SYCLops: A SYCL Specific LLVM to MLIR Converter

Alexandre Singer
Huawei Canada Research Centre
Markham, Canada
alex.singer@huawei.com

Frank (Fang) Gao
Huawei Canada Research Centre
Markham, Canada
fang.gao1@huawei.com

Kai-Ting Amy Wang
Huawei Canada Research Centre
Markham, Canada
kai.ting.wang@huawei.com

ABSTRACT

There is a growing need for higher level abstractions for device kernels in heterogeneous environments, and the multi-level nature of the MLIR infrastructure perfectly addresses this requirement. As SYCL begins to gain industry adoption for heterogeneous applications and MLIR continues to develop, we present SYCLops: a converter capable of translating SYCL specific LLVM IR to MLIR. This will allow for both target and application specific optimizations within the same framework to exploit opportunities for improvement present at different levels.

CCS CONCEPTS

• **Software and its engineering → Compilers**.

KEYWORDS

SYCL, LLVM, MLIR, IR Converter, Heterogeneous Computing

ACM Reference Format:
Alexandre Singer, Frank (Fang) Gao, and Kai-Ting Amy Wang. 2022. SYCLops: A SYCL Specific LLVM to MLIR Converter. In *International Workshop on OpenCL (IWOCL'22), May 10–12, 2022, Bristol, United Kingdom, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3529538.3529992

1 INTRODUCTION

With the rapid growth of machine learning in recent years, the demand for higher efficiency and performance, specifically for these tasks, have given rise to a wave of domain-specific architectures [3, 12].

SYCL is a standard for single-source heterogeneous programming for acceleration offload and has gained adoption in industry and academia. Existing implementations of *SYCL* compilers are often based on the *LLVM Compiler Infrastructure* project, of which the popular *Clang* compiler is a part of; however, *LLVM* was built to optimize for CPUs, often leaving the *device kernel*, the code to run on the target accelerator, suboptimal.

In order to generate more performant code for accelerators, we introduce *SYCLops*, a converter that can raise the device code from LLVM IR to frameworks expressing higher levels of abstraction (such as *MLIR*), along with the information that the *SYCL* abstraction encapsulated.

IWOCL'22, May 10–12, 2022, Bristol, United Kingdom, United Kingdom

ACM ISBN 978-1-4503-9658-5/22/05…$15.00
https://doi.org/10.1145/3529538.3529992

This work will build on top of *Intel*'s open source *oneAPI* Data Parallel C++ (*DPC++*) compiler based on *LLVM*, which implements the compiler and runtime support for *SYCL* [4].

2 BACKGROUND

2.1 SYCL

SYCL is an abstraction layer upon C++ for constructing programs for heterogeneous systems [1]. It provides an interface for simplifying the management of memory, parallelism, and synchronization of host and device code.

oneAPI's *SYCL* implementation uses a single-source multiple compiler-passes (SMCP) design [4], in which device and host code can be compiled and optimized separately. We will be taking advantage of this design to enable SYCLops only on the device code to generate kernels targeting accelerator hardware, be it *General-Purpose Graphics Processing Units*, *Field Programmable Gate Arrays*, or others.

2.2 LLVM

Under the LLVM project, C++ (and *SYCL* by extension) source code is parsed by *Clang*, and emitted into LLVM intermediate representation (IR). This is to facilitate the use of a common optimization infrastructure for different programming languages and targets. The IR is then optimized through a series of passes before being lowered through a target specific backend to generate the corresponding binary. Before this lowering, however, SYCLops can take the LLVM IR and convert it to *MLIR* for additional optimizations.

2.3 MLIR

MLIR is an open framework for compiler design [14] and a part of the popular LLVM compiler project [13]. It uses static-single-assignment form for its IR, and is designed to be able to mix and match multiple different levels of abstraction at the same time, allowing for a wider range of optimizations. For this very reason, we have chosen MLIR as one of our target backends: to combine high level abstractions offered by *SYCL* with target-specific optimizations necessary for high performance applications.

Special MLIR types of note are the `memref` and `index` types:

- The `memref` type encapsulates the shape of a multi-dimensional array in memory. For example, `memref<2x?xf32>` can be seen as a pointer to a 2D array whose first dimension is 2, second dimension is dynamic, and each element is a single precision floating point.
- The `index` type is MLIR's way of expressing an integer whose width is local to the target machine. The `index` type is used extensively in the *Affine* and *SCF* dialects (see below).

Dialects are a grouping of *Operations* in MLIR that are related to a level of abstraction. There are a few important dialects of note for this project, including the *Affine*, *SCF*, and *Arithmetic* dialects.

Affine Dialect. The *Affine* dialect provides a powerful abstraction for affine analysis and operations. It contains the necessary information to perform polyhedral analysis and transformation on complex loop structures, and will be one of the main target dialects for SYCLops.

Important *Affine* Operations of note are:

- `affine.load`
- `affine.store`
- `affine.for`
- `affine.if`

SCF Dialect. As powerful as the *Affine* dialect may be, it is limited to affine program structures. To represent non-affine control flow, SYCLops will fall back to target the Static Control Flow (*SCF*) dialect.

Arithmetic Dialect. Almost all arithmetic instructions within LLVM IR have their counterparts in the *Arithmetic* dialect. As such, these operations will be generated accordingly.

3 DESIGN PRINCIPLES

To make for a robust and functional converter, SYCLops adheres to the following design principles.

3.1 Extensibility

In order to accommodate for other projects, SYCLops was designed to be extensible to other backends. For example, as opposed to only MLIR, SYCLops should also be able to generate hybrid script that can be used to target AKG (based on TVM); as described by W. Feng, et al [11].

To achieve this, SYCLops will require two modules: a base module, which will be shared between each backend, to handle the interpretation of the incoming LLVM IR and a backend-specific sub module that will translate the interpreted LLVM IR to the target IR. Regardless of the backend, SYCLops needs to understand the SYCL constructs being passed into it, as well as the control flow of the incoming IR; the backend module can retrieve this information from the base module and use it to generate the corresponding IR. Ideally, the backend-specific side of the code generation should never have to decipher any SYCL constructs.

3.2 Preserves Program Structure

The main goal of SYCLops is to convert the input LLVM IR to the target IR. It should not attempt to optimize the code as this may compromise program structure or function.

For targets like MLIR, instead of performing optimizations within SYCLops, it would be more efficient to generate sub-optimal MLIR code and then write MLIR passes to optimize it for a given hardware within the MLIR compiler. This would give the compiler more control over how the code is lowered and what optimizations occur. This is important because different hardware require different optimizations to be optimal; for example, an optimization for a CPU may be sub-optimal or even illegal for an accelerator.

For these reasons, it is best to generate the IR as close to the input LLVM IR as possible.

3.3 Block and CFG Separation

More often than not, the LLVM IR will not lower one-to-one to the target IR. There are instructions and control flow present in LLVM IR that cannot be expressed the same way.

Assuming the target IR is of a higher level of abstraction as compared to LLVM IR, the representation of control flow may be completely different. This is because LLVM IR does not contain dedicated instructions for *Loops* or *If Statements*. Instead it uses *Basic Blocks*, collections of LLVM instructions, linked through branch instructions. Conversely, backends such as MLIR have dedicated operations for these instructions abstracted into the Affine and SCF dialects [2, 8].

However, ignoring the control flow instructions, Basic Blocks themselves should be target-independent. Thus the blocks of the target IR should be generated independent of the control flow.

3.4 Appropriate Error Handling

The error handling of SYCLops should be strict. At no point should it generate invalid code. If SYCLops runs into a situation that it cannot handle, e.g. a control flow that cannot be represented in the target IR, it should throw an error to the user explaining the error and its cause.

The point of this design principle is to simplify the debugging and maintenance of SYCLops to ensure stability and longevity.

4 CONVERTER DESIGN

Figure 1 shows an overview of the SYCLops converter.

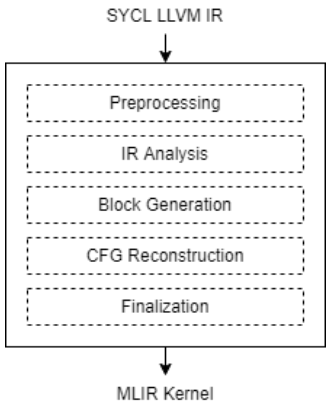


Figure 1: SYCLops design overview.

4.1 Preprocessing

The *Preprocessing* stage transforms the input LLVM IR, making it easier to parse. This greatly simplifies SYCLops, such that the code being converted will always be in an expected form. After this stage, SYCLops will not attempt to transform input IR. This avoids unintended changes in structure during the conversion process.

SYCLops uses LLVM *passes* to transform the IR. Aligning with the *Program Structure Preservation* design principle, these preprocessing passes will not change the function of the input LLVM IR. The *Preprocessing* stage uses two types of passes: *Conversion Simplification Passes* and *Control Flow Simplification Passes*.

The *Conversion Simplification Passes* are preprocessing passes designed to prepare the LLVM IR for instruction generation; where the LLVM Instructions will be converted to the target IR. These passes simplify the *Block Generation* stage of SYCLops by removing specific corner cases.

One of the many Conversion Simplification Passes is the *Simplify Select Logic Pass* which, as seen in Figure 2, turns Boolean select instructions into *AND*/*OR* instructions when possible. These *AND*/*OR* instructions are easier to work with and prevents corner cases when working with Boolean conditions.

```
%AND = select i1 %A, i1 %B, i1 false
%OR = select i1 %A, i1 true, i1 %B
```

(a) Boolean select instructions

```
%AND = and i1 %A, %B
%OR = or i1 %A, %B
```

(b) Resulting *AND* and *OR* instructions

Figure 2: The Simplify Select Logic Pass.

The *Control Flow Simplification Passes* are preprocessing passes chosen to simplify the control flow of the incoming LLVM IR. These passes simplify the *CFG Generation* stage of SYCLops by making the *Control Flow Graph* easier to work with.

The two main passes of interest here are the *Loop Simplify Pass* and the *LCSSA Pass* [6]. The *Loop Simplify Pass* will convert all loops into *Loop Simplified Form* which will ensure that all loops have: a preheader, a single backedge, and dedicated exits [5]; and the *LCSSA Pass* was used to simplify the conversion of ϕ node instructions.

4.2 IR Analysis

After the *Preprocessing* stage, the IR is analyzed and two main pieces of information are gathered from the incoming IR: the shape of the incoming SYCL argument pointers and the control flow.

There are two types of pointers that SYCLops expects as arguments to the kernel: SYCL *Uniform Shared Memory* (USM) pointers and SYCL *Buffer accessors*. The shape information of these pointers are gathered into a specialized *Shape* class that stores general information about the pointer. The *Shape* class will store the rank, the size of each dimension, the element type, and the address space of each pointer.

oneAPI's USM pointers are bare pointers that can be casted to arrays of fixed length in the LLVM IR, as shown in Figure 3a, so the shape information is easily decoded. As shown in Figure 3b, the shape of oneAPI's SYCL Buffers are represented dynamically using *SYCL Ranges* in the LLVM IR. These *SYCL Ranges* may represent static shapes; however they are defined on the host side. So, to SYCLops, which only operates on the device side, SYCL Buffers always appear to have dynamic shape. Consequently, the size of each dimension in the *Shape* class must be stored as pointers to the SYCL Ranges which are also stored as *Shapes*. The *Shape* information will be used to either create static `memrefs` (in the case of USM) or dynamic `memrefs` (in the case of Buffers).

After the *Shape* information, the components of the LLVM loops are gathered using the information provided by the LLVM *Loop Analysis Pass* [6]. *Dominator Tree Analysis* is then used to provide information for generating a *Control Flow Graph* (CFG) of the given IR [6]. This graph will be used in the *CFG Reconstruction* stage to reconstruct the control flow in the target backend.

If SYCLops struggles to analyze the CFG of the incoming IR, it will crash according to the *Appropriate Error Handling* design principle. The control flow that SYCLops struggles with is detailed in section 6: Future Work.

4.3 Block Generation

As explained in the *Block and CFG Separation* design principle, the conversion of the kernel code is separated into a *Block Generation* stage and a *CFG Reconstruction* stage. This design principle simplifies the conversion process by only generating operations into blocks in the *Block Generation* stage, and the *CFG Reconstruction* stage links these blocks together.

Following the *Extensibility* design principle, the previous two stages have been target-independent; thus, they would be found in the base module of SYCLops. However, since this stage is target-specific, the *Block Generation* will be found in the sub module.

The incoming LLVM IR is made up of *Basic Blocks* linked together by branch instructions that form connections between Basic Blocks. For each Basic Block, SYCLops will convert all store instructions into `affine.store` operations. Just before generating each store, however, it must generate the operands of the store op. This happens recursively for the operands until it hits the base case of either a constant or an argument; in which case, the constant or argument will be generated. This recursive approach was done to avoid generating redundant operations.

While iterating over the Basic Blocks, any *loop latch blocks* or *if headers blocks* are collected. After generating all of the blocks, SYCLops will then generate the `affine.for` and `affine.if` operations for the control flow based on the loop latches and the if header blocks it found. However, as explained in the *Background* section, the Affine dialect cannot support all possible *For* and *If* control flows. In these cases, `scf.for` and `scf.if` ops are generated instead.

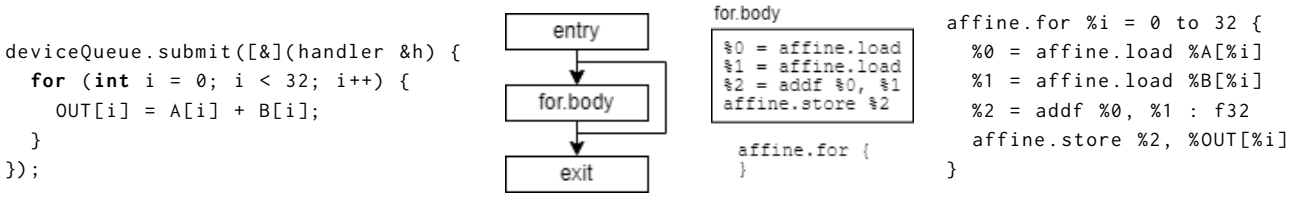
```
%"array" = Type { [16 x [32 x float]] }
define spir_kernel void @ker(%"array" addrspace(1)* %ptr)
{ ... }
```

(a) A SYCL USM pointer cast to a 16x32xf32 array in LLVM IR

```
%"range" = Type { [2 x i64] }
define spir_kernel void @ker(float addrspace(1)* %ptr,
                             %"range" %shape)
{ ... }
```

(b) A 2D f32 SYCL Buffer in LLVM IR

Figure 3: SYCL pointer arguments.



(a) SYCL kernel **(b) LLVM blocks** **(c) Block generation** **(d) CFG reconstruction**

Figure 4: SYCLops converting a SYCL kernel (a) to MLIR (d).

Since MLIR does not contain dedicated operations for ϕ nodes, special consideration needed to be taken when converting these instructions to MLIR. ϕ nodes in LLVM are equivalent to MLIR *block arguments* [7] and are used for values that change based on the block the execution came from; however, in the Affine dialect, one does not have direct access to the block arguments. This is because the block arguments for ops such as the `affine.for` op and the `affine.if` op are used for special purposes. So the ϕ nodes must be analyzed to translate them properly to MLIR.

ϕ nodes, when converting to MLIR, can be one of three cases:

(1) Induction variables, used by *For* ops.
(2) *Iteration Arguments*, used to initialize and update a value within *For* ops.
(3) *Escaping Scalars*, MLIR's version of LCSSA ϕ nodes [7], used to get values from outside of a region.

Pattern matching is used to distinguish which case a given ϕ node belongs to and return an appropriate value as an operand. For case 1, the induction variable of the *For* op, once it is generated, will be returned. For case 2, the ϕ node will be used to generate the *Iteration Argument* and this argument will be returned. For case 3, the ϕ node will be used to generate the yield values for a *For*/*If* op and the result of the *For*/*If* op will be returned.

After this stage, all LLVM instructions that needed to be converted have been generated into their corresponding MLIR blocks and all ops used for control flow have been generated. Thus, with the exception of the *Function* operation which will be generated in the *Finalization* stage, the following stages will not generate any more operations.

4.4 CFG Reconstruction

Now that the operations for the function have been generated. The blocks will need to be linked together according to the Control Flow Graph (CFG) of the incoming LLVM IR.

The *Dominator Tree* from the *IR Analysis* is traversed recursively and precedence of control flow operations is determined based on nodal analysis. In MLIR, blocks cannot be expressed in the same way as LLVM IR, where Basic Blocks are chained together; thus MLIR blocks will need to be merged wherever necessary.

4.5 Finalization

At this point, all of the operations that represent the function are in one block. The *Finalization* stage will take this block and insert it into a function. For MLIR, this would simply create an `std.func` op and merge all operations into its body block.

The arguments to this function will likely not match the arguments of the original kernel; especially for targeting MLIR since not all arguments will have been used and all pointer arguments will have been converted into `memrefs`. However, the user may want to lower the MLIR back to LLVM IR and the generated kernel would have to link to the original kernel. Thus, SYCLops will also generate an LLVM function called the *Trampoline Function*.

The *Finalization* stage will replace the contents of the original LLVM function with instructions that interface to the generated kernel. It does this by creating a call instruction to the kernel and adding instructions that will load and cast the incoming arguments into the types that the kernel expects. This is done so that when the generated MLIR kernel is lowered back to LLVM, it can be linked and inlined back into the original function. As shown in Figure 5, special consideration is taken with respect to `memref` arguments.

```
func @mlir_kernel(%arg0 : memref<16x32xf32, 1>)
{ ... }
```

(a) A kernel in MLIR that takes a single, static memref argument

```
define spir_kernel void @ker(
                        %"array" addrspace(1)* %ptr) {
  %ptr.cast = bitcast %"array" addrspace(1)* %ptr
                                to float addrspace(1)*
  call void @mlir_kernel(float addrspace(1)* %ptr.cast,
                        float addrspace(1)* %ptr.cast,
                        0, 16, 32, 32, 1)
  ret void
}
```

(b) The generated trampoline function call

Figure 5: SYCLops trampoline function generation.

4.6 Summary

Figure 4 shows an example that summarizes the conversion from SYCL C++ (4a) to MLIR (4d) of an element-wise addition kernel.

The converter preprocesses and analyses the LLVM IR generated from the SYCL source file shown in Figure 4a. Each Basic Block of the LLVM IR, Figure 4b, is parsed and used to generate the MLIR blocks shown in Figure 4c. These blocks are then linked together according to the Control Flow Graph to produce the kernel shown in Figure 4d. Finally, the block is inserted into an MLIR `std.func` and a trampoline call is generated that will call the kernel once it is lowered back into LLVM IR.

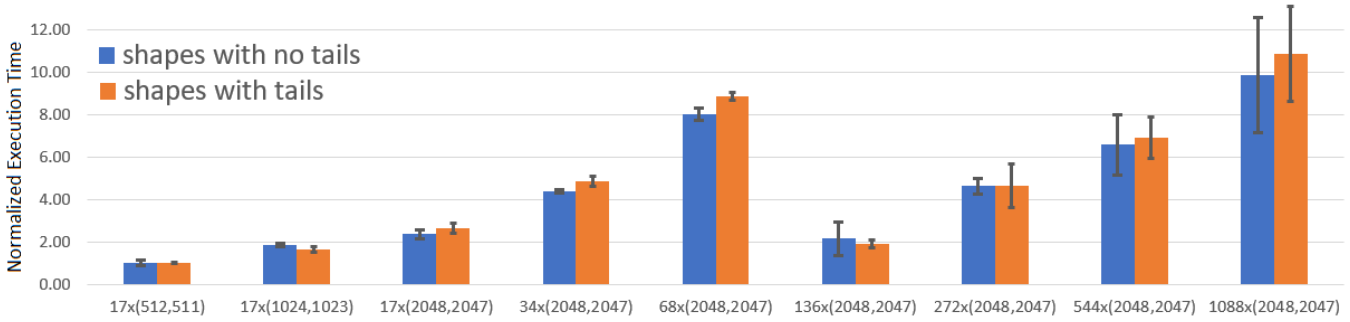


Figure 6: 2D-Relu with varying shapes.

5 EVALUATION

We demonstrate SYCLops at work inside Huawei's SYCL compiler with performance data gathered by offloading tasks to the AICORE device. The measurements are performed on an Ascend 910 server [3]. We demonstrate performance tooling using the CCE plugin interface support [11].

5.1 Software Stack Overhead

To perform our expirements, SYCL device queues were used, which may have overhead affecting our results. We quantify the overhead associated with the queue `submit()` and `wait()` abstractions by measuring the elapsed time (i.e. `end-start`) surrounding the Relu kernel shown in Figure 7. Relu, aka *the rectifier*, is a common *neural network activation function* used in machine learning applications; we use it in our expirements as it is easily tiled and vectorized to many different shapes.

While measuring time using `rdtsc` or reading the `cntvct` register may be more accurate, we use our own timer implementation which utilizes `std::chrono::high_resolution_clock` for portability reasons. We also measure the elapsed time surrounding a direct function call to Relu. The code is compiled at `-O2 -fsycl` and run on host. We apply loops around the `submit()` and `wait()` calls to gain coarser time measurements and to achieve a stable i-cache behaviour.

Overall, the time needed to execute Relu with the queuing abstraction is roughly two times that of a direct function call. However, the absolute overhead remains less than a millisecond. If the workload to be offloaded to a device is sufficiently coarse grained, the abstraction overhead is considered negligible.

5.2 Kernel Scalability Analysis

To demonstrate the scalability benefits of entering MLIR, we perform experiments using the 2D-Relu kernel shown in Figure 7. Our current compiler does not yet support auto-parallelizing and hence only a single AICORE is utilized.

For each testcase, the methodology is as follows: We conduct 3 warm-up runs before taking 10 execution time measurements, where the top and bottom times are evicted; for a total of 8 runs. We then compute the average execution time and standard deviation of these 8 runs.

```
/* 2D Relu with shape NxM */
void relu(_Array *IN_acc, _Array *OUT_acc) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      float input = (*IN_acc)[i][j];
      (*OUT_acc)[i][j] = input < 0.f ? 0.f : input;
    }
  }
}
int main () {
  ....
  const property_list &PropList =
    {sycl::property::queue::enable_profiling()};
  queue deviceQueue(default_selector{}, PropList);

  ....

  /* take start time */
  deviceQueue.submit([&](handler &cgh) {
    auto kern = [=]() { relu(IN_acc, OUT_acc); };
    cgh.single_task<class relu>(kern);
  });
  deviceQueue.wait();
  /* take end time */
  ....
}
```

Figure 7: 2D Relu kernel in SYCL.

We normalize all execution times against the execution time for the shape pair `17x(512,511)`. As shown in Figure 6, we conduct runs for pairs of shapes. That is, the amount of computation between shape pair `17x512` and `17x511` are similar. However, for the second shape in the pair, the total collapsed loop trip count is not divisible by 16 and thus triggers the compiler to generate code for handling the leftover elements during vectorization.

Despite the first shape of each pair always performs more work, the left, blue, bars are consistently shorter than the right, orange, bars except in the cases of `17x(1024,1023)` and `136x(2048x2047)`. This suggests computing the leftover elements using the scalar unit incurs some overhead.

Another interesting observation is that while each shape pair doubles in total size comparing to the previous shape pair, the normalized execution times do not double each time. In fact, the normalized execution time drops significantly going from shape pair

`68x(2048,2047)` to `136x(2048,2047)`. This is due to the fact that at a certain threshold, that lies between these two shape pairs, the double buffering optimization activates. With double buffering, the DMA transfer can be effectively overlapped with the computations, achieving good reduction in execution time.

One last observation is that, as the data ingest volume increases towards larger shape pairs, variability increases. While we chose to place the data on unified shared memory (i.e. with `malloc_share`), we suspect the large data volumes trigger complex interactions between the underlying device memory and the OS virtual memory system. Since large volumes likely span multiple pages, TLB management overhead may play into effect.

5.3 MLIR Optimization Study

To study the effectiveness of MLIR's Affine transformations, such as *Affine Loop Fusion* and *Affine Super Vectorize*, we use the 4D-Sigmoid kernel shown in Appendix A with a single task offload to the AICORE. Following a similar performance methodology, we plot speedups against scalar code performance with both loop fusion and vectorization disabled; this will represent the kernel without any MLIR transformations applied to it, as if it was not converted to MLIR.

The sigmoid kernel is comprised of element-wise exponential, reciprocal, negation and addition operations. The two `affine.for` loop nests are fusible without violating any dependency constraints. As shown in Figure 8, fusion provides a speedup of 1.05x in the case of scalar code performance. This small speedup likely comes from the slight increase in the instruction level parallelism within the fused loop nest and the elimination of the intermediate store and load operations to the temporary buffer. Since buffers are allocated in the fast on-chip cache, and there is no traffic to the global memory between the two loop nests in this case, fusion thus does not harvest the benefit of cache locality.

Vectorization, on the other hand, provides a much more defined speedup. It is clearly a key optimization; however, the end-to-end time is dominated by the DMA operations needed to move input and output data between the global memory and the on-chip cache. As such, reducing the computation time via vectorization, however effective, only renders 1.43x and 1.47x speedups. A kernel with higher compute intensity (i.e. a high compute to data consumption ratio) can be used for future studies to better illustrate the benefit of vectorization.

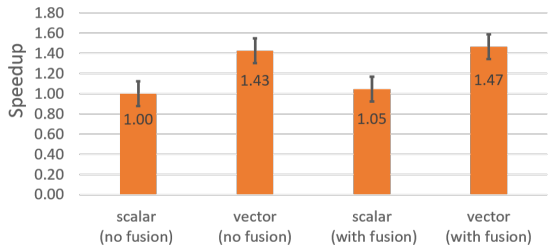


Figure 8: 4D-Sigmoid with fusion and vectorization.

5.4 Converter Functionality Demonstration

Appendix B shows a Kmeans kernel containing imperfect loop nests, Iteration Arguments, escaping scalars, and conditional branching being converted to MLIR using SYCLops. This shows that SYCLops is capable of generating interesting machine learning kernels.

6 FUTURE WORK

The decision to convert SYCL device code from LLVM IR, generated by oneAPI, to MLIR's Affine, SCF, and Arithmetic dialects brings with it limitations that have not yet been addressed. Currently, the SYCLops converter cannot handle complex control flow or many built-in SYCL functions, and we have left these as part of the future work.

6.1 Complex Control Flow

Due to the nature of oneAPI and how it compiles SYCL C++ to LLVM, a complex CFG may arise. Using the transformations and analyses described in the *Preprocessing* section, SYCLops is able to handle basic control flow; however, if the control flow becomes sufficiently complex, SYCLops will struggle to detect the control flow and would be unable to express it within MLIR. In practice we have found the following control flow to be too complex for the current SYCLops converter:

(1) Loops with multiple induction variables or update logic that cannot be expressed using a "step" value; for example, a loop with update logic `i *= 2`.
(2) Kernels with exits that cannot be simplified to a single return or branches that create flow that cannot be expressed as simple *For* or *If* operations.

The complex loops described above (1) cannot be expressed as Affine or SCF *For* operations, however they can be expressed as `scf.while` operations. More logic will have to be put into the *Preprocessing* and *Block Generation* stages to convert these types of loops.

The kernels with complex conditional branching (2) are too complicated to be expressed in the SCF dialect and instead would need to be expressed in MLIR's Control Flow (*CF*) dialect. More logic will need to be added to the *CFG Reconstruction* stage to insert `cf.br` and `cf.cond_br` operations into the kernel.

6.2 Built-In SYCL Functions

As described in §4.19 of the 2020 SYCL specification [1], built-in functions are provided for algorithms, math, and more. In the LLVM IR generated by oneAPI, these built-in functions appear as external functions. SYCLops will try to use the names of these functions to map them to an associated MLIR op. Many of these functions can be expressed in MLIR through its different dialects, such as the *Math* dialect; however not all SYCL built-in functions have an associated operation in MLIR yet.

In order to support all SYCL built-in functions, there are three promising solutions:

(1) MLIR's dialects could be extended to include all possible built-in functions. This would require adding lowering logic for all of these new ops.

(2) SYCLops could convert the functions directly into their basic math operations. For example, the built-in math function "hypot" has no equivalent in MLIR yet; however, since hypot is $\sqrt{x^2 + y^2}$, and all of these operations exist in MLIR, SYCLops could lower hypot directly into these more basic operations.
(3) A SYCL dialect could be created in MLIR. This dialect would contain all of the different functions and types described in the SYCL specifications, including the built-in ops. This dialect would then handle how best to lower to standard MLIR.

We believe solution 3 is the best option. Solution 1 would require many changes to the standard MLIR dialects, in which we cannot guarantee parity between these MLIR dialects and the SYCL specification. Solution 2 could also work, however it puts a lot of strain on SYCLops; it would make more sense to use MLIR's conversion infrastructure to handle these built-in functions. A SYCL dialect would be the best of both of these solutions, simplifying the work required of SYCLops and leveraging the conversion passes present in MLIR. Other teams could also help contribute and maintain this new dialect, keeping it up to date with the SYCL specification.

7 RELATED WORK

Polygeist [15] is a solution proposed by Moses et al detailing a method to take affine C code and lower it to MLIR directly from the Clang AST. It enters MLIR through the SCF and Standard dialects before raising the SCF dialect to the Affine dialect when possible.

Progressive Raising [10] is a technique proposed by Chelini et al that explains that C/C++ code can be lowered to MLIR using the *MLIR Extraction Tool*. This code enters MLIR in the Affine dialect and can then be progressively transformed and raised into the *Linalg* dialect.

The CIL Project implemented the *CIL* dialect [16], which is designed to map C/C++ code directly from Clang.

All of these previous works are designed to lower general C/C++ to MLIR, independent of host or device code; hence why SYCLops exists. SYCLops works with SYCL to lower LLVM IR device code directly to the Affine dialect, while host code remains in LLVM IR.

8 CONCLUSION

We present SYCLops, a converter capable of taking device kernels, written in SYCL, compiled to LLVM IR, and translating them into MLIR. This allows the compiler to take advantage of the flexibility offered by MLIR, before being lowered back into LLVM. We demonstrated the scalability of SYCLops by analyzing a 2D Relu kernel for varying shapes, the benefits of the MLIR optimizations by comparing various Affine transforms on a 4D Sigmoid kernel, and the functionality of SYCLops using Kmeans as an example.

ACKNOWLEDGMENTS

Thank you to Jiashu Wang, Xun Deng, and ZiChun Ye for their IR to IR converter [17] which was the precursor to SYCLops. Also thank you to Rasool Maghareh and Wilson Feng for their work on the CCE plugin interface for SYCL profiling.

REFERENCES

[1] 2020. Khronos SYCL Working Group: SYCL Specification - Generic heterogeneous computing for modern C++. https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf. Revision Date: June 30, 2020.
[2] 2022. 'affine' Dialect. https://mlir.llvm.org/docs/Dialects/Affine/.
[3] 2022. Huawei Ascend 910. https://www.hisilicon.com/en/products/Ascend/Ascend-910. Revision Date: 2022.
[4] 2022. Intel oneAPI DPC++/C++ Compiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html. Accessed: March 31, 2022.
[5] 2022. LLVM Loop Terminology (and Canonical Forms). https://llvm.org/docs/LoopTerminology.html. Revision Date: February 22, 2022.
[6] 2022. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html. Revision Date: February 22, 2022.
[7] 2022. MLIR Rationale. https://mlir.llvm.org/docs/Rationale/Rationale/.
[8] 2022. 'scf' Dialect. https://mlir.llvm.org/docs/Dialects/SCFDialect/.
[9] 2022. sycl-bench. https://github.com/bcosenza/sycl-bench.
[10] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. 2021. Progressive Raising in Multi-level IR. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 15–26. https://doi.org/10.1109/CGO51591.2021.9370332
[11] Wilson Feng, Rasool Maghareh, and Kai-Ting Amy Wang. 2021. Extending DPC++ with Support for Huawei Ascend AI Chipset. In *International Workshop on OpenCL* (Munich, Germany) *(IWOCL'21)*. Association for Computing Machinery, New York, NY, USA, Article 13, 4 pages. https://doi.org/10.1145/3456669.3456684
[12] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (jan 2019), 48–60. https://doi.org/10.1145/3282307
[13] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.
[14] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308
[15] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Affine C in MLIR *(IMPACT 2021)*. 12 pages. https://acohen.gitlabpages.inria.fr/impact/impact2021/papers/IMPACT_2021_paper_1.pdf
[16] Prashantha NR, Vinay Madhusudan, Ranjith Kumar, and Srihari. 2020. *2020 LLVM Developers' Meeting: "Common MLIR Dialect for C/C++ and Fortran"*. Youtube. https://www.youtube.com/watch?v=3gcw-8C9UbA
[17] Jiashu Wang, Xun Deng, Kai-Ting Amy Wang, and ZiChun Ye. 2021. Adapting SYCL's SIMT Programming Paradigm for Accelerators via Program Reconstruction. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21)* (Lemont, IL, USA) *(ICPP Workshops '21)*. Association for Computing Machinery, New York, NY, USA, Article 22, 6 pages. https://doi.org/10.1145/3458744.3473354

A AFFINE TRANSFORMATION EXAMPLES

Figure 10 shows a Sigmoid kernel, generated from a SYCL source file using SYCLops. Using the *mlir-opt tool*, the *Affine Loop Fusion* pass is used to fuse the two loops together and the *Affine Super Vectorize* is used to vectorize the memory accesses.

B CONVERSION EXAMPLES

Figure 9 shows a SYCL Kmeans kernel, adapted from sycl-bench [9], being converted to the Affine dialect using the SYCLops converter.

```
deviceQueue.submit([&](handler &cgh) {
  auto kern = [=](id<1> idx) {
  size_t gid = idx[0];
  if (gid < PROBLEM_SIZE) {
    int index = 0;
    float min_dist = FLT_MAX;
    for (size_t i = 0; i < NCLUSTERS; i++) {
      float dist = 0;
      for (size_t l = 0; l < NFEATURES; l++) {
        dist += ((*features_acc)[l * PROBLEM_SIZE + gid] -
                 (*clusters_acc)[i * NFEATURES + l]) *
                ((*features_acc)[l * PROBLEM_SIZE + gid] -
                 (*clusters_acc)[i * NFEATURES + l]);
      }
      if (dist < min_dist) {
        min_dist = dist;
        index = gid;
      }
    }
    (*membership_acc)[gid] = index;
  }
};
cgh.parallel_for<class kmeans>(range(PROBLEM_SIZE), kern);
});
```

(a) SYCL source code snippet

```
#set = affine_set<()[s0] : (-s0 + 3071 >= 0)>
module attributes {llvm.data_layout = "",
                   llvm.target_triple = "spir64-unknown-unknown"} {
  func @mlir_kmeans(%arg0: memref<3xi64, 1>, %arg1: memref<6144xf32, 1>,
                    %arg2: memref<9216xf32, 1>, %arg3: memref<3072xi32, 1>) {
    %c0_i32 = arith.constant 0 : i32
    %cst = arith.constant 0.000000e+00 : f32
    %cst_0 = arith.constant 5.000000e+05 : f32
    %0 = affine.load %arg0[0] : memref<3xi64, 1>
    %1 = arith.index_cast %0 : i64 to index
    affine.if #set()[%1] {
      %2 = arith.trunci %0 : i64 to i32
      %3:2 = affine.for %arg4 = 0 to 3 iter_args(%arg5 = %cst_0,
                                                 %arg6 = %c0_i32) -> (f32, i32){
        %4 = affine.for %arg7 = 0 to 2 iter_args(%arg8 = %cst) -> (f32) {
          %8 = affine.load %arg1[%arg7 * 3072 + symbol(%1)]:memref<6144xf32, 1>
          %9 = affine.load %arg2[%arg4 * 2 + %arg7] : memref<9216xf32, 1>
          %10 = arith.subf %8, %9 : f32
          %11 = arith.mulf %10, %10 : f32
          %12 = arith.addf %arg8, %11 : f32
          affine.yield %12 : f32
        }
        %5 = arith.cmpf olt, %4, %arg5 : f32
        %6 = select %5, %4, %arg5 : f32
        %7 = select %5, %2, %arg6 : i32
        affine.yield %6, %7 : f32, i32
      }
    affine.store %3#1, %arg3[symbol(%1)] : memref<3072xi32, 1>
  }
  return
  }
}
```

(b) SYCLops output

Figure 9: Kmeans kernel.

```
$ mlir-opt sigmoid.mlir
func @sigmoid(%arg0: memref<8x8x16x32xf32,1>, %arg1: memref<8x8x16x32xf32,1>) {
  %cst = arith.constant 1.000000e+00 : f32
  %cst_0 = arith.constant 0.000000e+00 : f32
  %0 = memref.alloca() : memref<8x8x16x32xf32, 1>
  affine.for %arg2 = 0 to 8 {
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 16 {
        affine.for %arg5 = 0 to 32 {
          %1 = affine.load %arg0[%arg2, %arg3, %arg4, %arg5]
                                : memref<8x8x16x32xf32, 1>
          %2 = arith.subf %cst_0, %1 : f32
          %3 = math.exp %2 : f32
          affine.store %3, %0[%arg2, %arg3, %arg4, %arg5]
                                : memref<8x8x16x32xf32, 1>
        }
      }
    }
  }
  affine.for %arg2 = 0 to 8 {
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 16 {
        affine.for %arg5 = 0 to 32 {
          %1 = affine.load %0[%arg2, %arg3, %arg4, %arg5]
                                : memref<8x8x16x32xf32, 1>
          %2 = arith.addf %1, %cst : f32
          %3 = arith.divf %cst, %2 : f32
          affine.store %3, %arg1[%arg2, %arg3, %arg4, %arg5]
                                : memref<8x8x16x32xf32, 1>
        }
      }
    }
  }
  return
}

$ mlir-opt sigmoid.mlir -affine-loop-fusion -affine-scalrep
func @sigmoid(%arg0: memref<8x8x16x32xf32,1>, %arg1: memref<8x8x16x32xf32,1>) {
  %cst = arith.constant 1.000000e+00 : f32
  %cst_0 = arith.constant 0.000000e+00 : f32
  %0 = memref.alloca() : memref<8x8x16x32xf32, 1>
  affine.for %arg2 = 0 to 8 {
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 16 {
        affine.for %arg5 = 0 to 32 {
          %1 = affine.load %arg0[%arg2, %arg3, %arg4, %arg5]
                                : memref<8x8x16x32xf32, 1>
          %2 = arith.subf %cst_0, %1 : f32
          %3 = math.exp %2 : f32
          %4 = arith.addf %3, %cst : f32
          %5 = arith.divf %cst, %4 : f32
          affine.store %5, %arg1[%arg2, %arg3, %arg4, %arg5]
                                : memref<8x8x16x32xf32, 1>
        }
      }
    }
  }
  return
}

$ mlir-opt sigmoid.mlir -affine-loop-fusion -affine-scalrep \
                        -affine-super-vectorize="virtual-vector-size=8,16,32" \
                        -affine-loop-normalize
func @sigmoid(%arg0: memref<8x8x16x32xf32,1>, %arg1: memref<8x8x16x32xf32,1>) {
  %c0 = arith.constant 0 : index
  %c0_0 = arith.constant 0 : index
  %c0_1 = arith.constant 0 : index
  %cst = arith.constant 1.000000e+00 : f32
  %cst_2 = arith.constant 0.000000e+00 : f32
  %0 = memref.alloca() : memref<8x8x16x32xf32, 1>
  affine.for %arg2 = 0 to 8 {
    %cst_3 = arith.constant dense<1.000000e+00> : vector<8x16x32xf32>
    %cst_4 = arith.constant dense<0.000000e+00> : vector<8x16x32xf32>
    %cst_5 = arith.constant 0.000000e+00 : f32
    %1 = vector.transfer_read %arg0[%arg2, %c0, %c0_0, %c0_1], %cst_5
                                : memref<8x8x16x32xf32, 1>, vector<8x16x32xf32>
    %2 = arith.subf %cst_4, %1 : vector<8x16x32xf32>
    %3 = math.exp %2 : vector<8x16x32xf32>
    %4 = arith.addf %3, %cst_3 : vector<8x16x32xf32>
    %5 = arith.divf %cst_3, %4 : vector<8x16x32xf32>
    vector.transfer_write %5, %arg1[%arg2, %c0, %c0_0, %c0_1]
                                : vector<8x16x32xf32>, memref<8x8x16x32xf32, 1>
  }
  return
}
```

Figure 10: Affine transformations on a Sigmoid kernel.