizes

oneMKL on Radeon Pro VII







CUDA oneMKL (hipSYCL)

- oneMKL with hipSYCL competitive with native rocBLAS
- oneMKL and rocBLAS both seem to show overhead of $\approx 10^{-4}$ s.

Conclusion

- First attempt to implement oneAPI independently from DPC++
- We have shown that hipSYCL can support
  - SYCL 2020
  - oneAPI building blocks (Level Zero)
  - oneAPI libraries (oneMKL)
- ...and deliver competitive performance
- $\Rightarrow$ **It is indeed possible to have a multi-compiler ecosystem for oneAPI!**
- Spec ambiguities, design differences, non-standard APIs and not well-known best practices can hinder portability in practice
  - Portability across implementations requires explicit attention by the programmer!