



arm

Vectorization in MLIR

Towards scalable vectors and matrices (part 2)

Andrzej Warzyński

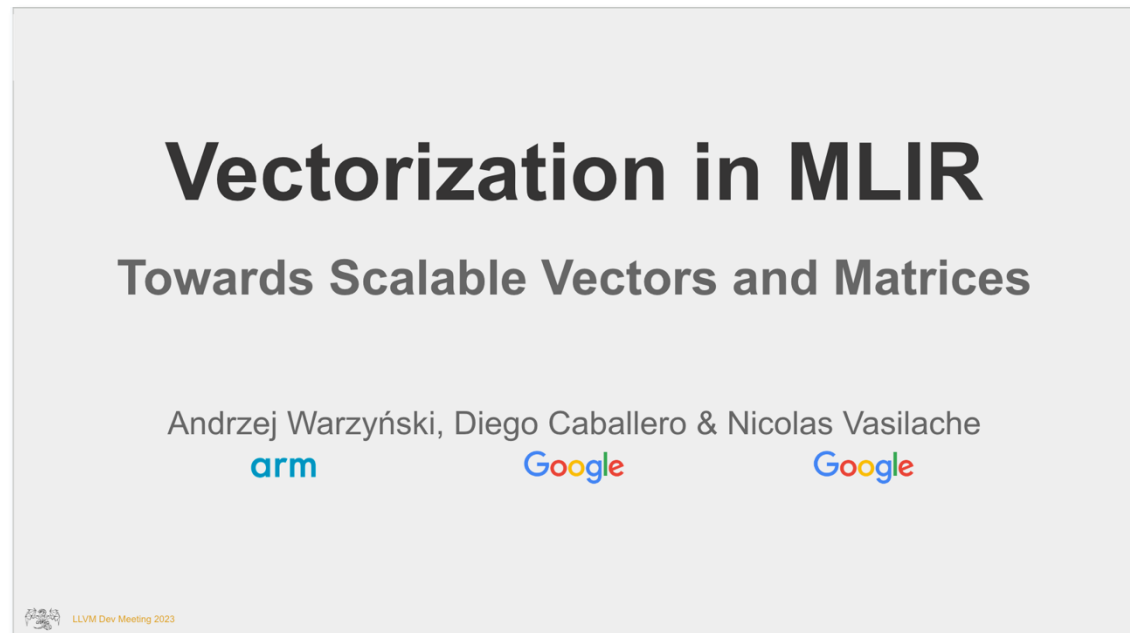
7/10/24

Agenda

What's new?


Part 1 (last year)

- + Presentation from LLVM Dev '23 ([link](#))
 - Higher-level overview
 - Focused on linalg.matmul
 - SVE + SME support - WIP



Vectorization in MLIR
Towards Scalable Vectors and Matrices

Andrzej Warzyński, Diego Caballero & Nicolas Vasilache

LLVM Dev Meeting 2023

Part 2 (this year)

- + **Finer details**
 - Deeper dive into Linalg Vectorizer
 - What works well, what doesn't
 - Ops other than linalg.matmul
- + **Updates on Scalable Vectors and Matrices**
 - Key challenges and new additions
- + **Perf numbers**
 - Vectorization for SVE + SME

All code examples have been trimmed to fit on the slides!
(use the links provided for working code)

Example: linalg.matmul

+ Input IR

- Linalg matmul on tensors, static or dynamic

```
func.func @matmul(  
    %A: tensor<1920x135xf32>,  
    %B: tensor<135x1080xf32>,  
    %C: tensor<1920x1080xf32>) -> tensor<1920x1080xf32>  
  
    %C_out = linalg.matmul ins(%A, %B: tensor<1920x135xf32>, tensor<135x1080xf32>)  
                outs(%C: tensor<1920x1080xf32>)  
    -> tensor<1920x1080xf32>  
  
    return %C_out : tensor<1920x1080xf32>  
}
```

Linalg Vectorization: High-level Overview

Diego Caballero – slide from previous LLVM Dev

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

This presentation!



arm

Linalg Vectorizer

Finer details

Step 2: “Loop body” Vectorization

Linalg Vectorizer (Vectorization.cpp)

Transform Dialect “Driver” Ops

```
transform.structured.vectorize ... vector_sizes [4, 4]
```

“Optional”, user-specified.

```
transform.structured.vectorize_children_and_apply_patterns
```

Transform Dialect tutorials (by Alex Zinenko):

- * <https://mlir.llvm.org/docs/Tutorials/transform/>
- * Controllable Transformations in MLIR ([link](#))

C++ “Driver” hooks

```
/// Emit a suitable vector form for an operation.  
mlir::linalg::vectorize(...) {  
  vectorizeConvolution();  
  vectorizeAsTensorPackOp();  
  vectorizeAsLinalgGeneric();  
  ...  
}
```

```
// Patterns (showing only 2!)  
vector::TransferReadOp::getCanonicalizationPatterns()  
vector::TransferWriteOp::getCanonicalizationPatterns()
```

Step 2: “Loop body” Vectorization

Simple case

```
#matmul_trait = {  
  indexing_maps = [  
    affine_map<(m, n, k) -> (m, k)>,  
    affine_map<(m, n, k) -> (k, n)>,  
    affine_map<(m, n, k) -> (m, n)>  
  ],  
  iterator_types = ["parallel", "parallel", "reduction"]  
}
```

vector.transfer_read %A ... : vector<8x16xf32>

```
func.func @matmul(%A: tensor<8x16xf32>, %B: tensor<16x32xf32>, %C: tensor<8x32xf32>) {
```

```
  linalg.generic #matmul_trait
```

```
    ins(%A, %B : tensor<8x16xf32>, tensor<16x32xf32>) outs(%C : tensor<8x32xf32>) {
```

```
      ^bb(%a: f32, %b: f32, %c: f32) :
```

```
        %d = arith.mulf %a, %b: f32
```

```
        %e = arith.addf %c, %d: f32
```

```
        linalg.yield %e : f32
```

```
    } -> tensor<8x32xf32>
```

arith.mulf %vec_a, %vec_b : vector<8x32xf32>

```
  return
```

```
}
```

vector.multi_reduction <add>, %vec_c, %vec_d [2] : vector<8x32x16xf32> to vector<8x32xf32>

Step 2: “Loop body” Vectorization

Dynamic shapes

Static sizes

```
#matmul_trait = {
  indexing_maps = [
    affine_map<(m, n, k) -> (m, k)>,
    affine_map<(m, n, k) -> (k, n)>,
    affine_map<(m, n, k) -> (m, n)>
  ],
  iterator_types = ["parallel", "parallel", "reduction"]
}

func.func @matmul(%A: tensor<8x16xf32>,
                 %B: tensor<16x32xf32>,
                 %C: tensor<8x32xf32>) {
  linalg.generic #matmul_trait
    ins(%A, %B : tensor<8x16xf32>, tensor<16x32xf32>)
    outs(%C : tensor<8x32xf32>) {
      ^bb(%a: f32, %b: f32, %c: f32) :
        %d = arith.mulf %a, %b: f32
        %e = arith.addf %c, %d: f32
        linalg.yield %e : f32
    } -> tensor<8x32xf32>
  return
}
```

VS

Dynamic sizes

```
#matmul_trait = {
  indexing_maps = [
    affine_map<(m, n, k) -> (m, k)>,
    affine_map<(m, n, k) -> (k, n)>,
    affine_map<(m, n, k) -> (m, n)>
  ],
  iterator_types = ["parallel", "parallel", "reduction"]
}

func.func @matmul(%A: tensor<?x?xf32>,
                 %B: tensor<?x?xf32>,
                 %C: tensor<?x?xf32>) {
  linalg.generic #matmul_trait
    ins(%A, %B : tensor<?x?xf32>, tensor<?x?xf32>)
    outs(%C : tensor<?x?xf32>) {
      ^bb(%a: f32, %b: f32, %c: f32) :
        %d = arith.mulf %a, %b: f32
        %e = arith.addf %c, %d: f32
        linalg.yield %e : f32
    } -> tensor<?x?xf32>
  return
}
```


Step 1: Vector-level tiling

Potential source of dynamic shapes

```
%1 = linalg.matmul ins(%A, %B: tensor<1920x135xf32>, tensor<135x1080xf32>)  
      outs(%C: tensor<1920x1080xf32>) -> tensor<1920x1080xf32>
```

```
transform.structured.tile_using_for %matmul tile_sizes [8, 16, 1]
```

Step 1: Vector-level Tiling

```
#map = affine_map<(d0) -> (-d0 + 1080, 16)>
```

```
scf.for %m = %c0 to %c1920 step %c8 ...
```

```
  scf.for %n = %c0 to %c1080 step %c16 ...
```

```
    scf.for %k = %c0 to %c135 step %c1 ...
```

```
      %3 = affine.min #map(%n)
```

```
      %A_tile = tensor.extract_slice %A[%m, %k] [8, 1] ... : tensor<1920x135xf32> to tensor<8x1xf32>
```

```
      %B_tile = tensor.extract_slice %B[%k, %n] [1, %3] ... : tensor<135x1080xf32> to tensor<1x?xf32>
```

```
      %C_tile = tensor.extract_slice %C[%m, %n] [8, %3] ... : tensor<1920x1080xf32> to tensor<8x?xf32>
```

```
      %4 = linalg.matmul ins(%A_tile, %B_tile : tensor<8x1xf32>, tensor<1x?xf32>)
```

```
        outs(%C_tile : tensor<8x?xf32>) -> tensor<8x?xf32>
```

Dynamic sizes

When to specify vector sizes?

+ Input with dynamic shapes

- Dynamic shapes mean that the Vectorizer has no way of guessing what sizes to use.
- User must specify what vector sizes to use.

+ Optimal utilization of vector registers

- You know your architecture better, tell the vectorizer what sizes to use!
- Usually done through tiling ... often leads to dynamic shapes.
 - + Even when the input was static!

+ Any downsides? Masking!

- The vectorizer must be conservative
 - + It assumes that user-specified sizes won't match the actual input/output sizes – hence masks.
- If the masks do match static sizes (yes, that's always checked), no masks are used.

Vector masking

MLIR vs Hardware

+ Regular outer-product (MLIR)

```
%res = vector.outerproduct %lhs, %rhs: vector<8xf32>, f32
```

+ Masked outer-product (MLIR)

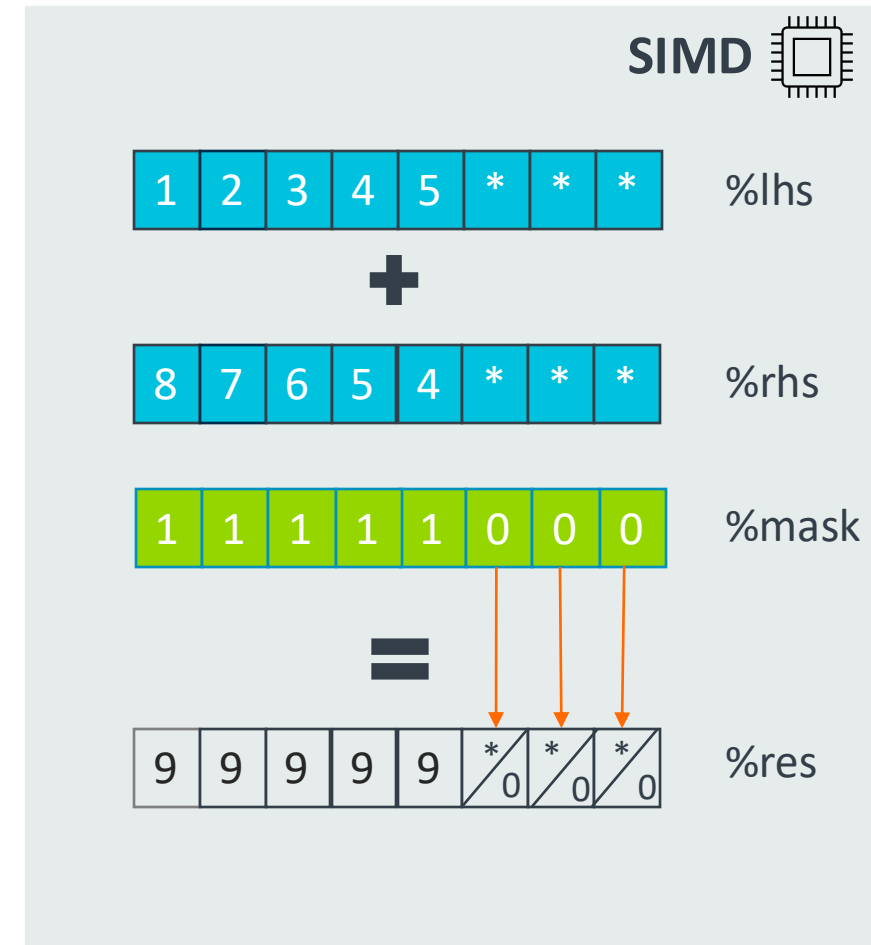
```
%res = vector.mask %mask {  
  vector.outerproduct %lhs, %rhs: vector<8xf32>, f32  
} : vector<8xi1> -> vector<8xf32>
```

+ What's the problem?

- Computing masks is not free
- Most tiles won't require them

Vector Masking

(hardware ISA view)



How to avoid masks?

```
scf.for %m = %c0 to %c1920 step %c8 ...
  scf.for %n = %c0 to %c1080 step %c16 ...
    scf.for %k = %c0 to %c135 step %c1 ...
      %4 = linalg.matmul ins(%A_tile, %B_tile : tensor<8x1xf32>, tensor<1x?xf32>)
        outs(%C_tile : tensor<8x?xf32>) -> tensor<8x?xf32>
```

```
%main_loop, %remainder_loop = transform.loop.peel %loop_to_peel
```

Loop peeling

Complete Example upstream: [link](#)

```
scf.for %m = %c0 to %c1920 step %c8 ...
  // Main loop
  scf.for %n = %c0 to %c1072 step %c16 ...
    scf.for %k = %c0 to %c135 step %c1 ...
      linalg.matmul ins(%A_tile, %B_tile : tensor<8x1xf32>, tensor<1x?xf32>)
        outs(%C_tile : tensor<8x?xf32>) -> tensor<8x?xf32>

  // Remainder loop
  scf.for %arg5 = %c1072 to %c1080 step %c16 iter_args(%arg6 = %1) -> (tensor<1920x1080xf32>)
    scf.for %k = %c0 to %c135 step %c1 ...
      linalg.matmul ins(%A_tile, %B_tile : tensor<8x1xf32>, tensor<1x?xf32>)
        outs(%C_tile : tensor<8x?xf32>) -> tensor<8x?xf32>
```

Static steps

Vectorizing tensor.extract

What should happen here?

```
%c79 = arith.constant 79 : index
%1 = linalg.generic {
  indexing_maps =
    [affine_map<(d0, d1) -> (d0, d1)>],
  iterator_types =
    ["parallel", "parallel"]
} outs(%output : tensor<1x4xf32>) {

^bb0(%out: f32):
  %2 = linalg.index 1 : index
  %3 = affine.apply affine_map<(d0, d1)
    -> (d0 + d1)>(%2, %idx)
  %extracted = tensor.extract %src[%c79, %3]
    : tensor<80x16xf32>
  linalg.yield %extracted : f32
} -> tensor<1x4xf32>
```

?

Gather load
`vector.gather %src ... vector<1x4xf32>`

?

Contiguous load
`vector.transfer_read %src ... : vector<1x4xf32>`

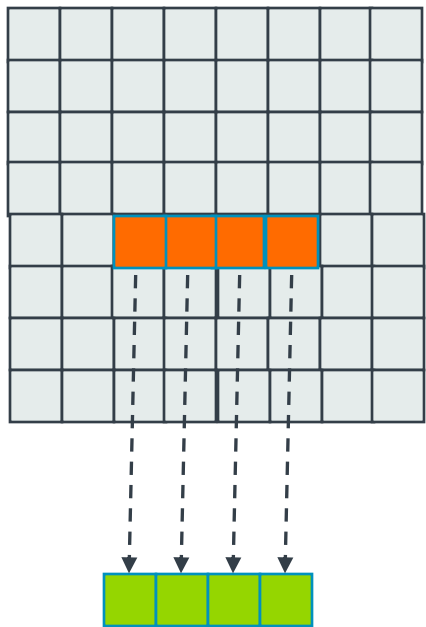
?

Broadcast
`%val = vector.transfer_read %src ... : f32`
`vector.broadcast %val : vector<1x4xf32>`

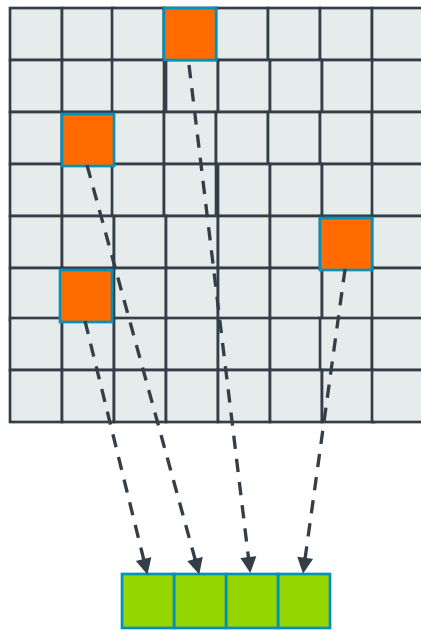
Vectorizing tensor.extract

Simplified view (after vectorization)

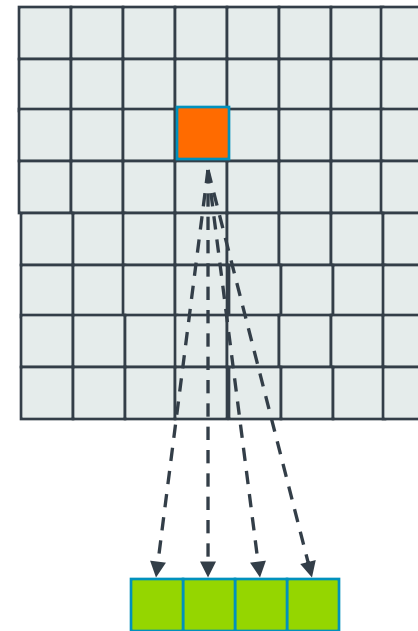
- + Using 2D tensors - in practice, tensors are n-D
- + Contiguous are loads much easier to identify when loading a 1-D “slice”



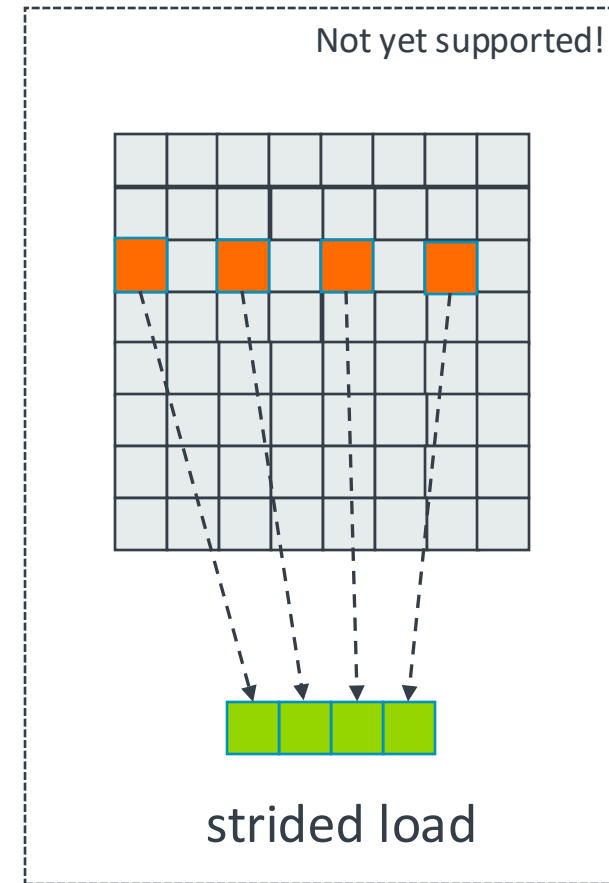
contiguous load



gather load



scalar load + broadcast



strided load

Vectorizing tensor.extract

“Jumping” one column
on every iteration

```
%c79 = arith.constant 79 : index
%1 = linalg.generic {
  indexing_maps =
    [affine_map<(d0, d1) -> (d0, d1)>],
  iterator_types = ["parallel", "parallel"]
} outs(%output : tensor<1x4xf32>) {
^bb0(%out: f32):
  %2 = linalg.index 1 : index
  %3 = affine.apply affine_map<(d0, d1)
    -> (d0 + d1)>(%2, %idx)
  %extracted = tensor.extract %src[%c79, %3]
    : tensor<80x16xf32>
  linalg.yield %extracted : f32
} -> tensor<1x4xf32>
```

vector.transfer_read
(contiguous load)

Slide 13

“Jumping” one row
on every iteration

```
%cst = arith.constant 3 : index
%1 = linalg.generic {
  indexing_maps =
    [affine_map<(d0, d1) -> (d0, d1)>],
  iterator_types = ["parallel", "parallel"]
} outs(%output : tensor<1x4xf32>) {
^bb0(%out: f32):
  %2 = linalg.index 1 : index
  %3 = affine.apply affine_map<(d0, d1)
    -> (d0 + d1)>(%2, %idx)
  %extracted = tensor.extract %src[%3, %cst]
    : tensor<80x16xf32>
  linalg.yield %extracted : f32
} -> tensor<1x4xf32>
```

vector.gather
(gather load)

Constant idxs

```
%c79 = arith.constant 79 : index
%c3 = arith.constant 3 : index
%1 = linalg.generic {
  indexing_maps =
    [affine_map<(d0, d1) -> (d0, d1)>],
  iterator_types = ["parallel", "parallel"]
} outs(%output : tensor<1x4xf32>) {
^bb0(%out: f32):
  %extracted = tensor.extract %src[%c79, %c3]
    : tensor<80x16xf32>
  linalg.yield %extracted : f32
} -> tensor<1x4xf32>
```

vector.transfer_read
+ vector.broadcast
(scalar load + broadcast)

arm

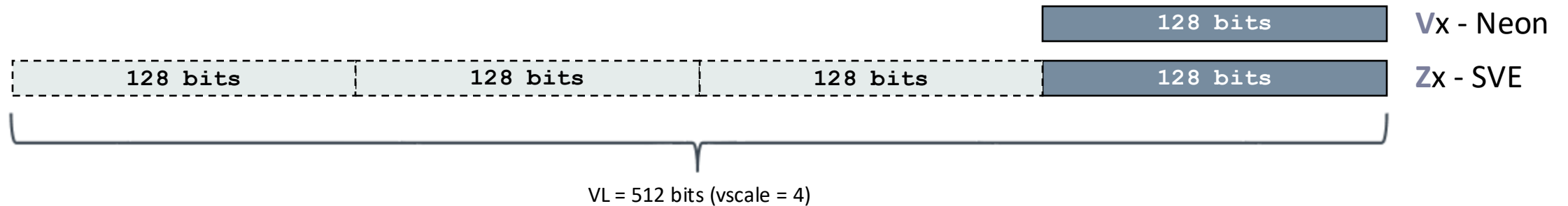
SVE + SME updates

SVE - Scalable Vector Extension ([link](#))
SME – Scalable Matrix Extension ([link](#))

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 masking (**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 Matrix Extension (SME)

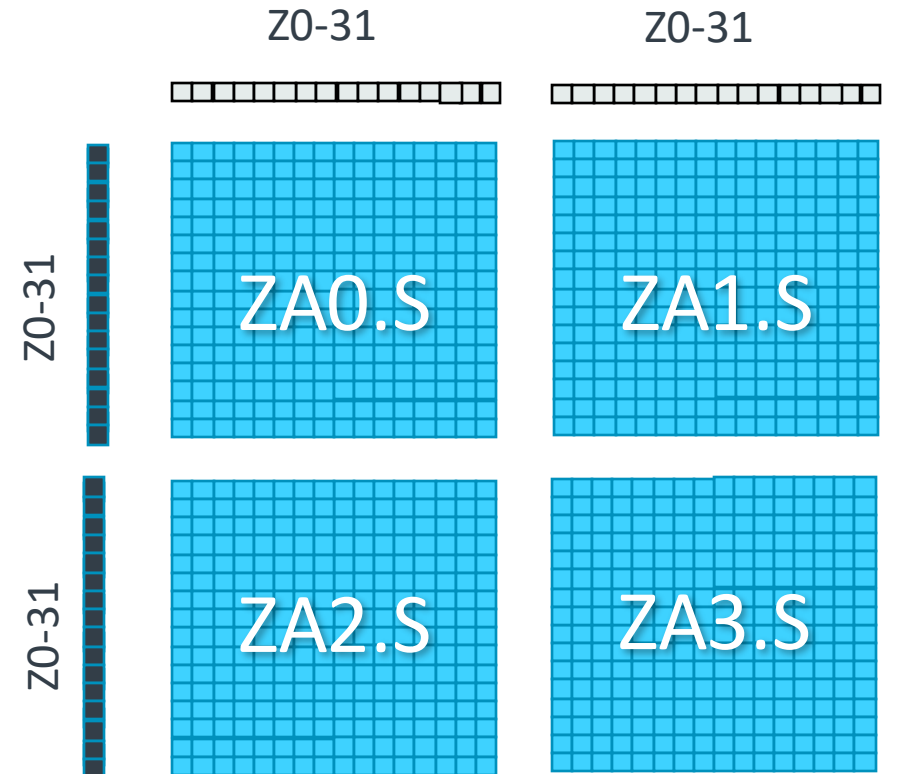
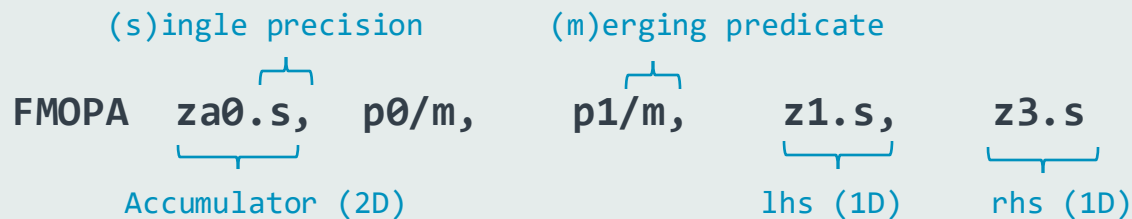
Not only new instructions!

+ Scalable **2D accumulator** - **ZA**

- 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-ZA1.H	8*vscale
i32/f32	4	(4*vscale) x (4*vscale)	ZA0-ZA3.S	4*vscale

+ **Outer product** instructions:



32-bit element tiles

SVL = 512 bits,
4 virtual tiles (16x16)

Step 1: Vector-level Tiling

Tiling for SME (and SVE) leads to dynamic tiles

```
%vscale = vector.vscale  
%c8_vscale = arith.muli %vscale, %c8 : index
```

```
scf.for %m = %c0 to %c1920 step %c8_vscale ... {  
  scf.for %n = %c0 to %c1080 step %c8_vscale ... {  
    scf.for %k = %c0 to %c135 step %c1 ... {
```

```
      %col_A = tensor.extract_slice %tile_A[0, %k] ... : tensor<?x135xf32> to tensor<?x1xf32>  
      %row_B = tensor.extract_slice %tile_B[%k, 0] ... : tensor<135x?xf32> to tensor<1x?xf32>  
      %tile_C = tensor.extract_slice %arg5[0, 0] ... : tensor<?x?xf32> to tensor<?x?xf32>
```

```
      %res = linalg.matmul ins(%col_A, %row_B : tensor<?x1xf32>, tensor<1x?xf32>)  
              outs(%tile_C : tensor<?x?xf32>) -> tensor<?x?xf32>
```

```
    }  
  }  
}
```

Tile sizes for SME: [8] x [8] x 1

No escape from dynamic shapes!

At Tensor level, scalability is modelled through dynamic shapes.

Getting rid of masks (before)

Scalable tile size -> masks

```
#map = affine_map<()>[s0] -> (-(1080 mod s0) + 1080)>
%vscale = vector.vscale
%c4_vscale = arith.muli %vscale, %c4 : index
%ub = affine.apply #map()[%c4_vscale]

scf.for %i = %c0 to %ub step %c4_vscale ... {
  // 1. Extract a slice.
  %slice = tensor.extract_slice %arg[0, %i] [1, %c4_vscale] [1, 1] : tensor<1x?xf32> to tensor<?xf32>

  // 2. Create a mask for the slice.
  %dim = tensor.dim %slice, %c0 : tensor<?xf32>
  %mask = vector.create_mask %dim : vector<[4]xi1>

  // 3. Read the slice and do some computation.
  %lhs = vector.transfer_read %slice[%c0] ... %mask ... : tensor<?xf32>, vector<[4]xf32>
  %vec = vector.mask %mask {
    vector.outerproduct %lhs, %rhs {kind = #vector.kind<add>} : vector<[4]xf32>, f32
  } : vector<[4]xi1> -> vector<[4]xf32>

  // 4. Write the new value.
  %write = vector.transfer_write %vec, %slice[%c0], %mask ... : vector<[4]xf32>, tensor<?xf32>

  // 5. Insert and yield the new tensor value.
  %result = tensor.insert_slice %write into %arg[0, %i] [1, %c4_vscale] [1, 1] : tensor<?xf32> into tensor<1x?xf32>
}
```

Loop upper-bound guaranteed to be a multiple of %c4_vscale by e.g. tiling.

Always = %c4_vscale

Getting rid of masks (after)

ValueBoundsAnalysis to the rescue!



No masks!

```
#map = affine_map<()>[s0] -> (-(1080 mod s0) + 1080)>
%vscale = vector.vscale
%c4_vscale = arith.muli %vscale, %c4 : index
%ub = affine.apply #map()[%c4_vscale]

scf.for %i = %c0 to %ub step %c4_vscale ... {
  // 1. Extract a slice.
  %slice = tensor.extract_slice %arg[0, %i] [1, %c4_vscale] [1, 1] : tensor<1x?xf32> to tensor<?xf32>

  // 2. Create a mask for the slice.
  // Gone entirely after -canonicalize

  // 3. Read the slice and do some computation.
  %lhs = vector.transfer_read %slice[%c0], %cst {in_bounds = [true]} : tensor<?xf32>, vector<[4]xf32>
  %vec = vector.outerproduct %lhs, %rhs {kind = #vector.kind<add>} : vector<[4]xf32>, f32

  // 4. Write the new value.
  %3 = vector.transfer_write %vec, %slice[%c0] {in_bounds = [true]} : vector<[4]xf32>, tensor<?xf32>

  // 5. Insert and yield the new tensor value.
  %result = tensor.insert_slice %3 into %arg[0, %i] [1, %c4_vscale] [1, 1] : tensor<?xf32> into tensor<1x?xf32>
}
```

Getting rid of masks (references)

+ “Computing Bounds of SSA Values in MLIR” ([link](#))

- EuroLLVM ‘24 presentation by Matthias Springer

+ ScalableValueBoundsConstraintSet ([link](#))

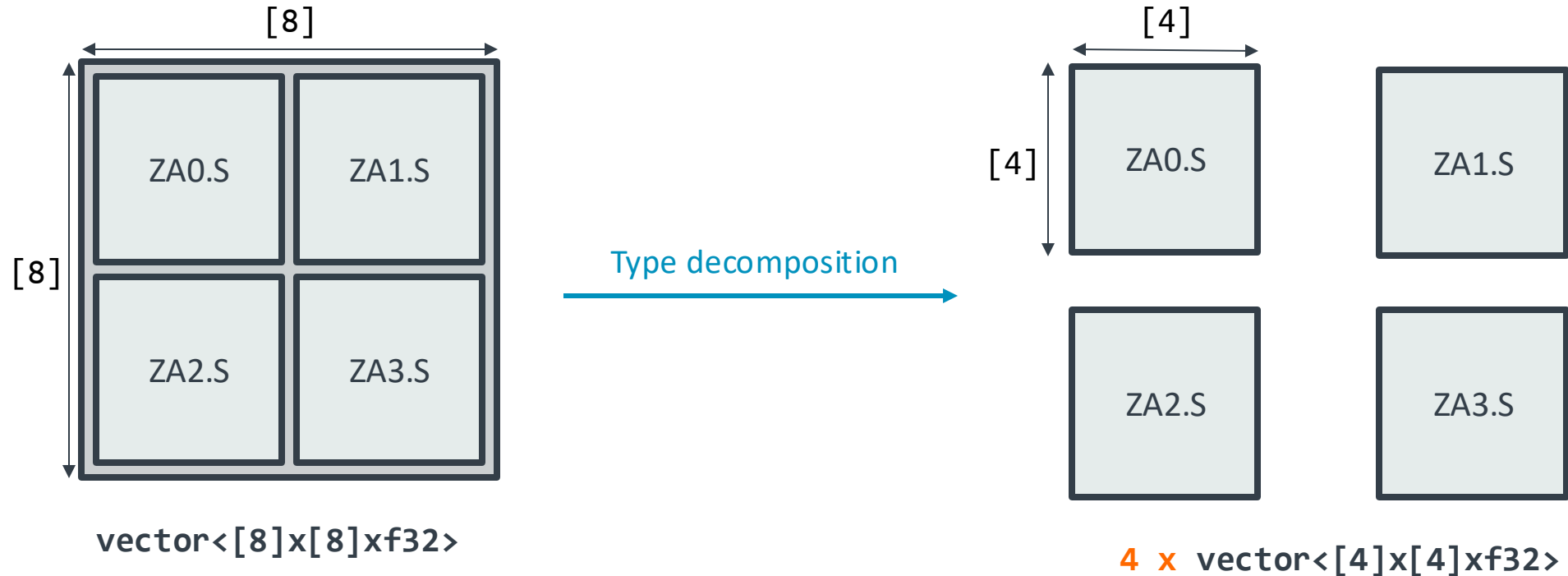
- A specialization of `ValueBoundsConstraintSet` for scalable vectors.
- Added to avoid masks and help with performance for scalable vectors.

+ Example from slides – based on eliminate-masks.mlir ([link](#))

- Use `-canonicalize` for the masks to disappear entirely
 - + Otherwise, you’ll only see **vector.create_mask** being replaced with **vector.constant.mask** (still good!)
- Use `--debug` to see ValueBoundsAnalysis debug output.

Step 4: Vector Dimensionality Legalization

Vector Legalization Pass for SME



+ See VectorLegalizationPass in mlir/lib/Dialect/ArmSME ([link](#))

Step 4: Vector Dimensionality Legalization (SME)

```
%tile_C_out = vector.mask %op_mask {  
    vector.outerproduct %col_A, %row_B, %acc {kind = #vector.kind<add>} : vector<[8]xf32>, vector<[8]xf32>  
}  
: vector<[8]x[8]xi1> -> vector<[8]x[8]xf32>
```

Type decomposition

```
...  
%vscale = vector.vscale  
%c4_vsacle = arith.muli %vscale, %c4 : index  
%num_cols_2 = arith.addi %num_cols_1, %c4_vsacle : index  
  
%op_mask_2 = vector.create_mask %num_rows_1, %num_cols_2 : vector<[4]x[4]xi1>  
%sve_vec_A_2 = vector.scalable.extract %col_A[0] : vector<[4]xf32> from vector<[8]xf32>  
%sve_vec_B_2 = vector.scalable.extract %row_B[4] : vector<[4]xf32> from vector<[8]xf32>  
%res_2 = vector.mask %op_mask_2 {  
    vector.outerproduct %sve_vec_A_2, %sve_vec_B_2, %acc_3 ... : vector<[4]xf32>, vector<[4]xf32> }  
: vector<[4]x[4]xi1> -> vector<[4]x[4]xf32>
```

4 x times

(1 per SME tile)

Virtual Tile Allocation

A very low-level step specific to ArmSME

- + SME's [TileAllocationPass](#) builds on top of MLIR's [Liveness API](#)
 - Implements a linear scan register allocator
 - Helps us avoid expensive spilling/filling through memory

```
func.func @sme_tile_allocation(%cond: i1) {
  %tileA = arm_sme.get_tile : vector<[4]x[4]xf32>
  cf.cond_br %cond, ^bb2, ^bb1
^bb1:
  %tileB = arm_sme.get_tile : vector<[16]x[16]xi8>
  "test.some_use"(%tileB) : (vector<[16]x[16]xi8>) -> ()
  cf.br ^bb3
^bb2:
  %tileC = arm_sme.get_tile : vector<[4]x[4]xf32>
  "test.some_use"(%tileC) : (vector<[4]x[4]xf32>) -> ()
  "test.some_use"(%tileA) : (vector<[4]x[4]xf32>) -> ()
  cf.br ^bb3
^bb3:
  return
}
```

```
func.func @sme_tile_allocation(%arg0: i1) {
  %0 = arm_sme.get_tile {tile_id = 0 : i32} : vector<[4]x[4]xf32>
  cf.cond_br %arg0, ^bb2, ^bb1
^bb1: // pred: ^bb0
  %1 = arm_sme.get_tile {tile_id = 0 : i32} : vector<[16]x[16]xi8>
  "test.some_use"(%1) : (vector<[16]x[16]xi8>) -> ()
  cf.br ^bb3
^bb2: // pred: ^bb0
  %2 = arm_sme.get_tile {tile_id = 1 : i32} : vector<[4]x[4]xf32>
  "test.some_use"(%2) : (vector<[4]x[4]xf32>) -> ()
  "test.some_use"(%0) : (vector<[4]x[4]xf32>) -> ()
  cf.br ^bb3
^bb3: // 2 preds: ^bb1, ^bb2
  return
}
```

"Injects" valid SME virtual tile IDs

arm

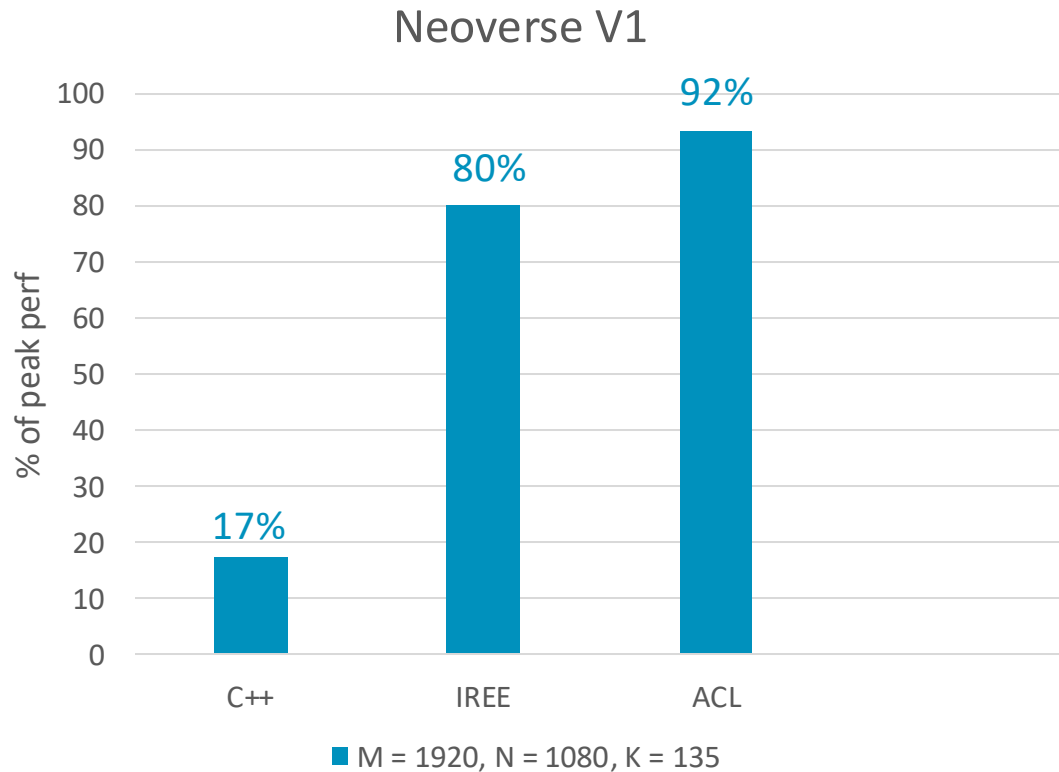
Benchmark results

(Proof of concept evaluation)

Matmul + Conv

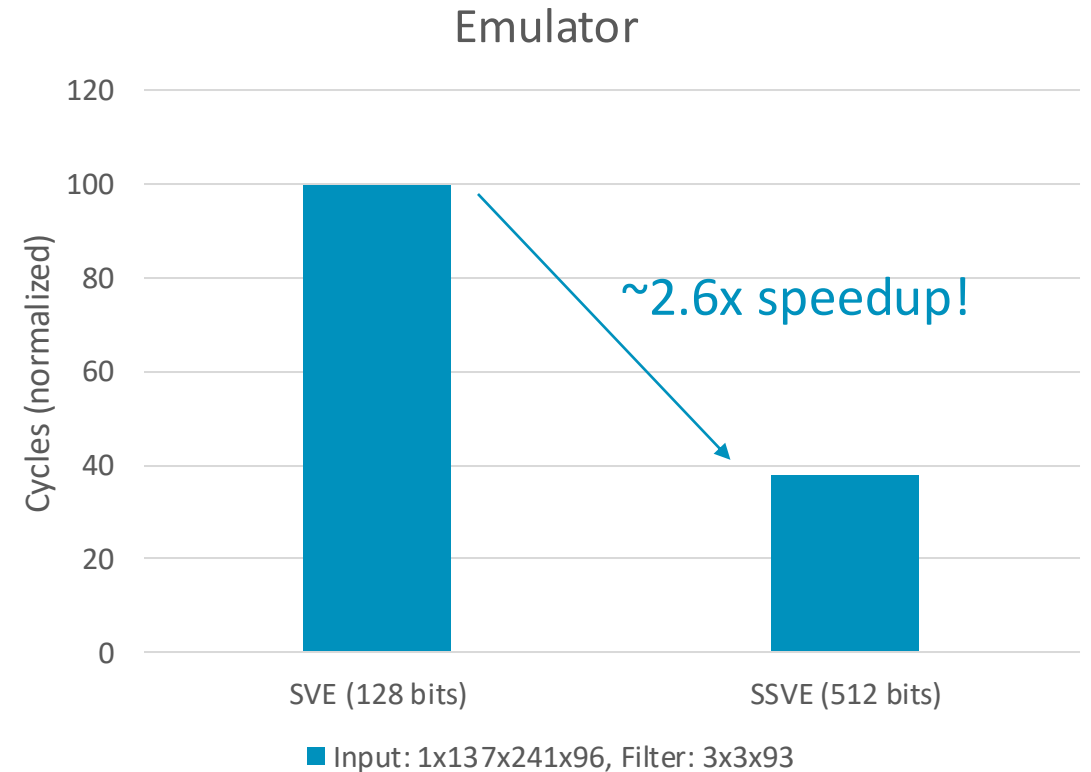
linalg.matmul

C++ vs IREE (SVE) vs ACL (SVE)



linalg.depthwise_conv_2d_nhwc_hwc

SVE vs SSVE



Disclaimers + related work

+ **POC results - more work needed to make this easily reproducible**

+ **Why only one matmul?**

- These results are not consistent across problem sizes.
 - + Used hand-tuned tile sizes. We should use a cost-model instead.
- Dimensions extracted from an internal demo (so “real”).

+ **Numbers close to known research results (good, but could be better)**

- “MLIR-based Code Generation for High-Performance Machine Learning on AArch64” ([link](#))
 - + Johanna Gustafson, 2024, reports ~86% peak perf on AArch64
- “High Performance Code Generation in MLIR: An Early Case Study with GEMM” ([link](#))
 - + Uday Bondhugula, 2020, reports ~82% peak perf on X86
- The authors are not using Linalg or IREE. Only one problem size is benchmarked.

+ **Used IREE as our e2e compiler (<https://iree.dev>)**

- “Unveiling the Inner Workings of IREE” by Mahesh Ravishankar ([link](#))
- IREE SHA: fa670d64ea

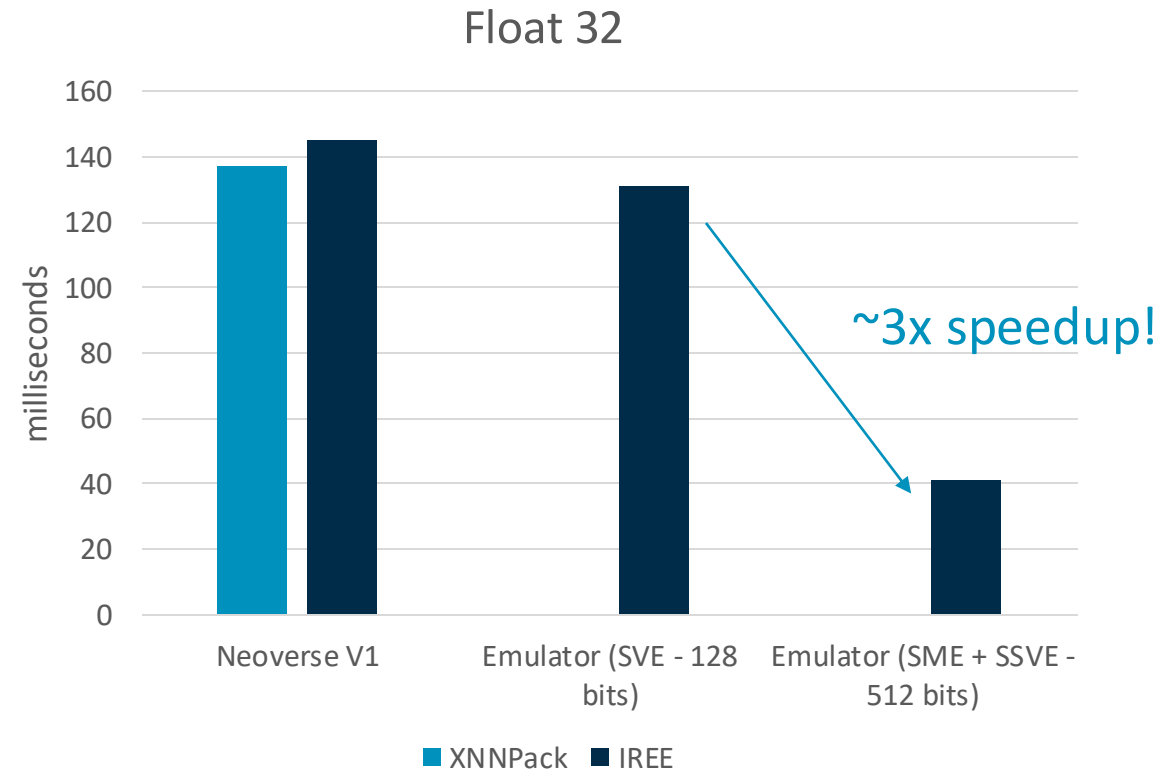
SME results

Does the perf scale? Yes!

Context

- + Benchmark using internal **CNN pipeline**
 - Mix of matmuls and depthwise convolutions
 - Represents a real end-user App
- + Evaluation done on **two platforms**:
 - **Neoverse V1** (publicly available hardware)
 - + XNNpack (libs) vs IREE (codegen)
 - **SME Emulator** (internal prototype)
 - + SVE (“host” CPU) – **128 bits** vs SSVE/SME– **512 bits**
- + XNNPack results – **pre KleidiAI** integration ([link](#))
 - Things are bound to get faster once integrated
- + Perf results are good, but there’s room for more!
 - Very pleased with the perf scaling!

SVE vs SME



Final points



arm

