Polyhedral Compilation Opportunities in MLIR

Uday Bondhugula

Indian Institute of Science

udayb@iisc.ac.in
Introduction: Role of Compiler Infrastructure

MLIR
  Representation
  Polyhedral Framework: A Quick Intro
  Polyhedral Notions in MLIR
    Data types

High-performance code generation in MLIR

Opportunities and Conclusions
COMPILERS - THE EARLY DAYS

- Pascal
- ALGOL
- ADA
- PL/8
- C
- IBM 801
- S/370
- Motorola 68000
- Power
- PowerPC

languages, targets ⇒ ∗

Not scalable!

Uday Bondhugula, IISc
M languages, N targets \Rightarrow M \times N compilers! Not scalable!
With an common IR, we have $M + N + 1$ compilers!
How do modern compilers look?
LLVM: modular, reusable, open-source: $M + 1 + 1 + N/k$
MODERN COMPILERS - LLVM IR BASED

- C
- C++ → Clang AST
- Objective-C
- Rust → HIR/MIR
- Swift → SIL
- Julia → Julia AST
- TensorFlow Graph → XLA HLO
- LabVIEW

Flow: C++ → Clang AST → HIR/MIR → LLVM IR → LLVM Machine IR

- x86
- x86-64
- Power
- ARM
- PTX
- ... (target desc.)

- But too level for ML/AI programming models/hardware

Uday Bondhugula, IISc
Fast forward to ML/AI compute era
ML/AI Compilation Problem

Explosion of ML/AI programming models, languages, frameworks

Compiler Infrastructure?

Explosion of AI chips and accelerators
A proliferation of IRs

- TensorFlow graphs (Google)
- XLA IR / HLO (Google)
- Onnx (Facebook, Microsoft)
- Glow (Facebook)
- Halide IR, TVM (universities)
- Stripe (PlaidML, now Intel)
- nGraph (Intel)
- ...

As a result: a proliferation IRs
Fast Forward to ML/AI

Explosion of ML/AI programming models, languages, frameworks

Explosion of AI chips and accelerators

Uday Bondhugula, IISc
Fast Forward to ML/AI

Explosion of ML/AI programming models, languages, frameworks

TF  ○  K  ...  ?

Explosion of AI chips and accelerators

Uday Bondhugula, IISc
In Comes MLIR

- Open-sourced by Google in Apr 2019
- Designed and built as an IR from day 0!
MLIR: Multi-level Intermediate Representation

1. Ops (general purpose to domain specific) on tensor types / memref types

%patches = "tf.reshape"(%patches, %minus_one, %minor_dim_size)
: (tensor<? x ? x ? x ? x f32>, index, index) → tensor<? x ? x f32>

%mat_out = "tf.matmul"(%patches_flat, %patches_flat)[transpose_a : true]
: (tensor<? x ? x f32>, tensor<? x ? x f32>) → tensor<? x ? x f32>

%vec_out = "tf.reduce_sum"(%patches_flat) [axis: 0]
: (tensor<? x ? x f32>) → tensor<? x f32>

2. Loop-level / mid-level form

for (i = 0; i < N; i++)
for (k = 0; k < N; k++)
for (i = 0; i < N; i++)
S1
S2
for (j = 0; j < N; j++)
S1
for (j = 0; j < N; j++)
S2
0 <= i <= N−1
0 <= j <= N−1
0 <= k <= N−1

affine.for %i = 0 to 8 step 4 {
  affine.for %j = 0 to 8 step 4 {
    affine.for %k = 0 to 8 step 4 {
      affine.for %ii = #map0(%i) to #map1(%i) {
        affine.for %jj = #map0(%j) to #map1(%j) {
          affine.for %kk = #map0(%k) to #map1(%k) {

            %5 = affine.load [%arg0[%ii, %kk]] : memref<8x8xvector<64xf32>>
            %6 = affine.load [%arg1[%kk, %jj]] : memref<8x8xvector<64xf32>>
            %7 = affine.load [%arg2[%ii, %jj]] : memref<8x8xvector<64xf32>>
            %8 = mulf %5, %6 : vector<64xf32>
            %9 = addf %7, %8 : vector<64xf32>

            affine.store %9, [%arg2[%ii, %jj]] : memref<8x8xvector<64xf32>>
          }
        }
      }
    }
  }
}

3. Low-level form: closer to hardware

%v1 = load %a[%i2, %i3] : memref<256x64xvector<16xf32>>
%v2 = load %b[%i2, %i3] : memref<256x64xvector<16xf32>>
%v3 = addf %v1, %v2 : vector<16xf32>
store %v3, [%d[%i2, %i3]] : memref<256x64xvector<16xf32>>
MLIR Design Principles / Features

1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels
4. First class abstractions for multi-dimensional arrays (tensors), loop nests, and more
5. Very flexible, extensible
MLIR Design Principles / Features

1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels
4. First class abstractions for multi-dimensional arrays (tensors), loop nests, and more
5. Very flexible, extensible
MLIR Design Principles / Features

1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels
4. First class abstractions for multi-dimensional arrays (tensors), loop nests, and more
5. Very flexible, extensible
1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels
4. First class abstractions for multi-dimensional arrays (tensors), loop nests, and more
5. Very flexible, extensible
Introduction: Role of Compiler Infrastructure

MLIR

Representation

Polyhedral Framework: A Quick Intro
Polyhedral Notions in MLIR
Data types

High-performance code generation in MLIR

Opportunities and Conclusions
MLIR: Multi-level Intermediate Representation

1. Ops (general purpose to domain specific) on tensor types / memref types

```mlir
%patches = "tf.reshape"(%patches, %minus_one, %minor_dim_size) : (tensor<? x ? x ? x ? x f32>, index, index) -> tensor<? x ? x f32>
%mat_out = "tf.matmul"(%patches_flat, %patches_flat){transpose_a : true} : (tensor<? x ? x f32>, memref<? x ? x f32>) -> tensor<? x ? x f32>
%vec_out = "tf.reduce_sum"(%patches_flat) {axis: 0} : (tensor<? x ? x f32>) -> tensor<? x f32>
```

2. Loop-level / mid-level form

```mlir
affine.for %i = 0 to 8 step 4 {
    affine.for %j = 0 to 8 step 4 {
        affine.for %k = 0 to 8 step 4 {
            affine.for %ii = #map0(%i) to #map1(%i) {
                affine.for %jj = #map0(%j) to #map1(%j) {
                    affine.for %kk = #map0(%k) to #map1(%k) {
                        %5 = load %arg0[%ii, %kk] : memref<8x8xvector<64xf32>>
                        %6 = load %arg1[%kk, %jj] : memref<8x8xvector<64xf32>>
                        %7 = load %arg2[%ii, %jj] : memref<8x8xvector<64xf32>>
                        %8 = mulf %5, %6 : vector<64xf32>
                        %9 = addf %7, %8 : vector<64xf32>
                        store %9, %arg2[%ii, %jj] : memref<8x8xvector<64xf32>>
                    }
                }
            }
        }
    }
}
```

3. Low-level form: closer to hardware

```mlir
%v1 = load %a[%i2, %i3] : memref<256x64xvector<16xf32>>
%v2 = load %b[%i2, %i3] : memref<256x64xvector<16xf32>>
%v3 = addf %v1, %v2 : vector<16xf32>
store %v3, %d[%i2, %i3] : memref<256x64xvector<16xf32>>
```
MLIR - Basic Concepts

- SSA, typed
- Module/Function/Block/Operation structure
- Operations can hold a “region” (a list of blocks)

```mlir
func @testFunction(%arg0: i32) {
  %x = call @thingToCall(%arg0) : (i32) -> i32
  br ^bb1
  ^bb1:
    %y = addi %x, %x : i32
    return %y : i32
}
```
SSA REPRESENTATION

- Functional SSA representation
- No $\phi$ nodes
- Instead, basic blocks take arguments

```plaintext
func @condbr_simple() -> (i32) {
  %cond = "foo"() : () -> i1
  %a = "bar"() : () -> i32
  %b = "bar"() : () -> i64
  cond_br %cond, ^bb1(%a : i32), ^bb2(%b : i64)
}

^bb1(%x : i32):
  %w = "foo_bar"(%x) : (i32) -> i64
  br ^bb2(%w : i64)

^bb2(%y : i64):
  %z = "abc"(%y) : (i64) -> i32
  return %z : i32
}
```
MLIR OPERATIONS

- Operations always have a name and source location info
- Operations may have:
  - Arbitrary number of SSA operands and results
  - Attributes: guaranteed constant values
  - Regions

```mlir
%2 = dim %1, 1 : tensor<1024x? x f32>
// Dimension to extract is guaranteed integer constant, an attribute
%x = alloc() : memref<1024x64 x f32>
%y = load %x[%i, %j] : memref<1024x64 x f32>
```
Operations in MLIR can have nested regions

```mlir
func @loop_nest_unroll(%arg0: index) {
  affine.for %arg1 = 0 to 100 step 2 {
    affine.for %arg2 = 0 to #map1(%arg0) {
      %0 = "foo"() : () -> i32
    }
  }
  return
}
```

Use cases: besides affine for/if, shielding inner control flow, closures/lambdas, parallelism abstractions like OpenMP, etc.
**Dialects in MLIR**

- **Dialect**: A collection of operations, types, and attributes suitable for a specific task
- Typically corresponds to a programming model’s entry point into MLIR, a backend, or a well-defined abstraction
- Example dialects: TensorFlow dialect, NGraph dialect, Affine dialect, Linalg dialect, NVIDIA GPU dialect, LLVM dialect
- You can have a mix of dialects
OUTLINE

Introduction: Role of Compiler Infrastructure

MLIR
  Representation
  Polyhedral Framework: A Quick Intro
  Polyhedral Notions in MLIR
    Data types

High-performance code generation in MLIR

Opportunities and Conclusions
for (t = 0; t < T; t++)
for (i = 1; i < N+1; i++)
for (j = 1; j < N+1; j++)
    \( A[(t+1)\%2][i][j] = f((A[t\%2][i+1][j], A[t\%2][i][j], A[t\%2][i-1][j],
    A[t\%2][i][j+1], A[t\%2][i][j-1]); \)

1. Domains
   - Every statement has a domain or an index set – instances that have to be executed
   - Each instance is a vector (of loop index values from outermost to innermost)
     \( D_S = \{[t, i, j] \mid 0 \leq t \leq T - 1, \ 1 \leq i, j \leq N \} \)

2. Dependences
   - A dependence is a relation between domain / index set instances that are in conflict (more on next slide)

3. Schedules
   - are functions specifying the order in which the domain instances should be executed
   - Eg: \( T((t, i, j)) = (t, t + i, j) \)
for (i = 1; i <= N - 1; i++)
    for (j = 1; j <= N - 1; j++)
        A[i][j] = f(A[i-1][j], A[i][j-1]);

**Domain:** \{[i, j] \mid 1 \leq i, j \leq N - 1\}

Original space \((i, j)\)
for (i = 1; i <= N - 1; i++)
for (j = 1; j <= N - 1; j++)
A[i][j] = f(A[i-1][j], A[i][j-1]);

Dependences:

1. \{[i, j] \rightarrow [i + 1, j] \mid 1 \leq i \leq N - 2, 0 \leq j \leq N - 1\} — (1,0)
2. \{[i, j] \rightarrow [i, j + 1] \mid 1 \leq i \leq N - 1, 0 \leq j \leq N - 2\} — (0,1)
for (i = 1; i <= N - 1; i++)
for (j = 1; j <= N - 1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);

Dependences:

1. \{[i, j] \rightarrow [i + 1, j] \mid 1 \leq i \leq N - 2, 0 \leq j \leq N - 1\} — \((1,0)\)
2. \{[i, j] \rightarrow [i, j + 1] \mid 1 \leq i \leq N - 1, 0 \leq j \leq N - 2\} — \((0,1)\)
**Domains, Dependences, and Schedules**

\[ \text{for } (i = 1; i \leq N - 1; i++) \]
\[ \text{for } (j = 1; j \leq N - 1; j++) \]
\[ A[i][j] = f(A[i-1][j], A[i][j-1]); \]

\[ \text{for } (t1=2;t1\leq2+N-2;t1++) \{ \]
\[ \text{#pragma omp parallel for private(lbv,ubv) \]
\[ \text{for } (t2 = \max(1,t1-N+1); t2 \leq \min(N-1,t1-1); t2++) \{ \]
\[ \text{a[(t1-t2)][t2] = a[(t1-t2) - 1][t2] + a[(t1-t2)][t2 - 1]; \]
\[ \} \]

**Schedule:** \( T(i,j) = (i+j,j) \) (a multi-dimensional timestamp)

- Dependences: \((1,0)\) and \((0,1)\) now become \((1,0)\) and \((1,1)\) resp.
- Inner loop is now parallel
**Domains, Dependences, and Schedules**

```c
for (i = 1; i <= N - 1; i++)
for (j = 1; j <= N - 1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

```c
for (t1=2;t1<=2*N-2;t1++) {
    #pragma omp parallel
    for private(lbv,ubv)
    for (t2 = max(1,t1-N+1); t2 <= min(N-1,t1-1); t2++) {
        a[(t1-t2)][t2] = a[(t1-t2) - 1][t2] + a[(t1-t2)][t2 - 1];
    }
}
```

**Schedule:** $T(i,j) = (i + j, j)$ (a multi-dimensional timestamp)
- Dependences: $(1,0)$ and $(0,1)$ now become $(1,0)$ and $(1,1)$ resp.
- Inner loop is now parallel
Introduction: Role of Compiler Infrastructure

**MLIR**
- Representation
- Polyhedral Framework: A Quick Intro
- Polyhedral Notions in MLIR
  - Data types

High-performance code generation in MLIR

Opportunities and Conclusions
AFFINE FUNCTIONS

- Affine for functions is linear + constant
  - Addition of identifiers, multiplication with a constant, floordiv, mod, ceildiv
    with respect to a positive constant
- Examples of affine functions of $i, j$:
  - $i + j, 2i - j, i + 1, 2i + 5,$
  - $i/128 + 1, i\%8, (i + j)/8,$
  - $((d0 \times 9216 + d1 \times 128) \mod 294912) \floordiv 147456$
- Not affine: $ij, i/j, j/N, i^2 + j^2, a[j]$
Polyhedral Notions in MLIR

▶ IR structures
  ▶ Affine maps
  ▶ Integer sets
▶ Operations
  1. affine.for
  2. affine.if
  3. affine.graybox (still a proposal)
  4. affine.apply
Polyhedral Notions in MLIR

- IR structures
  - Affine maps
  - Integer sets
- Operations
  1. affine.for
  2. affine.if
  3. affine.graybox (still a proposal)
  4. affine.apply
Affine Maps in MLIR

- An affine map maps zero or more identifiers to one or more result affine expressions

\[
\begin{align*}
#map1 &= (d0) \rightarrow ((d0 \ \text{floordiv} \ 4) \ \text{mod} \ 2) \\
#map2 &= (d0) \rightarrow (d0 - 4) \\
#map3 &= (d0) \rightarrow (d0 + 4) \\
#map4 &= (d0, d1) \rightarrow (d0 \ast 16 - d1 + 15) \\
#map5 &= (d0, d1, d2, d3) \rightarrow (d2 - d0 \ast 16, d3 - d1 \ast 16)
\end{align*}
\]

- Why affine maps? What can they express?
  - Loop IV mappings for nearly every useful loop transformation, data layout transformations, placement functions / processor mappings / distributions: block, cyclic, block-cyclic, multi-dimensional array subscripts, loop bound expressions, conditionals
Where are Affine Maps Used in MLIR?

1. IV remappings: to map old IVs to new IVs
   - \((i, j)\) Identity
   - \((j, i)\) Interchange
   - \((i, i + j)\) Skew j
   - \((2i, j)\) Scale i by two
   - \((i, j + 1)\) Shift j
   - \(\left\lfloor \frac{i}{32} \right\rfloor, \left\lfloor \frac{j}{32} \right\rfloor, i, j\) Tile (rectangular)

2. Loop bounds
3. Memref access subscripts
4. As an attribute for any instruction:

   \#map = (d0) -> (2*d0 - 1)

   ```mlir
   affine.for %i1 = 0 to #map(%N) {
     affine.for %i2 = 0 to 3 {
       %v1 = affine.load %0[%i1 + %i2] : memref<100xf32>
       "op1"(%v1) : (f32) -> ()
     }
   }
   %v = "op"(%s, %t) {map: (d0, d1) -> (d1, d0)} : (f32) -> (f32)
   ```
▶ Affine expressions on the LHS that are $\geq$ or $= 0$
▶ Can be used to model several things besides `affine.if`

```plaintext
#set0 = (i)[N, M] : (i >= 0, -i + N >= 0, N - 5 == 0, -i + M + 1 >= 0)
```
Uses affine maps for lower and upper bounds

SSA values bind to dimensions and symbols of the maps

```c
#map6 = (d0) -> (480, d0 * -480 + 2048)
#map7 = (d0) -> (d0 * 60)
#map8 = (d0) -> (696, d0 * 60 + 60)

affine.for %arg3 = 0 to 5 {
    affine.for %arg4 = 0 to 12 {
        affine.for %arg5 = 0 to 128 {
            affine.for %arg6 = #map7(%arg4) to min #map8(%arg4) {
                affine.for %arg7 = 0 to min #map6(%arg3) {
                    affine.for %arg8 = 0 to 16 {
                        affine.for %arg9 = 0 to 3 {
                            %0 = affine.load %arg0[%arg6 * 3 + %arg9, %arg3 * 480 + %arg7] : memref<2088x2048xf64>
                            %1 = affine.load %arg1[%arg3 * 480 + %arg7, %arg5 * 16 + %arg8] : memref<2048x2048xf64>
                            %2 = affine.load %arg2[%arg6 * 3 + %arg9, %arg5 * 16 + %arg8] : memref<2088x2048xf64>
                            %3 = mulf %0, %1 : f64
                            %4 = addf %3, %2 : f64
                            affine.store %4, %arg2[%arg6 * 3 + %arg9, %arg5 * 16 + %arg8] : memref<2088x2048xf64>
                        }
                    }
                }
            }
        }
    }
}
```

Uday Bondhugula, IISc
Uses an integer set

SSA values bind to dimensions and symbols of the integer set

```c
affine if (d0, d1) : (d1 - d0 >= 0) (%arg0, %arg0) {
  %cf10 = addf %cf9, %cf9 : f32
}
```
What about non-affine?

- Control flow, multi-dimensional array subscripts, loop bounds
- Things that change with loop IVs, things that are constant but unknown (symbols/parameters in polyhedral literature), and things that are known constants
- There are restrictions on what can be used as “symbols” or “parameters” for polyhedral purposes.
What about non-affine?

Control flow, multi-dimensional array subscripts, loop bounds

Things that change with loop IVs, things that are constant but unknown (symbols/parameters in polyhedral literature), and things that are known constants

There are restrictions on what can be used as “symbols” or “parameters” for polyhedral purposes.
What about non-affine?

- Control flow, multi-dimensional array subscripts, loop bounds
- Things that change with loop IVs, things that are constant but unknown (symbols/parameters in polyhedral literature), and things that are known constants
- There are restrictions on what can be used as “symbols” or “parameters” for polyhedral purposes.
Grayboxes introduce a new polyhedral scope / symbol context
Allow modeling "non-affine" control flow / subscripts / bounds maximally via affine constructs without outlining functions

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    // Non-affine loop bound for k loop
    for (k = 0; k < pow(2, j); k++)
      for (l = 0; l < N; l++)
        // block loop body
        ...
```

%c2 = constant 2 : index
affine.for %i = 0 to %n {
  affine.for %j = 0 to %n {
    affine.graybox [] = () {
      %pow = call @powi(%c2, %j)
      affine.for %k = 0 to %pow {
        affine.for %l = 0 to %n {
          ...
        }
      }
    return
    } // graybox end
  } // %j
} // %i

Uday Bondhugula, IISc
OUTLINE

Introduction: Role of Compiler Infrastructure

MLIR
  Representation
  Polyhedral Framework: A Quick Intro
    Polyhedral Notions in MLIR
      Data types

High-performance code generation in MLIR

Opportunities and Conclusions
Types Relevant for Dense Matrices / Tensors

1. **tensor** A value that is a multi-dimensional array of elemental values

   ```
   %d = "tf.Add"(%e, %f) : (tensor<?x42x?xf32>, tensor<?x42x?xf32>) -> tensor<?x42x?xf32>
   ```

2. **memref** A buffer in memory or a view on a buffer, has a layout map, memory space qualifier, symbols bound to its dynamic dimensions

   ```
   %N = affine.apply (d0) -> (8 * (d0 ceildiv 8)) (%S)
   %M = affine.apply (d0) -> (2 * d0) (%N)
   #tmap = (d0, d1) -> (d1 floordiv 32, d0 floordiv 128, d1 mod 32, d0 mod 128)
   %A = alloc() : memref<1024x64xf32, #tmap, /*hbm=*/0>
   %B = alloc(%M, %N)[%x, %y] : memref<?x?xf32, #tmap, /*scratchpad=*/1>

   #shift = (d0, d1)[s0, s1] -> (d0 + s0, d1 + s1)
   %C = alloc(%M, %M)[%x, %y] : memref<?x?xf32, #shift, /*scratchpad=*/1>
   ```
1. **tensor** A value that is a multi-dimensional array of elemental values

```plaintext
%d = "tf.Add"(%e, %f) : (tensor<?x42x?xf32>, tensor<?x42x?xf32>) -> tensor<?x42x?xf32>
```

2. **memref** A buffer in memory or a view on a buffer, has a layout map, memory space qualifier, symbols bound to its dynamic dimensions

```plaintext
%S = affine.apply (d0) -> (8 * (d0 ceildiv 8)) (%S)
%M = affine.apply (d0) -> (2 * d0) (%N)
#tmap = (d0, d1) -> (d1 floordiv 32, d0 floordiv 128, d1 mod 32, d0 mod 128)
%A = alloc() : memref<1024x64xf32, #tmap, /*hbm=*/0>
%B = alloc(%M, %N)[%x, %y] : memref<?x?xf32, #tmap, /*scratchpad=*/1>

#shift = (d0, d1)[s0, s1] -> (d0 + s0, d1 + s1)
%C = alloc(%M, %M)[%x, %y] : memref<?x?xf32, #shift, /*scratchpad=*/1>
```
Outline

Introduction: Role of Compiler Infrastructure

MLIR
- Representation
  - Polyhedral Framework: A Quick Intro
  - Polyhedral Notions in MLIR
    - Data types

High-performance code generation in MLIR

Opportunities and Conclusions
State-of-the-Art Deep Learning Systems: Current Landscape

- Primarily driven by hand-optimized highly tuned libraries (manual or semi-automatic at most)
- Expert/Ninja programmers
- Not a scalable approach! — bleeds resources, not modular, too much repetition
Van Zee and Van de Geijn 2015 work on BLIS/FLAME has shown how to modularize/structure such Ninja implementations (Goto’s/OpenBLAS) for auto-generation.

Low et al. 2015 shows how parameters for such systematic implementations could be derived completely analytically!

Close to absolute machine peak performance achievable in a structured/more productive way (for Intel / AMD multicores)!

MLIR and its infrastructure could take this approach even further.

Turn a ninja / esoteric art into a more productive, automatable, and scalable approach.
Van Zee and Van de Geijn 2015 work on BLIS/FLAME has shown how to modularize/structure such Ninja implementations (Goto’s/OpenBLAS) for auto-generation.

Low et al. 2015 shows how parameters for such systematic implementations could be derived completely analytically!

Close to absolute machine peak performance achievable in a structured/more productive way (for Intel / AMD multicores!)

MLIR and its infrastructure could take this approach even further.

Turn a ninja / esoteric art into a more productive, automatable, and scalable approach.
Van Zee and Van de Geijn 2015 work on BLIS/FLAME has shown how to modularize/structure such Ninja implementations (Goto’s/OpenBLAS) for auto-generation.

Low et al. 2015 shows how parameters for such systematic implementations could be derived completely analytically!

Close to absolute machine peak performance achievable in a structured/more productive way (for Intel / AMD multicores)!

MLIR and its infrastructure could take this approach even further.

Turn a ninja / esoteric art into a more productive, automatable, and scalable approach.
OpenBLAS/BLIS Approach to Tiling

Uday Bondhugula, IISc
OpenBLAS/BLIS Approach to Tiling

Schedule:

\[(i, j, k) \rightarrow \left( \frac{j}{N_C}, \frac{k}{K_C}, \frac{i}{M_C}, \frac{j}{N_R}, \frac{i}{M_R}, k, j \% N_R, i \% M_R \right)\]
RECREATING DGEMM IN MLIR

MLIR performance with various opts enabled/disabled

<table>
<thead>
<tr>
<th>Options</th>
<th>GLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec</td>
<td>1.51</td>
</tr>
<tr>
<td>vec + cache tile</td>
<td>10.79</td>
</tr>
<tr>
<td>vec + cache tile + pack</td>
<td>15.57</td>
</tr>
<tr>
<td>vec + cache tile + reg tile</td>
<td>22.53</td>
</tr>
<tr>
<td>All opts (vec + cache tile + reg tile + pack)</td>
<td>61.94</td>
</tr>
<tr>
<td>Absolute peak</td>
<td>75.2</td>
</tr>
</tbody>
</table>
RECREATING DGEMM IN MLIR

Within 9% of MKL/OpenBLAS performance!
Recreating SGEMM in MLIR

Within 2% of MKL/OpenBLAS performance!
Introduction: Role of Compiler Infrastructure

**MLIR**
- Representation
- Polyhedral Framework: A Quick Intro
- Polyhedral Notions in MLIR
  - Data types

High-performance code generation in MLIR

**Opportunities and Conclusions**
opportunities

- Migrate and rebuild existing polyhedral infrastructure in a principled way on MLIR ⇒ greater impact / industry transfer / reuse
- Transform both iteration spaces and data spaces; better phase ordering / interaction with SSA
- Building new DSLs/programming models? Use MLIR!
- Building new ML/AI chips? Create an MLIR backend!
Migrate and rebuild existing polyhedral infrastructure in a principled way on MLIR ⇒ greater impact / industry transfer / reuse

Transform both iteration spaces and data spaces; better phase ordering / interaction with SSA

Building new DSLs/programming models? Use MLIR!
Building new ML/AI chips? Create an MLIR backend!
OPPORTUNITIES

➤ Migrate and rebuild existing polyhedral infrastructure in a principled way on MLIR ⇒ greater impact / industry transfer / reuse
➤ Transform both iteration spaces and data spaces; better phase ordering / interaction with SSA
➤ Building new DSLs/programming models? Use MLIR!
➤ Building new ML/AI chips? Create an MLIR backend!
OPPORTUNITIES

- Migrate and rebuild existing polyhedral infrastructure in a principled way on MLIR ⇒ greater impact / industry transfer / reuse
- Transform both iteration spaces and data spaces; better phase ordering / interaction with SSA
- Building new DSLs/programming models? Use MLIR!
- Building new ML/AI chips? Create an MLIR backend!
Need for reusable and modular common IR infrastructure to lower compute graphs to high-performance code

- Lowering should be progressive — input and output of passes/utilities should be easy to represent and transform
- Infrastructure for analysis and transformation should be reused, not replicated
- High-performance libraries and code generators should coexist, interoperate, and compose
- General-purpose and domain-specific techniques can coexist on the same IR infrastructure
CONCLUSIONS

- Need for reusable and modular common IR infrastructure to lower compute graphs to high-performance code
- Lowering should be **progressive** — input and output of passes/utilities should be **easy to represent and transform**
- Infrastructure for analysis and transformation should be **reused**, not replicated
- High-performance libraries and code generators should coexist, interoperate, and compose
- General-purpose and domain-specific techniques can coexist on the same IR infrastructure
CONCLUSIONS

- Need for reusable and modular common IR infrastructure to lower compute graphs to high-performance code
- Lowering should be **progressive** — input and output of passes/utilities should be **easy to represent and transform**
- Infrastructure for analysis and transformation should be **reused**, not replicated
- High-performance libraries and code generators should coexist, interoperate, and compose
- General-purpose and domain-specific techniques can coexist on the same IR infrastructure
Interested?

1. Contribute to MLIR (part of LLVM now):
https://github.com/llvm-project/llvm

2. Several collaboration opportunities with academia and industry!

3. Several employment opportunities!

4. Pointers
   4.1 MLIR documentation: https://mlir.llvm.org
   4.2 My branches: https://github.com/llvm-project/bondhugula/