How Much SYCL Does a Compiler Need? Experiences from the Implementation of SYCL as a Library for nvc++

Aksel Alpay Vincent Heuveline
Heidelberg University

Speaker: Aksel Alpay

IWOCL '22

Motivation

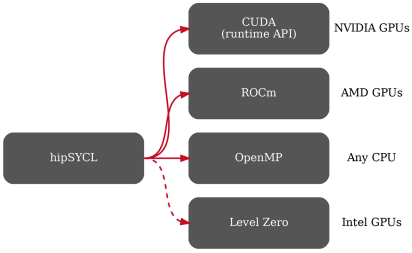
- Multiple SYCL implementations support CUDA devices through clang (DPC++, hipSYCL)
- No direct NVIDIA support
  - New NVIDIA hardware might not be immediately supported
  - CUDA installation not sufficient, users also need clang/LLVM
  - NVIDIA not helping with problems

⇒ **Is there a way to implement SYCL as a pure library for official NVIDIA compilers?**

…yes! and we have done it with hipSYCL and nvc++!

Introduction

hipSYCL



- Multi-backend architecture
- Aggregates multiple toolchains
  - OpenMP / clang CUDA / clang HIP / clang SYCL / **nvc++**

`https://github.com/hipSYCL/featuresupport`

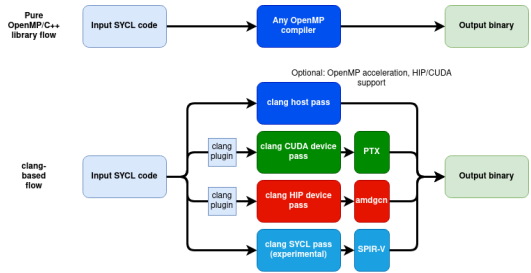
Optional lambda naming	✓ (PR)
Subgroups	✓ (PR)
In-order queues	✓ (PR)
Explicit dependencies (`depends_on()`)	✓ (PR)
Backend interop API	✓ (PR)
Reductions	✓ (PR)
Group algorithms	✓ (PR)
New device selector API	✓ (PR)
Aspect API	✓ (PR)
Deduction guides	✓ (PR)
`atomic_ref`	✓ (PR)

USM / reductions / sub-groups / group algorithms / optional lambda naming / …

`https://github.com/illuhad/hipSYCL`

- Open source
- Used in production by large projects
- Extensions such as buffer-USM interoperability
- Supported by SYCL libraries, e.g. oneMKL
- Supports most of SYCL 2020

How is CUDA currently targeted?



- clang plugin mainly detects kernels, walks AST and inserts CUDA `__global__`, `__device__` attributes where necessary
- clang CUDA toolchain handles the rest
- $\rightarrow$ **SYCL code is compiled as CUDA C++ code, but CUDA attributes are optional**
- CUDA C++ code can be used within kernels (used inside hipSYCL headers)

NVC++

- Part of NVIDIA's HPC SDK
- Supports OpenMP, OpenACC, Parallel STL offloading
- Also supports CUDA C++ in order to be able to leverage existing CUDA C++ algorithm libraries like CUB
- Since these models can be mixed, needs CUDA attributes to be optional!

```
1  template<class F>
2  __global__ void parallel_for(F f){ f(); }
3
4  // Will be implicitly __device__
5  void h(){}
6
7  int main(){
8    parallel_for<<<1,1>>>([](){
9      h();
10   });
11 }
```

- NVC++ already provides the most important feature of hipSYCL's clang plugin!
- We need NVC++'s CUDA support over e.g. STL algorithms to implement lower-level features (barriers, `nd_range parallel_for`, local memory, …)

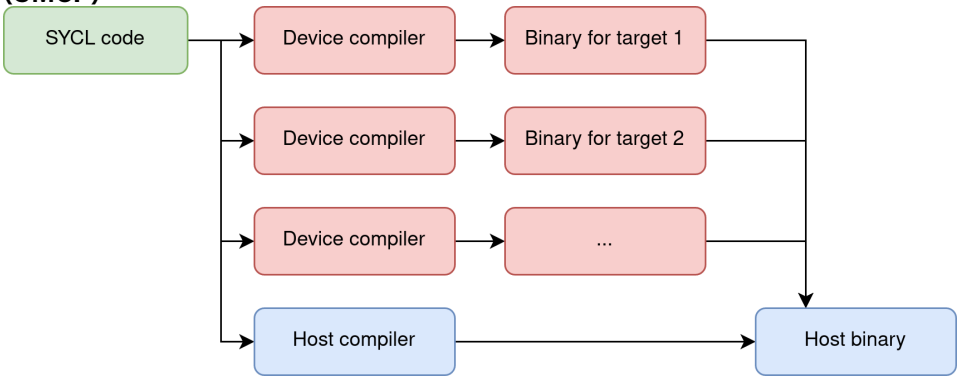
hipSYCL's CUDA support builds on:

- SYCL runtime invoking CUDA runtime functions
- SYCL kernel headers written in CUDA C++ (atomics, math builtins, group algorithms, …)
- Clang plugin (core functionality provided by nvc++)

With NVC++, hipSYCL's CUDA backend should be able to operate as a library for nvc++!

…but does SYCL allow it?

SYCL 2020 specification, section 3.12.1: **Single source multiple compiler passes (SMCP)**

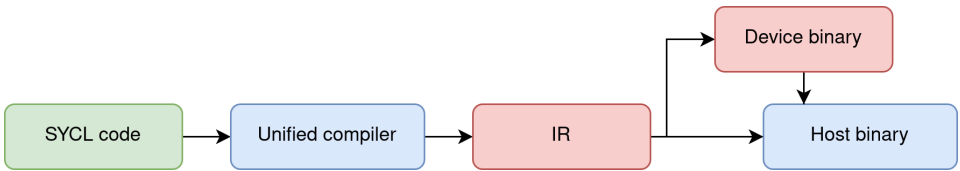


(Technique that most current implementations use)

…but does SYCL allow it?

SYCL 2020 specification, section 3.12.2: **Single source single compiler pass (SSCP)**

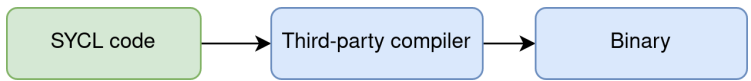
With this technique, the vendor implements a custom compiler that reads each SYCL source file only once, and that compiler generates the host code as well as the device images for the SYCL kernel functions. As in the SMCP case, each device image could either contain native device ISA or an intermediate language.



…but does SYCL allow it?

SYCL 2020 specification, section 3.12.3: **Library-only implementation**

It is also possible to implement SYCL purely as a library, using an off-the-shelf host compiler with no special support for SYCL. In such an implementation, each kernel may run on the host system.



...but does SYCL allow it?

- NVC++ follows an SSCP approach: Single compilation pass for both host and device.
- A SYCL implementation acting as a library for NVC++ is an implementation with third-party SSCP compiler and library-only characteristics.
- Library-only implementations are mainly intended to target the host by the specification; in practice there is little difference to a device library-only implementation.

$\Rightarrow$ SYCL as a library for NVC++ is likely legal, but might “bend” the specification slightly with regards to device library-only implementations.

Consequences of SSCP

- (in theory) better compilation times
- Macros cannot be used to specialize code between host and device passes!

```
1 // Common pattern in SYCL code -- spec only guarantees
2 // that this works in SMCP mode!
3 #ifdef __SYCL_DEVICE_ONLY__
4 // Code when compiling for device
5 #else
6 // Code when compiling for host
7 #endif
```

- NVC++ has `if target (nv::target::is_device){...}` and `if target (nv::target::is_host){...}`.
- Only works in control flow similar to regular `if()` and is not `constexpr`

Implementation

- Kernel launch: Implement kernel launches using CUDA <<<>>> syntax
  - Possible because hipSYCL headers pass a kernel launcher function object to the runtime for invocation. Kernel launcher can contain arbitrary CUDA code.
- Need generalization for `__SYCL_DEVICE_ONLY__` that also works in SSCP
  - `__SYCL_DEVICE_ONLY__` and other macros massively used in hipSYCL headers to select appropriate backend implementation (OpenMP/host, CUDA, HIP, SPIR-V)

```
namespace sycl {
float sin(float x) {
  __hipsycl_if_target_host(
    return std::sin(x);
  );
  __hipsycl_if_target_cuda(
    return cuda_builtins::sin(x);
  );
  ...
}
}
```

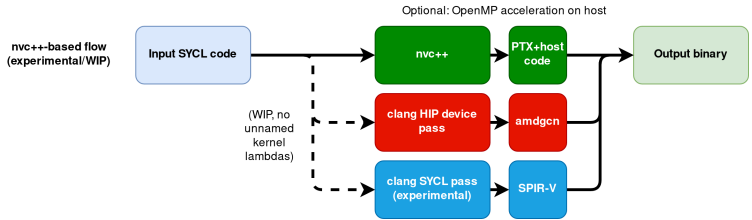
- Refactor headers to rely on `__hipsycl_if_target_*()`
- Note: Prevents data layout optimizations based on host/device

On-demand iteration space info

- In general, `group`, `item`, ...need to store global/local id of item and range
- As storage optimization, on device hipSYCL does not store this information and instead queries from backend builtins (e.g. CUDA `threadIdx`)
- $\rightarrow$ This storage optimization is not possible in NVC++ SSCP if we also want to support the CPU backend!

This is the only case of code divergence in kernels between hipSYCL's clang CUDA backend and the NVC++ CUDA backend.

Add new compilation flow driver



- Enable with `syclcc --hipsycl-targets="cuda-nvcxx"`

Limitations

Library-only implementations

While library-only implementations are explicitly allowed, some features are not (or not well) implementable → **Bug/contradiction in the specification**

The NVC++ library-only backend has the same issues as host library-only backends:

- Attributes (e.g. `reqd_work_group_size`)
- `kernel_bundle`: Limited control over kernel compilation and limited introspection

SYCL 1.2.1 hierarchical parallelism

In addition to typical host library-only limitations:

```
1  queue.submit([&](sycl::handler& cgh) {
2    cgh.parallel_for_work_group(
3      global_size / local_size, local_size,
4      [=](sycl::group<1> wg) {
5        // Code here is executed once per work group
6        wg.parallel_for_work_item([&](sycl::h_item<1> item) {
7          // Code here is executed once per work item
8      });
9    });
10 });
```

- Requires dedicated compiler support to obtain correct semantics on GPUs → not possible with NVC++
- discouraged in SYCL 2020 specification→ people shouldn't use anyway!
- hipSYCL's scoped parallelism proposal as potential successor is also implementable with NVC++

Compiler bugs

- Multiple NVC++ compiler bugs and crashes found using hipSYCL unit tests
- We are reporting bugs as we find and isolate them to NVIDIA
  - Some already fixed → Yes, you get NVIDIA support for using NVC++ with hipSYCL!
  - Mostly related to complex patterns used in unit tests, e.g. massively nested lambdas in scoped parallelism extension
  - Real code tends to work well

Performance

Performance

When comparing SYCL and CUDA, people usually compare **clang vs nvcc**.

- This changes multiple variables: Compiler frontends, code generation backends, programming model.

With the nvc++ library-only backend, we can **separate these effects!**

- Look at NVC++ compiling both SYCL and CUDA code (Same compiler, different model)
  - What is the impact of SYCL C++ abstractions?
- Look at hipSYCL using clang CUDA vs hipSYCL using NVC++ (Same model and code, different compiler)
  - What is the impact of using open source compilers vs. vendor compilers?

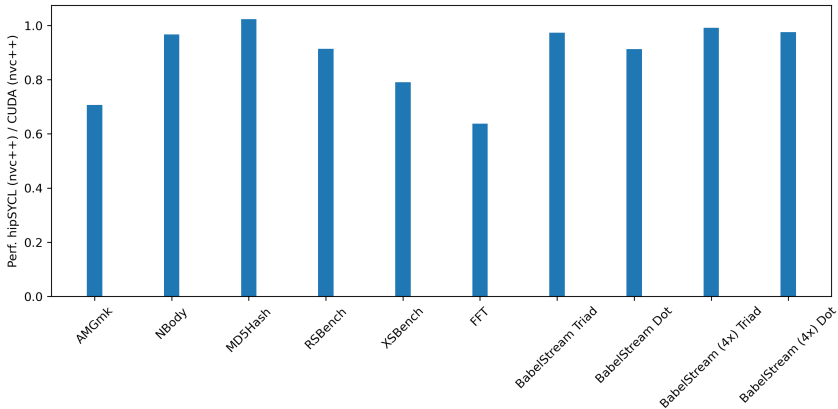
Experimental Setup

- GeForce GTX 1080 Ti
- Benchmarks from HeCBench[1] (benchmarks aggregated from multiple sources)
- …and BabelStream[2]
- hipSYCL 0.9.2 (1046a78), CUDA 11.6, clang 13, NVC++ 22.1

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

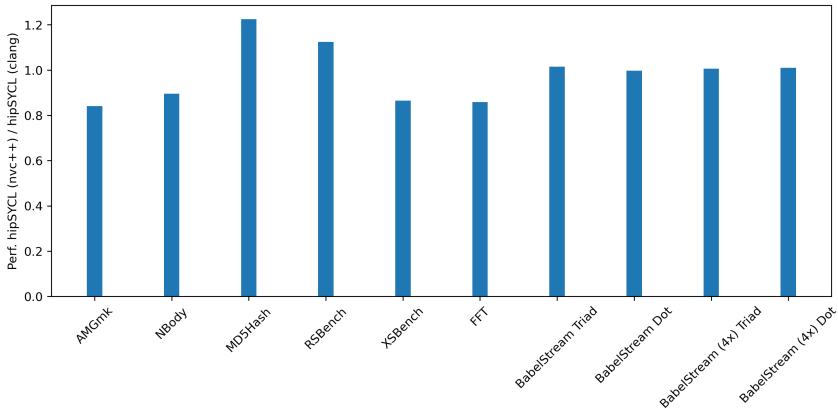
[2] Deakin T, Price J, Martineau M, McIntosh-Smith S. Evaluating attainable memory bandwidth of parallel programming models via BabelStream

Performance: SYCL vs CUDA with NVC++



- BabelStream C++ standard parallelism results comparable to CUDA
- (hip)SYCL runtime overheads for small kernels, especially if those use buffers

Performance: SYCL with NVC++ vs SYCL with clang



- Perf. **not** a reason to prefer vendor compilers!
- …Day 1 HW support, ease of deployment might be.

Conclusions

- First library-only **device** backend in a major SYCL implementation
- Use vendor-supported compiler for NVIDIA hardware
- Simplify SYCL deployment if NVIDIA HPC SDK is preinstalled; no LLVM needed
- Exact same code paths as hipSYCL's clang CUDA backend, except
  - No hierarchical parallelism
  - No storage optimizations for `item`,`group`,…on device (very small impact)
- Be aware that SYCL implementations might rely on SSCP! Avoid `__SYCL_DEVICE_ONLY__`!
  - SYCL spec does not provide any mechanisms to specialize code for host/device in SSCP mode ☹
- SYCL working group should resolve library-only issues
- No reason to only limit library-only backends to the host; spec should be clarified

But what does it mean for SYCL?

…SYCL can be implemented for devices as library for third-party vendor-supported compilers.

- $\Rightarrow$ SYCL could also be a portability layer for third-party compilers like e.g. Kokkos
- Prerequisite: Programming model with automatic kernel outlining (e.g., no explicit CUDA `__device__` attributes)
- For such a use case, SYCL should focus on library-only friendly programming models such as hipSYCL's scoped parallelism[3] over `nd_range` parallel for (difficult on CPU), or SYCL 1.2.1 hierarchical parallelism (difficult on GPU)

What does SYCL want to be?

[3] `https://github.com/illuhad/hipSYCL/blob/develop/doc/scoped-parallelism.md`