Recognizing Complex Arithmetic with MLIR

Nicholai Tukanov
Chengyue Wang

```
def cgemm(Ar, Ai, Br, Bi, Cr, Ci):
    Cr += tf.matmul(Ar, Br) - \
        tf.matmul(Ai, Bi)
    Cr += tf.matmul(Ar, Bi) + \
        tf.matmul(Ai, Br)
```

Electrical & Computer ENGINEERING

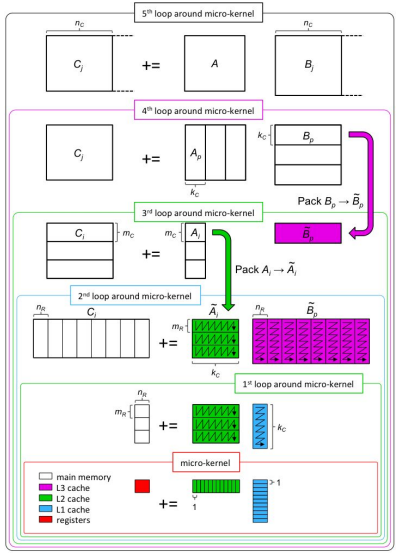
Introduction

Matrix multiplication is a backbone for many machine learning and scientific computing 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.

```
def real_gemm(m, n, k, A, B, C):
    for i in range(m):
        for j in range(n):
            for p in range(k):
                C[i][j] += A[i][p] * B[p][j]


def complex_gemm(m, n, k, A, B, C):
    for i in range(m):
        for j in range(n):
            for p in range(k):
                C[i][j].real() += A[i][p].real() * B[p][j].real() - \
                                  A[i][p].imag() * B[p][j].imag()

                C[i][j].imag() += A[i][p].real() * B[p][j].imag() - \
                                  A[i][p].imag() * B[p][j].real()
```




```
def cgemm(Ar, Ai, Br, Bi, Cr, Ci):
    Cr += tf.matmul(Ar, Br) - \
          tf.matmul(Ai, Bi)
    Cr += tf.matmul(Ar, Bi) + \
          tf.matmul(Ai, Br)
```

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.

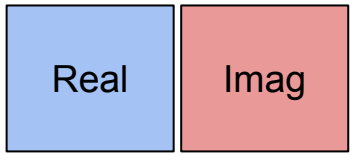
For our project, we investigate how to write a generic compiler pass using MLIR that replaces emulated complex arithmetic with direct complex arithmetic and the appropriate layout changes for the targeted hardware.

Is it worth it?

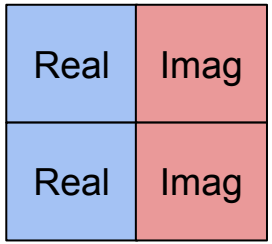
Before diving into writing the pass, we ensured ourselves that this optimization will result in better performance (specifically better operation throughput). Moreover, we experimented with different legal data formats to see which one results in the best performance.

We conducted our tests using Intel's rocket lake microarchitecture. On which, we targeted the Vector Neural Network Instruction (VNNI) set since they result in the highest peak theoretical throughput on the chip (~325 GFLOPS/core).

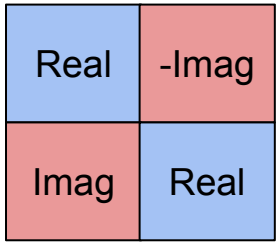
For our tests, we used 3 separate data layouts: de-interleaved, tile de-interleaved, and the special 1m format.



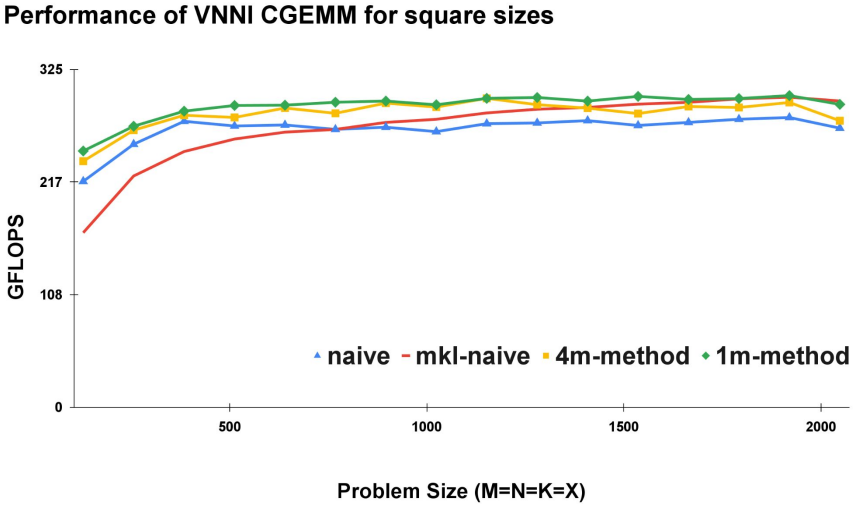
De-interleaved



Tiled de-interleaved
(optimized 4m method)



Special 1m format



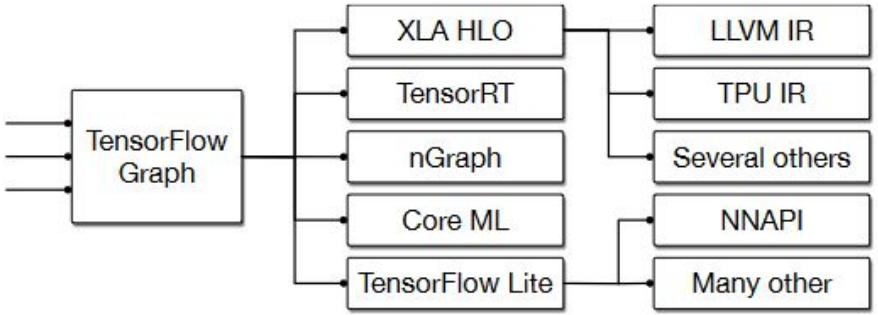
Performance of various high performance VNNI CGEMM kernels. The 1m method achieves the highest performance throughout the whole benchmark (up to 92% of peak).

The naive test represents the emulated complex arithmetic method. It also assumes that the data is already in de-interleaved format, which is the best possible case for it.

Electrical & Computer ENGINEERING

What is MLIR?

Simply put, MLIR is a compiler infrastructure/stack that allows for building reusable and extensible compilers for various dialects. The overall goal of MLIR is to make it easier to generate code for domain specific languages, target compilation for heterogeneous architectures (such as a system on chip), and link existing compilers together (such as LLVM or TVM).



Tensorflow model compilation and execution pipeline, which shows many distinct compilers.

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops and can have multiple blocks.          Region
  ^block(%argument: !d.type):                                    Block
    // Ops have function types (expressing mapping).
    %value = "nested.operation"() ({
      // Ops can contain nested regions.                          Region
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:                                                  Block
    "d.terminator"() [^block(%argument : !d.type)] : () -> ()
  })
// Ops can have a list of attributes.
{attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

An MLIR operation. This considered the main entity in MLIR

Why MLIR?

Given that matrix multiplication is a tensor operation, we initially choose to use Tensorflow combined with MLIR in order to write a compiler pass to optimize for complex arithmetic.

The reason for this is that MLIR can easily express tensor operations in a concise and abstract manner.

Moreover, with MLIR, we have the ability to easily convert optimized code between various dialects and computing architectures (i.e., write code once in a higher level language like Tensorflow and then compile it to get optimized binaries for either CPUs, GPUs, TPUs, FPGAs, and many more).

Lastly, Tensorflow is a popular machine learning language. Making it a great starting point for writing tensor operations that can be converted into an MLIR dialect and then mapped onto various architectures.

```
module attributes {tf.versions = {bad_consumers = [], min_consumer = 0 : i32, producer = 987 : i32}}  {
  func @__inference_cgemm_explict_19(%arg0: tensor<3x3xf32> {tf._user_specified_name = "Ar"},
    %arg1: tensor<3x3xf32> {tf._user_specified_name = "Ai"},
    %arg2: tensor<3x3xf32> {tf._user_specified_name = "Br"},
    %arg3: tensor<3x3xf32> {tf._user_specified_name = "Bi"},
    %arg4: tensor<3x3xf32> {tf._user_specified_name = "Cr"},
    %arg5: tensor<3x3xf32> {tf._user_specified_name = "Ci"}) -> tensor<3x3xcomplex<f32>>
    attributes {tf.entry_function = {control_outputs = "", inputs = "ar,ai,br,bi,cr,ci", outputs = "identity_RetVal"}} {
    %0 = "tf.MatMul"(%arg1, %arg3) {device = "", transpose_a = false, transpose_b = false} : (tensor<3x3xf32>, tensor<3x3xf32>) -> tensor<3x3xf32>
    %1 = "tf.MatMul"(%arg0, %arg3) {device = "", transpose_a = false, transpose_b = false} : (tensor<3x3xf32>, tensor<3x3xf32>) -> tensor<3x3xf32>
    %2 = "tf.MatMul"(%arg0, %arg2) {device = "", transpose_a = false, transpose_b = false} : (tensor<3x3xf32>, tensor<3x3xf32>) -> tensor<3x3xf32>
    %3 = "tf.Sub"(%2, %0) {device = ""} : (tensor<3x3xf32>, tensor<3x3xf32>) -> tensor<3x3xf32>
    %4 = "tf.MatMul"(%arg1, %arg2) {device = "", transpose_a = false, transpose_b = false} : (tensor<3x3xf32>, tensor<3x3xf32>) -> tensor<3x3xf32>
    %5 = "tf.AddV2"(%1, %4) {device = ""} : (tensor<3x3xf32>, tensor<3x3xf32>) -> tensor<3x3xf32>
    %6 = "tf.AddV2"(%5, %arg5) {device = ""} : (tensor<3x3xf32>, tensor<3x3xf32>) -> tensor<3x3xf32>
    %7 = "tf.AddV2"(%3, %arg4) {device = ""} : (tensor<3x3xf32>, tensor<3x3xf32>) -> tensor<3x3xf32>
    %8 = "tf.Complex"(%7, %6) {device = ""} : (tensor<3x3xf32>, tensor<3x3xf32>) -> tensor<3x3xcomplex<f32>>
    %9 = "tf.Identity"(%8) {device = ""} : (tensor<3x3xcomplex<f32>>) -> tensor<3x3xcomplex<f32>>
    return %9 : tensor<3x3xcomplex<f32>>
  }
}
```

An example MLIR in the Tensorflow dialect. This code was generated from the tf.function() below.

```
@tf.function
def cgemm_explict(A_r, A_i, B_r, B_i, C_r, C_i):
    return tf.complex(A_r @ B_r - A_i @ B_i + C_r, A_r @ B_i + A_i @ B_r + C_i)
```

Tensorflow function that computes emulated complex arithmetic.

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops and can have multiple blocks.        Region
  ^block(%argument: !d.type):                                   Block
    // Ops have function types (expressing mapping).
    %value = "nested.operation"() ({
      // Ops can contain nested regions.                        Region
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:                                                 Block
    "d.terminator"() [^block(%argument : !d.type)] : () -> ()
}) 
// Ops can have a list of attributes.
{attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

An MLIR operation. This considered the main entity in MLIR

Electrical & Computer ENGINEERING

Pass Infrastructure

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, 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.

However, according to MLIR Tensorflow developers, 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.

Toy Dialect

Toy language - a tensor-based language

Arithmetic operations supported:

"+", "-", "*"(elemwise mul), "@" (matrix mul)

Toy Language

```
def main() {
  var a = [[1, 2], [4, 6]];
  var b = [[2, 2], [4, 6]];
  var c = [[1, 3], [4, 6]];
  var d = [[1, 4], [4, 6]];
  var e = a@b - c@d;
  var f = a@c + b@d;
  var g = e*f;
  print(g);
}
```

We register three new operators in Toy Dialect

```
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>]> {
  ...
```

CmplexMulKernelOp doesn't exit in Toy language but will be instantiated when the complex matrix multiplication patterns are identified

--MyComplexFussPass

With our toy dialect, we created a matrix multiplication operation that takes in two memrefs. Using this operation, we wrote an emulated complex matrix multiplication with our toy dialect. We then pass this program through an MLIR OperationPass which replaces the emulated complex matrix multiplication with a call to a data format conversion routine (if needed) and high performance kernel.

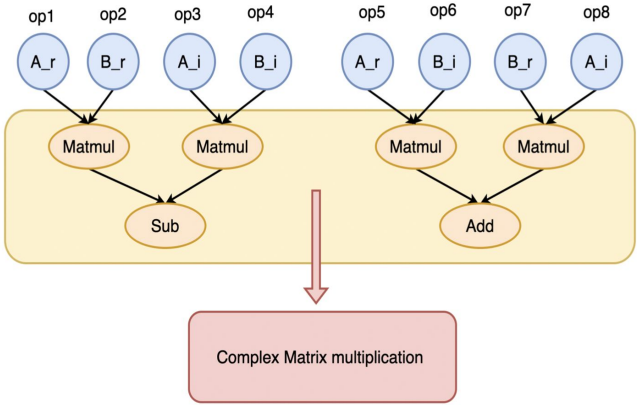


Fig. 4. DAG for the targeted patterns

Toy Dialect Before Our Pass

```
module {
  toy.func @main() {
    %0 = toy.constant dense<[[1.000000e+00, 4.000000e+00
    %1 = toy.constant dense<[[1.000000e+00, 3.000000e+00
    %2 = toy.constant dense<[[2.000000e+00, 2.000000e+00
    %3 = toy.constant dense<[[1.000000e+00, 2.000000e+00
    %4 = toy.matmul %3, %2 : tensor<2x2xf64>
    %5 = toy.matmul %1, %0 : tensor<2x2xf64>
    %6 = toy.sub %4, %5 : tensor<2x2xf64>
    %7 = toy.matmul %3, %1 : tensor<2x2xf64>
    %8 = toy.matmul %2, %0 : tensor<2x2xf64>
    %9 = toy.add %7, %8 : tensor<2x2xf64>
    %10 = toy.mul %6, %9 : tensor<2x2xf64>
    toy.print %10 : tensor<2x2xf64>
    toy.return
  }
}
```

Toy Dialect After Our Pass

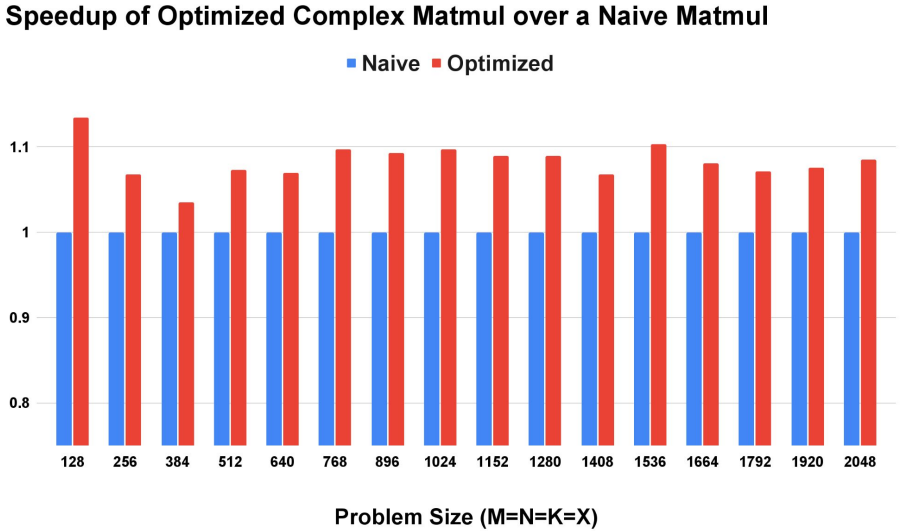
```
module {
  toy.func @main() {
    %0 = toy.constant dense<[[1.000000e+00, 4.000000e+00], [4.000000e+00, 6.000000e+00]]> : tensor<2x2xf64>
    %1 = toy.constant dense<[[1.000000e+00, 3.000000e+00], [4.000000e+00, 6.000000e+00]]> : tensor<2x2xf64>
    %2 = toy.constant dense<[[2.000000e+00, 2.000000e+00], [4.000000e+00, 6.000000e+00]]> : tensor<2x2xf64>
    %3 = toy.constant dense<[[1.000000e+00, 2.000000e+00], [4.000000e+00, 6.000000e+00]]> : tensor<2x2xf64>
    %4 = toy.complex_mul_kernel %3, %2, %3, %1 : (tensor<2x2xf64>, tensor<2x2xf64>, tensor<2x2xf64>, tensor<2x2xf64>) -> tensor<*xf64>
    %5 = toy.mul %4, %4 : (tensor<*xf64>, tensor<*xf64>) -> tensor<2x2xf64>
    toy.print %5 : tensor<2x2xf64>
    toy.return
  }
}
```

Pass 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. In general, we predict that our optimized code has an 8% speedup over a naive version with the assumption that the data layout has already been converted.



Surprises and Lessons Learned

- The sheer complexity behind writing an MLIR pass for the Tensorflow compiler infrastructure. Since our pass idea is fairly simple (basically it is pattern matching), we were surprised when we struggled to get information on how to write our own custom pass for Tensorflow. Furthermore, we also struggled to even find documentation that would generate MLIR for Tensorflow programs. In short, unlike LLVM's pass documentation, the Tensorflow and MLIR documentation behind different compiler passes for the TF dialect and how they are implemented are kept in an obscure manner, making it difficult for developers like us to pick up and start using.

- MLIR is the future of compiler technology. The MLIR compiler stack allows for easier conversion between dialects and, more importantly, easier to perform optimizations and generate code for multidimensional objects such as tensors. As heterogeneous computing becomes more ubiquitous, MLIR will be the steel to building the bridge to make it easier to write performant code that utilizes all the various compute devices.

- Something is always out of date...

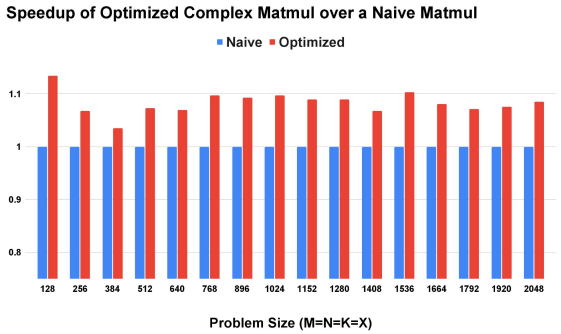
Conclusion and Future Work

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 optimized MLIR from a Tensorflow program, but this became infeasible due to limitations in the current Tensorflow API and MLIR infrastructure. 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 can correctly generate optimized code. From this code, we can predict that we increase the throughput of the code by a factor 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 (rather than linking an expert backend)

TensorFlow

```
def cgemm(Ar, Ai, Br, Bi, Cr, Ci):
    Cr += tf.matmul(Ar, Br) - \
        tf.matmul(Ai, Bi)
    Cr += tf.matmul(Ar, Bi) + \
        tf.matmul(Ai, Br)
```



Electrical & Computer ENGINEERING