The 11th International workshop on OpenCL and SYCL

IWOCL & SYCLcon 2023

# One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends

Aksel Alpay, Universität Heidelberg

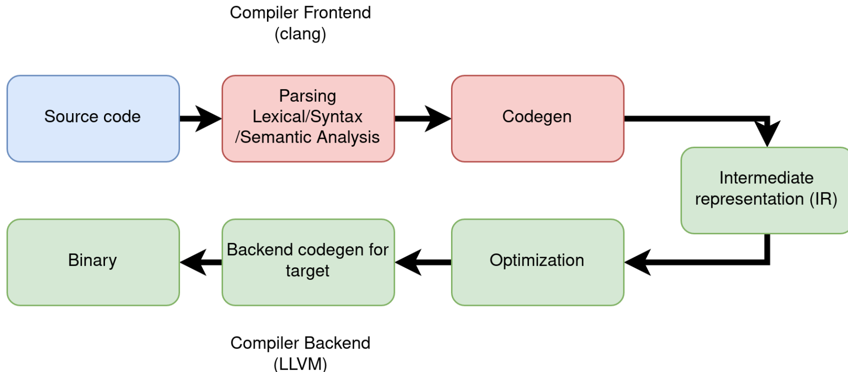
Vincent Heuveline, Universität Heidelberg

April 18-20, 2023 | University of Cambridge, UK iwocl.org

# Deployment of binaries

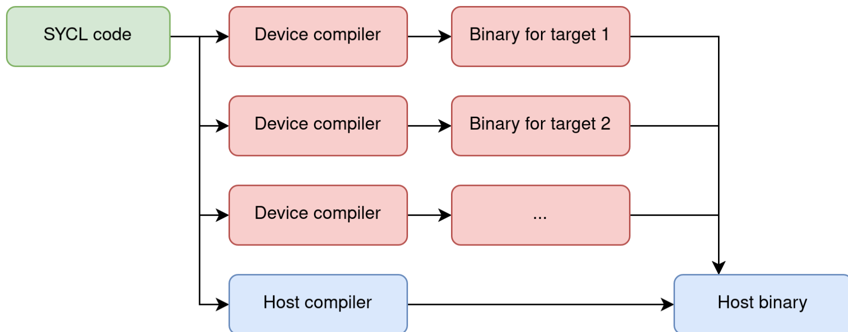
- Consider use cases where SYCL binaries need to be deployed and run on systems with unknown hardware configuration
  - Usually not important for HPC
  - ...but central e.g. for vendors of software who want to distribute binaries to their end users!
  - In this case, a single binary needs to be able to run on whatever hardware is available on the target system.
- How is this adressed in current SYCL implementations?

# Sketch of clang/LLVM architecture




How can this be extended to heterogeneous architectures, and create binaries that run not only on CPU, but also on e.g. GPU?

# SMCP (Single-source, multiple compiler passes)




- Each compiler invocation parses code again, and emits a binary in a backend-specific format
- All SYCL implementations, and almost all heterogeneous compilers (e.g. nvcc) use this model

# SMCP trouble

- Parsing modern C++ code is expensive
- As we add more backends and support for more hardware, compile times do not scale well!
- Some backends (ROCm) do not have an intermediate format – every GPU needs to be targeted explicitly
- Creating a binary using SMCP that runs on all hardware supported by hipSYCL requires roughly 40 compilation passes…(similarly for e.g. DPC++)
- **Very difficult (impractical?) to create a SYCL binary that "just runs" on any GPU**

Portable interchange formats (e.g. SPIR-V) to the rescue?

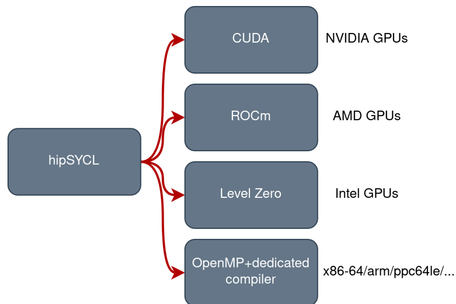
- Not universally supported
- Still want interop with backend-specific libraries, and hence need to rely on vendor-specific SYCL backends (e.g. CUDA/HIP) which rely on their own device binary format

# Towards a solution

1. A single binary device format shared across all SYCL backends and devices would eliminate need for multiple device compilation passes
2. If we could merge the generation of the shared device binary into the regular host compilation, the code would only have to be parsed once!

# Contributions

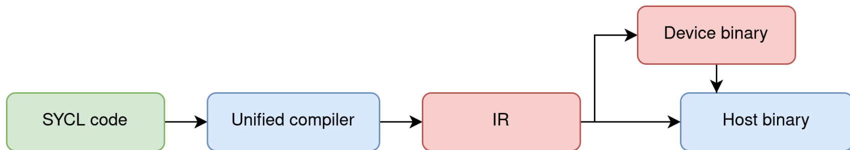
- First single-pass SYCL compiler (source is only parsed once)
- First SYCL compiler with unified code representation across backends (A single device binary embedded in the application that is shared across CUDA, ROCm, Level Zero backends)




All developments are publicly available as part of the hipSYCL SYCL implementation.

# SSCP (Single-source, single compiler pass)

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386




- Code is only parsed once
- Allows for faster compile times
- Not a new idea, but rarely used: Only major production compiler: NVIDIA nvc++. This has never been done in SYCL before.
  - Exception: Implementations of SYCL-as-library for nvc++ (motorSYCL[1], hipSYCL nvc++ compilation flow).
  - In that case, SYCL does not provide the SSCP compiler by itself and has little control.

[1] https://github.com/pkeir/motorsycl

# Idea: SSCP + generic code representation

Combining SSCP and a target/backend-independent device code format would

- Allow us to have a single compiler pass, no matter how many devices we want to run on – **significantly improving compile times**
- Single code representation means we could do analyses and optimizations such as runtime kernel fusion in a backend-independent way – **opening the door to accelerated development of new ideas**
- Make it straight-forward to extend hipSYCL support to new hardware

In practice: Generic code will be transformed to target-specific code at runtime. A single binary will automatically be able to run on “any” GPU.

Adopts the point of view:

- The application adapts its code at runtime to the hardware it finds on the system

# How it works

# General overview

**Stage 1**

- Typically happens at compile time (target device not yet known)
- During regular host compilation, copy LLVM IR generated for host
- Identify kernels and reduce IR to only code that is strictly required for kernels
- Make sure LLVM IR does not contain target-specific hints/builtin calls/...
- Embed LLVM IR bitcode for kernels in host application

**Stage 2**

- Typically happens at runtime (target device known)
- Take generic LLVM IR and compile to backend-specific format for device we want to run on (e.g. NVIDIA PTX)
- Optimize

# About the stability of LLVM IR

LLVM bitcode is not 100% stable across different LLVM versions: Newer LLVM versions can ingest code generated by older versions, but not vice versa.

- Stability of the device code format is less of a concern here than e.g. for OpenCL!
- Device code is generated by our compiler, and then accessed and processed by our runtime, and never leaves control of our stack.
- $\Rightarrow$ Device code format can be seen as an implementation detail
- Only potential issue: Linking different libraries/executables that were compiled with different versions of hipSYCL/LLVM.
- But: This is already not supported today because the hipSYCL runtime library does not have a stable ABI
- $\Rightarrow$ **A device code format with stronger stability guarantees would only inject additional steps into the compilation flow without benefit.**

# SSCP challenges

We need to be able to have different code paths for host/device or different device targets

- Users may want to create dedicated target-optimized code paths
- Implementation needs to figure out how to map e.g. builtins. In SMCP, this can easily be solved using macros:

```
1  double sin(double x) {
2  #ifdef __DEVICE_COMPILATION_PASS__
3    return __device_sin(x);
4  #elif defined(__HOST_COMPILATION_PASS__)
5    return std::sin(x);
6  #endif
7  }
```

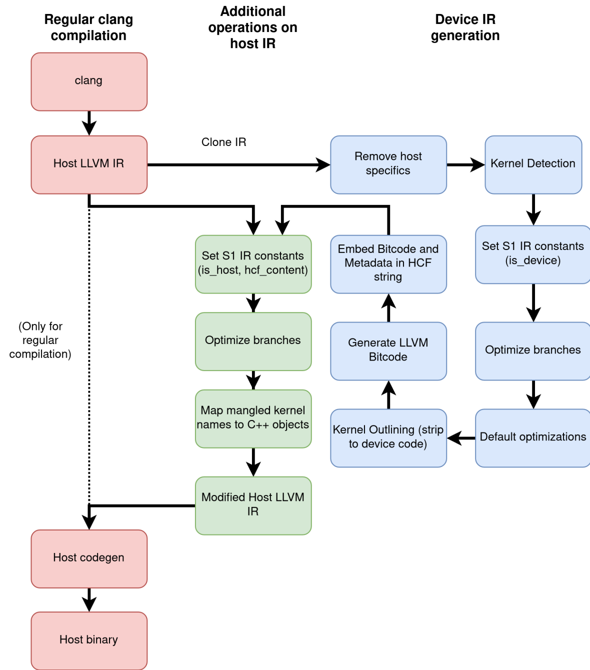
Macros cannot be used with SSCP for this, because code is only parsed once!

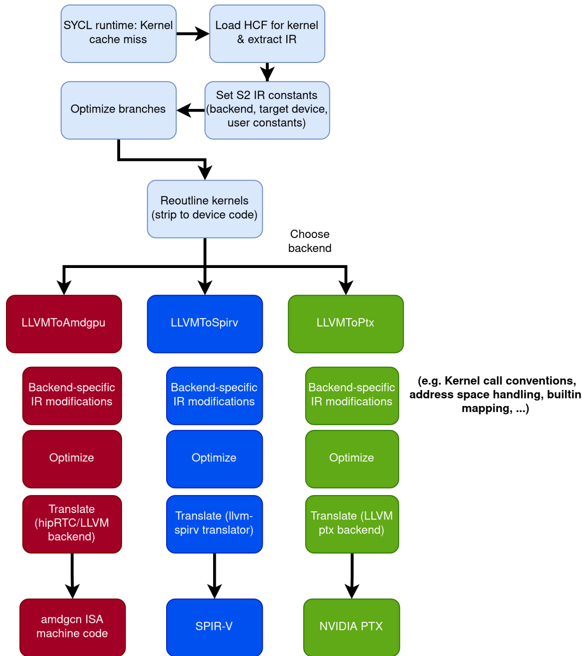
```
double sin(double x) {
  if(__hipsycl_sscp_is_device) {
    return __device_sin(x);
  } else {
    return std::sin(x);
  }
}
```

- `__hipsycl_sscp_is_device` is an **IR constant**
- There are IR constants for stage 1 and stage 2 compilation. Stage 2 IR constants can also be added by the user.
- Since IR constants are seen as constants for the optimizer, additional dead code elimination passes removes unneded branches

No branches will be in compiled code anymore!

**IR constants**: Global variable, which is not a C++ constant, but will be turned into a constant by the compiler in LLVM IR, and initialized with a value that is not known yet when code is parsed.


Regular clang compilation
Additional operations on host IR
Device IR generation
clang
Host LLVM IR
Clone IR
Remove host specifics
Kernel Detection
Set S1 IR constants (is_host, hcf_content)
Embed Bitcode and Metadata in HCF string
Set S1 IR constants (is_device)
(Only for regular compilation)
Optimize branches
Generate LLVM Bitcode
Optimize branches
Map mangled kernel names to C++ objects
Kernel Outlining (strip to device code)
Default optimizations
Modified Host LLVM IR
Host codegen
Host binary




Stage 2 compilation – LLVM-To-Backend Infrastructure:

- Core driver library
- One library per backend
- Backend-specific bitcode libraries
- Tools to carry out stage 2 compilation manually (e.g. for debugging, third-party tooling, ...)

# Current support

- NVIDIA, Intel, AMD GPUs. CPUs are planned.
- Not all SYCL functionality is supported yet
  - Atomics (now implemented)
  - Group algorithms (WIP)
  - SYCL 2020 reductions
  - Some extensions

# Results

Hardware:

- System 1: AMD Ryzen 7 4750U APU, 32GB RAM, Arch Linux
- System 2: Intel Core i7 8550U CPU with iGPU, 16GB RAM, NVIDIA GeForce MX150 GPU, Ubuntu 22.04
- System 3: Intel Core i7 8700 CPU, 64GB RAM, AMD Radeon Pro VII GPU, Ubuntu 20.04

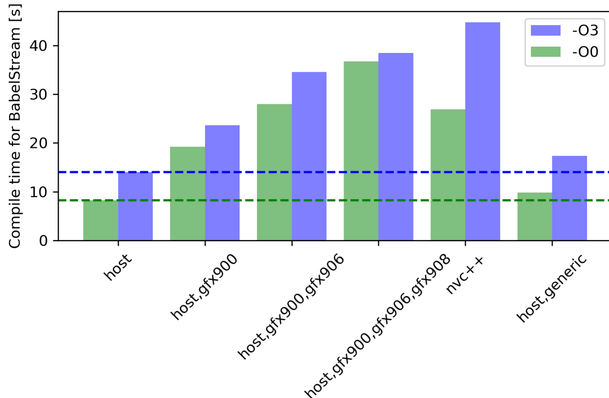
Benchmarks:

- BabelStream (memory benchmarks [Deakin et al. (2016)])
- CloverLeaf (2D hydrodynamics mini-app [Deakin et al. (2020)])
- miniBUDE (molecular docking mini-app [Poenaru et al. (2021)])
- RSBench/XSBench (monte-carlo neutron transport mini-apps [Tramm et al. (2014)])

(details on experimental setup in the paper)

# Compile Time: BabelStream

`host,generic` vs other hipSYCL compilation flows

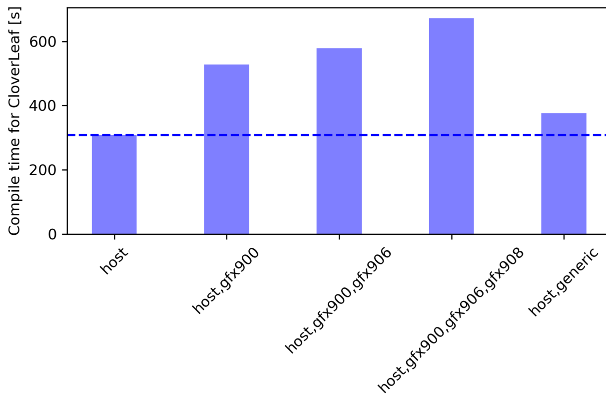




- Generic SSCP compiler only $\approx$ 20% slower than a regular clang host compilation
- This overhead is due to clang OpenMP frontend, disable using `-fno-openmp`
- Note: the other compilation flows are slower, while targeting less hardware!

# Compile Time: CloverLeaf

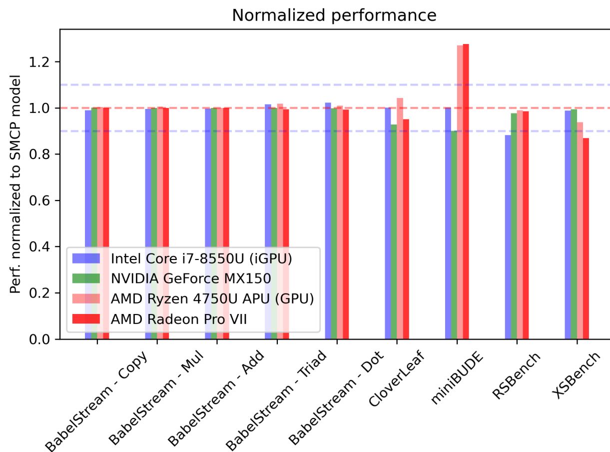
Similar results also for more complex applications!



# But have you just moved compile times to runtime?

- Previous SYCL compilers might do more work then necessary, since all device images for all backends need to be created ahead-of-time, but not all might be needed on the machine of the user
- Stage 2 compilation does not require parsing the code, so parsing costs have been removed
- Note that even in SYCL implementations today there is already runtime compilation taking place!
  - Drivers need to translate backend-specific IRs like PTX, SPIR-V to machine code
  - **SYCL applications need to already expect runtime compilation overheads and deal with this today!**
  - Our additional runtime compilation logic can be expected to add around $0.2\times$ to $1\times$ of the existing runtime compilation overheads to the first kernel invocation (details in the paper)
  - Existing strategies in SYCL apps to deal with overheads at the first kernel invocation will likely still work.

# Performance




- Performance within -13% to +27%
- Intel results are normalized to DPC++ perf, since the old hipSYCL SMCP SPIR-V compiler was too immature

Note: Development focus so far was functionality, not performance!

# Conclusion

- Significantly lower compile times (especially when targeting multiple backends/devices), similar performance, and instant binary portability
- Single binary that can adapt its embedded kernel code based on the hardware it finds on the system
- "Compile once, run (almost) anywhere"
- Robust platform to extend hipSYCL to new hardware, or implement features like profile guided optimizations, kernel fusion, specialization constants in a backend-independent manner
- Publicly available & works today even with complex applications like CloverLeaf
- just compile with `--hipsycl-targets=generic`

- **World's first single-pass SYCL implementation!**
- **World's first SYCL implementation with single, unified code representation that is shared across backends (Level Zero, CUDA, HIP)!**