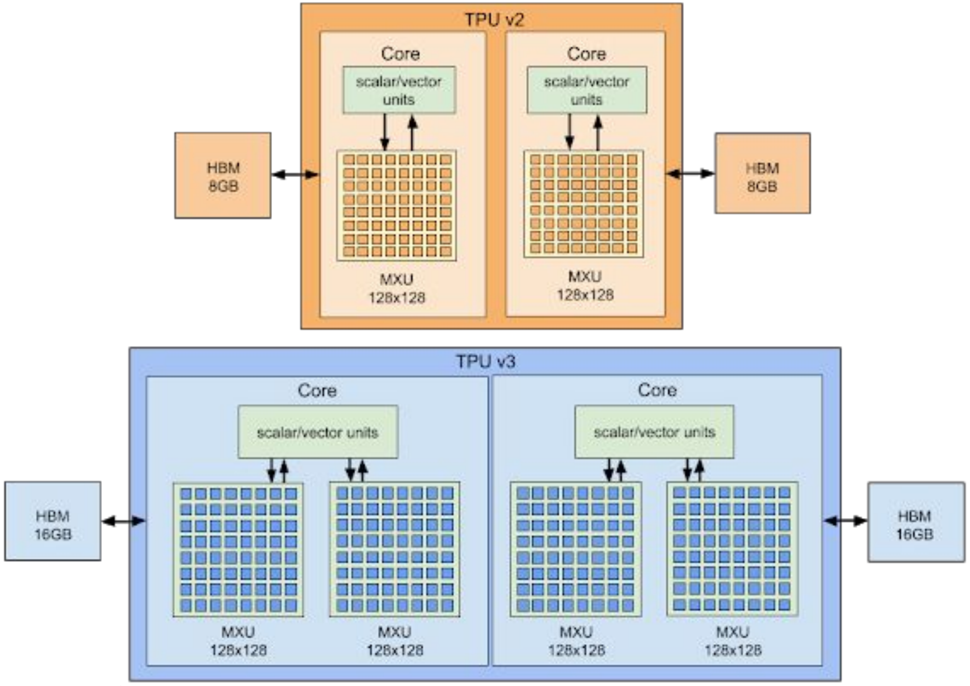
An overview of loop nest optimization, parallelization and acceleration in MLIR

MLIR4HPC, October 21, 2019

albertcohen@google.com
presenting the work of many

Programming Tiled SIMD Hardware

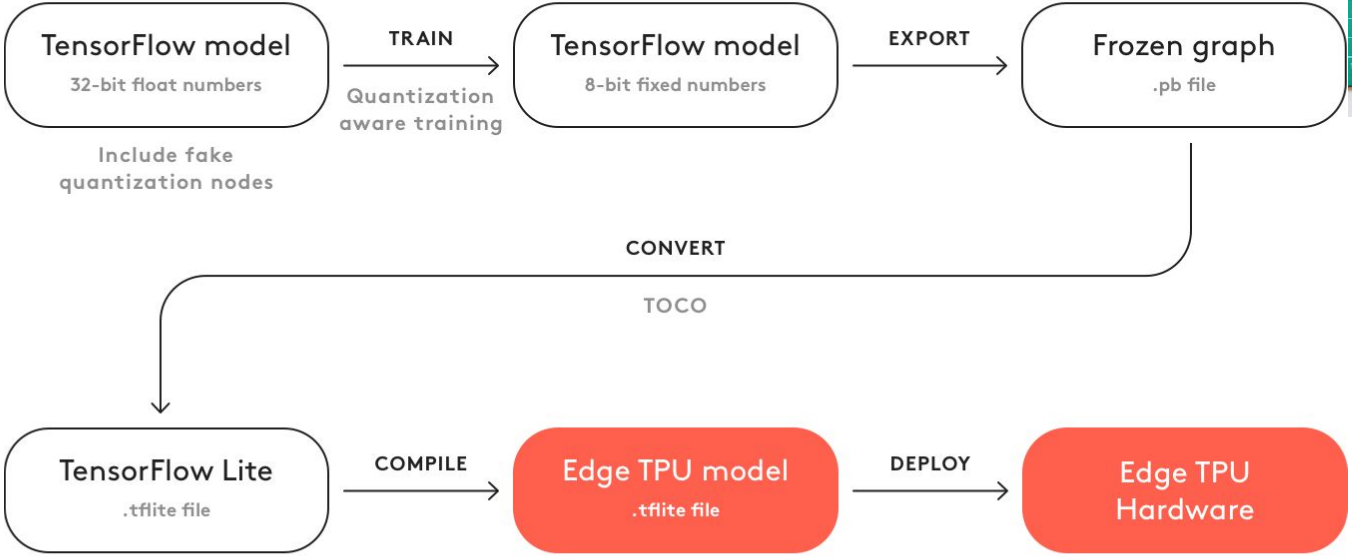


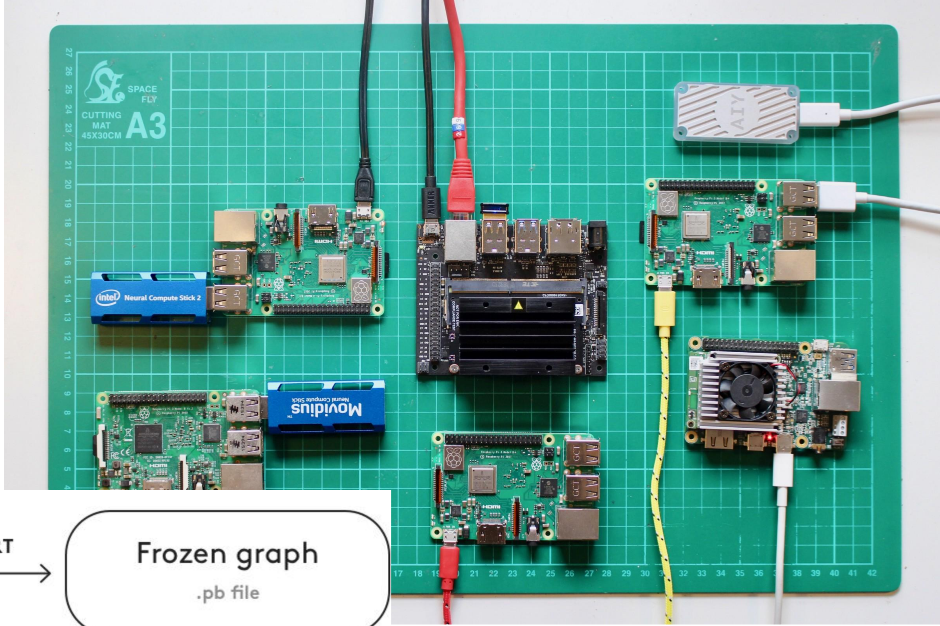
Google

From Supercomputing to Embedded HPC

Highly specialized hardware

e.g. Google Edge TPU





Edge and embedded computing zoo

Single Op Compiler

Tiled and specialized hardware

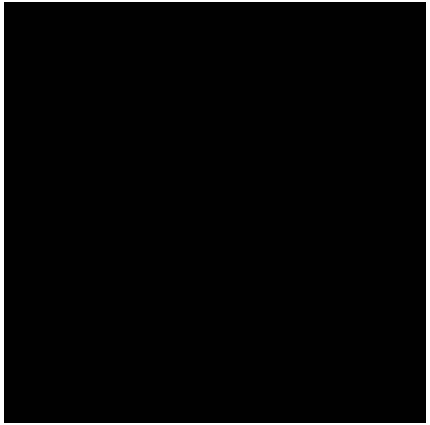
1. data layout
2. control flow
3. data flow
4. data parallelism

Example: Halide for image processing pipelines

https://halide-lang.org

Meta-programming API and domain-specific language (DSL)
for loop transformations, numerical computing kernels

Tiling in Halide



Tiled schedule:
strip-mine (a.k.a. split)
permute (a.k.a. reorder)

Vectorized schedule:
strip-mine
vectorize inner loop

Non-divisible bounds/extent:
strip-mine
shift left/up
redundant computation
(also forward substitute/inline operand)

Single Op Compiler

Tiled and specialized hardware

1. data layout
2. control flow
3. data flow
4. data parallelism

Example: Halide for image processing pipelines

https://halide-lang.org

XLA, TVM for neural networks

https://www.tensorflow.org/xla https://tvm.ai

TVM example: scan cell (RNN)

```
m = tvm.var("m")
n = tvm.var("n")
X = tvm.placeholder((m,n), name="X")
s_state = tvm.placeholder((m,n))
s_init = tvm.compute((1,n), lambda _,i: X[0,i])
s_do = tvm.compute((m,n), lambda t,i: s_state[t-1,i] + X[t,i])
s_scan = tvm.scan(s_init, s_do, s_state, inputs=[X])
s = tvm.create_schedule(s_scan.op)

// Schedule to run the scan cell on a CUDA device
block_x = tvm.thread_axis("blockIdx.x")
thread_x = tvm.thread_axis("threadIdx.x")
xo,xi = s[s_init].split(s_init.op.axis[1], factor=num_thread)
s[s_init].bind(xo, block_x)
s[s_init].bind(xi, thread_x)
xo,xi = s[s_do].split(s_do.op.axis[1], factor=num_thread)
s[s_do].bind(xo, block_x)
s[s_do].bind(xi, thread_x)
print(tvm.lower(s, [X, s_scan], simple_mode=True))
```

Google

Tiling... And Beyond?

1. But what about **symbolic bounds, sizes**, **shapes**?
2. **Other transformations**: fusion, fission, pipelining, unrolling... ?
3. **Composition** with other **transformations** and **mapping** decisions?
4. **Consistency** with ... ?
5. **Reuse** across library generation instances?
6. **Evaluating** cost functions, **enforcing** resource constraints?

⟶ Impact on compiler construction,
intermediate representations,
program analyses and transformations?

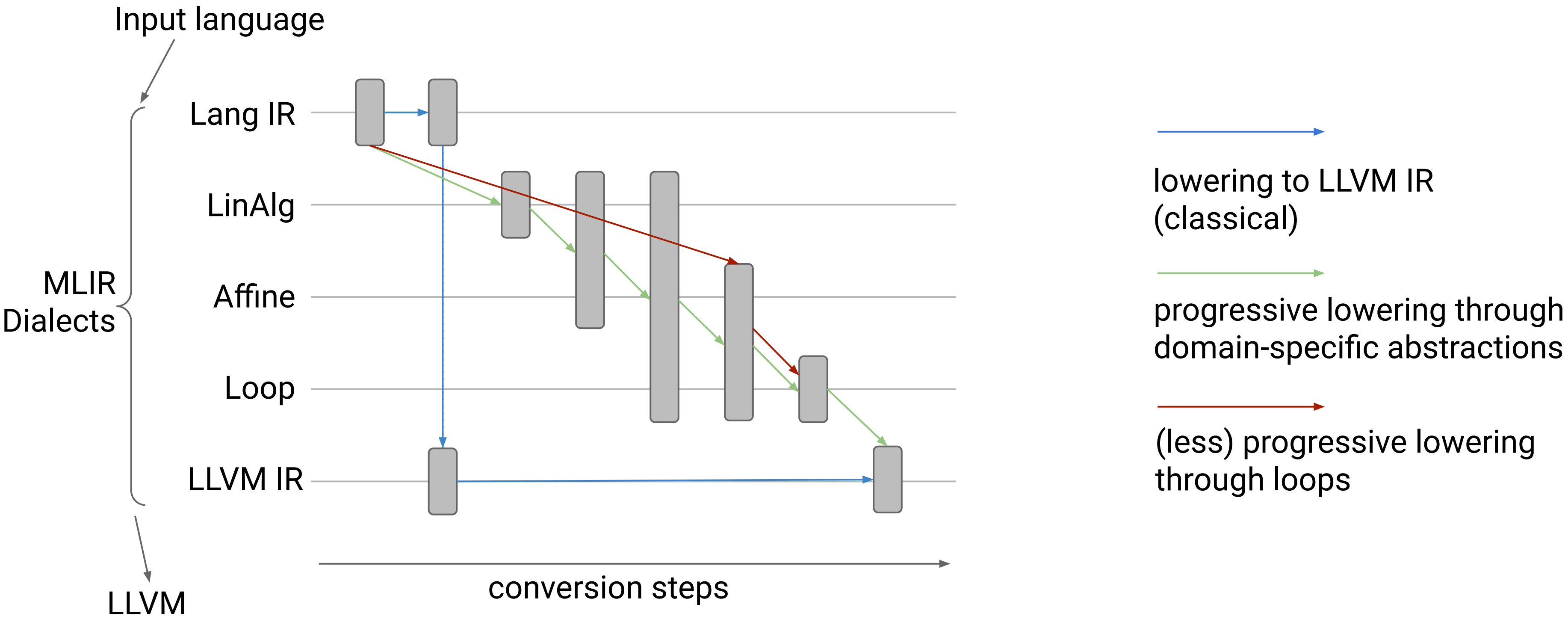
MLIR's answer

- the right abstraction at the right time
- gradual conversion, refinement, lowering
- extend and reuse

- *listen, learn as we go!*

MLIR Code Generation Flows

Tentative, Alex Zinenko's snapshot



Google

Example: Affine Dialect

for General-Purpose Loop Nest Optimization

from Uday Bondhugula and Andy Davis

Affine Dialect for Loop Nest Optimization

```
func @test() {
  affine.for %k = 0 to 10 {
    affine.for %l = 0 to 10 {
      affine.if (d0) : (d0 - 1 >= 0, -d0 + 8 >= 0)(%k) {
        // Call foo except on the first and last iteration of %k
        "foo"(%k) : (index) -> ()
      }
    }
  }
  return
}
```

With custom parsing/printing: affine.for operations with an attached region feels like a regular for!

Extra semantics constraints in this dialect: the if condition is an affine relationship on the enclosing loop indices.

```
#set0 = (d0) : (d0 - 1 >= 0, -d0 + 8 >= 0)
func @test() {
  "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> () {
  ^bb1(%i0: index):
    "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> ()
{
    ^bb2(%i1: index):
      "affine.if"(%i0) {condition: #set0} : (index) -> () {
        "foo"(%i0) : (index) -> ()
        "affine.terminator"() : () -> ()
      } { // else block
      }
      "affine.terminator"() : () -> ()
    }
    ...
```

Same code without custom parsing/printing: closer to the internal in-memory representation.

Google

Affine Dialect — Reshape Example

```
// MemRef temporary for reshape output.
%b = alloc() : memref<16x4xf32>

// Reshape R1 to R2.
for %i0 = 0 to 64 {
  %1 = load %a[%i0] : memref<64xf32>
  %2 = affine.apply (d0) -> (d0 floordiv 4) (%i0)
  %3 = affine.apply (d0) -> (d0 mod 4) (%i0)
  store %1, %b[%2, %3] : memref<16x4xf32>
}

// Use output of R2 reshape.
for %i1 = 0 to 16 {
  for %i2 = 0 to 4 {
    %4 = load %b[%i1, %i2] : memref<16x4xf32>
    "op0"(%4) : (f32) -> ()
  }
}
```

1. Compute slice bounds: reshape indexing transformation

```
#map1 = (d0, d1) -> (d0 * 4 + d1)
#map3 = (d0, d1) -> ((d0 * 4 + d1) floordiv 4)
#map4 = (d0, d1) -> ((d0 * 4 + d1) mod 4)
// Fused loop nest.
%b = alloc() : memref<1x4xf32>
%c0 = constant() : 0
for %i0 = 0 to 16 {
  for %i1 = 0 to 4 {
    %1 = affine.apply #map1(%i0, %i1)
    %2 = load %a[%1] : memref<64xf32>
    %3 = affine.apply #map3(%i0, %i1)
    %4 = affine.apply #map4(%i0, %i1)
    store %2, %b[%3, %4] : memref<16x4xf32>
  }
}
```

Affine Dialect — Reshape Example

```
// MemRef temporary for reshape output.
%b = alloc() : memref<16x4xf32>

// Reshape R1 to R2.
for %i0 = 0 to 64 {
  %1 = load %a[%i0] : memref<64xf32>
  %2 = affine.apply (d0) -> (d0 floordiv 4) (%i0)
  %3 = affine.apply (d0) -> (d0 mod 4) (%i0)
  store %1, %b[%2, %3] : memref<16x4xf32>
}

// Use output of R2 reshape.
for %i1 = 0 to 16 {
  for %i2 = 0 to 4 {
    %4 = load %b[%i1, %i2] : memref<16x4xf32>
    "op0"(%4) : (f32) -> ()
  }
}
```

1. **Compute slice bounds:** reshape indexing transformation
2. **Compute fusion cost**: destination loop depth 2 is min cost
3. **Fuse**: MemRef temporary '%b' for reshape contracted to 1x4xf32

```
#map1 = (d0, d1) -> (d0 * 4 + d1)
#map3 = (d0, d1) -> ((d0 * 4 + d1) floordiv 4)
#map4 = (d0, d1) -> ((d0 * 4 + d1) mod 4)
// Fused loop nest.
%b = alloc() : memref<1x4xf32>
%c0 = constant() : 0
for %i0 = 0 to 16 {
  for %i1 = 0 to 4 {
    %1 = affine.apply #map1(%i0, %i1)
    %2 = load %a[%1] : memref<64xf32>
    %3 = affine.apply #map4(%i0, %i1)
    store %2, %b[%c0, %3] : memref<1x4xf32>
    %4 = affine.apply #map4(%i0, %i1)
    %6 = load %b[%c0, %4] : memref<1x4xf32>
    "op0"(%6) : (f32) -> ()
  }
}
```

Google

Affine Dialect — Reshape Example

```
// MemRef temporary for reshape output.
%b = alloc() : memref<16x4xf32>

// Reshape R1 to R2.
for %i0 = 0 to 64 {
  %1 = load %a[%i0] : memref<64xf32>
  %2 = affine.apply (d0) -> (d0 floordiv 4) (%i0)
  %3 = affine.apply (d0) -> (d0 mod 4) (%i0)
  store %1, %b[%2, %3] : memref<16x4xf32>
}

// Use output of R2 reshape.
for %i1 = 0 to 16 {
  for %i2 = 0 to 4 {
    %4 = load %b[%i1, %i2] : memref<16x4xf32>
    "op0"(%4) : (f32) -> ()
  }
}
```

1. **Compute slice bounds:** reshape indexing transformation
2. **Compute fusion cost**: destination loop depth 2 is min cost
3. **Fuse**: MemRef temporary ‘%b’ for reshape contracted to 1x4xf32
4. **Load-Store Forwarding/Simplify**: MemRef temporary for reshape eliminated

```
// Fused loop nest.
for %i0 = 0 to 16 {
  for %i1 = 0 to 4 {
    %0 = affine.apply (d0, d1) -> (d0 * 4 + d1) (%i0, %i1)
    %1 = load %a[%0] : memref<64xf32>
    "op0"(%1) : (f32) -> ()
  }
}
```

Example: Linalg Dialect

for the Composition and Structural Decomposition of Linear Algebra Operations

from Nicolas Vasilache

Linalg Rationale

Propose a multi-purpose code generation path

- For mixing different styles of compiler transformations
  - Combinators (tile, fuse, communication generation on high level operations)
  - Loop-based (dependence analysis, fuse, vectorize, pipeline, unroll-and-jam)
  - SSA (use-def)
- That **does not require heroic analyses** and transformations
  - Declarative properties enable transformations w/o complex analyses
  - If/when good analyses exist, we can use them — ***More at LCPC on Oct 24***
- Beyond **black-box** numerical libraries
  - **Compiling loops + native library calls or hardware blocks**
  - Can evolve **beyond affine** loops and data
  - **Locking-in performance gains from good library implementations is a must**
  - **Optimize across loops and library calls for locality and customization**

Linalg Type System And Type Building Ops

- RangeType: RangeOp create a (min, max, step)-triple of `index` (intptr_t)

```
%0 = linalg.range %c0:%arg1:%c1 : !linalg.range
```

intptr_t intptr_t intptr_t

- Used for stepping over
  - loop iterations (loop bounds)
  - data structures

Linalg Type System And Type Building Ops

- ViewType: ViewOp creates an n-d *"indexing"* over a `MemRefType`

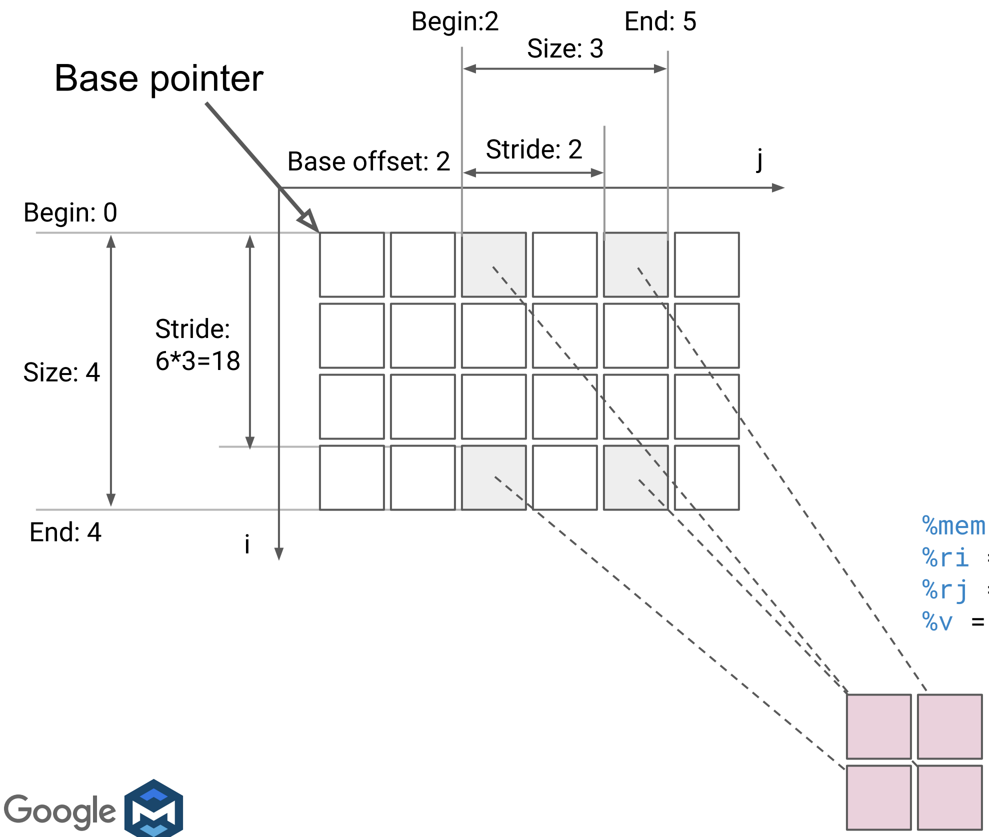
range range 2-D view

```
%8 = linalg.view %7[%r0, %r1] : !linalg.view<?x?xf32>
```

```
%9 = linalg.view %7[%r0, %row] : !linalg.view<?xf32>
```

range intptr_t 1-D view

View Type Descriptor in LLVM IR

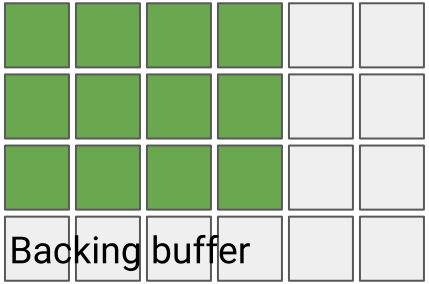


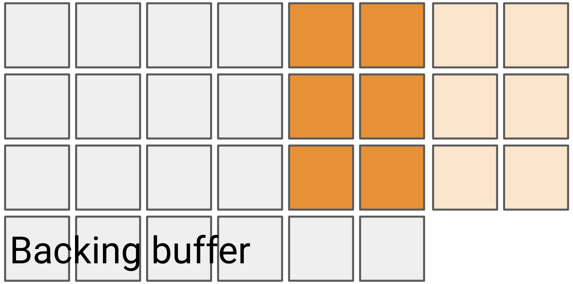

```
{ float*,   # base pointer
  i64,      # base offset
  i64[2]    # sizes
  i64[2] } # strides
```

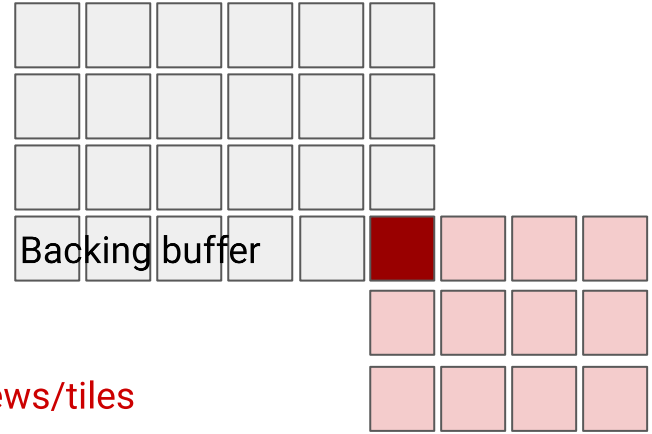
```
%memref = alloc() : memref<4x6 x f32>
%ri = linalg.range %c2:%c5:%c2 : !linalg.range
%rj = linalg.range %c0:%c4:%c3 : !linalg.range
%v = linalg.view %memref[%ri, %rj] : !linalg.view<?x?xf32>
```

Linalg View

- Simplifying assumptions for analyses and IR construction
  - E.g. non-overlapping rectangular memory regions (symbolic shapes)
  - Data abstraction encodes boundary conditions







Same library call, data structure adapts to full/partial views/tiles

matmul(vA, vB, vC)

Defining Matmul

- `linalg.matmul` operates on `view<?x?xf32>, view<?x?xf32>, view<?x?xf32>`

```
func @call_linalg_matmul(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>){
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  %rM = linalg.range %c0:%M:%c1 : !linalg.range
  %rN = linalg.range %c0:%N:%c1 : !linalg.range
  %rK = linalg.range %c0:%K:%c1 : !linalg.range
  %4 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  %6 = linalg.view %B[%rK, %rN] : !linalg.view<?x?xf32>
  %8 = linalg.view %C[%rM, %rN] : !linalg.view<?x?xf32>
  linalg.matmul(%4, %6, %8) : !linalg.view<?x?xf32>
  return
}
```

Lowering Between Linalg Ops: Matmul to Matvec

```
func @matmul_as_matvec(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  %rM = linalg.range %c0:%M:%c1 : !linalg.range
  %rK = linalg.range %c0:%N:%c1 : !linalg.range
  %5 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  affine.for %col = 0 to %N {
    %7 = linalg.view %B[%rK, %col] : !linalg.view<?xf32>
    %8 = linalg.view %C[%rM, %col] : !linalg.view<?xf32>
    linalg.matvec(%5, %7, %8) : !linalg.view<?xf32>
  }
  return
}
```

“Interchange” due to library impedance mismatch

Google

Lowering Between Linalg Ops: Matmul to Matvec

```
// In some notional index notation, we have defined:
//   Matmul as: C(i, j) = scalarC + A(i, r_k) * B(r_k, j)
//   Matvec as:    C(i) = scalarC + A(i, r_j) * B(r_j)
// So we must drop the `j` loop from the Matmul.
// Parallel dimensions permute: do it declaratively.
void linalg::MatmulOp::emitFinerGrainForm()
  auto *op = getOperation();
  ScopedContext scope(FuncBuilder(op), op->getLoc());
  IndexHandle j;
  auto *vA(getInputView(0)), *vB(...), *vC(...);
  Value *range = getViewRootIndexing(vB, 1).first;
  linalg::common::LoopNestRangeBuilder(&j, range)({
      matvec(vA, slice(vB, j, 1), slice(vC, j, 1)),
  });
}
```

Extracting/analyzing this information from transformed and tiled loops would take a lot of effort

With high-level dialects the problem goes away

Loop Tiling

tileSizes = {8, 9}

```
%c0 = constant 0 : index
%c1 = constant 1 : index
%M = dim %A, 0 : memref<?x?xf32>
%N = dim %C, 1 : memref<?x?xf32>
%K = dim %A, 1 : memref<?x?xf32>
%rM = linalg.range %c0:%M:%c1 :
%rN = linalg.range %c0:%N:%c1 :
%rK = linalg.range %c0:%K:%c1 :
%4 = linalg.view %A[%rM, %rK] :
%6 = linalg.view %B[%rK, %rN] :
%8 = linalg.view %C[%rM, %rN] :
linalg.matmul(%4, %6, %8) :
```

```
func @matmul_tiled_loops(%arg0: memref<?x?xf32>,
      %arg1: memref<?x?xf32>, %arg2: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %cst = constant 0.000000e+00 : f32
  %M = dim %arg0, 0 : memref<?x?xf32>
  %N = dim %arg2, 1 : memref<?x?xf32>
  %K = dim %arg0, 1 : memref<?x?xf32>
  affine.for %i0 = 0 to %M step 8 {
    affine.for %i1 = 0 to %N step 9 {
      affine.for %i2 = 0 to %K {
        affine.for %i3 = max(%i0, %c0) to min(%i0 + 8, %M) {
          affine.for %i4 = max(%i1, %c0) to min(%i1 + 9, %N) {
            %3 = cmpi "eq", %i2, %c0 : index
            %6 = load %arg2[%i3, %i4] : memref<?x?xf32>
            %7 = select %3, %cst, %6 : f32
            %9 = load %arg1[%i2, %i4] : memref<?x?xf32>
            %10 = load %arg0[%i3, %i2] : memref<?x?xf32>
            %11 = mulf %10, %9 : f32
            %12 = addf %7, %11 : f32
            store %12, %arg2[%i3, %i4] : memref<?x?xf32>
```

Boundary conditions

Google

Loop Tiling Declaration

- An op "*declares*" how to tile itself maximally on loops
  - For LinalgBase this is easy: perfect loop nests
  - Can be tiled declaratively with **mlir::tile**

```
void linalg::lowerToTiledLoops(mlir::Function *f,
                               ArrayRef<uint64_t> tileSizes) {
  f->walk([tileSizes](Operation *op) {
    if (emitTiledLoops(op, tileSizes).hasValue())
      op->erase();
  });
}
```

Works with imperfectly nested + interchange

```
llvm::Optional<SmallVector<mlir::AffineForOp, 8>>
linalg::emitTiledLoops(Operation *op, ArrayRef<uint64_t> tileSizes) {
  auto loops = emitLoops(op);
  if (loops.hasValue())
    return mlir::tile(*loops, tileSizes, loops->back());
  return llvm::None;
}
```

Google

View Tiling

```
func @matmul_tiled_views(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  affine.for %i0 = 0 to %M step 8 {
    affine.for %i1 = 0 to %N step 9 {
      %4 = affine.apply (d0) -> (d0 + 8)(%i0)
      %5 = linalg.range %i0:%4:%c1 : !linalg.range   needs range intersection
      %7 = linalg.range %c0:%K:%c1 : !linalg.range
      %8 = linalg.view %A[%5, %7] : !linalg.view<?x?xf32>
      %10 = linalg.range %c0:%M:%c1 : !linalg.range
      %12 = affine.apply (d0) -> (d0 + 9)(%i1)
      %13 = linalg.range %i1:%12:%c1 : !linalg.range   needs range intersection
      %14 = linalg.view %B[%10, %13] : !linalg.view<?x?xf32>
      %15 = linalg.view %C[%5, %13] : !linalg.view<?x?xf32>
      linalg.matmul(%8, %14, %15) : !linalg.view<?x?xf32>
```

Recursive linalg.**matmul** call

Google

View Tiling Declaration

- A LinalgOp *“declares”* how to tile itself with views
    - Step 1: *“declare”* mapping from loop to views
    - Step 2: tile loops by *tileSizes*
    - Step 3: apply *mapping* on tiled loops to get tiled views (i.e. sub-views)
    - Step 4: rewrite as tiled loops over sub-views

Tile and Fuse 2mm

- 3-D tiling + fusion
  two `linalg.matmul`

```
%c0 = constant 0 : index
%c1 = constant 1 : index
%M = dim %A, 0 : memref<?x?xf32>
%N = dim %C, 1 : memref<?x?xf32>
%K = dim %A, 1 : memref<?x?xf32>
%O = dim %E, 1 : memref<?x?xf32>
%rM = linalg.range %c0:%M:%c1 :
%rN = linalg.range %c0:%N:%c1 :
%rK = linalg.range %c0:%K:%c1 :
%4 = linalg.view %A[%rM, %rK] :
%6 = linalg.view %B[%rK, %rN] :
%8 = linalg.view %C[%rM, %rN] :
%9 = linalg.view %D[%rN, %rO] :
%10 = linalg.view %E[%rM, %rO] :
linalg.matmul(%4, %6, %8) :
linalg.matmul(%8, %9, %10) :
```

tileSizes = {7, 8, 9}

```
%M = dim %A, 0 : memref<?x?xf32>
%N = dim %C, 1 : memref<?x?xf32>
%K = dim %A, 1 : memref<?x?xf32>
%O = dim %E, 1 : memref<?x?xf32>
affine.for %i0 = 0 to %M step 7 {
  affine.for %i1 = 0 to %O step 8 {
    affine.for %i2 = 0 to %N step 9 {
      %5 = affine.apply (d0) -> (d0 + 7)(%i0)
      %6 = linalg.range %i0:%5:%c1 : !linalg.range
      %8 = affine.apply (d0) -> (d0 + 9)(%i2)
      %9 = linalg.range %i2:%8:%c1 : !linalg.range
      %10 = linalg.view %arg2[%6, %9] : !linalg.view<?x?xf32>
      %12 = affine.apply (d0) -> (d0 + 8)(%i1)
      %13 = linalg.range %i1:%12:%c1 : !linalg.range
      %14 = linalg.view %arg3[%9, %13] : !linalg.view<?x?xf32>
      %15 = linalg.view %arg4[%6, %13] : !linalg.view<?x?xf32>
      %17 = linalg.range %c0:%K:%c1 : !linalg.range
      %18 = linalg.view %arg0[%6, %17] : !linalg.view<?x?xf32>
      %20 = linalg.range %c0:%N:%c1 : !linalg.range
      %21 = linalg.view %arg1[%17, %20] : !linalg.view<?x?xf32>
      %22 = linalg.view %arg2[%6, %20] : !linalg.view<?x?xf32>
      linalg.matmul(%18, %21, %22) : !linalg.view<?x?xf32>
      linalg.matmul(%10, %14, %15) : !linalg.view<?x?xf32>
```

Tile and Fuse — Forward-Substitution/Inlining/Halide Style

- A LinalgOp *“declares”* how to tile itself with views

Loop and View Tiling

  - Step 1: *“declare”* mapping from loop to views
  - Step 2: tile loops by *tileSizes*
  - Step 3: apply *mapping* on tiled loops to get tiled views (i.e. sub-views)
  - Step 4: rewrite as tiled loops over sub-views
  - Step 5: follow SSA use-def chain to find producers of inputs
  - Step 6: clone producer of sub-view in local scope
  - Step 7: cleanup

Promotion to Local Memory

tileSizes = {3, 4, 5}

```
%c0 = constant 0 : index
%c1 = constant 1 : index
%M = dim %A, 0 : memref<?x?xf32>
%N = dim %C, 1 : memref<?x?xf32>
%K = dim %A, 1 : memref<?x?xf32>
%rM = linalg.range %c0:%M:%c1 :
%rN = linalg.range %c0:%N:%c1 :
%rK = linalg.range %c0:%K:%c1 :
%4 = linalg.view %A[%rM, %rK] :
%6 = linalg.view %B[%rK, %rN] :
%8 = linalg.view %C[%rM, %rN] :
linalg.matmul(%4, %6, %8) :
```

```
%3 = dim %0, 0 : memref<?x?xf32>
%4 = dim %0, 1 : memref<?x?xf32>
%5 = dim %1, 1 : memref<?x?xf32>
affine.for %i0 = 0 to %3 step 3 {
  affine.for %i1 = 0 to %5 step 4 {
    affine.for %i2 = 0 to %4 step 5 {
      %8 = linalg.range %i0:%i0+3:1 : !linalg.range
      %11 = linalg.range %i2:%i2+5:1 : !linalg.range
      %12 = linalg.view %0[%8, %11] : !linalg.view<?x?xf32>
      %15 = linalg.range %i1:%i1+4:1 : !linalg.range
      %16 = linalg.view %1[%11, %15] : !linalg.view<?x?xf32>
      %17 = linalg.view %2[%8, %15] : !linalg.view<?x?xf32>
      %18 = linalg.copy_transpose_reshape %12 : memref<?x?xf32, 1>
      %23 = linalg.view %18[..., ...] : !linalg.view<?x?xf32>
      %24 = linalg.copy_transpose_reshape %16 : memref<?x?xf32, 1>
      %29 = linalg.view %24[..., ...] : !linalg.view<?x?xf32>
      %30 = linalg.copy_transpose_reshape %17 : memref<?x?xf32, 1>
      %31 = dim %30, 0 : memref<?x?xf32, 1>
      %32 = dim %30, 1 : memref<?x?xf32, 1>
      %33 = linalg.range %c0:%31:%c1 : !linalg.range
      %34 = linalg.range %c0:%32:%c1 : !linalg.range
      %35 = linalg.view %30[%33, %34] : !linalg.view<?x?xf32>
      linalg.matmul (%23, %29, %35) : !linalg.view<?x?xf32>
      linalg.reshape_transpose_copy %30, %17 : memref<?x?xf32, 1>
      dealloc %18 : memref<?x?xf32, 1>
      dealloc %24 : memref<?x?xf32, 1>
      dealloc %30 : memref<?x?xf32, 1>
```

Google

MLIR for Loop Nests, Parallelism, Acceleration: Recap

MLIR is a great infrastructure for higher-level compilation

- Gradual and partial lowering, mixing dialects
- Reduce impedance mismatch at each level

MLIR provides all the infrastructure to build dialects and transformations

- At each level it is the same LLVM-style infrastructure

Check out github, mailing list, stay tuned for further announcements

Workshops: LCPC MLIR4HPC HiPEAC AccML CGO C4ML

Google