On The Compilation Performance of Current SYCL Implementations

Peter Thoman,
Facundo Molina Heredia and Thomas Fahringer

peter.thoman@uibk.ac.at

IWOCL & SYCLcon
10th INT'L WORKSHOP ON OPENCL & SYCL

universität innsbruck

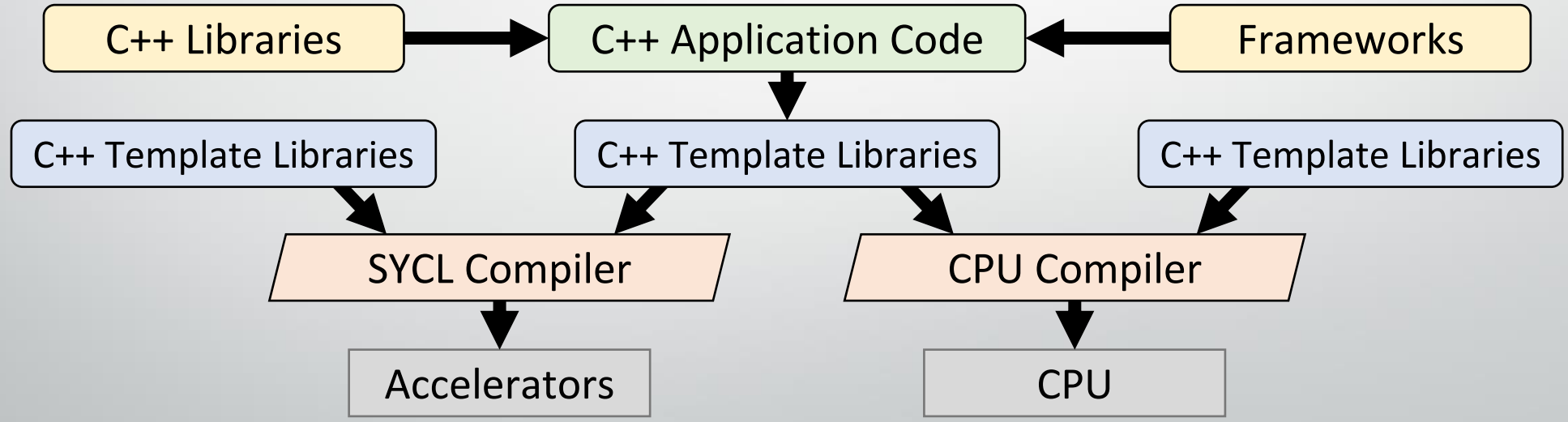
DPS
Distributed and Parallel Systems

Background, Motivation & Goals

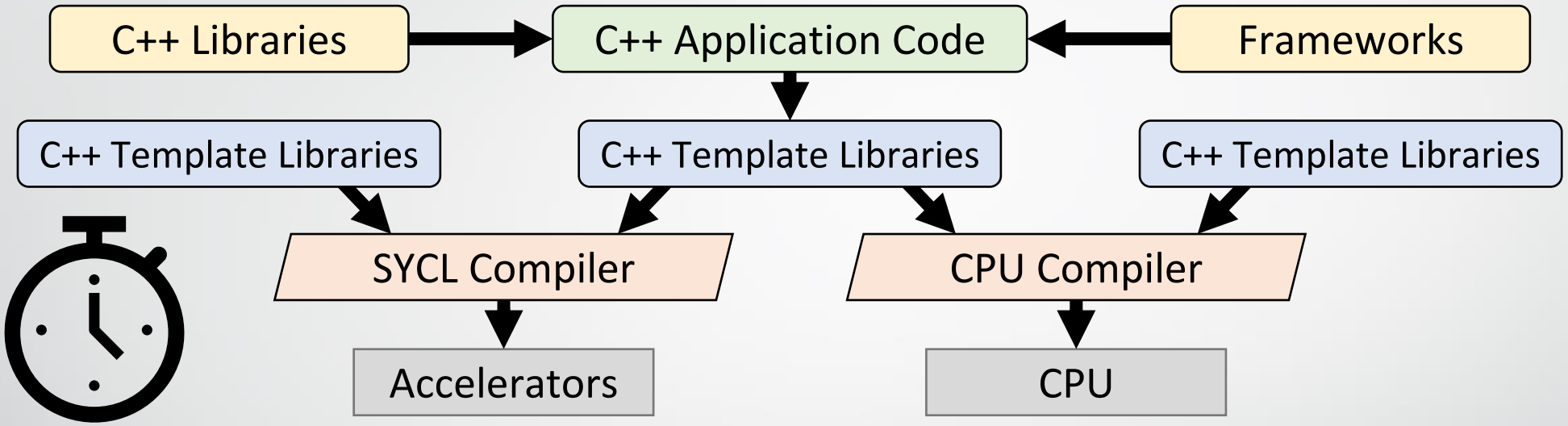
Background

"SYCL is ideal for accelerating larger C++-based engines and applications with performance portability."

https://www.khronos.org/sycl



Background

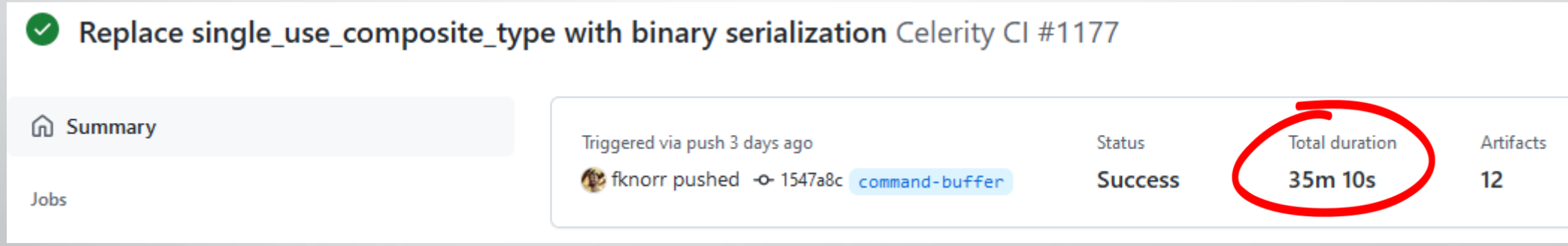


- “Larger” also means more source code
- SYCL is based on C++, and specifically uses quite a bit of templated code
  - Which gives us all the nice type safety, zero-cost abstractions, etc.
  - “Zero-cost” does *not* apply to compile time

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

Celerity
High-level C++ for Accelerator Clusters

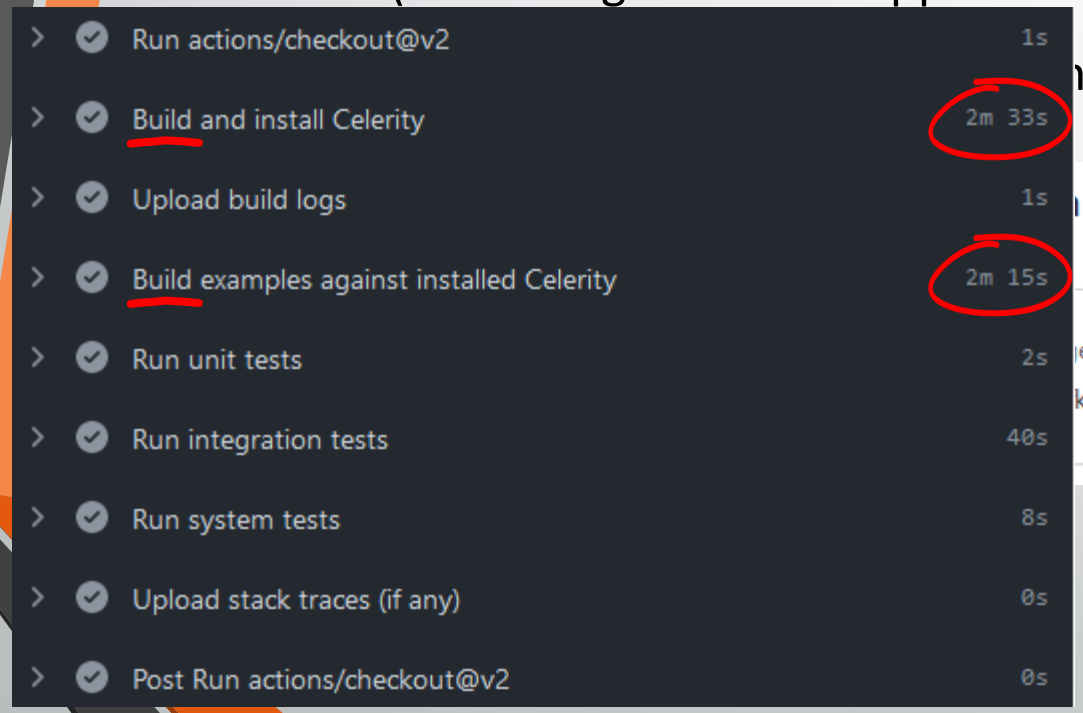


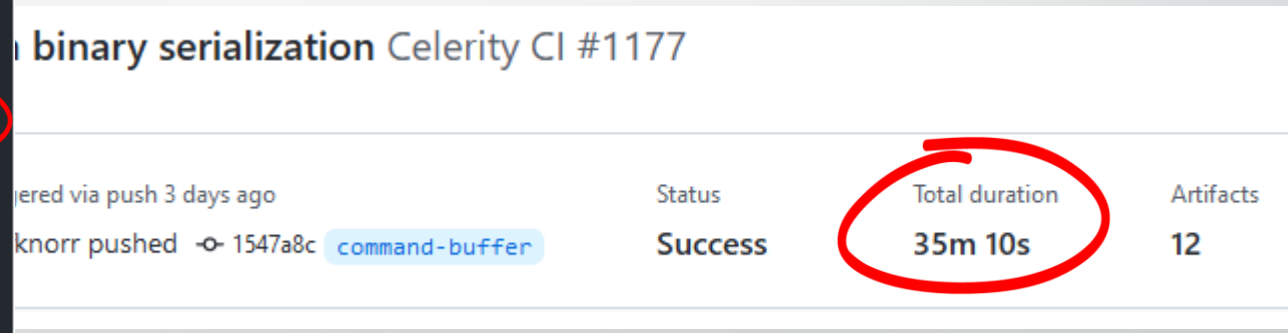
Motivation

- **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)

Celerity
High-level C++ for Accelerator Clusters

n 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)

	DPCPP	ComputeCpp	hipSYCL	triSYCL	neoSYCL	Sylkan
Targets	CPUs, OpenCL+SPIR-V, CUDA+PTX	SPUs, OpenCL+SPIR, OpenCL+PTX*	OpenMP, CUDA, ROCm	OpenMP, OpenCL+SPIR*	VEO	Vulkan+SPIR-V (non-compute)
Open Source	✓[a]	-	✓[b]	✓[c]	✓[d]	✓[e]
Reference	[27]	[29]	[2]	[7]	[20]	[31]
Github Stars	603	-	496	389	13	12

SYCL Implementations Evaluated

Identifier	Notes	Versions
ccpp	ComputeCPP (spir64 BE)	2.4, 2.5, 2.6, 2.7, 2.8
dpcpp_s	Data-parallel CPP (spir64 BE)	2022-01-13, 2021-11-14, 2021-09-15
dpcpp_n	Data-parallel CPP (CUDA BE)	2022-01-13, 2021-11-14, 2021-09-15
dpcpp_ns	Data-parallel CPP (both BEs)	2022-01-13, 2021-11-14, 2021-09-15
hipSYCL	HipSYCL (CUDA BE)	2022-01-13, 2021-09-15, 2021-07-17, 2021-05-18, 2021-03-19
triSYCL	TriSYCL (C++20 BE)	2022-01-12

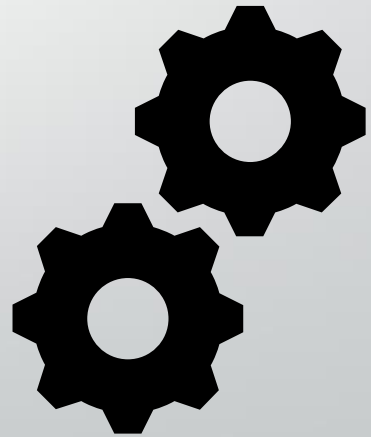
- Versions *usually* spaced 60 days apart – some exceptions:
  - hipSYCL: 2021-11-14 not stable
  - Dpcpp: build system changes before 2021-09
  - ComputeCPP: use numbered releases

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<kernel_1>(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

CPU	2x AMD EPYC 7282 16 Core *
Memory	256 GB DDR4-3200, 8 channel
Storage	Samsung NVMe SSD SM981 #
OS	Ubuntu Linux 20.04.2 LTS
Kernel	5.4.0-80-generic
Base Compiler	g++ 9.3.0

* frequency locked at 2.8 GHz all-core

generated output on ramdisk

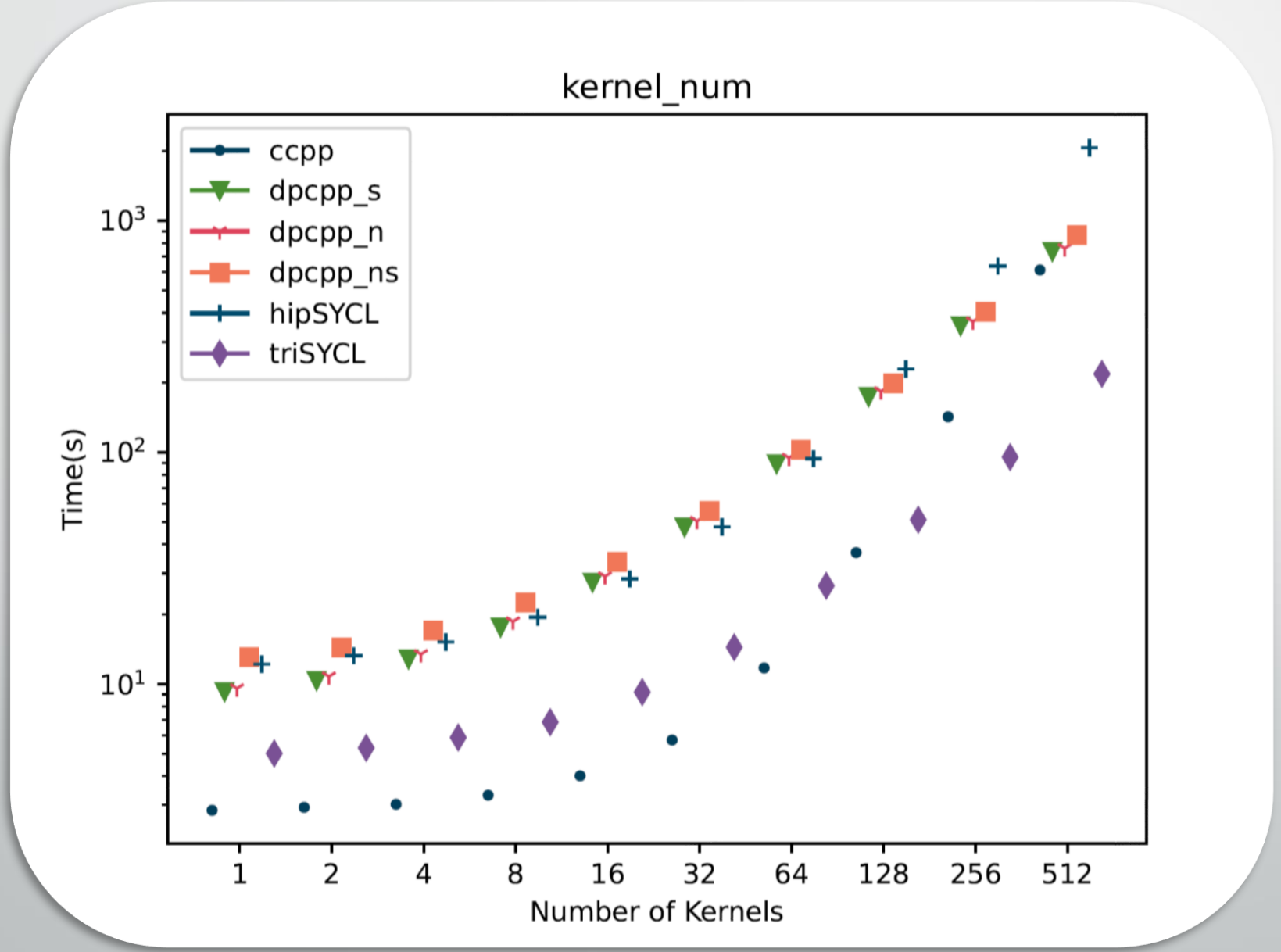
Statistics & Runs

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

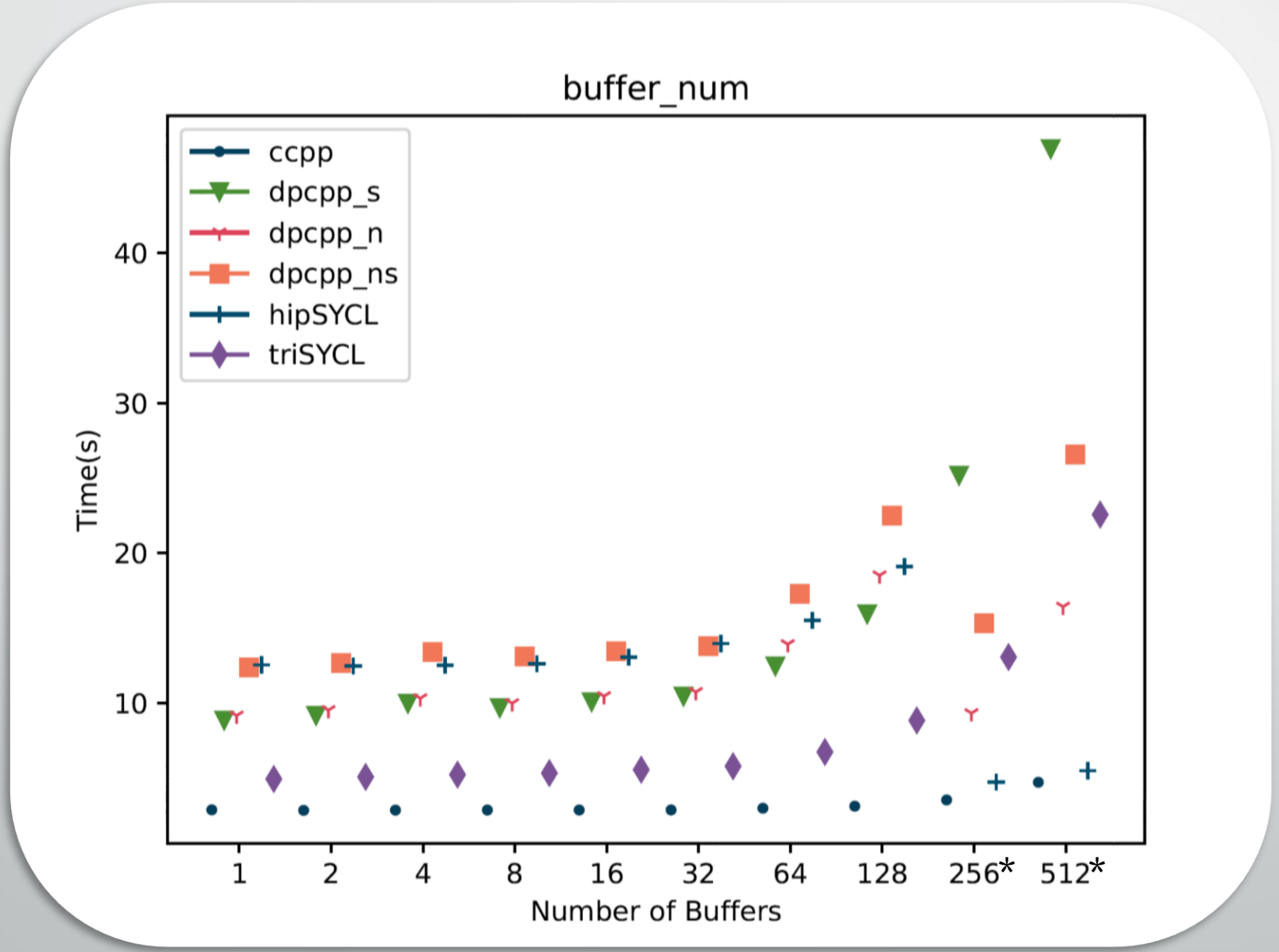
= **96000** measurements

→ Parallelize experiments
(verified minimal impact on subset)

Kernel Scaling

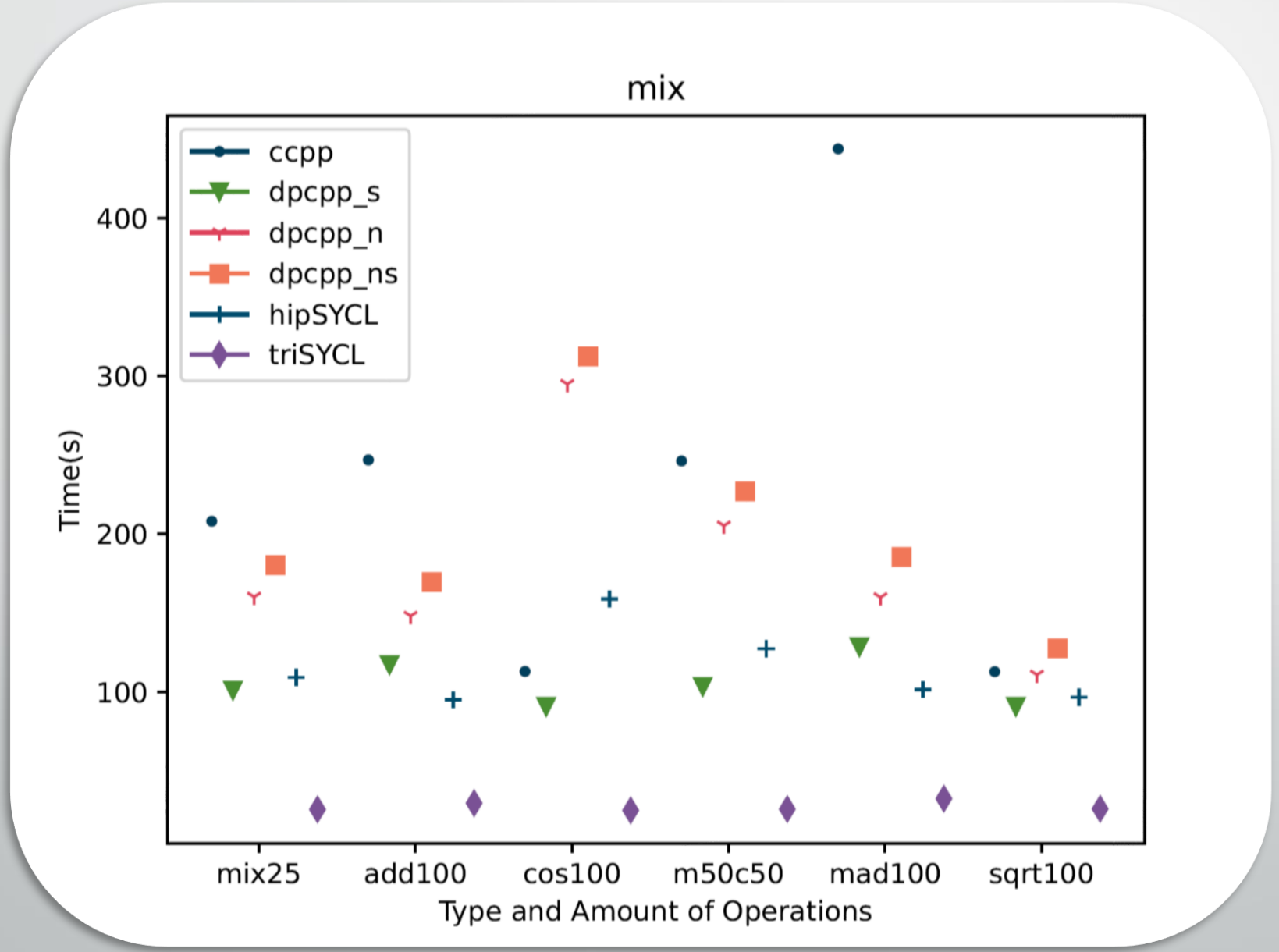


Buffer Scaling

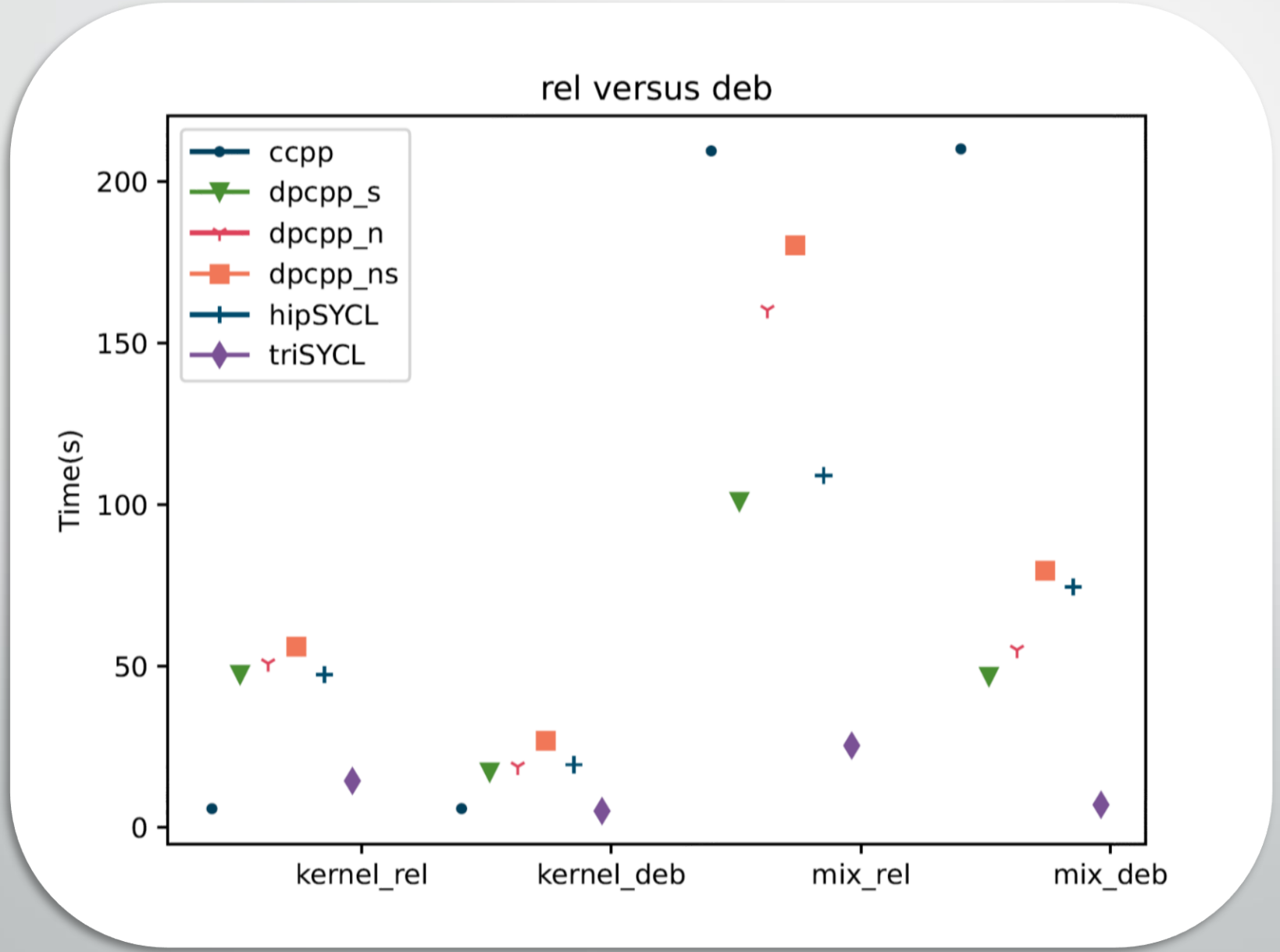


* dpcpp_n(s) and hipSYCL fail

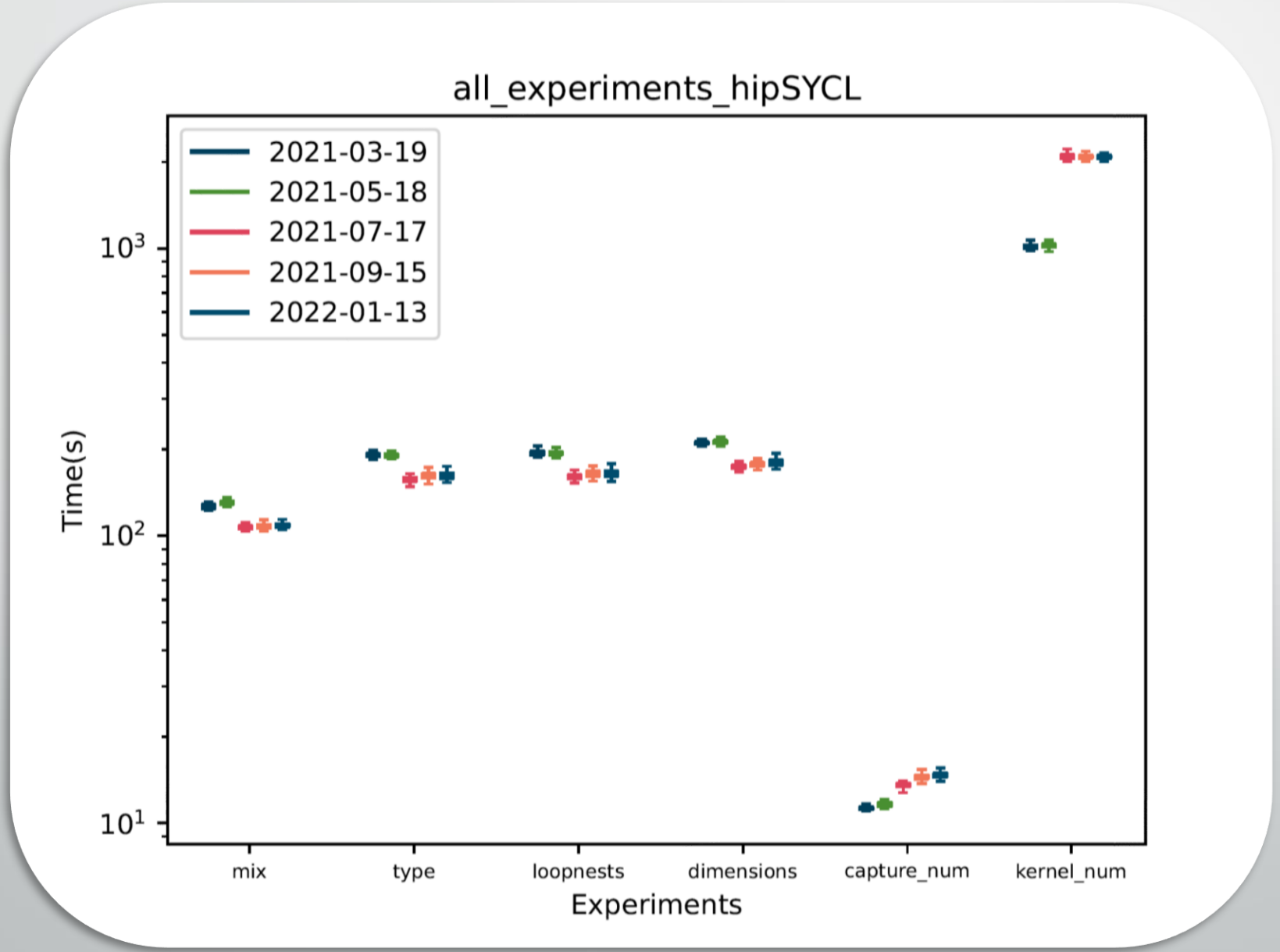
Instruction Mix



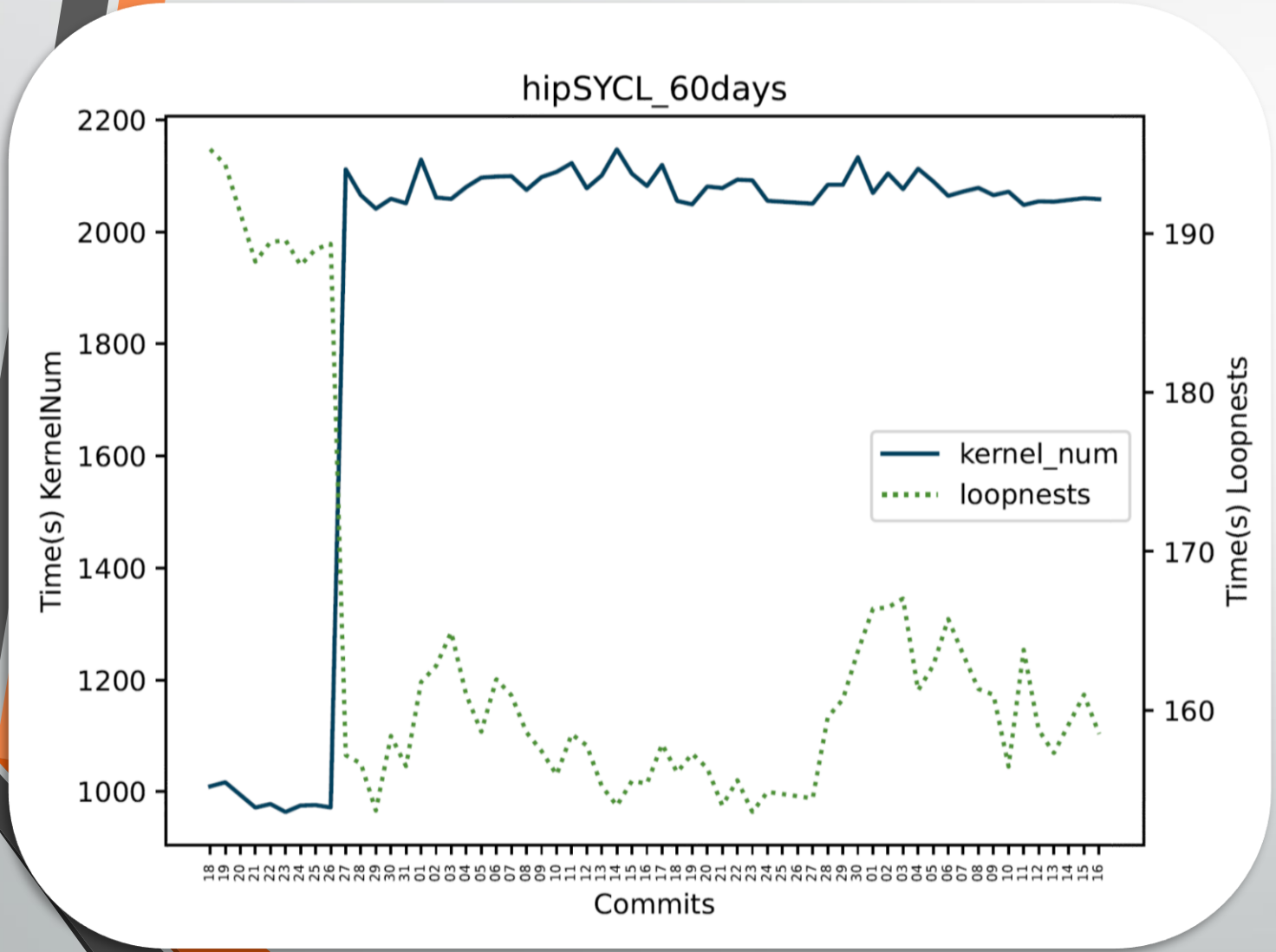
Release and Debug Configurations



Performance Over Time



Performance Over Time - Detail



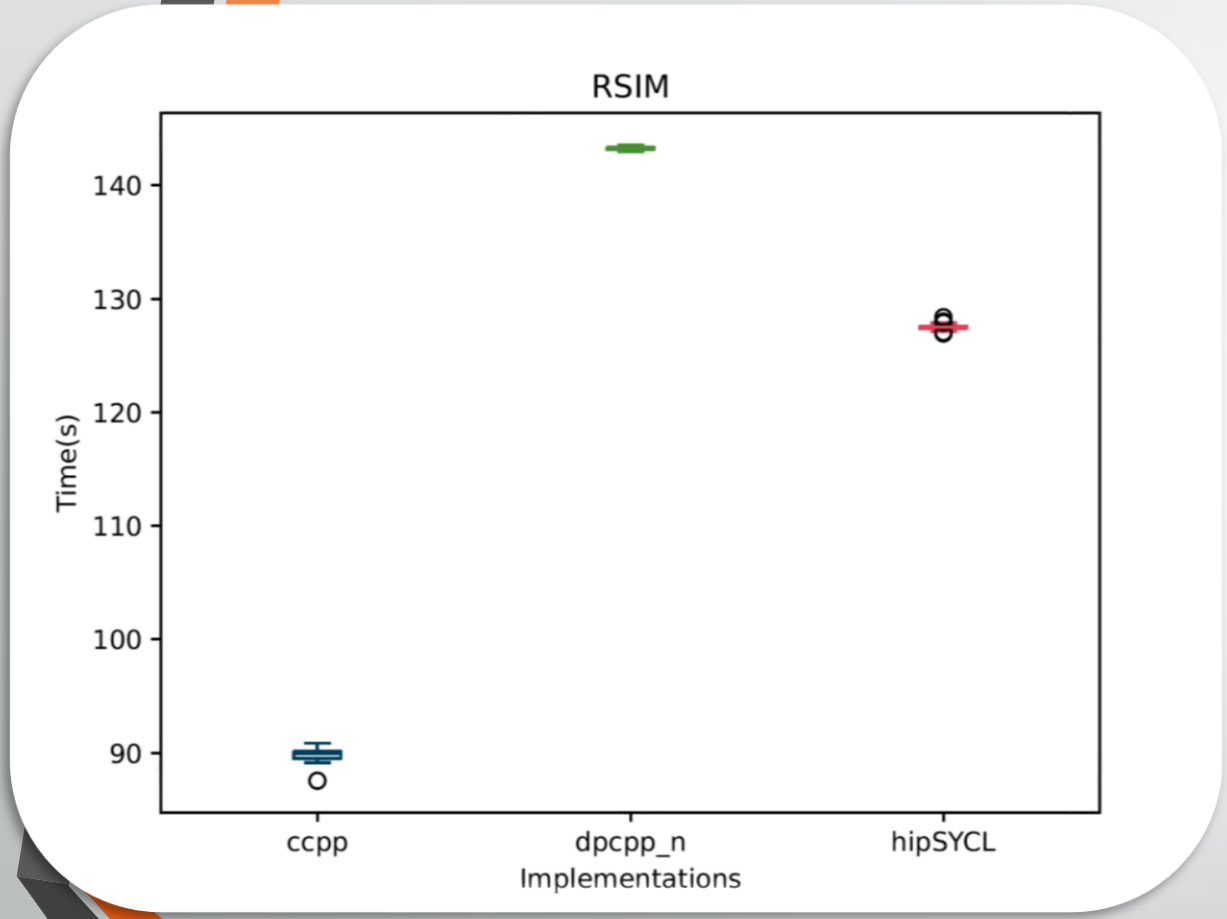
26th of May, 2021

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

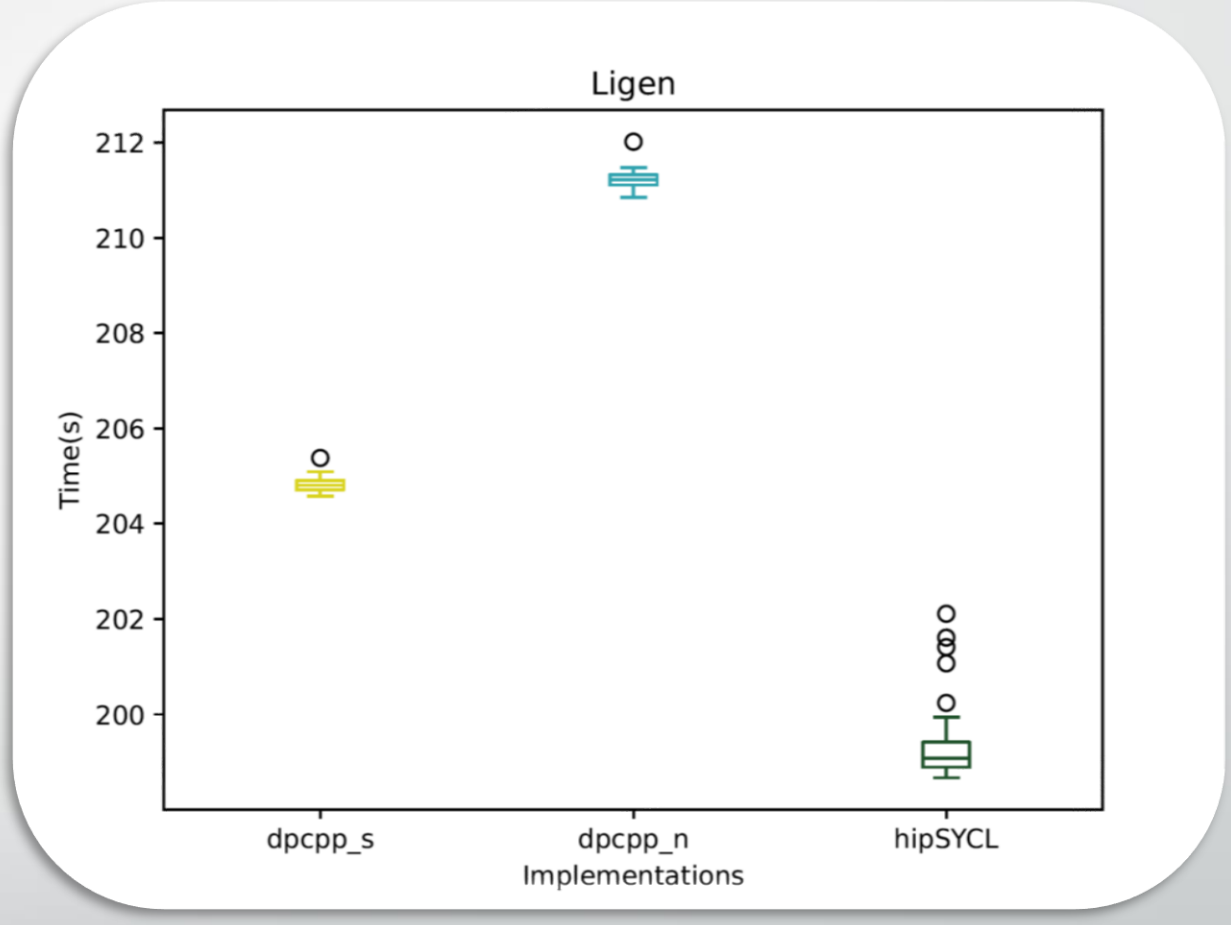
→ Far-reaching changes

→ 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).

universität innsbruck

DPS
Distributed and Parallel Systems