Classical Loop Nest Transformation Framework on MLIR

Vinay M, Ranjith Kumar H, Siddharth Tiwary, Prashantha NR
Compiler Tree Technologies
Existing Loop Transforms in MLIR

- Works on Affine dialect operations
- No generic Analysis framework yet
  - Dependence analysis are local to loop nests
- No unified driver for all loop transforms
- Most transformations works only if all the loops in a loop nest are `AffineForOp`
Not all Affine loops can be converted to Affine Ops

- Custom Types may not be converted / “cast”ed to std.memrefs
  - Example: Array of structures
    ```
    %3 = fc.allocate : !fc.ref<fc.array<10 x fc.struct_type<i32, f32>>>
    ```
    - Different loop nest transformations for different types?

- Memory Dependence analysis not just for Affine Ops
  - What happens to custom dialect operations inside Affine loops?
  - affine.store vs. std.store vs. vector.load vs. fc.load (or any custom dialect)
Not all Affine loops can be converted to Affine Ops

- In few cases, better to do loop transforms on higher level Dialects
  - Example: Fortran do loops with labels
- Restrictions on Affine Symbols and Dimensions
- All the loops in the loop nest may not be “affine.for”
- **Lower conversion rate to Affine Ops**
- AffineMap and AffineExpr can be freely used in custom Dialects
Heuristic Based Classic Loop Transformation Framework

- A proof-of-concept implementation of loop transformations along with a cost aware driver.
- Built as wrapper around Affine Dialect data structures
- Different Loop Transformations:
  - Unimodular Transformations (Loop Permute and Loop Blocking)
  - Loop Fission, Loop Fusion
- A basic profitability model based on cache utilization.
- AliasAnalysis (basic-aa), Dependence Analysis, etc ported from LLVM infrastructure
- Mem2reg, licm, etc as pre-processing steps
- Driver is currently written for Data Locality but it can be tuned for any custom workloads/ hardware.
Various inputs to the framework

- **Focus on SPEC CPU 2017 benchmarks: Fortran / C++ / C**
  - Fortran Dialect: FC compiler
    - Loop representations: `do, do while, forall, parallel do`
    - Array section operations: converted to affine.for
    - I/O operations
    - Various intrinsic functions
  - CIL Dialect: C/ C++ representation in MLIR
    - Low level IR (pointer type based)
    - Experimental path
- **TODO: Tensorflow XLA**
  - Affine loops generated from lhlo
FC and MLIR

MLIR Dialects

FC AST
- std Dialect
- FC Dialect
- SCF Dialect
- Affine Dialect

MLIR Optimizations
- Mem2reg, licm ...
- Loop Nest Transforms
- ...
- LLVM Lowering

LLVM IR
FC: Affine Dialect Conversion and Canonicalization of Loop Nests

**Fortran 90:**
```fortran
do i = 1, 10
    do j = 1, 20
        b(i+1, j+2) = c(i, j)
    enddo
endo
do
```

**Affine Dialect:**
```mlir
affine.for %arg0 = 1 to 11 {
    affine.for %arg1 = 1 to 21 {
        %4 = affine.load %1[%arg0, %arg1] : memref<10x20xi32, #map0>
        affine.store %4, %2[%arg0 + 1, %arg1 + 2] : memref<11x22xi32, #map0>
    }
}
```

**Loop Dialect:** (sub-optimal IR)
```mlir
scf.for %arg0 = %4 to %7 step %6 {
    %c1_i32_3 = constant 1 : i32
    %8 = index_cast %c1_i32_3 : i32 to index
    ...!
    scf.for %arg1 = %8 to %11 step %10 {
        %12 = load %1[%arg0, %arg1] {name = "c"} : memref<10x20xi32, #map0>
        %c1_i32_5 = constant 1 : i32
        %13 = index_cast %c1_i32_5 : i32 to index
        %14 = addi %arg0, %13 : i32
        %c2_i32 = constant 2 : i32
        %15 = index_cast %c2_i32 : i32 to index
        %16 = addi %arg1, %15 : index
        store %12, %2[%14, %16] {name = "b"} : memref<11x22xi32, #map0>
    }
}
```

**FC MLIR codegen**
Analysis Passes
Alias and Dependence Analysis

- **Alias Analysis**
  - Generic Infrastructure for existing / custom Dialect memory operations
  - Invoked using AliasSetTracker (ported from LLVM)
  - Implemented by BasicAA and Dependence Analysis
Alias and Dependence Analysis

- **Alias Analysis**
  - Generic Infrastructure for existing / custom Dialect memory operations
  - Invoked using AliasSetTracker (ported from LLVM)
  - Implemented by BasicAA and Dependence Analysis

- **Dependence Analysis:**
  - Ported from LLVM
  - Works on Affine data structures (mlir::AffineExpr)
  - Uses Alias Analysis
Alias and Dependence Analysis

● Alias Analysis
  ○ Generic Infrastructure for existing / custom Dialect memory operations
  ○ Invoked using AliasSetTracker (ported from LLVM)
  ○ Implemented by BasicAA and Dependence Analysis

● Dependence Analysis:
  ○ Ported from LLVM
  ○ Works on Affine data structures (mlir::AffineExpr)
  ○ Uses Alias Analysis

● Dependence Matrix
  ○ Wrapper on top of Dependence Analysis
  ○ Contains all the dependencies in the given loop nest.
  ○ Contains m x n dependence matrix, where ‘m’ is number of dependences and ‘n’ is number of loops in the nest
for (int i = 0; i < n; ++i) {
    for (int j = 1; j < m; ++j) {
        for (int k = 1; k < l; ++k) {
            a[i+1][j+1][k] = a[i][j][k] + a[i][j+1][k+1];
        }
    }
}
Legality of transformation
Loop Cost Analysis

- Gives out a cost for each loop in its nest based on cache misses.
  - Permute, Split, Fuse, Blocking, Prefetching and other cache related opts can use this data.
- Loop Cost for each loop is calculated as follows:
  - A penalty is assigned to the loop based on the amount of cache misses it will cause to the references in the loop nest.
  - Group the references that belong to the same cache line and assign penalty,
    - If the reference is a “scalar” value with respect loop then penalty us 1.
    - If the reference is a “strided” access w.r.t. the loop, then the penalty is $\frac{\text{TripCount}}{\text{CacheLineSize}}$
    - If the reference is a “non-strided” access w.r.t. the loop, then penalty is TripCount
  - Total Cost = Cost due to penalties $\times$ number of times the loop executes due to outer loops.
- Concerns:
  - Need to get CacheLineSize from Target to accurately calculate cost for a given processor.
Loop Cost

for (int i = 1; i < n; ++i) {
    for (int j = 1; j < n; ++j) {
        B[i][j+10] += C[j][i] + D[i][j];
    }
}

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B[i][j+10]</td>
<td>n / L</td>
</tr>
<tr>
<td>C[j][i]</td>
<td>n</td>
</tr>
<tr>
<td>D[i][j]</td>
<td>n/L</td>
</tr>
<tr>
<td>Total Cost</td>
<td>n ( n + 2n/L)</td>
</tr>
</tbody>
</table>

Strided access for j loop
Non-contiguous access for j loop
Contiguous access for j loop
Loop Cost

```c
for (int j = 1; j < n; ++j) {
    for (int i = 1; i < n; ++i) {
        B[i][j+10] += C[j][i] + D[i][j];
    }
}
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B[i][j+10]</td>
<td>n</td>
</tr>
<tr>
<td>C[j][i]</td>
<td>n/L</td>
</tr>
<tr>
<td>D[i][j]</td>
<td>n</td>
</tr>
<tr>
<td>Total Cost</td>
<td>n ( 2n + n / L )</td>
</tr>
</tbody>
</table>

Non-contiguous access for loop i
Contiguous access for i
Non-contiguous access for i loop
Pre-processing of Loop Nests
Pre-processing passes

- Helps in creating Perfect Loop Nests

- Promote Memory to Register (mem2reg):
  - Works similar to LLVM's mem2reg
  - Works on memrefs
  - No restriction on Alloca / Memory access operations (can be from affine / std, etc)

- Hoisting invariants (LICM):
  - Similar to LLVM's licm pass: Hoists invariants out of Loops
  - Uses Alias Analysis

- Sinking operations:
  - Tries to sink operations to innermost loop
  - Uses Alias Analysis

- Affine Normalization
  - Create one Affine map for the loop nest
Example

```plaintext
subroutine foo(a, b, c)
    integer :: a(10, 10), b(10, 10), c(10, 10)
    integer :: i, j
    integer :: k

    do i = 1, 10
        k = i * i
        do j = 2, 10
            a(i, j) = b(i, j) + c(j, i) + k
        end do
    end do
end subroutine
```

```plaintext
module {
    func @foo(%arg0: !fc.ref<!fc.array<1:10 x 1:10 x 132>>, %arg1: ...) {
        %c2_i32 = constant 2 : i32
        %c10_i32 = constant 10 : i32
        %c1_i32 = constant 1 : i32
        %c11_i32 = constant 11 : i32
        %0 = fc.allocate j : !fc.ref<i32>
        %1 = fc.allocate k : !fc.ref<i32>
        fc.do %arg3 = %c1_i32, %c10_i32, %c1_i32 {construct_name = ""} {
            %2 = muli %arg3, %arg3 : index
            %3 = index_cast %2 : index to i32
            fc.store %3, %1 [name = "k"] : !fc.ref<i32>
            fc.do %arg4 = %c2_i32, %c1_i32, %c1_i32 {construct_name = ""} {
                %4 = fc.load %arg1[%arg3, %arg4]
                %5 = fc.load %arg2[%arg4, %arg3]
                %6 = addi %4, %5 : i32
                %7 = fc.load %1 [name = "k"] : i32
                %8 = addi %6, %7 : i32
                fc.store %8, %arg0[%arg3, %arg4]
            } enddo {construct_name = ""}
        } enddo {construct_name = ""}
        return
    }
}
```
Example

```
module {
  func @foo(%arg0: !fc.ref<!fc.array<1:10 x 1:10 x i32>>, %arg1: ...) {
    %c2_i32 = constant 2 : i32
    %c10_i32 = constant 10 : i32
    %c1_i32 = constant 1 : i32
    %c11_i32 = constant 11 : i32
    %0 = fc.allocate j : !fc.ref<i32>
    %1 = fc.allocate k : !fc.ref<i32>
    fc.do %arg3 = %c1_i32, %c10_i32, %c1_i32 {construct_name = ""} {
      %2 = muli %arg3, %arg3 : index
      %3 = index_cast %2 : index to i32
      fc.store %3, %1 {name = "k"} : !fc.ref<i32>
    }
    fc.do %arg4 = %c2_i32, %c10_i32, %c1_i32 {construct_name = ""} {
      %4 = fc.load %arg1[%arg3, %arg4]
      %5 = fc.load %arg2[%arg4, %arg3]
      %6 = addi %4, %5 : i32
      %7 = fc.load %1 {name = "k"} : i32
      %0 = addi %6, %7 : i32
      fc.store %0, %arg0[%arg3, %arg4]
    } enddo {construct_name = ""}
  } enddo {construct_name = ""}
  return
}
```

```
module {
  func @foo(%arg0: !fc.ref<!fc.array<1:10 x 1:10 x i32>>, %arg1: ...) {
    %c2_i32 = constant 2 : i32
    %c10_i32 = constant 10 : i32
    %c1_i32 = constant 1 : i32
    %c11_i32 = constant 11 : i32
    %0 = fc.allocate j : !fc.ref<i32>
    %1 = fc.allocate k : !fc.ref<i32>
    fc.do %arg3 = %c1_i32, %c10_i32, %c1_i32 {construct_name = ""} {
      %5 = muli %arg3, %arg3 : index
      %6 = index_cast %5 : index to i32
      %7 = fc.load %arg1[%arg3, %arg4] ...
      %8 = fc.load %arg2[%arg4, %arg3] ...
      %9 = addi %7, %8 : i32
      %10 = addi %9, %6 : i32
      fc.store %10, %arg0[%arg3, %arg4]
    } enddo {construct_name = ""}
    return
  }
```

Loop Transformation Driver
Loop Transformation Driver

- Generic framework
  - Works on affine / scf /user-defined dialect by writing converter

- Algorithm:
  - Aggressively split the loop nest across Statements and Sibling loops
  - Run pre-processing on the loop nests (if needed)
  - Run the unimodular transformations on the single perfect loop nest
  - Aggressively fuse the loops whenever feasible

- Loop Fusion and Unimodular Transformations are driven using profitability models
Creation of Perfect Loop Nests: Loop Splitting

- Recursively split the loop nests based on Dependence Analysis to generate Perfect Loop Nests
- Input to Unimodular Transforms

```fortran
subroutine foo(a, b, c)
  integer :: a(10, 10), b(10, 10), c(10, 10)
  integer :: i, j

  do i = 1, 10
    c(i, i) = i * i
    do j = 2, 10
      a(i, j) = b(i, j) + c(i, j)
    end do
  end do
end subroutine
```

```fortran
subroutine foo(a, b, c)
  integer :: a(10, 10), b(10, 10), c(10, 10)
  integer :: i, j

  do i = 1, 10
    c(i, i) = i * i
    do j = 2, 10
      a(i, j) = b(i, j) + c(i, j)
    end do
  end do
end subroutine
```
Example

```plaintext
module {
  func @foo(%arg0: !fc.ref<!fc.array<1:10 x 1:10 x i32>>, %arg1: ...) {
    %2 i32 = constant 2 : i32
    %10 i32 = constant 10 : i32
    %c1 i32 = constant 1 : i32
    %c1 i32 = constant 11 : i32
    %0 = fc.allocate j : !fc.ref<i32>
    fc.do %arg3 = %c1 i32, %c10 i32, %c1 i32 {construct_name = ""} {
      %1 = muli %arg3, %arg3 : index
      %2 = index_cast %1 : index to i32
      fc.store %2, %arg2[%arg3, %arg3] ...
      fc.do %arg4 = %c2 i32, %c10 i32, %c1 i32 {construct_name = ""} {
        %3 = fc.load %arg1[%arg3, %arg4] ...
        %4 = fc.load %arg2[%arg3, %arg4] ...
        %5 = addi %3, %4 : i32
        fc.store %5, %arg0[%arg3, %arg4] ...
      } enddo {construct_name = ""}
  } enddo {construct_name = ""}
  return
}
```
```
Unimodular transformations

- Represented by a unimodular transformation matrix (determinant 1 or -1)
- Composition of loop permutation, skewing, reverse
- \( T * i = i' \), \( T \) is the transformation matrix, \( i \) and \( i' \) are dependence matrices
- Transformation is legal if the transformed dependence matrix is lexicographically positive
- Eg: for permute of (2-d loop nest), \( T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \)
Unimodular transformations

● Input: Perfect Loop Nests
● Analysis
  ○ Legality
  ○ Uses dependence analysis and then cost analysis on loop nest to output the optimal transformation matrix for the given loop nest
● Transformation
  ○ Generate loop bounds in transformed space
  ○ Perform **Fourier-motzkin elimination** to simplify the transformed bounds
  ○ Validate and update the loop bounds for all loops in the nest
  ○ Update all memory accesses
    i. Crate a map of old indvars -> new indvars
    ii. Rewrite the accesses using the new indvars information
Example: Matrix Multiplication (for vectorization)

for (int i = 1; i < n; ++i) {
    for (int j = 1; j < n; ++j) {
        A[i][j] = 0;
        for (int k = 1; k < n; ++k) {
            A[i][j] += B[i][k] * C[k][j];
        }
    }
}

for (int i = 1; i < n; ++i) {
    for (int j = 1; j < n; ++j) {
        A[i][j] = 0;
        for (int k = 1; k < n; ++k) {
            A[i][j] += B[i][k] * C[k][j];
        }
    }
}

Split

Unimodular Transforms (Permute)

Loop Cost aids it.

for (int i = 1; i < n; ++i) {
    for (int j = 1; j < n; ++j) {
        A[i][j] = 0;
        for (int k = 1; k < n; ++k) {
            A[i][j] += B[i][k] * C[k][j];
        }
    }
}

Aggressively apply splitting → unimodular transformations → fusion

Fuse

\[ T = \begin{pmatrix}
    1 & 0 & 0 \\
    0 & 0 & 1 \\
    0 & 1 & 0
\end{pmatrix} \]
## Loop Permutation

### Innermost Loop (i is outermost)

<table>
<thead>
<tr>
<th>Loop</th>
<th>A[i][j]</th>
<th>B[i][k]</th>
<th>C[k][j]</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>$\frac{n^3}{b}$</td>
<td>$n^2$</td>
<td>$\frac{n^3}{b}$</td>
<td>$2n^3/b + n^2$</td>
</tr>
<tr>
<td>k</td>
<td>$n^2$</td>
<td>$\frac{n^3}{b}$</td>
<td>$n^3$</td>
<td>$n^3(1+1/b) + n^2$</td>
</tr>
</tbody>
</table>

$b$ is the cache line size for the target

```c
for (int i = 1; i < n; ++i) {
for (int j = 1; j < n; ++j) {
for (int k = 1; k < n; ++k) {
    A[i][j] += B[i][k] * C[k][j];
}
}
}
for (int i = 1; i < n; ++i) {
    for (int j = 1; j < n; ++j) {
        A[i][j] = 0;
    }
}
```
## Loop Permutation

### j as innermost loop gives lesser cost!

<table>
<thead>
<tr>
<th>Innermost Loop (i is outermost)</th>
<th>A[i][j]</th>
<th>B[i][k]</th>
<th>C[k][j]</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>$n^3/b$</td>
<td>$n^2$</td>
<td>$n^3/b$</td>
<td>$2n^3/b + n^2$</td>
</tr>
<tr>
<td>k</td>
<td>$n^2$</td>
<td>$n^3/b$</td>
<td>$n^3$</td>
<td>$n^3(1+1/b) +n^2$</td>
</tr>
</tbody>
</table>

```cpp
for (int i = 1; i < n; ++i)
    for (int j = 1; j < n; ++j)
        A[i][j] = 0;

for (int i = 1; i < n; ++i) {
    for (int k = 1; k < n; ++k) {
        for (int j = 1; j < n; ++j) {
            A[i][j] += B[i][k] * C[k][j];
        }
    }
}
```
Loop Blocking

- Access data in blocks to exploit temporal and spatial locality
- Transform a loop at a depth into two loops:
  - One loop for iterating inside each block
  - One loop for iterating over the blocks
- Block size
  - fixed at compile time (each depth can have a different one)
  - depends on cache size and cache line size
  - determined by tuning
- Strip-mining and interchange

```c
for (int i = 0; i < n; ++i)
for (int j = 0; j < n; ++j)
```

Strip-mining

```c
for (int i = 0; i < n; ++i)
for (int j = 0; j < n; j+=B)
for (int jj = j; jj < min(n, j+B-1); jj++)
```

Interchange

```c
for (int j = 0; j < n; j+=B)
for (int i = 0; i < n; ++i)
for (int jj = j; jj < min(n, j+B-1); jj++)
```
Loop Blocking - matrix multiplication

for (int i = 0; i < n; ++i)
for (int j = 0; j < n; ++j)
for (int k = 0; k < n; ++k)
A[i][j] += B[i][k] * C[k][j];

Cache misses for array B: $n^3/b$
Cache misses for array C: $n^3$

for (int ii = 0; ii < n; ii+=B)
for (int jj = 0; jj < n; jj+=B)
for (int kk = 0; kk < n; kk+=B)
for (int i = ii; i < ii+B; ++i)
for (int j = jj; j < jj+B; ++j)
for (int k = kk; k < kk+B; ++k)
A[i][j] += B[i][k] * C[k][j];

Cache misses for array B: $B^2/b \cdot n^3/B^3$
$= n^3/(Bb)$
Cache misses for array C: $B^2/b \cdot n^3/B^3$
$= n^2/(Bb)$

for (int ii = 0; ii < n; ii+=B)
for (int jj = 0; jj < n; jj+=B)
for (int kk = 0; kk < n; kk+=B)
for (int i = 0; i < n; ++i)
for (int j = jj; j < jj+B; ++j)
for (int k = kk; k < kk+B; ++k)
A[i][j] += B[i][k] * C[k][j];

for (int ii = 0; ii < n; ii+=B)
for (int jj = 0; jj < n; jj+=B)
for (int kk = 0; kk < n; kk+=B)
for (int i = ii; i < ii+B; ++i)
for (int j = jj; j < jj+B; ++j)
for (int k = kk; k < kk+B; ++k)
A[i][j] += B[i][k] * C[k][j];
Loop Blocking

- Transformation: given a Loop-Nest $L_0, \ldots, L_k$
  - Strip-mine each $L_i$ in consideration into $L'_i$ and $L''_i$
  - Move all $L'_i$ to outside
- Strip-mining is always legal
- Loop interchange not always legal
  - All loops in consideration must be safe to be moved outside
  - Each such loop must have only “=” or “<” in all the dependence vectors
- Profitability
  - Look for good reuse candidate in outer-loop iterations
    - should carry small-threshold dependencies of any type carried by the loop
    - loop index occurs with small stride in contiguous dimension, and in no other dimension
  - Need to account for misses because of the outer-strip loops (for the dependencies carried by the innermost loop)
Results

- We could transform the hot loop nest in `bwaves_r` SPEC CPU 2017 benchmark see decent gain.
- We see around 70% gain in matmul() kernel, etc.
Next steps

- Add more Unimodular transformations
- Open source
- Integrate the Framework with TensorFlow XLA compiler
- Run more benchmarks
Thank You