ABSTRACT
Current SYCL implementations rely on multiple compiler invocations to generate code for host and device, and typically even employ one compiler invocation per required backend code format such as SPIR-V, PTX or amdgcn. This makes generating "universal" binaries that can run on all devices supported by a SYCL implementation very time-consuming, or outright impractical. The ability to generate such universal binaries is however important e.g. when a software vendor wishes to distribute binaries to users that rely on unknown hardware configurations.

To address this issue, we present the very first SYCL implementation with a single-source, single compiler pass (SSCP) design and a unified code representation across backends. This allows a single compiler invocation to generate a binary that can execute kernels on all supported devices, dramatically reducing both compile times as well as the user effort required to generate such universal binaries. Our work is publicly available as part of the hipSYCL SYCL implementation, and supports Intel GPUs through SPIR-V, NVIDIA GPUs through CUDA PTX and AMD GPUs through ROCm amdgcn code.

Our new compiler operates in two phases: At compile time, during the regular host compilation pass, it extracts the LLVM IR of kernels. This IR is then stored in a backend-independent fashion in the host binary. At runtime, the embedded LLVM IR is then lowered to the format required by backend drivers (e.g. PTX, SPIR-V, amdgcn). This approach enables portability of a single code representation even if backends do not support a common code format, while still allowing interoperability with vendor-specific optimized libraries.

We find that our new compiler can generate highly portable binaries that run on any NVIDIA, Intel or AMD ROCm GPU with only 20% additional compilation time compared to a regular clang host compilation. On our test system, this is roughly 2.2× faster than compiling with the existing hipSYCL compiler for just three AMD GPUs.

We also show that the cost of the additional runtime compilation steps can be expected to be approximately comparable to the cost of runtime compilation that backend drivers already perform today, e.g. to lower SPIR-V to machine code.

Lastly, we present early performance results on four different GPUs from three vendors. We find that performance is usually within 10% of current multipass SYCL compiler techniques, with the maximum deviations ranging from a performance regression of 13% to a speedup of 27%. This implies that compared to current SYCL compilation techniques, our new compiler achieves similar performance while substantially decreasing compile times, and increasing the portability of generated binaries.

CCS CONCEPTS
• Software and its engineering → Runtime environments; Parallel programming languages: Compilers.

KEYWORDS
C++, SYCL, oneAPI, compilers, parallelism, heterogeneity, parallel-runtimes, HIP, CUDA, GPU, SPIR-V, LLVM

ACM Reference Format:

1 INTRODUCTION
Current SYCL implementations such as hipSYCL or DPC++ tend to exhibit long compile times, and indeed, the question of compile times has recently attracted the attention of the SYCL community [25]. Compile times suffer especially when devices from multiple backends are targeted simultaneously. This use case of targeting multiple backends at the same time often arises in conjunction with the desire to ship a single "universal" binary that can run on a wide range of hardware, or even on all devices supported by a SYCL implementation.

While this may not be a primary concern in the field of high performance computing where the target HPC system is typically known at compile time, it may be of central importance e.g. for proprietary software vendors who wish to distribute binaries to their users, where the hardware configuration is usually unknown. Such considerations have led to the development of intermediate code representations (IR) such as SPIR-V [16] that are designed as portable code interchange formats. However, these IRs are not yet universally supported as a format that can be ingested by compute drivers supported by hardware vendors. For example, neither the AMD nor NVIDIA OpenCL implementations currently support SPIR-V ingestion.
To circumvent this limitation, SYCL implementations so far have focused on compiling to multiple backend-specific code representations in order to be able to target devices from multiple backends or vendors in a single binary. This however impacts compile times negatively, because code must be compiled multiple times. It also complicates the SYCL implementation, because features related to the compiled device code (e.g. linking) may have to be implemented separately for each code format.

The lack of a unified code representation that can be executed across devices from multiple vendors has already been identified previously as an issue in the SYCL community. Because SPIR-V compute shaders in Vulkan are well supported across GPU vendors, this has sparked the idea of using graphics APIs such as Vulkan as a backend for SYCL implementations [24]. A single Vulkan backend and a single code representation could then be used to target GPUs. This approach however comes with multiple downsides: First, since graphics APIs focus on GPUs, the unified code representation does not generalize well to other devices such as CPUs. Secondly, graphics APIs tend to not expose some functionality that HPC- or compute-oriented programming models support, such as SYCL 2020 shared USM allocations. And finally, it is difficult to access domain-specific optimized vendor libraries, e.g. for linear algebra, from graphics APIs, but easy to access them from vendor-supported compute models such as CUDA. Consequently, it can be desirable to support precisely those compute-oriented APIs as backends in SYCL implementations that are best supported by vendor compute ecosystems, instead of relying on graphics APIs.

In this work, we describe how universal SYCL binaries can be generated without negatively affecting compile times – or even improving compile times compared to the current state of the art, while simultaneously avoiding the issues affecting graphics API backends as mentioned above. To this end, we propose a SYCL implementation where a single compiler invocation can generate both the host code, as well as a representation of device code that is unified across multiple backends. At runtime, this representation is then lowered to backend-specific formats.

As such, our contributions are

1. the first single-pass SYCL compiler and
2. the first SYCL compiler with a unified code representation across multiple backends, as well as mechanisms to execute this code representation on NVIDIA GPUs, AMD GPUs, and Intel GPUs.

Our developments are open source and available on the main develop branch of the public hipSYCL github repository.

1. **SYCL**

The SYCL specification [17] defines a modern heterogeneous programming model based on pure C++. An important property of SYCL is the single source property, which refers to both host and device code residing in the same source file. This is opposed to separate source models such as e.g. OpenCL [20] where the device source code is separate from the host code. The single-source property allows e.g. C++ templates to work seamlessly across the host-device boundary. Multiple SYCL implementations are available, with the most well-known implementations being DPC++ [5], ComputeCpp [23] and hipSYCL [1] which together target a wide range of hardware from CPUs and GPUs from multiple vendors to FPGAs. These implementations however differ substantially in their design, and indeed, SYCL does not mandate one particular implementation choice, and maintains a large degree of implementation flexibility.

1.2 SYCL Implementation Options

The SYCL 2020 specification explicitly mentions in §3.12 several very different implementation choices:

1. Since SYCL expresses heterogeneous parallelism using pure C++, without additional syntactic elements, it is possible to implement it purely as a C++ library for third-party compilers. Such library-only approaches are available as an option for targeting the host CPU in hipSYCL and triSYCL [12]. Additionally, hipSYCL supports targeting NVIDIA GPUs in a library-only manner [2].
2. Arguably the most widely-adopted approach is the single-source, multiple compiler passes (SMCP) model, illustrated in figure 2. Here, a separate invocation of a specific SYCL device compiler generates a device binary. This device binary is then embedded into the application during the host compilation pass.
3. Lastly, the SYCL specification mentions the single-source, single compiler pass (SSCP) model. In this model, a single unified compiler invocation compiles for both host and device, such that code is only parsed a single time. There are two main ways of implementing an SSCP compiler: Early outlining, where device and host code are separated right
after parsing, but before generating any code\textsuperscript{3}, and \textit{late outlining}, where device and host code are separated in the IR. This is illustrated in figure 1. The SSCP model until now has rarely been used in heterogeneous production compilers, with the most prominent example being NVIDIA’s \texttt{nvcc}\textsuperscript{4} compiler. Within SYCL, SSCP compilers so far have only appeared in the context of library-only compilation flows running on top of \texttt{nvcc++}, as supported by hipSYCL \textsuperscript{2} or more recently motorSYCL \textsuperscript{15}. In such a scenario, the SSCP model is merely introduced as a side effect of relying on \texttt{nvcc++}, with little control by SYCL and no portability to non-NVIDIA hardware. Up to this work, to our knowledge there is no standalone, portable SSCP SYCL compiler.

\section{1.3 hipSYCL}

The work presented in this paper is implemented in hipSYCL. hipSYCL (see e.g. \cite{1}, \cite{3}) is a SYCL implementation providing OpenMP \cite{4}, CUDA \cite{12}, ROCm \cite{10} and Level Zero \cite{14} backends. As such, it can target CPUs as well as GPUs from NVIDIA, AMD, and Intel. One aspect that makes hipSYCL particularly interesting as a platform to develop new approaches of implementing SYCL is that it is designed to support a wide range of different compilation models and, already contains support for diverse compilation flows. This includes library-only compilation flows targetting CPUs with OpenMP, clang-based SMCP compilation flows which augment existing toolchains like HIP or CUDA with support for SYCL constructs, optionally with additional compiler support for the host pass to accelerate SYCL constructs on CPU \cite{19}. Additionally, a library-only compilation flow for the CUDA backend using \texttt{nvcc++} is also available. Because \texttt{nvcc++} is also based on an SSCP design, hipSYCL already contains infrastructure in its headers to handle both SMCP and SSCP compilation flows. This in particular includes mechanisms to specialize code for host or device such that they generalize to both SMCP and SSCP cases.

\section{1.4 LLVM and clang}

Our work relies on the clang C++ frontend, and the LLVM compiler infrastructure \cite{15}. The clang and LLVM infrastructure is extensible with custom compiler passes using plugins. This is leveraged by hipSYCL. LLVM provides an intermediate representation, the LLVM IR, which is generated by compiler frontends, and then processed by LLVM optimization passes and LLVM backends. While LLVM IR in principle is backend-agnostic, in practice different LLVM backends may follow different conventions or have different restrictions or requirements regarding the specific constructs in LLVM IR that they accept. Because of this, it is generally not possible to e.g. feed LLVM IR into a backend if the IR was not generated with the specifics of that backend in mind. LLVM IR can be stored either in a textual representation, or a binary bitcode representation.

\section{2 TOWARDS A NEW SYCL COMPILER DESIGN}

\subsection{2.1 Compile Times}

As previously mentioned, current SYCL implementations such as DPC++ or hipSYCL can suffer from long compile times – especially when compiling for multiple backends in an attempt to generate universal binaries. This is because of three reasons:

(1) SYCL is built on modern C+++, which by itself can suffer from long compile times, especially for code relying heavily on C++ templates;

(2) In the SMCP design used by current compilers, at least one additional device compilation pass is required, which exacerbates the issue;

(3) Compile times of current implementations are even more problematic if multiple backends are targeted simultaneously, e.g. in order to create a binary that can run on a range of hardware that is as wide as possible. Because there is no unified code representation across backends, these compilers need to execute one device compilation pass per binary format of the backend (e.g. one pass to generate PTX for CUDA GPUs, one to generate SPIR-V for Intel GPUs etc). Additionally, if a backend does not support a code representation that can run on all of the backend’s devices as is the case for AMD ROCm, then even one pass per targeted device may be required.

Because SYCL is tied to C++, (1) cannot be solved or circumvented easily. (2) however can be solved by moving to an SSCP model where host and device compilation passes are unified. (3) can be solved by additionally introducing a unified code representation across all backends, such that compile times are independent of the number of targeted backends or devices. This unified code representation can then be lowered to backend- or device-specific formats at runtime, when a kernel is executed for the first time on a given device. However, it is important to note that this does not imply that compile times are merely shifted to runtime, but instead there are actual savings:

(1) When generating a universal binary and having to generate separate compiled devices binaries for different backends, we may end up compiling for more targets than the user will ultimately run the binary on. This is because the binary in that case will be compiled to support \textit{all} possible users and their respective hardware configurations, and will be unable to take into account the needs of the \textit{individual user}. This argument applies to any just-in-time or runtime compilation design.

(2) In the SMCP model, the additional compilation passes typically also involve parsing the code again\textsuperscript{5}. In the SSCP model however, code will only be parsed a single time. Even if the steps of lowering the code to target-specific formats will have to be performed eventually, this does not require parsing the code again.

\textsuperscript{3}le relying only on the abstract syntax tree (AST)

\textsuperscript{4}https://developer.nvidia.com/hpc-sdk

\textsuperscript{5}because e.g. different macros might be set depending on the backend or device to allow for optimized code paths
2.2 Implications of SSCP Compiler Designs

The SSCP implementation model differs substantially from the SMCP model. Because this can be noticable by users, it is useful to discuss some of the implications of the SSCP model.

One important difference is how code paths can be specialized between host and device (or different backends, or different devices within one backend). This is not only important for users to create optimized code paths, it is also important for the implementation itself. For example, a SYCL math function such as `sycl::sin()` may have to be mapped to different implementations, depending on the target backend or target device. The SYCL specification requires that SMCP compilers define the macro `__SYCL_DEVICE_ONLY__` during the device compilation pass. Implementations typically define similar macros to denote the backend that is currently compiled for, or even the device\(^6\). These macros can then be used to conveniently specialize code.

In an SSCP design, this is not an option, because there is only a single compilation pass, and consequently there can only be one set of macro definitions during the entire compilation. Instead, in the SSCP model it is more useful to encode information about conditional compilation such that it is represented inside the IR as part of control flow. This is especially true for late outlining SSCP models. For example, nvc++ introduces `if target()` statements for this purpose. These statements obey similar rules as regular if-statements (e.g. they may only appear as part of control flow), however their condition must only depend on values that are known when lowering the IR to machine code. The compiler can then ensure that only the correct branches for the target device are present in the generated machine code.

However, while exposing macros for code specialization is a convenient property of SMCP SYCL implementations, it can also be problematic in other cases. Consider for example the code in listing 1.

```cpp
sycl::queue q;
int* data_ptr = /* some device data allocation */
q.single_task( [=](){
  #ifdef __SYCL_DEVICE_ONLY__
  do_something(data_ptr);
  #endif
});
```

**Code listing 1: Specializing device code using macros**

Here, `do_something(data_ptr)` is only invoked in the device compilation pass. However, there is a subtle issue: C++ lambdas with automatic capture clause (`[=]`) as typically used in SYCL only capture those variables that are actually used inside the lambda body. In the device compilation pass, `data_ptr` is used, and therefore will be stored as part of the lambda object. However, during host compilation, `data_ptr` is not used, because the `__SYCL_DEVICE_ONLY__` macro is not set. This causes the lambda to not capture `data_ptr` during the host pass. Consequently, the data layout of the lambda object as seen during the host and device compilation passes diverge. This can cause undefined behavior when passing the lambda object as an argument to the kernel. In principle, an SMCP compiler could potentially detect and diagnose this situation – but it does require extra effort on behalf of the implementation, and the reported error might still be surprising for the user. In an SSCP model however, such a scenario is prevented by design.

Similarly, the SMCP model can introduce additional obstacles when implementing SYCL 2020 optional kernel lambda naming. This feature enables invoking kernels without providing an explicit kernel name\(^7\), as was required in earlier SYCL versions. When invoking a kernel, the SYCL runtime needs to know the name of the kernel in the device binary, such that the requested kernel can be found in device code. In an SMCP scenario however, host and device code are generated by separate compiler invocations. Because C++ lambda objects do not have a well defined name\(^8\), there is no guarantee that in those different compiler invocations the name that the compiler generates is identical. Generating a stable name that can be used to reliably identify kernels across multiple compiler invocations can be highly non-trivial\(^9\). At the same time, this is not an issue in an SSCP model. In that case, host and device code are generated within the same compiler invocation, and therefore symbol names will be shared by design.

Overall, it is apparent that our suggested design of combining a single-pass compiler with a unified code representation provides advantages beyond compile times: It also simplifies implementing certain SYCL constructs, and can enforce by design that the data layout of objects that are shared across the host-device boundary remains compatible. The unified device code representation additionally allows implementing functionality such as device code linking, SYCL specialization constants, or more experimental future functionality like task fusion in a backend-independent manner.

3 IMPLEMENTATION

We can separate the compilation process of our new compiler into two stages:

1. A first stage, where kernel code is extracted from the host compilation pass and embedded into the host binary in a backend-independent representation. At this stage, the target device or backend that will ultimately execute the code is not yet known.

2. A second stage, where the device code is lowered from the representation generated in the first stage to a representation that can be ingested by a backend. During this phase, it is assumed that the target device is known. Because of this, the second stage typically takes place at runtime when the device code is first executed. However, in principle it could also be executed ahead-of-time (e.g. at compile time) for a particular device.

In the following, we describe how we have implemented this functionality in hipSYCL and LLVM. This is achieved by extending the clang plugin in hipSYCL with additional LLVM passes. Our

---

\(^{1}\)SYCL 1.2.1 requires e.g. `q.single_task<class mykernelname>(...)`.

\(^{6}\)they are typically just enumerated by the compiler during name mangling

\(^{7}\)https://reviews.llvm.org/D103112
implementation follows the SSCP model with late outlining as opposed to early outlining. This choice is motivated by the late outlining model in principle allowing for optimizations across the host-device boundary that are more difficult to implement otherwise.

3.1 Stage 1: Kernel Extraction and Embedding as Generic Device Code

The issue of extracting kernel code in a single compilation pass is closely related to the question of specializing code for host and device code (or different devices) as described in section 2.2. The extraction of kernel code needs to accomplish two interconnected goals:

(1) Untangling the implementation of SYCL functionality for device from their host implementations (e.g. builtins). This is required because hipSYCL also supports dispatching kernels as native C++ code to the host CPU, and SYCL functionality therefore typically needs at least two implementations.

(2) Outlining the kernel code, i.e. detecting kernel entry points and all code reachable from there.

3.1.1 IR Constants. In order to implement (1) and to address the issue of specializing code for different targets as a whole, we introduce what we refer to as IR constants. IR constants are global, non-constant C++ variables, that will be turned by the compiler into a constant with a known value in LLVM IR at a later time. This allows LLVM to treat these variables as constant for the purpose of optimization passes even if their value is not yet known when the C++ code is parsed. These constants can either be set during the first compilation stage (stage 1 IR constant), or the second stage (stage 2 IR constant).

This concept is similar to SYCL 2020 specialization constants. While SYCL specialization constants as a software abstraction can be implemented in many different ways (or even be emulated), IR constants are one concrete mechanism to potentially implement specialization constants.

After setting IR constants, the compiler runs LLVM constant propagation and dead code elimination passes to optimize branches depending on IR constants. This is followed by a kernel outlining pass that removes functions which are no longer used by kernels from device IR, or from host IR if they are no longer used on the host. It is important to note that our additional passes are run as early as possible in the LLVM optimization pipeline such that if-branches are not modified by other LLVM optimizations.

During stage 1 compilation, the supported IR constants are limited to core functionality. Important stage 1 IR constants include is_device which is set to 1 inside device code, and to 0 inside host code, and hcf_content which will be set to a string containing the embedded device image. This allows specializing code for host or device as shown in listing 2.

3.1.2 Kernel Extraction. Kernel extraction works as follows: First, it is ensured that clang does not generate exception handling code inside kernels by attaching the noexcept attribute to all functions that are called in kernel code. This is the only step that takes place inside the clang frontend – all remaining steps operate on the level of LLVM IR.

Code listing 2: Specializing device code in an SSCP model

Once our additional LLVM passes are run, the LLVM IR generated by the normal clang host compilation is cloned. This clone will later become the device code. We first describe how this device IR is processed, and follow with the operations on the original host IR. The important steps that process LLVM IR for stage 1 compilation are also shown in figure 3.

Device IR processing. Kernel calls in the cloned IR are identified, and host specific properties (e.g. LLVM code generation attributes related to the host CPU) are removed. Kernels expect one parameter: A pointer to a data structure parameter that corresponds to the SYCL kernel lambda object or C++ function object. We decompose this parameter into its individual fundamental type members, such that they are passed to the kernel as individual parameters. This causes captured SYCL USM pointers or accessors to be passed as direct parameters to the kernel, which can improve address space inference during stage 2 compilation\(^{10}\). Important stage 1 IR constants such as is_device are set, and if-branches are optimized as previously described. Functions that are no longer part of the kernel function call graph after this step are removed since they are classified as host code. This code is only slightly optimized with a small set of default optimizations mainly intended to clean up the generated code. The majority of performance optimizations will take place later during stage 2 compilation, when the target device is known and all IR constants have been set.

Lastly, we prepare the obtained device IR for embedding it into the host IR, such that the hipSYCL runtime can access the device code. For this, we utilize the hipSYCL container format (HCF)\(^{11}\) which is a flexible file format that can contain an arbitrary number of binary code images as well as text-based hierarchical metadata that provides additional information about the contained kernels and images. Not storing such metadata directly in the LLVM IR allows our approach to function independently of the chosen device code format, which could potentially be subject to future experimentation. Important metadata that is contained in the HCF object includes a list of all kernels as well as their parameters and how they relate to the SYCL kernel function object\(^{12}\). Additionally, a list containing all functions that are referenced, but not defined in the device IR is stored. Similarly, a list is maintained that contains all functions that are exported, i.e. functions having the SYCL_EXTERNAL attribute. Together, these two lists allow the hipSYCL runtime to implement the SYCL_EXTERNAL feature by resolving the imported functions to exported functions from a different translation unit.

\(^{10}\)It also ensures that kernel function signatures are mostly independent of host ABI, which otherwise influences how data structures are passed as arguments.

\(^{11}\)https://github.com/illuhad/hipSYCL/blob/develop/doc/hcf.md

\(^{12}\)I.e., where in the C++ kernel lambda or function object which parameter can be found
at runtime, and linking required IR objects during stage 2 compilation.

At the moment, we directly utilize LLVM bitcode as embedded device code format inside the HCF object. This is an attractive choice because one of our primary motivations was improving compile times, and LLVM IR and its bitcode representation can be obtained with minimal overhead since LLVM IR already occurs naturally during clang compilation. One disadvantage of this approach is however that LLVM IR is not stable across different LLVM versions. Newer LLVM versions can generally read the bitcode representation of LLVM IR generated by older versions, but older versions generally cannot ingest bitcode generated by newer LLVM versions\(^\text{13}\). This could be an issue e.g. when linking a SYCL application with a SYCL library that was compiled with a different version of hipSYCL and LLVM. However, in practice the hipSYCL runtime library already does not make any guarantees regarding ABI stability, and therefore this additional limitation does not negatively affect the status quo. We would like to note that, should a stable format of the device code be required, the generated device code could easily be transformed to a more stable format for storage and interchange such as SPIR-V, and then returned to LLVM bitcode form for stage 2 compilation.

The end product of the device IR processing step is a string containing the HCF data, which in turn contains the device IR as LLVM bitcode and associated metadata.

Host IR processing. Once the HCF data string is obtained, it must be made available to the application by embedding it into the host application. This is accomplished by setting the IR constant string `hcf_content` inside host IR. Additionally, the `is_device` IR constant is set to 0, and branches are optimized similarly to the previously described handling of IR constants in device code. Finally, SYCL kernel lambdas from the C++ source code need to be associated with the name of the kernel function in the IR, such that the runtime can find and execute the right kernel. To this end, a special builtin is provided that maps C++ objects to the mangled kernel function names. At the end of host IR processing, a special pass implements this builtin.

3.1.3 Properties of the Generic Code Representation. To finish the discussion of the stage 1 compilation process, we summarize the properties of the generated device code. The generic representation for kernel code that is generated in the stage 1 compilation process follows the SPMD (single program, multiple data) model that is used by SYCL for kernels. It is LLVM IR without exception handling code, and with a backend-independent representation of SYCL builtins\(^\text{14}\). Calls to these builtins need to be replaced with backend-specific implementations during stage 2 compilation. Normally, LLVM IR intended for offload scenarios contains additional address space annotations for pointers that describe the referenced memory type (e.g., global memory, local memory, constant memory as defined by the SYCL memory model) that the pointee is stored in. In our device IR, there are no specific address spaces used at that point, and the IR refers to all pointers in the generic address space, similarly to regular host IR. This is because address space conventions follow backend-specific rules, and are therefore only handled during stage 2 when the target backend is known.

3.2 Stage 2: LLVM-to-backend Infrastructure

The main goal of stage 2 compilation is to translate the generic device code IR into a format that can be ingested by backends. To this end, we have added the llvm-to-backend infrastructure to hipSYCL, which provides an easily extensible interface to add new targets to the stage 2 compilation process.

Stage 2 compilation is illustrated in figure 4.

Typically, stage 2 compilation is triggered at runtime when the execution of a kernel is requested that is not yet stored as an executable object in the hipSYCL runtime’s kernel cache.

After loading the embedded HCF data and device IR, undefined SYCL_EXTERNAL functions are resolved based on the list of exported and imported symbols in the HCF data, and the necessary LLVM bitcode modules are linked.

Next, stage 2 IR constants are set. While the stage 2 compiler also supports user-defined IR constants, this now in particular includes IR constants that provide information on the target backend or device. Similarly to the IR constant handling during stage 1, this is followed by optimizing branches and reoutlining kernels. Reoutlining kernel code is important in case there are e.g. specialized
Figure 4: Illustration of the stage 2 compilation process

code paths for other backends, which might now be dead code after applying the IR constant that sets the current backend.

Afterwards, the generic LLVM-to-backend infrastructure hands over control to backend-specific mechanisms that transform the generic IR to backend-specific flavors. This includes in particular:

1. Applying kernel and function calling conventions;
2. Adding address space information to pointers and running address space inference passes;
3. Replacing the generic hipSYCL builtins with backend-specific implementations. This is currently achieved by linking with backend-specific bitcode libraries that provide these implementations;
4. Linking with additional bitcode libraries provided by e.g. CUDA or ROCm that implement math functions or lower-level device specific functionality (e.g. libdevice [6] on CUDA)

Afterwards, LLVM optimization passes are run. Currently, we execute LLVM’s default set of optimization passes corresponding to the -O3 optimization level.

Finally, the optimized LLVM IR needs to be transformed to a format that backends can execute. This requires again backend-specific mechanisms:

- The LLVM PTX backend is used to generate code for ingestion by CUDA.
- The LLVM/SPIR-V translator [15] is used to translate code to SPIR-V for ingestion by Intel GPUs. We use a patched version of the translator that interprets the default address space as the generic address space. This is more convenient because it reduces IR rewriting steps when adding address space information to pointers.
- To translate the LLVM IR to amdgcn code, the hipRTC [11] library is used, which supports ingesting LLVM IR starting with ROCm 5.3. Earlier ROCm versions can be supported using LLVM’s amdgcn backend directly. But since hipRTC also automatically links required ROCm bitcode libraries, this approach has the advantage of being more robust.

While the generic LLVM IR is typically translated to optimized backend-specific code at runtime by the hipSYCL runtime library, we also provide standalone tools for each backend to perform stage 2 compilation manually. This is intended for development, debugging, testing, or third-party developments that may wish to access our infrastructure without using the hipSYCL runtime library.

3.3 Implementation Status and Limitations

The generic SSCP compilation flow is currently implemented for SPIR-V, CUDA and amdgcn targets. It is not yet implemented for hipSYCL’s CPU backend. As a consequence, the hipSYCL CPU compilation flow currently has to be enabled in addition to the SSCP compilation flow if kernels are to be dispatched to not only GPUs, but also the host CPU. While implementing the generic SSCP flow for CPUs is planned for the future\(^1\), it is not a priority. Because a host compilation always has to take place\(^2\), we do not expect substantial compile time improvements by also targeting CPUs through the SSCP compiler. At the same time, optimizing compile times was one of our main motivations for this work.

Additionally, some SYCL features are not yet implemented. Most notably, this includes SYCL 2020 group algorithms (except for group barriers), atomics, and SYCL 2020 reductions. Some hipSYCL-specific extensions are also not yet available, such as the scoped parallelism [7] programming model. Adding support for these missing features is planned for the near future with a high priority in order to achieve functional parity with the other compilation flows.

The hierarchical parallelism model from SYCL 1.2.1 is not implemented, but its use is currently explicitly discouraged by the SYCL 2020 specification. Its support was therefore not a priority for this work.

4 RESULTS

4.1 Experimental Setup

In order to evaluate our new generic single-pass compiler, we use the following hardware configurations:

- System 1: AMD Ryzen 7 4750U APU, 32GB RAM, Arch Linux
- System 2: Intel Core i7 8550U CPU with integrated GPU, 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

\(^{15}\)This would e.g. allow hipSYCL kernel code to adapt to available CPU instruction sets when running a hipSYCL application on a different CPU architecture than the application was compiled for.

\(^{16}\)Even if no kernels are needed to run on the host, at least the non-kernel code of the host application has to be compiled.
As mentioned previously, we believe that the most interesting use case for this work is not in HPC, where the target system is typically known, but on systems run by end users where the configuration might be unknown by the application software vendor. This has motivated the hardware selection to include two notebooks (system 1 and 2) as well as a workstation (system 3).

In the following sections, we will be using the following software:
- hipSYCL commit 795cf4c built against LLVM 14
- ROCm 5.3
- CUDA 11.7
- NVIDIA HPC SDK 22.7
- Level Zero loader 1.8.12 and compute runtime 22.43.24595.30
- DPC++ 2022.1.0

As well as the benchmarks and mini apps:
- BabelStream [9][17]
- CloverLeaf for SYCL [8][18]
- miniBUDE [22][19]
- RSbench [26][20]
- XSbench [27][21]

Unless otherwise noted, software compiled from source has been built using -O3 optimization flags, or in cmake release configuration for cmake-based projects.

Note that the development so far has focused on functionality, not on optimizing for performance or compile times. Therefore, the results that we present in the following sections should be considered early results.

4.2 Compile Times

We first look at compile times of our new compilation flow. To this end, we use BabelStream [9], a collection of memory benchmarks (e.g. triad, dot products) for many programming models – including SYCL.

Figure 5 shows the measured compile times on system 2 for both unoptimized (-O0) and highly optimized cases (-O3). Taking into account both of these cases allows assessing the entire range of expected compile times.

Results are shown for six different hipSYCL compilations:

1. **host**: Only hipSYCL’s pure-library compilation flow for CPUs is enabled[22]. In this case, it acts as an OpenMP library for clang. This case serves as a baseline where no SYCL-specific compiler functionality is enabled.
2. **host.generic**: In addition to the host compilation flow, the new generic single-pass compiler is enabled.
3. **host,gfx900,...**: In addition the host compilation flow, one, two and three AMD GPUs are targeted using hipSYCL’s existing HIP compilation support using the SMCP model.
4. **nvcc++**: hipSYCL operates as a CUDA library for nvcc++, another single-pass compiler.

We have decided not to compare against other SYCL implementations, because different supported features and different implementations of the sycl.hpp header in different SYCL implementations will ultimately cause very different code to be compiled. This would make the interpretation difficult. We do however point out that it has been previously shown [25] that hipSYCL’s existing SMCP compilation flows compare roughly similarly to DPC++ in terms of compile times.

Firstly, note that even for the host compilation without optimization, compile times are already close to ten seconds, despite BabelStream being a relatively simple code. This is due to the size and complexity of the sycl.hpp header, which can take considerable time to parse. This is a more fundamental issue with SYCL that might require attention from the SYCL working group. Potential approaches to mitigate this issue could be to split this header into multiple headers, such that client code could only include the required components.

The new generic single-pass flow adds around 20% compile time compared to the host compilation in both the optimized and unoptimized cases. This overhead is however mostly not due to our additional stage 1 compilation logic by itself, which only takes around 0.5s in total. Rather, it is due to a combination of the more complex sycl.hpp header, which now contains additional code paths to handle the generic target, as well as LLVM’s handling of the large embedded string containing the device code.

As expected, because it only parses the code once, the new compilation flow substantially outperforms the older SMCP compilation flow (which takes around 30% longer), even when the SMCP flow targets only a single GPU. When targeting more GPUs, the SMCP flow gets progressively slower due to the additional compilation passes. In the optimized case for three GPUs, it is 2.2× slower than the new compiler.

When comparing the results, note also that the binaries generated by the generic compilation flow cannot run on much more hardware than any of the other tested configurations: The nvcc++ binaries can only run on NVIDIA GPUs. The binaries from the SMCP compilation flow for HIP can only run on exactly those three GPU architectures that were targeted. This is because ROCm does not have an IR. Therefore, to create a binary that is as portable as the one from the generic flow using the SMCP model, many additional compilation passes would be needed: One for every individual GPU architecture supported by AMD ROCm, plus one to generate NVIDIA PTX, and one to generate SPIR-V. Consequently, the compile times for a comparable binary would be even larger.

Finally, even though nvcc++ in principle is also an SSCP design, it is roughly 2.5× slower than our SSCP compiler, and at the same time, produces binaries that can only run on NVIDIA hardware.

BabelStream is a fairly simple code, consisting of only two translation units, only one of which contains kernels. To also investigate the behavior for more complex cases, we additionally present results for CloverLeaf [8]. CloverLeaf is a mini-app for 2D hydrodynamics codes on structured grids. It consists of 29 translation units, several of which contain kernels. Some of its kernels also rely on complex C++ template code. Figure 6 shows the compile times of CloverLeaf in release configuration on system 2. Unfortunately, nvcc++ could not compile the code due to a compiler crash. Again, the new generic compilation flow is around 20% slower than
4.3 Stage 2 compilation time

In the previous section, we have demonstrated improvements of the compile time compared to hipSYCL’s older SMCP and nvc++-based compilation flows. Of course, this omits that the binaries generated by the generic compile flow need to perform stage 2 compilation as an additional step at runtime. Therefore, we want to briefly discuss the impact of this.

First of all, it is important to note that most backends may already perform runtime compilation steps, if they are based on an IR. For example, NVIDIA PTX code will be compiled at runtime by the CUDA driver to machine code. Similarly, SPIR-V code will be lowered at runtime to machine code by drivers. Since many SYCL implementations, including DPC++, ComputeCpp and hipSYCL, rely on backend-specific IR representations to various extents, this is a phenomenon that SYCL applications need to already expect and deal with today. Consequently, we expect that most real world SYCL applications will already be structured such that there is some tolerance for the first kernel invocation to be slowed down by runtime compilation overhead. As such, the question of interest is not how long stage 2 compilation takes, but whether the cost of stage 2 compilation is so much larger compared to existing runtime compilation steps that applications will be negatively impacted.

Stage 2 compilation is generally processed per translation unit, i.e. all kernels from one translation unit are compiled when any kernel from that set is invoked for the first time. However, some backends may have additional compilation triggers. In particular, the SPIR-V backend currently recompiles kernels when a different amount of local memory is requested.

Figure 7 shows the distribution of the time taken by all runtime compilations when targeting NVIDIA GeForce MX150 and running both BabelStream and CloverLeaf on system 2. Runtime compilation consists of two steps:

1. the new stage 2 compilation phase from LLVM IR to backend IR (PTX or SPIR-V)
2. the translation from the backend IR (PTX or SPIR-V) to machine code that the backend drivers perform\(^\text{23}\)

The times have been normalized by the time of the second step, i.e. translation time from backend IR to machine code. As such, the normalized times correspond to the relative increase in runtime compilation time due to our additional stage 2 compilation steps compared to hipSYCL’s older SMCP compilation flows that directly store backend-specific IR in the application binary. Figure 8 shows the distribution when instead targeting the Intel Core i7 8550U iGPU.

For the CUDA GPU, the observed increase in runtime compilation time is between $1.2 \times$ and $2.4 \times$, and between $1.4 \times$ and $2 \times$ for the Intel GPU\(^\text{24}\). This means that we can expect the additional overheads that stage 2 compilation introduces to be roughly similar to the cost of the compilation step from IR to machine code that backend drivers already perform for current SYCL implementations. We conclude that, while there is additional non-negligible cost, the additional steps do not fundamentally change the order of magnitude of runtime compilation costs that SYCL developers can expect from their SYCL implementation. Consequently, SYCL applications designed to tolerate the runtime compilation behavior of current SYCL implementations are likely going to tolerate the additional stage 2 compilation costs as well.

\(^{23}\)\text{i.e. CUDA cuModuleLoadDataEx() or Level Zero zeModuleCreate()}

\(^{24}\)Because ROCm does not have an IR, a similar evaluation is not possible for AMD GPUs.
4.4 Performance

Lastly, we investigate the performance of the generated kernels of the new generic SSCP compiler. In addition to CloverLeaf and BabelStream, we also present results for miniBUDE [22], a compute-bound molecular docking mini-app, as well as the monte-carlo neutron transport mini-apps RSBench [26] and XSBench [26]. RSBench and XSBench by default only run a single kernel once. This makes their results prone to overheads due to data transfers or runtime compilation overhead. In order to obtain more reliable results for the performance of the kernel itself, we have modified their code to execute the kernel multiple times, and average over the measured runtimes while ignoring the first run. See section 4.1 for details on where the modified code can be found. Because the GPUs tested span a wide performance range from integrated GPUs to dedicated workstation GPUs, the problem sizes for the benchmarks had to be adapted in some cases to ensure that the kernels run sufficiently long on the more powerful Radeon Pro VII to allow for robust performance measurements on one hand, but on the other hand not excessively long and within the available resources on the slower notebook GPUs. The selected parameters are shown in table 1.

We are interested in how the kernel performance of the new SSCP compilation flow compares to the existing state of the art. Therefore, performance measurements where taken using both the new hipSYCL SSCP compilation flow, and the older SMCP flow in hipSYCL. Unfortunately, the current SMCP SPIR-V support for Intel GPUs in hipSYCL is very incomplete, and e.g. does not support local memory, which prevents it from running many of the applications. In order to still have a baseline on Intel hardware, we instead compare to DPC++ on Intel GPUs. Because DPC++ by default compiles with the -ffast-math flag, we have added this flag to the hipSYCL SSCP compilation command line in the Intel case in order to obtain comparable results.

Figure 9 shows the obtained results. For the BabelStream benchmark, performance is within 2% for all devices. This indicates that the new compilation flow can extract the same fraction of peak memory bandwidth as SMCP compilers on all backends. Including the mini-apps, the SPIR-V backend achieves performance within 2% in all cases except for RSBench, where it is around 12% slower than the kernel generated by the DPC++ SMCP compiler. One explanation for this is that not all planned optimizations in hipSYCL are implemented yet. For example, DPC++ removes unneeded kernel arguments while hipSYCL still always passes the complete SYCL kernel object to the kernel invocation. Especially in applications such as RSBench which tend to suffer from register pressure, this can be relevant.

The performance on the NVIDIA GPU is within 10% for all applications. The most notable regressions are for miniBUDE and CloverLeaf, where the SSCP-compiled application trails the SMCP case by 10% and 8% respectively. Currently, to translate the LLVM IR to PTX, the new compiler invokes clang with only basic -O3 optimization arguments, while the existing SMCP flow using the clang CUDA toolchain utilizes more complex compiler flags under the hood. Additionally, the SSCP compiler currently always generates code of the same PTX version as the device compute capability instead of the latest PTX version supported by clang. Investigating these differences in the compilation flow in terms of performance and integrating improvements accordingly will be the focus of future work.

The case of both AMD GPUs shows that in some cases, substantial performance improvements are also possible, with the compute-bound miniBUDE application outperforming the SMCP-compiled version by 27% when compiled with the new SSCP compiler. Here, it needs to be taken into account that a different code generation backend is used: For stage 2 compilation, the generic SSCP infrastructure hands over the LLVM IR to hipRTC, which performs the actual code generation and relies on AMD’s ROCm LLVM distribution.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Device</th>
<th>Run parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BabelStream</td>
<td>Radeon Pro VII</td>
<td>-s 160000000</td>
</tr>
<tr>
<td>Others</td>
<td>--file clover_bm64_short.in</td>
<td></td>
</tr>
<tr>
<td>CloverLeaf</td>
<td>Radeon Pro VII</td>
<td>--file clover_bm8_short.in</td>
</tr>
<tr>
<td>Others</td>
<td>--file clover_bm8_short.in</td>
<td></td>
</tr>
<tr>
<td>miniBUDE</td>
<td>Radeon Pro VII</td>
<td>--deck data/bm1 -w 128 -i 128</td>
</tr>
<tr>
<td>Others</td>
<td>--deck data/bm1 -w 128 -i 32</td>
<td></td>
</tr>
<tr>
<td>RSBench</td>
<td>All</td>
<td>-s small -m event</td>
</tr>
<tr>
<td>XSBench</td>
<td>Radeon Pro VII</td>
<td>-s large -m event</td>
</tr>
<tr>
<td>Others</td>
<td>--deck data/bm1 -w 128 -i 32</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Chosen parameters for benchmarks
Overall, we find that the performance of the new compiler infrastructure is usually within 10% of the performance of the current SMCP compiler, with all observed deviations ranging from a regression of 13% (XSBench on Radeon Pro VII) to an improvement of 27% (miniBUDE of Radeon Pro VII). These are very promising early results, considering that so far the development has focused on functionality, and not on performance.

Finally, we point out that kernel performance using the SSCP compiler is mostly independent of the optimization level that was used to compile the application. This is shown in figure 10, where the babelstream performance when using the SSCP compiler and -O0 is compared to the current SMCP compiler using -O3. The reason for this behavior is that kernel optimization takes place at runtime during stage 2 compilation. As a consequence, if an application is known to be bound by kernel performance, it is possible to compile it with the SSCP compiler and lower optimization levels while still generating similar performance. Because lower optimization levels imply faster compilation, this means that the improvements in compile time for the same performance can potentially be even larger than shown in section 4.3.

5 CONCLUSIONS
We have contributed the first single-pass SYCL compiler, and the first SYCL compiler with a unified device code representation across backends. Our developments are publicly available as part of the main hipSYCL github repository.

The new compiler relies on the single-source, single compiler pass (SSCP) model with late kernel outlining, and combines this model with a backend-independent device code representation based on LLVM IR, which is then embedded in the host application. At runtime, when a kernel is first executed, this device code is transformed to backend-specific code formats such as PTX, amdgcn or SPIR-V which can be ingested by device drivers.

This approach allows to generate a host binary that can offload to all GPUs supported by hipSYCL while only parsing the input code a single time during compilation. This is opposed to existing state-of-the-art SYCL compilers such as DPC++ or earlier hipSYCL compilation flows that rely on the single-source, multiple compiler passes (SMCP) model, where the code must be parsed and compiled separately for the host and device, and typically even separately for each backend. Overall, we find that this allows us to generate “universal” binaries that can be run on Intel, NVIDIA and AMD hardware with only a ≈20% increase in compile time compared to a regular clang host compilation. This is a substantial improvement compared to the state of the art, where compiling for merely three AMD GPUs (a small subset of all supported AMD GPUs, and an even smaller subset of all GPUs supported by hipSYCL) takes roughly 2.2x as long on our test system.

Investigating the cost of the additional runtime compilation steps reveals that they roughly take as long as the existing runtime compilation already carried out by drivers to generate machine code. Consequently, while there is additional overhead, the overall order of magnitude of runtime compilation overhead does not change compared to what SYCL applications already need to expect and deal with in previous SYCL implementation approaches.

Performance-wise, we have shown that the generated kernels compete well with the kernels generated by the current state-of-the-art compiler, with performance usually within 10%. The maximum observed performance regression was 13%, while also substantial performance improvements of up to 27% could be observed. These are very promising early results, considering that so far performance optimization has not yet been the development priority.

Additionally, we point out that our solution allows to substantially simplify the process of generating universal binaries from the user perspective, as no specific target needs to be specified.
at compile-time\textsuperscript{25} and compile times reduce significantly, while retaining the same levels of performance. With AMD ROCm not having an IR, it could even be argued that generating universal binaries might even have been impractical previously, since every AMD GPU needed to be targeted individually.

This work also provides several advantages as side effects that might be important for future work: Firstly, it introduces a generic compiler infrastructure that can be extended to new targets. This can substantially simplify porting hipSYCL to new hardware targets. With the Intel GPU support in the new generic SSCP compiler already superseding the previous SMCP SPIR-V support in hipSYCL in terms of functionality, the impact of this is already visible. This work therefore paves the way for robust hipSYCL support on more devices. Furthermore, the unified code representation enables sharing of compiler-related functionality across all backends. For example, SYCL 2020 specialization constants can be implemented in a uniform way, regardless of whether the backend supports specialization constants natively. The same is true for more experimental functionality like kernel- or task fusion. A unified code representation will also open up opportunities for third party tooling to interact with the hipSYCL compiler in a more uniform, well-defined way for all backends, creating a flexible, extensible platform.

All of this creates a robust starting point for future work and exploration. Initially, development priority will focus on achieving feature parity with the existing compilation flows, complemented by performance optimizations and support for the CPU backend through the generic SSCP compiler. However, the new compilation flow also opens the door for many experiments that were previously impossible in hipSYCL. This includes e.g. leveraging runtime compilation to automatically generate kernels that are aggressively specialized based on runtime kernel arguments.

For users, we expect that our work will lead to a more convenient interaction with SYCL: Once the new SSCP compiler is mature and can become the default compilation flow, users will no longer have to specify the targets they wish to compile for in the majority of cases – the generic SSCP compilation flow will automatically target all supported devices.

REFERENCES


\textsuperscript{3} Aksel Alpay, Balánt Soproni, Holger Wünsche, and Vincent Heuveline. 2022. Exploring the Possibility of a HipSYCL-Based Implementation of OneAPI. In International Workshop on OpenCL (Bristol, United Kingdom, United Kingdom) (IWOCL ’22). Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/3529538.3530005

\textsuperscript{4} OpenMP ARB. 2022. OpenMP. https://www.openmp.org/


\textsuperscript{10} Advanced Micro Devices. 2022. ROCm. https://github.com/RadeonOpenCompute/ROCm

\textsuperscript{11} Advanced Micro Devices. 2023. hipRTC. https://github.com/RocketChip/hipRTC

\textsuperscript{12} Ronan Kerrley et al. 2023. trisYCL. https://github.com/trisYCL/trisYCL

\textsuperscript{13} The Khronos Group. 2023. SPIRV-LLVM-Translator. https://github.com/KhronosGroup/LLVM/translator

\textsuperscript{14} Intel. 2023. Level Zero https://github.com/oneapi-src/llvm-level-zero

\textsuperscript{15} Paul Keir. 2023. MotorSYCL. https://github.com/pkeir/motorsycl


\textsuperscript{19} Joachim Meyer, Aksel Alpay, Holger Fröning, and Vincent Heuveline. 2022. Compiler-Aided Nd-Range Parallel-for Implementations on CPU in HipSYCL. In International Workshop on OpenCL (Bristol, United Kingdom, United Kingdom) (IWOCL ’22). Association for Computing Machinery, New York, NY, USA, Article 28, 3 pages. https://doi.org/10.1145/3529538.3530216


