13th International Workshop on OpenCL and SYCL

IWOCL 2025

Fast In-Memory Runtime Compilation of SYCL Code

Julian Oppermann – Codeplay Software

Lukas Sommer, Mehdi Goli – Codeplay Software
Chris Perkins, Greg Lueck – Intel

April 7-11, 2025 | Heidelberg, Germany | iwocl.org

KHRONOS GROUP

codeplay

Enabling AI & HPC To Be Open, Safe & Accessible To All

Established 2002 in **Edinburgh, Scotland**.

Grown successfully to around 100 employees.

In 2022, we became a **wholly owned subsidiary** of Intel.

Committed to expanding the **open ecosystem** for heterogeneous computing.

Through our involvement in oneAPI and SYCL governance, we help to **maintain and develop** open standards.

Developing at the forefront of **cutting-edge research**.

Currently involved in two research projects - **SYCLOPS** and **AERO**, both funded by the Horizon Europe Project.

Overview

- **`kernel_compiler`** extension
- In-memory compilation of SYCL code
- Preliminary performance results
- Conclusion

`kernel_compiler` extension

Motivation for runtime compilation

- Runtime (or online) compilation is useful when code is generated or specialised at runtime
  - Template metaprogramming, code generated from domain-specific language, etc.
  - Going beyond specialisation constants

- Canonical example: GEMM library
  - Highly optimised implementation picked specifically for target and shape of operation
  - Intractable to enumerate & precompile all combinations

- Orthogonal to JIT vs. AOT compilation modes in SYCL implementations
  - Kernels are **not** known at compile time of the application

`kernel_compiler` extension

- Presented last year at IWOCL
- This talk is an update on the SYCL language support
- Extension also handles OpenCL and SPIR-V
  - Ninja-optimised implementations, new hardware features without SYCL extension, etc.

Example – Kernel

```
auto constexpr SYCLSource = R"""(
#include <sycl/sycl.hpp>

namespace syclext = sycl::ext::oneapi;
namespace syclexp = sycl::ext::oneapi::experimental;

extern "C"
SYCL_EXT_ONEAPI_FUNCTION_PROPERTY((syclexp::nd_range_kernel<1>))
void iota(int start, int *ptr) {
  size_t id = syclext::this_work_item::get_nd_item<1>().get_global_linear_id();
  ptr[id] = start + id;
}
)""";
```

(1) (2) (3)

- Source string using **free-function kernel** syntax
  1. Marker property
  2. Function arguments == kernel arguments, order matters
  3. Free function to get iteration index

Example – Build

```
#include <sycl/sycl.hpp>

using namespace sycl;
namespace syclexp = sycl::ext::oneapi::experimental;

int main() {
  auto lang = syclexp::source_language::sycl;

  queue q;
  assert(q.get_device().ext_oneapi_can_compile(lang)); (1)

  kernel_bundle<bundle_state::ext_oneapi_source>
    kbSrc = syclexp::create_kernel_bundle_from_source( (2)
      q.get_context(), lang, SYCLSource);

  kernel_bundle<bundle_state::executable>
    kbExe = syclexp::build(kbSrc); (3)

  // cont'd on next slide ...
```

- Extension uses kernel bundle infrastructure

(1) Check availability of runtime compilation

(2) Create bundle in new state **`ext_oneapi_source`** from string

(3) Build into executable state

Note: `create_bundle_from_source(…)` and `build(…)` take additional arguments and may throw exceptions

Example – Run

```
  kernel k = kbExe.ext_oneapi_get_kernel("iota");

  constexpr size_t N = 128;
  int *buffer = sycl::malloc_shared<int>(N, q);

  q.submit([&](sycl::handler &cgh) {
     cgh.set_arg(0, 42);
     cgh.set_arg(1, buffer);
     cgh.parallel_for(range{N}, k);
   }).wait();

  sycl::free(buffer, q);

  return 0;
}
```

1

2

```
$ clang++ -fsycl iota.cpp –o iota
$ ./iota
```

3

1. Obtain kernel object from bundle by name
   - NB: Kernel function was declared as **`extern “C”`**
   - Will revisit in a moment

2. Set arguments by index and invoke kernel

3. No special flags required to compile and run the application

Properties

```
auto constexpr SYCLSource = R"""(
#include <sycl/sycl.hpp>
#include "foo/bar.h" (1)
#include "mylibrary.h" (4)

extern "C" SYCL_EXT_ONEAPI_FUNCTION_PROPERTY((
sycl::ext::oneapi::experimental::single_task))
void mytask(int x, int *ptr) { *ptr = func(x) + MY_MACRO; }
)""";

syclexp::include_files includes{"foo/bar.h", (1)
    "int func(int x) { return x * x; }"};
auto kbSrc =
  syclexp::create_kernel_bundle_from_source(ctx,
  lang, SYCLSource, syclexp::properties{includes});

std::string log;
auto kbExe = syclexp::build(kbSrc, syclexp::properties{
  (2) syclexp::save_log{&log}, syclexp::build_options{ (3)
    std::vector<std::string>{"-DMY_MACRO=3",
                             "-Imylib/include"}}
});                                    (4)
```

- Application controls the compilation via properties

(1) **`include_files`**
  - Define virtual headers that can be included in the source string

(2) **`save_log`**
  - Request the full compilation log (e.g. to see warnings)

(3) **`build_options`**
  - Pass supported DPC++ options to the runtime compiler

(4) E.g.: add include paths

Source code names

- SYCL compilation is challenging
  1. C++ function names are mangled in an implementation-specific way, e.g.: **`foo::bar()`** becomes **`_ZN3foo3barEv`**
  2. Name is not sufficient to resolve overloads
  3. And what about templates?

4. **`registered_names`** property accepts list of C++ expressions that reference a free-function kernel
   - Use for **`ext_oneapi_[has|get]_kernel`**
   - Triggers template instantiation

```
auto constexpr SYCLSource = R"""(
#include <sycl/sycl.hpp>

#define STK SYCL_EXT_ONEAPI_FUNCTION_PROPERTY((        \
  sycl::ext::oneapi::experimental::single_task_kernel))

namespace mykernels {
1 STK void K1() {}
  STK void K2(int x) {}
2 STK void K2(float x) {}
3 template<typename T> STK void K3(T x) {}
}
)""";

std::vector<std::string> kernelNames{
  "mykernels::K1",
  "(void(*)(int))mykernels::K2",
  "(void(*)(float))mykernels::K2",
  "mykernels::K3<short>"};

auto kbExe = syclexp::build(kbSrc, syclexp::properties{
  syclexp::registered_names{kernelNames}}); 4

for (auto &kn : kernelNames)
  assert(kbExe.ext_oneapi_has_kernel(kn));
```

How to try it out yourself

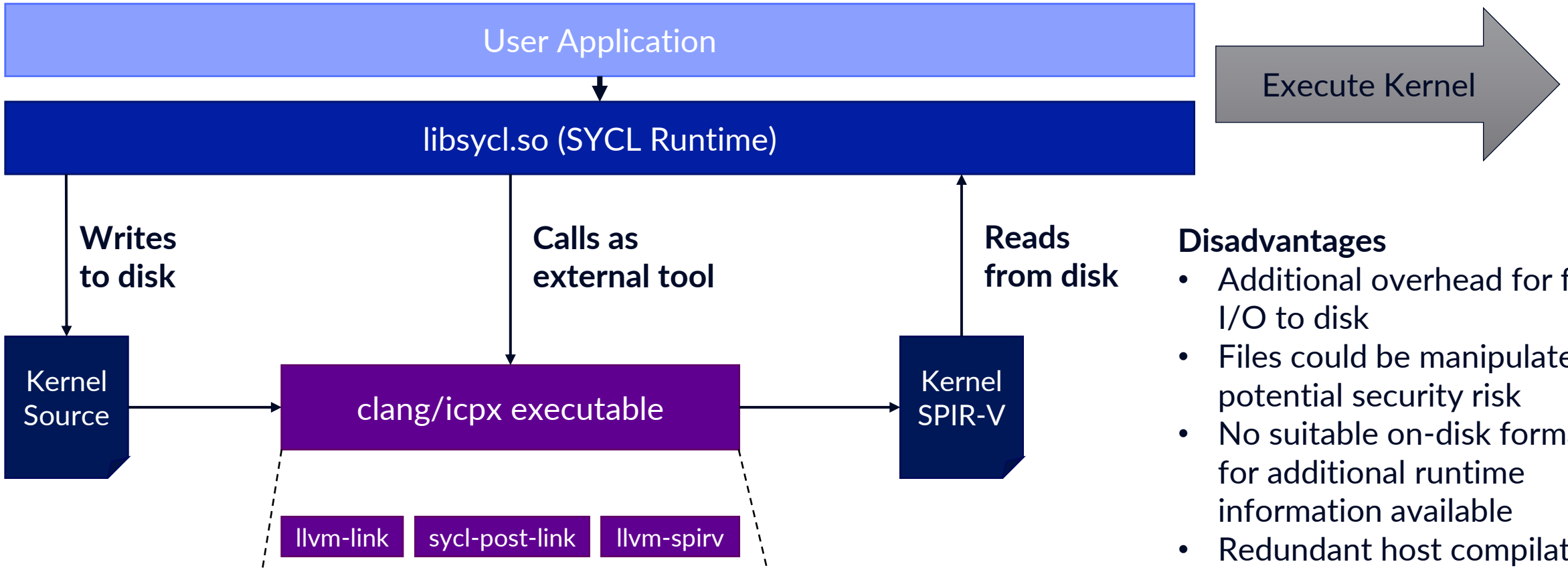
- SYCL runtime compilation included in DPC++ daily builds
- Planned to be part of the upcoming oneAPI 2025.2 release
- Refer to the extension specification for more information and current limitations

https://github.com/intel/llvm/releases

https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_kernel_compiler.asciidoc

In-memory compilation of SYCL code

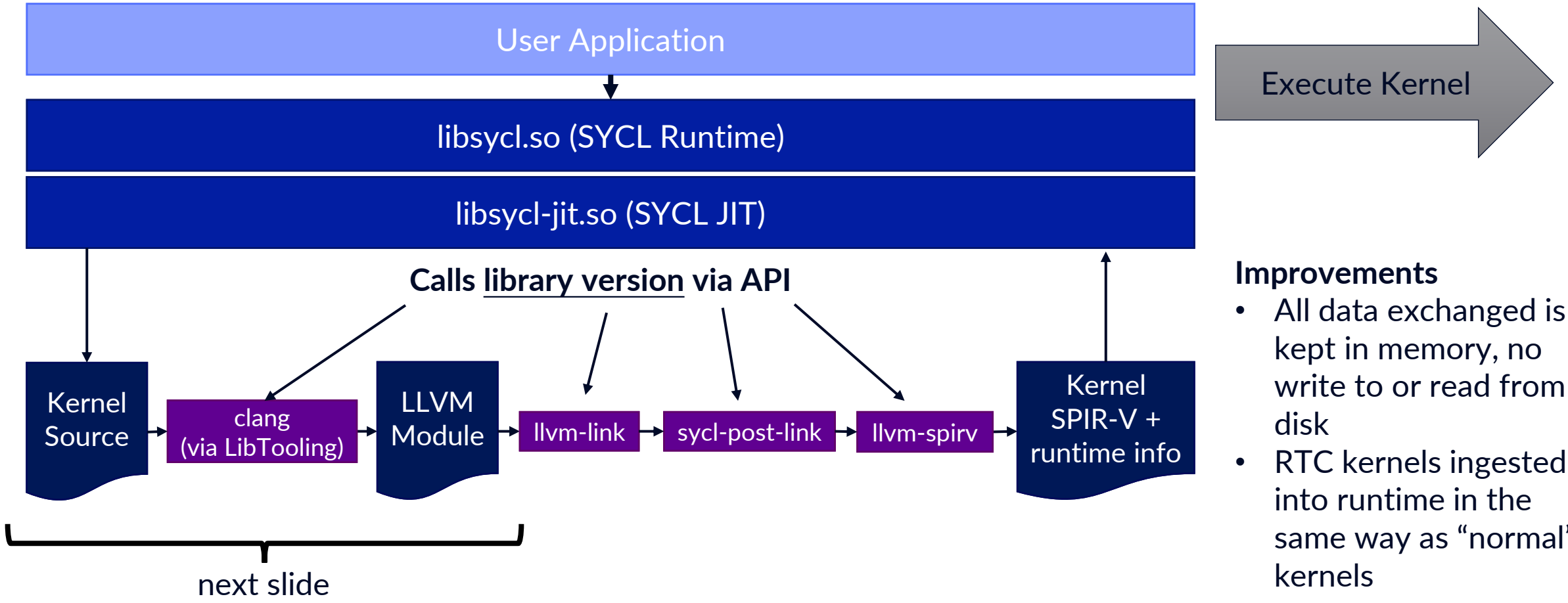
Invoke-based RTC implementation



Disadvantages

- Additional overhead for file I/O to disk
- Files could be manipulated, potential security risk
- No suitable on-disk format for additional runtime information available
- Redundant host compilation performed due to driver limitation

JIT-based RTC implementation

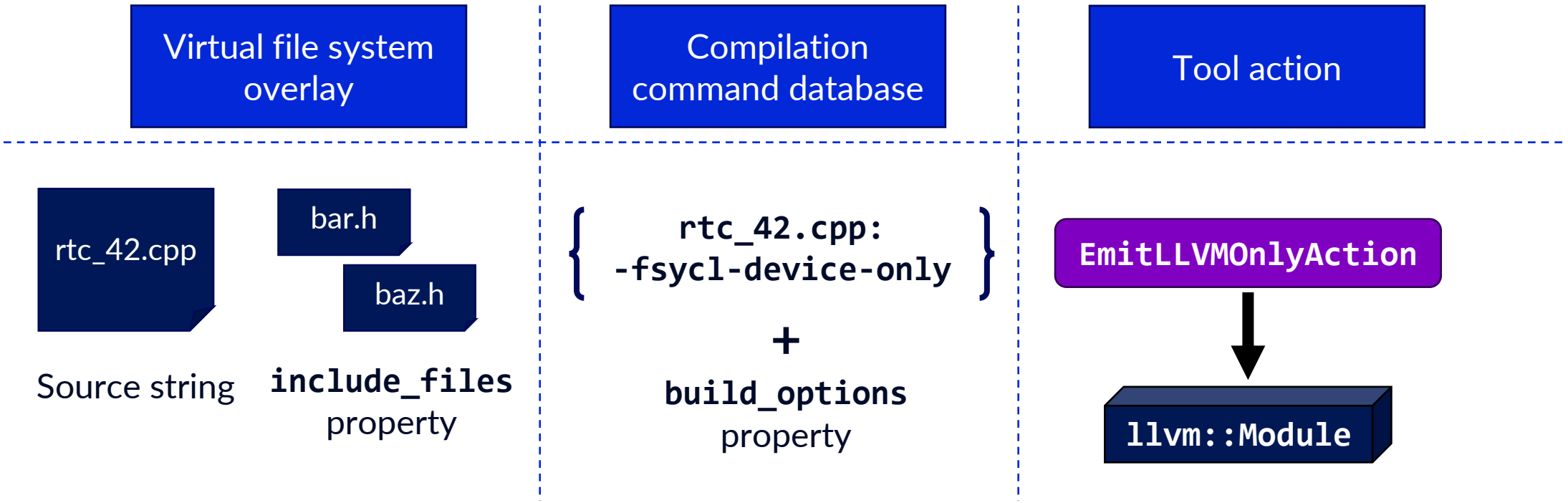


Improvements

- All data exchanged is kept in memory, no write to or read from disk
- RTC kernels ingested into runtime in the same way as “normal” kernels

Leveraging modular compiler technology

- Clang's LibTooling interface works with DPC++ and offers very high-level interface



Preliminary performance results

Methodology

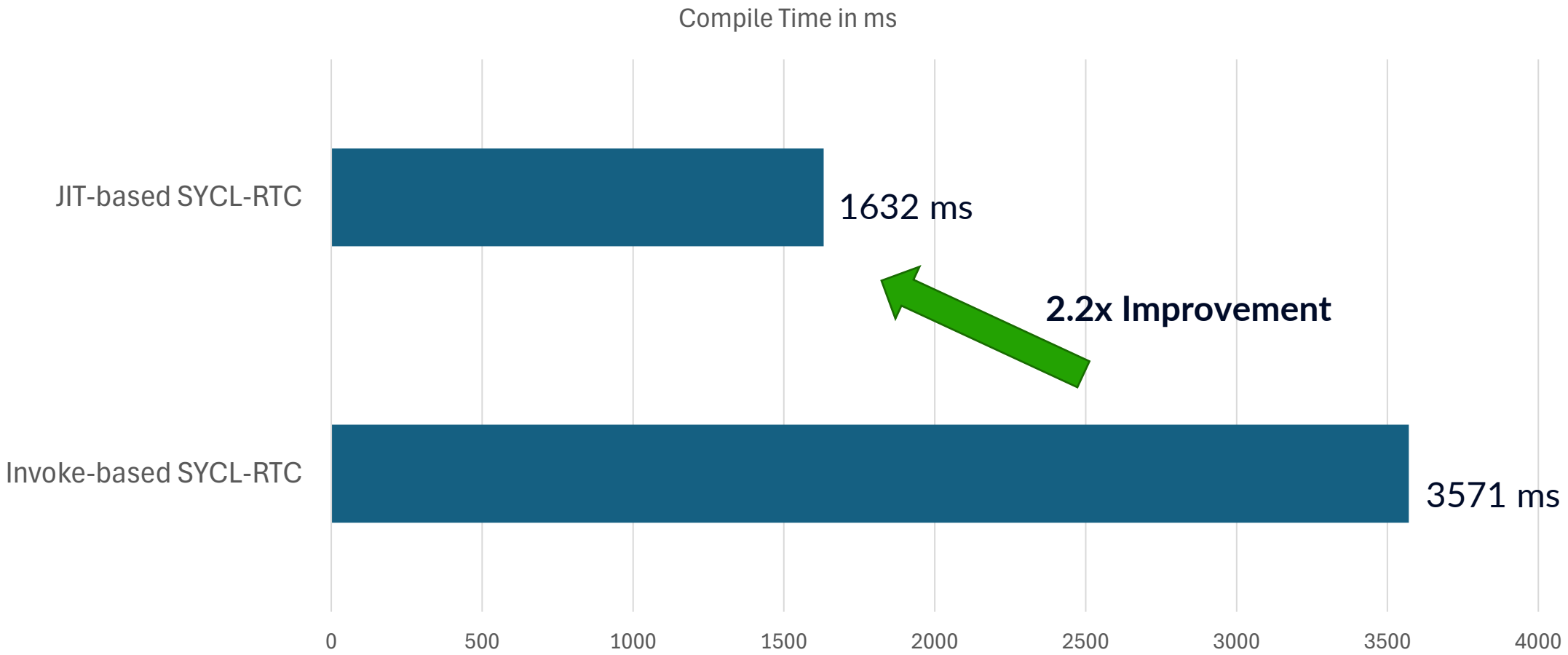
- Use simple vector addition kernel to compare compilation time of different implementations
  - Small kernel allows to assess the constant overhead
- Findings here do not only apply to SYCL-RTC, also hold for DPC++ SYCL compilation in general

```
const char *kernel_source = R"""(
#include <sycl/sycl.hpp>

extern "C" SYCL_EXTERNAL
SYCL_EXT_ONEAPI_FUNCTION_PROPERTY((sycl::ext::oneapi::experimental::nd_range_kernel<1>))
void vec_add(float* in1, float* in2, float* out){
  size_t id = sycl::ext::oneapi::this_work_item::get_nd_item<1>().get_global_linear_id();
  out[id] = in1[id] + in2[id];
}
)""";

sycl::kernel_bundle<bundle_state::ext_oneapi_source> source_bundle =
    sycl_exp::create_kernel_bundle_from_source(
        q.get_context(), source_language::sycl, kernel_source);

sycl::kernel_bundle<bundle_state::executable> exe_bundle =
    sycl_exp::build(source_bundle, source_bundle.get_devices());

sycl::kernel k = exe_bundle.ext_oneapi_get_kernel("vec_add");
```

Configuration details
DPC++ daily build 2025/03/14 (Git commit ID 61cb6d7)
https://github.com/intel/llvm/releases/tag/nightly-2025-03-14
CPU: 12th Gen Intel(R) Core(TM) i9-12900K
GPU: Intel(R) UHD Graphics 770
Environment:
`ONEAPI_DEVICE_SELECTOR=level_zero:gpu`
`NEO_CACHE_PERSISTENT=0`
Reported runtimes are averaged over 10 calls to `sycl_exp::build(…)`.

Performance varies by use, configuration and other factors.
Performance results are based on testing on 2025-03-31 and may not reflect all publicly available updates. No product or component can be absolutely secure. Your cost and results may vary. Intel technologies may require enabled hardware, software or service activation.

Performance comparison

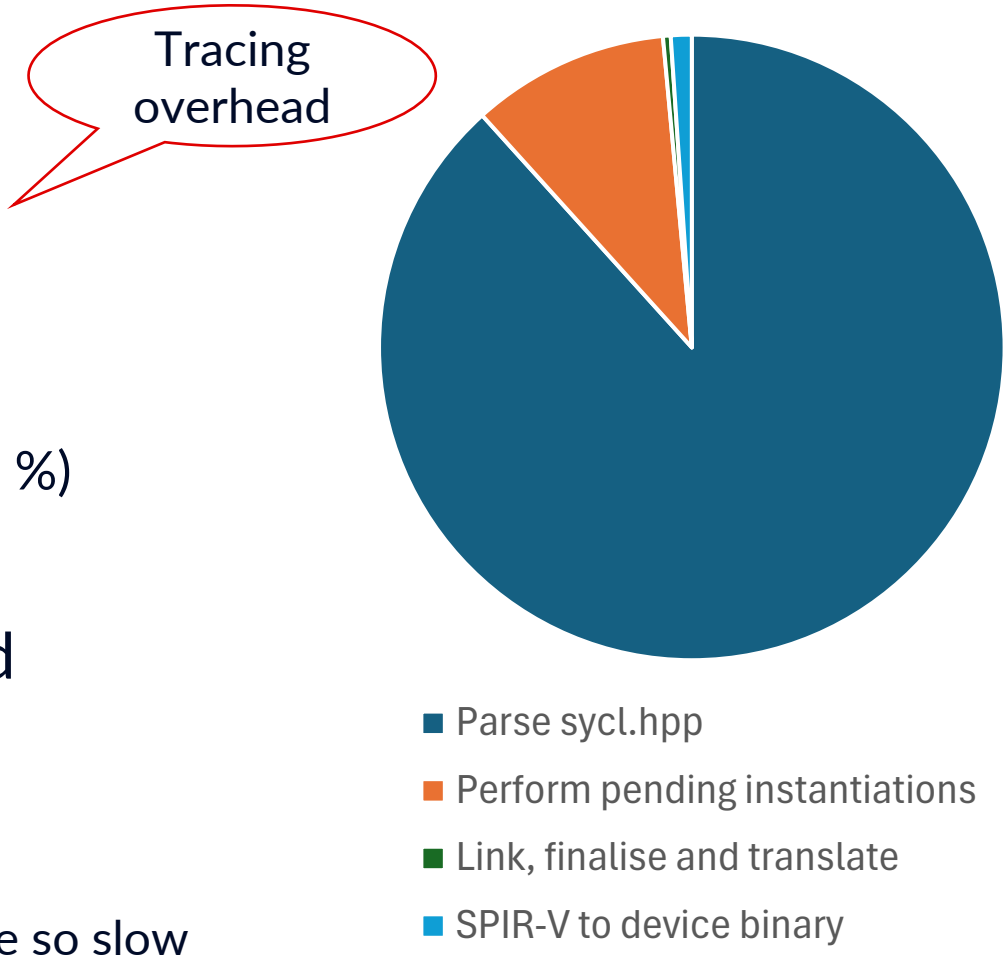


Notices and Disclaimers: Performance varies by use, configuration and other factors. Performance results are based on testing on 2025-03-31 and may not reflect all publicly available updates. See slide "Methodology" for configuration details. No product or component can be absolutely secure. Your cost and results may vary. Intel technologies may require enabled hardware, software or service activation.

Compilation time breakdown

- *Almost all time is spent in frontend*
- Time spent in runtime compiler: 1811 ms
  - **Frontend: 1785 ms (98.5%)**
  - Link, finalise and translate: 7 ms (0.4%)
  - SPIR-V to device binary translation: 19 ms (1.1 %)

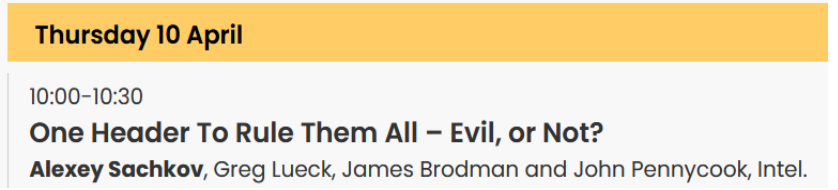
- Two major contributing phases in frontend
  - Parsing `sycl.hpp`: 1556 ms (85.9%)
    - ≈ 250 ms associated with extensions
  - Perform pending instantiations: 180 ms (9.9%)
    - Mere number of instantiations makes this phase so slow



Avenues for faster compilation

- Use pre-compiled headers to process `sycl.hpp` faster
  - AST serialised into binary format, all-but-eliminates parsing step
  - Supported by clang, but not yet for DPC++

- Modularise `sycl.hpp` to process fewer headers
  - Do not include all extensions by default, maybe even skip core features (e.g. accessors)
  - Specification work and user opt-in required



- Use persistent cache
  - Caches the LLVM module obtained from frontend invocation
  - Cache hit: 115 ms (7% of 1632 ms) for vector add example

Conclusion

Conclusion

- RTC **extends** SYCL's toolbox for runtime specialisation
  - Wrap header-only libraries and instantiate templates on demand
  - Generate the best kernel for given input data and target device
  - Fuse user-supplied code-snippet into kernel
  - …
  - More generic/powerful than existing specialisation approaches

- (Almost) no limitations for kernel complexity or extension usage
  - Implemented as thin layer on top of DPC++ codebase

- Available to play around with!

13th International Workshop on OpenCL and SYCL

IWOCL 2025

Fast In-Memory Runtime Compilation of SYCL Code

Julian Oppermann – Codeplay Software

Lukas Sommer, Mehdi Goli – Codeplay Software
Chris Perkins, Greg Lueck – Intel

April 7-11, 2025 | Heidelberg, Germany | iwocl.org

KHRONOS GROUP

Disclaimers

A wee bit of legal

Performance varies by use, configuration and other factors.

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.

Your costs and results may vary.

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