Vectorization in MLIR

Towards Scalable Vectors and Matrices

Andrzej Warzyński, Diego Caballero & Nicolas Vasilache

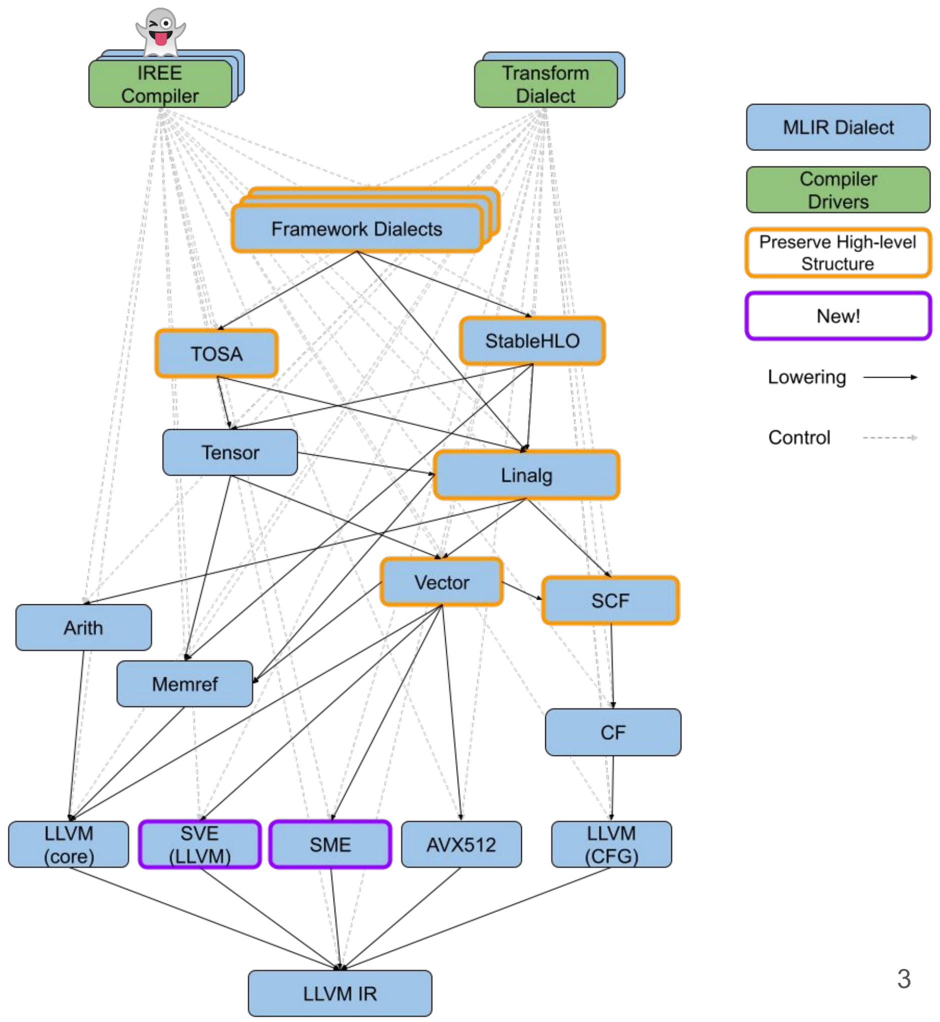
Linalg + Vector

Representation in MLIR

Our Compiler Tools

- We are using MLIR, obviously :)
    - Today's focus: Linalg, Vector, SVE and SME

- Compiler Drivers
    - **IREE ML Compiler:** end-to-end pipeline-based ML compiler
    - **Transform Dialect:** IR to describe optimization strategies and how to apply them



Linalg Dialect

- Introduces structured ops at tensor and memref levels
- We use Linalg on tensors (vs memrefs)

Generic Linalg Op

```
%31 = linalg.generic {
  indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>,
                   affine_map<(d0, d1) -> (d0, 0)>],
  iterator_types = ["parallel", "reduction"]}
  ins(%a : tensor<64x32xf32>)
  outs(%b : tensor<64xf32>)
  {
    ^bb0(%in: f32, %out: f32):
      %0 = arith.addf %in, %in : f32
      linalg.yield %0 : f32
  } -> tensor<64xf32>
```

High-level op structure traits

Named Ops

```
%c_out = linalg.matmul
  ins(%a, %b : tensor<?x?xf32>, tensor<?x?xf32>)
  outs(%c_in : tensor<?x?xf32>) -> tensor<?x?xf32>


%conv = linalg.conv_2d_nhwc_fhwc {
  dilations = dense<1> : tensor<2xi64>,
  strides = dense<1> : tensor<2xi64> }
  ins (%input, %filter:
    tensor<...xf32>, tensor<...xf32>)
  outs (%init: tensor<...xf32>) -> tensor<...xf32>
```

Vector Dialect

- Multi-dimensional vectors from the get-go
  - Matrices and higher-rank vectors, e.g.

```
vector<16x16xf32>
```

```
vector<4x16x8x32xf32>
```

  - Proper modeling of multi-dim operations like transposes

- Two levels of abstraction:

High level: focus on structure and more abstract comp.

```
%c_out = vector.contract {
  indexing_maps = [ affine_map<...],
  iterator_types = ["parallel", "parallel", "reduction"],
  kind = #vector.kind<add>
  %a_matrix, %b_matrix, %c_in_block ...

%vx = vector.transfer_read %tensor[%idx0, %idx1],
  { permutation_map = affine_map<(d0, d1)->(d1, d0)>}
  : tensor<?x?xf32>, vector<3x7xf32>

%0 = vector.mask %mask, %passthru {
    arith.divsi %b, %c : vector<16xf32>
} : vector<16xi1>, vector<16xf32 -> vector<16xf32>
```

Low level: ops closer to HW

```
%vfma = vector.fma %a, %b, %c : vector<8x4xf32>

%vr = vector.reduction <minf>, %arg0 : vector<16xf32> into f32

%vs = vector.shuffle %2, %b[0, 6]: vector<3xf32>, vector<4xf32>

%ve = vector.extract %arg5[7] : vector<8x32xf32>
```

Linalg Vectorization Approach

Progressive Vectorization

Linalg Vectorization: High-level Overview

Four progressive vectorization steps:

1. **Vector-level Tiling**
   - Tile the Linalg operation with vector sizes
   - Create the vector loop nest structure
   - Apply padding or peeling (remainder loop), if necessary

2. **“Loop Body” Vectorization (Linalg Vectorizer)**
   - Leverage Linalg op structure to generate vector code
   - Generic vectorization path and specialized vectorization paths (e.g., convolutions)

3. **High-level to Low-level Vector Lowering**
   - Refine vector ops with canonicalization patterns and HW information in mind

4. **Vector Dimensionality (Rank) Legalization**
   - Unroll n-D vector (vector ops to SCF loops) to the vector dimensionality supported by the target

Example: linalg.matmul

- Input IR
  - Matmul on tensors with dynamic shapes

```
func.func @matmul(
      %a: tensor<?x?xf32>,
      %b: tensor<?x?xf32>,
      %c: tensor<?x?xf32>) -> tensor<?x?xf32> {

      %c_out = linalg.matmul ins(%a, %b: tensor<?x?xf32>, tensor<?x?xf32>)
                             outs(%c_in: tensor<?x?xf32>) -> tensor<?x?xf32>

      return %c_out : tensor<?x?xf32>
}
```

Step 1: Vector-level Tiling

Tile `linalg.matmul` by vector sizes <8, 32, 1>

Extract tensor slices for the tile

```
scf.for %n = %c0 to %N step %c8 … {
  scf.for %m = %c0 to %M step %c32 … {
    scf.for %k = %c0 to %K step %c1 … {
      %a_tile = tensor.extract_slice %a … : tensor<?x?xf32> to tensor<?x1xf32>
      %b_tile = tensor.extract_slice %b … : tensor<?x?xf32> to tensor<1x?xf32>
      %c_in_tile = tensor.extract_slice %c_in … : tensor<?x?xf32> to tensor<?x?xf32>
      %c_out = linalg.matmul ins(%a_tile, %b_tile : tensor<?x1xf32>, tensor<1x?xf32>)
                          outs(%c_in_tile : tensor<?x?xf32>) -> tensor<?x?xf32>
    }
  }
}
```

No vector ops just yet!

Step 2: Loop Body Vectorization

```
scf.for %n = %c0 to %N step %c8 ... {
  scf.for %m = %c0 to %M step %c32 ... {
    scf.for %k = %c0 to %K step %c1 ... {
      %a_mask = vector.create_mask ... : vector<8x1xi1>
      %b_mask = vector.create_mask ... : vector<1x32xi1>
      %c_in_mask = vector.create_mask ... : vector<8x32xi1>
      %contract_mask = vector.create_mask ... : vector<8x32x1xi1>

      %a_in = vector.transfer_read %a_tile ... %a_mask : tensor<?x1xf32>, vector<8x1xf32>
      %b_in = vector.transfer_read %b_tile ... %b_mask : tensor<1x?xf32>, vector<1x32xf32>
      %c_in = vector.transfer_read %c_in_tile ... %c_in_mask : tensor<?x?xf32>, vector<8x32xf32>

      %c_out = vector.mask %contract_mask { vector.contract {
        indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d2)>,
                         affine_map<(d0, d1, d2) -> (d2, d1)>,
                         affine_map<(d0, d1, d2) -> (d0, d1)>],
        iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>}
        %a_in, %b_in, %c_in : vector<8x1xf32>, vector<1x32xf32> into vector<8x32xf32>
        } : vector<8x32x1xi1> -> vector<8x32xf32>
```

2-D vector loads inferred from indexing maps

'`vector.contract`' preserves part of original op structure

Step 3: High-level to Low-level Vector Lowering

- vector.contract → vector.outerproduct

```
%transposed_a = vector.transpose %45, [1, 0] : vector<8x1xf32> to vector<1x8xf32>
%vec_a = vector.extract %transposed_a[0] : vector<1x8xf32>
%vec_b = vector.extract %46[0] : vector<1x32xf32>
%vec_c_out = vector.mask %op_mask {
  vector.outerproduct %vec_a, %vec_b, %vec_c_in {kind = #vector.kind<add>}
    : vector<8xf32>, vector<32xf32>
} : vector<8x32xi1> -> vector<8x32xf32>
```

- vector.outerproduct → vector.fma

'vector.mask' lowering

```
%vec_a0 = vector.broadcast %110 : f32 to vector<32xf32>
%vec_c_in0 = vector.extract %vec_c[0] : vector<8x32xf32>
%fma0 = vector.fma %vec_a0, %vec_b0, %vec_c_in0 : vector<32xf32>
%vec_c_out0 = arith.select %fma_mask, %fma0, %vec_c_in0 : vector<32xi1>, vector<32xf32>
```

8x unrolled

Step 4: Vector Dimensionality Legalization

- 2-D vector loads/stores → 1-D vector loads/stores

```
%vec_c_in00 = vector.maskedload %c_in_tile[%c0, %c0], ... : vector<32xf32>
%vec_c_in01 = vector.maskedload %c_in_tile[%c1, %c0], ... : vector<32xf32>
%vec_c_in02 = vector.maskedload %c_in_tile[%c2, %c0], ... : vector<32xf32>
...
%vec_c_in08 = vector.maskedload %c_in_tile[%c7, %c0], ... : vector<32xf32>
```

Compiler Drivers for Linalg Vectorizer

Option1: IREE - end-to-end MLIR based compiler (link)

```
$ iree-compile --iree-hal-target-backends=llvm-cpu \
      --iree-llvmcpu-target-triple=”aarch64-none-linux-gnu" \
      --iree-llvmcpu-target-cpu-features="+sve" example.mlir
```

Option 2: Transform Dialect - MLIR dialect that can drive transformations (link)

```
$ mlir-opt matmul.mlir --test-transform-dialect-interpreter='transform-file-name=sequence.mlir'
```

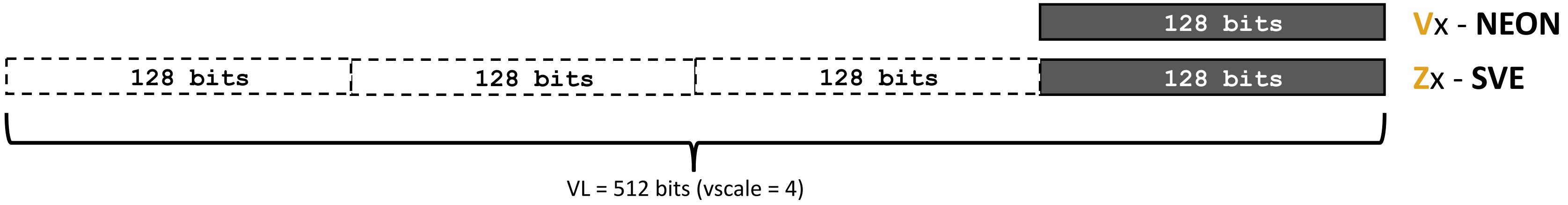
```
transform.sequence failures(propagate) {
  ^bb0(%arg1: !transform.any_op):
    %matmul_0 = transform.structured.match ops{["linalg.matmul"]} in %arg1: ...
    %tiled_matmul, %loops:3 = transform.structured.tile_using_for %matmul_0 [8, 32, 1] : ...
    transform.structured.vectorize %tiled_matmul vector_sizes [8, 32, 1] : ...
}
```

Scalable Vectors and Matrices

SVE + SME Case Study

Scalable Vector Extension (SVE)

- 32 scalable vector registers (`Z0-Z31`):
  - **128-2048** bits vector length is decided by implementation
  - Always **vscale** * **base size** (128 bits)



- ISA designed for **Vector Length Agnostic** (VLA) programming:
  - **VL** (vector length) is unknown at compile-time, but **known at run-time**
  - 16 scalable predicate registers to facilitate this (`p0-p15`)

```
void foo(int *out, int *a, int *b, int N)
{
    for (int i=0; i<N; i++)
        out[i] = a[i] + b[i];
}
```

clang -O2
-march=armv8a+sve

```
.L_loopStart:
ld1w z1.s, p1/Z, [x1, x9, LSL #2]
ld1w z2.s, p1/Z, [x2, x9, LSL #2]
add z1.s, p1/M, z1.s, z2.s
st1w z1.s, p1, [x0, x9, LSL #2]
incw x9
whilelt p1.s, x9, x3
b.first .L_loopStart
```

Increment by **vscale!**

Scalable Vectors in LLVM and MLIR (vscale)

- The value of **vscale** is not known at compile time.
  - Use `llvm.vscale` (LLVM IR) or `vector.vscale` (MLIR) to get an SSA representation.
- **LLVM**

```
define float @add_f32(<vscale x 8 x float> %a, <vscale x 4 x float> %b) {
     %r1 = call @llvm.vector.reduce.fadd.f32.nxv8f32(float -0.0, <vscale x 8 x float> %a)
     %r2 = call @llvm.vector.reduce.fadd.f32.nxv4f32(float -0.0, <vscale x 4 x float> %b)
     %r = fadd %r1, %r2
     ret float %r
}
```

- **MLIR**

```
llvm.func @vector_splat_1d_scalable() -> vector<[4]xf32> {
     %0 = llvm.mlir.constant(dense<0.000000e+00> : vector<[4]xf32>) : vector<[4]xf32>
     llvm.return %0 : vector<[4]xf32>
}
```

Step 1: “Scalable” Vector-level Tiling

```
scf.for %n = %c0 to %N step %c8 … {
  %vscale = vector.vscale
  %step_vs = arith.muli %vscale, %c32 : index
  scf.for %m = %c0 to %M step %step_vs … {
    scf.for %k = %c0 to %K step %c1 … {
      %a_tile = tensor.extract_slice %a … : tensor<?x?xf32> to tensor<?x1xf32>
      %b_tile = tensor.extract_slice %b … : tensor<?x?xf32> to tensor<1x?xf32>
      %c_in_tile = tensor.extract_slice %c_in … : tensor<?x?xf32> to tensor<?x?xf32>
      %c_out = linalg.matmul ins(%a_tile, %b_tile : tensor<?x1xf32>, tensor<1x?xf32>)
                             outs(%c_in_tile : tensor<?x?xf32>) -> tensor<?x?xf32>
    }
  }
}
```

New! Scalable step computation.

Updated loop step.

Step 2: “Scalable” Loop Body Vectorization

```
scf.for %n = %c0 to %N step %c8 ... {
  scf.for %m = %c0 to %M step %step_vs ... {
    scf.for %k = %c0 to %K step %c1 ... {
      %a_mask = vector.create_mask ... : vector<8x1xi1>
      %b_mask = vector.create_mask ... : vector<1x[32]xi1>
      %c_in_mask = vector.create_mask ... : vector<8x[32]xi1>
      %contract_mask = vector.create_mask ... : vector<8x[32]x1xi1>

      %a_in = vector.transfer_read %a_tile ... %a_mask : tensor<?x1xf32>, vector<8x1xf32>
      %b_in = vector.transfer_read %b_tile ... %b_mask : tensor<1x?xf32>, vector<1x[32]xf32>
      %c_in = vector.transfer_read %c_in_tile ... %c_mask : tensor<?x?xf32>, vector<8x[32]xf32>

      %c_out = vector.mask %contract_mask { vector.contract {
        indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d2)>,
                         affine_map<(d0, d1, d2) -> (d2, d1)>,
                         affine_map<(d0, d1, d2) -> (d0, d1)>],
        iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>}
        %a_in, %b_in, %c_in : vector<8x1xf32>, vector<1x[32]xf32> into vector<8x[32]xf32>
        } : vector<8x[32]x1xi1> -> vector<8x[32]xf32>
    }
  }
}
```

New! Scalable vectors.

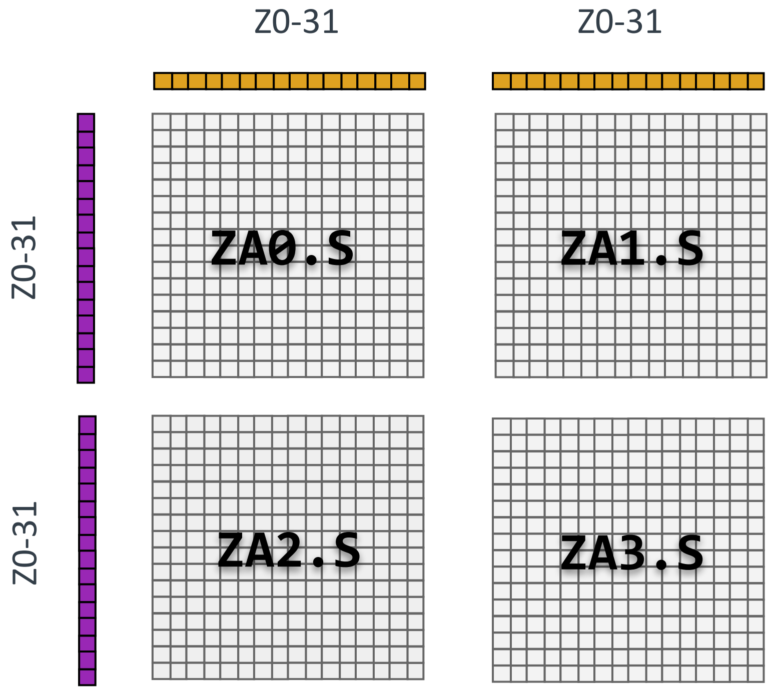
Scalable Matrix Extension

- **New!** Scalable 2D **ZA accumulator**
  - Horizontal & vertical "slice" access.
  - Contains virtual tiles.
  - Number of tiles depends on element type, e.g. :

Type	# tiles	ZA virtual tile dims	Tile names	Z reg dims
i16	2	(8***vscale**) x (8***vscale**)	**ZA0-Z1**.h	8***vscale**
i32/f32	4	(4***vscale**) x (4***vscale**)	**ZA0-ZA3**.s	4***vscale**

- **New!** Outer product instructions:

```
        (s)ingle precision        (m)erging predicate
FMOPA  za0.s,  p0/m,  p1/m,  z1.s,  z3.s
         Accumulator (2D)            lhs (1D)  rhs (1D)
```



32-bit element tiles

SVL = 512 bits

4 virtual tiles (16 x 16)

SME Dialect

- **"Regular" Ops**

```
arm_sme.move_vector_to_tile_slice
arm_sme.store_tile_slice
arm_sme.load_tile_slice
arm_sme.tile_load
arm_sme.tile_store
arm_sme.zero
```

"Virtual" Ops

```
arm_sme.cast_tile_to_vector
arm_sme.cast_vector_to_tile
arm_sme.get_tile_id
```

LLVM Intrinsic Ops

```
arm_sme.intr.ld1b.horiz
arm_sme.intr.ld1b.vert
arm_sme.intr.ld1d.horiz
arm_sme.intr.ld1d.vert
```

- **Function attributes**
  - **`arm_streaming`** - enable **Streaming** SVE (SSVE) mode inside function
  - **`arm_za`** - enable access to SME **ZA** storage inside function

```
func.func @example() attributes {arm_streaming, arm_za} {
      %two = arith.constant dense<2> : vector<[16]x[16]xi8>
}
```

- **Passes:**
  - "allocate-arm-sme-tiles" or "enable-arm-streaming"

Steps 2+3: Tiling + Vectorization for SME

```
%vscale = vector.vscale
%step_vs = arith.muli %vscale, %c4 : index
scf.for %m = %c0 to %N step %step_vs ... {
  scf.for %n = %c0 to %M step %step_vs ... {
    scf.for %k = %c0 to %K step %c1 … {
      %a_mask = vector.create_mask ... : vector<[4]x1xi1>
      %b_mask = vector.create_mask ... : vector<1x[4]xi1>
      %op_mask = vector.create_mask ... : vector<[4]x[4]x1xi1>

      %a_col = vector.transfer_read %a_tile ... %a_mask : tensor<?x1xf32>, vector<[4]x1xf32>
      %b_row = vector.transfer_read %b_tile ... %b_mask : tensor<1x?xf32>, vector<1x[4]xf32>
      %c_in = vector.transfer_read %c_in_tile ... %c_in_mask : tensor<?x?xf32>, vector<[4]x[4]xf32>

      %c_out = vector.mask %op_mask {
           vector.outerproduct %a_col, %b_row, %c_in {kind = #vector.kind<add>} :
                 vector<[4]xf32>, vector<[4]xf32>
      } : vector<[4]x[4]xi1> -> vector<[4]x[4]xf32>
    }
  }
}
```

Both "parallel" loops are scalable

2D scalable "tiles" (dimensions guided by SME virtual tile dims)

Outer-product is NOT lowered to vector.fma!

Step 5: Lowering to the SME dialect

Work in progress!

```
%vscale = vector.vscale
%step_vs = arith.muli %vscale, %c4 : index

scf.for %m = %c0 to %N step %step_vs ... {
  scf.for %n = %c0 to %M step %step_vs ... {
    scf.for %k = %c0 to %K step %c1 … {

       %c_in = arm_sme.tile_load %c_in_tile[%c0, %c0] : … vector<[4]x[4]xf32>
       %tile_id = arm_sme.cast_vector_to_tile %c_in : vector<[4]x[4]xf32> to i32

       "arm_sme.intr.mopa"(%tile_id, %a_mask, %b_mask, %a_col, %b_row) :
          (i32, vector<[4]xi1>, vector<[4]xi1>, vector<[4]xf32>, vector<[4]xf32>) -> ()
    }
  }     Accumulator  lhs pred        rhs pred         lhs          rhs
}
```

Load into an SME tile.

“Virtual” Op encapsulating tile allocation.

Takeways

And all the things that we didn’t cover

Anything else?

- What have you missed?
  - Other **vectorization strategies** e.g. padding, peeling, packing, no masking.
  - Deeper dive into **IREE** and the **Transform Dialect** - very powerful MLIR drivers!
- Why Linalg+Vector?
  - Linalg preserves high-level information, which allows …
    - Progressive, very finely tuned vectorization!
    - Modularity of the vectoriser.
  - Multidimensional vectors by default, e.g matrices.

References

References

- Structured Code Generation in MLIR
    - Structured Transformations With Vectors (link)
    - Codegen Dialects Overview (link)
    - Controllable Transformations in MLIR (link)
    - Structured Codegen With Tensors (link)
    - The Anatomy of a Linalg Op (link)
- SVE + SME
    - SVE Programming Examples - Learn The Architecture (link)
    - The Scalable Matrix Extension (link)