Recognizing Complex Arthimetic

NICHOLAI TUKANOV, Electrical and Computer Engineering, Carneige Mellon University
CHENGYUE WANG, Electrical and Computer Engineering, Carneige Mellon University

Matrix multiplication is the backbone for a good deal of scientific computing and signal processing applications. Because of this, it is the canonical example for high performance computing experts. Normally, when matrix multiplication is discussed, it is often referred to in the real domain. When extended to the complex domain, the underlying computation is altered, which fundamentally changes the approaches when writing high performance kernels. When expressing complex matrix multiplication, it is best to use a specific kernel that accepts a specific data layout in order to achieve the best performance of your program. However, if it is the case that a higher level language, such as Tensorflow, does not support direct complex arithmetic, then one is forced to use real domain kernels to emulate the complex arithmetic. This leaves performance on the table by not taking advantage of the extra data reuse and special functional units and data layouts for complex arithmetic.

For our research project, we investigate how to write a generic compiler pass using MLIR that replaces emulated complex arithmetic with direct high performance complex arithmetic along with the appropriate layout changes for the targeted hardware. In our search, we initially focused on utilizing Tensorflow to generate MLIR and then run a custom pass on it, but this became infeasible due to a limitation in the current Tensorflow API. Instead, we wrote a simple toy dialect and a respective pass for it to optimize complex arithmetic. With this, we were able to predict the increase in the program's overall throughput by an average factor of 8% for an Intel rocket lake architecture.

Additional Key Words and Phrases: complex, matmul, mlir, tensorflow, hpc, heterogenous computing

1 INTRODUCTION

Matrix multiplication is the backbone for a good deal of scientific computing and signal processing applications. Because of this, it is the canonical example for high performance computing experts. Normally, when matrix multiplication is discussed, it is often referred to in the real domain. When extended to the complex domain, the underlying computation is altered, which fundamentally changes the approaches when writing high performance kernels.

Mathematically, complex matrix multiplication is defined as $C = AB$ where $C \in \mathbb{C}^{m \times n}$, $A \in \mathbb{C}^{m \times k}$, and $B \in \mathbb{C}^{n \times k}$. This operation can be naively computed using four separate real domain matrix multiplications, where assume that the real and imaginary values are stored in separate matrices (shown with a subscript).

$$C_r = A_r * B_r - A_i * B_i + C_r$$
$$C_i = A_r * B_i + A_i * B_r + C_i$$

In languages like Tensorflow, due to a lack of type support, the four real matrix multiplication method (naive 4m) is the only way to represent complex arithmetic within an ML model.

Authors' addresses: Nicholai Tukanov, ntukanov@cmu.edu, Electrical and Computer Engineering, Carneige Mellon University; Chengyue Wang, cw4@cmu.edu, Electrical and Computer Engineering, Carneige Mellon University.

```
@tf.function
def tf_cgemm(Ar, Ai, Br, Bi, Cr, Ci):
    Cr += tf.matmul(Ar, Br) - tf.matmul(Ai, Bi)
    Ci += tf.matmul(Ar, Bi) + tf.matmul(Ai, Br)
```

However, by casting complex matrix multiplication into this method, we lose the ability to take advantage of specialized complex high performance kernels, since the code is compiled down to 4 separate calls to a real domain matrix multiplies. The naive 4m method also forces the user to use the de-interleaved complex format, which separates real and imaginary values into two different matrices. This goes against the BLAS standard [1], thus adding more overhead to the system.

For our project, we investigate how to implement a compiler pass to replace the naive 4m method of complex arithmetic with a high performance kernel, converting the data layout if necessary. Given that our pass operates at the level of matrix multiplication operations, we focus on using the MLIR compiler stack to write this pass due to its ability to concisely represent optimizations on high level tensor operations [2–4]. Initially, in our search, we focused on using Tensorflow as the target language in which the naive 4m method is written in and the respective MLIR is generated for. But, due to an inability in the current Tensorflow compiler infrastructure API [5], it is not possible to write a custom pass in time for the project. Thus, we utilized a toy dialect designed for simple a complex matrix multiplication, for which we wrote a pass using the MLIR's OperationPass [4] infrastructure.

So far, there has not been anyone that has attempted to write a tensorflow compiler pass that converts naive 4m methods to an optimized kernel, with a data format change if necessary. However, the Tensorflow compiler has seen a lot of work put into as of recently. Specifically, the Tensorflow compiler team has been focusing on an API rewrite for the compiler pass using their existing MLIR infrastructure [5]. Moreover, there has been a lot of work towards creating new compilers with MLIR for other domain specific languages like PyTorch, as well as, architecture specific languages for things like the Xilinix AIE [6–10]. We also see in the world of high performance computing, there has been Bondhugula who used MLIR to generate optimized double precision real domain matrix multiplication kernels and compare them against expert implementations [11].

Contributions.

(1) We showcase the need for having a compiler pass to convert the naive 4m method into an optimized kernel call by comparing standalone implementations.
(2) We demonstrate how to write such a pass on a toy MLIR dialect designed for complex matrix multiplication using the MLIR compiler stack. Through our generated code from the pass, we predict that the program's overall throughout increases by an average of 8% for various square problem sizes.

2 IS NAIVE 4M BAD?

Before diving into writing the pass, we first ensured ourselves that this optimization will result in better performance, specifically better operation throughput. Therefore, we wrote standalone implementations of the naive 4m and high performance complex kernels. The kernels target the VNNI instruction set on an Intel rocket lake

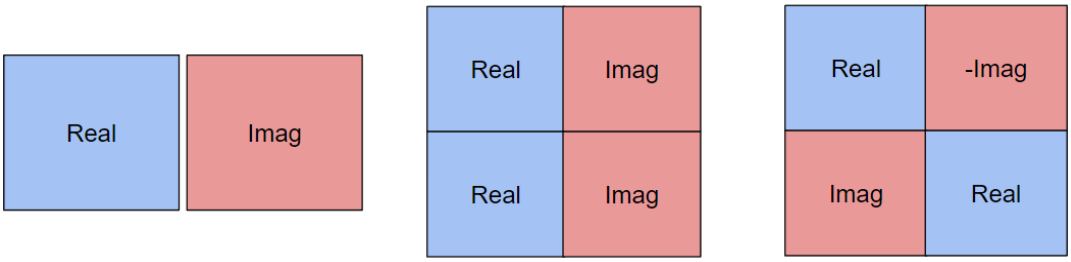

Fig. 1. Various layouts for different high performance complex kernels. Each block represents a matrix of values. Leftmost: de-interleaved, middle: tiled de-interleaved, rightmost: special 1m format [12]

.

machine, which has a theoretical peak throughput of about 325 GFLOPS/core. All tests that were conducted were within C/C++ and compiled with GCC 11. All performance that is reported on uses a single thread of execution.

For our tests, we also varied the kernels in order to find the best data layout. These data formats come from the ideas presented within the BLAS-like Library Instantiation Software (BLIS) [12, 13]. We represented these data layouts pictorially within Figure 1.

2.1 Kernel Evaluation

In order to prove that the optimization is meaningful, we test on common problem sizes found in areas such as machine learning. Specifically, these problem sizes correspond to $M = N = K$ (square sizes). We vary the sizes of our matrices from 128 complex elements to 2048 in steps of 128.

For our naive implementation, we used 4 separate high performance real domain kernels (naive 4m) while our high performance implementation use 1 specialized complex kernel with a specific data layout. From Figure 2, we can see that the special 1m method outperforms the others for almost all the problems sizes; getting all the way up to 92% of peak. Thus, we conclude that optimizing out the naive 4m method is beneficial to the overall program performance.

3 EXPERIMENTS

3.1 Overview

We began investigating how to write a compiler pass within TensorFlow using their MLIR toolsets and dialects. However, as we progressed, it became apparent that writing such a pass would be infeasible for this project.

This is due to the fact that MLIR is still an emerging technology and the infrastructure and its documentation is in its infancy. To be specific, after putting great effort into trying different approaches, we realized that writing our pass would have required us to have a deep understanding of the Tensorflow system and their code organization, which was out of scope for this project.

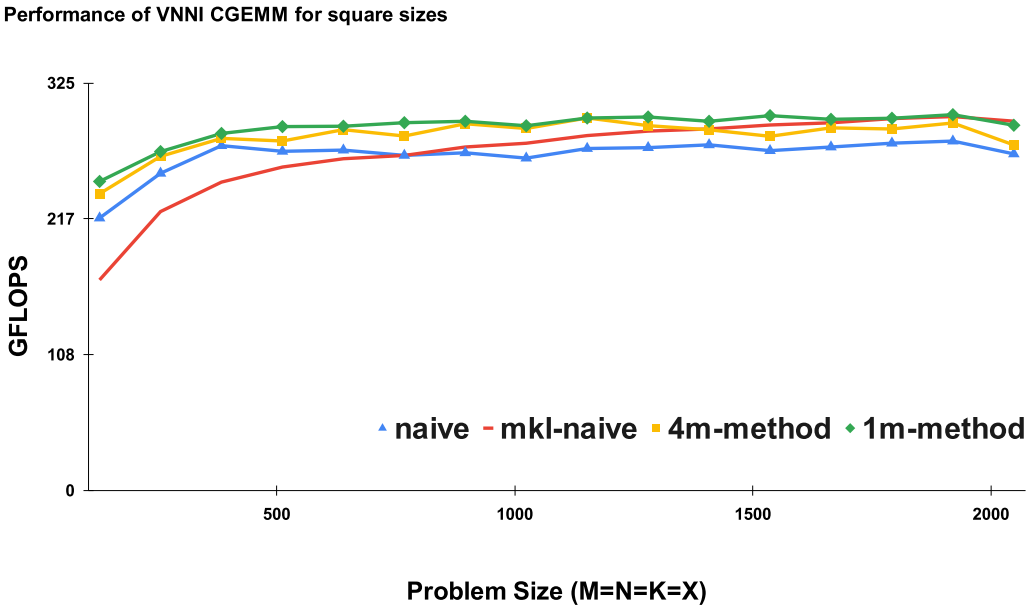

Fig. 2. This plot showcases standalone performance for various expert implementations of complex matrix multiplication. Through this plot, we can see that utilizing a specialized layout and kernel provides better performance over a naive emulated complex matrix multiplication. Furthermore, we also see that the 1m method provides the best overall performance for the VNNI set.

However, according to MLIR Tensorflow developers [5], there is an API rework for their compiler infrastructure in the works in order to allow developers to more easily create custom passes. Thus, in the future, we can implement the idea of the pass within Tensorflow's compiler infrastructure.

In order, to still showcase and evaluate our idea, we choose to implement our compiler pass for a toy dialect specific for complex arithmetic. The logic from this pass can then be easily converted into a pass for the Tensorflow dialect once the API rework is complete. Furthermore, since we know what the final code output should look like, we are able to evaluate our pass similarly to our standalone tests.

3.2 From Tensorflow to LLVM Attempt

We begin investigating how to write a compiler pass within TensorFlow using their MLIR toolsets and we are able to run the different toolkits to see the transformations between MLIR dialects. The pipeline that is expected to work is shown in the Figure 4. At the beginning, we tried to use the

```
tf.mlir.experimental.convert_graph_def(func_obj, pass_pipeline = 'convert-tf-control-flow-to-scf')
```

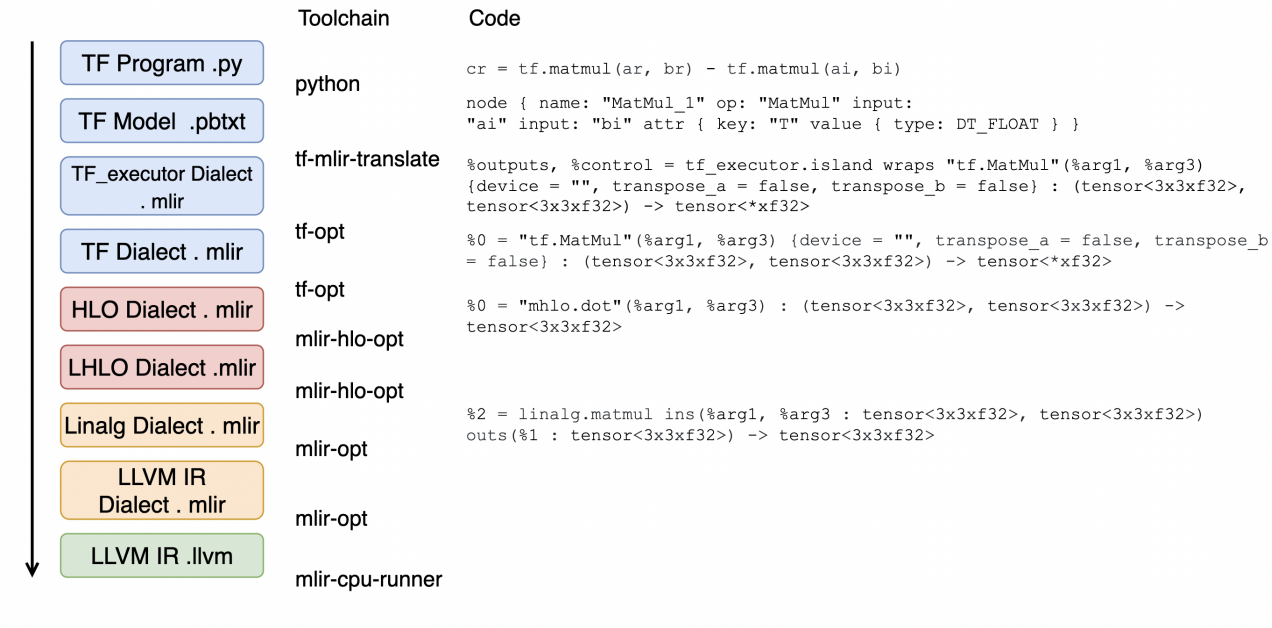

Fig. 3. Pipeline to generate and execute LLVM IR from Tensorflow code

to directly convert model to MLIR code in the python; however, we find that the MLIR code generated using this method forever assumes that "return" is empty, which causes the entire model will be dead code eliminated after lowering to the Linalg Dialect. Therefore we change to use

```
cgemm_naive_tf_func.get_concrete_function(ar, ai, br, bi, cr, ci).graph.as_graph_def()
```

to generate .pbtxt model first and then use tf-mlir-translate tool to convert it into mlir code. However, as we go forward, we find that we are unable to perform TensorFlow to Linalg lowering correctly when want to use mlir-hlo-opt tool to convert LHLO dialect into Linalg Dialect, because we cannot change "tensor" to "memref" because the HLO tool are updated to eliminate the "-lhlo-legalize-to-linalg" pass and the "–hlo-legalize-to-memref" pass weirdly doesn't take any effect in changing "tensor" to be "memref". Furthermore, we want to link generated MLIR program to externel C function wrappers to pass in the input data to the model; however, we have been encountering core dumps issues. And moreover, we look into Linalg Dialect and attempt to add a pass onto it, but due to a lot of complexity we met, we finally decided to try to build our own pass on the top of a simpler dialect.

3.3 Complex Pattern Matching Pass Based on Modified Toy Dialag

In order to showcase our idea of complex arithmetic recognition, we finally decide to implement our compiler pass based on Toy Dialect [14]. Toy Dialect accepts a simple tensor-based language called Toy that allows users to define functions, perform some math computation, and print results. However, Toy language has limited syntax support and only supports addition and element-wise multiplication arithmetic operations, we expand the Toy language

and modify Toy Dialect's parser to ensure the we can write the code with the complex matrix multiplication patterns in the Toy language. Based on the modified language, we implement a pass called MyComplexFusePass to merge any matrix multiplication calls that are recognized as in a complex pattern into a new complex matrix multiplication function call.

3.3.1 Toy Language and Dialect Modification. In order to allow Toy language supporting complex matrix multiplication patterns, we add two necessary arithmetic operations to the language and modify the Toy Dialect's parser and Toy AST-to-MLIR generator accordingly:

```
#1. subtraction
notion is "-"
example usage:
  var a = [[1, 2], [3, 4]]; # define a 2x2 tensor
  var b = [[1, 2], [3, 4]]; # define a 2x2 tensor
  var c = a - b; # perform element-wise subtraction
#2. matrix multiplication.
notion is "@"
example usage:
  var d = a @ b; # perform matrix multiplication
```

Now our entire list of arithmetic operations is: `"+"`, `"-"`, `"*"`, `"@"` , with the priority: `"+" = "-" < "*" = "@"`. Note that the `"@"` is the matrix multiplication of tensors and `"*"` is the element-wise multiplication of tensors.

Apart from the parser and Toy AST-to-MLIR generator, we also modify the operator definition table to register new Toy operators in the Toy Dialect. We register three new operators, in addition to subtraction and matrix multiplication that are also add to the Toy language, we also define an operator called complex matrix multiplication to be instantiated when the complex matrix multiplication patterns are identified.

```
def SubOp : Toy_Op<"sub",
    [NoSideEffect, DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
  ...
  let arguments = (ins F64Tensor:$lhs, F64Tensor:$rhs);
  let results = (outs F64Tensor);
  let builders = [ OpBuilder<(ins "Value":$lhs, "Value":$rhs)>];
}
def MatmulOp : Toy_Op<"matmul",
    [NoSideEffect, DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]>{
  ... // simliar to SubOp
}
def ComplexMulKernelOp : Toy_Op<"complex_mul_kernel",
    [NoSideEffect,DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
  ...
```

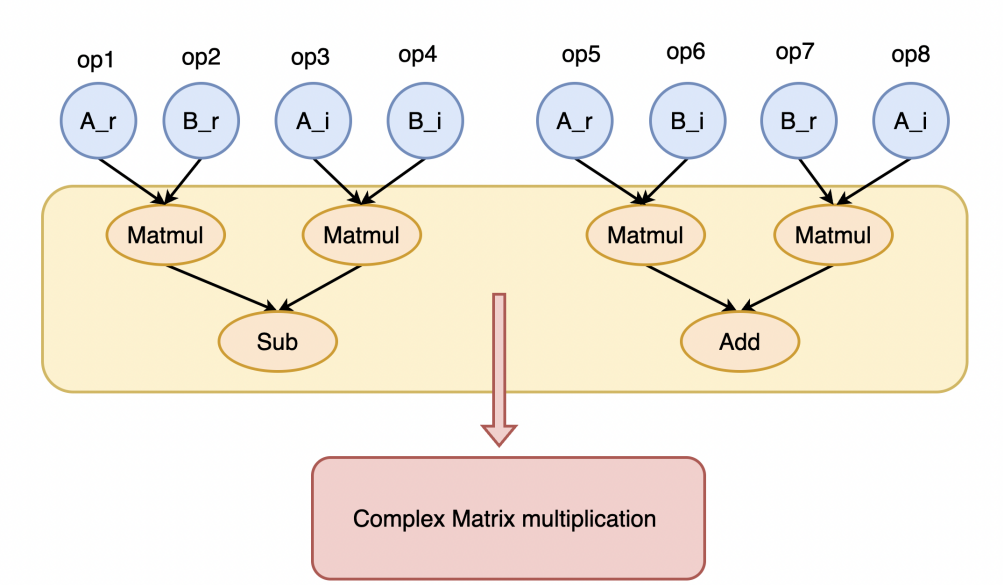

Fig. 4. DAG for the targeted patterns

```
15   let arguments = (ins F64Tensor:$lhs, F64Tensor:$rhs, F64Tensor:$lhs2, F64Tensor:$rhs2 );
16   let results = (outs F64Tensor);
17   let builders = [
18     OpBuilder<(ins "Value":$lhs,"Value":$rhs,"Value":$lhs2,"Value":$rhs2 )>];
19 }
```

The new Toy operators are indicated in the above code. Operator "Toy::SubOp" and "Toy::MatmulOp" accept two input tensors as arguments and operator "Toy::ComplexMulKernelOp" accepts four input tensors as arguments. The toy dialect provides the ToyToLLVMLoweringPass pass for lowering down Toy operators to LLVM dialect and for our project we only lowering down the "Toy::SubOp" to arith Dialect.

3.3.2 Complex Pattern Matching Pass. We implement a pass that recognizes complex patterns from normal matrix multiplication operators and replaces the matched operators into a operator designed specifically for managing complex matrix multiplications. As shown in the figure 4, the patterns we want to match have 1) one subtraction operation with both operands resulting from a matrix multiplication operations, 2) one addition operation with both operands resulting from a matrix multiplication operations, 3) 8 inputs are from two complex numbers A and B. In addition to the above constraints, we also make an assumption that the real and imaginary parts of two complex numbers are distinct, which is reasonable because having the same input is a huge condition in the high-performance computing world and should be handled by other special passes. Based on the this assumption we composition the third constraint to be three conditions of relationship between 8 input operands as position shown in the figure 4: 1) op1, op2, op3, op4 are distinct; op5, op6, op7, op8 are distinct; 2) [op1, op2,

Fig. 5. Toy Dialect Transformation by Pass MyComplexFussPass

op3, op4] and [op5, op6, op7, op8] has the same value range 3) assume op1 matches opX in one of op5, op6, op7, op8, op2 will not have the same value with the another operands linked with the same matmul operation of opX. Based on the pattern matching logic we described above, we demonstrate our entire algorithm below:

```
# we maintain three vectors, mul_add_exprs and mul_sub_exprs are for recording the operations
    linked in the form of  () * () + () * () or () * () - () * () and erase_ops_vector is to
    save the operands for the matched patterns.
llvm::SmallPtrSet<mlir::Operation *, 16> mul_add_exprs;
llvm::SmallPtrSet<mlir::Operation *, 16> mul_sub_exprs;
std::vector<std::vector<mlir::Operation*>> erase_ops_vector;
for each operation op:
    check if the current op is linked as the form () * () + () * () or  () * () - () * ()
    if yes
        check mul_add_exprs & mul_sub_exprs if there any pairs match a complex pattern
            if yes
                add all of the linked operands to the erase_ops_vector
            if no
                add the current operation into mul_add_exprs or mul_sub_exprs
for each elem in erase_ops_vector
     create and insert a ComplexMulKernel operation and erase the original operations
```

Our pass can successfully detect a various of complex patterns. As shown in the figure 5, our pass MyComplex-FussPass correctly recognizes the complex pattern existing in the Toy Dialect mlir code and transform one "sub",

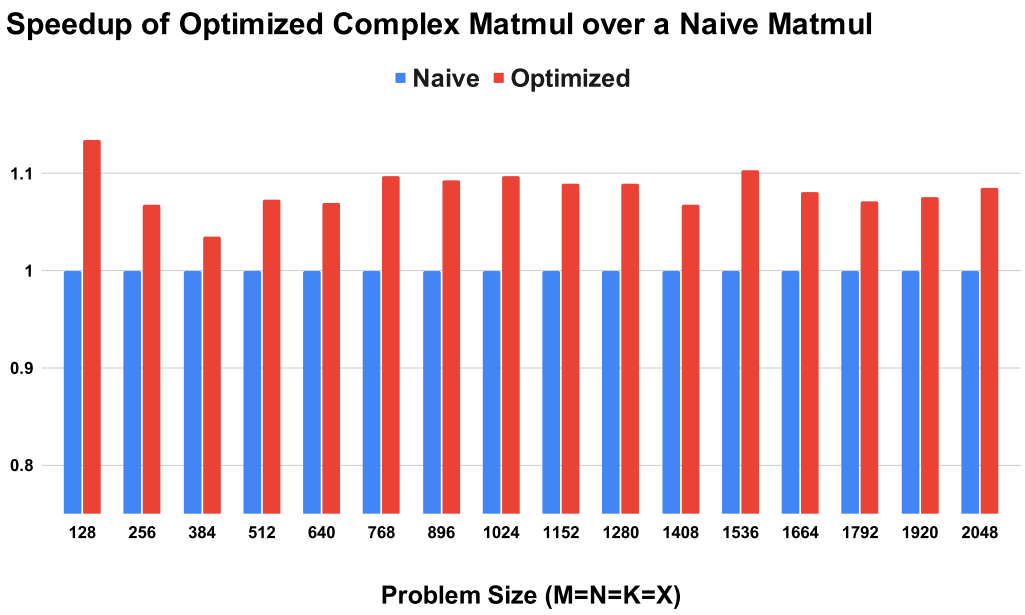

Fig. 6. This plot shows us the predicted performance improvement through our custom MLIR pass. With this, we can see that we get an average speedup of 8% for various problem sizes. This test also assumes the best case scenario in which the data layout conversion has already been applied.

one "add", and four "matmul" operations into one single "complexmulkernel" operation. We also create five more tests to validate the correctness of our pass and the results can be viewed in the "Toy/Ch7/test" directory.

4 EVALUATION

Due to the current status of the MLIR infrastructure, we were unable to run our generated code for our compiler pass. However, to prove that our compiler pass still performs the logic that we expect, we show the generated code from our toy dialect compiler pass. For this, we are able to see that we can replace naive complex arithmetic with an optimized kernel call.

Since we know the performance of the optimized kernel call through our standalone tests, we are able to predict what speedup our generated code would have over the naive version. Thus, we take the naive 4m implementation (with the assumption that the data is in de-interleaved) and compare against the best performing complex kernel (with the specialized data layout). We showcase the speedup the optimized complex kernel call has over the naive implementation in Figure 6. From the plot, we can predict that our optimized code will have an 8% speedup over a naive version with the assumption that the data layout has already been converted.

5 SURPRISES AND LESSONS LEARNED

5.1 Frustrations with MLIR and Tensorflow

From our investigation, we were surprised to see how young MLIR still is. A lot of the information about MLIR is only touching the surface and still focused on expanding its stack. For example, when we were attempting to use Tensorflow's MLIR infrastructure (mind you this is where MLIR spawned), the sheer complexity of getting a simple pass like the one we choose was shocking.

We spent most of our time working by digging through Tensorflow's and MLIR's documentation. When we first attempted to generate MLIR code using our tensorflow programs, we needed to find their experimental branch of code which allows for managing MLIR objects. Moreover, this experimental branch was still limited in its feature. Specifically, it is limited in its ability to create and run custom passes [5]. This is in stark contrast to LLVM's pass documentation, which documents exactly how a pass should be implemented and how to execute it.

In short, the Tensorflow compiler stack and MLIR documentation are still in an obscure manner and can only target very specific architectures within their environments. Making it difficult for developers like us to pick up and use.

5.2 MLIR's Potential

Even though MLIR is still young and has quirks that need to be worked out, it is quite obvious to see when working with MLIR that it is the future of compiler technology. The reason for this is that MLIR allows for easier conversion between high level and low level dialects. Making it easier to write optimizations at different levels of abstraction. With this, it becomes straightforward to write code to be optimized for devices that have heterogeneous compute systems. In general, MLIR can be seen as the steel to the bridge that will make it easier for users to write high performance code that utilizes various compute architectures.

6 CONCLUSION AND FUTURE WORK

In this project, we have laid out the results of our investigation into writing a compiler pass to replace emulated complex arithmetic with a high performance kernel, converting data layouts if needed. Our compiler pass targeted the MLIR compiler stack since it best suited our needs of representing high level tensor operations. Initially, we focused on generating MLIR from a Tensorflow program, but this became infeasible due to a limitation in the current Tensorflow API. Thus, we shifted our focus to writing a pass for toy dialect specifically designed for complex matrix multiplication. With this dialect, we showed that our pass increased the throughput of the code by an average factor of about 8%. Looking forward, we would like to expand the pass to generate code for other compute devices such as GPUs, FPGAs, and specialized accelerators; add logic to auto select devices which provide the fastest end to end operation latency; auto generate conversion routines from a specified data layout input and output; auto generate a performant complex matrix multiplication kernel to replace the emulated one.

7 ACKNOWLEDGMENTS

Thank you to Professor Todd Mowry and Chrisma Pakha in helping us for this project. Your lessons and feedback were helpful in understanding how to proceed for this project, as well as, future projects related to optimizing compilers and LLVM in general. Moreover, this work is important in our personal research, and we are happy to be had been able to work on it for your class.

REFERENCES

[1] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[2] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[3] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[4] Operation pass. https://mlir.llvm.org/docs/PassManagement/#operation-pass, 2022.

[5] Custom mlir tensorflow pass. https://github.com/tensorflow/tensorflow/issues/55750, 2022.

[6] Circt: Circuit ir compilers and tools. https://github.com/llvm/circt, 2022.

[7] Flang. https://github.com/llvm/llvm-project/tree/main/flang, 2022.

[8] Iree: Intermediate representation execution environment. https://github.com/google/iree, 2022.

[9] Mlir-based aiengine toolchain. https://github.com/Xilinx/mlir-aie, 2022.

[10] The torch-mlir project. https://github.com/llvm/torch-mlir, 2022.

[11] Uday Bondhugula. High performance code generation in mlir: An early case study with gemm. *arXiv preprint arXiv:2003.00532*, 2020.

[12] Field G. Van Zee. Implementing high-performance complex matrix multiplication via the 1m method. *SIAM Journal on Scientific Computing*, 42(5):C221–C244, September.

[13] Field G. Van Zee and Tyler Smith. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Transactions on Mathematical Software*, 44(1):7:1–7:36, July 2017.

[14] Mlir toy tutorial. https://mlir.llvm.org/docs/Tutorials/Toy/, 2022.