SYCLops: A SYCL Specific LLVM to MLIR Converter

IWOCL & SYCLcon 2022 – May 2022

Alexandre Singer
(Presenter)

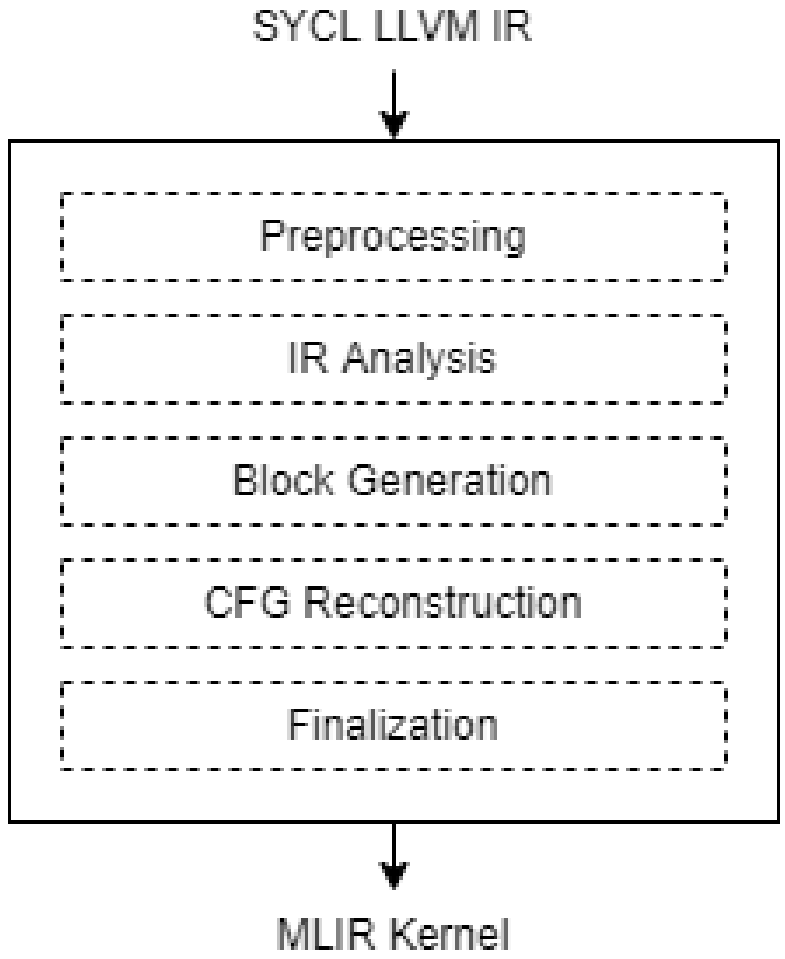
Frank (Fang) Gao

Kai-Ting Amy Wang

Huawei Heterogeneous Compiler Lab, Canada

Agenda

- Background
  - LLVM
  - MLIR
  - oneAPI's SYCL Implementation
- The SYCLops Converter
  - Overview
  - Design Principles
  - Design
- Evaluation
- Future Work



Background: LLVM

https://llvm.org/Logo.html

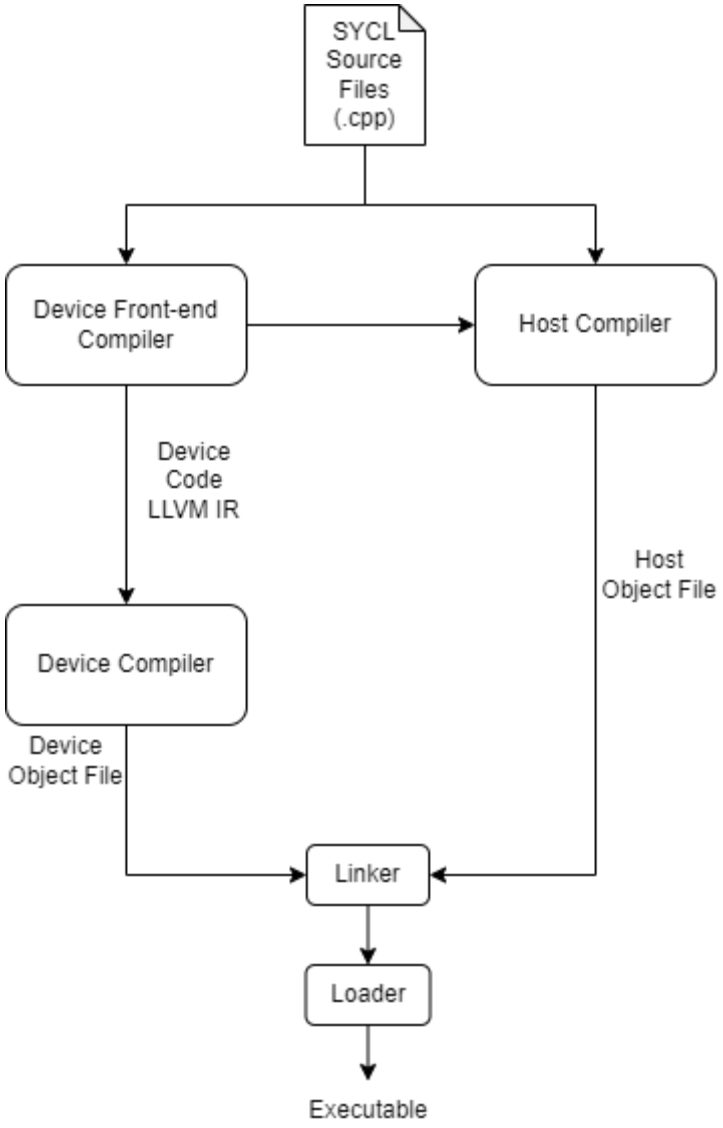
Background: Multi-Level Intermediate Representation (MLIR)

- Simplifies higher-level optimizations.
- Multi-Level → Dialects
  - **Affine Dialect:** affine analysis, control flow, and operations.
  - **SCF Dialect:** non-affine control flow.
  - **Arithmetic Dialect:** arithmetic operations.
- MLIR Types of interest
  - **MemRef:** MLIR's array representation.
  - **Index:** device-specific width integer.

https://mlir.llvm.org/

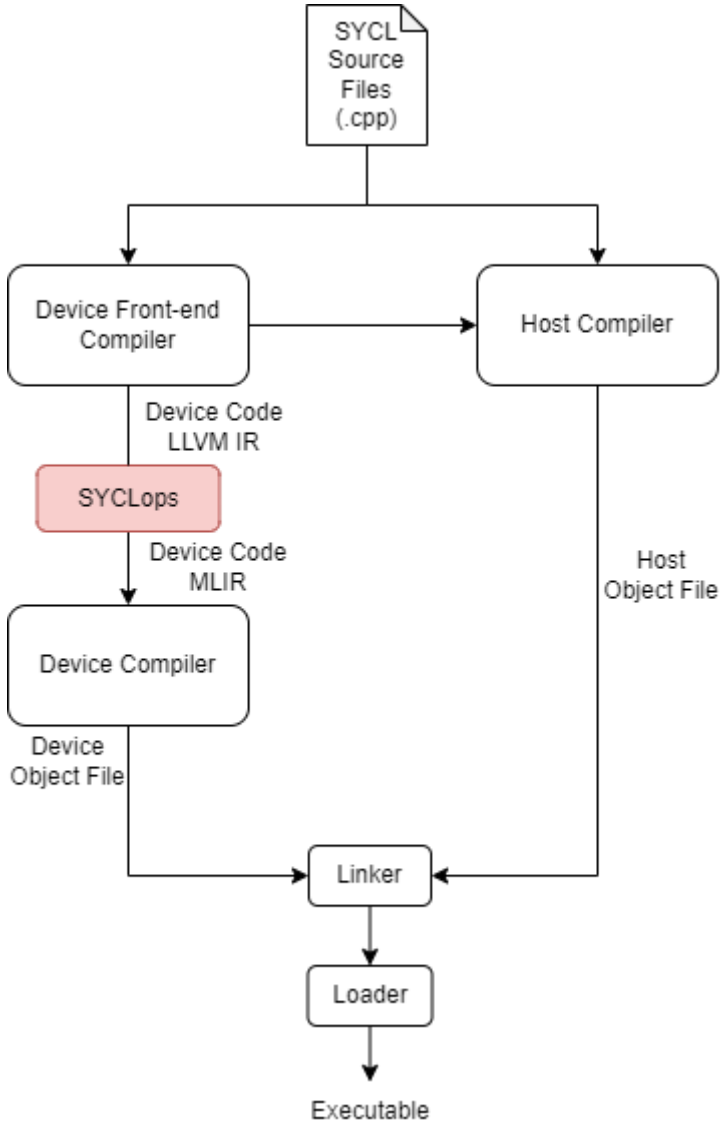
Background: oneAPI's SYCL Implementation

- LLVM-based.
- Compiles the host and device through two separate compilers.
- The Device Front-end Compiler emits LLVM IR to the Device Compiler.
- Device Compiler optimizes and lowers the LLVM IR to an Object File to be executed on the device.
- Problem: oneAPI's Device Front-end Compiler is unable to emit MLIR code.



The SYCLops Converter: Overview

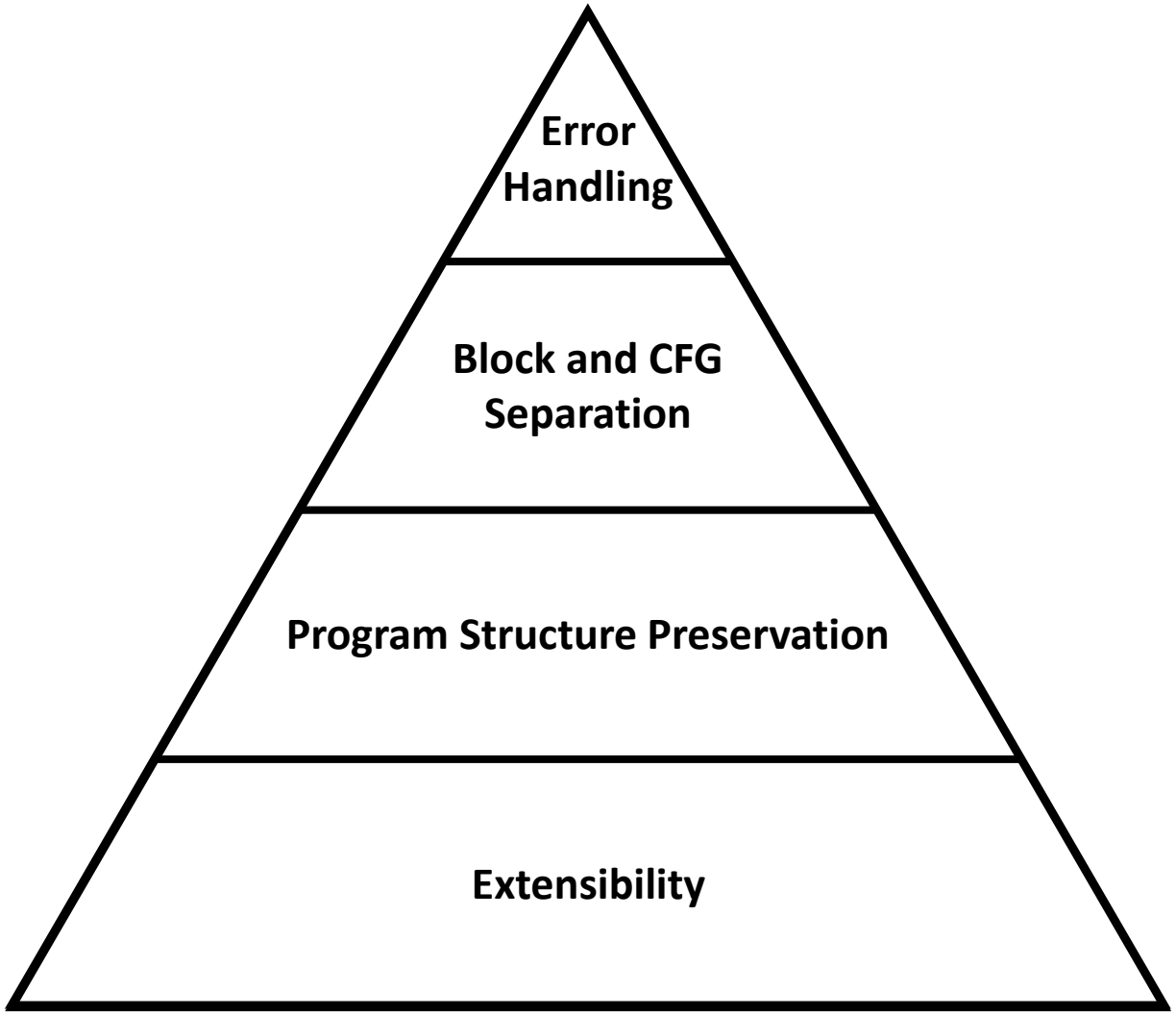
- LLVM IR → MLIR.
- Higher level optimizations within Device Compiler.
- The MLIR code can be lowered back to LLVM IR for more transformations and lowering.
- Minimal changes to oneAPI.



The SYCLops Converter: Why LLVM to MLIR?

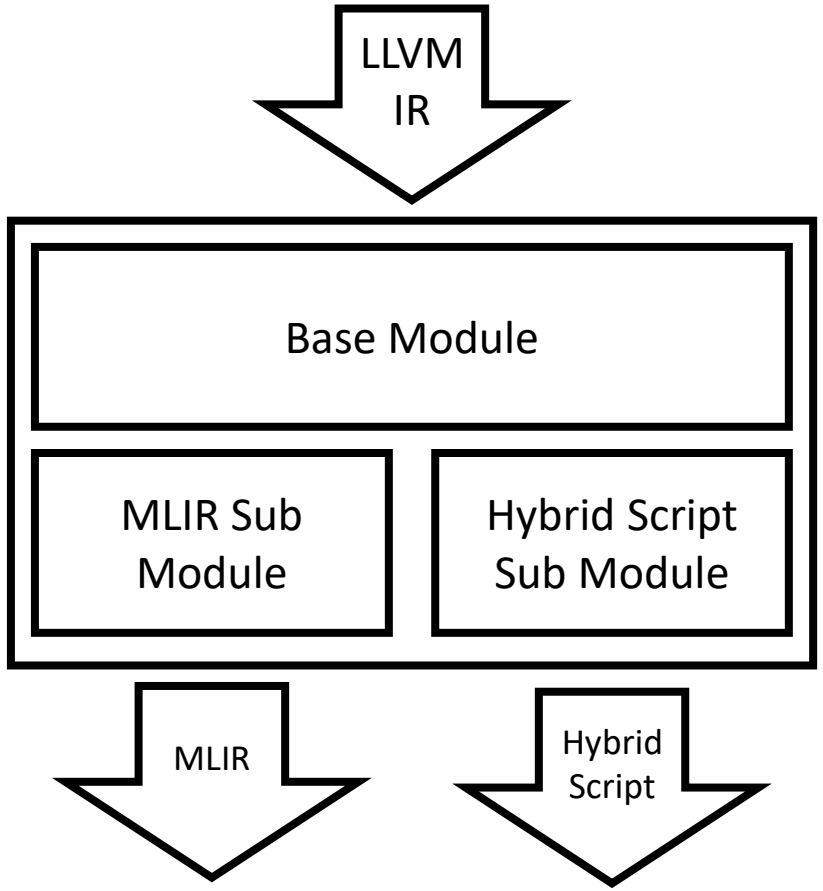
- Why SYCL, to LLVM, to MLIR; instead of SYCL directly to MLIR?
  1. MLIR is relatively young compared to LLVM.
  2. It is easier to enter into LLVM first.
  3. Many other projects are not production ready yet and are for generating general C++ source files, not SYCL source files.
  4. SYCLops was designed to target device code, as opposed to device and host.

The SYCLops Converter: Design Principles



Design Principles: Extensibility

- SYCLops was designed to be extensible.
- Two types of modules:
  1. **Base Module:** Interprets the SYCL constructs and control flow being passed in.
  2. **Backend-specific Sub Modules:** Works with the Base Module to translate the LLVM IR into the target IR.
- As well as targeting MLIR, SYCLops could be used to generate hybrid script that can be used to target AKG (based on TVM).

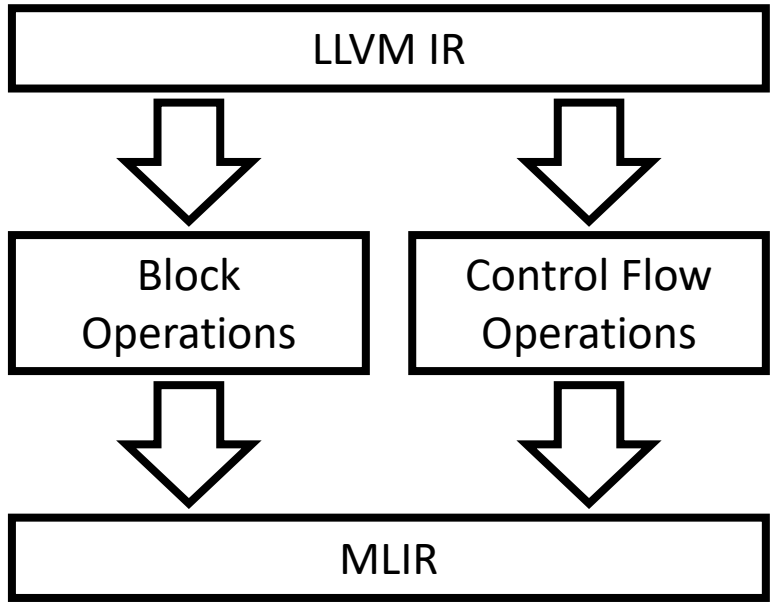


Design Principles: Program Structure Preservation

- Main purpose of SYCLops is to **convert** not **optimize**.
- More efficient to generate sub-optimal code, and then write compiler passes to optimize it for a given hardware.
- Allows for compilers to have more control over the optimizations.
- SYCLops will generate code as close to the incoming LLVM IR as possible.

Design Principles: Block and CFG Separation

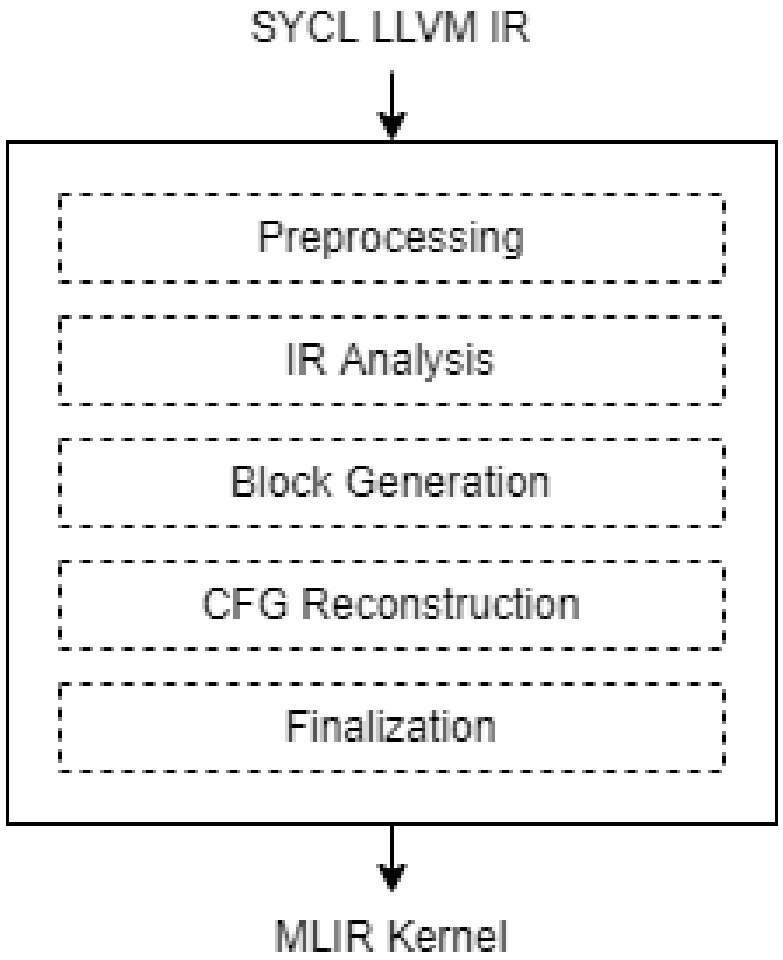
- The control flow of the incoming IR will likely be completely different.
  - LLVM does not have dedicated instructions for *For Loops* or *If Statements*.
- Other instructions, such as arithmetic and memory loads, are likely to be very similar.
- Thus, SYCLops separates the conversion of the control flow instructions from the other Block instructions.



Design Principles: Appropriate Error Handling

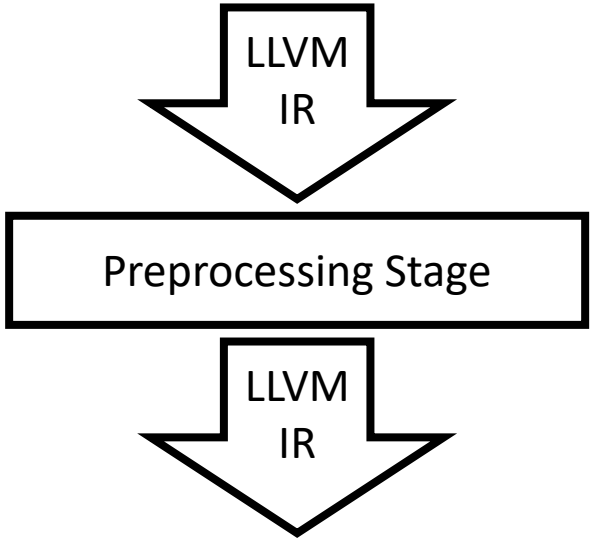
- SYCLops follows strict error handling.
- Unacceptable for SYCLops to emit invalid code.
  - Must convert the incoming IR correctly or crash trying.
- Simplifies the debugging and maintenance of the SYCLops converter.
- Ensures stability and longevity.

The SYCLops Converter: Design



Design: Preprocessing

- Transformation passes designed to simplify the conversion process.
- Ensure that the incoming IR is in an expected form.
- Two types of passes:
  1. **Conversion Simplification Passes**: Prepares the IR for instruction generation.
  2. **Control Flow Simplification Passes:** Simplifies the control flow of the incoming IR.
- After this stage, the IR will not be changed.

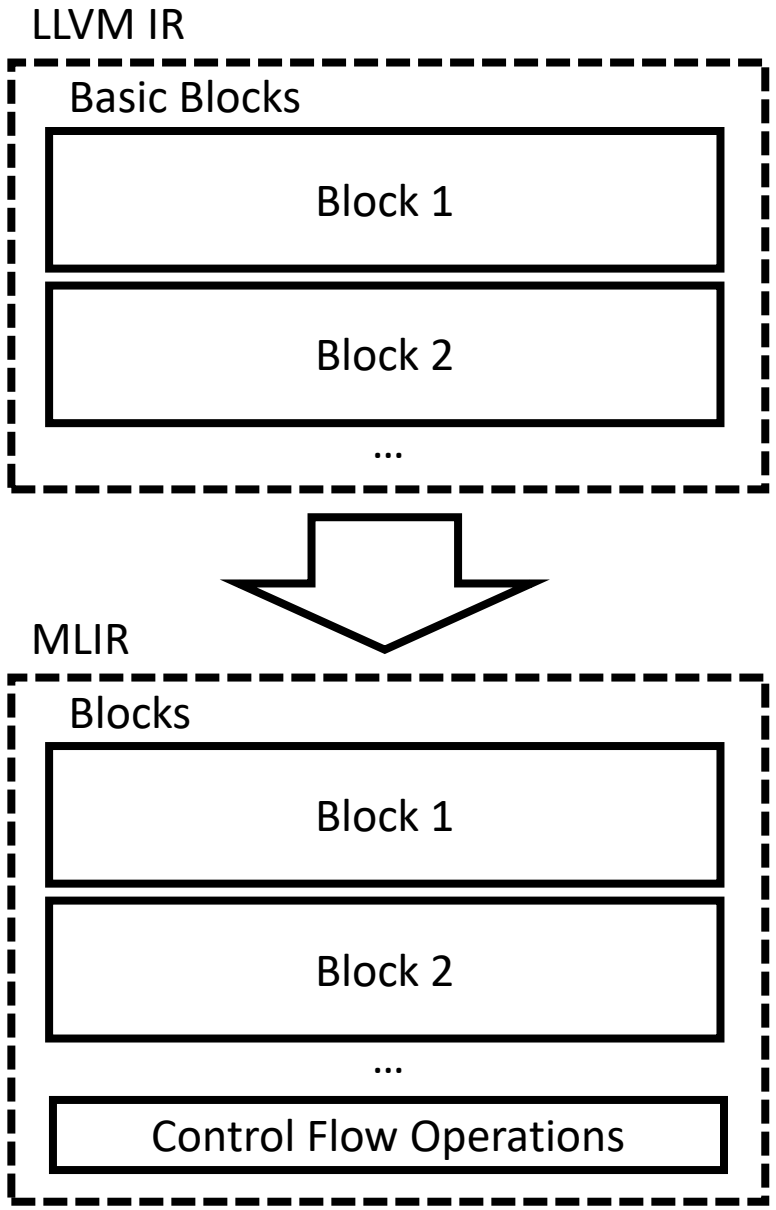


Design: IR Analysis

- Extracts two main pieces of information:
  1. The shape of pointers.
  2. The control flow of the incoming IR.
- Specialized *Shape* class
  - Contains the rank, size of each dimension, element type, and address space of a pointer.
  - The *Shape* objects are stored for use later in the conversion process.
- A Control Flow Graph (CFG) is generated of the incoming IR.
  - Used to reconstruct the control flow in the target IR.

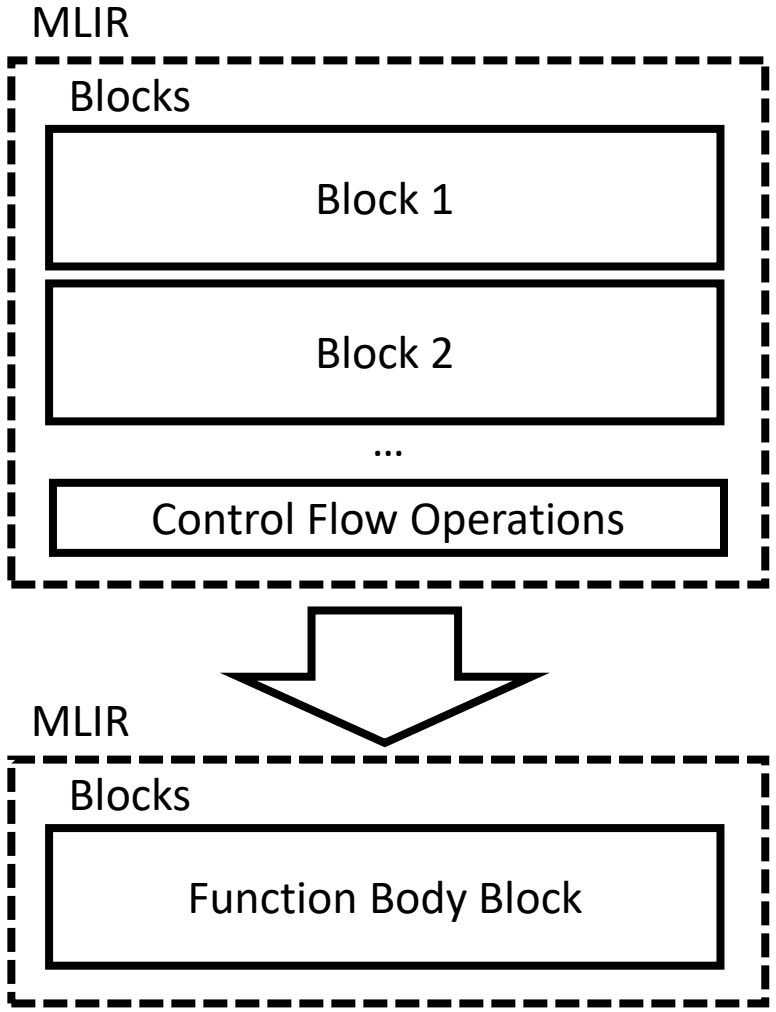
Design: Block Generation

- Where the conversion begins.
- LLVM IR Basic Blocks are traversed, converting non-control flow instructions to MLIR and inserting into MLIR blocks.
- *Loop latch blocks* and *If header blocks* are collected and used to generate *For* and *If* operations after the blocks are generated.
- After this point, all operations have been generated for the given function.



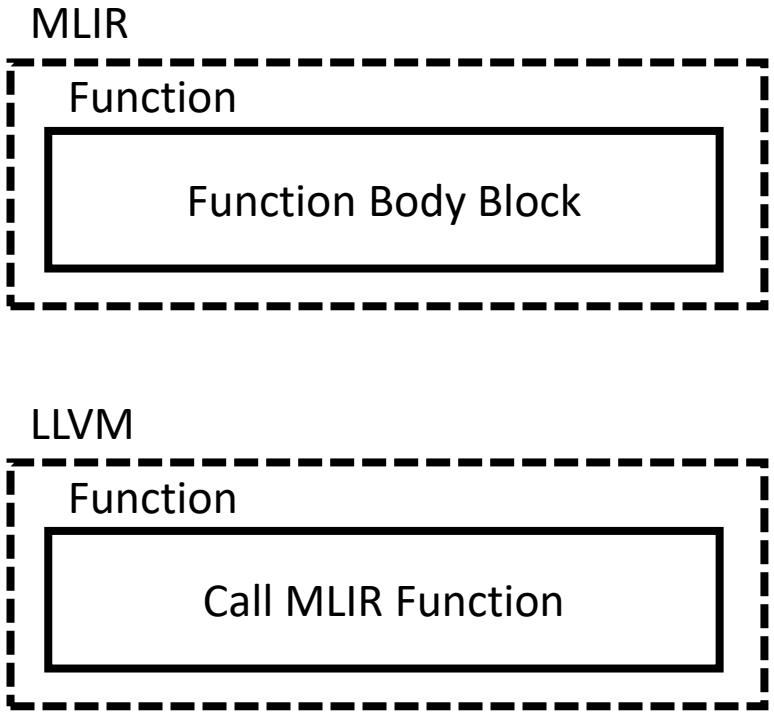
Design: CFG Reconstruction

- MLIR Blocks are not connected together.
- Connects the MLIR blocks and control flow operations together according to the Control Flow Graph.
- CFG is recursively traversed, merging the blocks together and inserting the *For*/*If* operations where needed.
- All operations will exist within a single Function Body Block.



Design: Finalization

- Inserts the Function Body Block into an MLIR func op.
- MLIR represents pointers differently to LLVM and not all function arguments will be used.
  - MLIR Function prototype will not match original LLVM Function prototype.
- A Trampoline Function is used.
  - The original contents of the LLVM function are replaced with a function call which will call the output of the converter as if it were to be lowered to LLVM IR.



Design: Summary

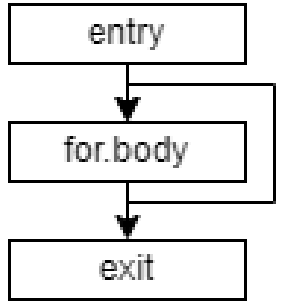
```
deviceQueue.submit([&](handler &h) {
  for (int i = 0; i < 32; i++) {
    OUT[i] = A[i] + B[i];
  }
});
```

oneAPI's Device Front-end Compiler

LLVM Device Kernel Function

Preprocessing

IR Analysis



Block Generation

for.body

```
%0 = affine.load
%1 = affine.load
%2 = addf %0, %1
affine.store %2
```

```
affine.for {
}
```

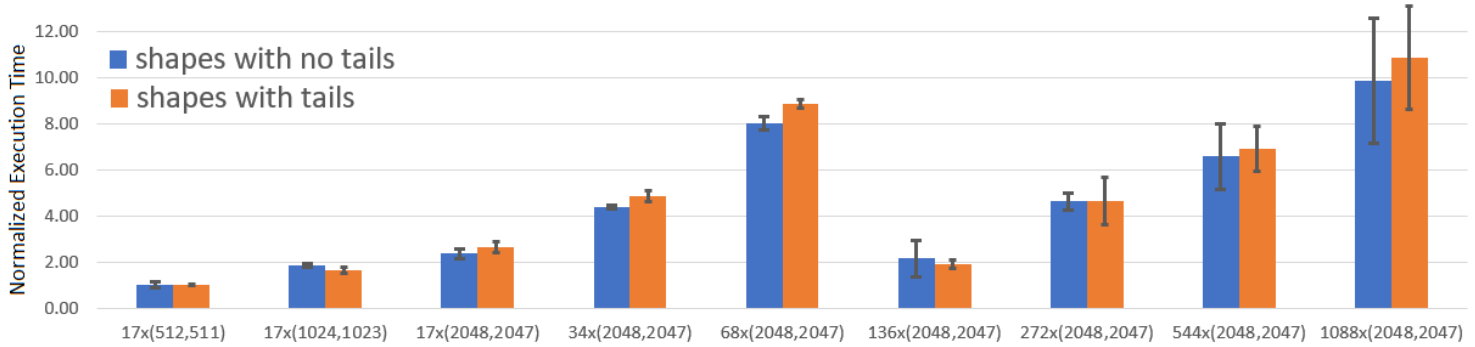
```
affine.for %i = 0 to 32 {
  %0 = affine.load %A[%i]
  %1 = affine.load %B[%i]
  %2 = addf %0, %1 : f32
  affine.store %2, %OUT[%i]
}
```

CFG Reconstruction

Evaluation: Overview

- Perform three experiments:
  1. **Kernel Scalability Analysis:** Does SYCLops perform as expected on a real MLIR Compiler?
  2. **MLIR Optimization Study:** How much improvement does MLIR optimizations provide for a given kernel?
  3. **Converter Functionality Demonstration:** Does SYCLops work on complicated and interesting machine learning kernels.
- All tests are run inside Huawei's SYCL compiler, performed on an Ascend 910 server.

Evaluation: Kernel Scalability Analysis

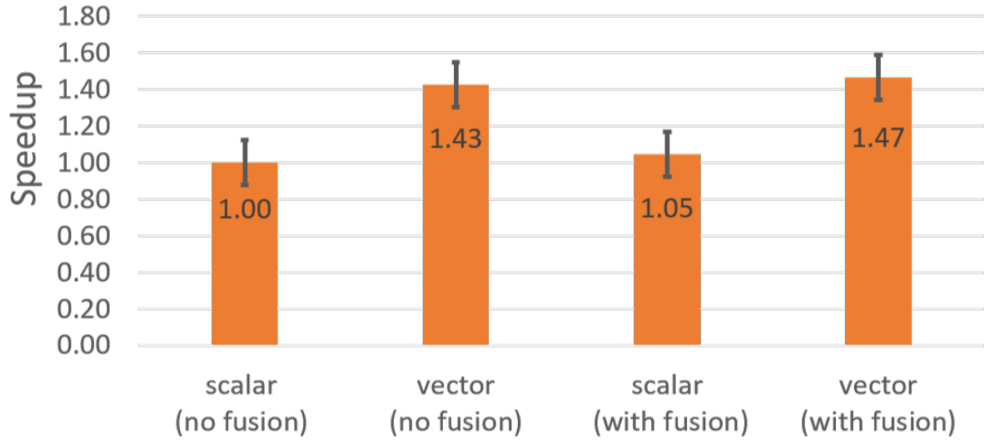


- Experiments were performed on 2D Relu kernels of varying sizes.
- As to be expected, as shape size increases so does execution time; where 32B aligned memory (no tails) is usually faster than non-32B aligned memory (with tails).
  - Consequence of the Ascend hardware's minimum DMA size.
- Dramatic increase in performance between the shapes 68x2048 and 136x2048 is caused by double buffering within the compiler being activated.
- As shown, SYCLops is able to generate legal MLIR code that achieves expected performance results.

Evaluation: MLIR Optimization Study

```
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
}
```

- Sigmoid kernel on the left was generated with SYCLops.
- Investigate two transformations: *Affine Loop Fusion* and *Affine Super Vectorize*.
- Found 1.43x and 1.47x speedup from vectorization and 1.05x speedup from loop fusion.



Evaluation: Converter Functionality Demonstration

```
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);
});
```

Conditional branching

Iteration arguments and escaping scalars

Imperfect loop nest

Kmeans kernel in SYCL C++ source file

```
#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
  }
}
```

SYCLops MLIR output for the Kmeans kernel

Future Work

- There are two main situations that SYCLops is unable to convert:
  1. **Complex Control Flow:** Loops that cannot be expressed in canonicalized form or kernels with exits that cannot be simplified to a single exit cannot be converted by SYCLops yet.
  2. **Built-in SYCL functions:** SYCL has many built-in functions that cannot be expressed in MLIR yet.