arm Research

Vector Architecture for HPC and ML

Alex Rico
Staff Research Engineer
Architecture Research

Severo Ochoa Research Seminars
BSC, UPC, July 2nd, 2018

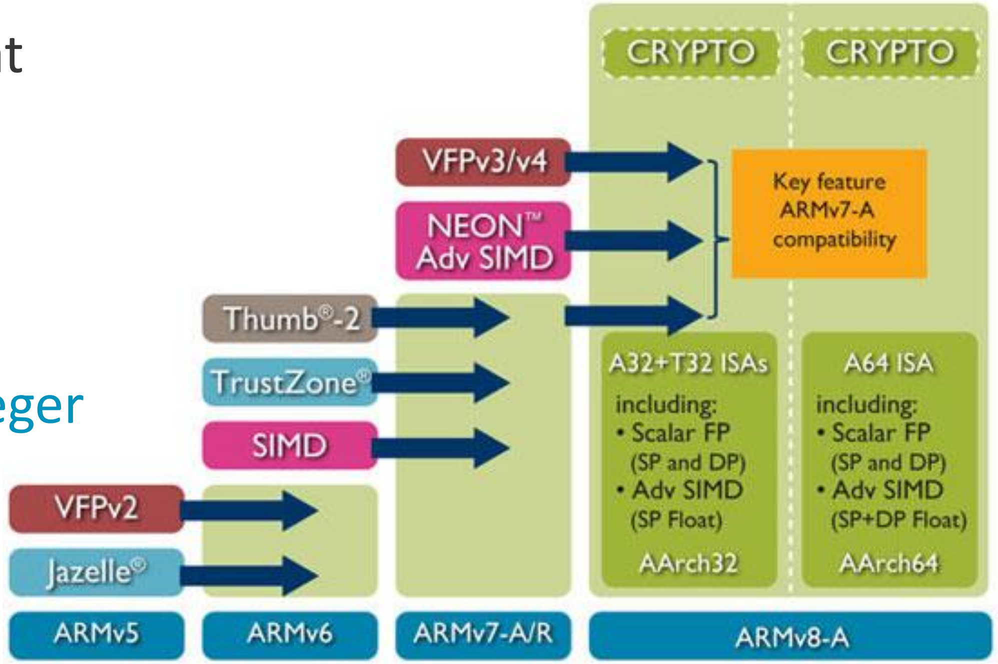
Arm Architecture

Armv7 Advanced SIMD (*aka* Arm NEON instructions) now 12 years old

- Integer, fixed-point and non-IEEE single-precision float
- 16 × 128-bit vector registers

AArch64 Advanced SIMD was an evolution

- Gained full IEEE double-precision float and 64-bit integer vector ops
- 32 × 128-bit vector registers



Scalable Vector Extension – SVE

Significantly extends vector processing capabilities of AArch64

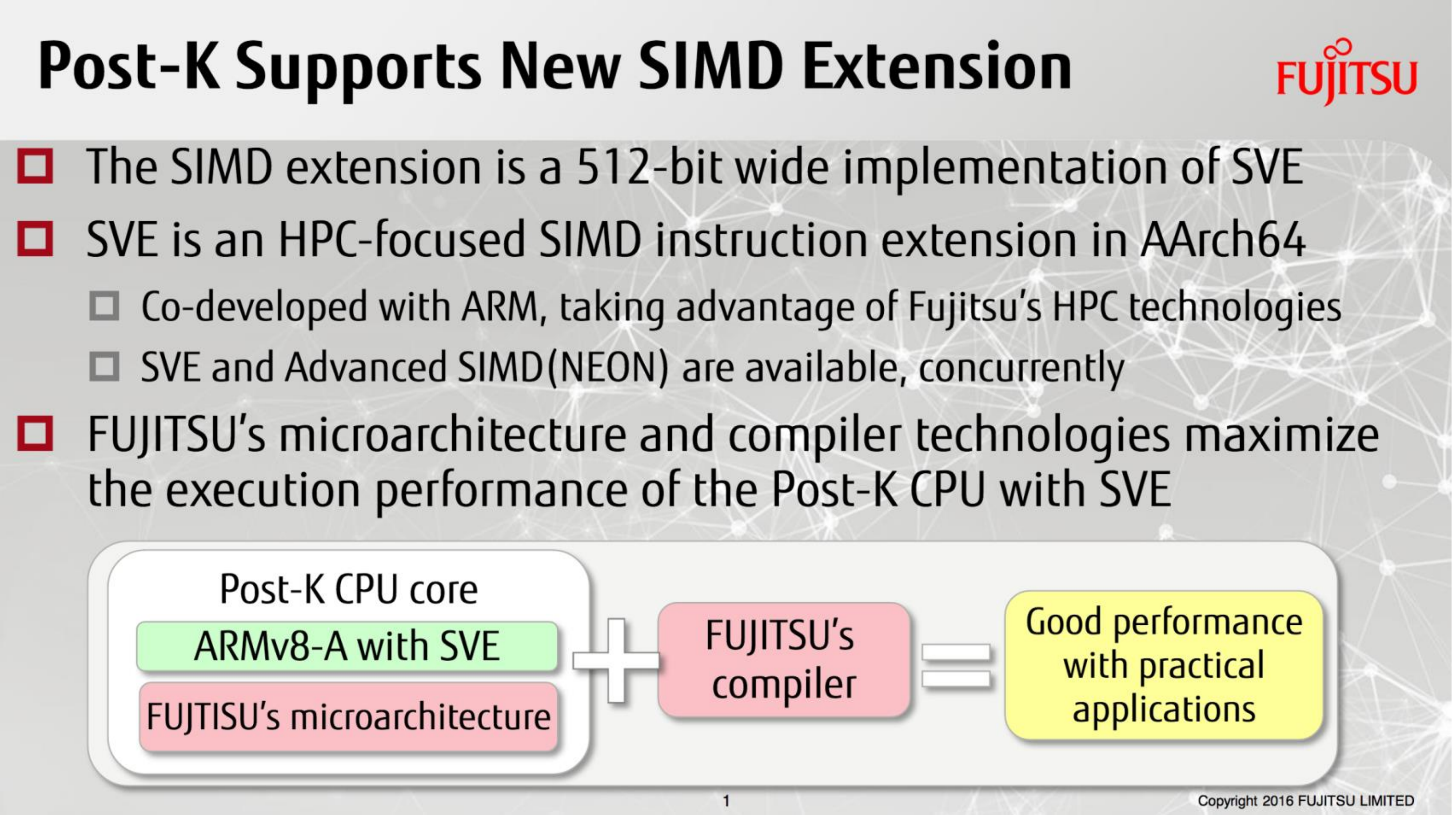
Enables implementation choices of vector lengths – 128 to 2048 bits

- *Vector Length Agnostic* (VLA) programming adjusts dynamically to the available VL
- No need to recompile, or to rewrite hand-coded SVE assembler or C intrinsics

Focus is HPC scientific workloads and machine learning, not media/image processing

Will enable advanced vectorizing compilers to extract more fine-grain parallelism from existing code and so reduce software deployment effort

Post-K Supercomputer goes Arm with SVE



Post-K Supports New SIMD Extension

- The SIMD extension is a 512-bit wide implementation of SVE
- SVE is an HPC-focused SIMD instruction extension in AArch64
  - Co-developed with ARM, taking advantage of Fujitsu's HPC technologies
  - SVE and Advanced SIMD(NEON) are available, concurrently
- FUJITSU's microarchitecture and compiler technologies maximize the execution performance of the Post-K CPU with SVE

Post-K CPU core

ARMv8-A with SVE

FUJTISU's microarchitecture

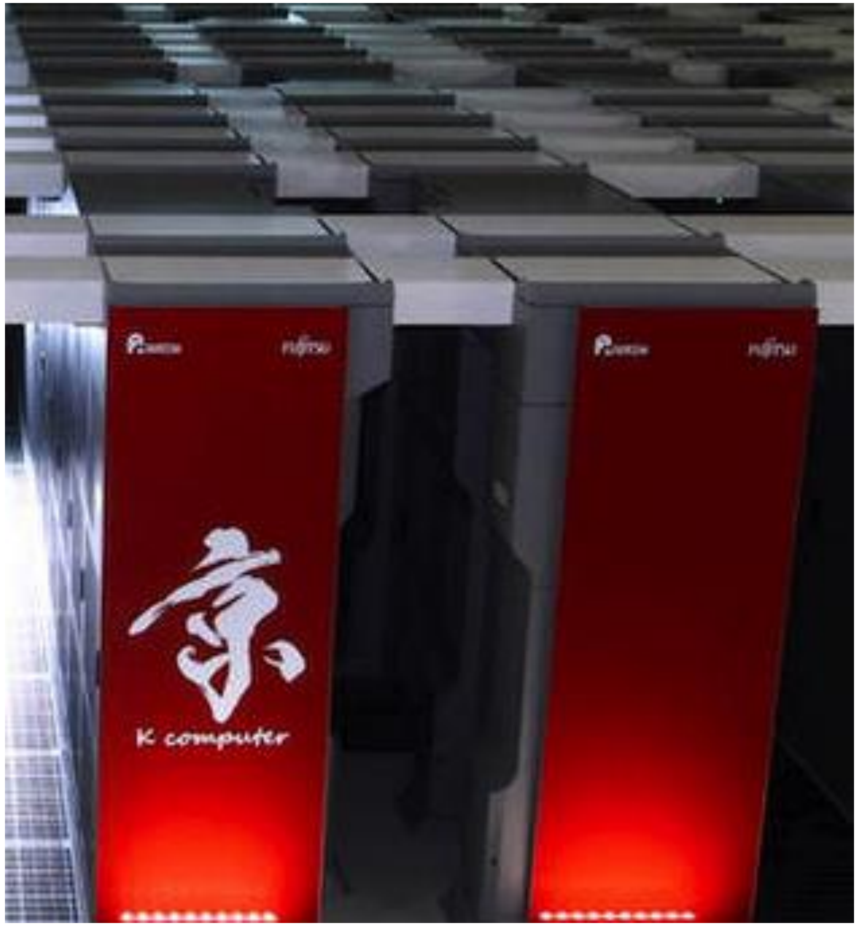
+

FUJITSU's compiler

=

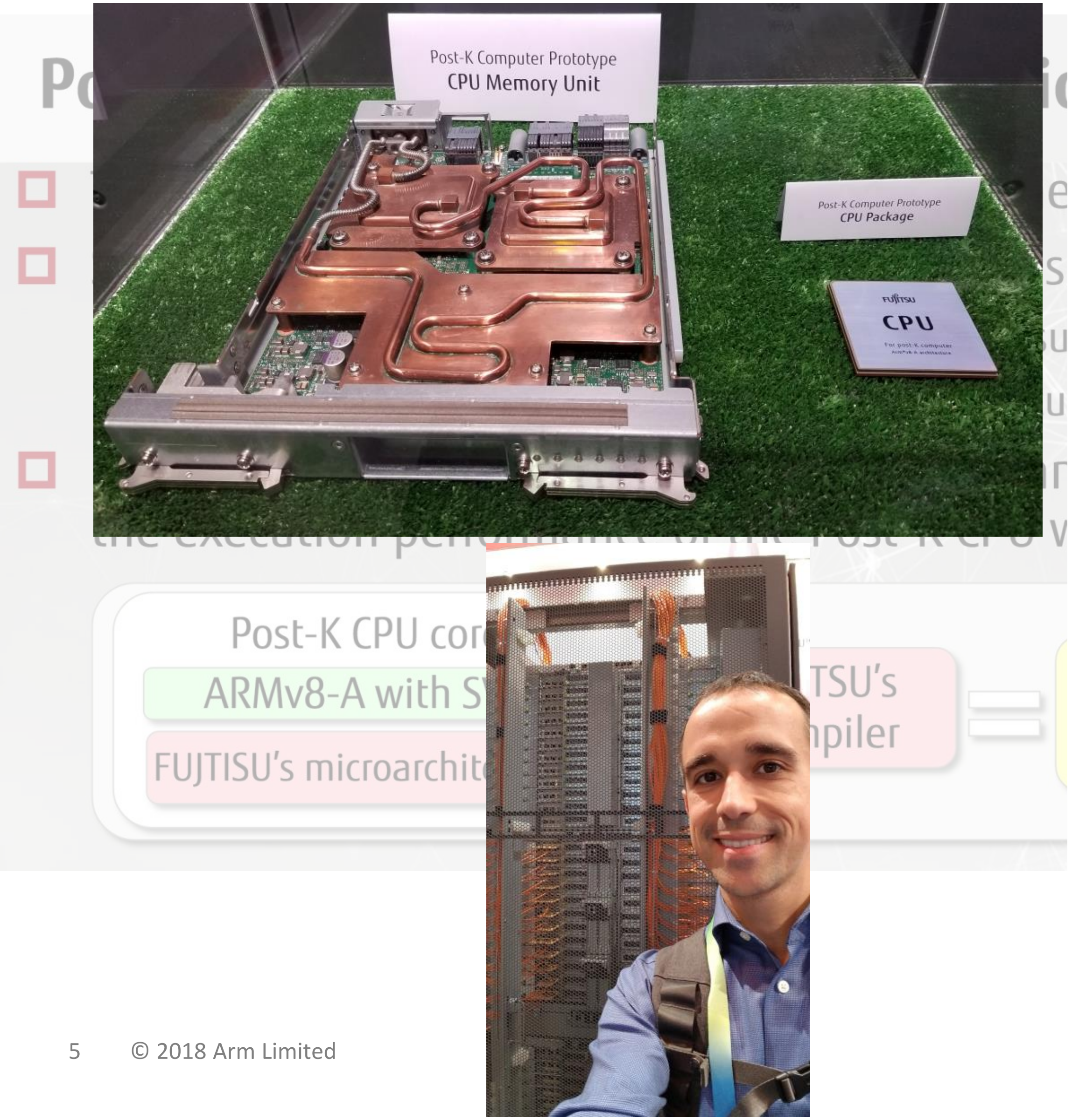
Good performance with practical applications

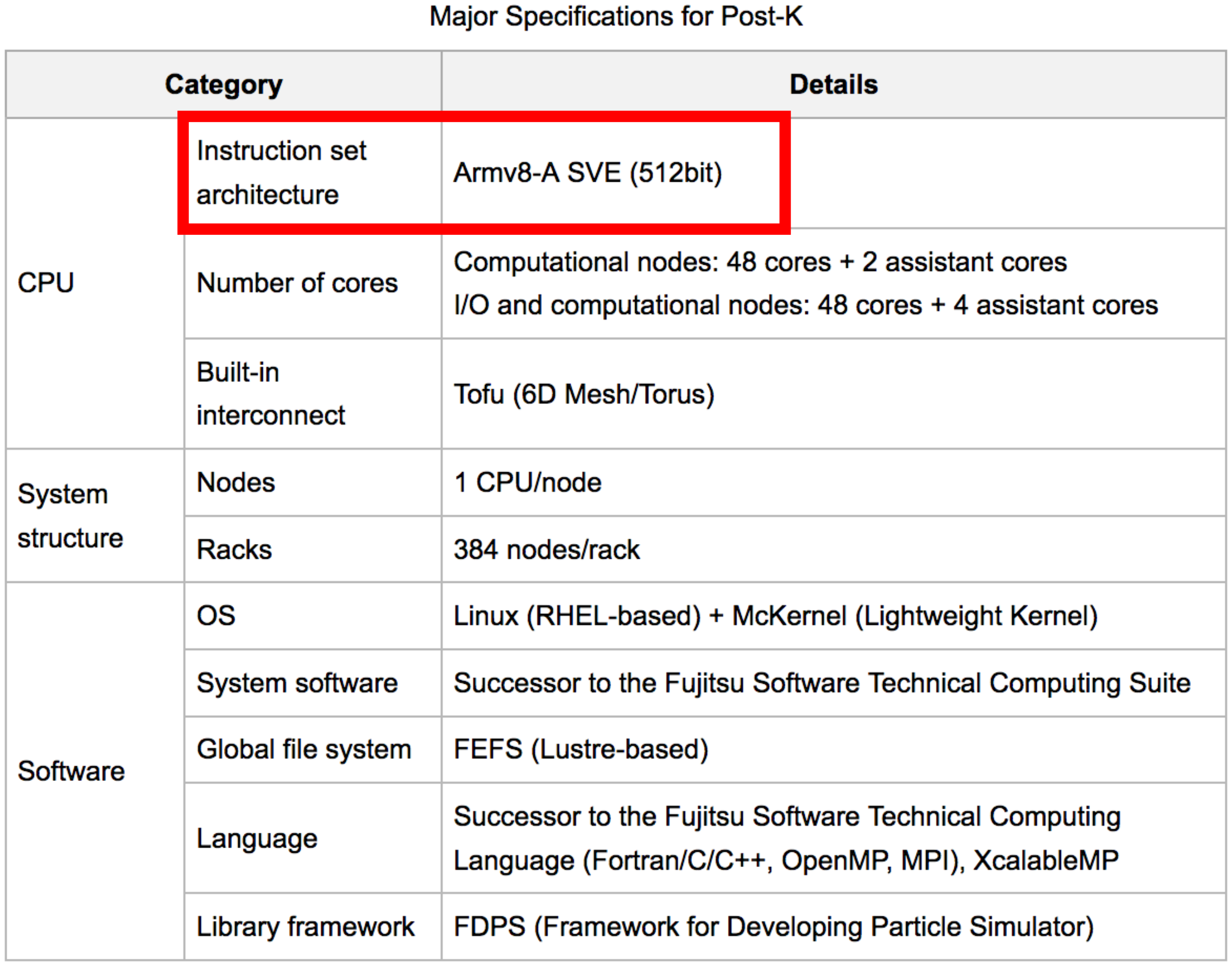
1



slides from Fujitsu

Post-K Prototype at ISC18





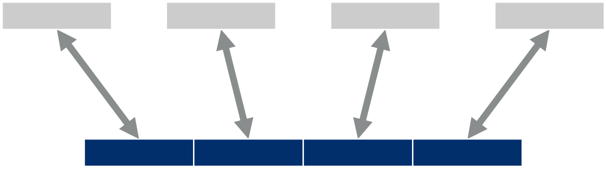
Major Specifications for Post-K

Category		Details
CPU	Instruction set architecture	Armv8-A SVE (512bit)
	Number of cores	Computational nodes: 48 cores + 2 assistant cores I/O and computational nodes: 48 cores + 4 assistant cores
	Built-in interconnect	Tofu (6D Mesh/Torus)
System structure	Nodes	1 CPU/node
	Racks	384 nodes/rack
Software	OS	Linux (RHEL-based) + McKernel (Lightweight Kernel)
	System software	Successor to the Fujitsu Software Technical Computing Suite
	Global file system	FEFS (Lustre-based)
	Language	Successor to the Fujitsu Software Technical Computing Language (Fortran/C/C++, OpenMP, MPI), XcalableMP
	Library framework	FDPS (Framework for Developing Particle Simulator)

slides from Fujitsu

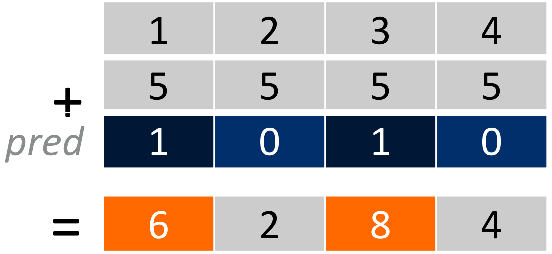
Introducing the Scalable Vector Extension (SVE)

A vector extension to the ARMv8-A architecture with some major new features:



Gather-load and scatter-store

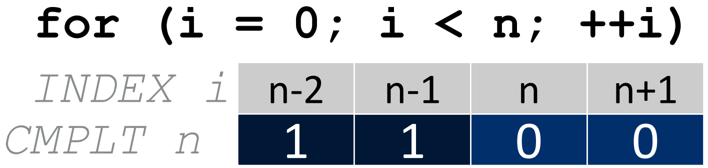
Loads a single register from several non-contiguous memory locations.



	1	2	3	4
+	5	5	5	5
pred	1	0	1	0
=	6	2	8	4

Per-lane predication

Operations work on individual lanes under control of a predicate register.



```
for (i = 0; i < n; ++i)
```

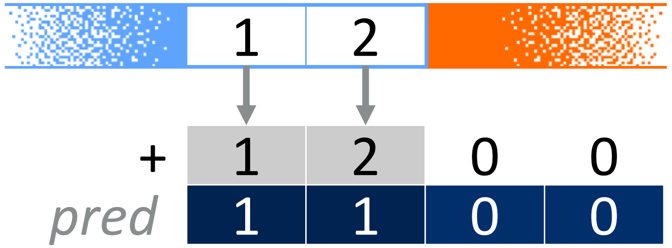
INDEX i	n-2	n-1	n	n+1
CMPLT n	1	1	0	0

Predicate-driven loop control and management

Eliminate scalar loop heads and tails by processing partial vectors.

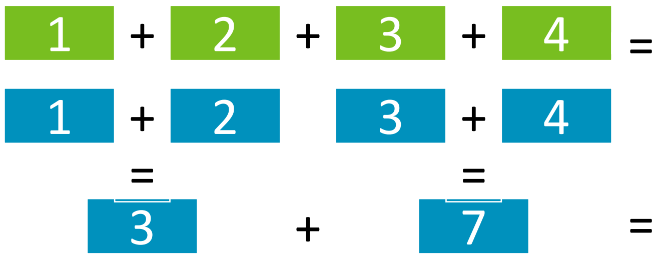
Introducing the Scalable Vector Extension (SVE)

A vector extension to the ARMv8-A architecture with some major new features:



Vector partitioning and software-managed speculation

First Faulting Load instructions allow memory accesses to cross into invalid pages.



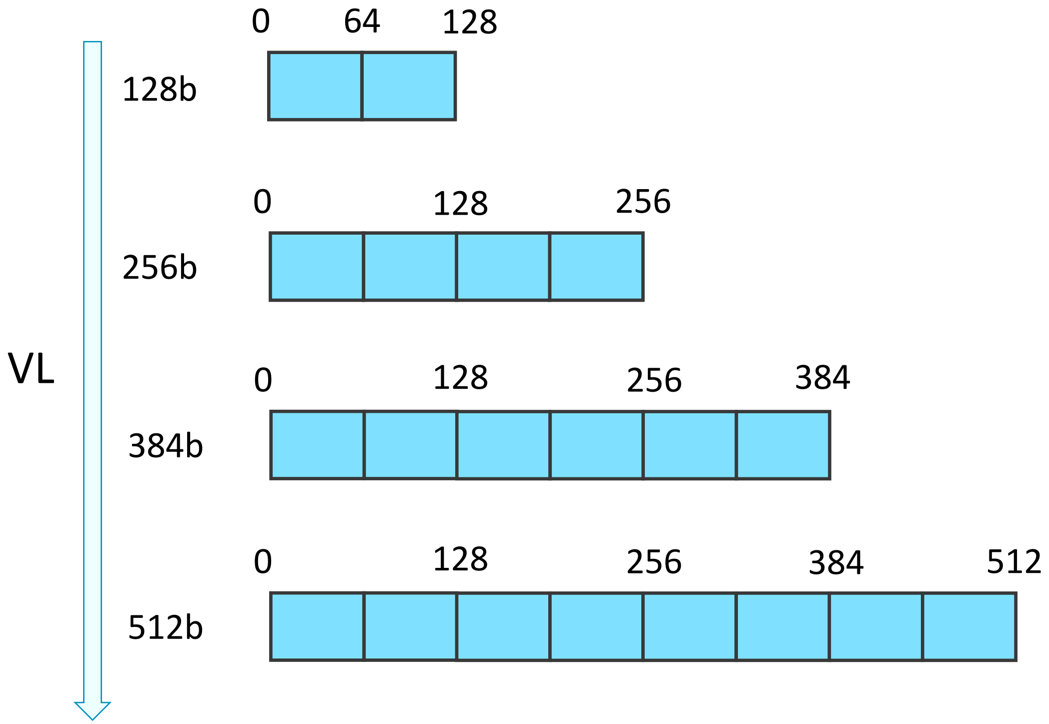
Extended floating-point horizontal reductions

In-order and tree-based reductions trade-off performance and repeatability.

What's the Vector Length?

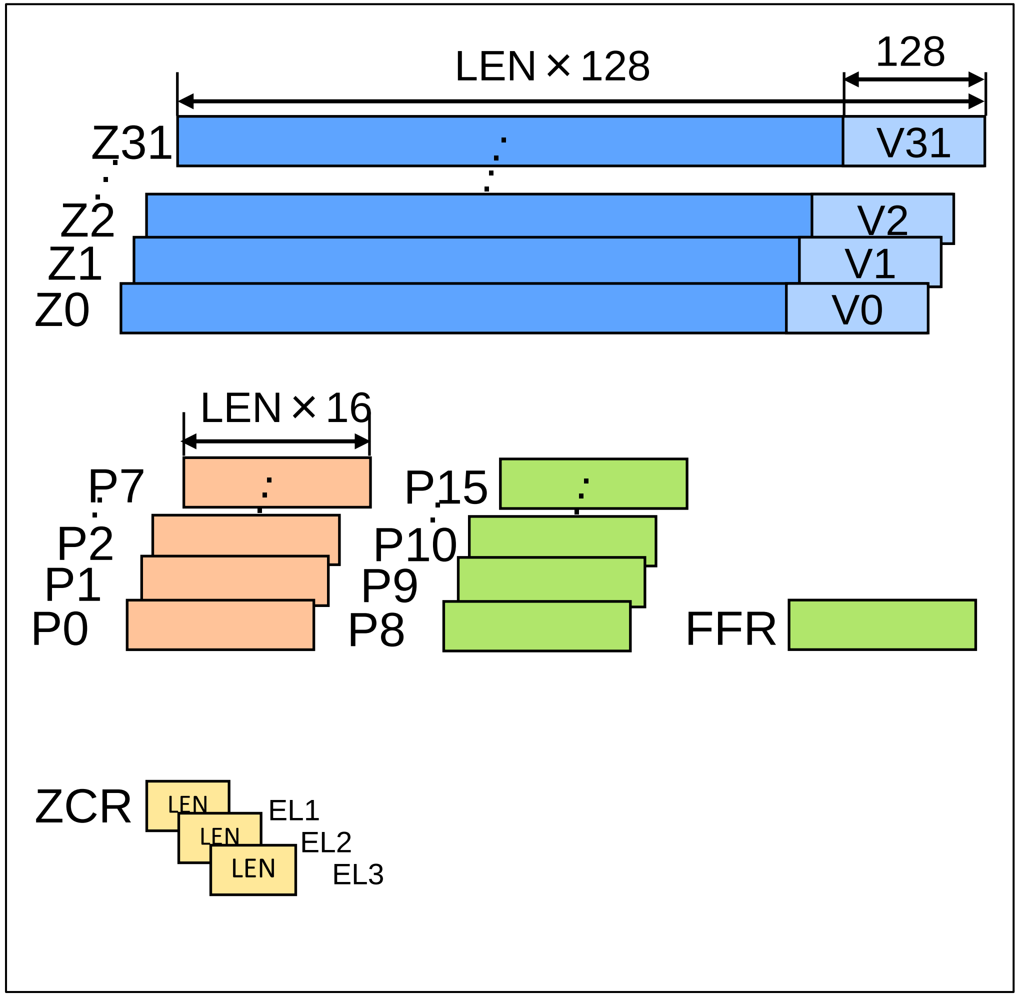
There is no preferred vector length

- Vector Length (VL) is the CPU implementor's choice, from 128 to 2048 bits, in increments of 128
- Adopting a Vector Length Agnostic (VLA) code generation style makes code portable across all possible vector lengths
- VLA is made possible by the per-lane predication, predicate-driven loop control, vector partitioning and software-managed speculation features of SVE
- No need to recompile, or to rewrite hand-coded SVE assembler or C intrinsics



SVE – Architectural State

- Scalable vector registers
  - Z0-Z31 extending NEON's V0-V31
    - DP & SP floating-point
    - 64, 32, 16 & 8-bit integer
- Scalable predicate registers
  - P0-P7 lane masks for ld/st/arith
  - P8-P15 for predicate manipulation
  - FFR *first fault register*
- Scalable vector control registers
  - ZCR_ELx vector length (LEN=1..16)
  - Exception / privilege level EL1 to EL3



SVE Visual Examples

daxpy (scalar)

```
void daxpy(double *x, double *y, double a, int n)
{
   for (int i = 0; i < n; i++) {
      y[i] = a * x[i] + y[i];
   }
}
```

```
// x0 = &x[0]
// x1 = &y[0]
// x2 = &a
// x3 = &n
daxpy_:
        ldrsw       x3, [x3]
        mov         x4, #0
        ldr         d0, [x2]
        b           .latch
.loop:
        ldr         d1, [x0, x4, lsl #3]
        ldr         d2, [x1, x4, lsl #3]
        fmadd       d2, d1, d0, d2
        str         d2, [x1, x4, lsl #3]
        add         x4, x4, #1
.latch:
        cmp         x4, x3
        b.lt        .loop
        ret
```

daxpy (SVE)

daxpy (scalar)

Loop fiberization: pulling multiple scalar iterations into a vector

```
daxpy_:
        ldrsw   x3, [x3]
        mov                 x4, #0
        whilelt p0.d, x4, x3
        ld1rd   z0.d, p0/z, [x2]
.loop:
        ld1d                z1.d, p0/z, [x0, x4, lsl #3]
        ld1d                z2.d, p0/z, [x1, x4, lsl #3]
        fmla                z2.d, p0/m, z1.d, z0.d
        st1d                z2.d, p0, [x1, x4, lsl #3]
        incd    x4
.latch:
        whilelt p0.d, x4, x3
        b.first .loop
        ret
```

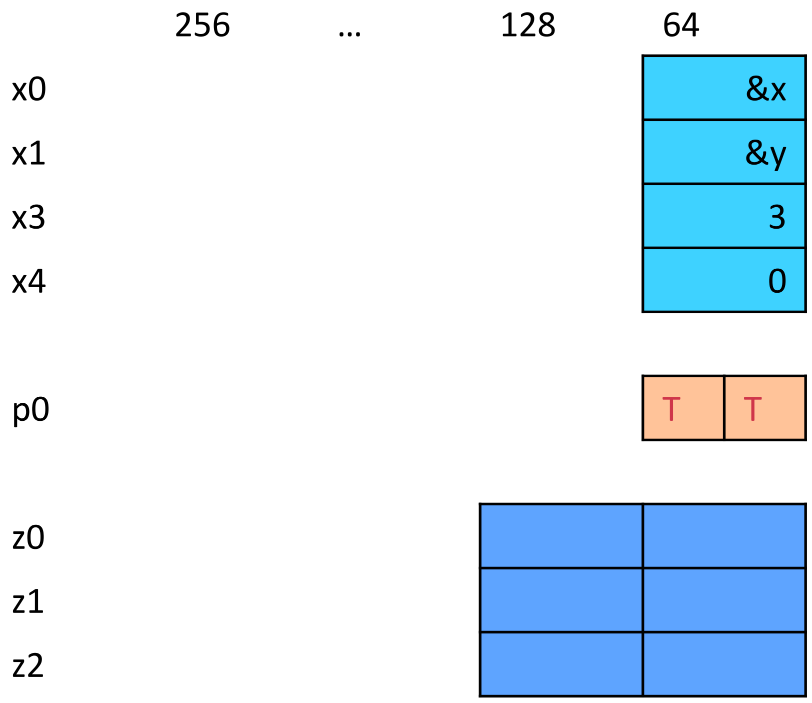
```
daxpy_:
        ldrsw                 x3, [x3]
        mov                   x4, #0
        ldr                   d0, [x2]
        b                     .latch
.loop:
        ldr                   d1, [x0, x4, lsl #3]
        ldr                   d2, [x1, x4, lsl #3]
        fmadd                 d2, d1, d0, d2
        str                   d2, [x1, x4, lsl #3]
        add                   x4, x4, #1
.latch:
        cmp                   x4, x3
        b.lt                  .loop
        ret
```

How do we handle the non-multiples of VL?
What happens at different vector lengths?

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

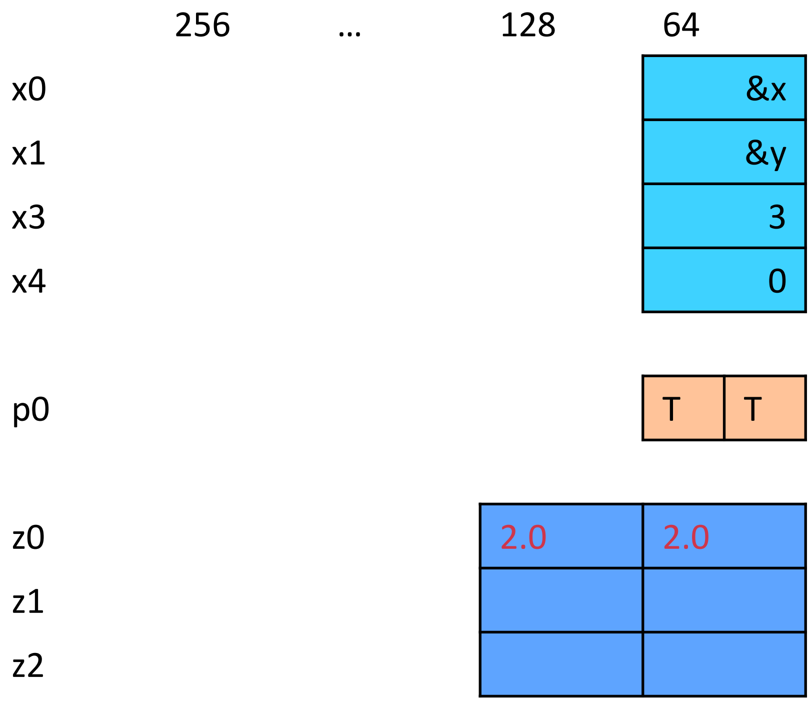


CYCLES 2

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```



256 … 128 64

x0 &x

x1 &y

x3 3

x4 0

p0 T T

z0 2.0 2.0

z1

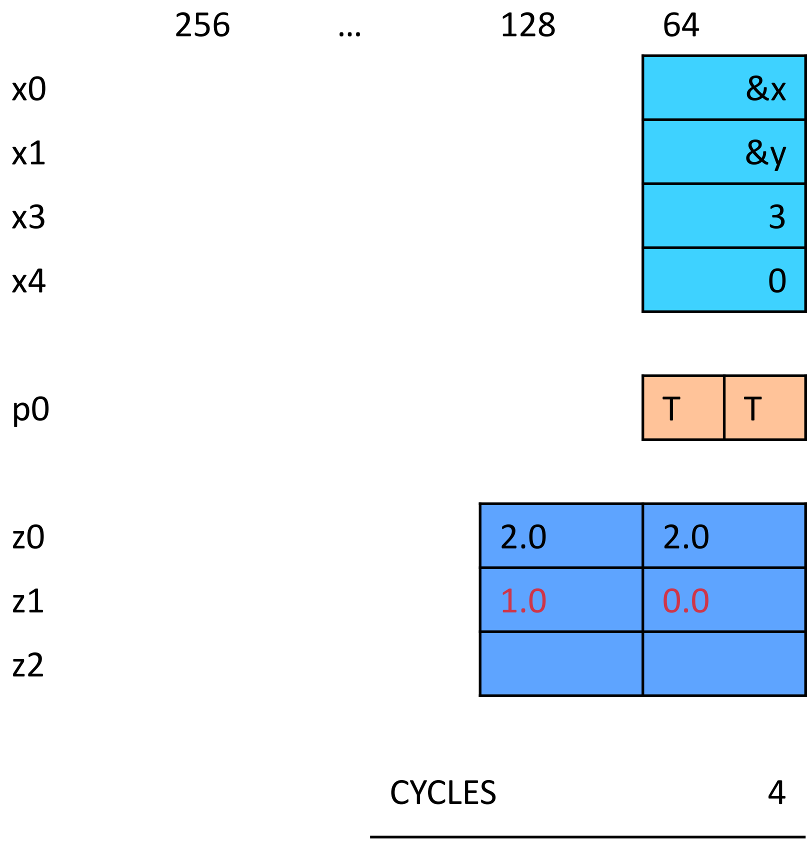
z2

CYCLES 3

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```



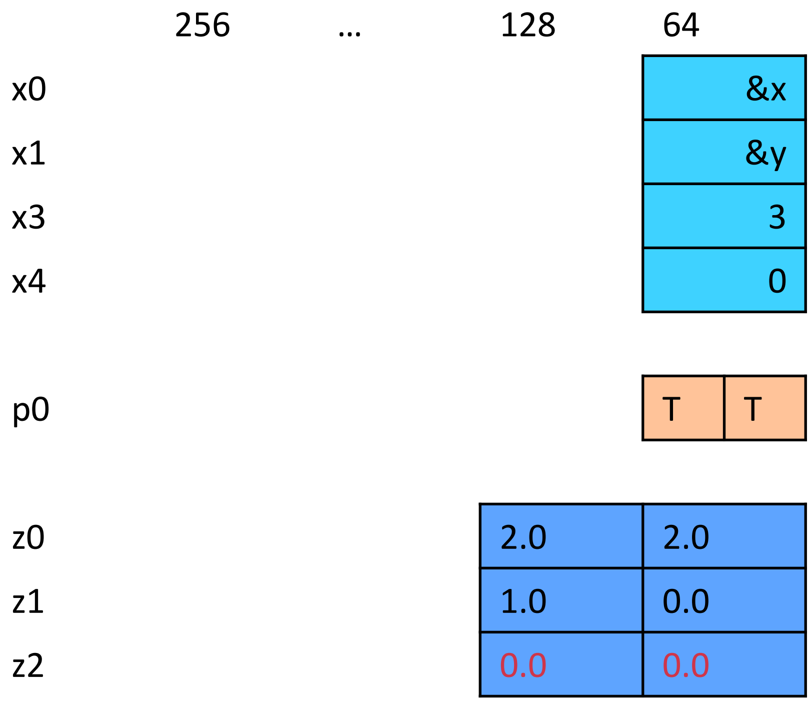
	256	...	128		64
x0					&x
x1					&y
x3					3
x4					0
p0				T	T
z0				2.0	2.0
z1				1.0	0.0
z2					

CYCLES 4

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw    x3, [x3]
    mov      x4, #0
    whilelt  p0.d, x4, x3
    ld1rd    z0.d, p0/z, [x2]
.loop:
    ld1d     z1.d, p0/z, [x0,x4,lsl #3]
    ld1d     z2.d, p0/z, [x1,x4,lsl #3]
    fmla     z2.d, p0/m, z1.d, z0.d
    st1d     z2.d, p0, [x1,x4,lsl #3]
    incd     x4
.latch:
    whilelt  p0.d, x4, x3
    b.first  .loop
    ret
```



	256	…	128	64
x0				&x
x1				&y
x3				3
x4				0
p0				T T
z0			2.0	2.0
z1			1.0	0.0
z2			0.0	0.0

CYCLES 5

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

	256	…	128	64
x0				&x
x1				&y
x3				3
x4				0
p0				T T
z0			2.0	2.0
z1			1.0	0.0
z2			2.0	0.0

CYCLES 6

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw    x3, [x3]
    mov      x4, #0
    whilelt  p0.d, x4, x3
    ld1rd    z0.d, p0/z, [x2]
.loop:
    ld1d     z1.d, p0/z, [x0,x4,lsl #3]
    ld1d     z2.d, p0/z, [x1,x4,lsl #3]
    fmla     z2.d, p0/m, z1.d, z0.d
    st1d     z2.d, p0, [x1,x4,lsl #3]
    incd     x4
.latch:
    whilelt  p0.d, x4, x3
    b.first  .loop
    ret
```

	256	...	128	64
x0				&x
x1				&y
x3				3
x4				0
p0				T T
z0			2.0	2.0
z1			1.0	0.0
z2			2.0	0.0

CYCLES 7

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

	256	…	128	64
x0				&x
x1				&y
x3				3
x4				2
p0				T T
z0			2.0	2.0
z1			1.0	0.0
z2			2.0	0.0

CYCLES 8

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

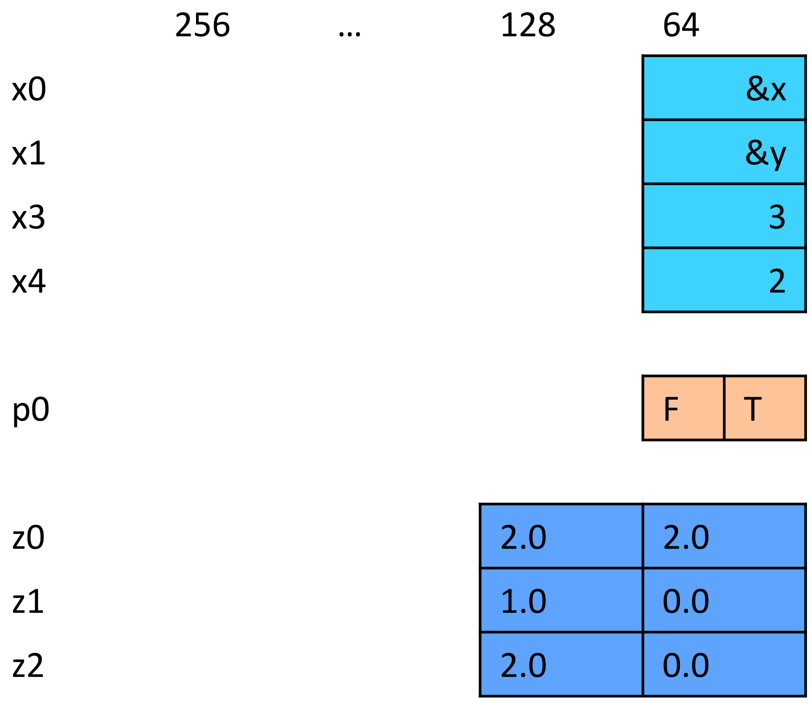
	256	...	128	64
x0				&x
x1				&y
x3				3
x4				2
p0				F T
z0			2.0	2.0
z1			1.0	0.0
z2			2.0	0.0

CYCLES 9

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw    x3, [x3]
    mov      x4, #0
    whilelt  p0.d, x4, x3
    ld1rd    z0.d, p0/z, [x2]
.loop:
    ld1d     z1.d, p0/z, [x0,x4,lsl #3]
    ld1d     z2.d, p0/z, [x1,x4,lsl #3]
    fmla     z2.d, p0/m, z1.d, z0.d
    st1d     z2.d, p0, [x1,x4,lsl #3]
    incd     x4
.latch:
    whilelt  p0.d, x4, x3
➡   b.first  .loop
    ret
```



256 … 128 64

x0 &x

x1 &y

x3 3

x4 2

p0 F T

z0 2.0 2.0

z1 1.0 0.0

z2 2.0 0.0

CYCLES 10

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw    x3, [x3]
    mov      x4, #0
    whilelt  p0.d, x4, x3
    ld1rd    z0.d, p0/z, [x2]
.loop:
    ld1d     z1.d, p0/z, [x0,x4,lsl #3]
    ld1d     z2.d, p0/z, [x1,x4,lsl #3]
    fmla     z2.d, p0/m, z1.d, z0.d
    st1d     z2.d, p0, [x1,x4,lsl #3]
    incd     x4
.latch:
    whilelt  p0.d, x4, x3
    b.first  .loop
    ret
```

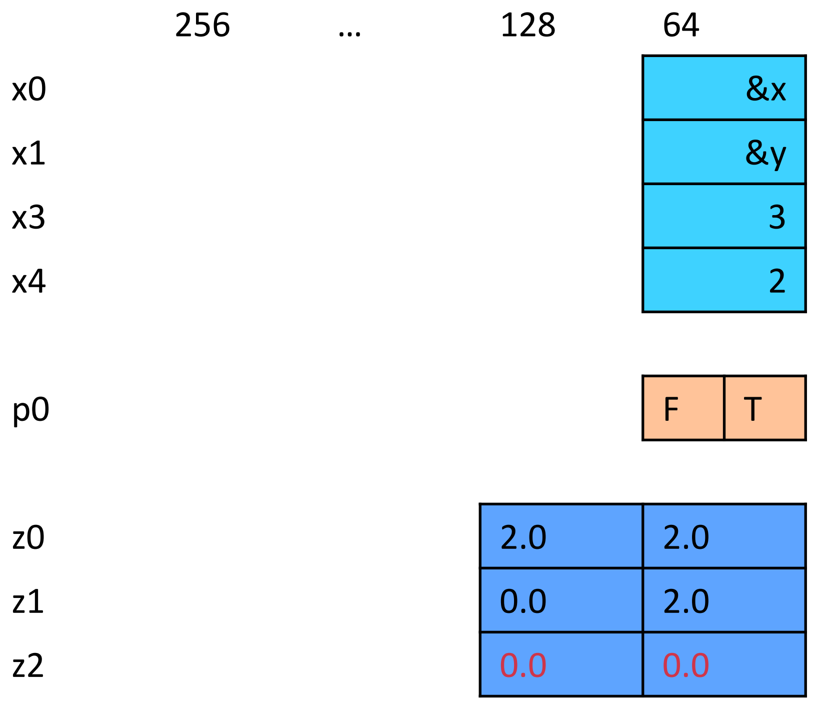
	256	…	128	64
x0				&x
x1				&y
x3				3
x4				2
p0				F T
z0			2.0	2.0
z1			0.0	2.0
z2			2.0	0.0

CYCLES	11

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```



CYCLES	12

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

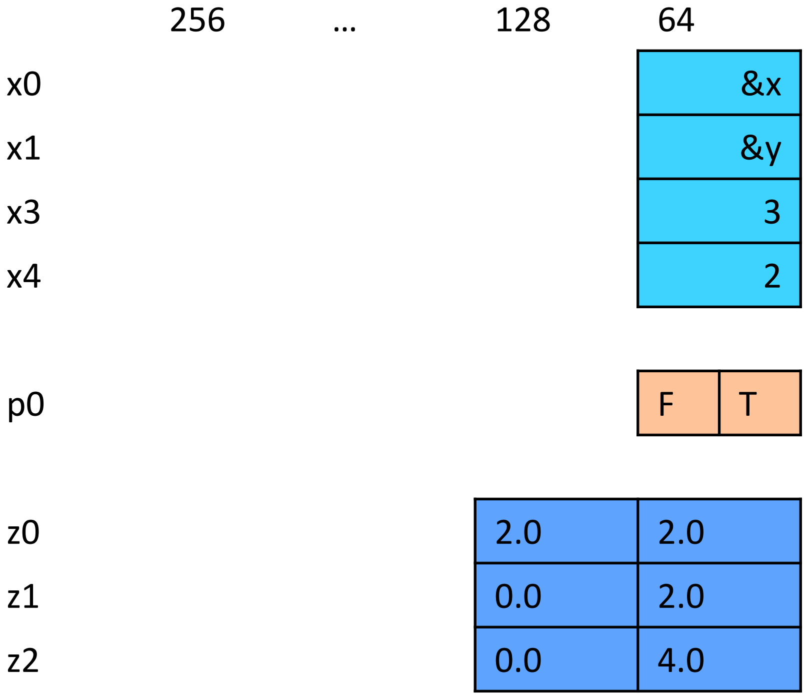
	256	...	128	64
x0				&x
x1				&y
x3				3
x4				2
p0				F T
z0			2.0	2.0
z1			0.0	2.0
z2			0.0	4.0

CYCLES 13

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```



	256	…	128	64
x0				&x
x1				&y
x3				3
x4				2
p0				F T
z0			2.0	2.0
z1			0.0	2.0
z2			0.0	4.0

CYCLES 14

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

	256	...	128	64
x0				&x
x1				&y
x3				3
x4				4
p0				F T
z0			2.0	2.0
z1			0.0	2.0
z2			0.0	4.0

CYCLES	15

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

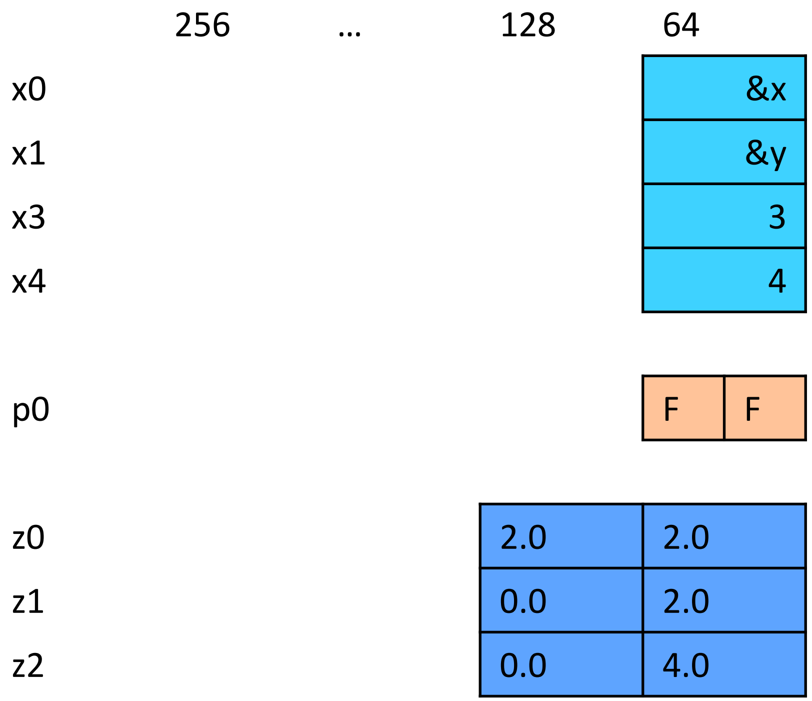
	256	...	128	64
x0				&x
x1				&y
x3				3
x4				4
p0				F F
z0			2.0	2.0
z1			0.0	2.0
z2			0.0	4.0

CYCLES	16

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```



256 … 128 64

x0 &x

x1 &y

x3 3

x4 4

p0 F F

z0 2.0 2.0

z1 0.0 2.0

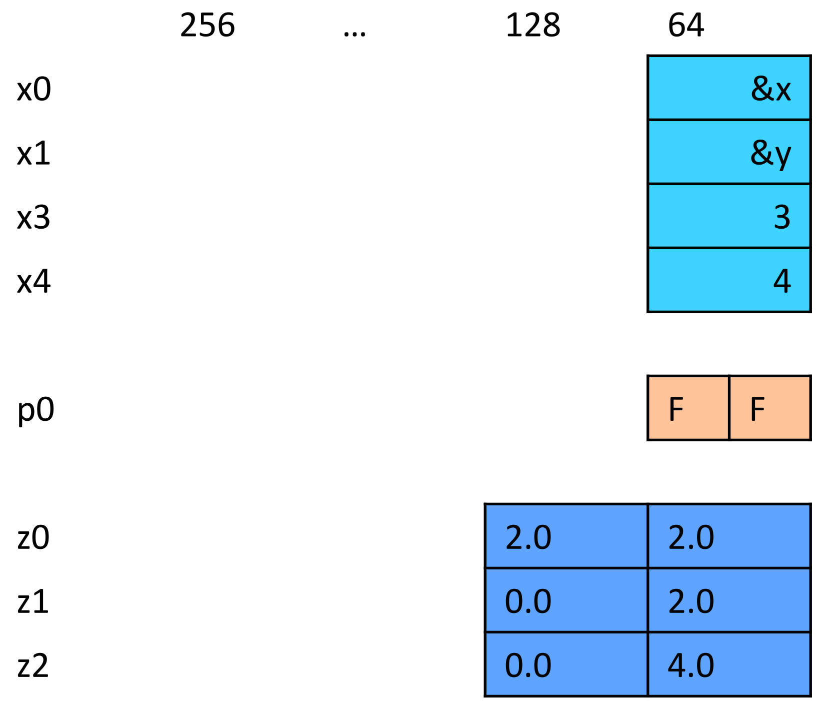
z2 0.0 4.0

CYCLES 17

daxpy (SVE – 128b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```



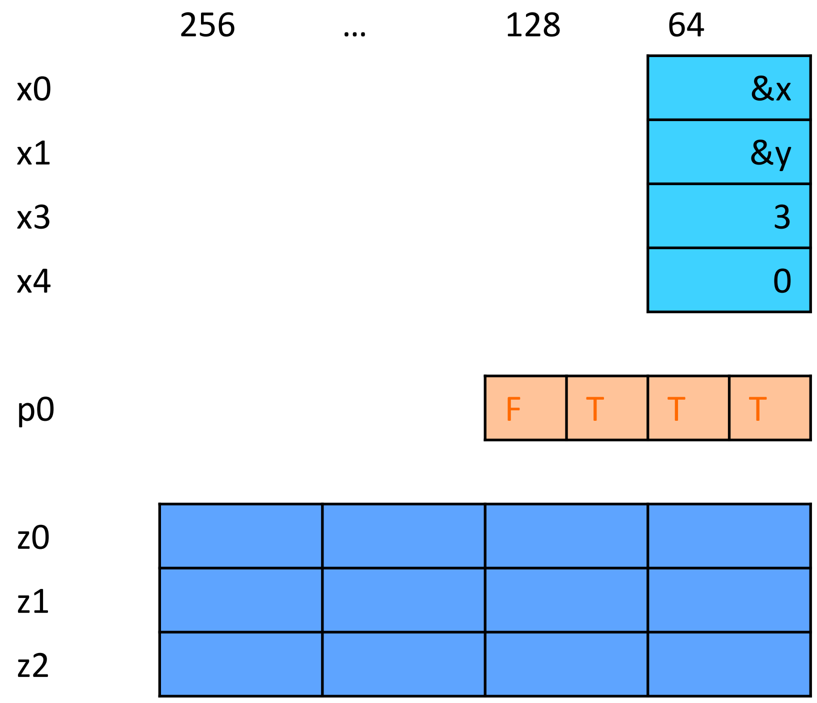
	256	…	128	64
x0				&x
x1				&y
x3				3
x4				4
p0			F	F
z0			2.0	2.0
z1			0.0	2.0
z2			0.0	4.0

CYCLES 18

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw    x3, [x3]
    mov      x4, #0
    whilelt  p0.d, x4, x3
    ld1rd    z0.d, p0/z, [x2]
.loop:
    ld1d     z1.d, p0/z, [x0,x4,lsl #3]
    ld1d     z2.d, p0/z, [x1,x4,lsl #3]
    fmla     z2.d, p0/m, z1.d, z0.d
    st1d     z2.d, p0, [x1,x4,lsl #3]
    incd     x4
.latch:
    whilelt  p0.d, x4, x3
    b.first  .loop
    ret
```

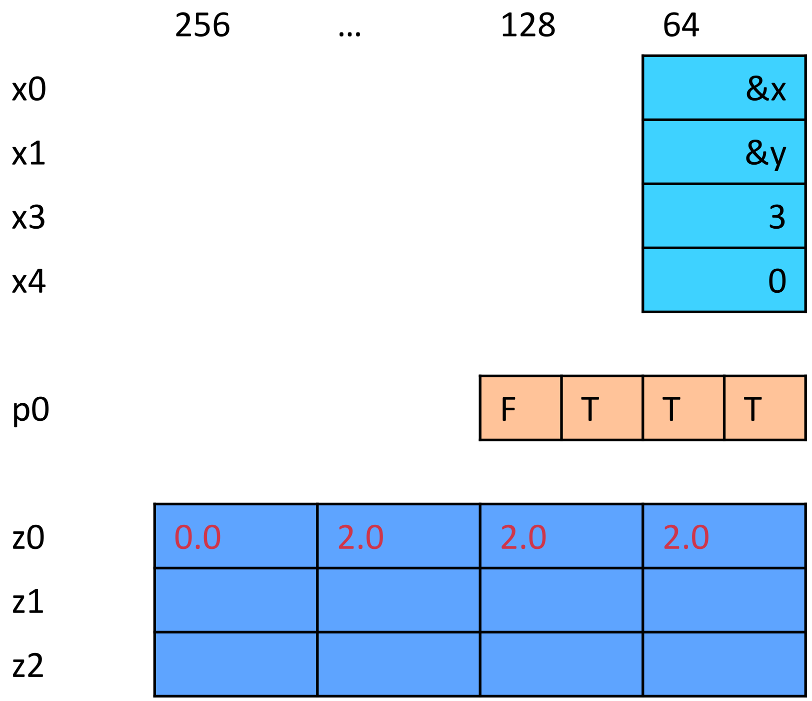


CYCLES 2

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

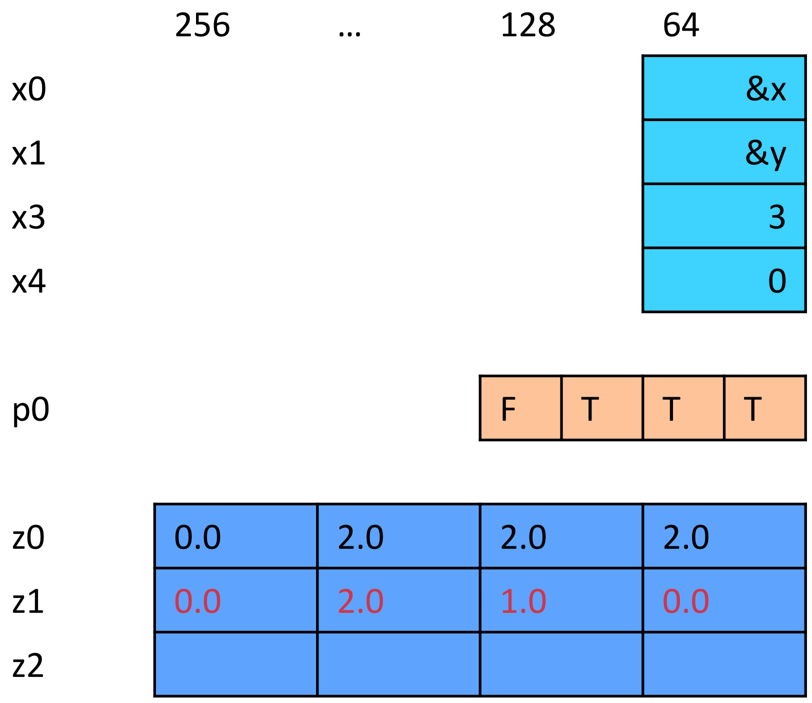


CYCLES 3

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
     ldrsw    x3, [x3]
     mov      x4, #0
     whilelt  p0.d, x4, x3
     ld1rd    z0.d, p0/z, [x2]
.loop:
     ld1d     z1.d, p0/z, [x0,x4,lsl #3]
     ld1d     z2.d, p0/z, [x1,x4,lsl #3]
     fmla     z2.d, p0/m, z1.d, z0.d
     st1d     z2.d, p0, [x1,x4,lsl #3]
     incd     x4
.latch:
     whilelt  p0.d, x4, x3
     b.first  .loop
     ret
```



	256	…	128	64
x0				&x
x1				&y
x3				3
x4				0
p0	F	T	T	T
z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2				

CYCLES 4

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

	256	…	128	64
x0				&x
x1				&y
x3				3
x4				0

p0	F	T	T	T

z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	0.0	0.0	0.0

CYCLES 5

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

	256	…	128	64
x0				&x
x1				&y
x3				3
x4				0

p0	F	T	T	T

z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	4.0	2.0	0.0

CYCLES 6

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

	256	...	128	64
x0				&x
x1				&y
x3				3
x4				0
p0	F	T	T	T
z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	4.0	2.0	0.0

CYCLES 7

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

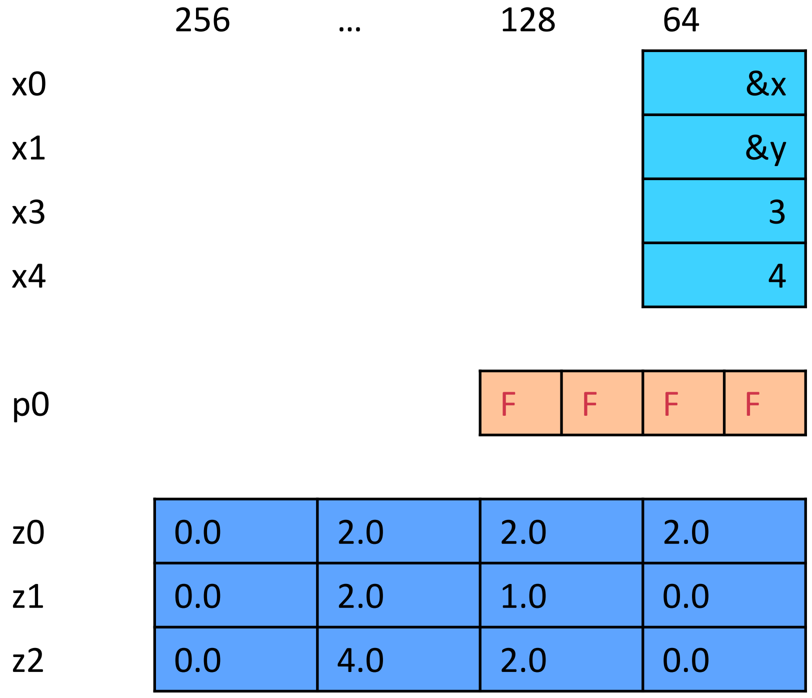
	256	…	128	64
x0				&x
x1				&y
x3				3
x4				4
p0	F	T	T	T
z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	4.0	2.0	0.0

CYCLES	8

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```



	256	...	128	64
x0				&x
x1				&y
x3				3
x4				4
p0			F F	F F
z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	4.0	2.0	0.0

CYCLES 9

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

	256	...	128	64
x0				&x
x1				&y
x3				3
x4				4

p0	F	F	F	F

z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	4.0	2.0	0.0

CYCLES	10

daxpy (SVE – 256b)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

	256	…	128	64
x0				&x
x1				&y
x3				3
x4				4
z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	4.0	2.0	0.0

p0	F	F	F	F

CYCLES 11

Fault-tolerant Speculative Vectorization

Some loops have dynamic exit conditions that prevent vectorization

- E.g., the loop breaks on a particular value of the traversed array

‘b’	‘s’	‘c’	‘-’	‘u’	‘p’	‘c’	0	0
✓	✓	✓	✓	✓	✓	✓	✓	X

The access to unallocated space does not trap if it is not the first element

- Faulting elements are stored in the first-fault register (FFR)
- Subsequent instructions are predicated using the FFR information to operate only on successful element accesses

strlen (scalar)

```
int strlen(const char *s) {
   const char *e = s;
   while (*e) e++;
   return e - s;
}
```

```
// x0 = s
strlen:
    mov     x1, x0              // e=s
.loop:
    ldrb    x2, [x1],#1         // x2=*e++
    cbnz    x2, .loop           // while(*e)
.done:
    sub     x0, x1, x0          // e-s
    sub     x0, x0, #1          // return e-s-1
    ret
```

strlen (SVE)

```
strlen:
    mov       x1, x0
    ptrue     p0.b
.loop:
    setffr
    ldff1b    z0.b, p0/z, [x1]
    rdffr     p1.b, p0/z
    cmpeq     p2.b, p1/z, z0.b, #0
    brkbs     p2.b, p1/z, p2.b
    incp      x1, p2.b
    b.last    .loop
    sub       x0, x1, x0
    ret
```

Suboptimal implementation

strlen (scalar)

```
// x0 = s
strlen:
    mov       x1, x0               // e=s
.loop:
    ldrb      x2, [x1],#1          // x2=*e++
    cbnz      x2, .loop           // while(*e)
.done:
    sub       x0, x1, x0           // e-s
    sub       x0, x0, #1           // return e-s-1
    ret
```

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
     mov      x1, x0
     ptrue    p0.b
.loop:
     setffr
     ldff1b   z0.b, p0/z, [x1]
     rdffr    p1.b, p0/z
     cmpeq    p2.b, p1/z, z0.b, #0
     brkbs    p2.b, p1/z, p2.b
     incp     x1, p2.b
     b.last   .loop
     sub      x0, x1, x0
     ret
```

128 … 64 … 32 16

x0	0	s
x1	0	

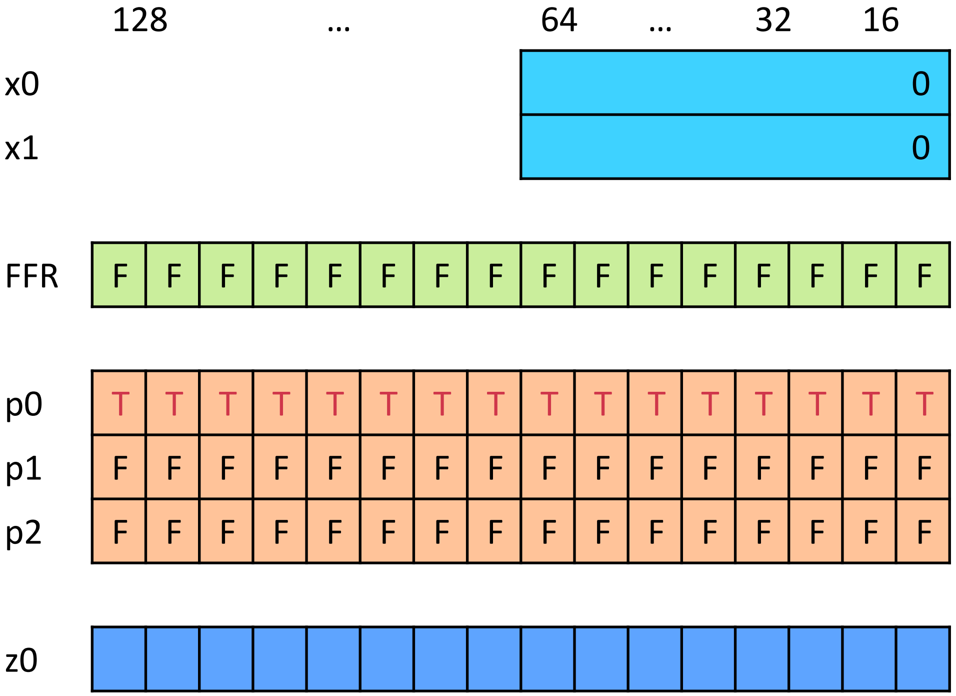
FFR	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
p0	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
p1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
p2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
z0																

CYCLES 0

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
      mov       x1, x0
      ptrue     p0.b
.loop:
      setffr
      ldff1b    z0.b, p0/z, [x1]
      rdffr     p1.b, p0/z
      cmpeq     p2.b, p1/z, z0.b, #0
      brkbs     p2.b, p1/z, p2.b
      incp      x1, p2.b
      b.last    .loop
      sub       x0, x1, x0
      ret
```



128 ... 64 ... 32 16

x0 0

x1 0

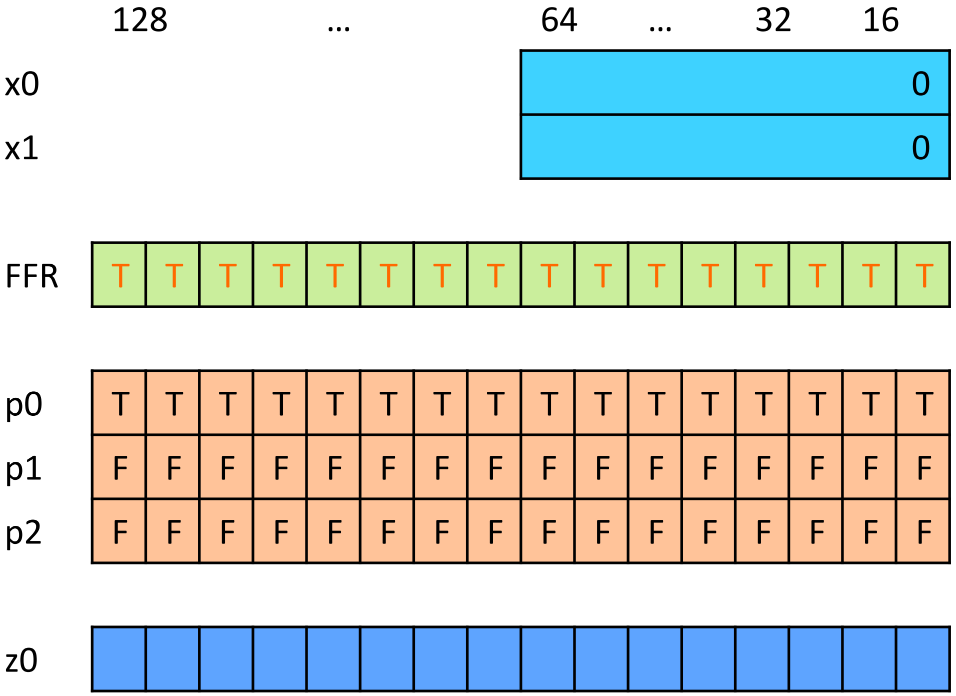
FFR	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
p2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
z0																

CYCLES 1

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
     mov      x1, x0
     ptrue    p0.b
.loop:
     setffr
     ldff1b   z0.b, p0/z, [x1]
     rdffr    p1.b, p0/z
     cmpeq    p2.b, p1/z, z0.b, #0
     brkbs    p2.b, p1/z, p2.b
     incp     x1, p2.b
     b.last   .loop
     sub      x0, x1, x0
     ret
```



x0	0
x1	0

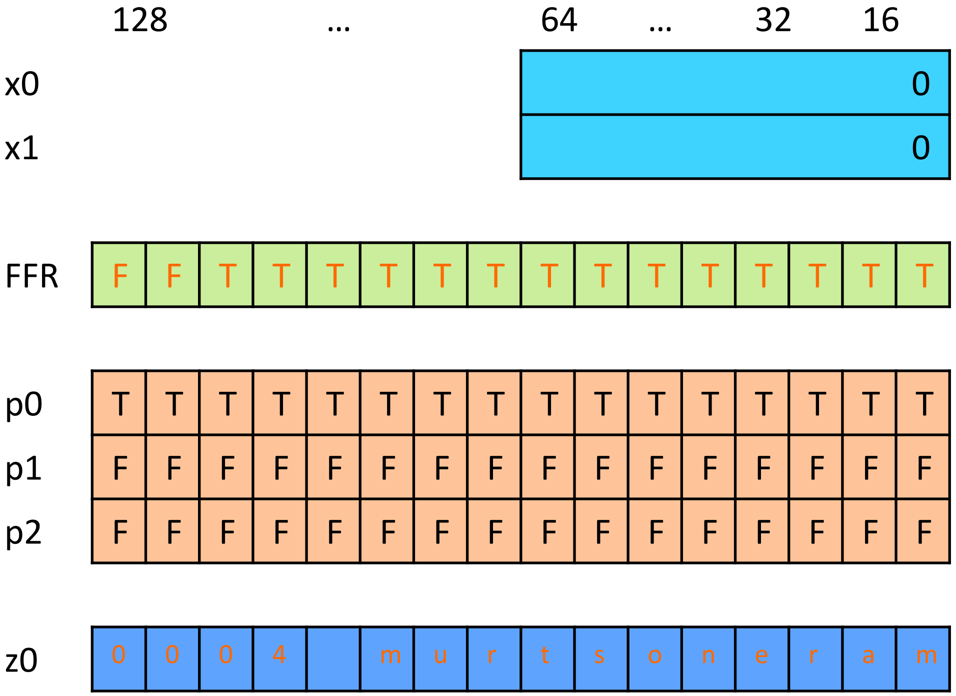
FFR	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
p2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
z0																

CYCLES 2

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’
	X	X														

```
strlen:
    mov      x1, x0
    ptrue    p0.b
.loop:
    setffr
⇨   ldff1b   z0.b, p0/z, [x1]
    rdffr    p1.b, p0/z
    cmpeq    p2.b, p1/z, z0.b, #0
    brkbs    p2.b, p1/z, p2.b
    incp     x1, p2.b
    b.last   .loop
    sub      x0, x1, x0
    ret
```



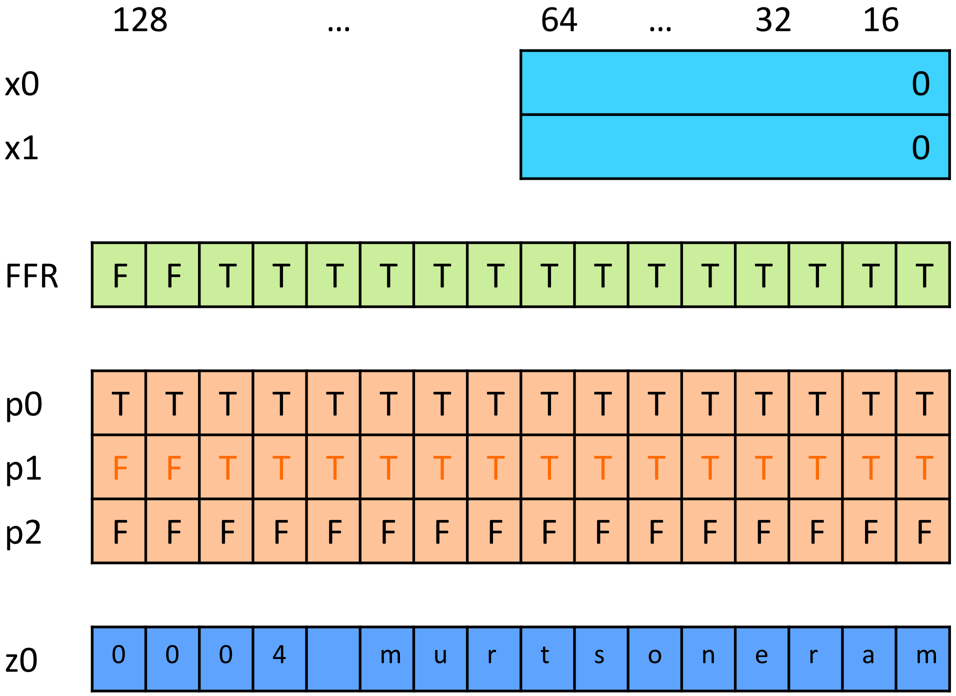
x0	0															
x1	0															
FFR	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
p2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
z0	0	0	0	4		m	u	r	t	s	o	n	e	r	a	m

CYCLES 3

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
     mov      x1, x0
     ptrue    p0.b
.loop:
     setffr
     ldff1b   z0.b, p0/z, [x1]
     rdffr    p1.b, p0/z
     cmpeq    p2.b, p1/z, z0.b, #0
     brkbs    p2.b, p1/z, p2.b
     incp     x1, p2.b
     b.last   .loop
     sub      x0, x1, x0
     ret
```



x0	0
x1	0

FFR	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
z0	0	0	0	4		m	u	r	t	s	o	n	e	r	a	m

CYCLES 4

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
     mov       x1, x0
     ptrue     p0.b
.loop:
     setffr
     ldff1b    z0.b, p0/z, [x1]
     rdffr     p1.b, p0/z
     cmpeq     p2.b, p1/z, z0.b, #0
     brkbs     p2.b, p1/z, p2.b
     incp      x1, p2.b
     b.last    .loop
     sub       x0, x1, x0
     ret
```

	128	...	64	...	32	16
x0						0
x1						0

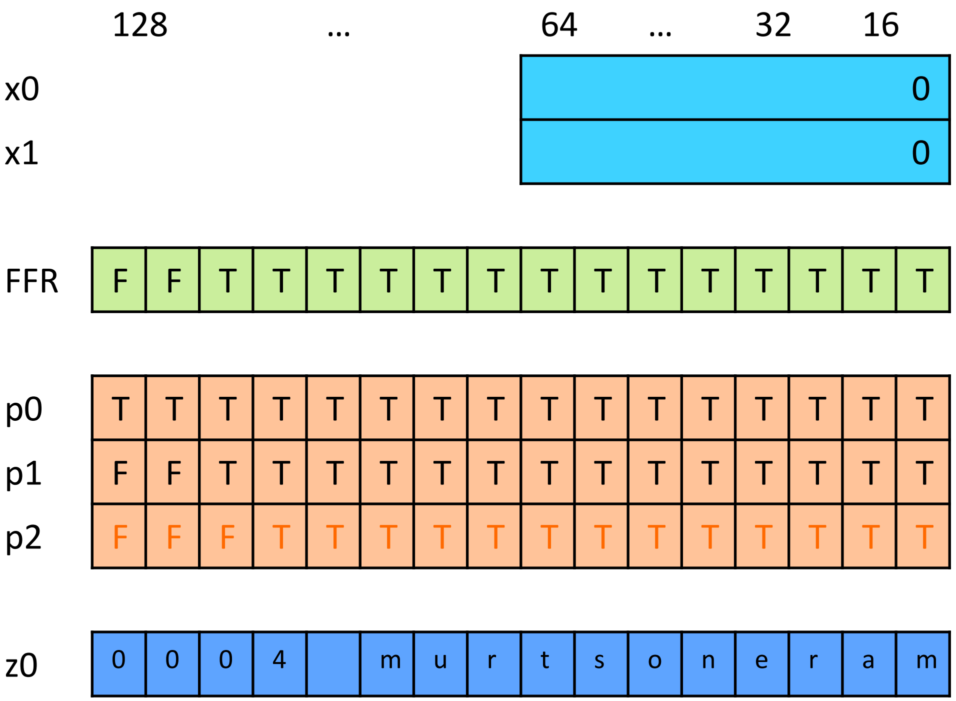
FFR	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p2	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
z0	0	0	0	4		m	u	r	t	s	o	n	e	r	a	m

CYCLES 5

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
    mov      x1, x0
    ptrue    p0.b
.loop:
    setffr
    ldff1b   z0.b, p0/z, [x1]
    rdffr    p1.b, p0/z
    cmpeq    p2.b, p1/z, z0.b, #0
    brkbs    p2.b, p1/z, p2.b
    incp     x1, p2.b
    b.last   .loop
    sub      x0, x1, x0
    ret
```



x0		0
x1		0

FFR	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p2	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T
z0	0	0	0	4		m	u	r	t	s	o	n	e	r	a	m

CYCLES 6

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
    mov      x1, x0
    ptrue    p0.b
.loop:
    setffr
    ldff1b   z0.b, p0/z, [x1]
    rdffr    p1.b, p0/z
    cmpeq    p2.b, p1/z, z0.b, #0
    brkbs    p2.b, p1/z, p2.b
    incp     x1, p2.b
    b.last   .loop
    sub      x0, x1, x0
    ret
```

128 ... 64 ... 32 16

x0	0
x1	13

FFR	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p2	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T
z0	0	0	0	4		m	u	r	t	s	o	n	e	r	a	m

CYCLES 7

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
     mov      x1, x0
     ptrue    p0.b
.loop:
     setffr
     ldff1b   z0.b, p0/z, [x1]
     rdffr    p1.b, p0/z
     cmpeq    p2.b, p1/z, z0.b, #0
     brkbs    p2.b, p1/z, p2.b
     incp     x1, p2.b
     b.last   .loop
     sub      x0, x1, x0
     ret
```

128 … 64 … 32 16

x0	0
x1	13

FFR	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T

p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p2	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T

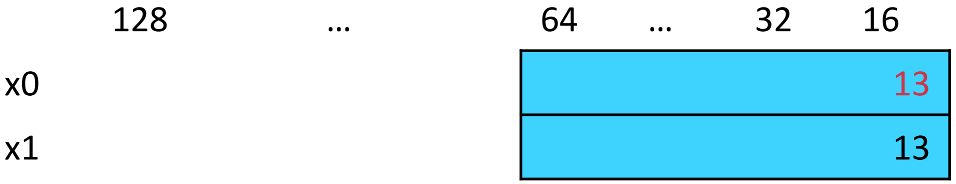
z0	0	0	0	4		m	u	r	t	s	o	n	e	r	a	m

CYCLES	8

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
        mov      x1, x0
        ptrue    p0.b
.loop:
        setffr
        ldff1b   z0.b, p0/z, [x1]
        rdffr    p1.b, p0/z
        cmpeq    p2.b, p1/z, z0.b, #0
        brkbs    p2.b, p1/z, p2.b
        incp     x1, p2.b
        b.last   .loop
        sub      x0, x1, x0
        ret
```



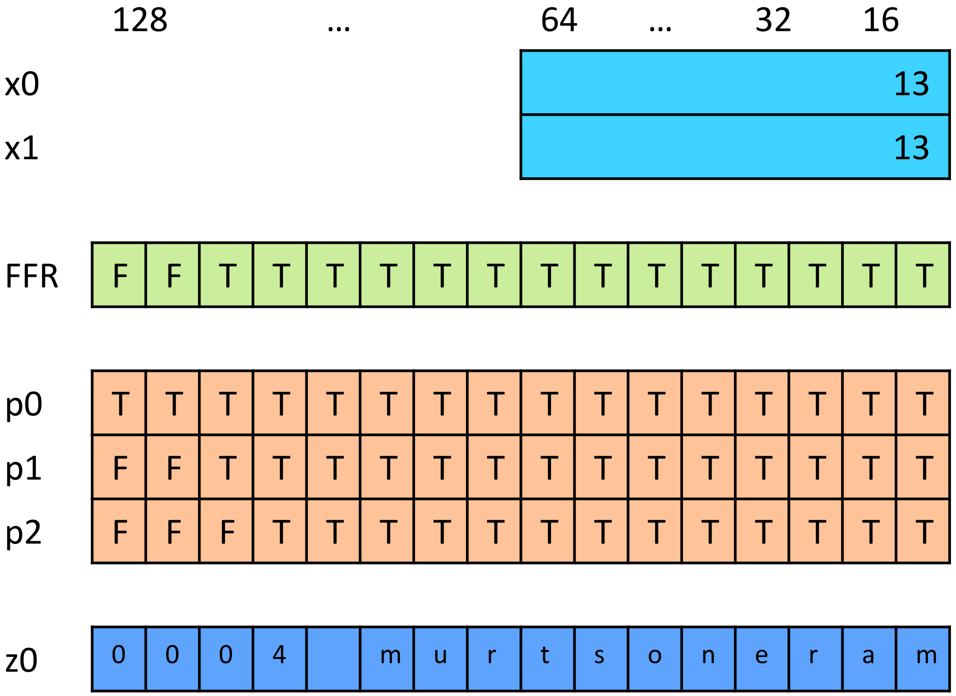
FFR	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p2	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T
z0	0	0	0	4		m	u	r	t	s	o	n	e	r	a	m

CYCLES 9

strlen (SVE)

Arrays	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s[]	0	0	0	‘4’	‘ ’	‘m’	‘u’	‘r’	‘t’	‘s’	‘o ’	‘n’	‘ e ’	‘r’	‘a’	‘m’

```
strlen:
    mov     x1, x0
    ptrue   p0.b
.loop:
    setffr
    ldff1b  z0.b, p0/z, [x1]
    rdffr   p1.b, p0/z
    cmpeq   p2.b, p1/z, z0.b, #0
    brkbs   p2.b, p1/z, p2.b
    incp    x1, p2.b
    b.last  .loop
    sub     x0, x1, x0
    ret
```



128 … 64 … 32 16

x0	13
x1	13

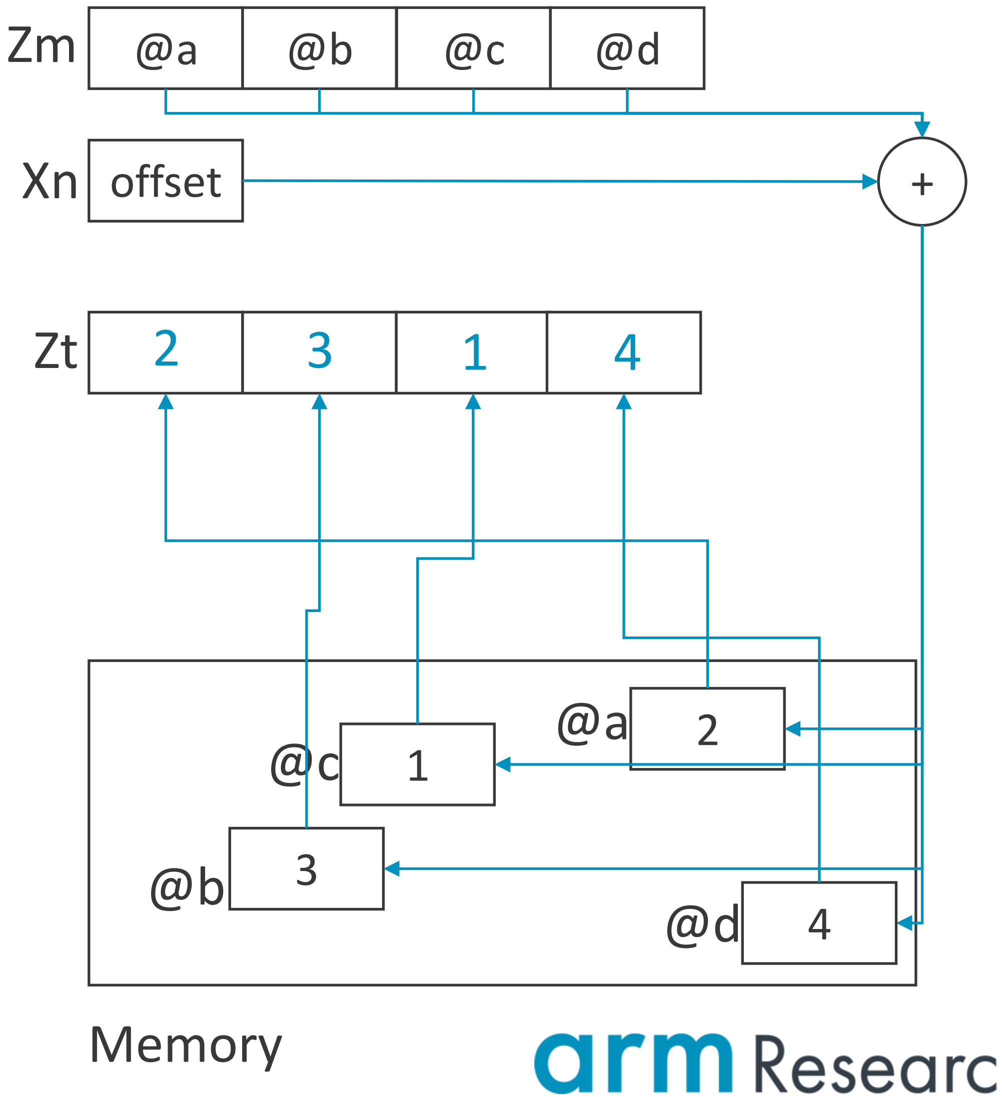
FFR	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p1	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
p2	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T
z0	0	0	0	4		m	u	r	t	s	o	n	e	r	a	m

CYCLES 10

Gather-Load & Scatter-Store

Gather/Scatter Operations are Good and Evil

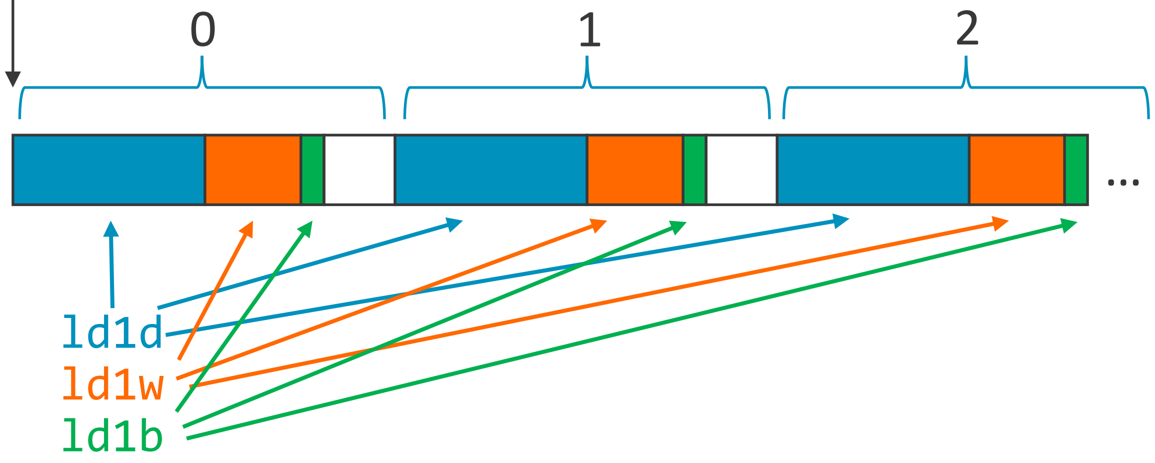
- Enable vectorization of codes with non-adjacent accesses on adjacent lanes
- Examples:
  - Outer loop vectorization
  - Strided accesses (larger than +1)
  - Random accesses
- Performance implementation dependent
  - Worst case one separate access per element
- `LD1D <Zt>.D, Ps/Z [<Xn>, <Zm>.D]`



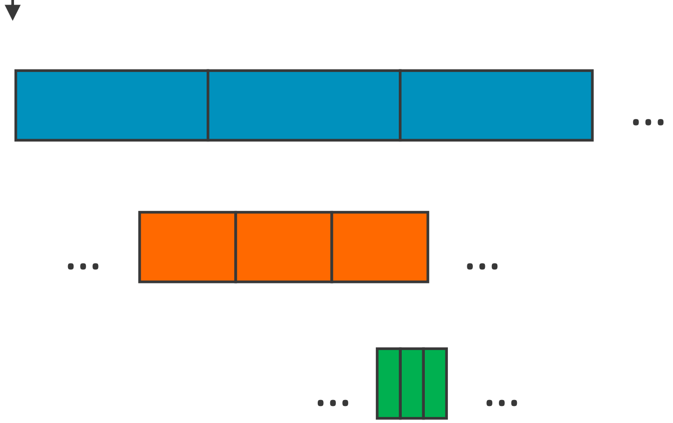
arm Research

Array of Structures vs. Structure of Arrays

```
typedef struct {
    uin64_t num_projects;
    float   caffeine;
    bool    cule_nmerengue;
} HPCProgrammer_t;

HPCProgrammer_t programmers[N];
```



```
typedef struct {
    uin64_t num_projects[N];
    float   caffeine[N];
    bool    cule_nmerengue[N];
} HPCProgrammer_t;

HPCProgrammer_t programmers;
```



Non-temporal Loads & Stores

SVE Non-Temporal Vector Instructions

- `LDNT1D { <Zt1>.D }, <Pglo>/Z, [<Xn|SP>, <Xm>, LSL #3]`
- `STNT1D { <Zt1>.D }, <Pglo>, [<Xn|SP>{, #<simm4>, MUL VL}]`

From the Arm ARM (Architecture Reference Manual):

Non-temporal contiguous load and stores include a ***hint to the memory system*** *that this is a "streaming" access, and the memory locations* ***are not expected to be accessed again soon*** *so do not need to be retained in local caches.*

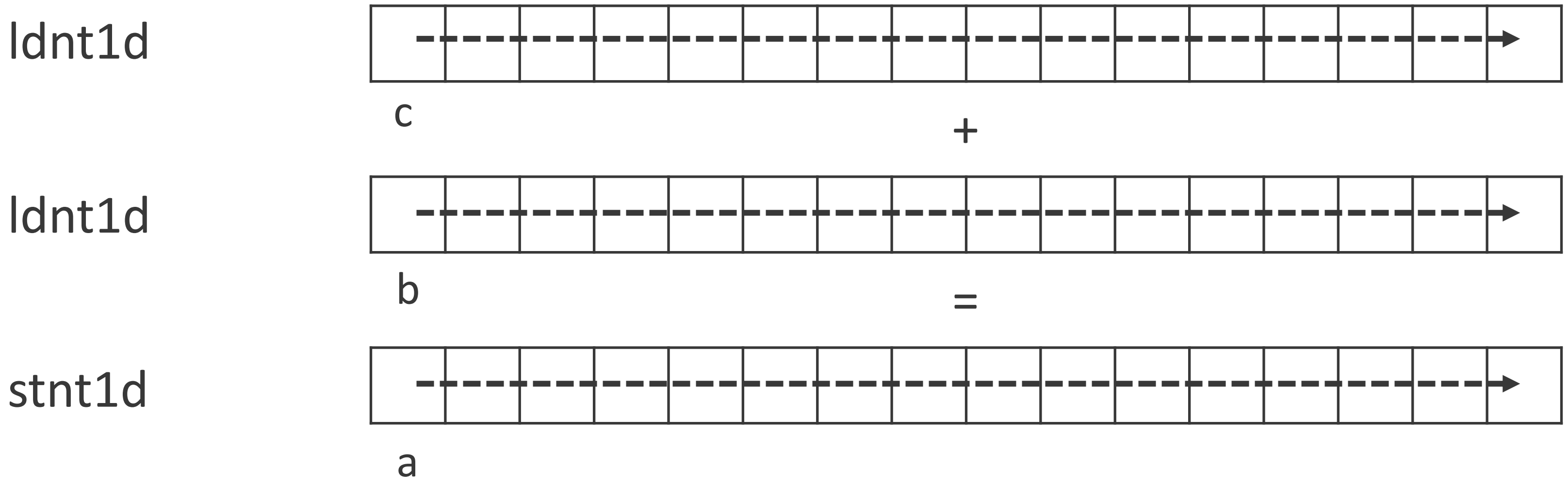
Being Non-temporal is Not Enough

Vector Addition

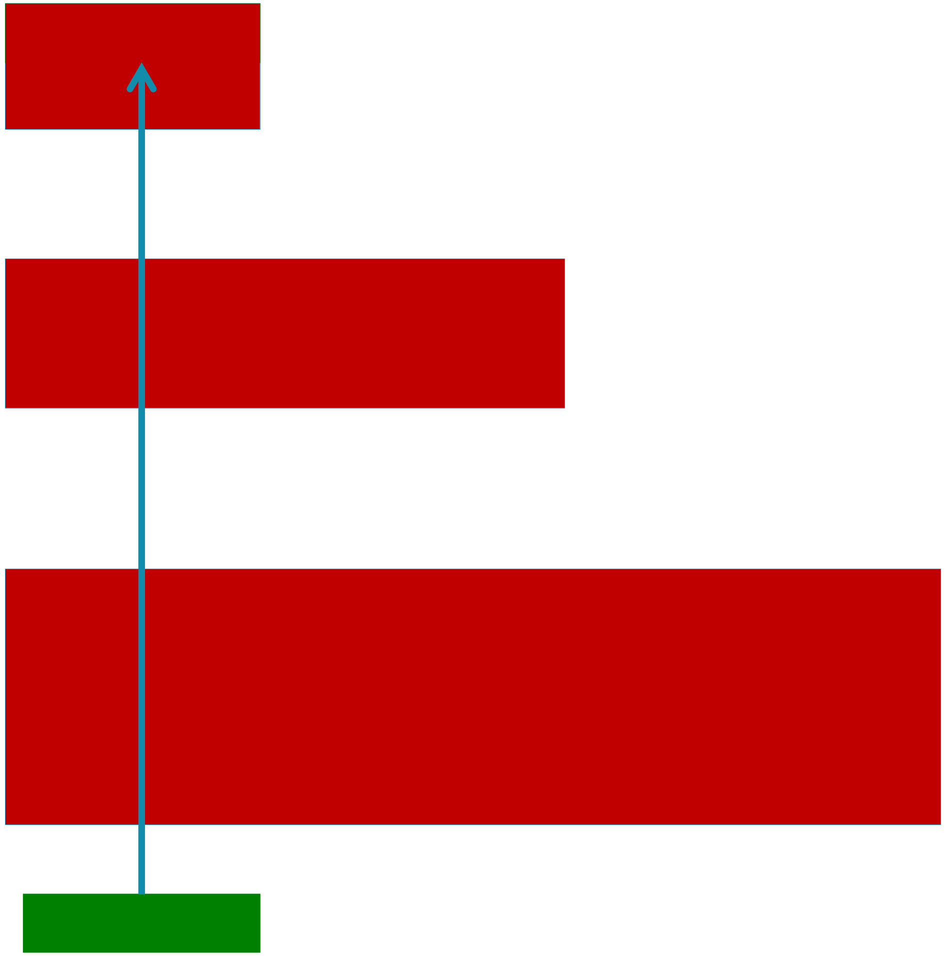
```
for (i=0; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

No benefit if all accesses are temporal

Target to leave space for *temporal* accesses

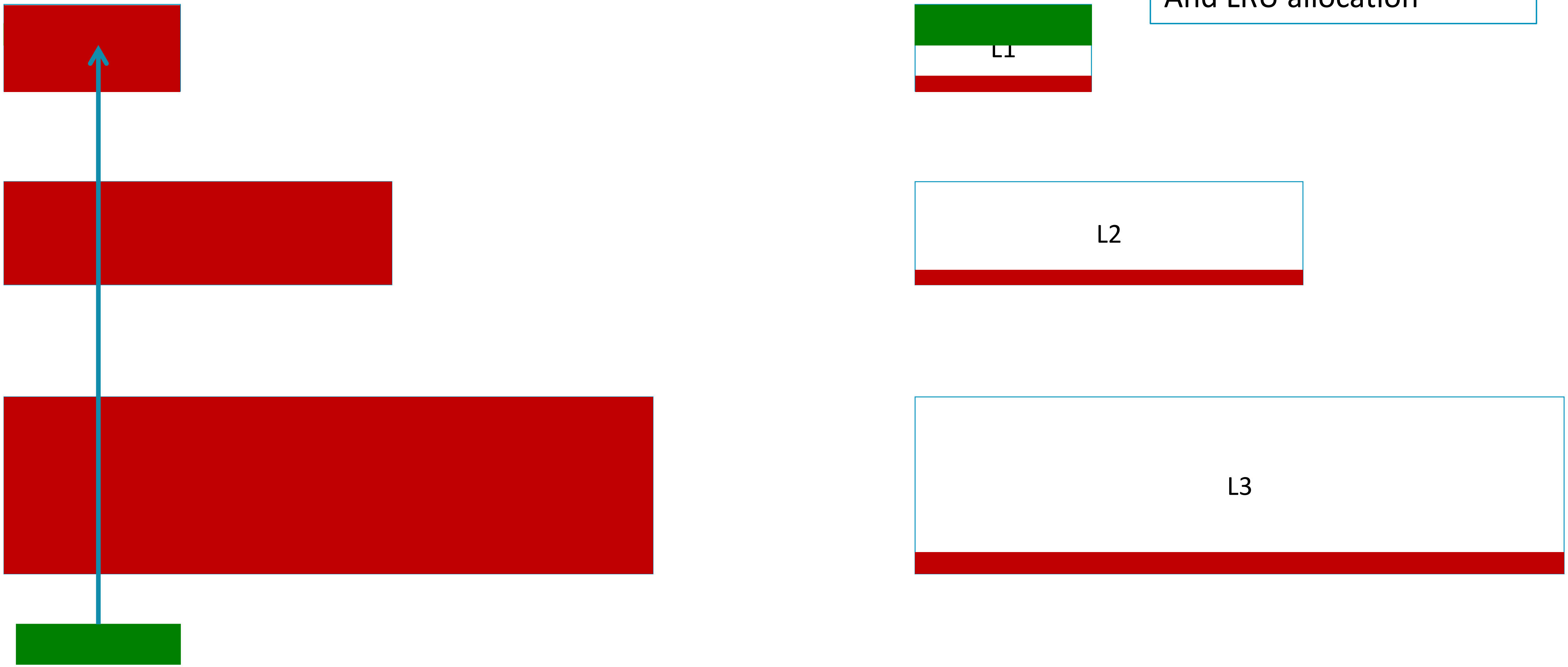


Mixed Temporal and Non-temporal



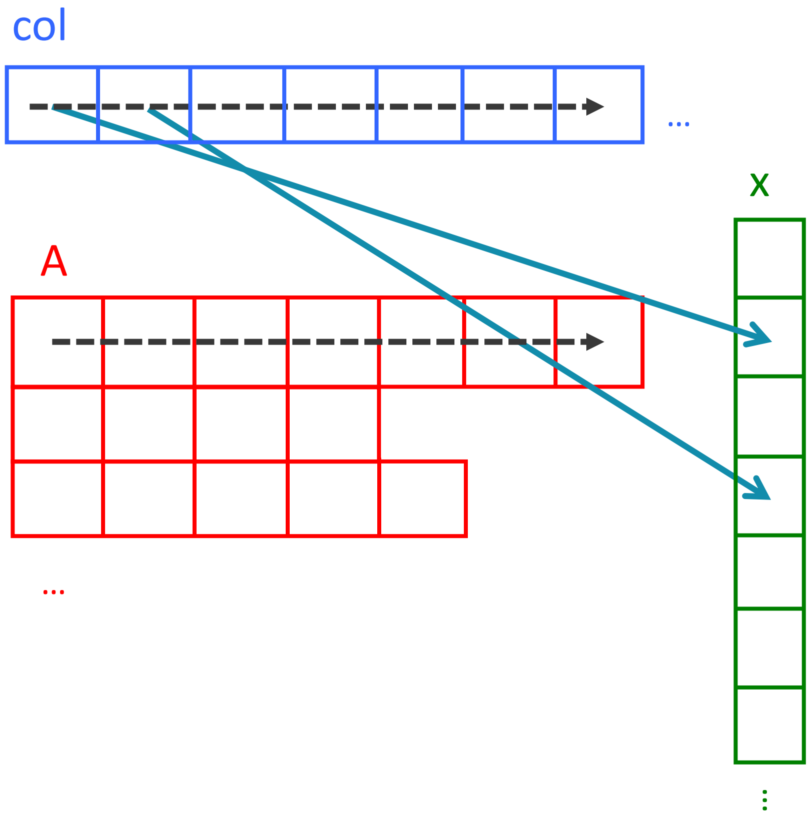
Mixed Temporal and Non-temporal

With non-temporal gather
And LRU allocation



Sparse Matrix Vector

```
for(m=row_start[j]; m<row_start[j+1]; m++)
          y[j] += A[m] * x[col[m]];
```



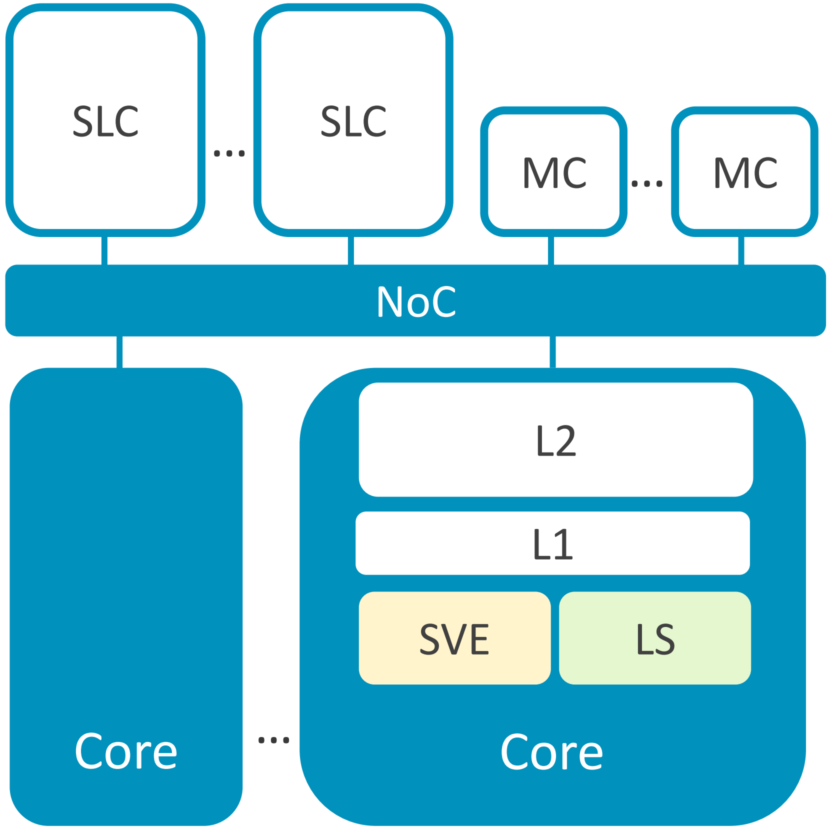
```
ldnt1d    z3.d, p1/z, [x0]
```

```
whilelt p1.d, xzr, x4
ld1sw   z1.d, p1/z, [x2]                     // z1 = &col[]
ld1d    z2.d, p1/z, [x1, z1.d, lsl #3]  // z2 = &x[&col[]]
ld1d    z3.d, p1/z, [x0]                     // z3 = &A[]

fmla    z0.d, p1/m, z2.d, z3.d

add     x2, x2, x7      // add half vector reg length (in bytes)
addvl   x0, x0, 1       // add vector register length (in bytes)
subs    x4, x4, x8      // Remaining length
bgt     .L4

faddv   d0, p0, z0.d    // result for y[j]
```

Vector Architecture Design Trade-offs

arm

Cache Coherent Vector Microarchitecture



Cores with one or more SVE and Load/Store (LS) unit(s)

Private L1 and L2 caches (considered part of the core)

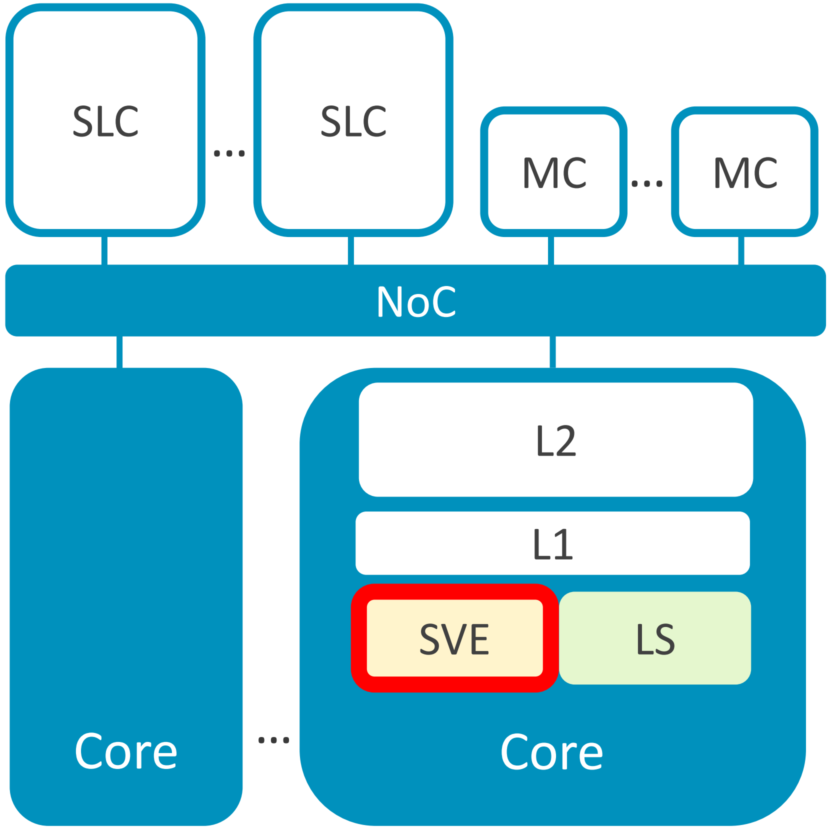
System-level cache (SLC) shared among cores

Memory controllers (MC)

Network on chip (NoC) interconnects cores, SLCs and MCs

Disclaimer: Logical representation, not representative of a physical implementation

SVE Execution Pipeline



Vector length

- Vectorized code will execute less instructions
- Vector register file size

Number of execution units and width

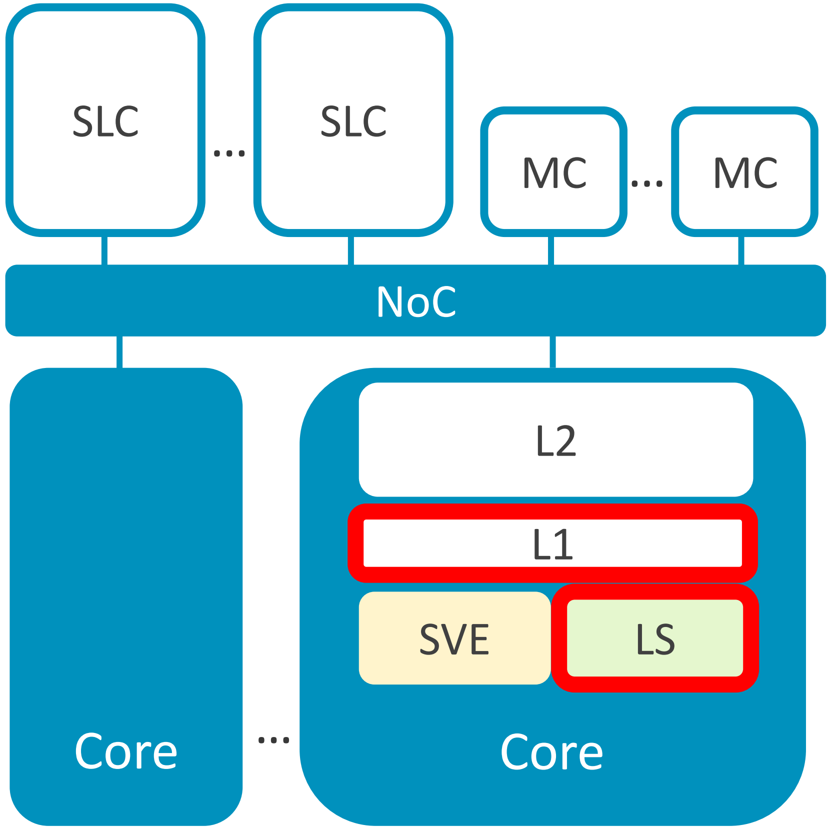
- Determines computation throughput

Vector instruction latencies, cracking, etc…

Example:

- Core implements SVE-256 – registers are 256-bit wide
  - There are two execution units of 256 bits (dual issue)
  - Peak throughput per core is 512b/cycle
- A smaller core could implement SVE-256 but one 128b exec unit
  - Each instruction would use two issue cycles
  - Peak throughput per core would be 128b/cycle

Load-Store Execution Pipeline



Number of Load-Store execution units

L1 maximum access width

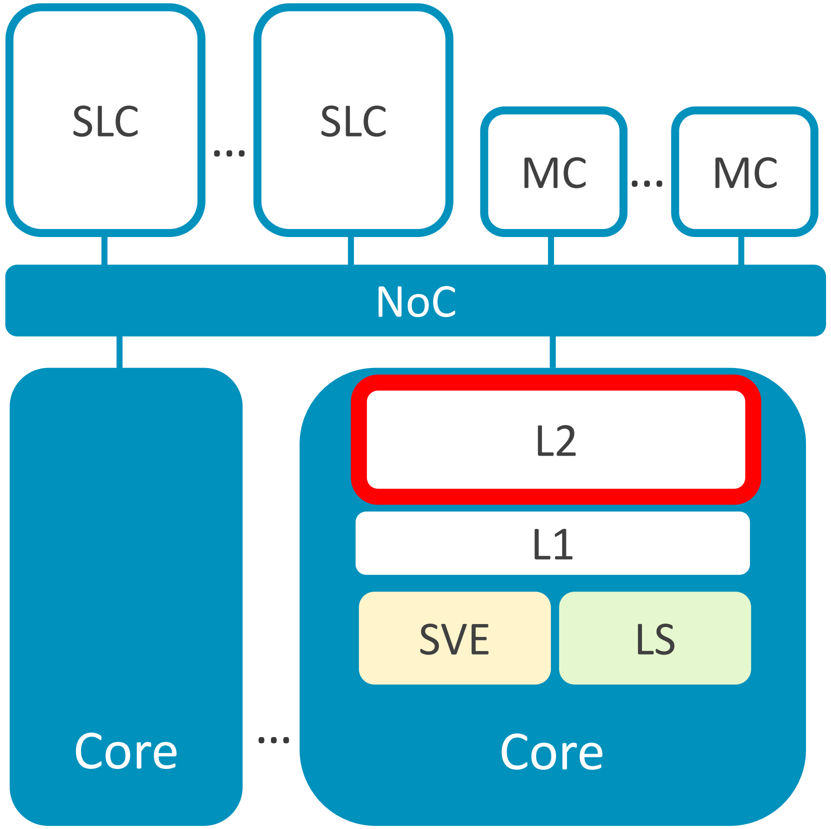
L1 concurrent accesses

- Number of ports
- Number of banks/arrays

L1 cache size

Prefetching aggressiveness

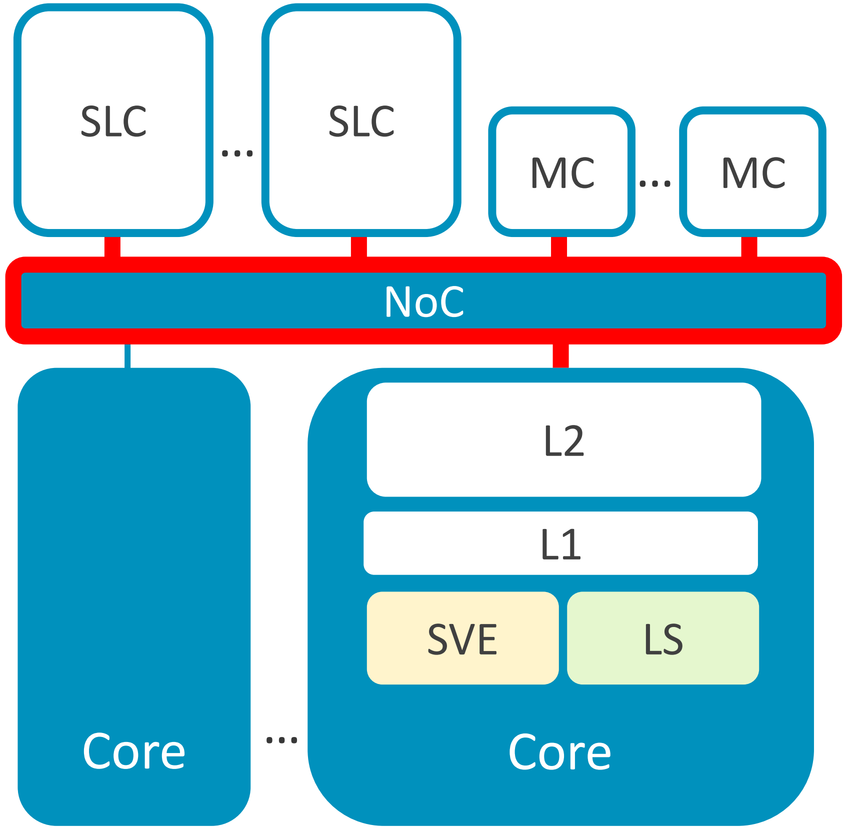
L2 Cache



L2 size to filter NoC accesses

Prefetching aggressiveness

Network on Chip

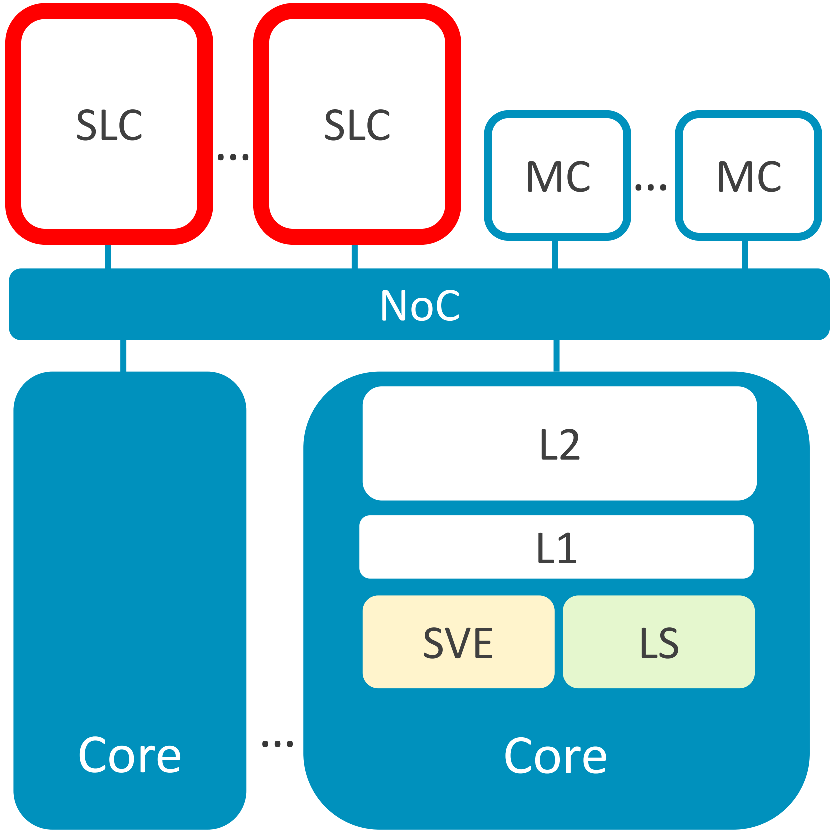


Bandwidth

Connectivity – topology, number of links

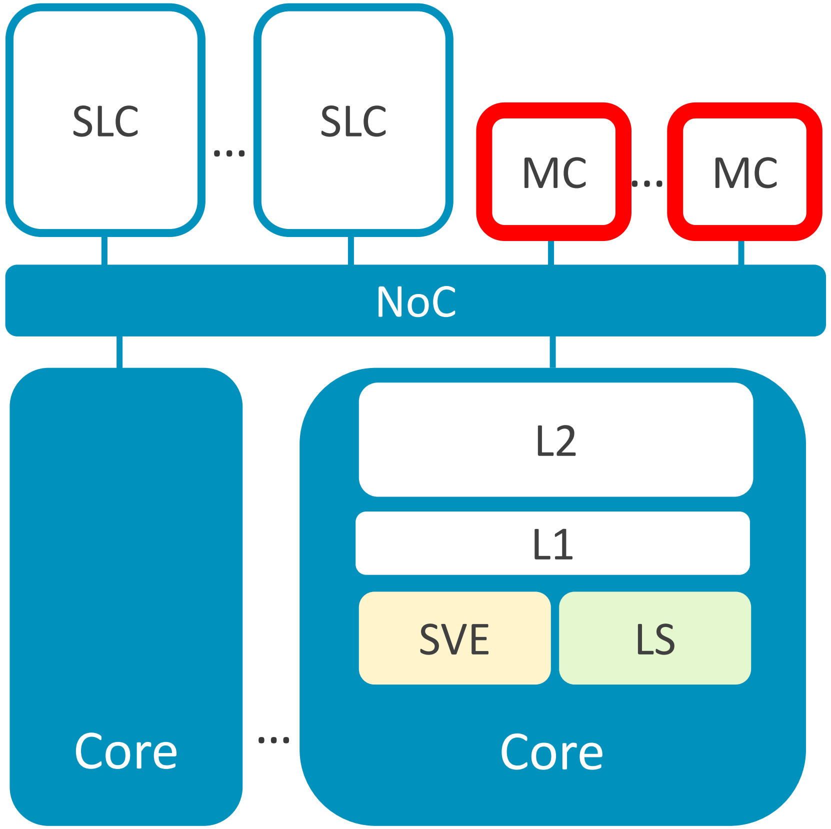
Routing to reduce congestion

System Level Cache



SLC size, prefetching, replacement to filter main memory accesses

Memory

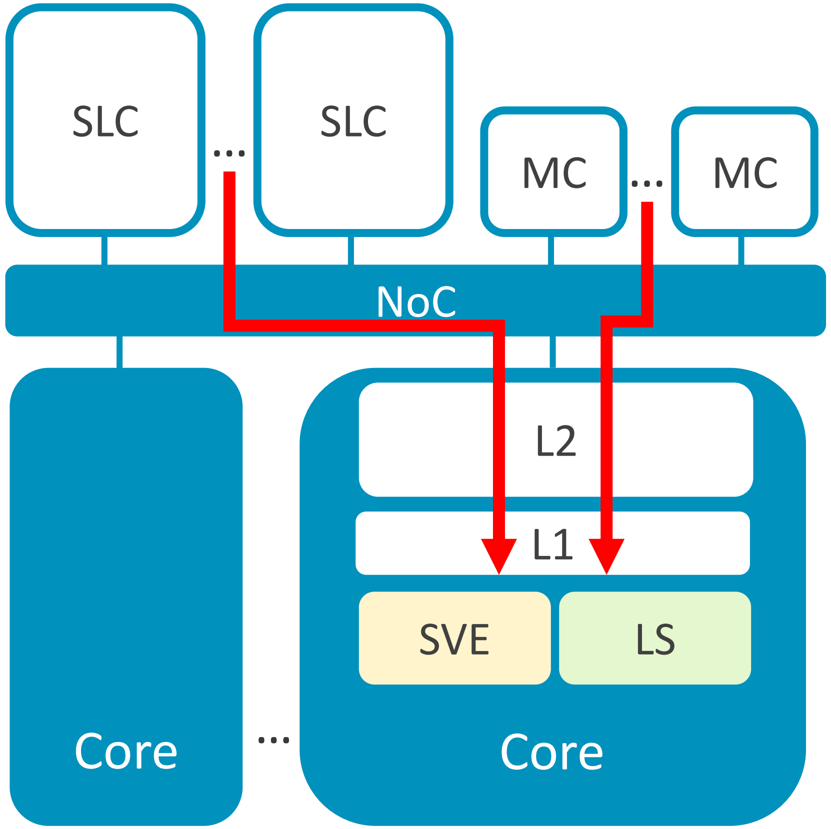


Memory bandwidth

- Channels, banks, width,…

HBM vs DRAM vs NVM…

Basic Concept



← Memory hierarchy and NoC to feed data to SVE units

← SVE unit configuration for target throughput

SVE Programming and Tools

SVE Programming

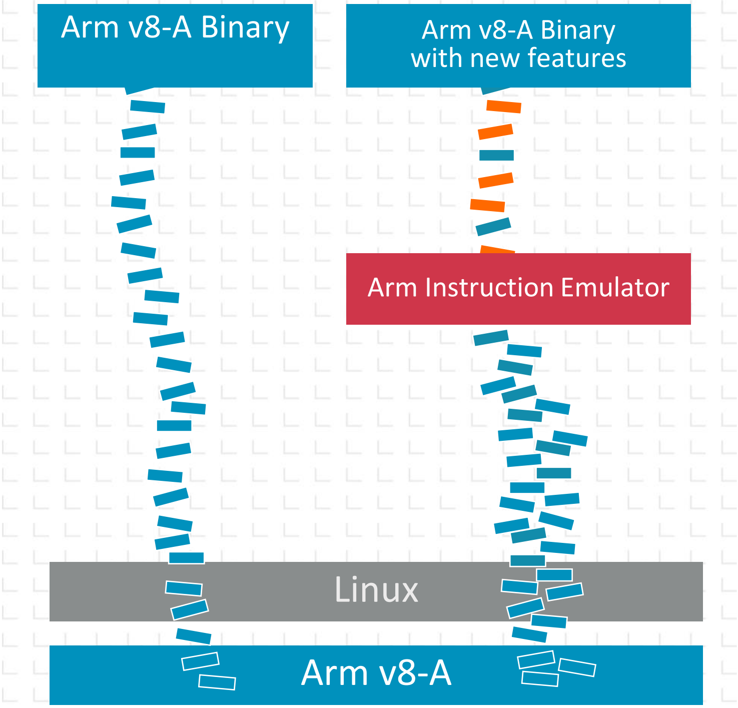
Assembly	Full ISA Specification: The Scalable Vector Extension for Armv8-A Lots of worked examples in A sneak peek into SVE and VLA programming
Intrinsics	Arm C Language Extensions for SVE Arm Scalable Vector Extensions and application to Machine Learning
Compiler	Autovectorization – GCC, Arm Compiler for HPC, Cray, Fujitsu Help the compiler: OpenMP #pragma omp parallel for **simd**

SVE Tools

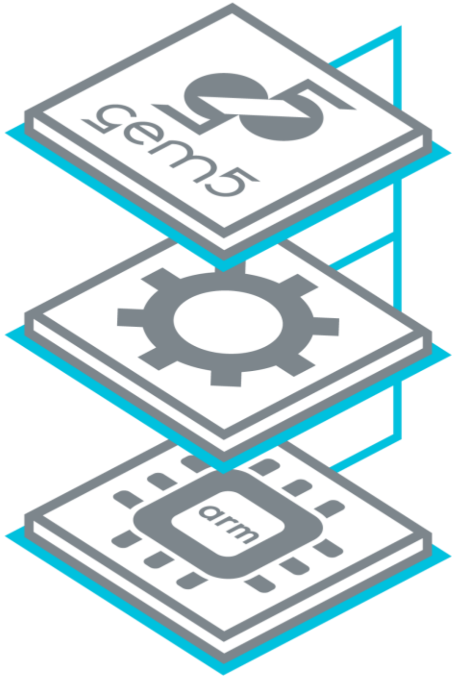
Arm Compiler

arm COMPILER

Arm Instruction Emulator



Research Enablement Kit



arm COMPILER

Commercial C/C++/Fortran compiler with best-in-class performance

Compilers tuned for Scientific Computing and HPC

Latest features and performance optimizations

Commercially supported by Arm

Tuned for Scientific Computing, HPC and Enterprise workloads

- Processor-specific optimizations for various server-class Arm-based platforms
- Optimal shared-memory parallelism using latest Arm-optimized OpenMP runtime

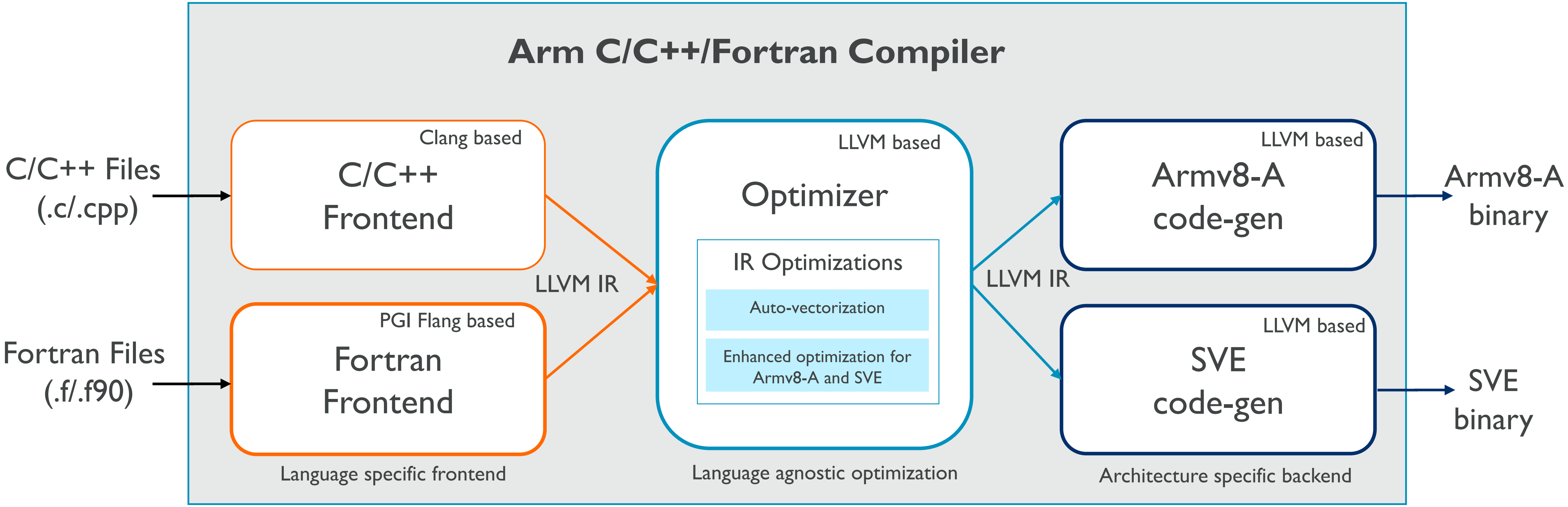
Linux user-space compiler with latest features

- C++ 14 and Fortran 2003 language support with OpenMP 4.5*
- Support for Armv8-A and SVE architecture extension
- Based on LLVM and Flang, leading open-source compiler projects

Commercially supported by Arm

- Available for a wide range of Arm-based platforms running leading Linux distributions – RedHat, SUSE and Ubuntu

Arm Compiler – Building on LLVM, Clang and Flang projects



Arm C/C++/Fortran Compiler

C/C++ Files
(.c/.cpp)

Clang based
C/C++
Frontend

Fortran Files
(.f/.f90)

PGI Flang based
Fortran
Frontend

Language specific frontend

LLVM IR

LLVM based
Optimizer

IR Optimizations

Auto-vectorization

Enhanced optimization for Armv8-A and SVE

Language agnostic optimization

LLVM IR

LLVM based
Armv8-A
code-gen

Armv8-A
binary

LLVM based
SVE
code-gen

SVE
binary

Architecture specific backend

Arm Instruction Emulator

Develop your user-space applications for future hardware today

Develop software for tomorrow’s hardware today

Runs at close to native speed

Commercially Supported by ARM

Start porting and tuning for future architectures early

- Reduce time to market, Save development and debug time with Arm support

Run 64-bit user-space Linux code that uses new hardware features on current Arm hardware

- SVE support available now. Support for 8.x planned.
- Tested with Arm Architecture Verification Suite (AVS)

Near native speed with commercial support

- Emulates only unsupported instructions
- Integrated with other commercial Arm tools including compiler and profiler
- Maintained and supported by Arm for a wide range of Arm-based SoCs

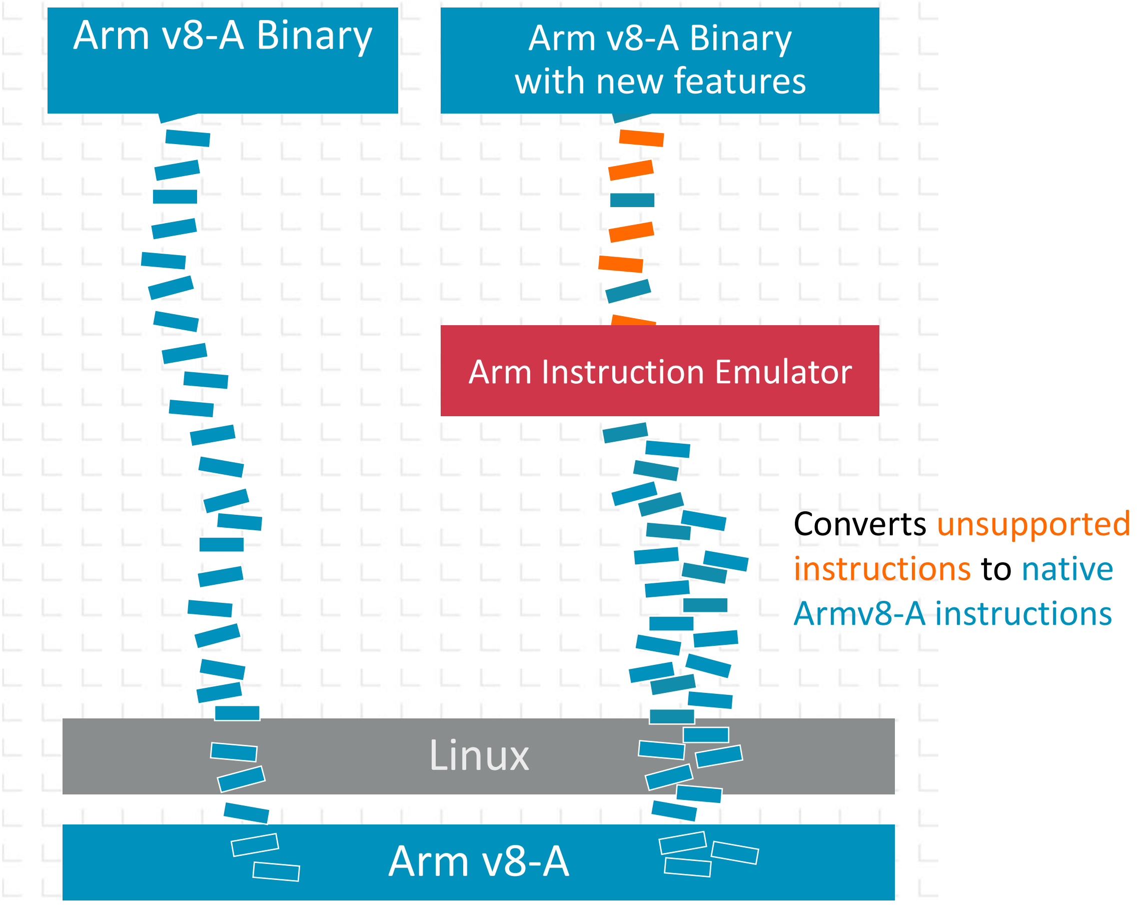
Arm Instruction Emulator

Develop your user-space applications for future hardware today

Run Linux user-space code that uses new hardware features (SVE) on current Arm hardware

Simple “black box” command line tool

```
$ armclang hello.c --march=armv8+sve
$ ./a.out
Illegal instruction
$ armie –msve-vector-bits=256 ./a.out
Hello
```



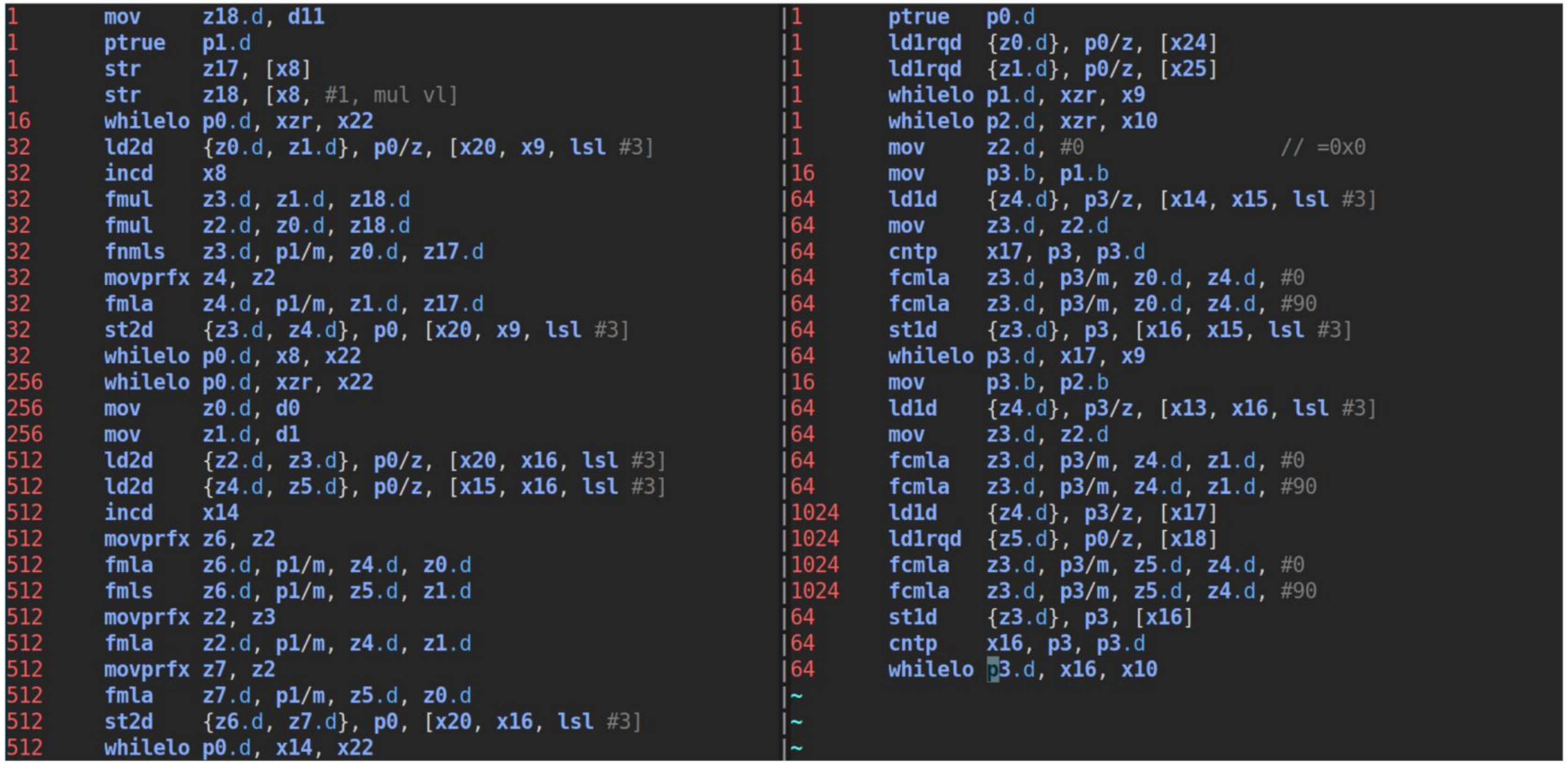
Converts unsupported instructions to native Armv8-A instructions

DynamoRIO Instruction Count

- Counts AArch64 Instructions and specifically SVE instructions
- SVE instructions are recorded in binary.sve.instrs
- decode utility can be used to make the traces human-readable

```
armie -c inscount -e -msve-vector-bits=256 ./sve_binary
cat sve_binary.sve.instrs \
| awk '{printf("%s %d\n",$2,$3)}'\
| while read x c; do decode $x $c; done
```

Instruction Count - ZGEMM example



```
1        mov     z18.d, d11
1        ptrue   p1.d
1        str     z17, [x8]
1        str     z18, [x8, #1, mul vl]
16       whilelo p0.d, xzr, x22
32       ld2d    {z0.d, z1.d}, p0/z, [x20, x9, lsl #3]
32       incd    x8
32       fmul    z3.d, z1.d, z18.d
32       fmul    z2.d, z0.d, z18.d
32       fnmls   z3.d, p1/m, z0.d, z17.d
32       movprfx z4, z2
32       fmla    z4.d, p1/m, z1.d, z17.d
32       st2d    {z3.d, z4.d}, p0, [x20, x9, lsl #3]
32       whilelo p0.d, x8, x22
256      whilelo p0.d, xzr, x22
256      mov     z0.d, d0
256      mov     z1.d, d1
512      ld2d    {z2.d, z3.d}, p0/z, [x20, x16, lsl #3]
512      ld2d    {z4.d, z5.d}, p0/z, [x15, x16, lsl #3]
512      incd    x14
512      movprfx z6, z2
512      fmla    z6.d, p1/m, z4.d, z0.d
512      fmls    z6.d, p1/m, z5.d, z1.d
512      movprfx z2, z3
512      fmla    z2.d, p1/m, z4.d, z1.d
512      movprfx z7, z2
512      fmla    z7.d, p1/m, z5.d, z0.d
512      st2d    {z6.d, z7.d}, p0, [x20, x16, lsl #3]
512      whilelo p0.d, x14, x22
```

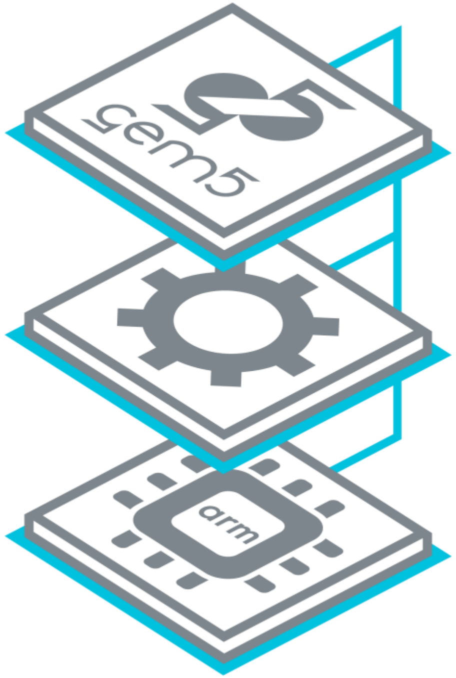
```
|1       ptrue   p0.d
|1       ld1rqd  {z0.d}, p0/z, [x24]
|1       ld1rqd  {z1.d}, p0/z, [x25]
|1       whilelo p1.d, xzr, x9
|1       whilelo p2.d, xzr, x10
|1       mov     z2.d, #0                 // =0x0
|16      mov     p3.b, p1.b
|64      ld1d    {z4.d}, p3/z, [x14, x15, lsl #3]
|64      mov     z3.d, z2.d
|64      cntp    x17, p3, p3.d
|64      fcmla   z3.d, p3/m, z0.d, z4.d, #0
|64      fcmla   z3.d, p3/m, z0.d, z4.d, #90
|64      st1d    {z3.d}, p3, [x16, x15, lsl #3]
|64      whilelo p3.d, x17, x9
|16      mov     p3.b, p2.b
|64      ld1d    {z4.d}, p3/z, [x13, x16, lsl #3]
|64      mov     z3.d, z2.d
|64      fcmla   z3.d, p3/m, z4.d, z1.d, #0
|64      fcmla   z3.d, p3/m, z4.d, z1.d, #90
|1024    ld1d    {z4.d}, p3/z, [x17]
|1024    ld1rqd  {z5.d}, p0/z, [x18]
|1024    fcmla   z3.d, p3/m, z5.d, z4.d, #0
|1024    fcmla   z3.d, p3/m, z5.d, z4.d, #90
|64      st1d    {z3.d}, p3, [x16]
|64      cntp    x16, p3, p3.d
|64      whilelo p3.d, x16, x10
|~
|~
|~
```

Left: reference FORTRAN implementation, right: custom kernel.

Arm Research Enablement Kit – System Modeling Using gem5

https://developer.arm.com/research/research-enablement/system-modeling

- HPI: First Armv8-A based CPU timing model released by Arm
- Documentation about the HPI core model (based on MinorCPU)
- Documentation about running benchmarks (PARSEC)
- Useful scripts (clone.sh, read_results.sh)
  - Using the current mainline gem5 source code
  - SVE patches will be upstreamed after completing beta testing



More on SVE

http://developer.arm.com/hpc

- Full SVE specification: Arm Architecture Reference Manual Supplement, SVE for ARMv8-A
- Intrinsics: Arm C Language Extensions for SVE and
- Lots of worked examples in A sneak peek into SVE and VLA programming
- Optimized machine learning in Arm SVE and application to Machine Learning

Future SVE

Contact me for opportunities

arm.com/careers

Internships 2019 (check September-October)

- Sophia Antipolis, France
- Cambridge, UK

Full-time positions

arm Research Summit

17-19 September 2018
Robinson College, Cambridge, UK

- Architecture & Memory
- Biotechnology
- Flexible Electronics
- High Performance Computing
- Internet of Things
- Large Scale Systems
- Machine Learning
- Security & Servers

With more still to be announced!

arm.com/summit

arm Research

Thank You!

Danke!

Merci!

谢谢!

ありがとう!

Gracias!

Kiitos!

arm Research