arm Research

Vector Processing with Arm SVE

Alex Rico

Staff Research Engineer

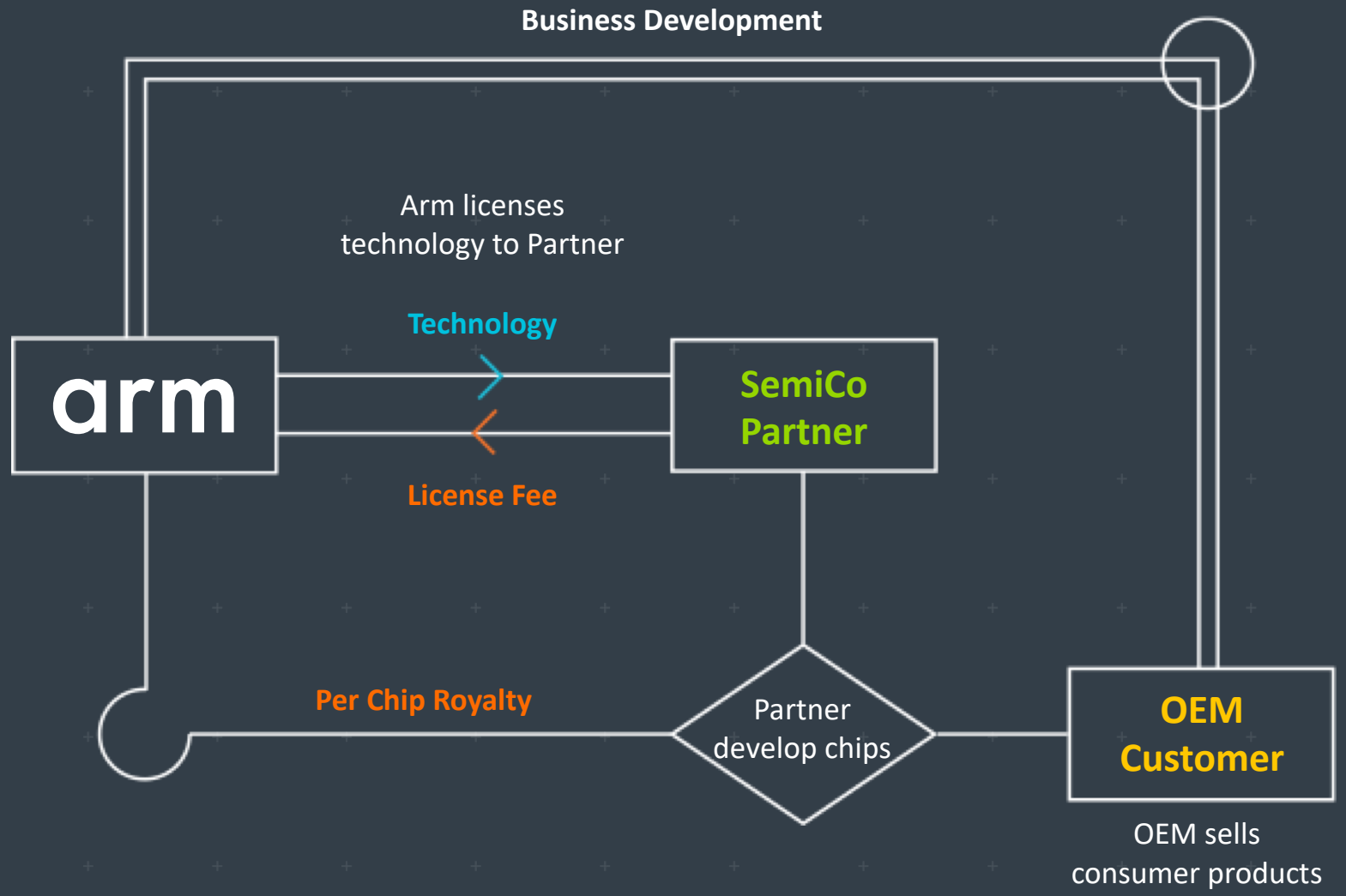
System on Chip Design Seminar

October 5th, 2018

A continuous partnership model

Arm develops technology that is licensed to semiconductor companies.

Arm receives an upfront license fee and a royalty on every chip that contains its technology.



The architects of global possibilities

The global leader in the development of licensable technology

- R&D outsourcing for semiconductor companies

Focused on freedom and flexibility to innovate

- Technology reused across multiple applications

With a partnership based culture & business model

- Licensees take advantage of learnings from a uniquely collaborative ecosystem

>1,400

licenses, growing by >100 every year

17.7bn

Arm-based chips shipped in FY2016

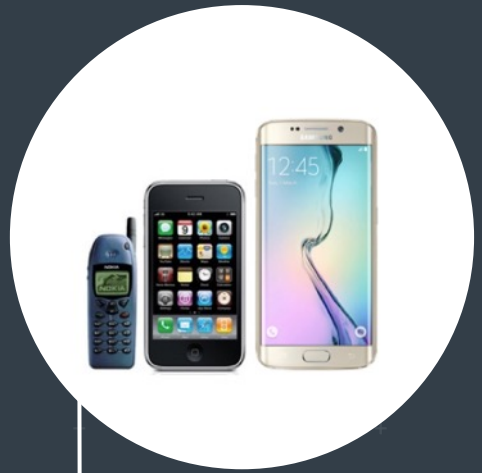
>460 licensees

Industry leaders and high-growth start-ups; chip companies and OEMs

From inception to now

1990

Joint venture between Acorn Computers and Apple.



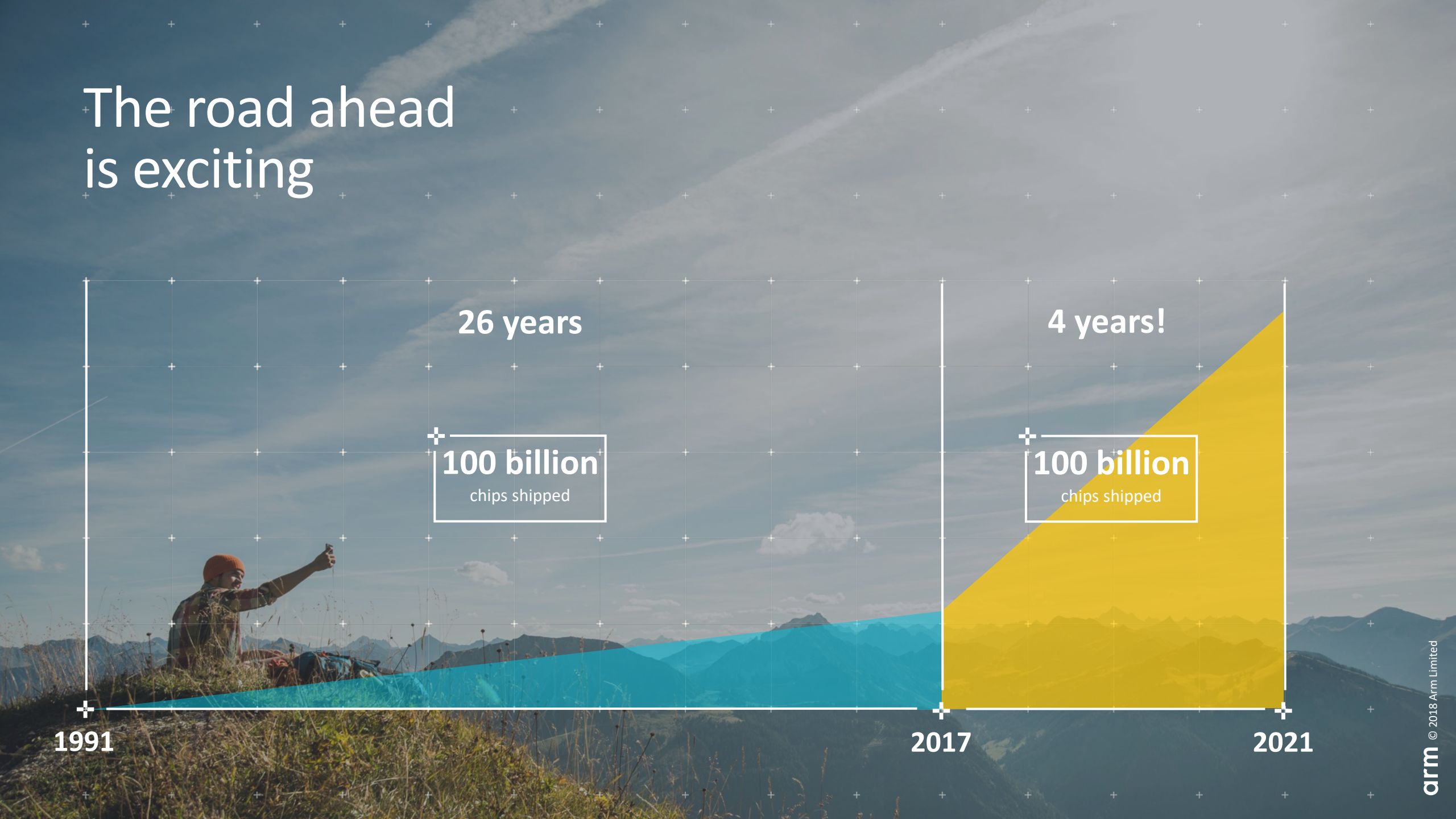
Designed into first mobile phones and then smartphones.

1993 onwards

Today

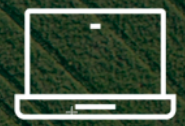
Now all electronic devices can use intelligent Arm technology.

The road ahead is exciting



Distributing intelligence from edge to cloud

On-device learning for enhanced user privacy

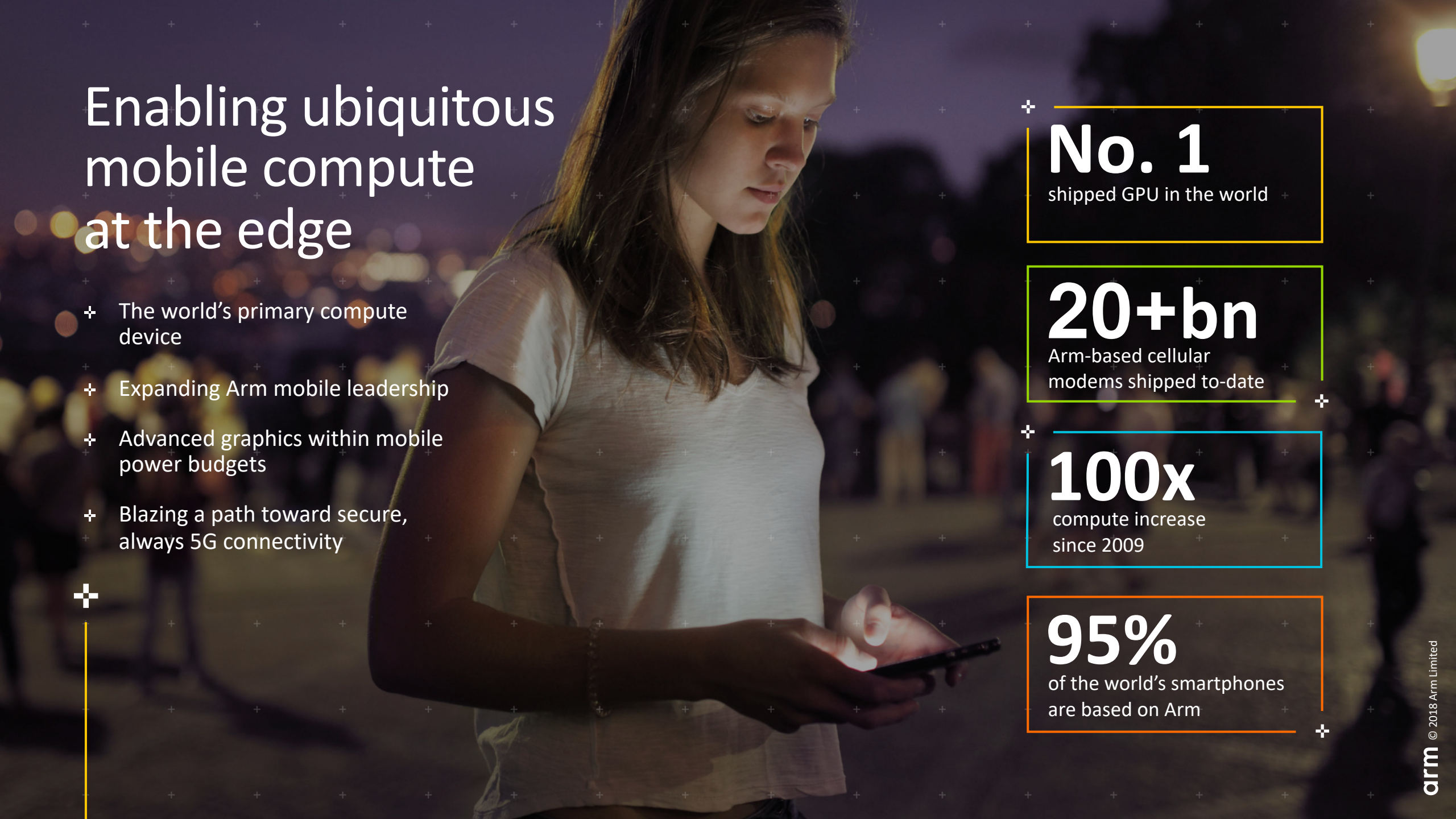


Compute performance to deliver a hi-fidelity world

Real-time inference for autonomous systems

Security and privacy for your data

4k, HDR and 5G for more human-like interfaces



Enabling ubiquitous mobile compute at the edge

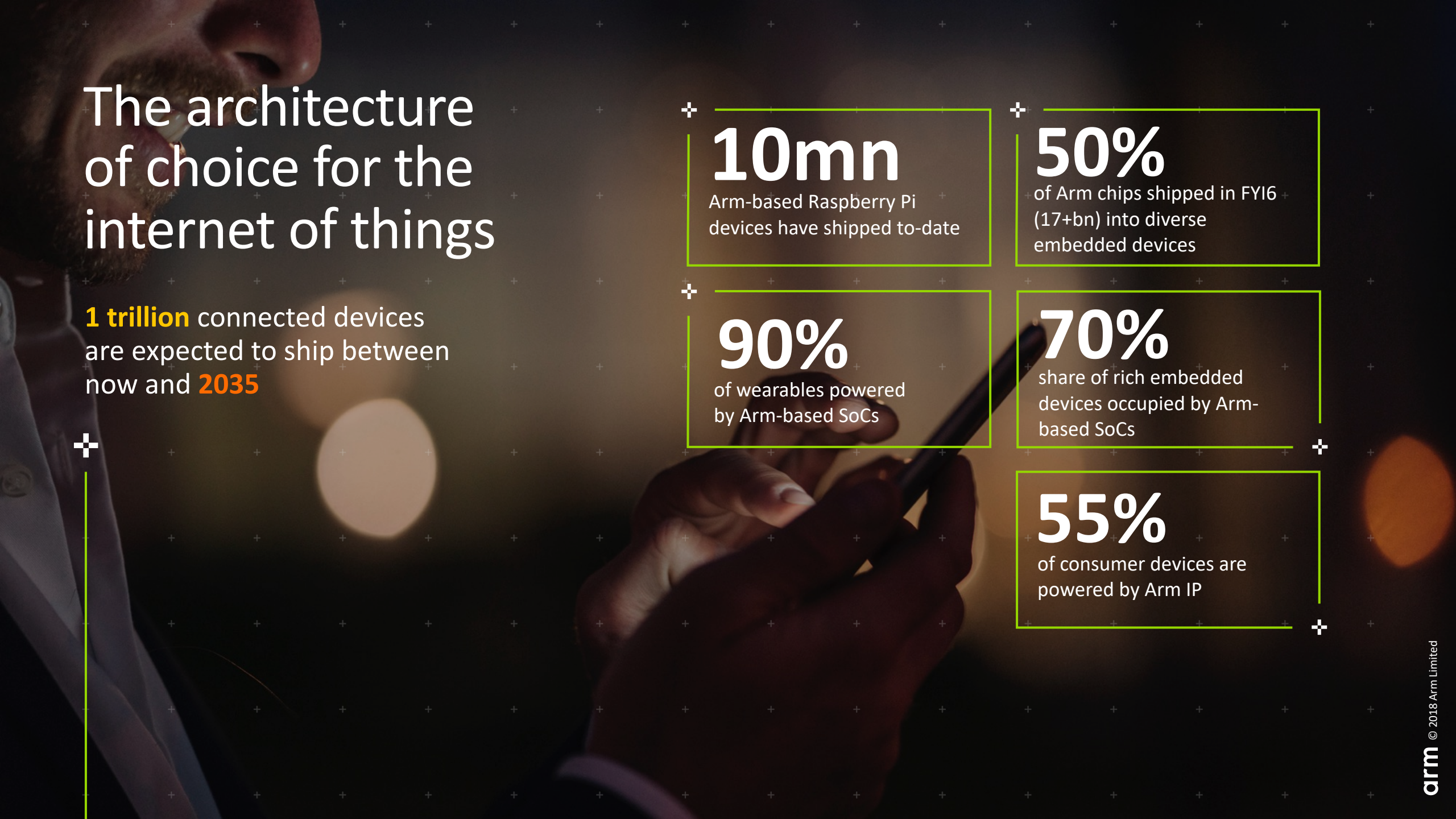
- The world's primary compute device
- Expanding Arm mobile leadership
- Advanced graphics within mobile power budgets
- Blazing a path toward secure, always 5G connectivity

No. 1
shipped GPU in the world

20+bn
Arm-based cellular modems shipped to-date

100x
compute increase since 2009

95%
of the world's smartphones are based on Arm



The architecture of choice for the internet of things

1 trillion connected devices are expected to ship between now and **2035**

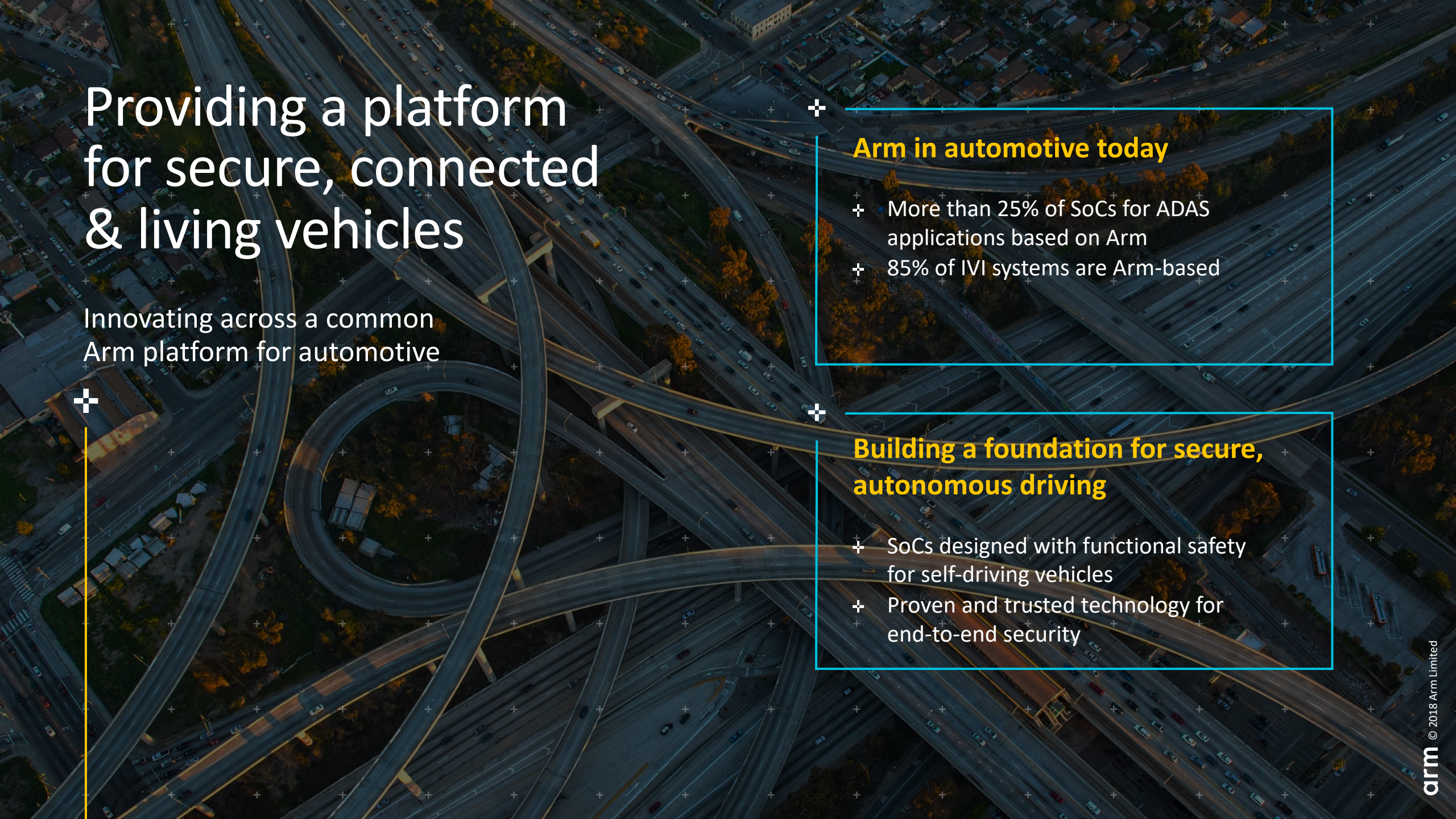
10mn
Arm-based Raspberry Pi devices have shipped to-date

50%
of Arm chips shipped in FYI6 (17+bn) into diverse embedded devices

90%
of wearables powered by Arm-based SoCs

70%
share of rich embedded devices occupied by Arm-based SoCs

55%
of consumer devices are powered by Arm IP



Providing a platform for secure, connected & living vehicles

Innovating across a common Arm platform for automotive

Arm in automotive today

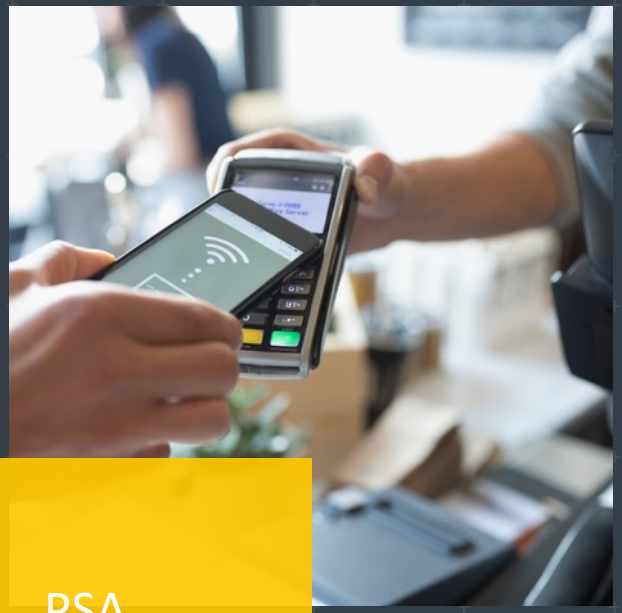
- More than 25% of SoCs for ADAS applications based on Arm
- 85% of IVI systems are Arm-based

Building a foundation for secure, autonomous driving

- SoCs designed with functional safety for self-driving vehicles
- Proven and trusted technology for end-to-end security

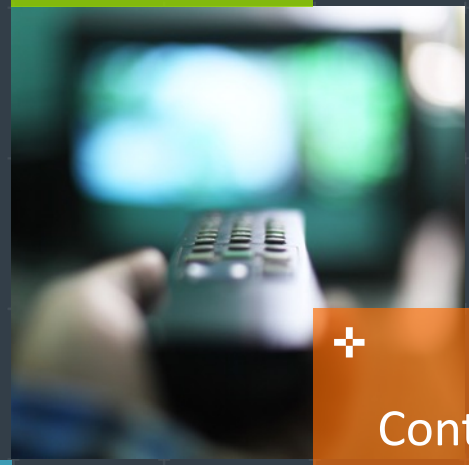
Protecting billions of devices today

arm Trust Zone
Security System

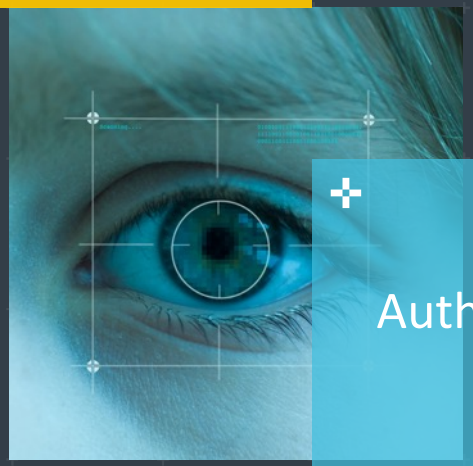


Enterprise Security

PSA

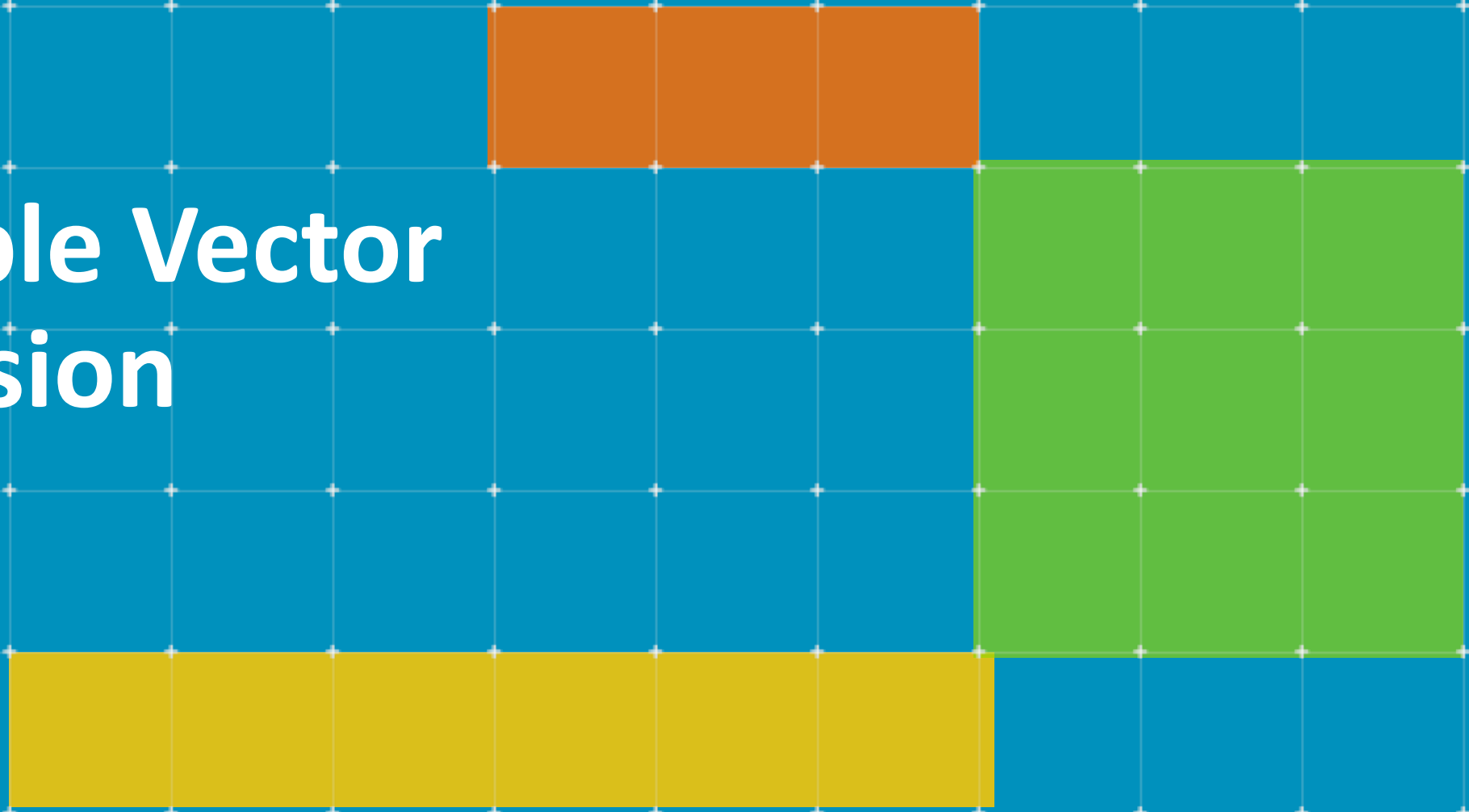


Content Protection



Authentication

Scalable Vector Extension



arm Research

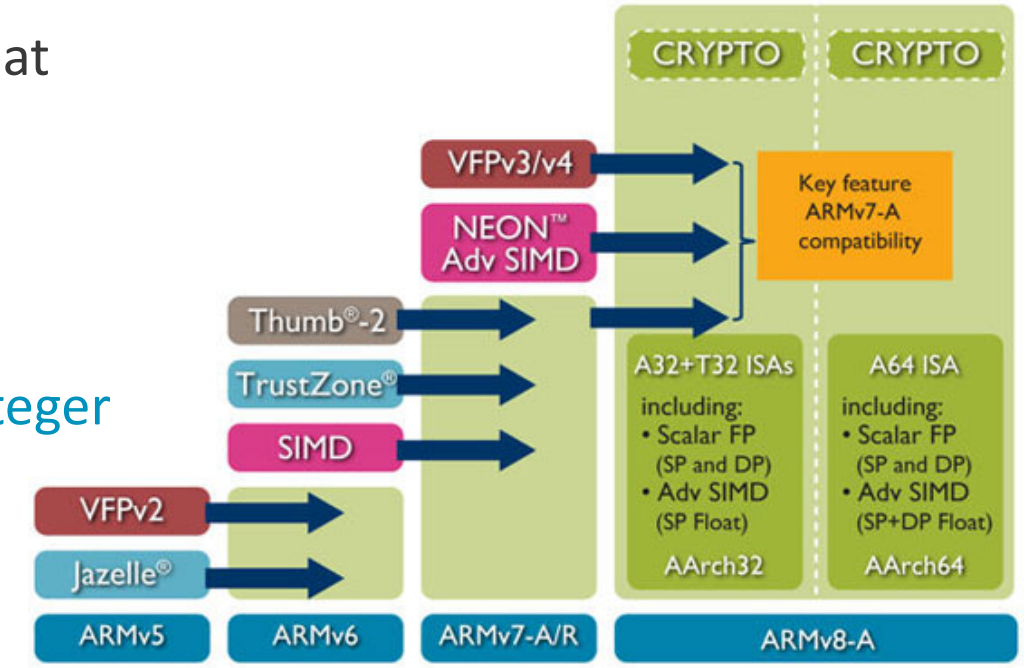
Arm Architecture

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



arm Research

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

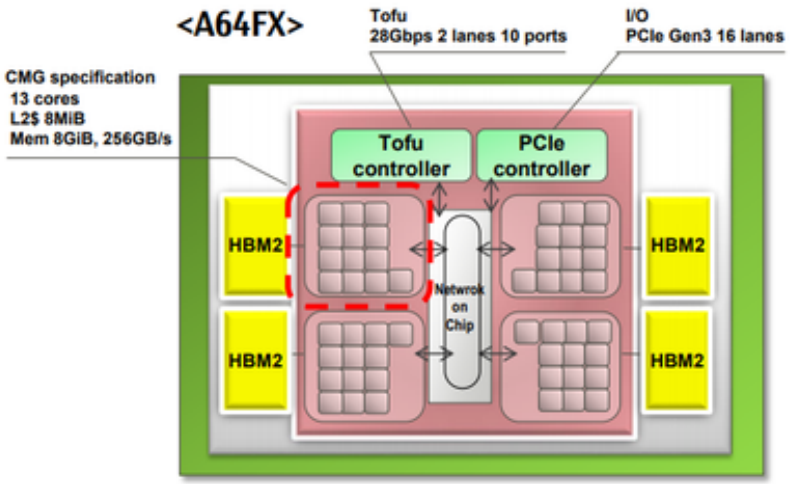
Will enable advanced vectorizing compilers to extract more fine-grain parallelism from existing code and so reduce software deployment effort

Post-K Supercomputer goes Arm with SVE

A64FX Chip Overview

FUJITSU

- **Architecture Features**
  - Armv8.2-A (AArch64 only)
  - SVE 512-bit wide SIMD
  - 48 computing cores + 4 assistant cores*
    *All the cores are identical
  - HBM2 32GiB
  - Tofu 6D Mesh/Torus
    28Gbps x 2 lanes x 10 ports
  - PCIe Gen3 16 lanes
- **7nm FinFET**
  - 8,786M transistors
  - 594 package signal pins
- **Peak Performance (Efficiency)**
  - >2.7TFLOPS (>90%@DGEMM)
  - Memory B/W 1024GB/s (>80%@Stream Triad)



	A64FX (Post-K)	SPARC64 XIfx (PRIMEHPC FX100)
ISA (Base)	Armv8.2-A	SPARC-V9
ISA (Extension)	SVE	HPC-ACE2
Process Node	7nm	20nm
Peak Performance	>2.7TFLOPS	1.1TFLOPS
SIMD	512-bit	256-bit
# of Cores	48+4	32+2
Memory	HBM2	HMC
Memory Peak B/W	1024GB/s	240GB/s x2 (in/out)

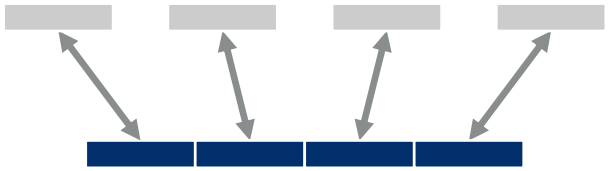
6

slides from Fujitsu

arm Research

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

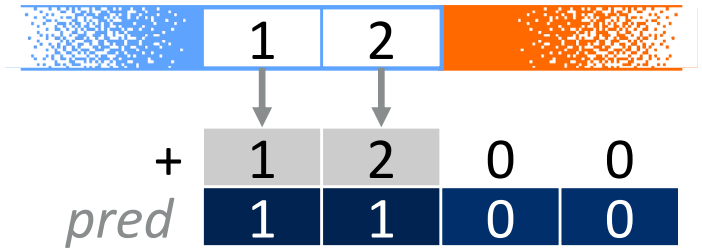
Predicate-driven loop control and management

Eliminate scalar loop heads and tails by processing partial vectors.

arm Research

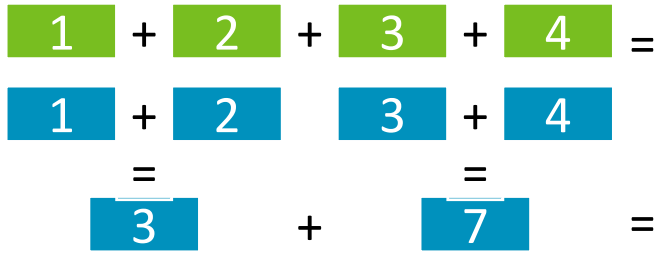
Introducing the Scalable Vector Extension (SVE)

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

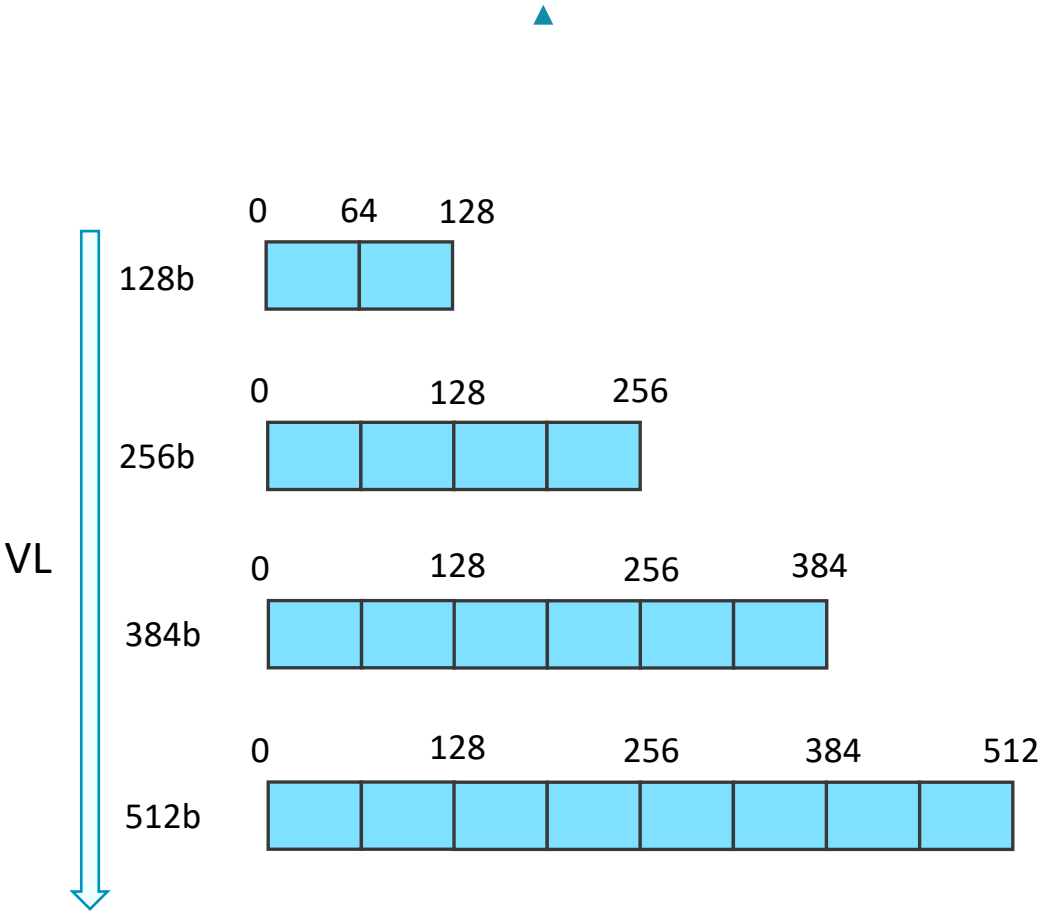
In-order and tree-based reductions trade-off performance and repeatability.

arm Research

What's the vector length?

There is **no** preferred vector length

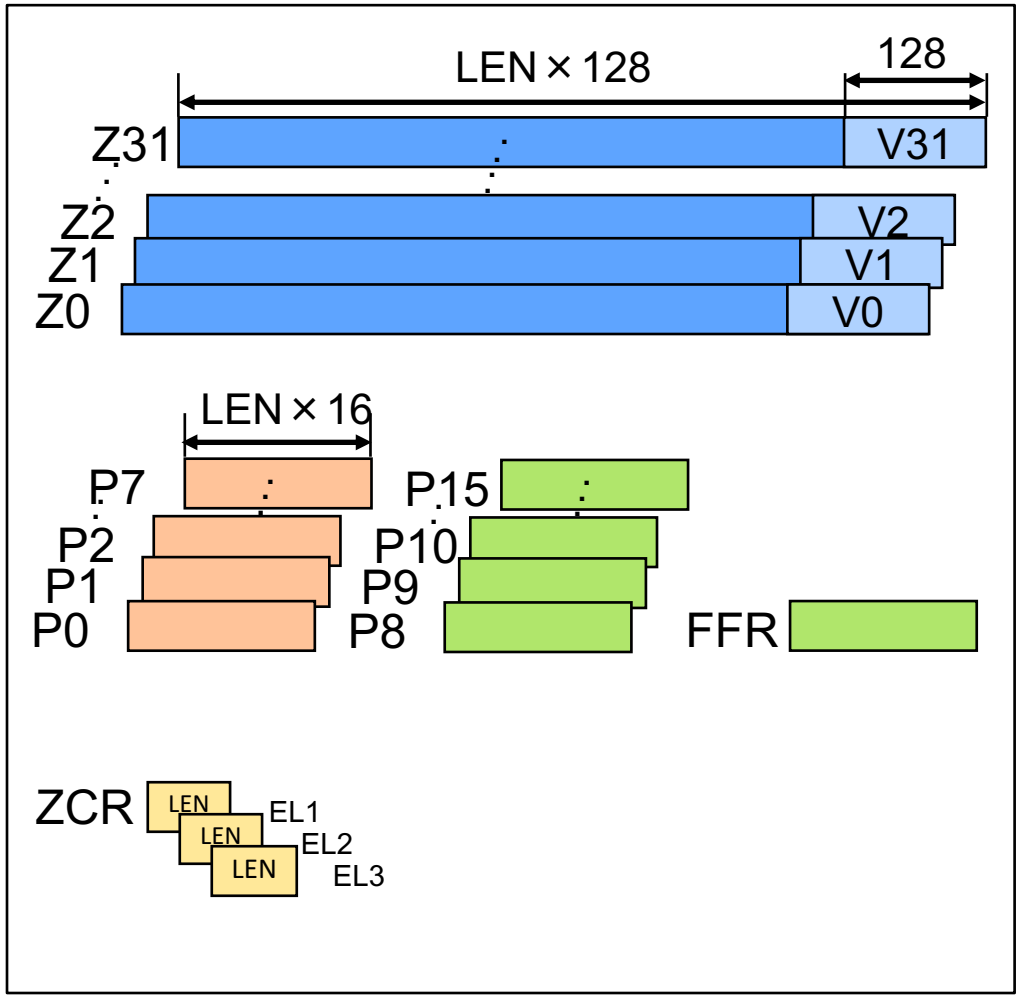
- Vector Length (VL) is the CPU implementor's choice, from 128 to 2048 bits, in increments of 128
- Adopting a Vector Length Agnostic (VLA) code generation style makes code portable across all possible vector lengths
- VLA is made possible by the per-lane predication, predicate-driven loop control, vector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



arm Research

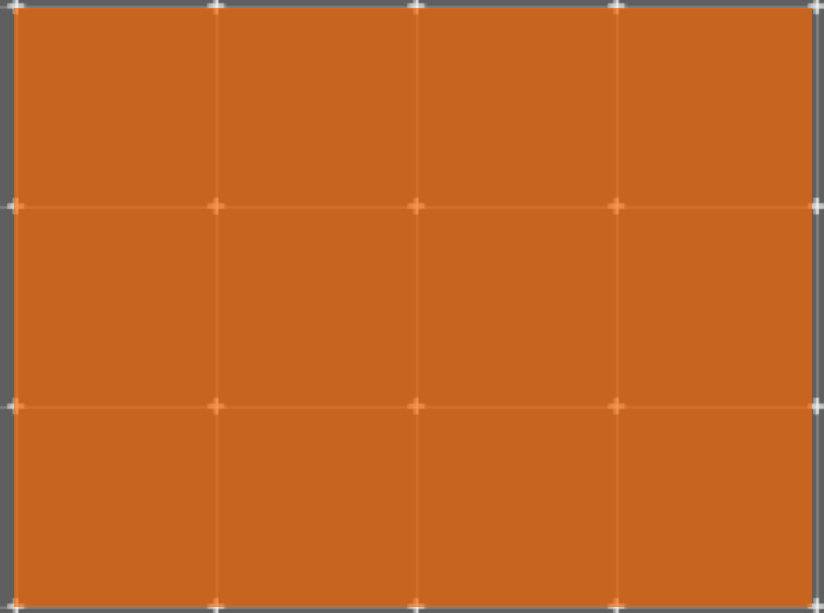
SVE – architectural state

- 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



arm Research

SVE Visual Examples



arm

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)

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 (scalar)

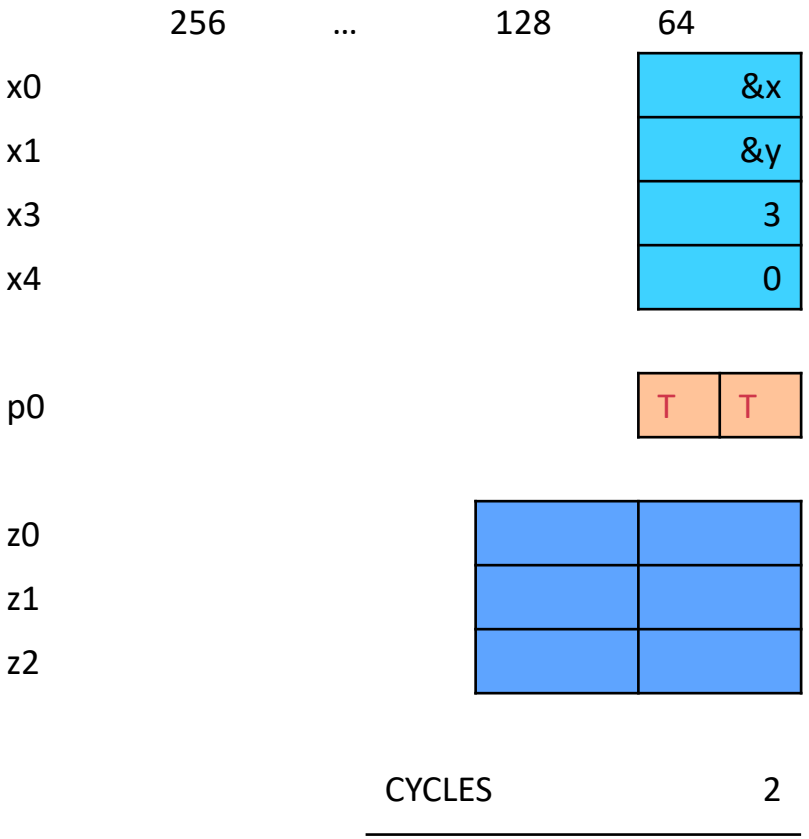
```
daxpy_:
        ldrsw           x3, [x3]
        mov             x4, #0
        ldr             d0, [x2]
        b               .latch
.loop:
        ldr             d1, [x0, x4, lsl #3]
        ldr             d2, [x1, x4, lsl #3]
        fmadd           d2, d1, d0, d2
        str             d2,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


arm Research

daxpy (SVE – 128b)

```
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
```

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

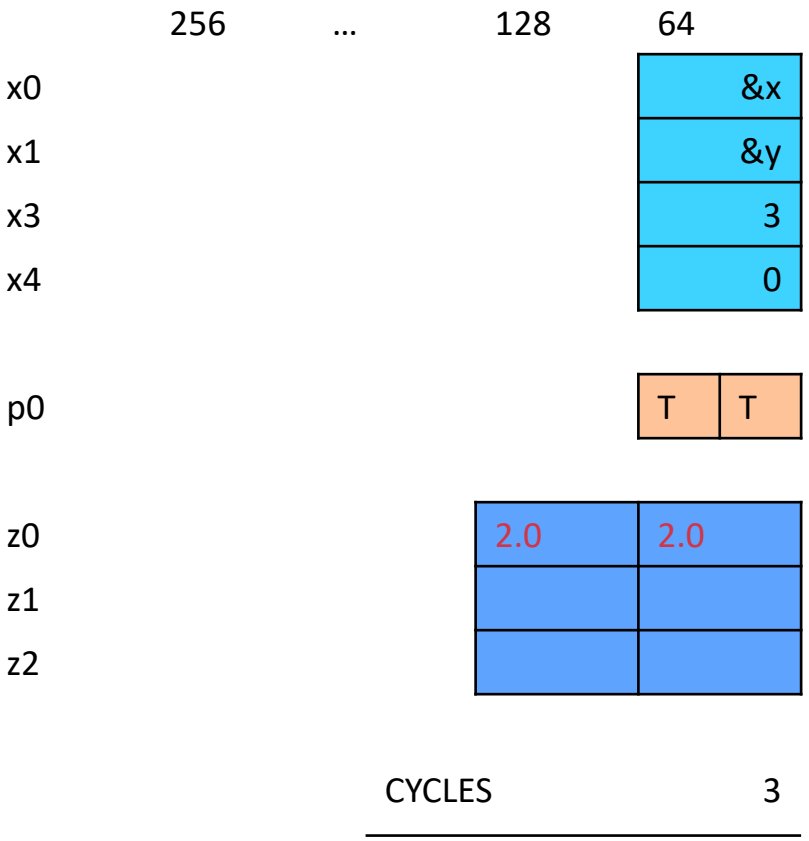


arm Research

daxpy (SVE – 128b)

```
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
```

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

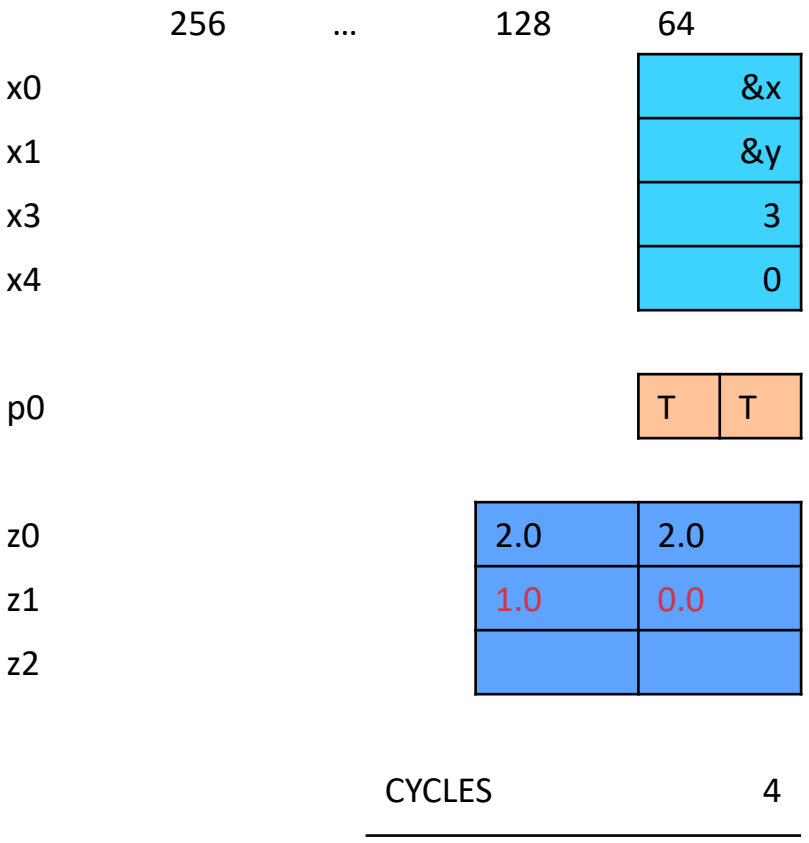


arm Research

daxpy (SVE – 128b)

```
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
```

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



arm Research

daxpy (SVE – 128b)

```
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
```

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

	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)

```
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
```

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

	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

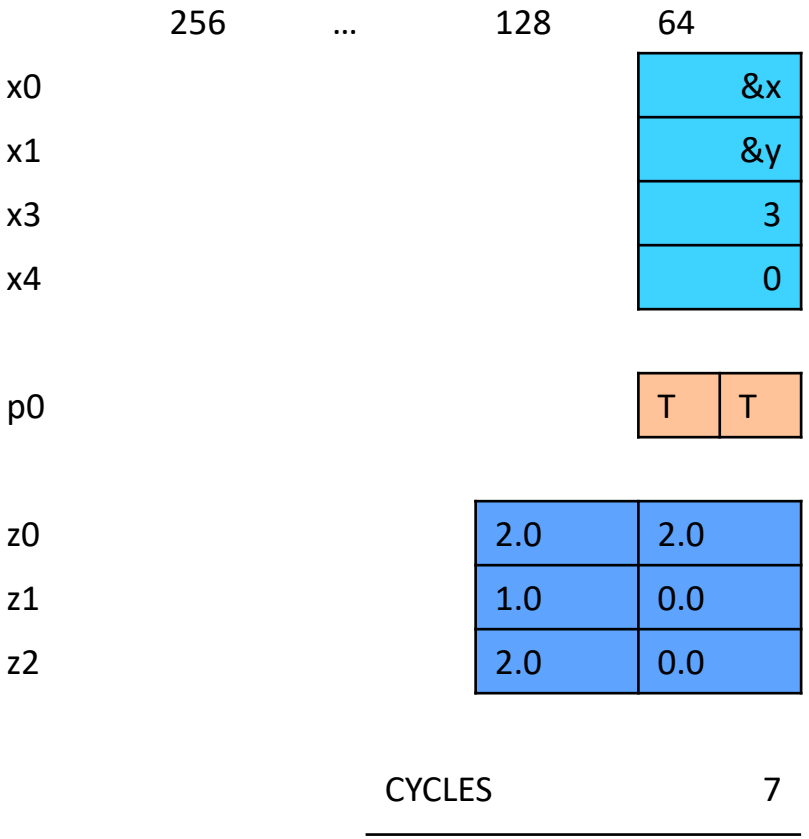
CYCLES 6

arm Research

daxpy (SVE – 128b)

```
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
```

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



arm Research

daxpy (SVE – 128b)

```
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
```

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

	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

arm Research

daxpy (SVE – 128b)

```
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
```

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

	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

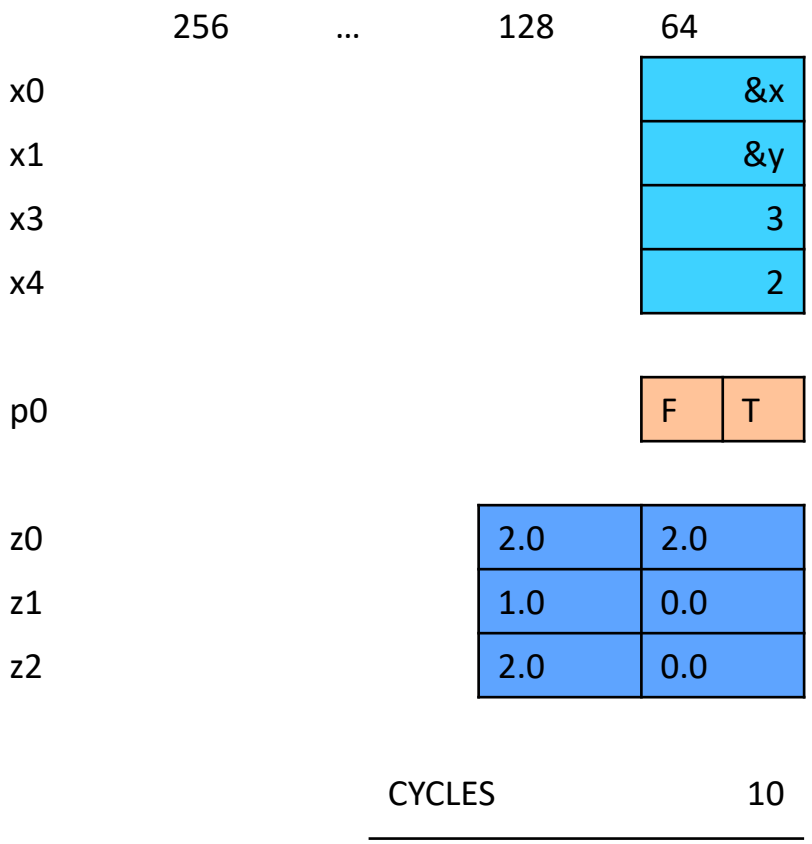
CYCLES 9

arm Research

daxpy (SVE – 128b)

```
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
```

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

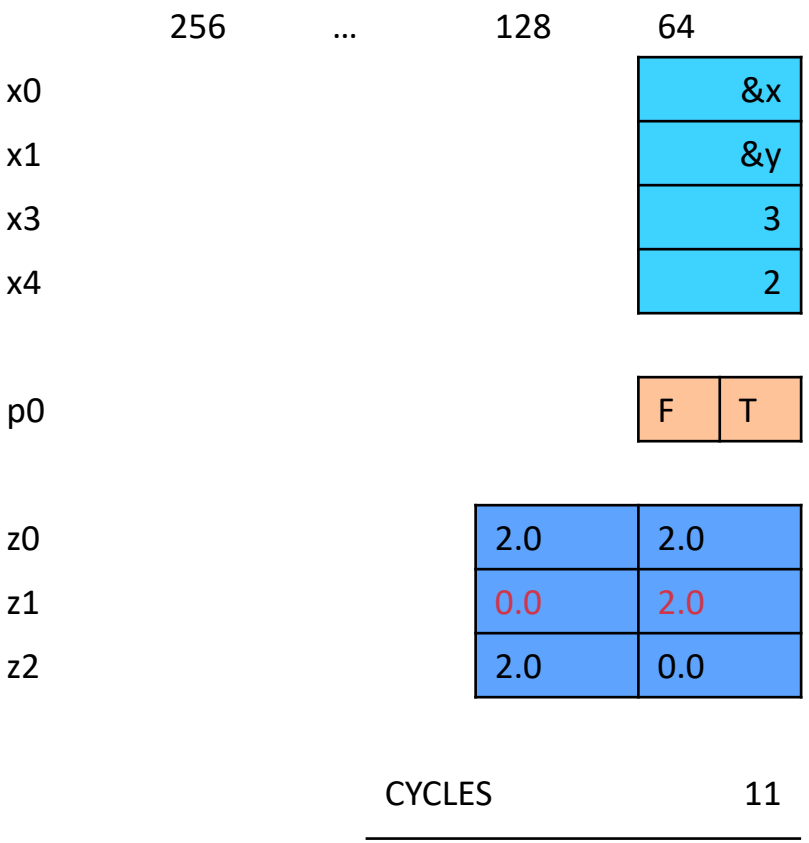


arm Research

daxpy (SVE – 128b)

```
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
```

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



arm Research

daxpy (SVE – 128b)

```
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
```

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

	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	0.0

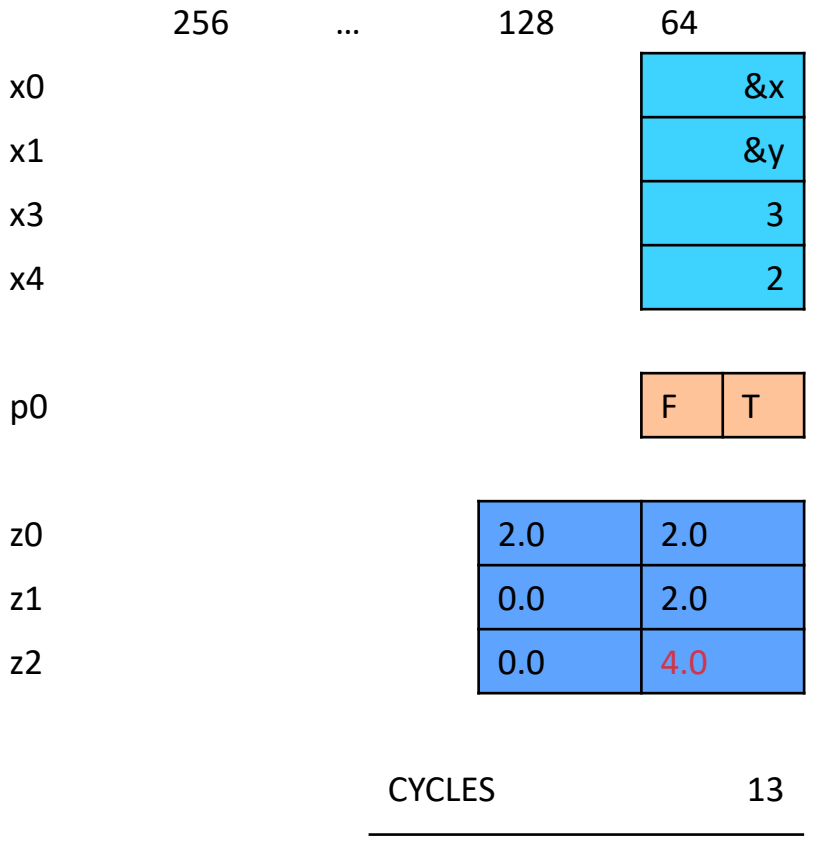
CYCLES 12

arm Research

daxpy (SVE – 128b)

```
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
```

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



CYCLES	13

arm Research

daxpy (SVE – 128b)

```
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
```

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

	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

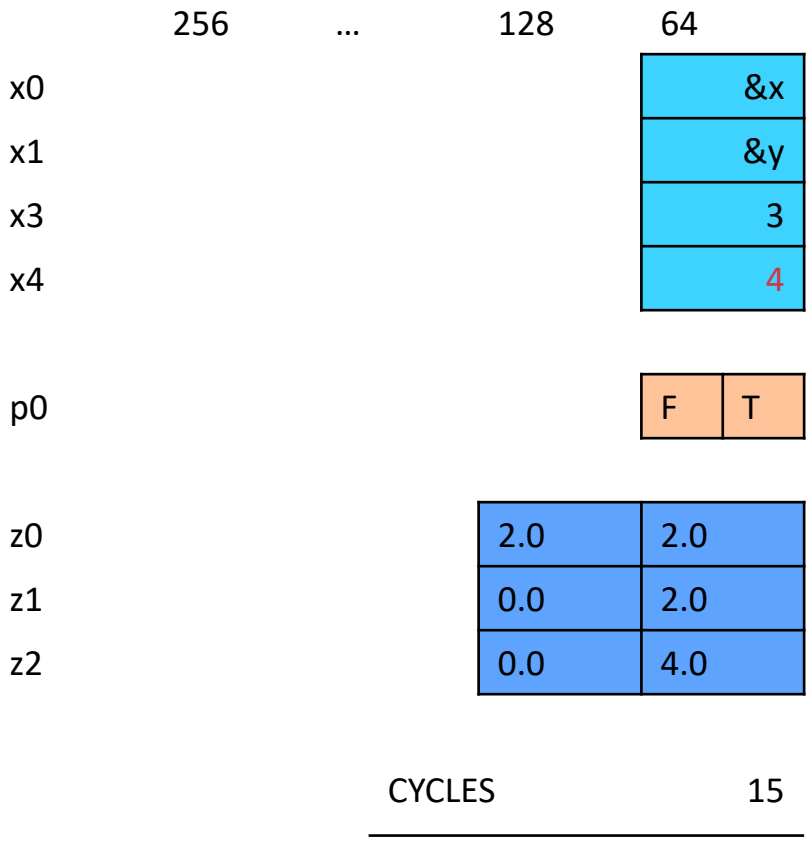
CYCLES 14

arm Research

daxpy (SVE – 128b)

```
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
```

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



CYCLES 15

arm Research

daxpy (SVE – 128b)

```
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
```

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

	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

arm Research

daxpy (SVE – 128b)

```
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
```

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

	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)

```
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
```

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

	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

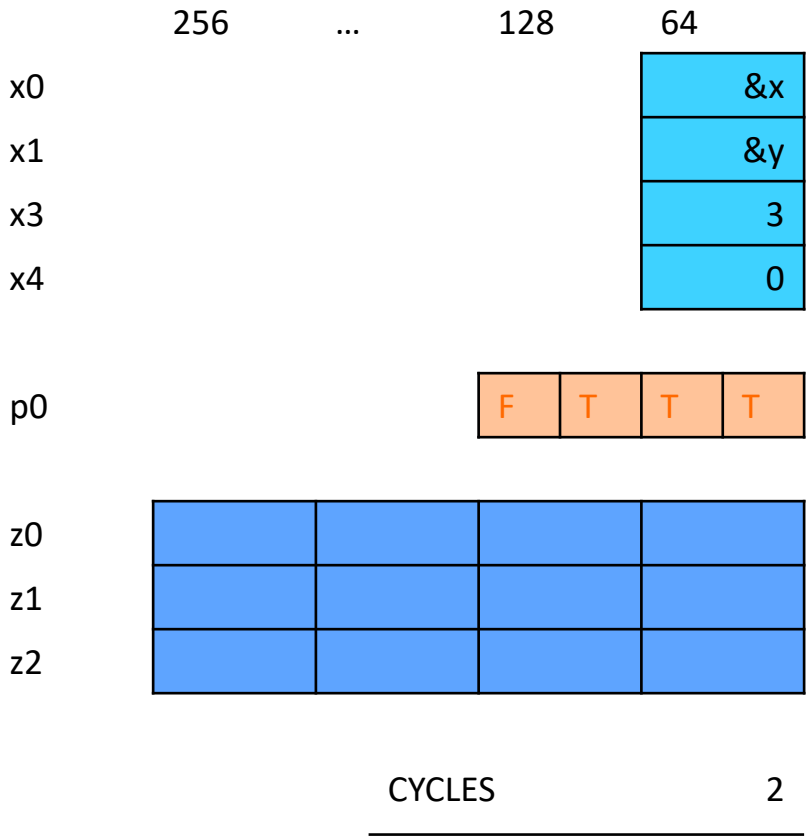
CYCLES 18

arm Research

daxpy (SVE – 256b)

```
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
```

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

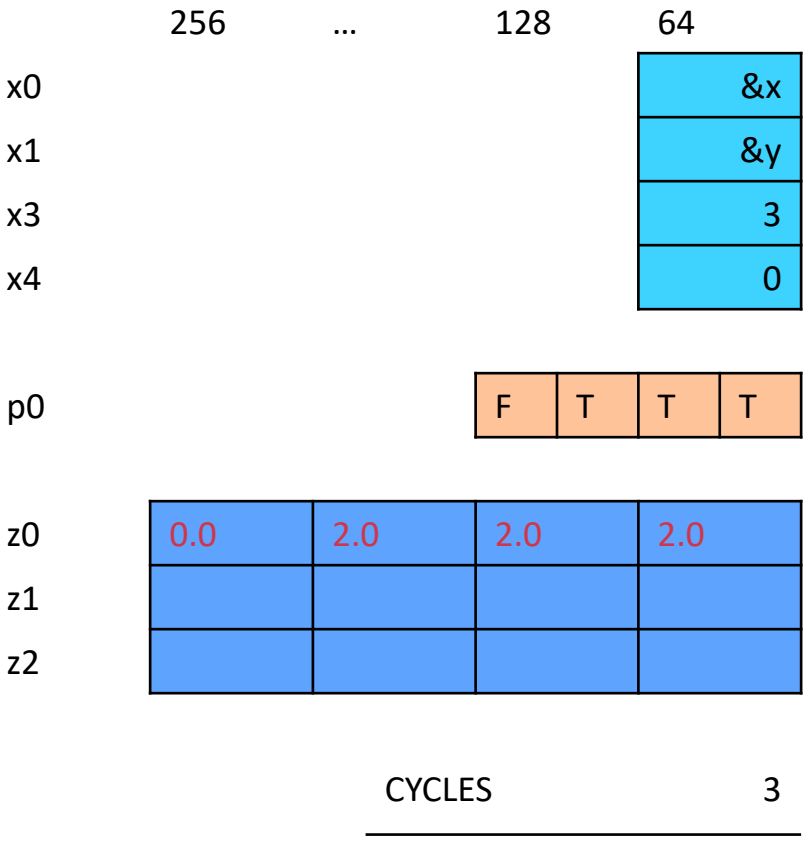


arm Research

daxpy (SVE – 256b)

```
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
```

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

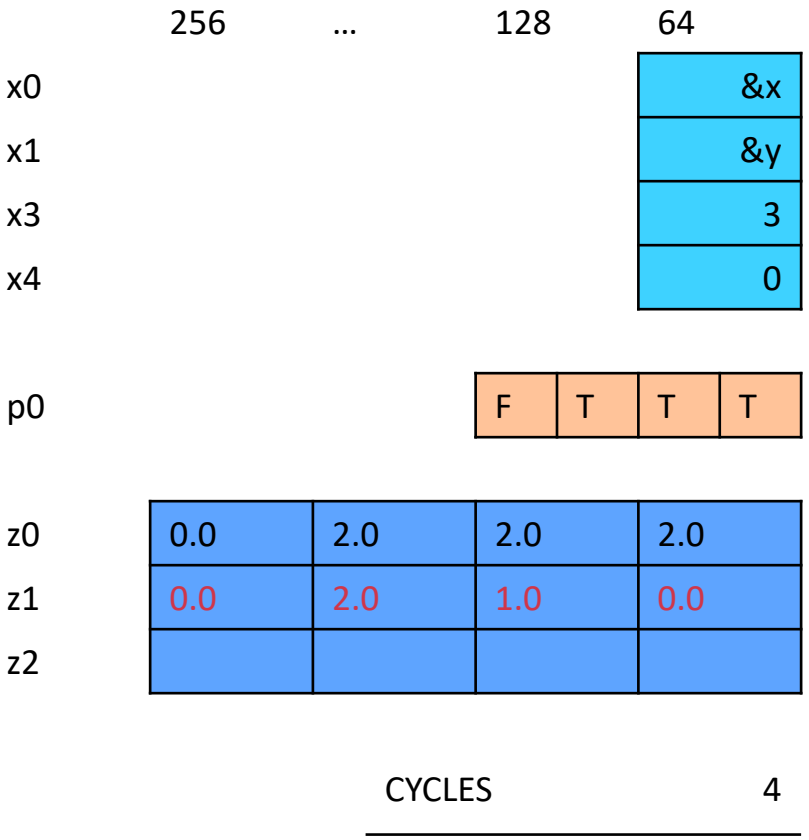


arm Research

daxpy (SVE – 256b)

```
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
```

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



arm Research

daxpy (SVE – 256b)

```
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
```

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

	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

arm Research

daxpy (SVE – 256b)

```
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
```

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

	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

arm Research

daxpy (SVE – 256b)

```
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
```

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

	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

arm Research

daxpy (SVE – 256b)

```
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
```

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

	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)

```
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
```

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

	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

arm Research

daxpy (SVE – 256b)

```
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
```

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

	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

arm Research

daxpy (SVE – 256b)

```
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
```

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

	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	11

arm Research

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

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)

```c
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
```

Simplified and suboptimal implementation

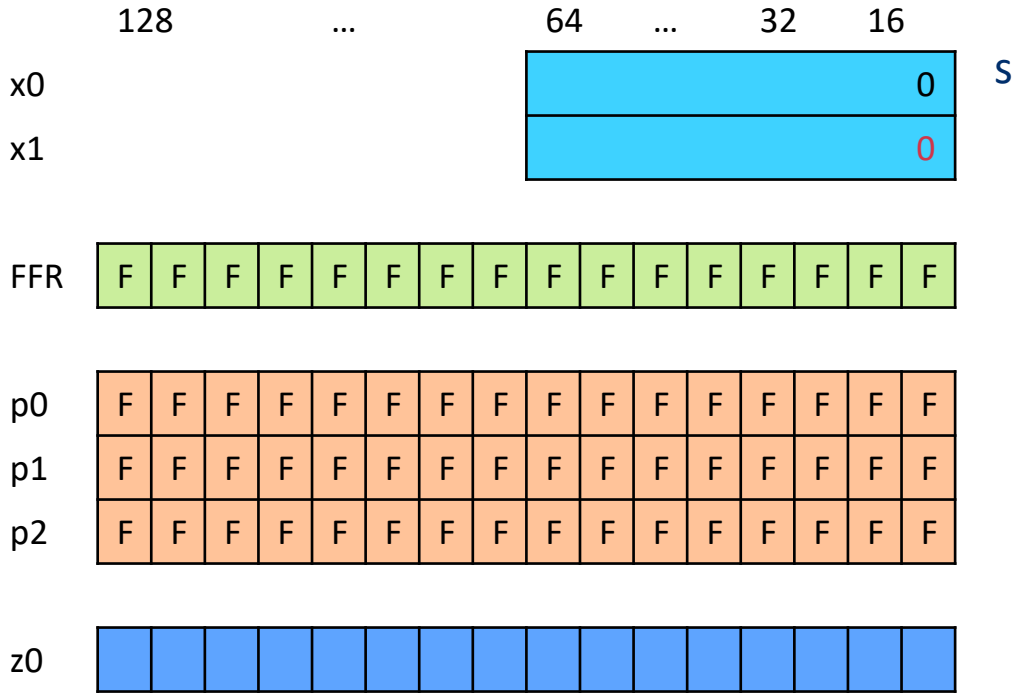
strlen (scalar)

```
// 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	‘s’	‘n’	‘r’	‘o’	‘h’	‘g’	‘n’	‘o ’	‘l’	‘ ’	‘e’	‘h’	‘t’

```
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
```



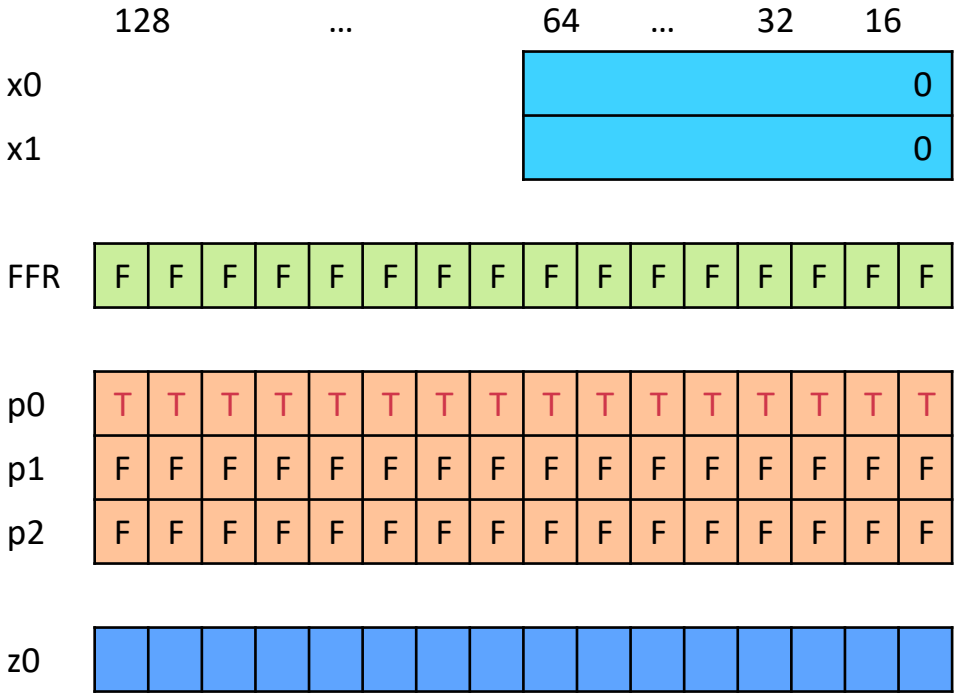
CYCLES 0

arm Research

strlen (SVE)

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

```
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
```



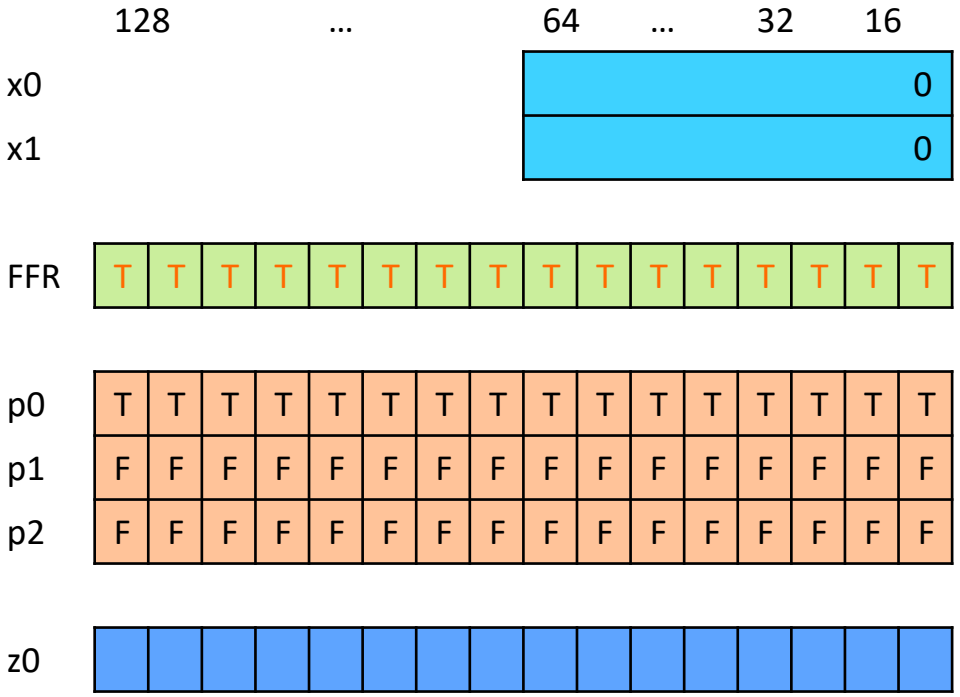
CYCLES 1

arm Research

strlen (SVE)

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

```
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
```



CYCLES 2

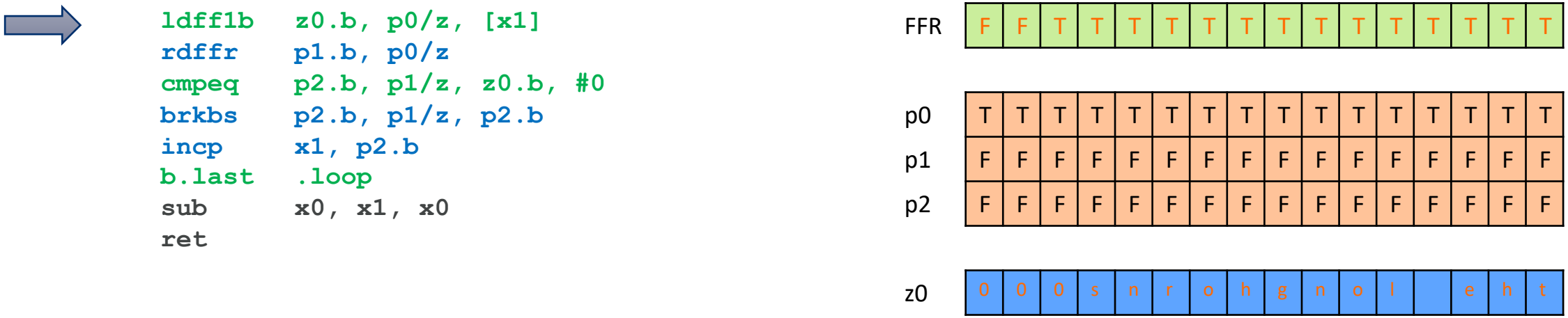
arm Research

strlen (SVE)

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

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
```



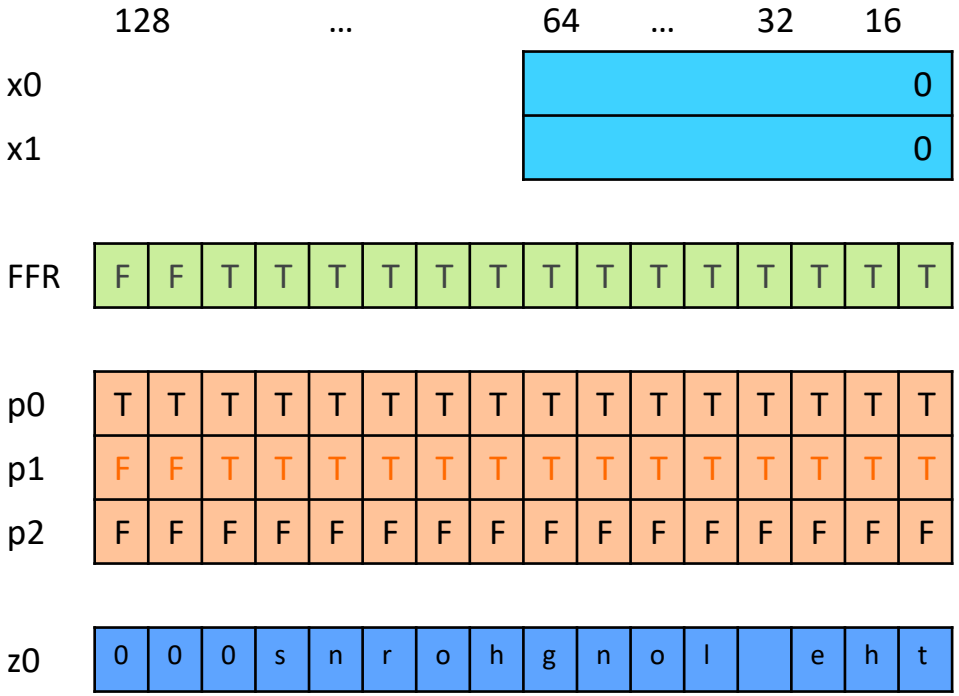
CYCLES 3

arm Research

strlen (SVE)

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

```
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
```



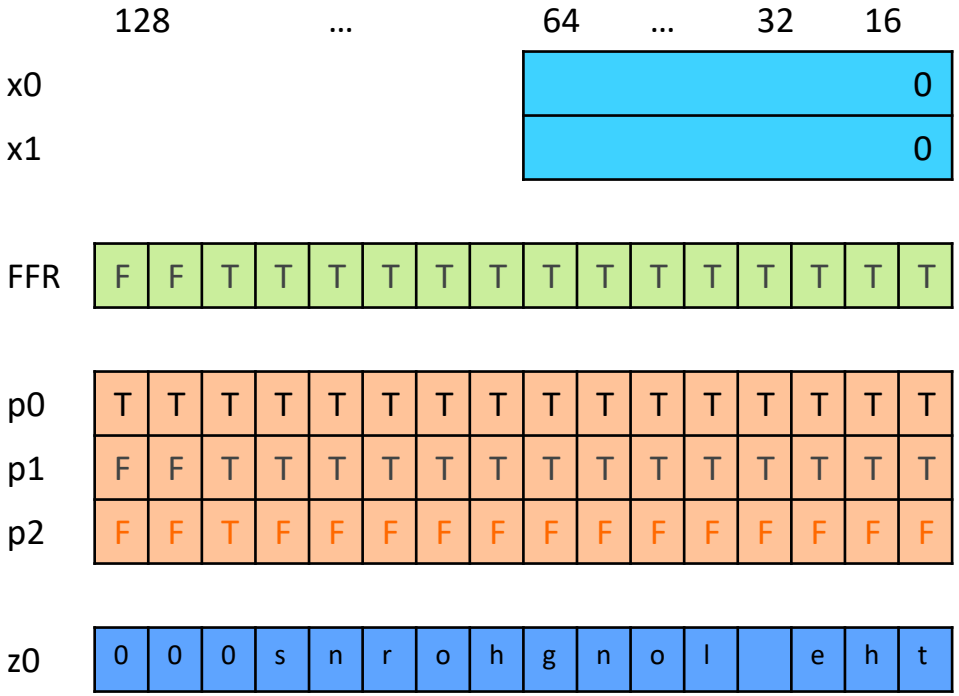
CYCLES 4

arm Research

strlen (SVE)

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

```
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
```



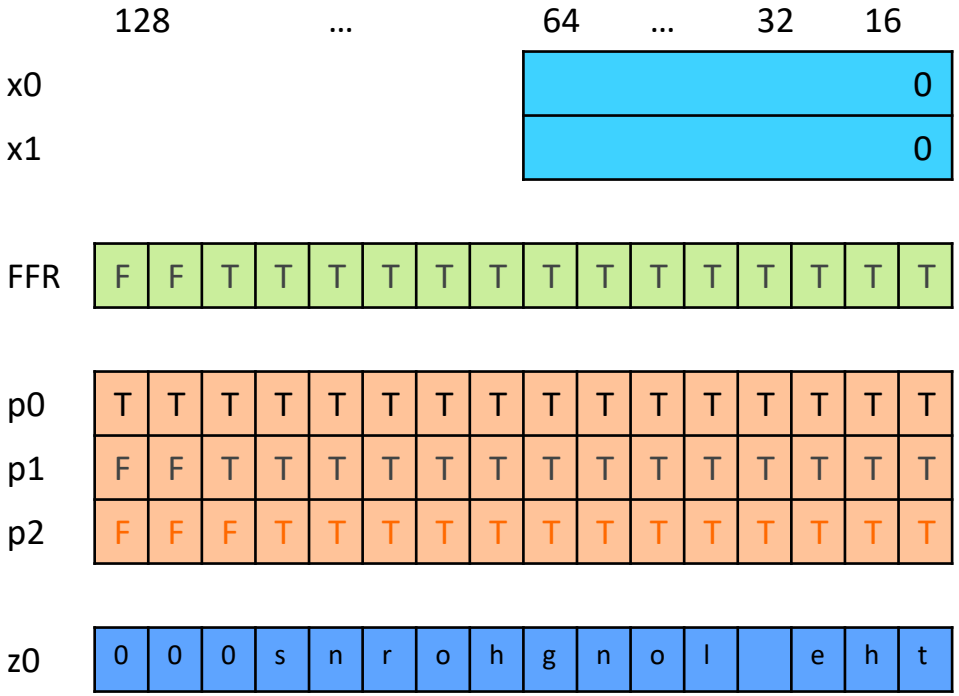
CYCLES 5

arm Research

strlen (SVE)

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

```
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
```



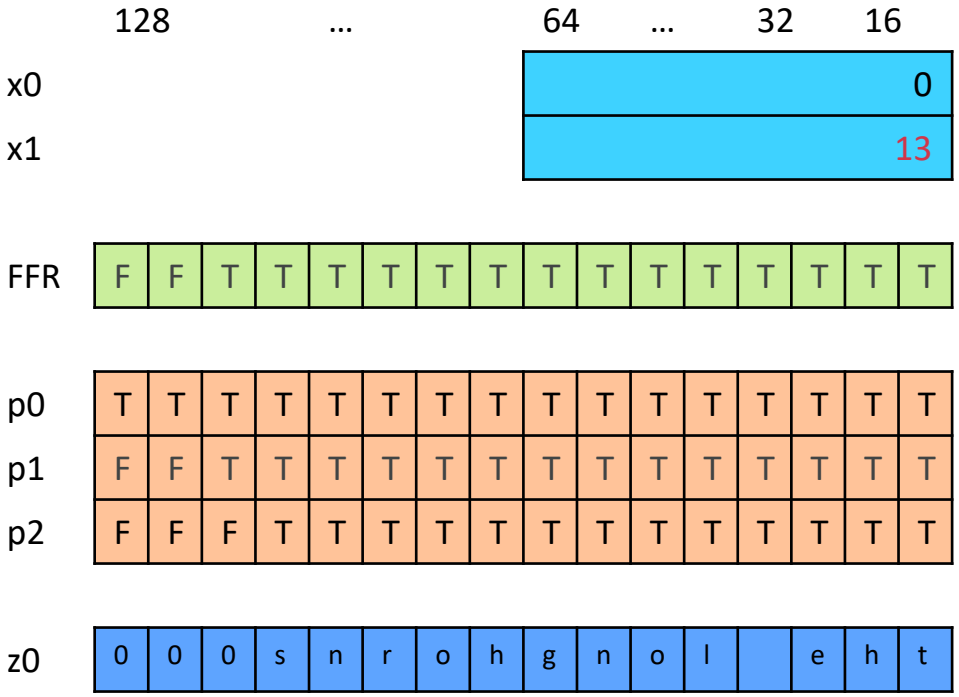
CYCLES 6

arm Research

strlen (SVE)

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

```
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
```



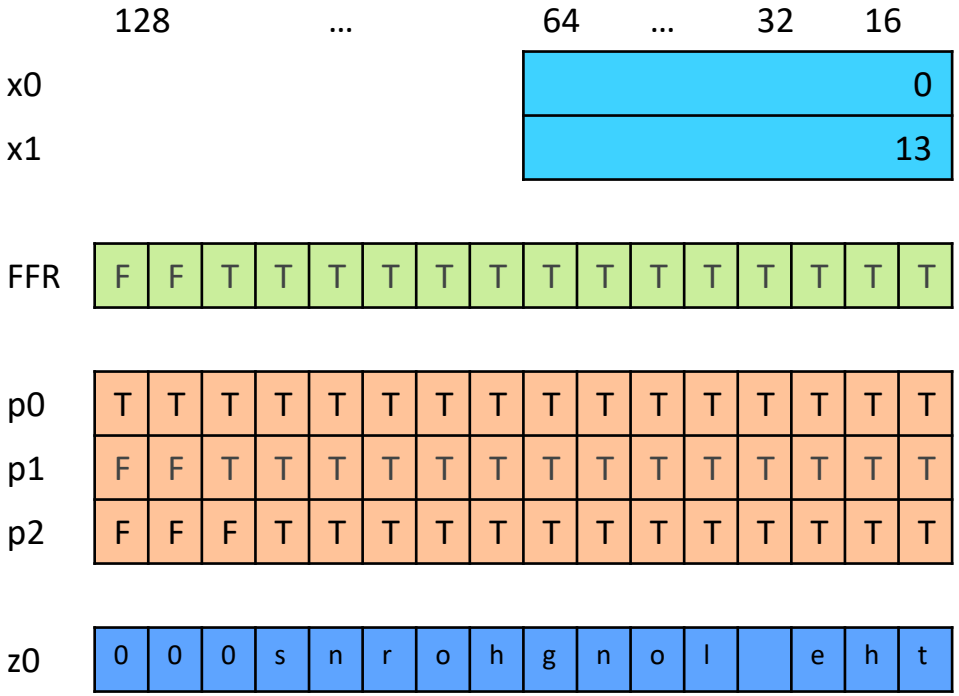
CYCLES 7

arm Research

strlen (SVE)

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

```
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
```



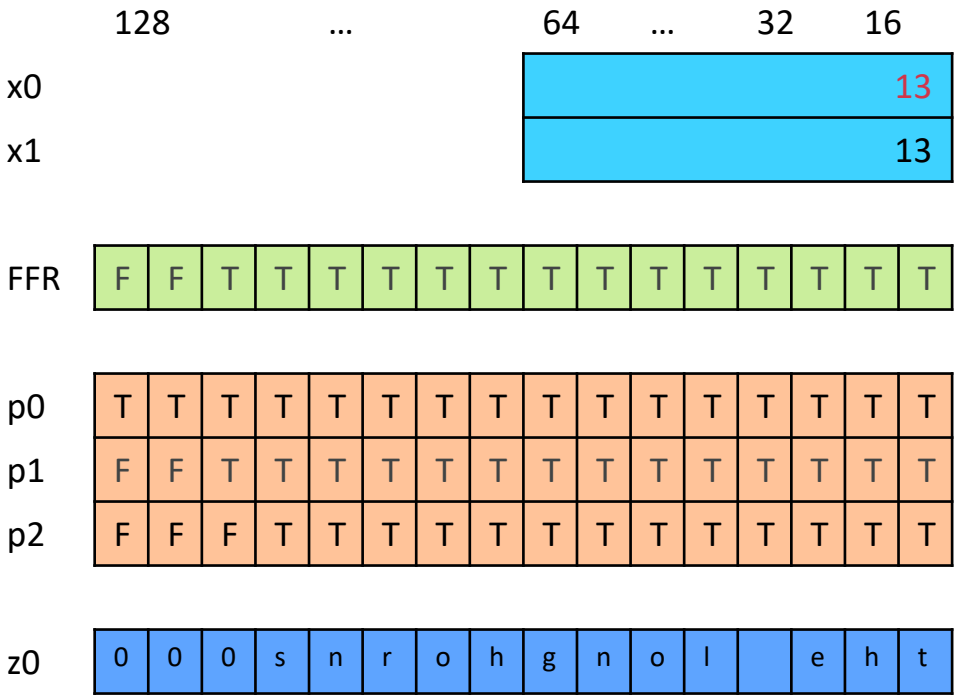
CYCLES	8

arm Research

strlen (SVE)

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

```
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
```



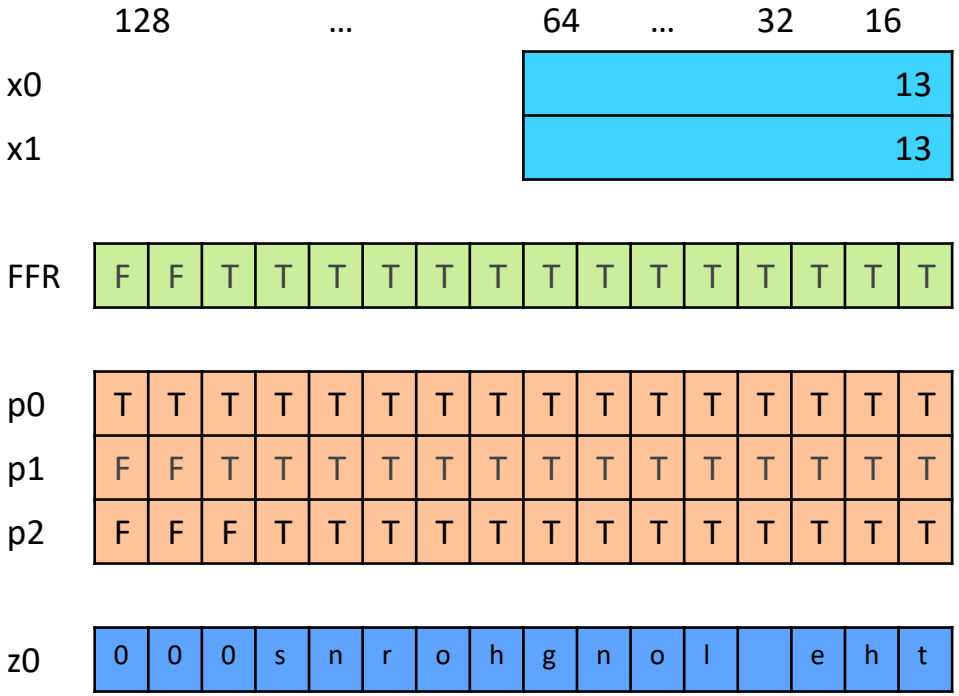
CYCLES 9

arm Research

strlen (SVE)

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

```
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
```



CYCLES	10

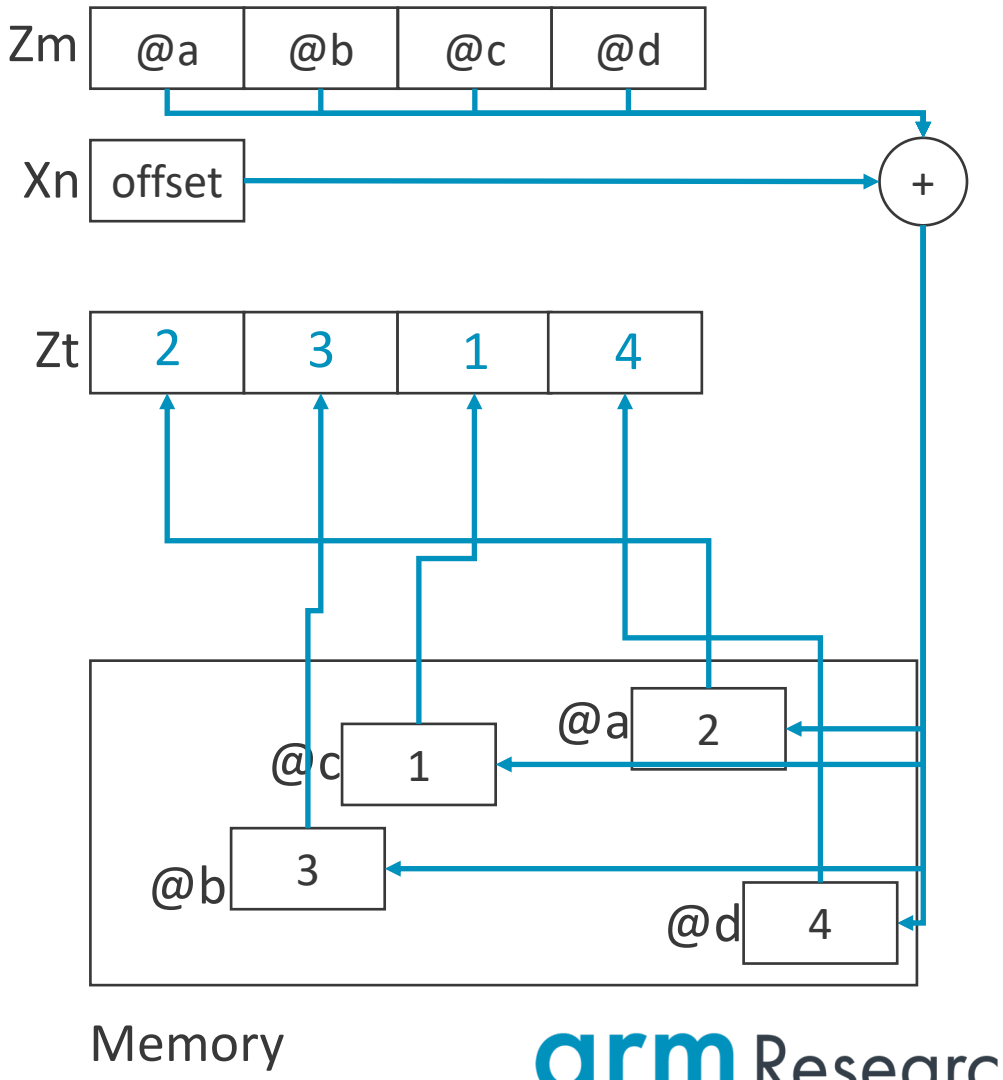
arm Research

Gather-Load & Scatter-Store

arm

Gather/Scatter Operations are Good and Evil

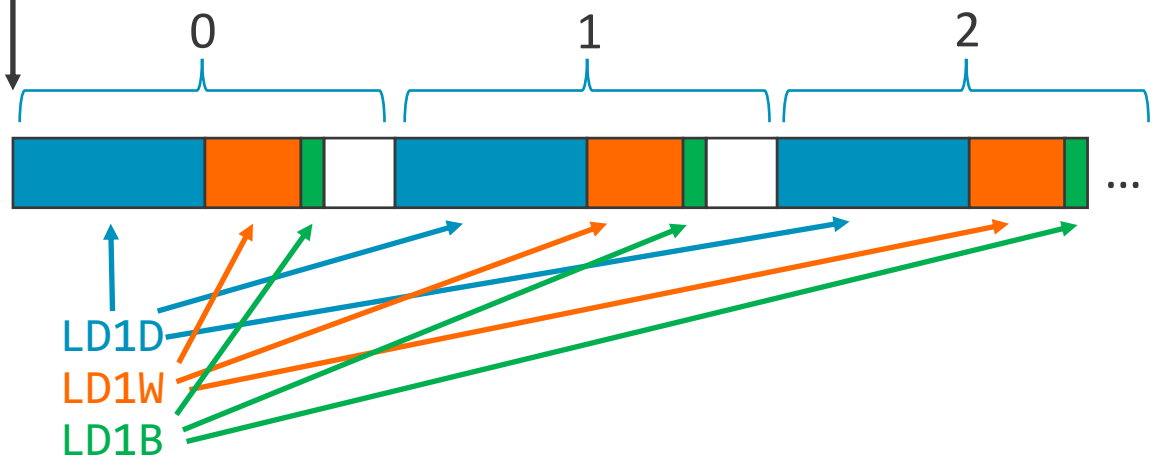
- Enable vectorization of codes with non-adjacent accesses on adjacent lanes
- Examples:
  - Outer loop vectorization
  - Strided accesses (larger than +1)
  - Random accesses
- Performance implementation dependent
  - Worst case one separate access per element
- `LD1D <Zt>.D, Ps/Z [<Xn>, <Zm>.D]`



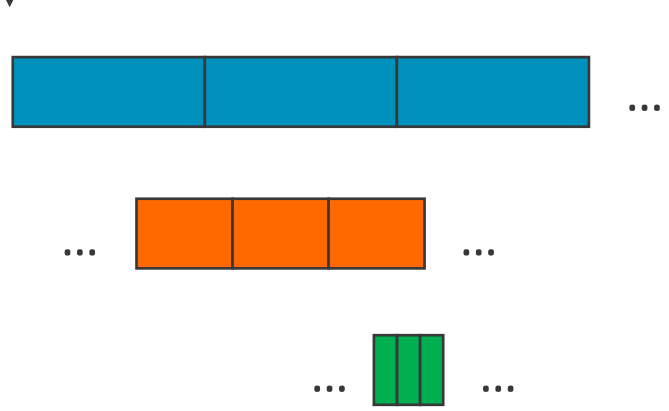
arm Research

Array of Structures vs. Structure of Arrays

```
typedef struct {
    uin64_t num_projects;
    float caffeine;
    bool vim_nemacs;
} Programmer_t;

Programmer_t programmers[N];
```



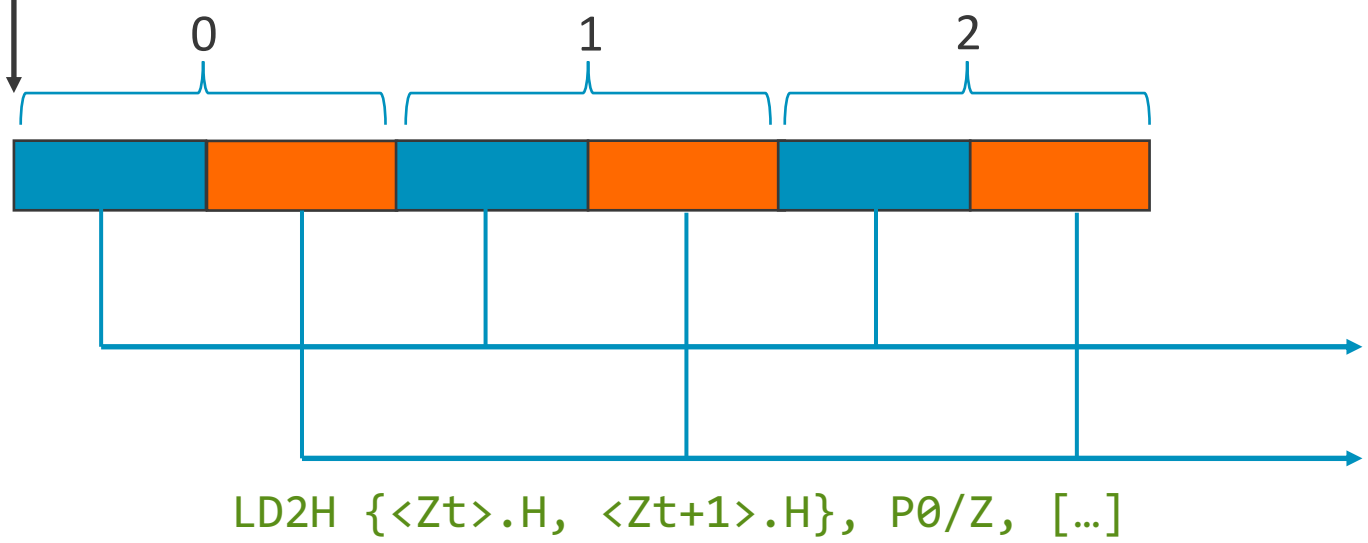

```
typedef struct {
    uin64_t num_projects[N];
    float caffeine[N];
    bool vim_nemacs[N];
} Programmer_t;

Programmer_t programmers;
```



arm Research

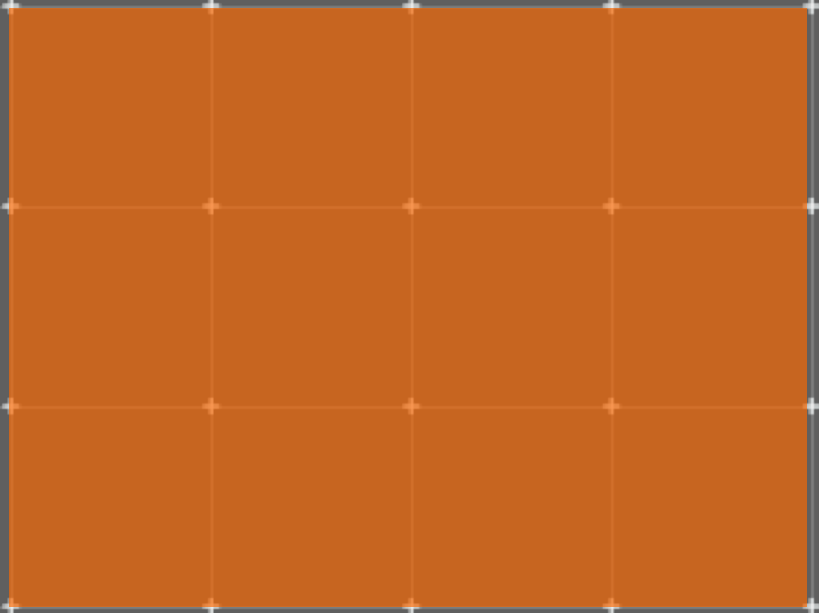
Contiguous Multi-Register Structure Loads/Stores

```
typedef struct {
    uint16_t num_students;
    uint16_t num_projects;
} ;

Researcher_t researchers[N];
```



arm Research

Non-temporal Loads & Stores



arm

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.*

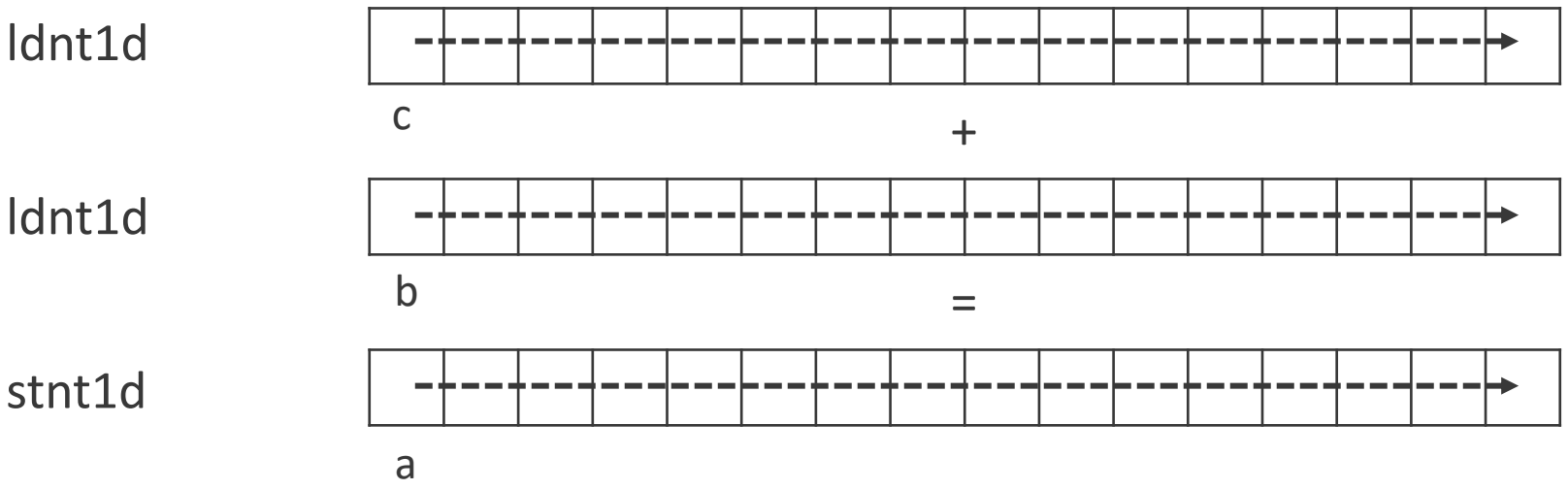
Being non-temporal is not enough

Vector Addition

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

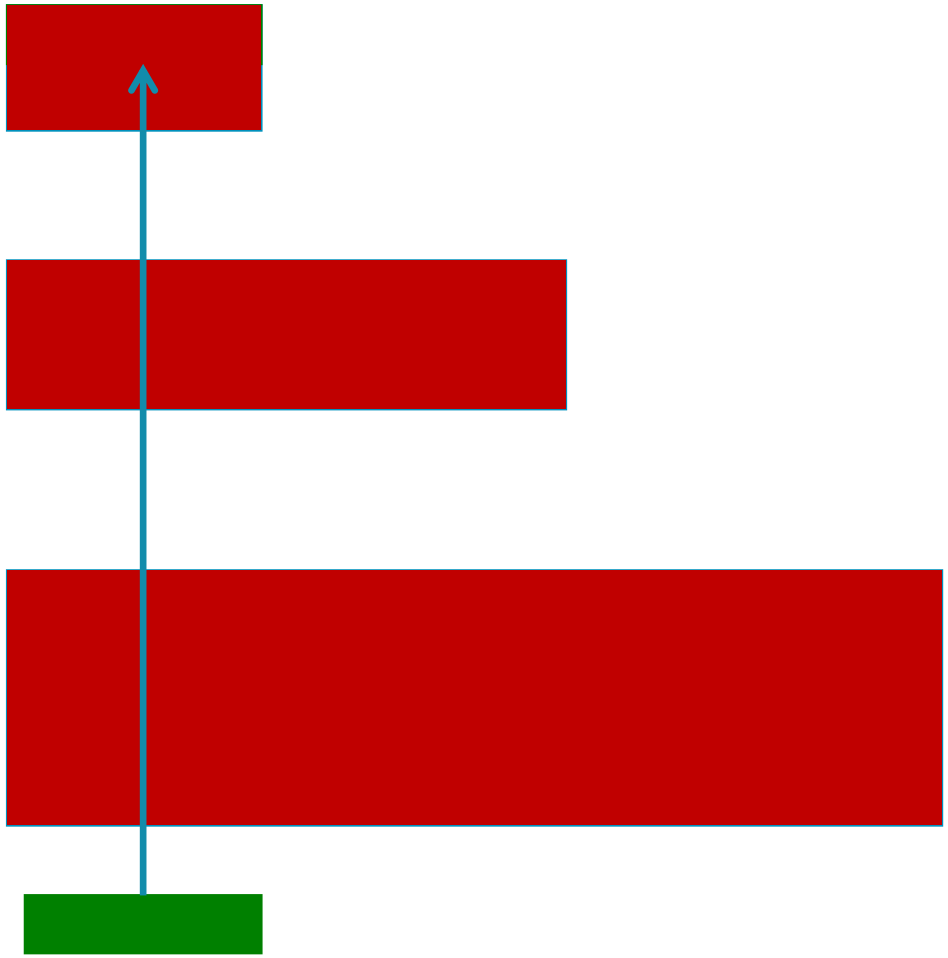
No benefit if all accesses are temporal

Target to leave space for *temporal* accesses

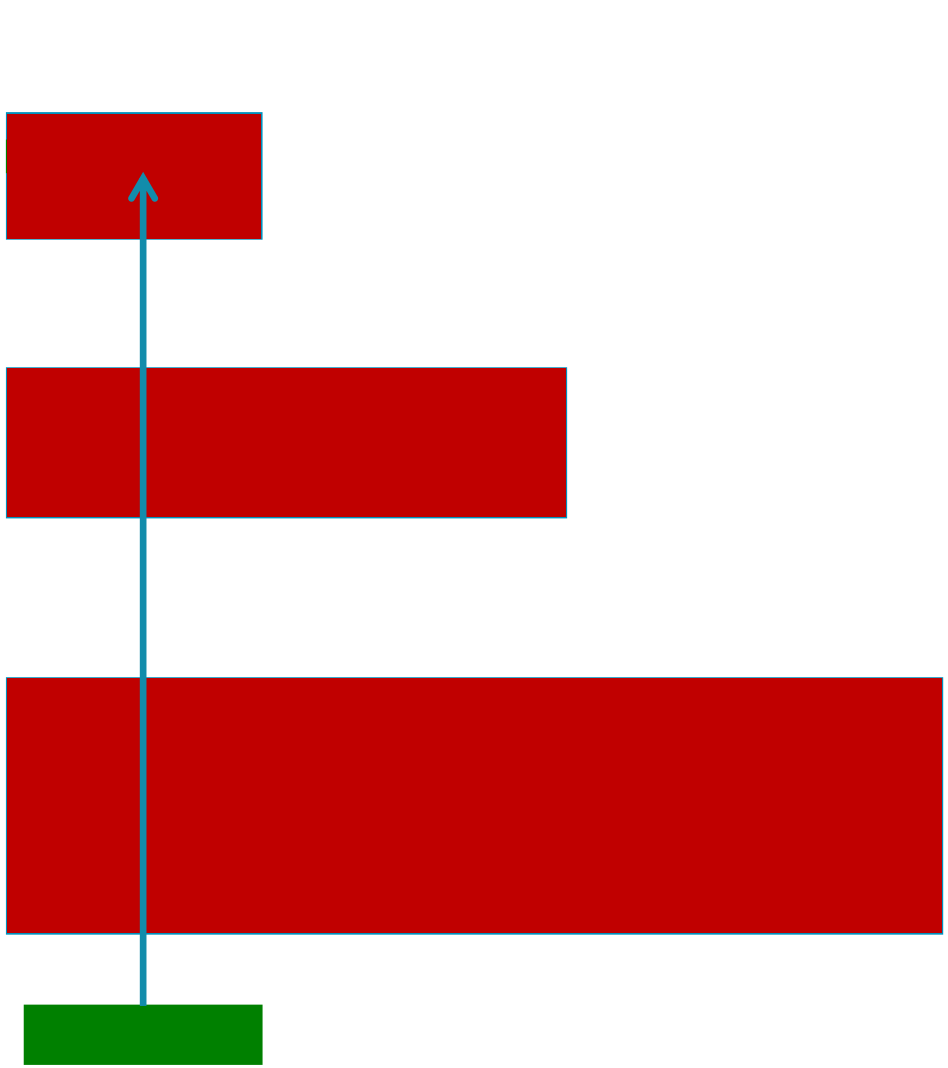


arm Research

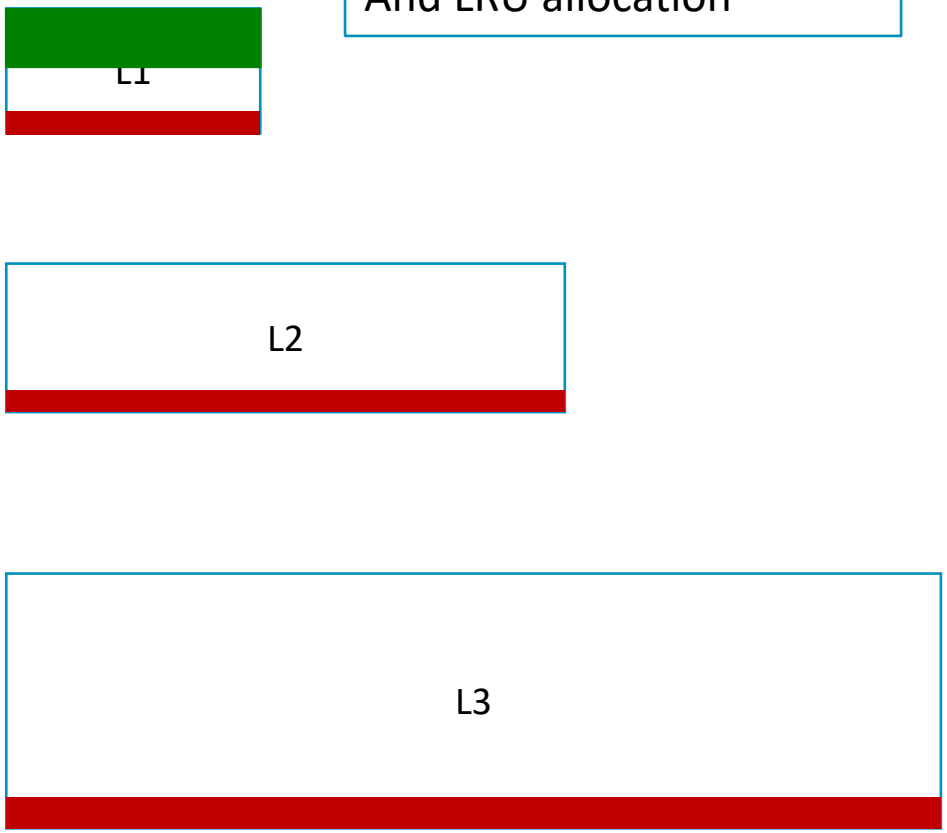
Mixed temporal and non-temporal



Mixed temporal and non-temporal



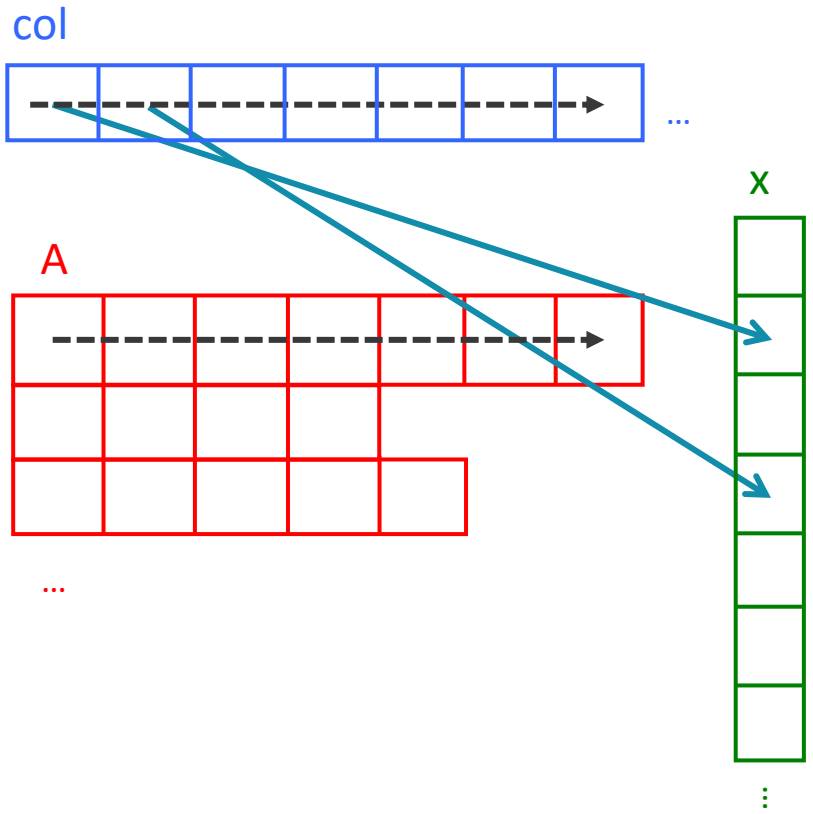
With non-temporal gather
And LRU allocation



arm Research

Sparse Matrix Vector

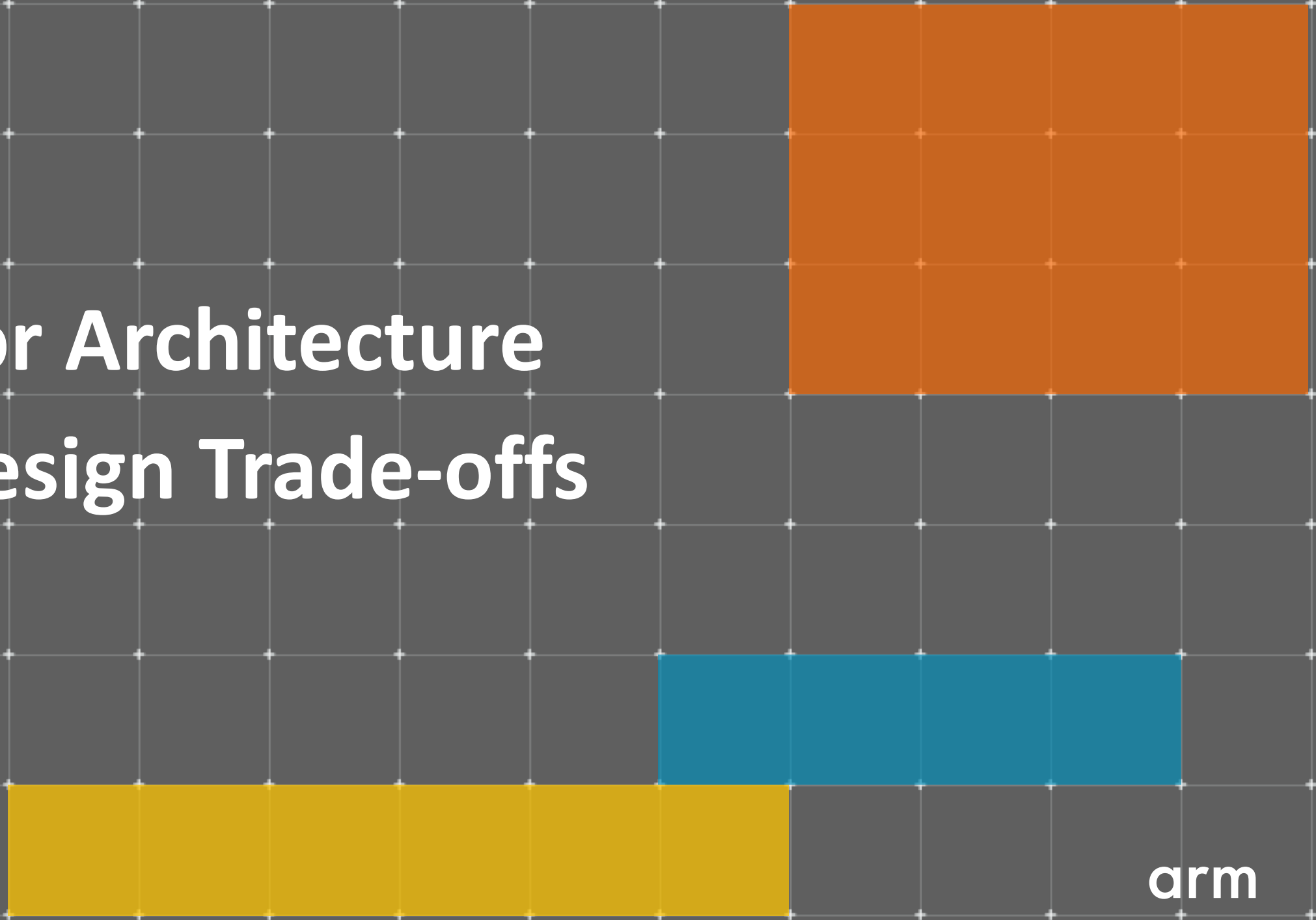
```
for(m=row_start[j]; m<row_start[j+1]; m++)
          y[j] += A[m] * x[col[m]];
```




```
                                        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]
```

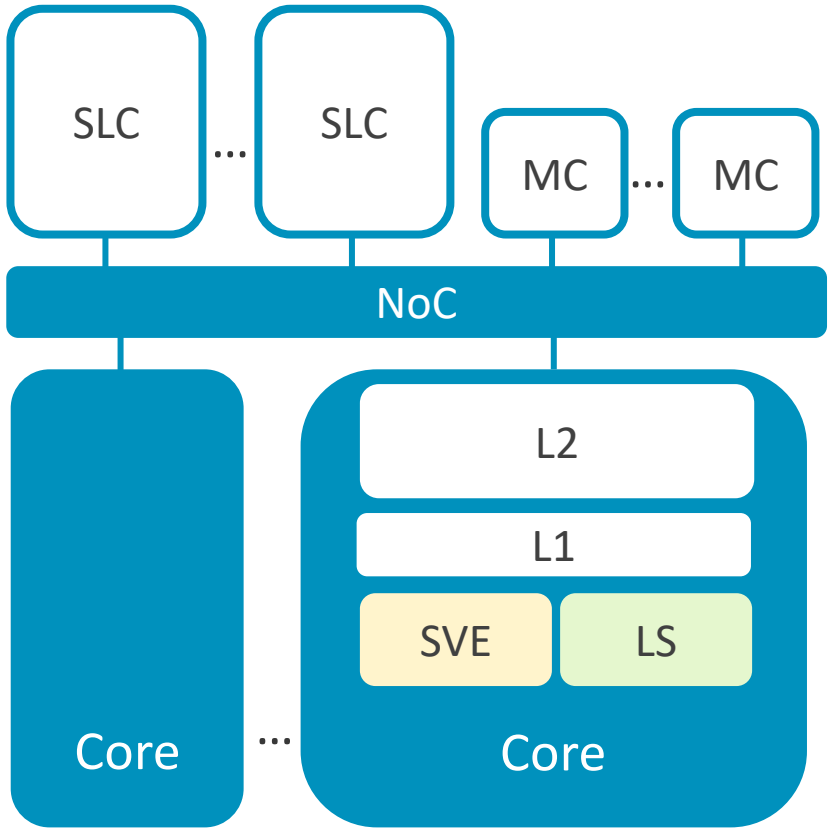

arm Research

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

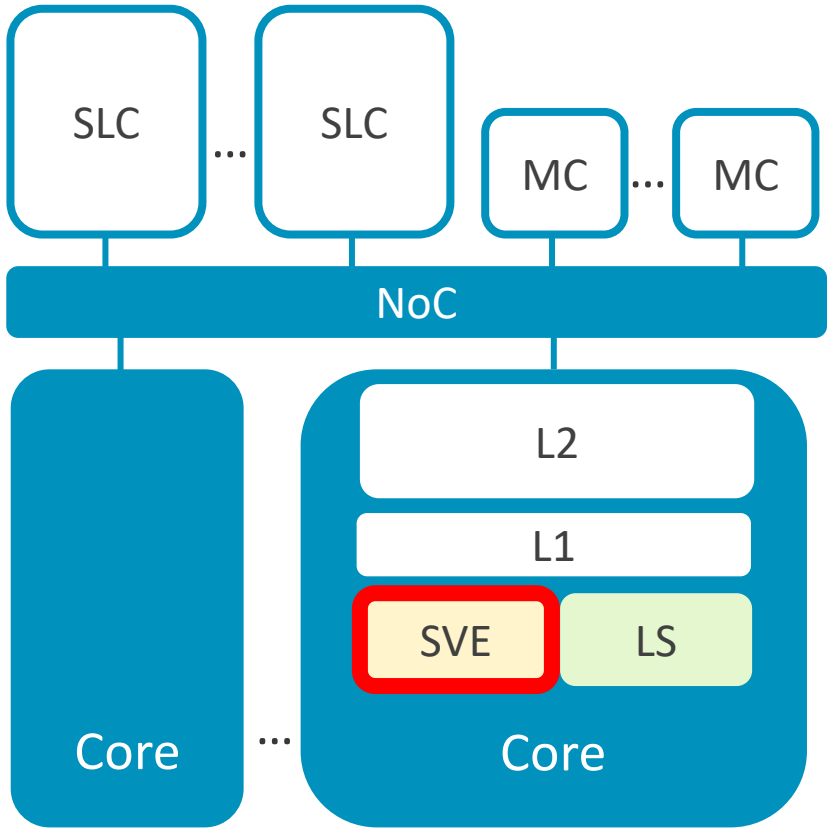
Memory controllers (MC)

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

Disclaimer: Logical representation, not representative of a physical implementation

arm Research

SVE Execution Pipeline



Vector length

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

Number of execution units and width

- Determines computation throughput

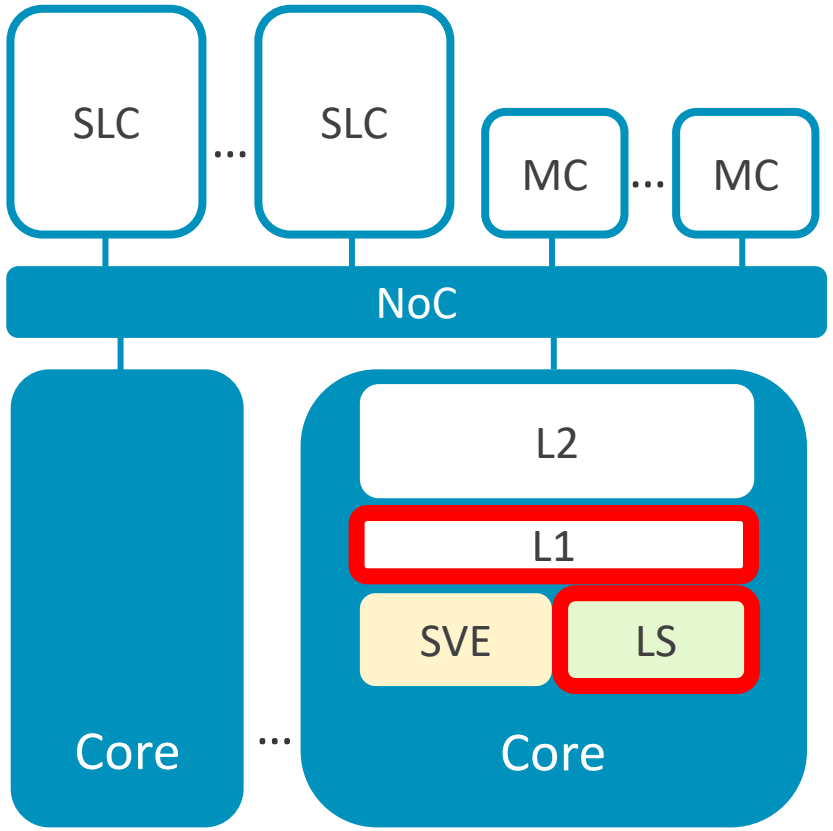
Vector instruction latencies, cracking, etc…

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

arm Research

Load-Store Execution Pipeline



Number of Load-Store execution units

L1 maximum access width

L1 concurrent accesses

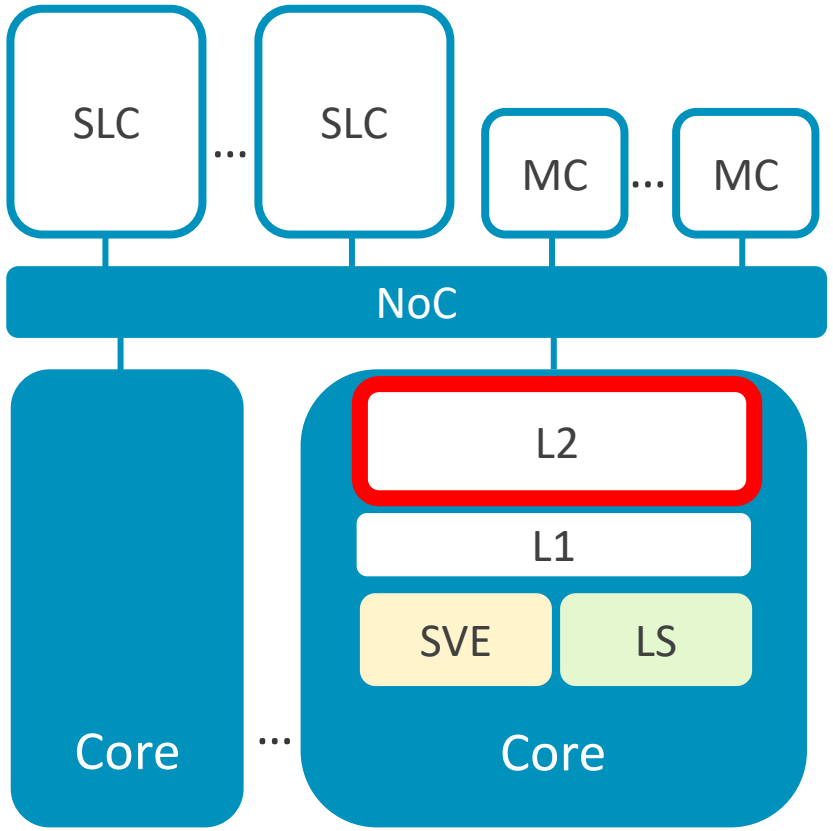
- Number of ports
- Number of banks/arrays

L1 cache size

Prefetching aggressiveness

arm Research

L2 Cache

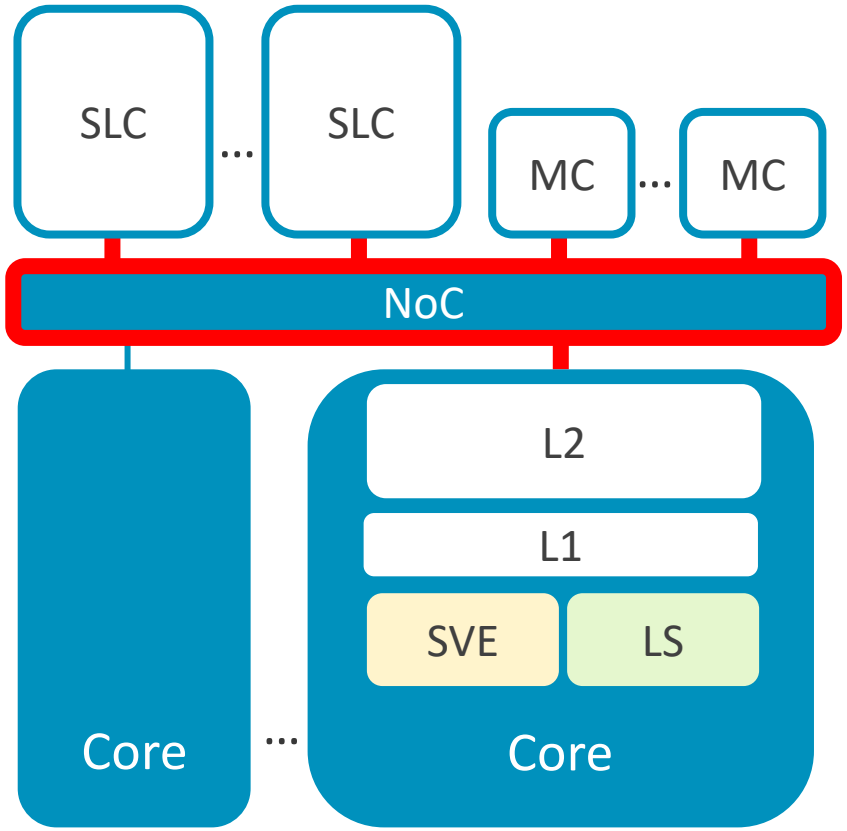


L2 size to filter NoC accesses

Prefetching aggressiveness

arm Research

Network on Chip



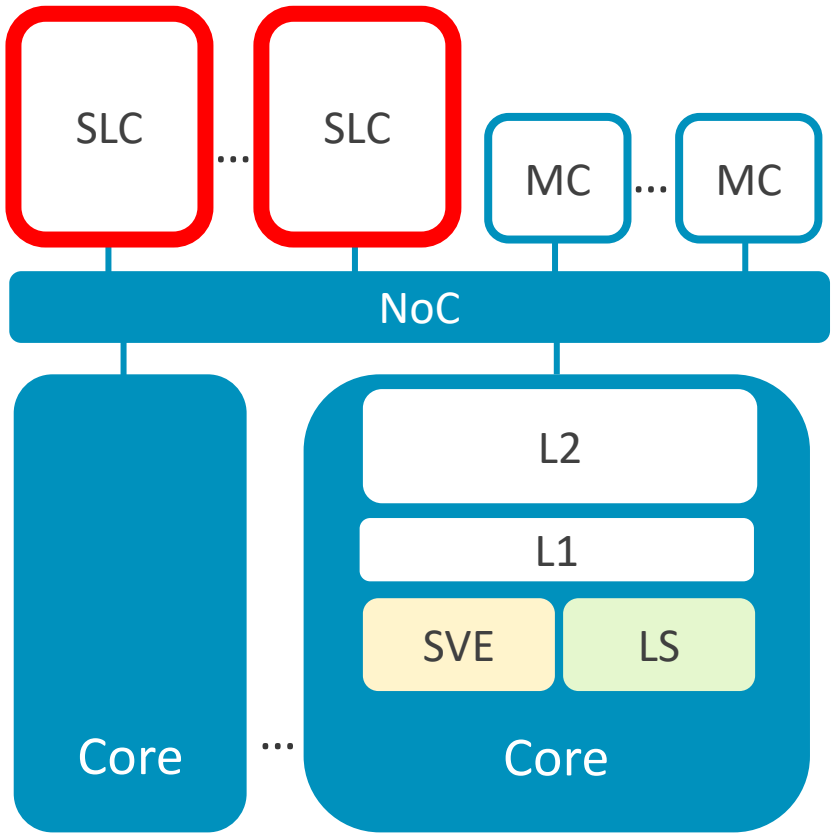
Bandwidth

Connectivity – topology, number of links

Routing to reduce congestion

arm Research

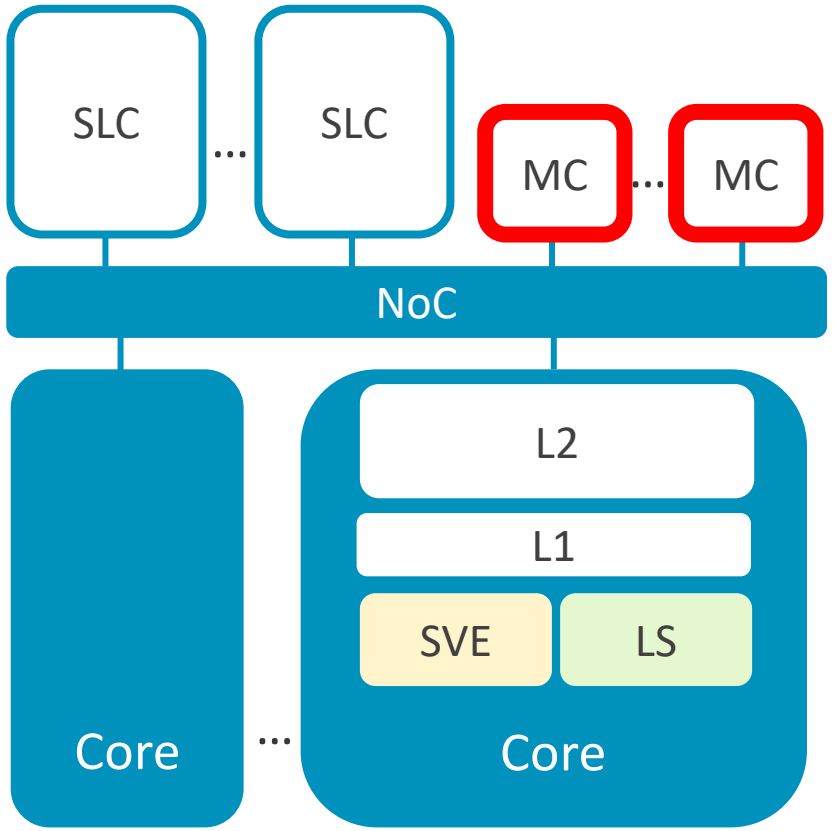
System Level Cache



SLC size, prefetching, replacement to filter main memory accesses

arm Research

Memory



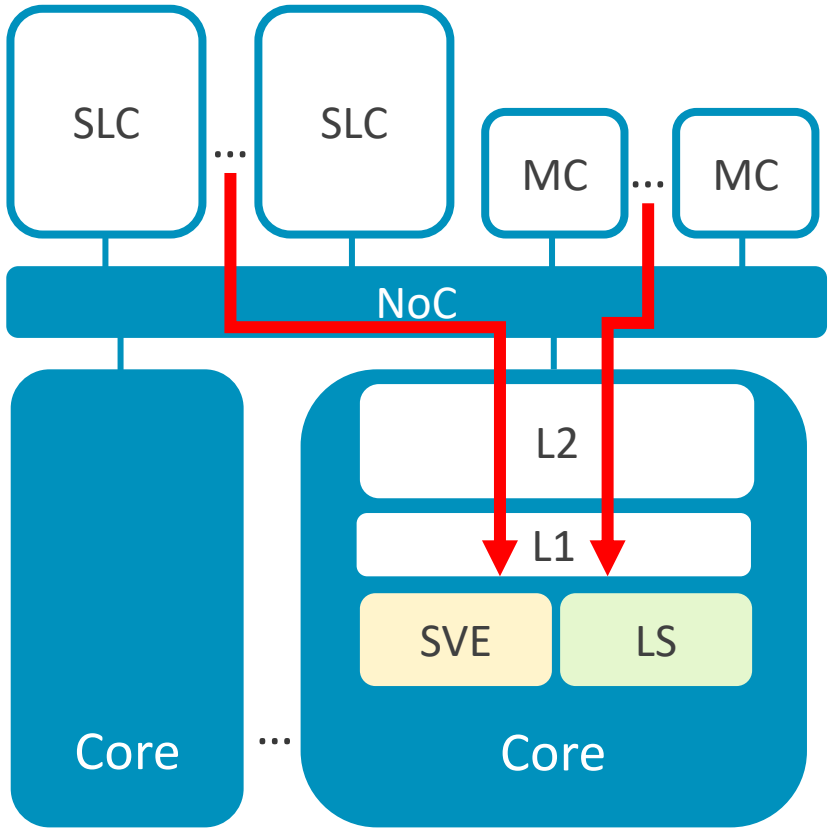
Memory bandwidth

- Channels, banks, width,…

HBM vs DRAM vs NVM…

arm Research

Basic Concept

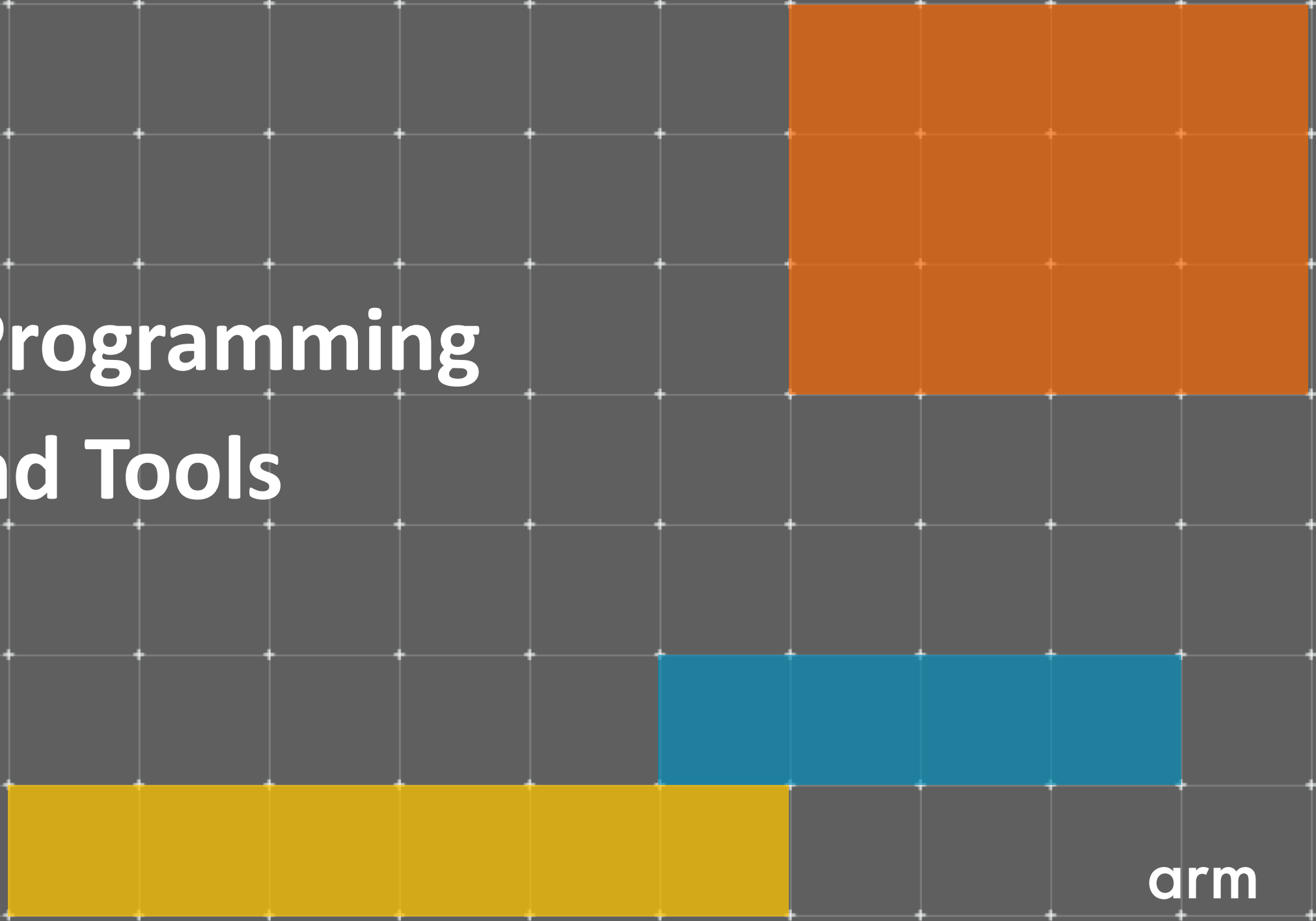


← Memory hierarchy and NoC to feed data to SVE units

← SVE unit configuration for target throughput

arm Research

SVE Programming and Tools



arm

SVE Programming

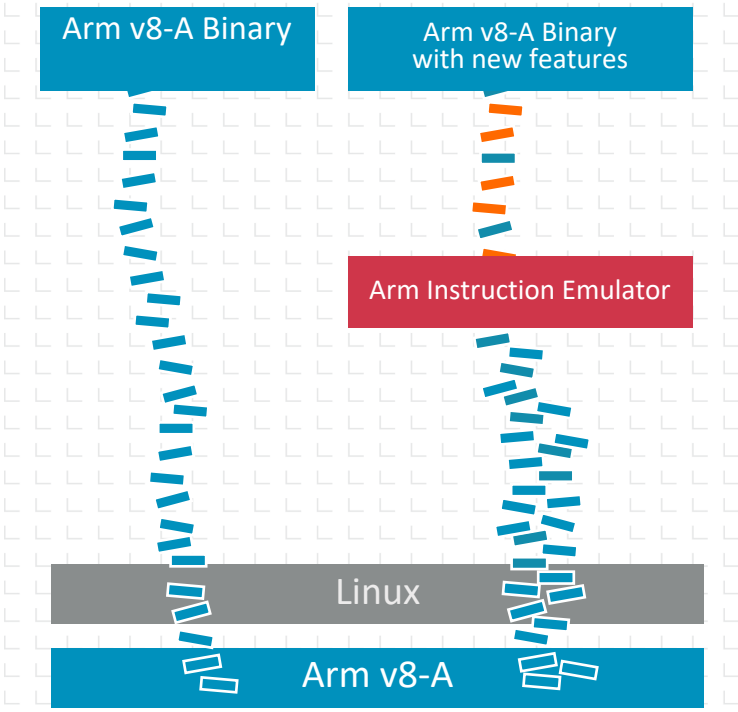
Assembly	Full ISA Specification: The Scalable Vector Extension for Armv8-A Lots of worked examples in A sneak peek into SVE and VLA programming
Intrinsics	Arm C Language Extensions for SVE Arm Scalable Vector Extensions and application to Machine Learning
Compiler	Autovectorization – GCC, Arm Compiler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

SVE Tools

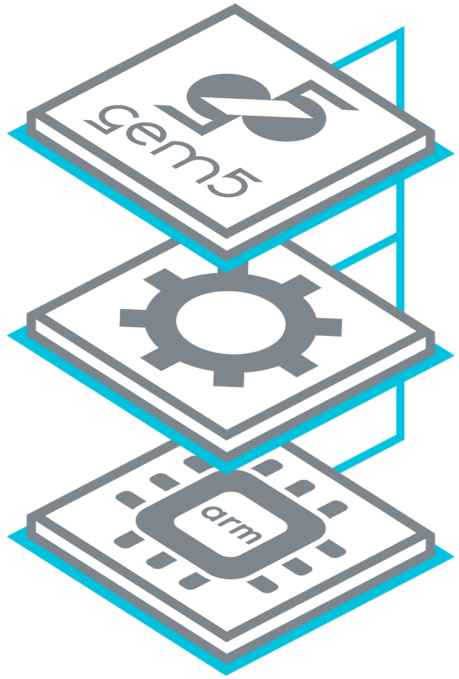
Arm Compiler

arm COMPILER

Arm Instruction Emulator



Research Enablement Kit



arm Research

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

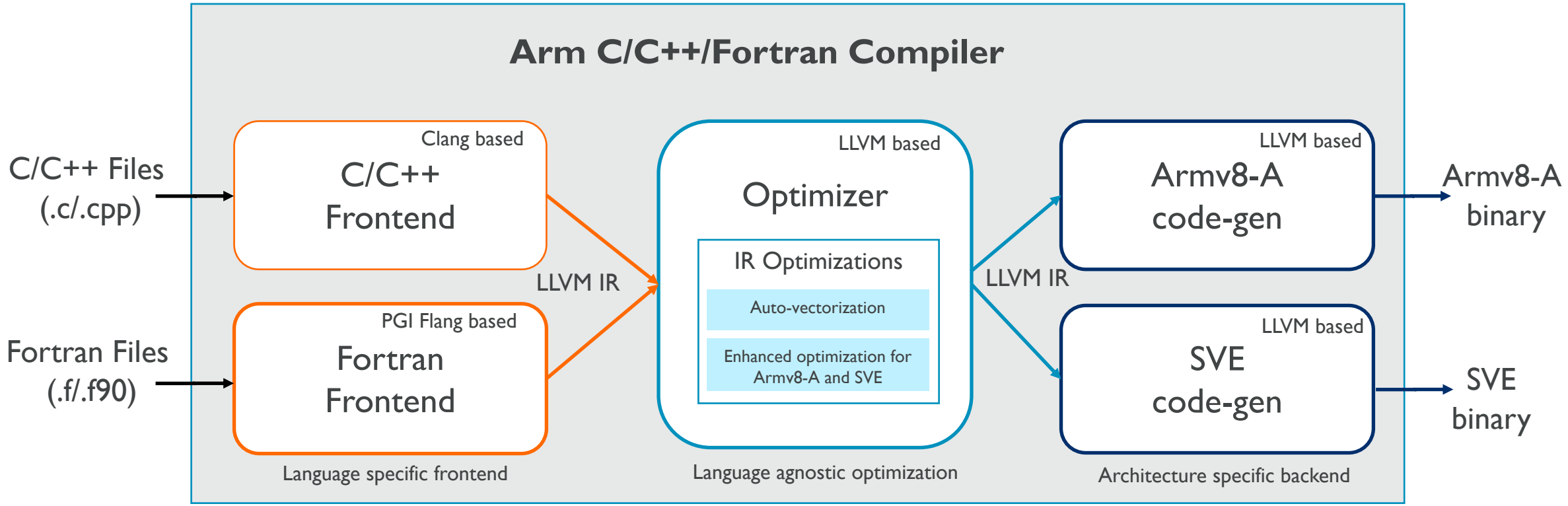
- C++ 14 and Fortran 2003 language support with OpenMP 4.5*
- Support for Armv8-A and SVE architecture extension
- Based on LLVM and Flang, leading open-source compiler projects

Commercially supported by Arm

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

arm Research

Arm Compiler – Building on LLVM, Clang and Flang projects



arm Research

SVE Compiler Support

Feature	Upstream GCC	Upstream LLVM	Arm Compiler 6 (For bare metal)	Arm HPC Compiler (for Linux user-space)
SVE asm and disasm	Yes	Yes	Yes	Yes
SVE code generation	Yes	No Planned for 2018-19	Yes	Yes
SVE Arm C Language Extensions (“intrinsics”)	No Planned for GCC9 (2019)	No Planned for 2018-19	Yes	Yes
Auto-vectorization	Basic More improvements planned for GCC9	None Planned for 2019-20	Advanced	Advanced

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 Research

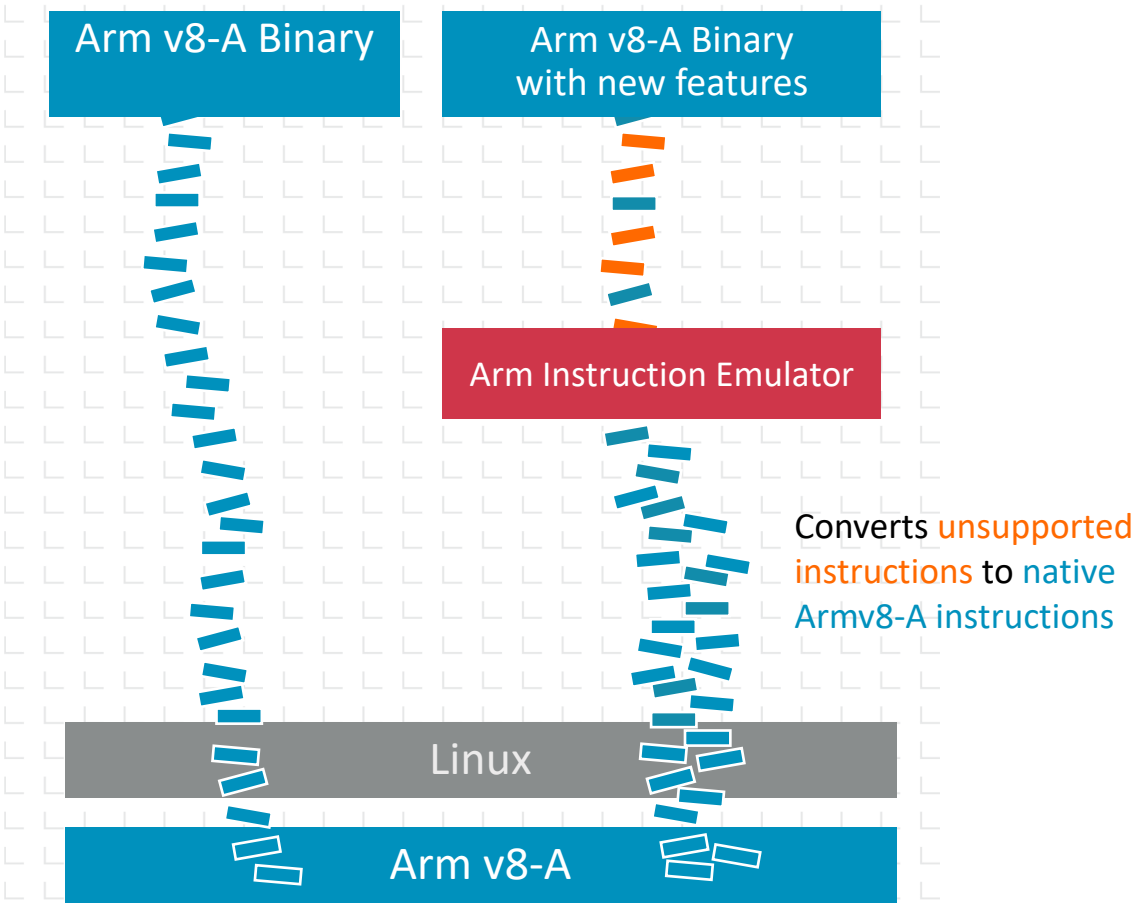
Arm Instruction Emulator

Develop your user-space applications for future hardware today

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

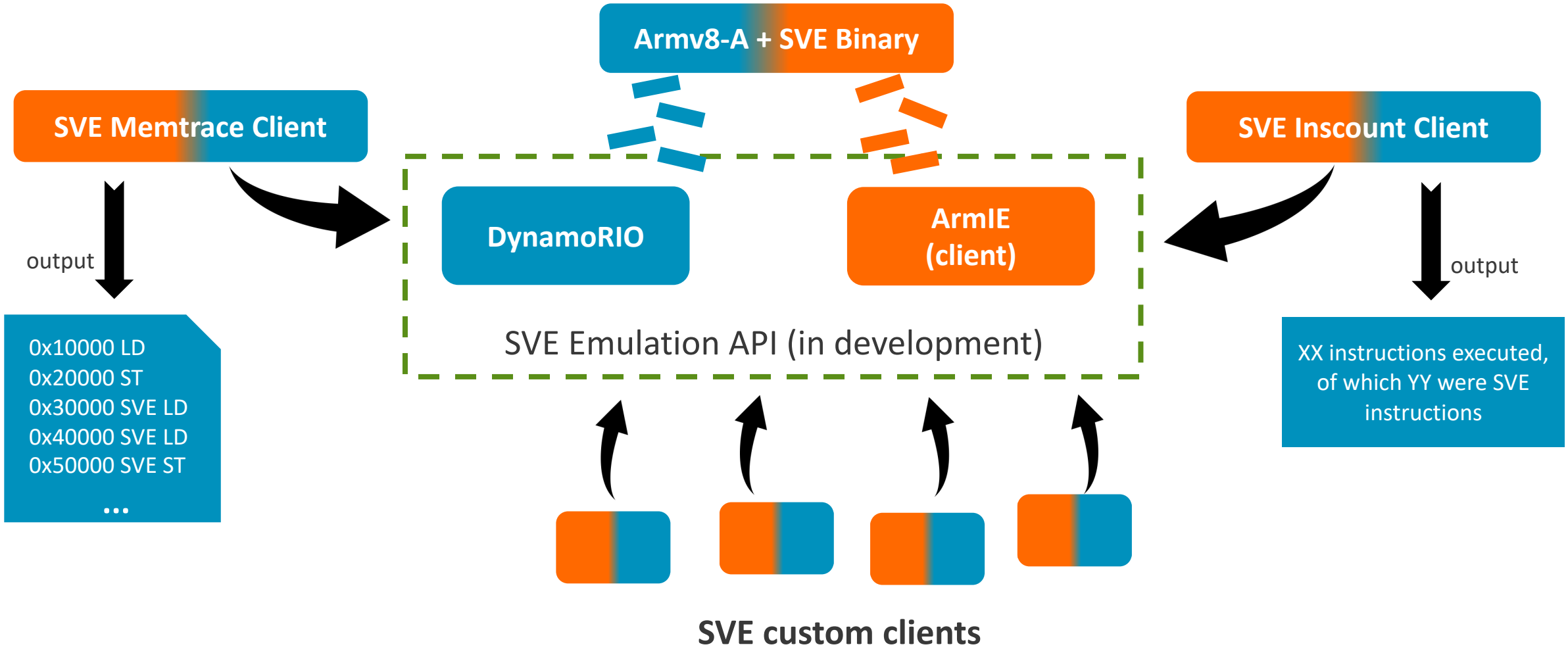
Simple “black box” command line tool

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



arm Research

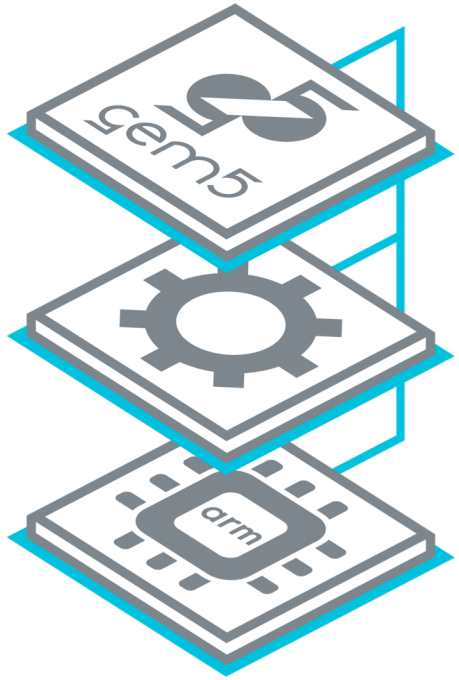
Instrumenting Aarch64 and SVE



arm Research

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

Internships & Full-time Opportunities

arm.com/careers

Twitter: @alexrico46

LinkedIn: alexrico

arm Research

arm Research

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!

arm Research

