Understanding and Extending MLIR for High-Performance Computation - Part I

ECE 718

Da Ma

Brighter World

McMaster University

Introduction to MLIR

- MLIR stands for **Multi-Level Intermediate Representation**.
- It is a **compiler framework**, which can represent computations at multiple abstraction levels (or you may say **IR**s).

Why? Addresses complexity in modern computing (e.g., machine learning, heterogeneous hardware - CPU/GPU/TPU…).

Related Work: LLVM

- Like MLIR, LLVM is a **compiler framework** for building compilers
- LLVM helps compilers translate high-level code (e.g., C++) into machine code.
- It provides optimization passes to improve code efficiency by performing instruction selection, register allocation, loop unrolling, etc., as we have seen in this course.

What MLIR and LLVM exactly are?

Frameworks vs. Compilers

- MLIR and LLVM are **frameworks** that **offer reusable components** such as IRs, optimizations, and code generation features needed to build compilers.
- A **compiler** is the program that actually performs the translation and optimization on high-level source code to make them efficient machine code by using such frameworks.
- Instead of directly converting high-level code into machine code, the compiler frameworks convert the code into **intermediate representations (IR)**. This IR is like a middle step, which allows for many optimizations to be performed before generating the final machine code.

Relationship between MLIR and LLVM

- MLIR: Focuses on **high-level**, domain-specific optimizations (e.g., matrix computations…).
- LLVM: Focuses on **low-level**, hardware-specific optimizations (e.g., register allocation…).
- Workflow:
  - MLIR optimizes high-level abstractions
  - Code is lowered to other LLVM IR
  - LLVM generates machine code
- In conclusion, MLIR is responsible for bridging the gap between high-level frameworks (e.g., TensorFlow) and low-level code generation (handled by LLVM).

MLIR Dialects

- Dialects define the rules for representing and optimizing specific types of operations in MLIR.
- Each dialect corresponds to a specific domain or layer.
- Example dialects in MLIR:
  - Affine dialect: Handles loop operations (useful for loop transformations like tiling).
  - Memref dialect: Handles memory related optimizations
  - And more……

We can DIY any dialects that we need to optimize any specific task!

Demo: An end-to-end process from source language to IRs, and finally results

0 - A new language called “Toy”

- Simple, only serves as an educational example.
- Only data type: tensors in 64-bit floating point type (as “double” in C)
- Only builtin functions:
  - transpose() and print().
  - element-wise multiplication/addition by operand “*”/”+”.
- User can define functions.

```
def multiply_transpose(a, b) {
  return transpose(a) * transpose(b);
}

def main() {
  var a = [[1, 2, 3], [4, 5, 6]];
  var b<2, 3> = [1, 2, 3, 4, 5, 6];

  var c = multiply_transpose(a, b);
  var d = multiply_transpose(b, a);
  var e = multiply_transpose(c, d);
  var f = multiply_transpose(a, c);
}
```

Demo: An end-to-end process from DSL code to IRs, and finally results

1 - Recognize a new function and convert it into a dialect in IR

- **Goal**: Handle normal matrix multiplication (matmul) in Toy.
- Toy language itself doesn't have a built-in understanding of matmul().
- Please don’t confuse the task: We **don’t** fully build it in the language, but we inform the compiler how to handle it as part of the **dialect of Toy**.

```
module {
  toy.func @multiply_transpose(%arg0: tensor<*xf64>, %arg1: tensor<*xf64>) -> tensor<*xf64> {
    %0 = toy.transpose(%arg0 : tensor<*xf64>) to tensor<*xf64>
    %1 = toy.transpose(%arg1 : tensor<*xf64>) to tensor<*xf64>
    %2 = toy.mul %0, %1 : tensor<*xf64>
    toy.return %2 : tensor<*xf64>
  }
  toy.func @main() {
    %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = toy.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>
    %2 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00]> : tensor<6xf64>
    %3 = toy.reshape(%2 : tensor<6xf64>) to tensor<2x3xf64>
    %4 = toy.generic_call @multiply_transpose(%1, %3) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %5 = toy.generic_call @multiply_transpose(%3, %1) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %6 = toy.transpose(%1 : tensor<2x3xf64>) to tensor<*xf64>
    %7 = toy.generic_call @matmul(%6, %3) : (tensor<*xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    toy.print %5 : tensor<*xf64>
    toy.return
  }
}
```

❌ Not Recognizing matmul

Demo: An end-to-end process from DSL code to IRs, and finally results

1 - Recognize a new function and convert it into a dialect in IR

- **Goal**: Handle normal matrix multiplication (matmul) in Toy.
- Toy language itself doesn't have a built-in understanding of matmul().
- Please don’t confuse the task: We **don’t** fully build it in the language, but we inform the compiler how to handle it as part of the **dialect of Toy**.

✔ **Recognizing matmul**

```
module {
  toy.func private @multiply_transpose(%arg0: tensor<*xf64>, %arg1: tensor<*xf64>) -> tensor<*xf64> {
    %0 = toy.transpose(%arg0 : tensor<*xf64>) to tensor<*xf64>
    %1 = toy.transpose(%arg1 : tensor<*xf64>) to tensor<*xf64>
    %2 = toy.mul %0, %1 : tensor<*xf64>
    toy.return %2 : tensor<*xf64>
  }
  toy.func @main() {
    %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = toy.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>
    %2 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00]> : tensor<6xf64>
    %3 = toy.reshape(%2 : tensor<6xf64>) to tensor<2x3xf64>
    %4 = toy.generic_call @multiply_transpose(%1, %3) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %5 = toy.generic_call @multiply_transpose(%3, %1) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %6 = toy.transpose(%1 : tensor<2x3xf64>) to tensor<*xf64>
    %7 = "toy.matmul"(%6, %3) : (tensor<*xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    toy.print %7 : tensor<*xf64>
    toy.print %5 : tensor<*xf64>
    toy.return
  }
}
```

Demo: An end-to-end process from DSL code to IRs, and finally results

2 - Implement optimizations to the dialect, as whatever you want

- **Goal**: Add one possible optimization, here I implemented **the reordering of matrix multiplication chain**.
- A x (B x C) and (A x B) x C has different computational cost, depending on the dimensions of A, B, C.

```
def transpose_transpose(x) {
  return transpose(transpose(x));
}

def main() {
  var a<2, 3> = [[1, 2, 3], [4, 5, 6]];
  var b = transpose_transpose(a);
  var c = matmul(matmul(transpose(a),b),transpose(a));
  print(c);
  print(b);
}
```

There is a chain of matrix multiplication in the Toy source code

Demo: An end-to-end process from DSL code to IRs, and finally results

2 - Implement optimizations to the dialect, as whatever you want

- **Goal**: Add one possible optimization, here I implemented **the reordering of matrix multiplication chain**.
- A x (B x C) and (A x B) x C has different computational cost, depending on the dimensions of A, B, C.

```
module {
  toy.func @main() {
    %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = toy.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>
    %2 = "toy.matmul"(%1, %0) : (tensor<3x2xf64>, tensor<2x3xf64>) -> tensor<3x3xf64>
    %3 = "toy.matmul"(%2, %1) : (tensor<3x3xf64>, tensor<3x2xf64>) -> tensor<3x2xf64>
    toy.print %3 : tensor<3x2xf64>
    toy.print %0 : tensor<2x3xf64>
    toy.return
  }
}
```

- (AxB)xC has a cost of 36.
- Ax(BxC) has a cost of 24. ***<- This is what we have as shown above.***

Demo: An end-to-end process from DSL code to IRs, and finally results

3 - Enable other optimizations for the new dialect

- **Goal**: Achieve **shape inference**.
- Benefits: Better memory allocation, more efficient cache usage, helpful for catching errors such as dimension mismatch, ……

Without shape inference:

```
%4 = "toy.matmul"(%3, %2) : (tensor<*xf64>, tensor<*xf64>) -> tensor<*xf64>
```

With shape inference:

```
%2 = "toy.matmul"(%1, %0) : (tensor<3x2xf64>, tensor<2x3xf64>) -> tensor<3x3xf64>
```

Demo: An end-to-end process from DSL code to IRs, and finally results

4 - Partially lower to lower-level dialects

- **Goal**: Lower the current code into other dialects, here we will do **affine** dialects for matmul.
- What we've done was optimization with our customized dialect "matmul" from a higher level.
- But this is far away from the machine code.
- We need to implement the logic, according to which the compiler can translate the toy dialect to lower level abstractions to approach the machine code.

Let's see how the code is changed with live demo!

Demo: An end-to-end process from DSL code to IRs, and finally results

5 - Lower to LLVM and run the code

- **Goal**:
  - Check the different levels of abstractions.
  - Run the code with JIT to see the execution results when the code is exported to LLVM IR.
- Play with these flags to see the code at different levels:
  - -emit=mlir
  - -emit=mlir-affine
  - -emit=mlir-llvm
  - -emit=llvm

Let’s see how the code is changed with live demo!

Conclusion

- What we just achieved in demo:
  - Implemented a dialect called "matmul" for the Toy language.
  - Defined rules for code handling and optimization specific to this dialect.
  - Implemented the rule of translation this dialect into lower levels.
- How is MLIR meaningful in reality?
  - The same approach applies to **other domain-specific languages** like TensorFlow or PyTorch.
  - People can also implement the optimization rules by dialects to fit for various tasks and environments with different computational resources.

Additional Demo - 1

- **Demo Objective**: Show how MLIR based compiler optimizes computations with loops
- **Example**: Affine loop transformation
- **Tools**: MLIR command-line tools to apply transformations
- **Steps**:
  - Write the code in MLIR
  - Apply optimization passes
  - Observe the transformation

Additional Demo - 2

- **Demo Objective**: Show Abstract Syntax Tree (AST)
- **Tools**:
  - MLIR command-line tools to apply transformations
  - Helper scripts to process MLIR form code and print AST
- **Steps**:
  - Build the simple project
  - Execute it to check AST

Code repo: https://github.com/Da-Dylan-Ma/MLIR-demo/tree/main

Thank you!

Brighter World

McMaster University

References

[1] “MLIR,” mlir.llvm.org. https://mlir.llvm.org/