Targeting SME from MLIR

MLIR ODM

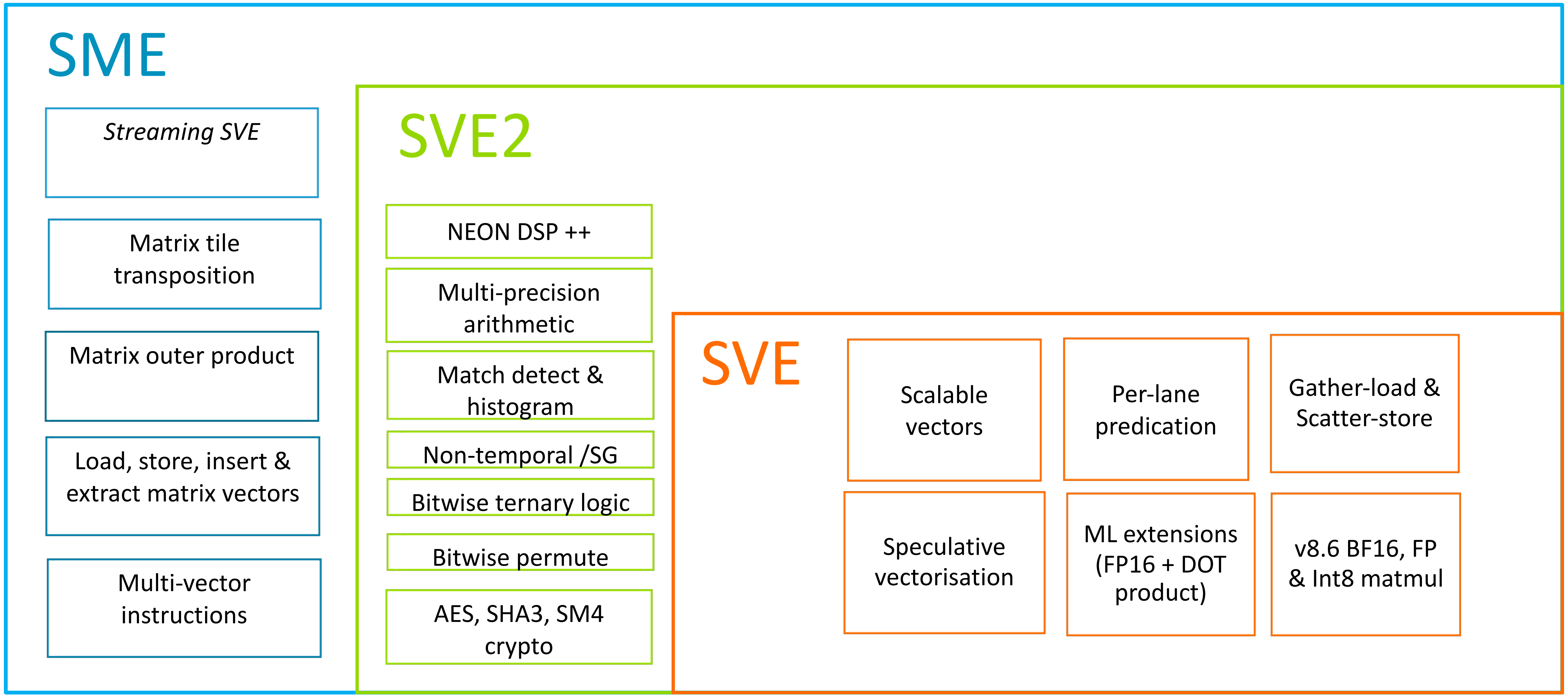
Andrzej Warzyński

June 2023

Intro

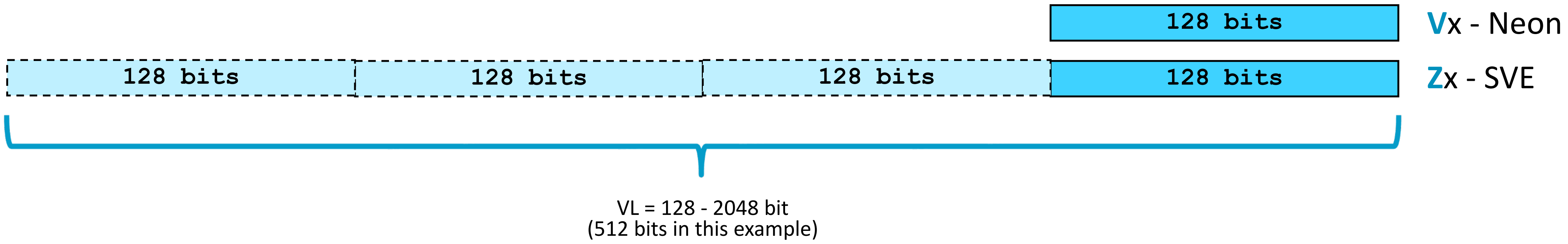
- **Key question:** what's the best way to target CPU "accelerators" from MLIR?
- **SME** – Scalable Matrix Extension
  - An outer-product engine
  - Builds on the Scalable Vector Extensions (SVE and SVE2)
  - Adds new capabilities to efficiently process matrices
  - No hardware at the moment, but you can use QEMU
- **MLIR** will most likely be one of major sources of code run on SME
  - Good and … scalable support are key.
- Please use the **specs** as the ultimate source of truth:
  - SME Instructions
  - Arm Architecture Reference Manual Supplement
- Thank you to my lovely arm colleagues and friends for help with these slides!
  - Blame me (Andrzej) for all the mistakes.

SME extends SVE2

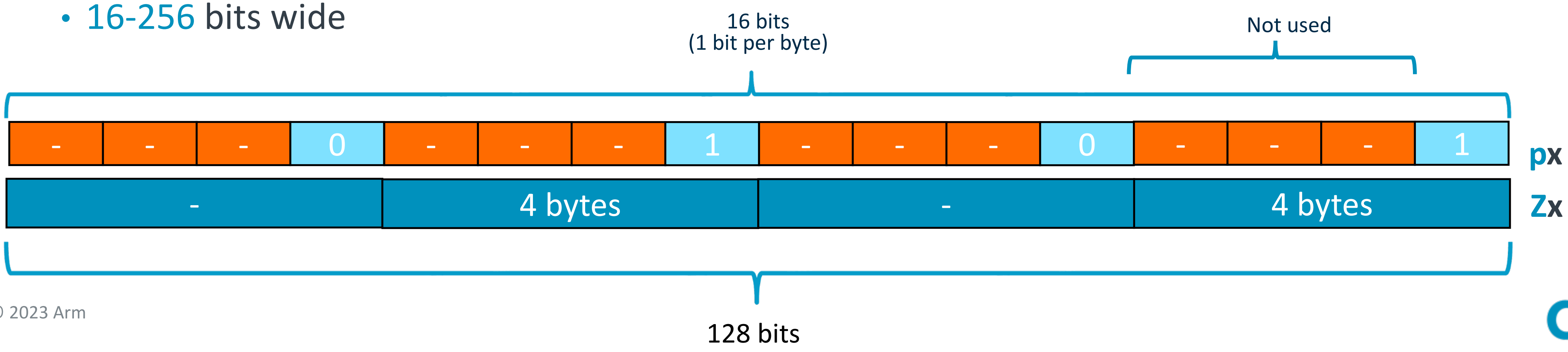

SME
Streaming SVE
Matrix tile transposition
Matrix outer product
Load, store, insert & extract matrix vectors
Multi-vector instructions
SVE2
NEON DSP ++
Multi-precision arithmetic
Match detect & histogram
Non-temporal /SG
Bitwise ternary logic
Bitwise permute
AES, SHA3, SM4 crypto
SVE
Scalable vectors
Per-lane predication
Gather-load & Scatter-store
Speculative vectorisation
ML extensions (FP16 + DOT product)
v8.6 BF16, FP & Int8 matmul

Scalable Vector registers – SVE intro

- 32 scalable vector registers (`Z0-Z31`):
  - 128-2048 bits vector length is decided by **implementation**
  - VL (vector length) is **unknown at compile-time**, but **known at run-time**



- 16 scalable predicate registers (`p0-p15`):
  - 16-256 bits wide



Scalable vectors in LLVM and MLIR

- **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>
}
```

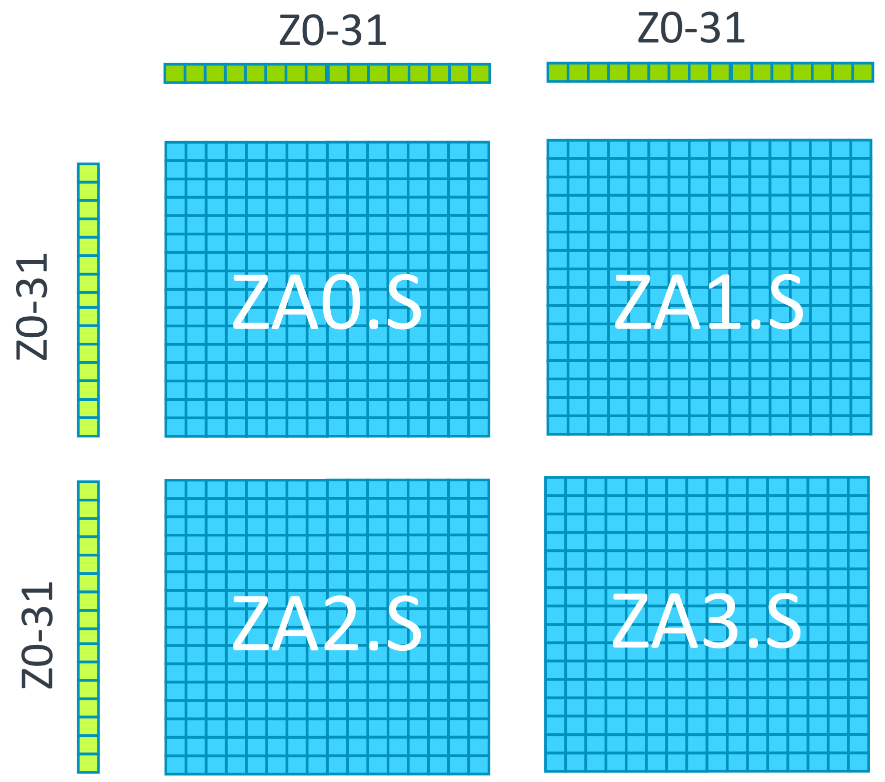
- The actual value of **vscale** is not known at compile time.
  - Use LLVM's llvm.vscale or MLIR's vector.vscale to get an SSA value that represents it.

SME Register state

Not only new instructions!

- 32 scalable SVE vector **Z registers**
- 16 scalable SVE predicate **P registers**
- **New!** Scalable 2D **ZA accumulator**
  - With horizontal & vertical "slice" access.
- **ZA** contains virtual tiles depending on element size:

element	# tiles	ZA vritual tile dims	tile names	Z reg dims
i8	1	(16*vscale) x (16*vscale)	ZA0.B	16*vscale
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**
i64/f64	8	(2*vscale) x (2*vscale)	ZA0-ZA7.D	2*vscale
i128	16	(1*vscale) x (1*vscale)	ZA0-ZA15.Q	1*vscale



32-bit element tiles

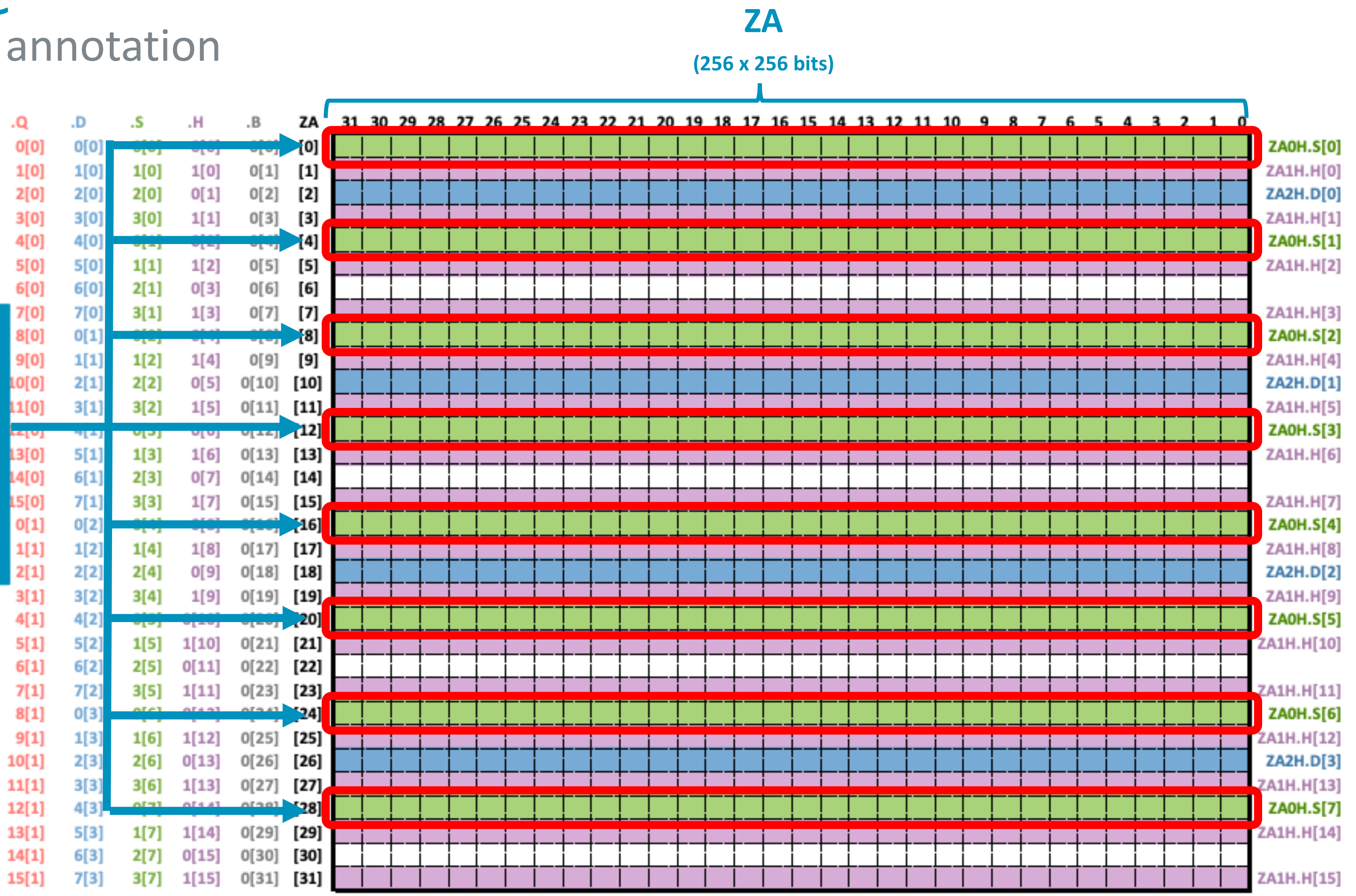
(SVL = 512 bits, 16x16 tiles)

SME Virtual Tile

Spec diagram with my annotation

ZA0.s

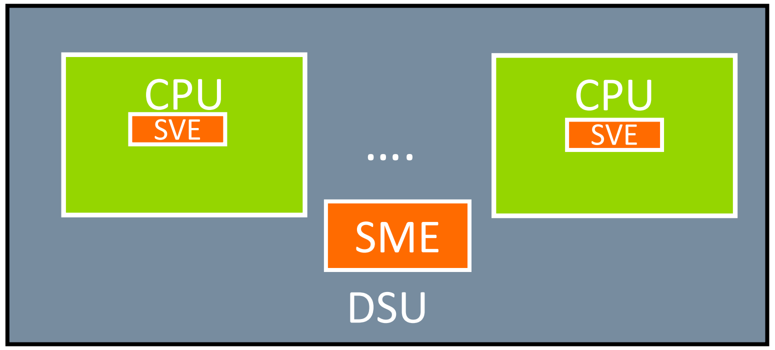
- 8x8xi32 (256x256 bit SME)
  - .s -> single word (32 bit)
- 3 other “virtual” tiles for i32:
  - ZA1.s, ZA2.s, ZA3.s
- More views available!

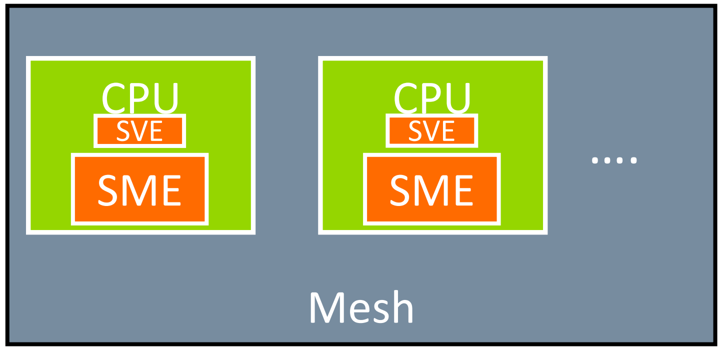


ZA
(256 x 256 bits)
.Q .D .S .H .B ZA
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0[0] 0[0] [0] ZA0H.S[0]
1[0] 1[0] 1[0] 1[0] 0[1] [1] ZA1H.H[0]
2[0] 2[0] 2[0] 0[1] 0[2] [2] ZA2H.D[0]
3[0] 3[0] 3[0] 1[1] 0[3] [3] ZA1H.H[1]
4[0] 4[0] [4] ZA0H.S[1]
5[0] 5[0] 1[1] 1[2] 0[5] [5] ZA1H.H[2]
6[0] 6[0] 2[1] 0[3] 0[6] [6]
7[0] 7[0] 3[1] 1[3] 0[7] [7] ZA1H.H[3]
8[0] 0[1] [8] ZA0H.S[2]
9[0] 1[1] 1[2] 1[4] 0[9] [9] ZA1H.H[4]
10[0] 2[1] 2[2] 0[5] 0[10] [10] ZA2H.D[1]
11[0] 3[1] 3[2] 1[5] 0[11] [11] ZA1H.H[5]
[12] ZA0H.S[3]
13[0] 5[1] 1[3] 1[6] 0[13] [13] ZA1H.H[6]
14[0] 6[1] 2[3] 0[7] 0[14] [14]
15[0] 7[1] 3[3] 1[7] 0[15] [15] ZA1H.H[7]
0[1] 0[2] [16] ZA0H.S[4]
1[1] 1[2] 1[4] 1[8] 0[17] [17] ZA1H.H[8]
2[1] 2[2] 2[4] 0[9] 0[18] [18] ZA2H.D[2]
3[1] 3[2] 3[4] 1[9] 0[19] [19] ZA1H.H[9]
4[1] 4[2] [20] ZA0H.S[5]
5[1] 5[2] 1[5] 1[10] 0[21] [21] ZA1H.H[10]
6[1] 6[2] 2[5] 0[11] 0[22] [22]
7[1] 7[2] 3[5] 1[11] 0[23] [23] ZA1H.H[11]
8[1] 0[3] [24] ZA0H.S[6]
9[1] 1[3] 1[6] 1[12] 0[25] [25] ZA1H.H[12]
10[1] 2[3] 2[6] 0[13] 0[26] [26] ZA2H.D[3]
11[1] 3[3] 3[6] 1[13] 0[27] [27] ZA1H.H[13]
12[1] 4[3] [28] ZA0H.S[7]
13[1] 5[3] 1[7] 1[14] 0[29] [29] ZA1H.H[14]
14[1] 6[3] 2[7] 0[15] 0[30] [30]
15[1] 7[3] 3[7] 1[15] 0[31] [31] ZA1H.H[15]

SME – possible implementations

- The spec leaves some key details for implementations to define





- **Streaming Mode SVE (SSVE)** is introduced to differentiate from **non-streaming** SVE
  - Enabled/disabled through smstart and smstop
  - Also used to enable/disable **ZA**

```
// Enable/disable ZA
define void @toggle_pstate_za() {
  call void @llvm.aarch64.sme.za.enable()
  call void @llvm.aarch64.sme.za.disable()
  ret void
}
```

LLVM IR

```
llc -mtriple=aarch64 -mattr=+sme
```

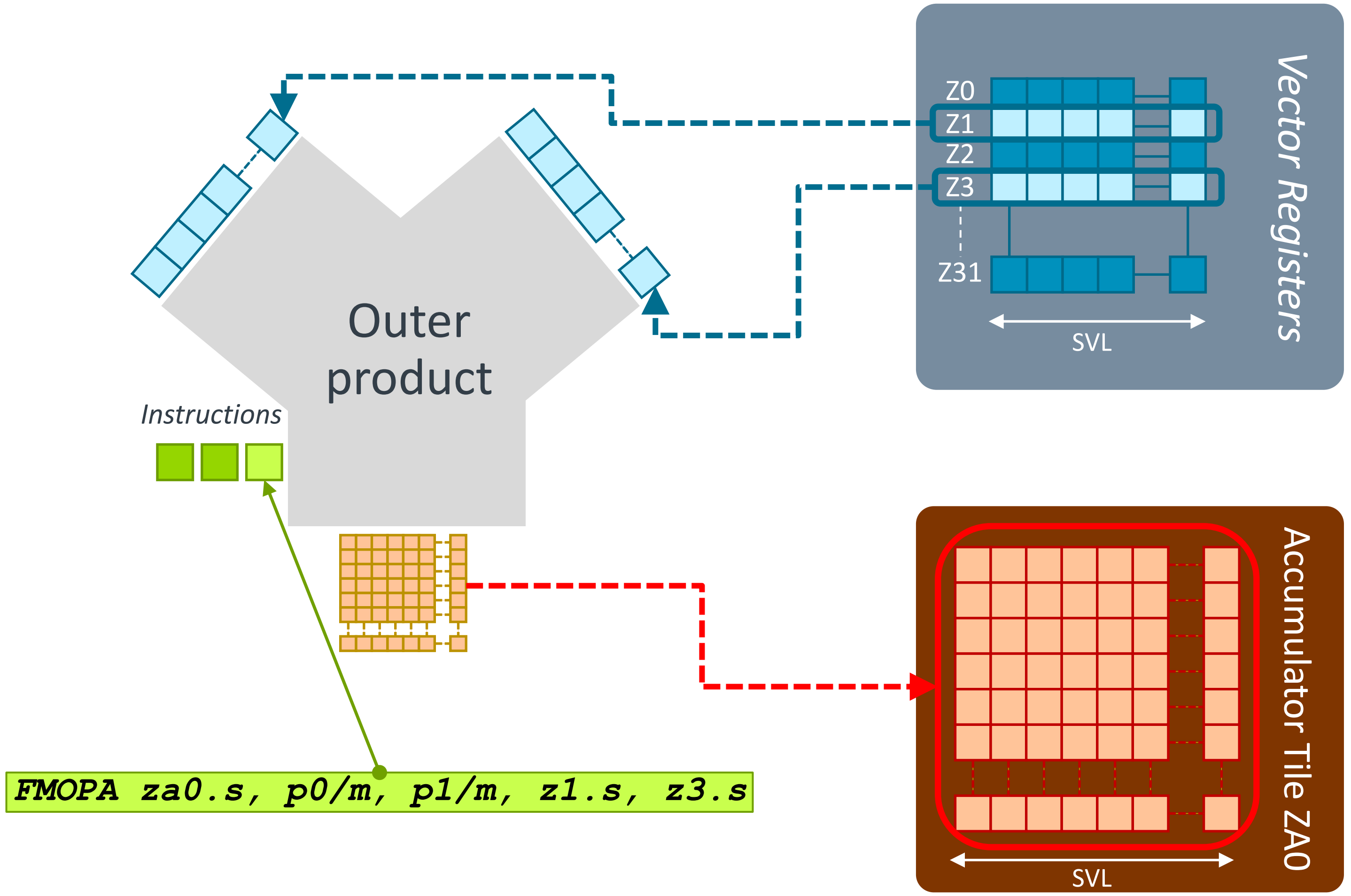
```
toggle_pstate_za:
        .cfi_startproc
// %bb.0:
        smstart  za
        smstop   za
        ret
.Lfunc_end0:
```

Assembly

SVE vs SSVE

- **VL** (Vector Length) != **SVL** (Streaming Vector Length)
  - e.g. 128 bits vs 512 bits, or
  - 128 bits vs 128 bits
- **SSVE** != **SVE 2** (at ISA level)
  - By default, some SVE 2 instructions are not supported in the streaming mode
    - Example: gather loads and scatter stores, NEON instructions
  - The list is relatively short and **depends on implementation**
    - There are architectural flags that you can query
- **Streaming** vs **non-streaming mode**
  - **ZA** and **Z** registers are cleared upon execution mode transition
  - Nested streaming mode kernels are supported
  - See LLVM intrinsics that model this:
    - aarch64_pstate_sm_enabled, aarch64_pstate_sm_compatible , aarch64_pstate_za_shared etc
    - Designed to enable ACLE (Arm C Language Extension). Can be re-used in MLIR if we want to.

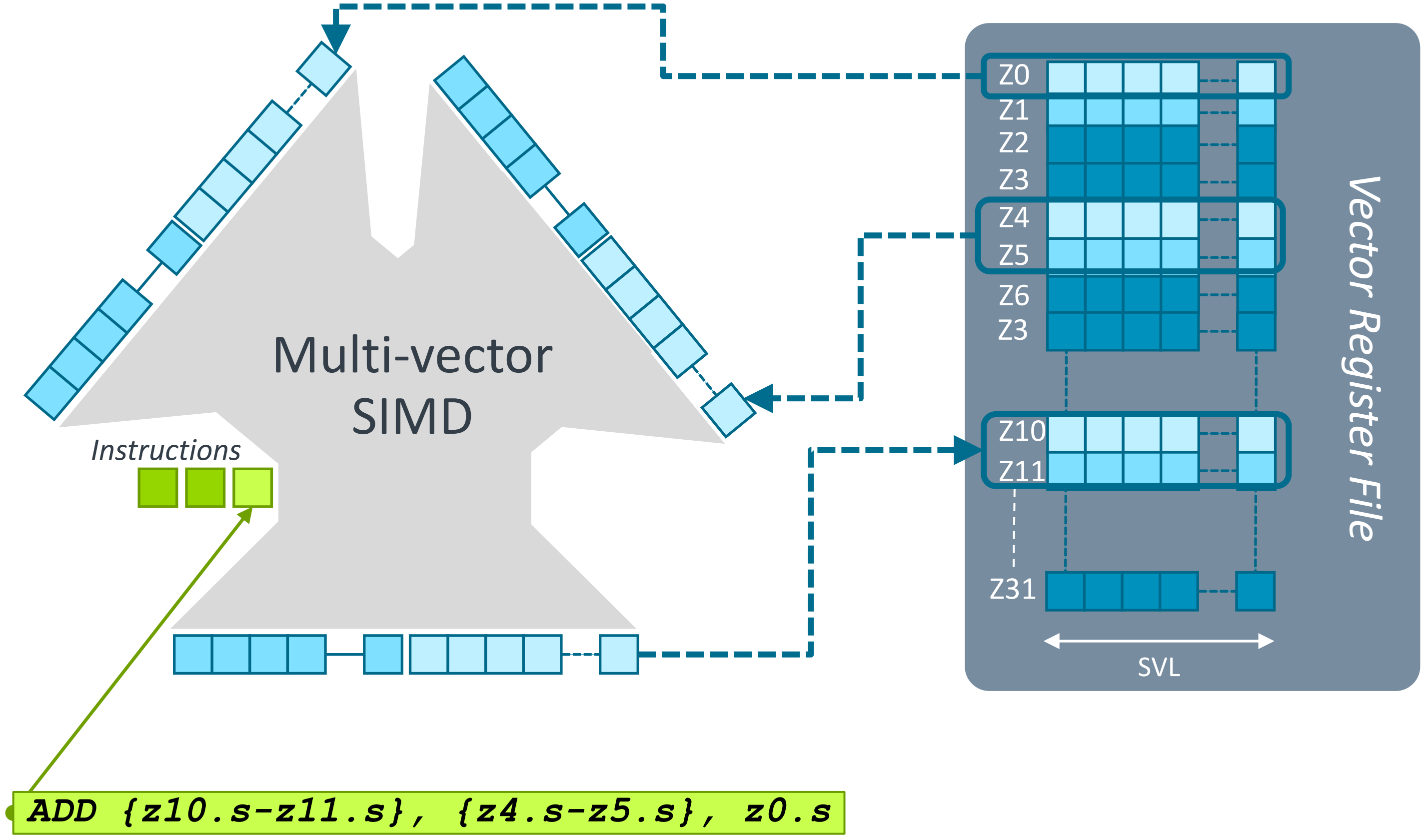
Outer product (Z→ZA)



```
FMOPA za0.s, p0/m, p1/m, z1.s, z3.s
```

- An **outer product** of the source vectors
  - Reads two SVE **Z** input vectors
  - Updates an entire virtual **ZA** tile
  - **Spec:** FMOPA (non-widening)
- **2-D predication:**
  - 1 predicate register per input vector
  - masks individual row/column updates
- **Syntax**:
  - This variant operates on "**s**ingle" words (z1.**s**)
  - Uses "**m**erging" predication (p1/**m**)
  - Tile number is "**0**" (za**0**.s)
- **Usage**:
  - Matrix-matrix multiply, linear equation solvers, matrix inversion, 1D/2D filters, etc.

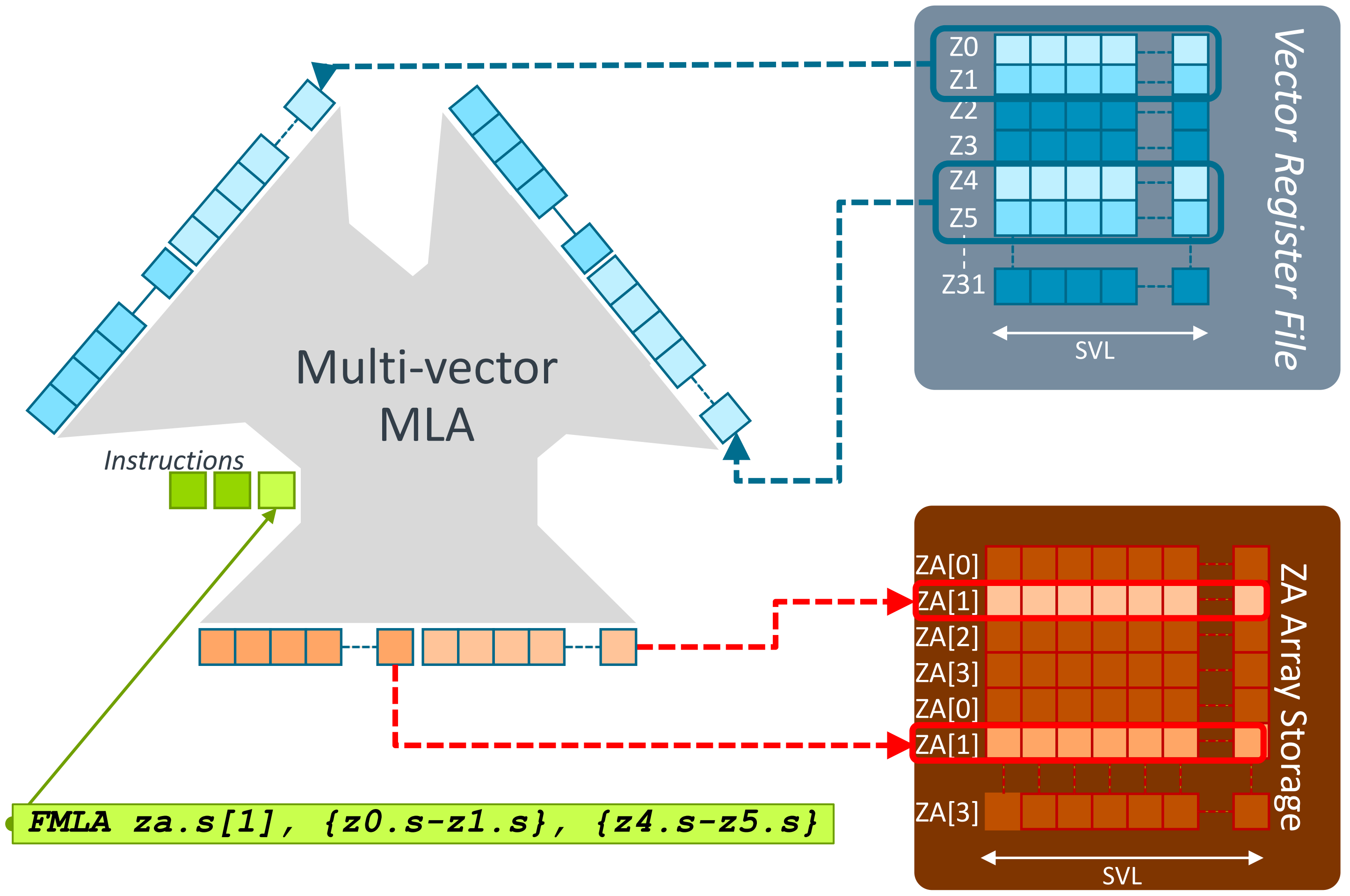
Multi-vector add (Z→Z)



```
ADD {z10.s-z11.s}, {z4.s-z5.s}, z0.s
```

- **Add** replicated single vector to multi-vector with multi-vector result
  - 2 x **Z** and 4 x **Z** vector variants
  - **Spec:** ADD (to vector)
- **Other** multi-vector **Z→Z** insts:
  - min/max, add/sub, shift, convert
  - Usage:
    - pre/post-processing of matrix data
  - **Spec:** SMAX, SMIN, SRSHL

Non-widening multi-vector multiply-add (Z→ZA)



```
FMLA za.s[1], {z0.s-z1.s}, {z4.s-z5.s}
```

- Multi-vector fused **multiply-add**
  - Results added to **ZA** single-vector group
  - **2 x Z and 4 x Z vector variants**
  - **Spec:** FMLA (multiple vectors)
- **Re-uses ZA array:**
  - 64 additional vector registers at SVL=512
- **Syntax:**
  - For simplicity, I hard-coded the output vector group as "**1**" (za.s[1])
- **Widening variants:**
  - **BFMLAL**, **SMLALL**

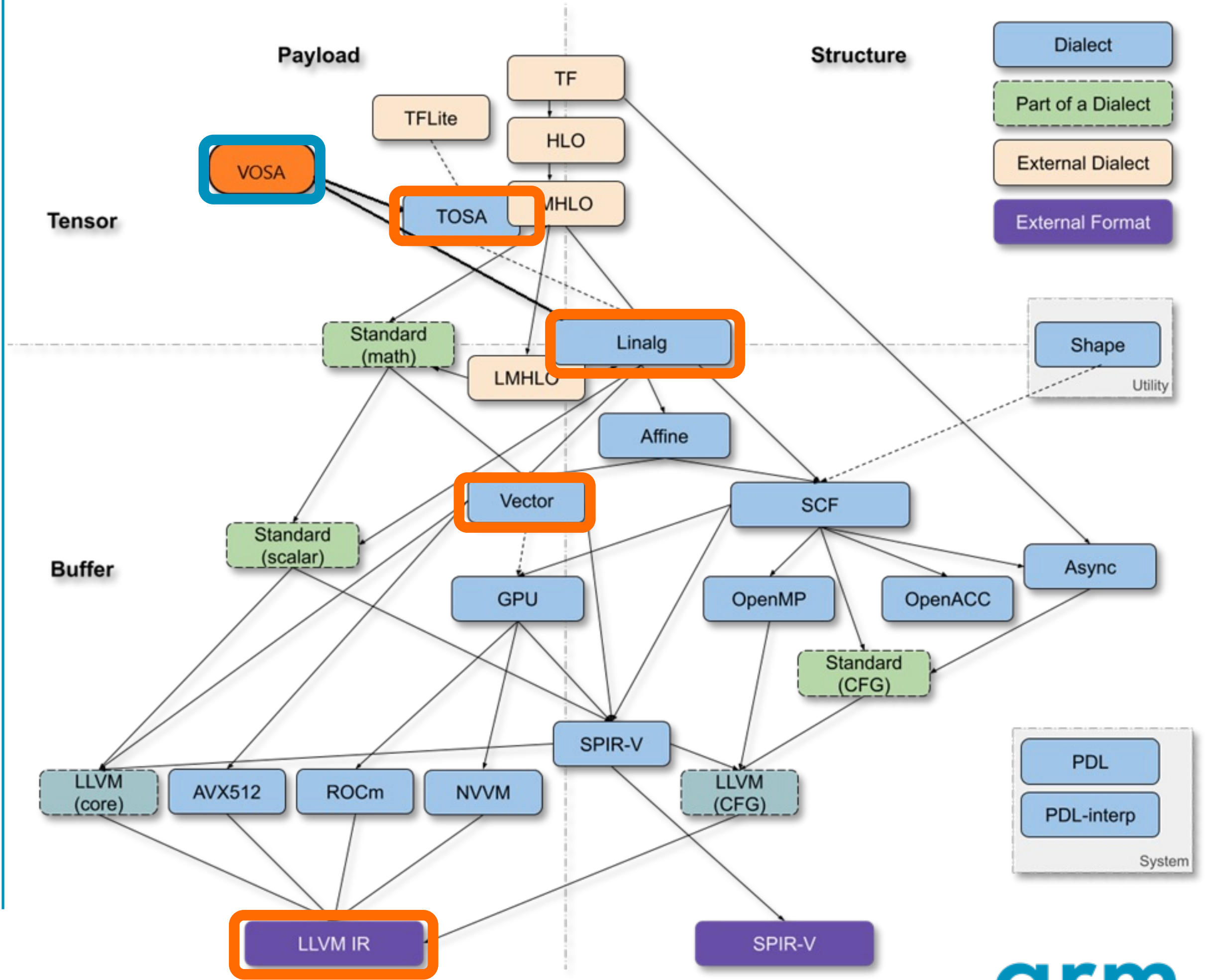
Targeting SME from MLIR - proposal

Key design principles

- **Goals**:
  - Prioritise re-use: vectorization, tiling, etc
  - Don't leak any architectural constraints …
    - … beyond what's unavoidable
  - End-to-end integration tests
    - Introduce early, use for e.g. validation
  - Support **TOSA** and **VOSA**:
    - VOSA - sibling of TOSA for Computer Vision

- **Assumptions (for now)**:
  - No nested kernels
    - To avoid tricky ABI considerations
  - No mixed types when accessing **ZA**
    - To avoid different tile sizes/types
  - Each kernel to fully own and utilize **ZA**
    - No sharing means fewer problems

Key dialects to re-use



SME in MLIR – Lowering to LLVM

- We need to make sure that the **Streaming Mode** and **ZA** are enabled
  - Leveraging LLVM backend and ACLE work as much as possible
- **Func dialect -** function attributes:

```
func.func @arm_sme() attributes {arm_streaming, arm_za} {
    return
}
```

- **LLVM dialect -** LLVM attributes + intrinsics:

```
llvm.func @arm_sme() attributes {arm_locally_streaming} {
    "arm_sme.intr.za.enable"() : () -> ()
    "arm_sme.intr.za.disable"() : () -> ()
    llvm.return
}
```

SME in MLIR – Lowering linalg.matmul

```
func.func @matmul(%mat_A: memref<6x8xf32>, %mat_B: memref<6x8xf32>, %mat_C: memref<8x8xf32>) {
    linalg.matmul ins(%mat_A, %mat_B: memref<6x8xf32>, memref<6x8xf32>) outs(%mat_C: memref<8x8xf32>)
    return
}
```

Linalg

```
func.func @outerproduct_matmul(%mat_A_tr: memref<6x8xf32>, %mat_B: memref<6x8xf32>, %mat_C: memref<8x8xf32>) {
    (…)
    // Compute tile 0 of C
    %tile0_C = vector.transfer_read %mat_C[%c0, %c0], %cst {in_bounds = [true, true]} : memref<8x8xf32>, vector<4x4xf32>
    %op_0 = vector.outerproduct %A_col_0_0, %B_row_0_0, %tile0_C {kind = #vector.kind<add>} : vector<4xf32>, vector<4xf32>

    // Compute tile 1 of C
    %tile1_C = vector.transfer_read %mat_C[%c0, %c4], %cst {in_bounds = [true, true]} : memref<8x8xf32>, vector<4x4xf32>
    %op_6 = vector.outerproduct %A_col_0_0, %B_row_0_1, %tile1_C {kind = #vector.kind<add>} : vector<4xf32>, vector<4xf32>

    (…)
    // Compute tile 0 of C
    %tile2_C = vector.transfer_read %mat_C[%c4, %c0], %cst {in_bounds = [true, true]} : memref<8x8xf32>, vector<4x4xf32
    %op_12 = vector.outerproduct %A_col_0_1, %B_row_0_0, %tile2_C {kind = #vector.kind<add>} : vector<4xf32>, vector<4xf32>

    (…)
    // Compute tile 0 of C
    %tile3_C = vector.transfer_read %mat_C[%c4, %c4], %cst {in_bounds = [true, true]} : memref<8x8xf32>, vector<4x4xf32>
    %op_18 = vector.outerproduct %A_col_0_1, %B_row_0_1, %tile3_C {kind = #vector.kind<add>} : vector<4xf32>, vector<4xf32>
}
```

Vector

SME in MLIR – linalg.matmul

- Transform dialect sequence to lower linalg.matmul to something SME-friendly:

```
transform.sequence failures(propagate) {
^bb0(%arg1: !transform.any_op):
    %0 = transform.structured.match ops{["linalg.matmul"]} in %arg1 : (!transform.any_op) -> !transform.any_op
    %tiled, %loops:2 = transform.structured.tile %0 [4, 4] : (!transform.any_op) ->
        (!transform.any_op, !transform.op<"scf.for">, !transform.op<"scf.for">)
    %1 = get_closest_isolated_parent %tiled : (!transform.any_op) -> !transform.any_op
    %2 = transform.structured.vectorize %1 : (!transform.any_op) -> !transform.any_op
    %3 = transform.structured.match ops{["scf.for"]} in %2 : (!transform.any_op) -> !transform.op<"scf.for">
    transform.loop.unroll %3 { factor = 2 } : !transform.op<"scf.for">
    transform.apply_patterns to %arg1 {
         transform.apply_patterns.vector.lower_contraction lowering_strategy = "outerproduct"
   } : !transform.any_op
}
                                                                                                         Transform
```

- We know that MLIR can generate code that's friendly for SME!
  - Full example on GitHub Gist

Progress so far

- **Scalable vectorization:**
  - [RFC] Scalable Vectorisation in Linalg
  - [mlir][Vector] Add basic scalable vectorization support to Linalg vectorizer
- **Streaming SVE enablement**:
  - [RFC] Supporting Armv9 Scalable Matrix Extension (SME) Streaming SVE (SSVE) mode in MLIR
  - [mlir] Add pass to enable Armv9 Streaming SVE mode
- **Testing:**
  - [mlir][aarch64] Enable MLIR integration tests for SVE/SME under emulation
  - [mlir][ArmSME] Add tests for Streaming SVE
- **Lowering to SME from MLIR:**
  - [mlir][ArmSME] Dialect and Intrinsic Op Definition
  - [mlir][ArmSME] Add basic lowering of vector.transfer write to zero

What about prior art?

- Why wouldn't we re-use all the amazing work that went into the AMX dialect?
  - MLIR ODM presentation by Aart Bik
  - AMX Dialect
- We do, it is a great source of inspiration! However …
  - AMX is not scalable
  - The number of tile registers is fixed (it is not in the SME case)
  - The AMX dialect is not connected to Linalg nor to Vector dialects
    - We would like to target SME from Linalg
  - AMX is implemented as a HWV layer
    - Is that what we need for SME? Not clear!
- 2D vectors are just a specialization of MLIR's n-D Vectors
    - No need for anything special for AMX and/or SME

Q: What's the best way to target CPU "accelerators" from MLIR?