HPC Asia 2019

arm

Arm SVE and Machine Learning

Francesco Petrogalli, Arm HPC compilers

Jan 14th, 2019

The most expensive operation of every Machine Learning algorithm is

Matrix Multiplication*

*some kind of.

SVE does not mandate a single, fixed vector length.

384-bit

128-bit

LEN x 128-bit

2048-bit

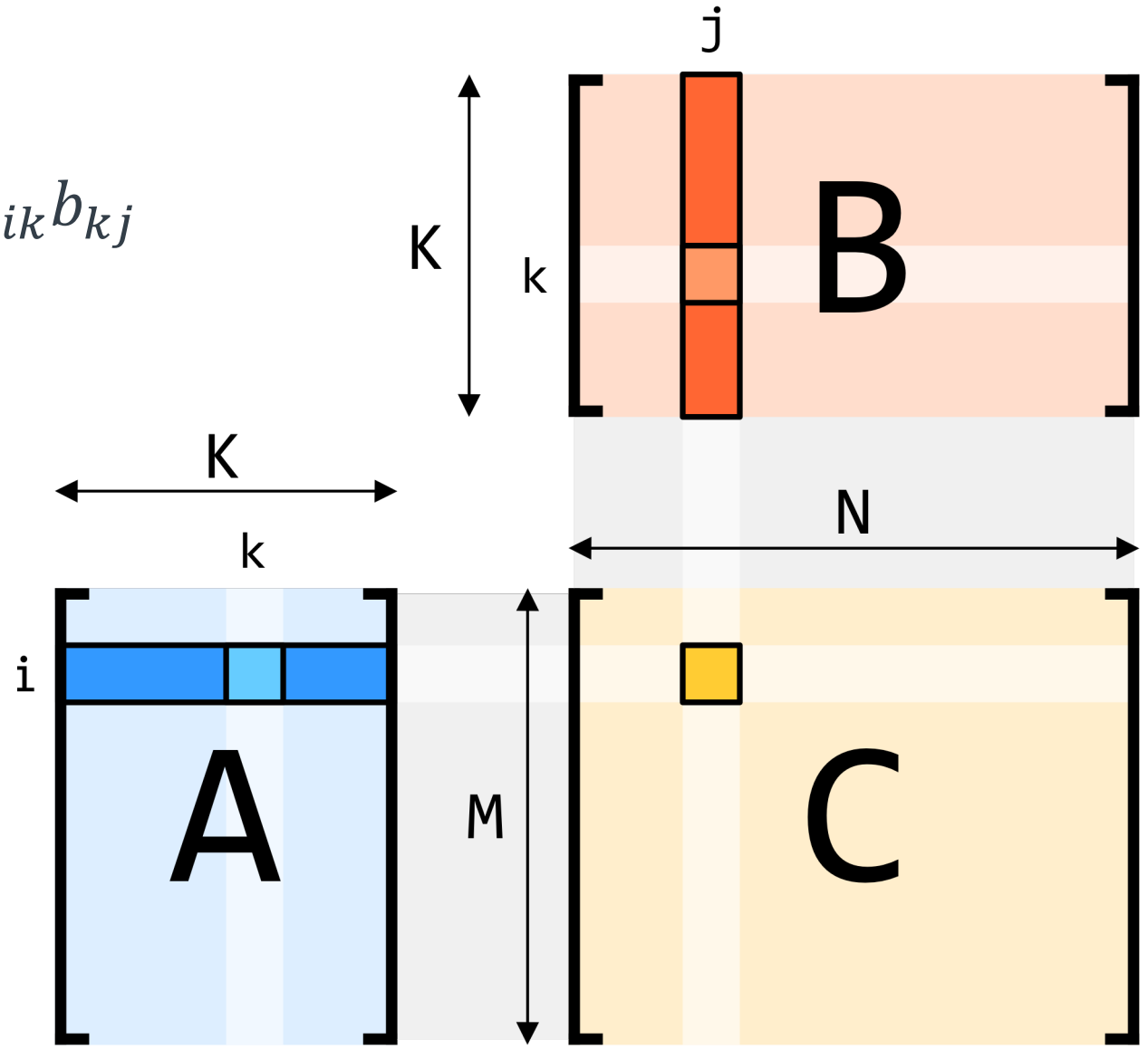
256-bit

1024-bit

arm

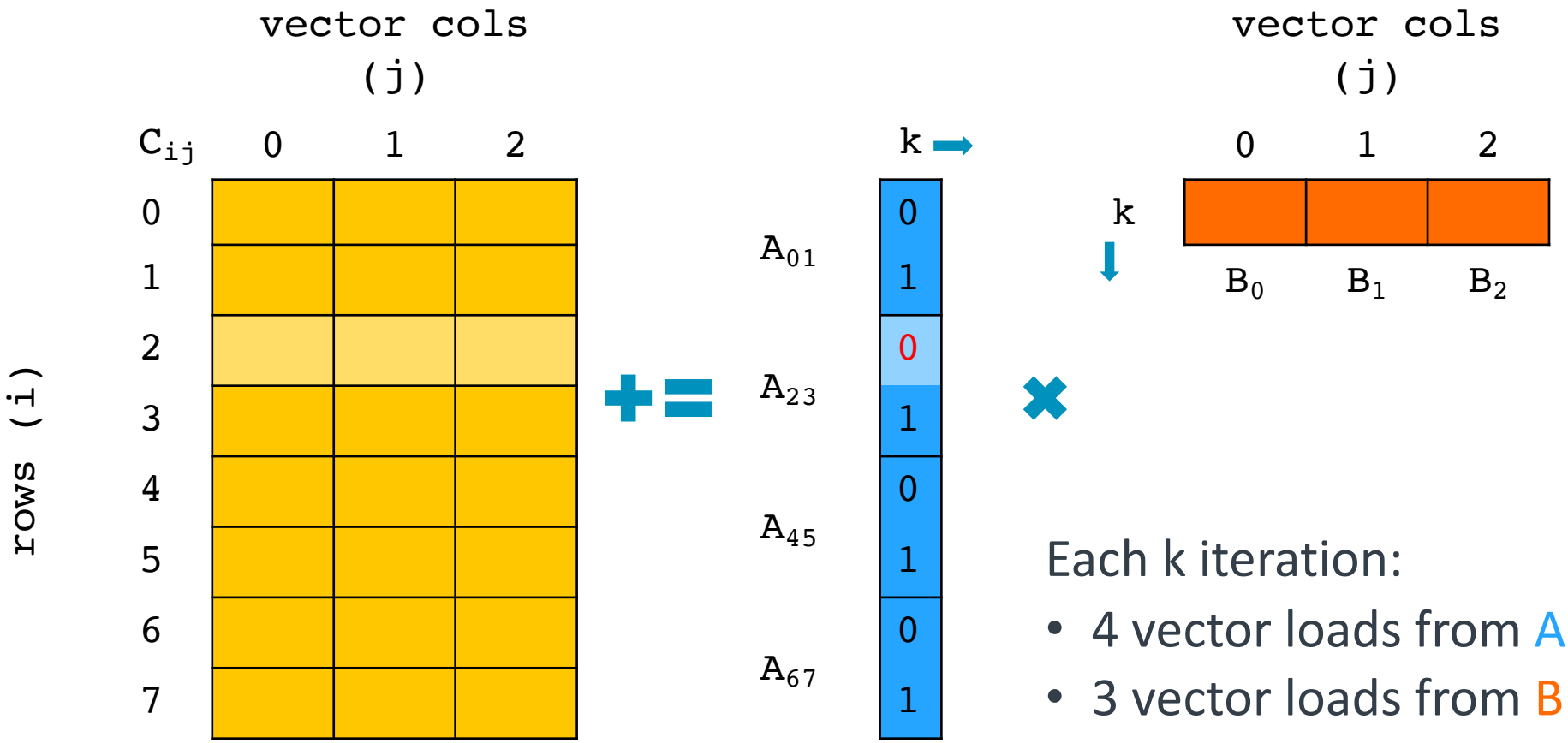
GEMM

$$c_{ij} \mathrel{+}= \sum_{k=0}^{K-1} a_{ik} b_{kj}$$



arm

DGEMM kernel (NEON, 128-bit, 24 accumulators)



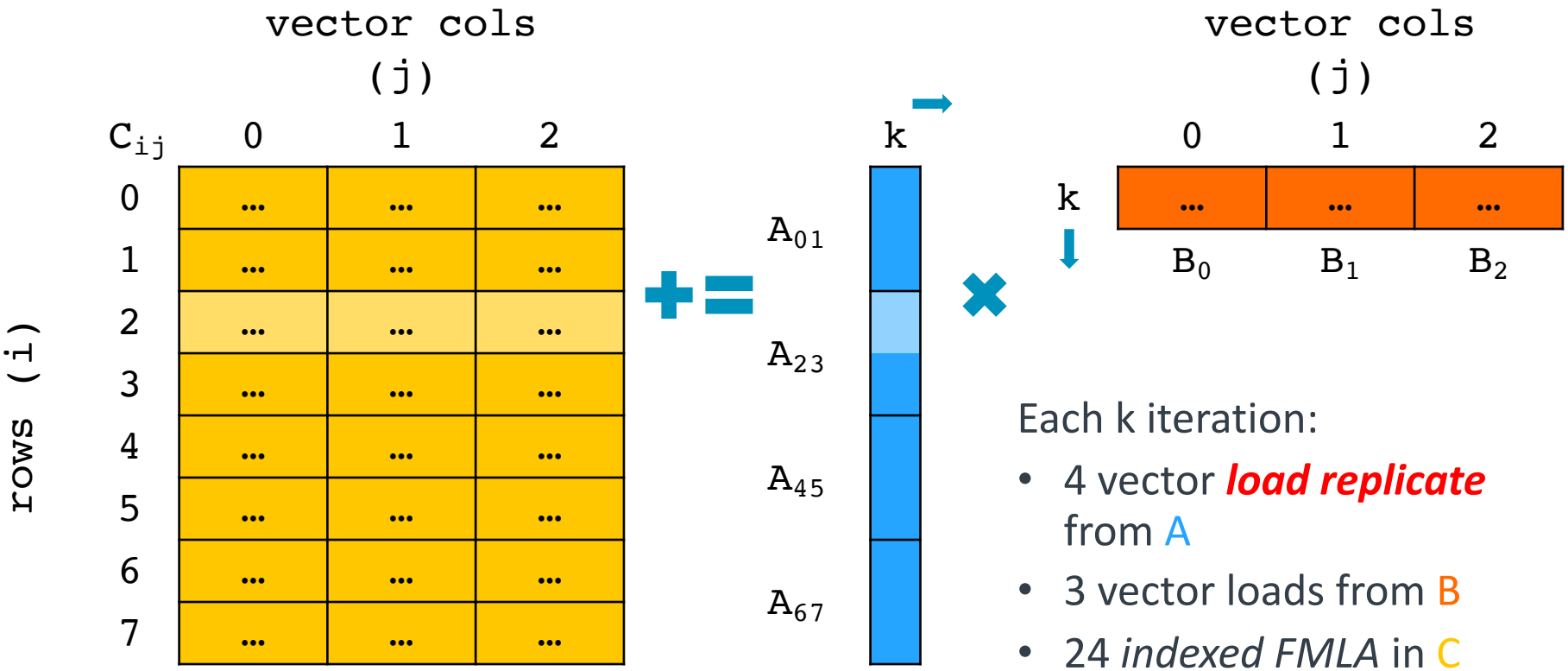
Each k iteration:

- 4 vector loads from A
- 3 vector loads from B
- 24 *indexed FMLA* in C

```
fmla C20.2d, B0.2d, A23.d[0]
fmla C21.2d, B1.2d, A23.d[0]
fmla C22.2d, B2.2d, A23.d[0]
```


arm

DGEMM kernel (SVE, ***LEN x 128-bit***, 24 accumulators)



Each k iteration:

- 4 vector ***load replicate*** from A
- 3 vector loads from B
- 24 *indexed FMLA* in C

```
fmla C20.d, B0.d, A23.d[0]
fmla C21.d, B1.d, A23.d[0]
fmla C22.d, B2.d, A23.d[0]
```


arm

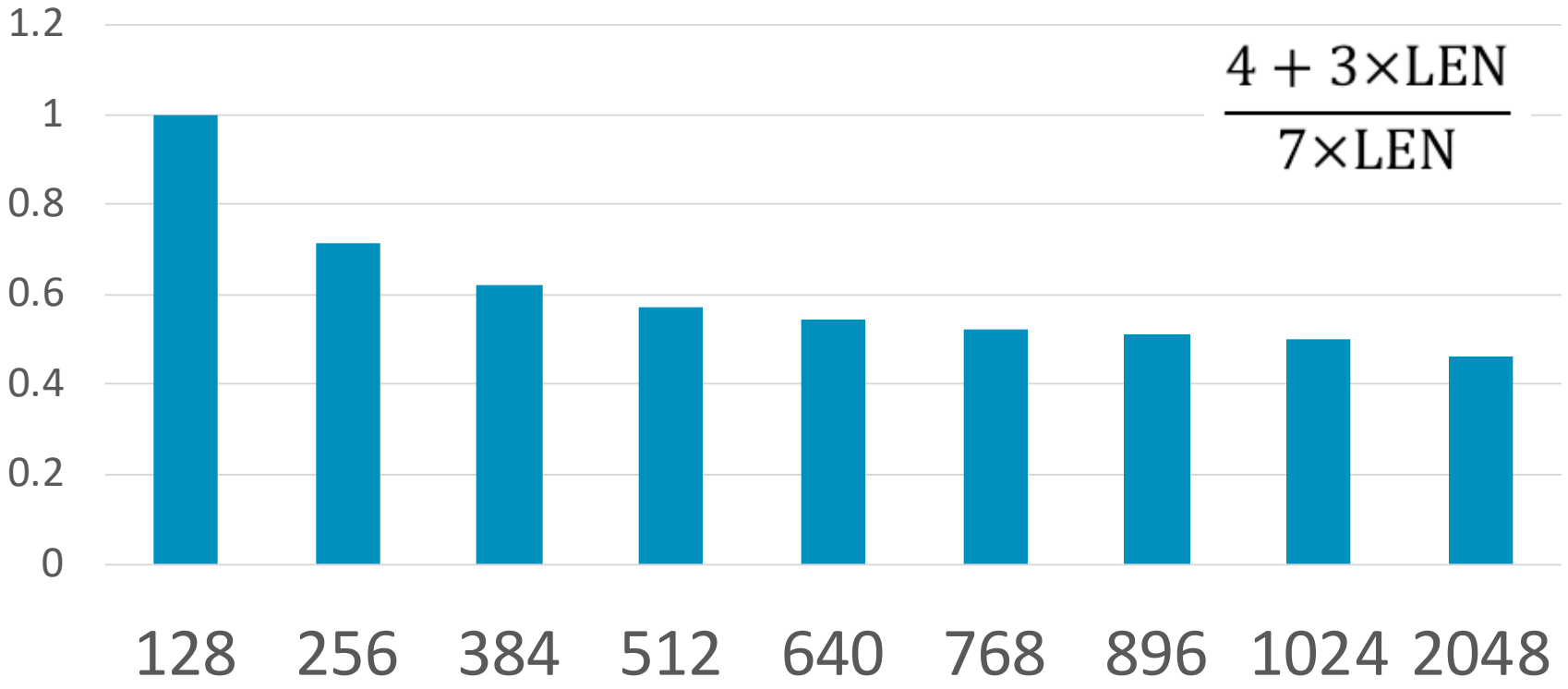
Describe load replicate

DGEMM: SVE vs NEON

	Each k iteration	Bytes in one k iteration	C area	SVE/NEON A and B data reads for same C area
SVE	• 4 vector ***load replicate*** from A • 3 vector loads from B	16B x 4 + 16B x LEN x 3	24 x 16B x LEN	$\frac{4 + 3\times\mathrm{LEN}}{7\times\mathrm{LEN}}$
NEON	• 4 vector loads from A • 3 vector loads from B	16B x 7	24 x 16B	

arm

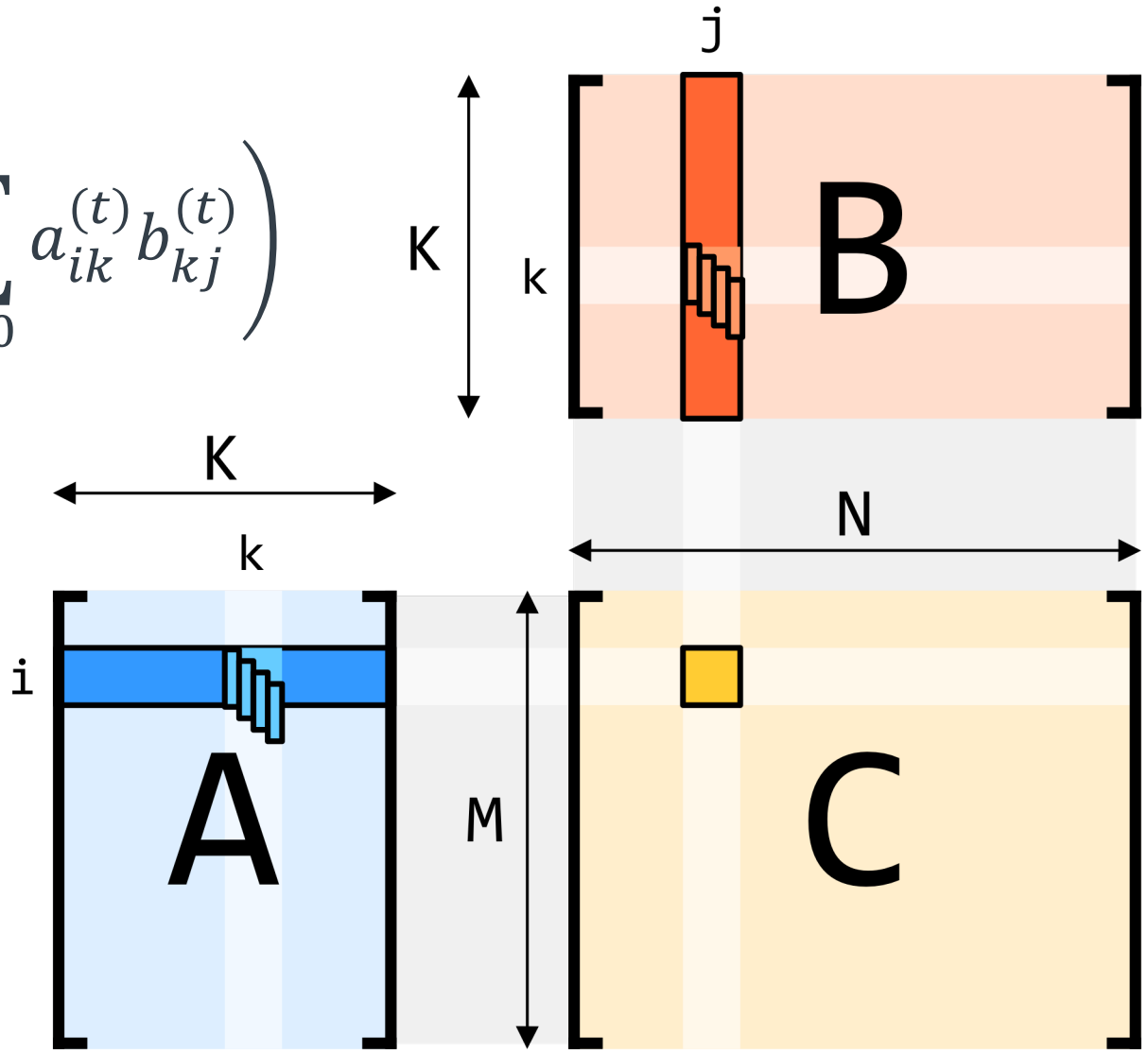
DGEMM: SVE more memory efficient than LEN times NEON



arm

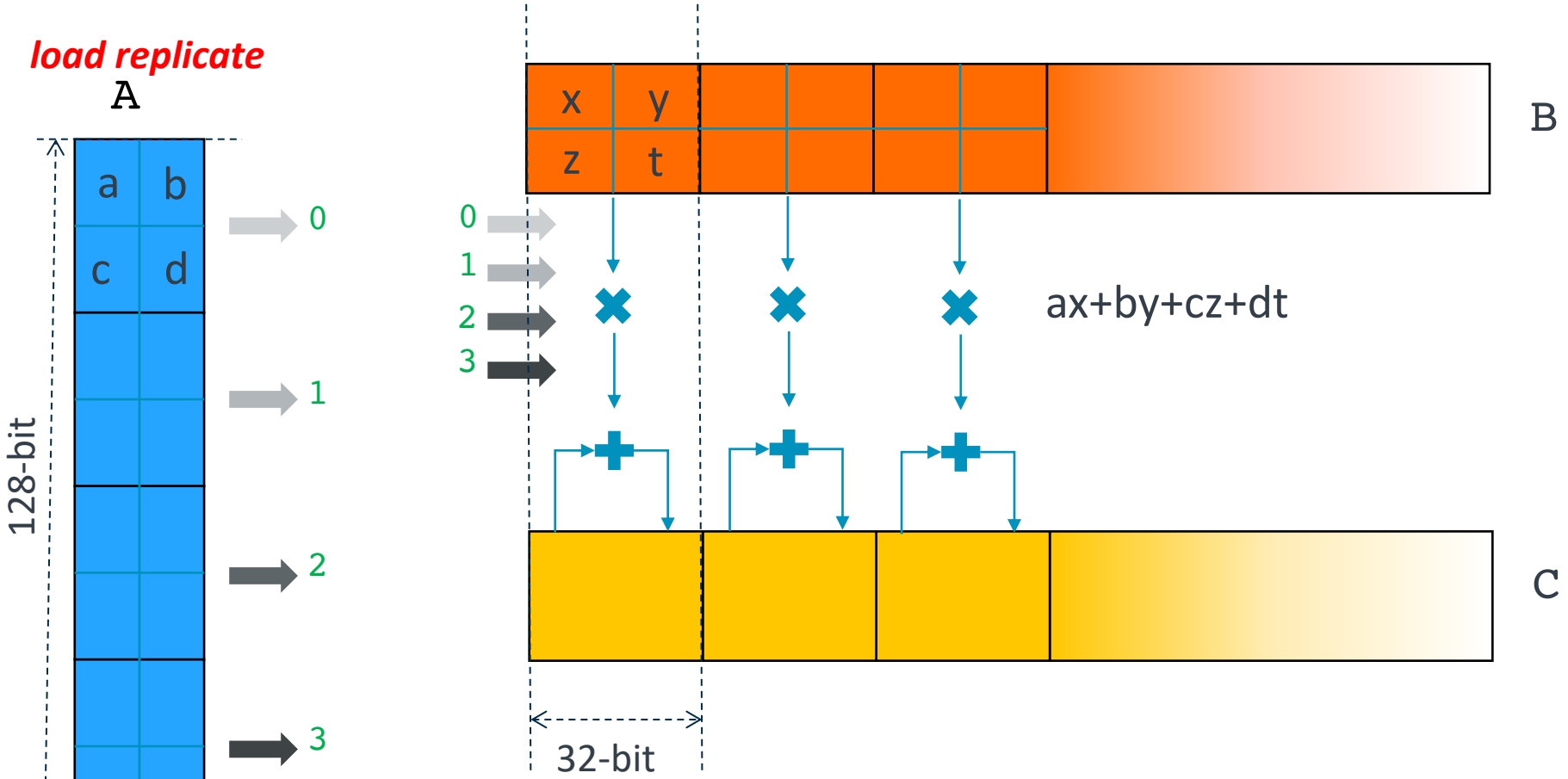
GEMMlowp

$$c_{ij} \mathrel{+}= \sum_{k=0}^{K-1} \left(\sum_{t=0}^{3} a_{ik}^{(t)} b_{kj}^{(t)} \right)$$



arm

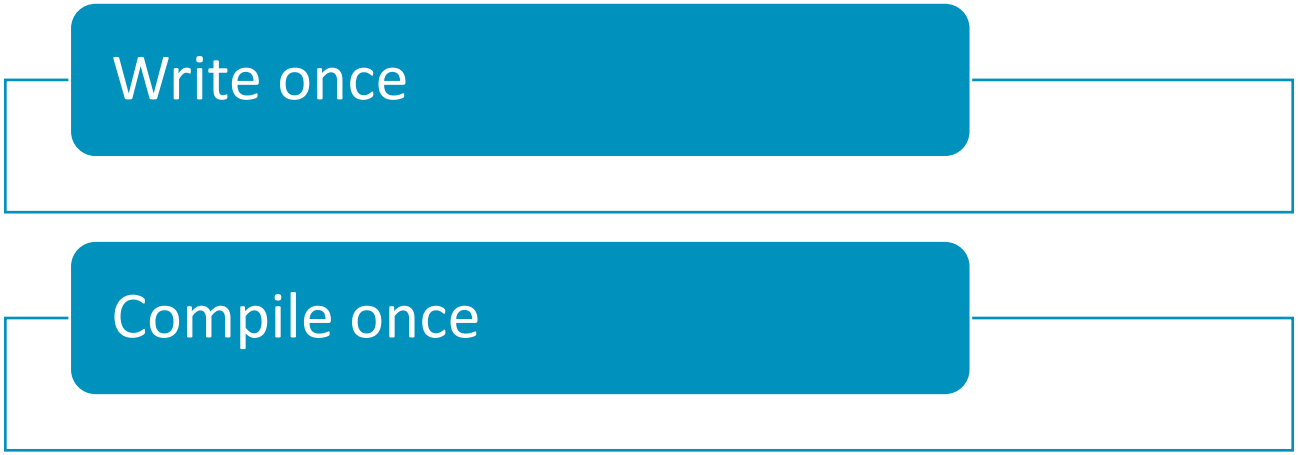
Dot product instruction




```
SDOT Zda.S, Zn.B, Zm.B[0]
SDOT Zda.S, Zn.B, Zm.B[1]
SDOT Zda.S, Zn.B, Zm.B[2]
SDOT Zda.S, Zn.B, Zm.B[3]
```


arm

Vector Length Agnostic programming model



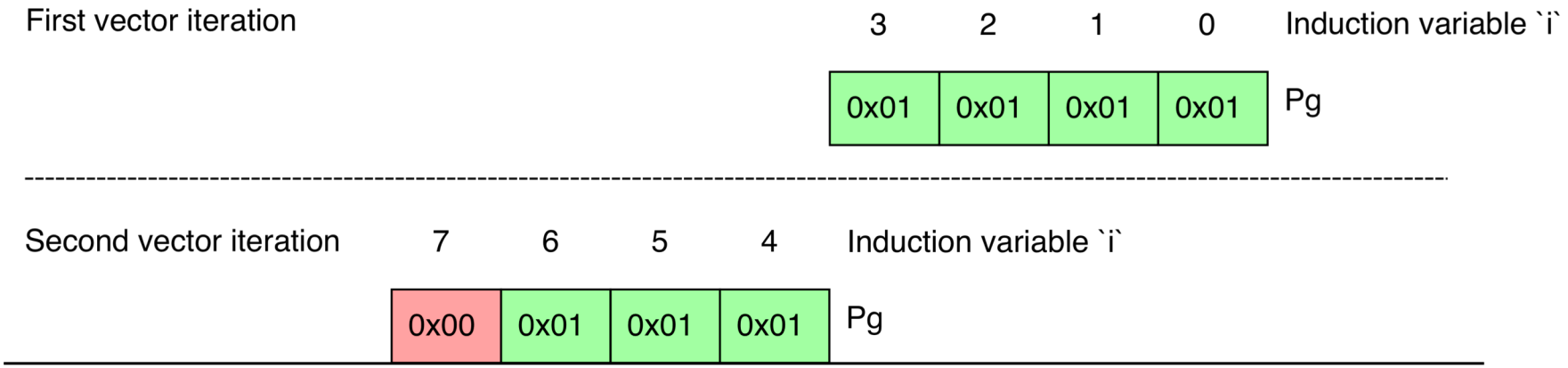
```
float *a = /* ... */;
int N = /* ... */;
for (int i = 0; i < N; ++i)
  a[i] = 2.0 * a[i];
```

```
for (int i = 0; i < N; ++i)
  a[i] = 2.0 * a[i];
```

Restore NEON vectorization

Predication – get rid of the loop tail

```
for (int i = 0 ; i < N; i += 4) {
  svbool_t Pg = svwhilelt_b32(i, N);
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```



arm

Go wider than NEON – how much?

```
for (int i = 0 ; i < N; i += 8) {
  svbool_t Pg = svwhilelt_b32(i, N);
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

Go wider than NEON – how much?

```
for (int i = 0 ; i < N; i += 12) {
  svbool_t Pg = svwhilelt_b32(i, N);
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

Go wider than NEON – how much?

```
for (int i = 0 ; i < N; i += 32) {
  svbool_t Pg = svwhilelt_b32(i, N);
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

How about ANY width?

```
for (int i = 0 ; i < N; i += svcntw()) {
  svbool_t Pg = svwhilelt_b32(i, N);
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

SVE Arm C Language Extension (VLA)

```
for (int i = 0 ; i < N; i += svcntw()) {
  svbool_t Pg = svwhilelt_b32(i, N);
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

C11: _Generic

SVE : svmul_x
NEON: vmulq_n_f32

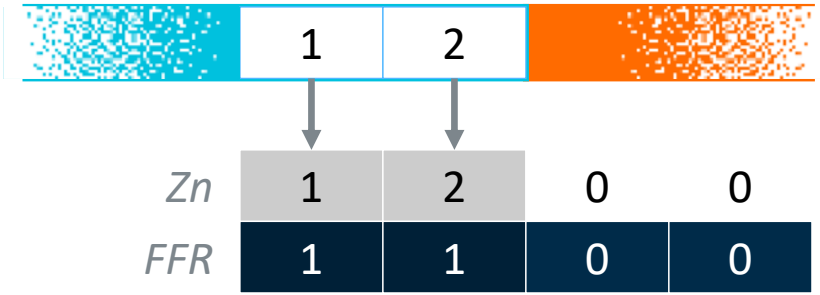
arm

Is it safe to load vectors of unknown size?

```
// Unoptimized A64 scalar strlen
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
```

```
// Unoptimized SVE VLA strlen
strlen:
    mov     x1, x0              // e=s
    ptrue   p0.b                    // p0=true
.loop:
    setffr                      // ffr=true
    ldff1b z0.b, p0/z, [x1]  // p0:z0=ldff(e)
    rdffr   p1.b, p0/z          // p0:p1=ffr
    cmpeq   p2.b, p1/z, z0.b, #0 // p1:p2=(*e==0)
    brkbs   p2.b, p1/z, p2.b        // p1:p2=until(*e==0)
    incp        x1, p2.b            // e+=popcnt(p2)
    b.last .loop                // last active=>!break
    sub     x0, x1, x0          // return e-s
    ret
```

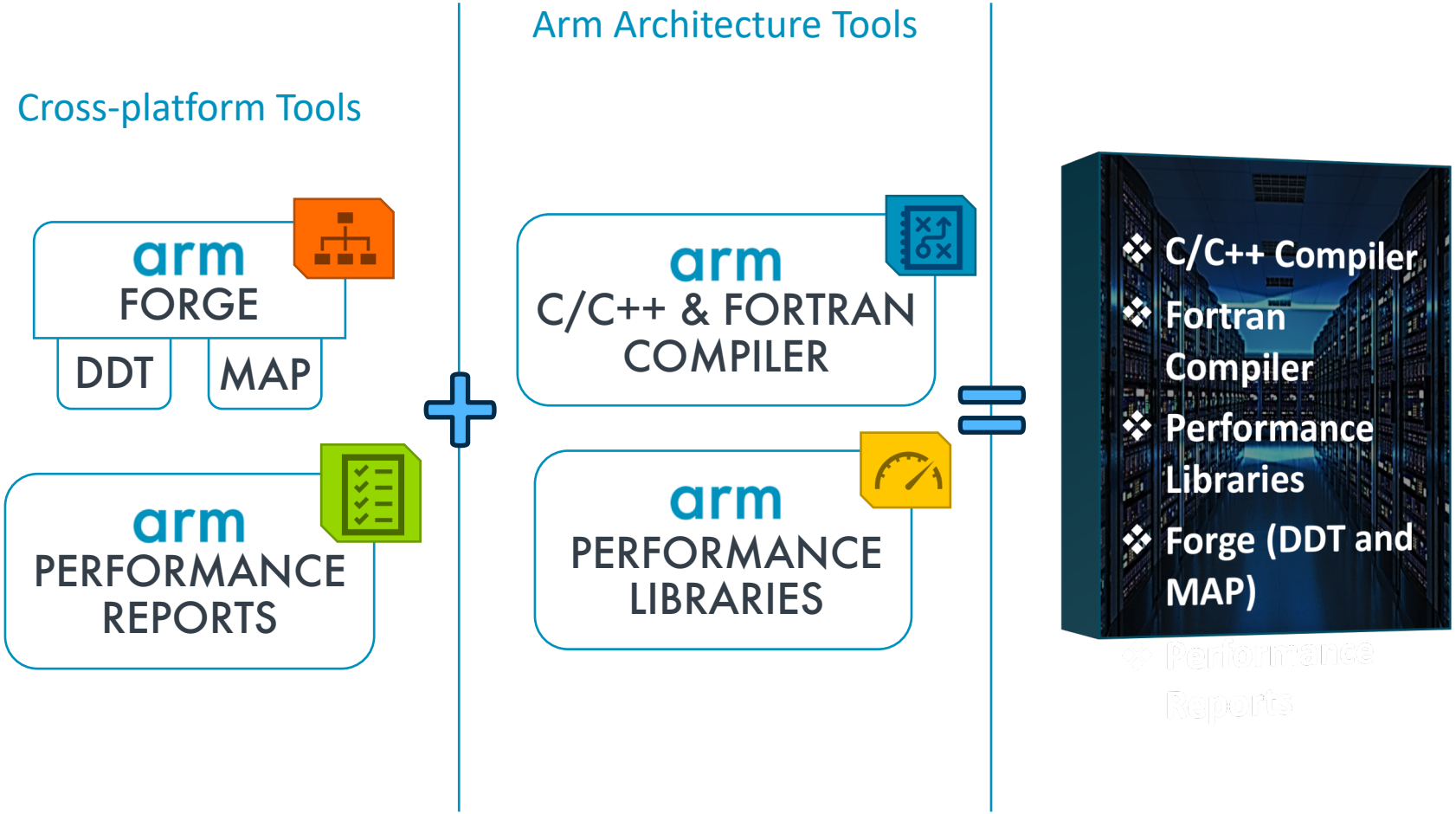
```
// ---------------------------------
//      int strlen(const char *s) {
//          const char *e = s;
//          while (*e) e++;
//          return e - s;
//      }
// ---------------------------------
// x0 = s
```



arm

Arm's solution for HPC, SVE, VLA

Combines cross-platform tools with Arm only tools for a comprehensive solution



arm

arm

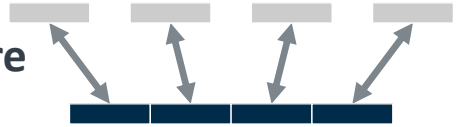
Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
تشكر

Why is it called Scalable Vector Extension?

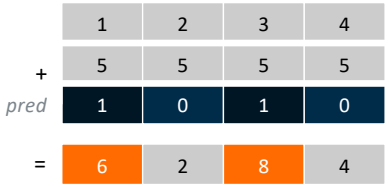
- SVE does not mandate a single, fixed vector length
  - Vector Length (VL) is hardware implementation choice of 128 to 2048 bits.
  - Vector Length Agnostic (VLA) programming paradigm adapts to the available vector length.
- SVE begins to address traditional barriers to auto-vectorization
  - Compilers often cannot vectorize due to intra-vector control and data dependencies.
  - Software-managed speculative vectorization allows more loops to be vectorized by a compiler.

New features in SVE

- **Gather-load and scatter-store**

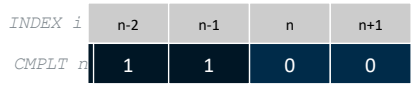


- **Per-lane predication**

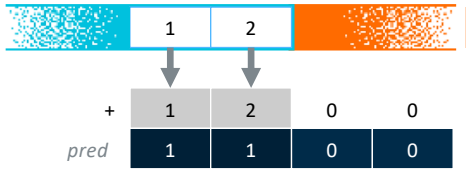


- **Predicate-driven loop control and management**

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



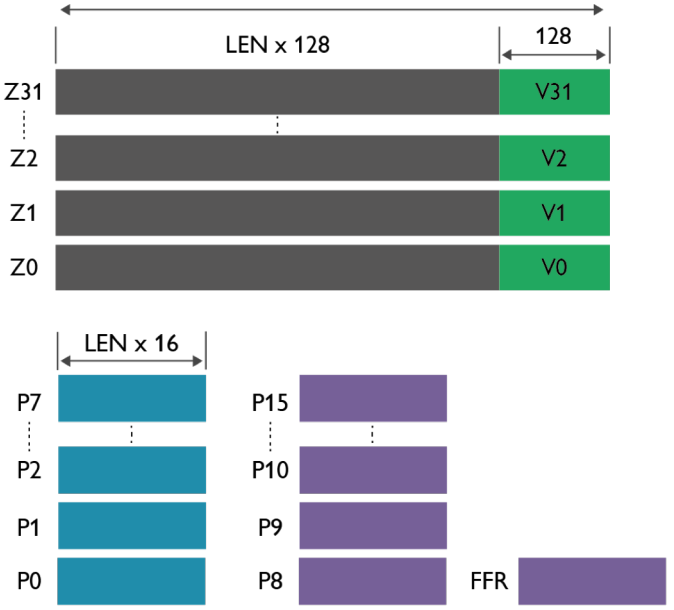
- **Vector partitioning and software-managed speculation**



arm

New SVE registers

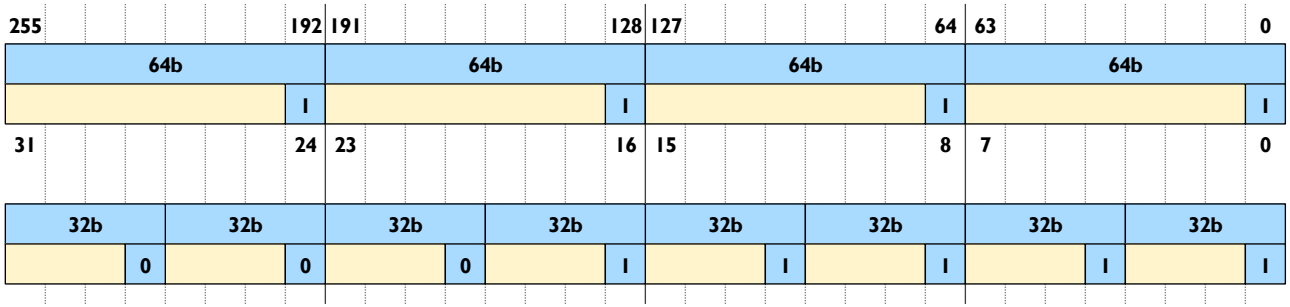
- Scalable vector registers
  - Z0-Z31 extending NEON's V0-V31.
  - Packed DP, SP & HP floating-point elements.
  - Packed 64, 32, 16 & 8-bit integer elements.
- Scalable predicate registers
  - P0-P7 governing predicates for load/store/arithmetic.
  - P8-P15 additional predicates for loop management.
  - FFR first fault register for speculation.



arm

Predication

- 16 predicate registers defined (P0-P15)
  - 1 bit per lane ∴ 1 predicate bit per 8 vector bits (lowest predicate bit per lane is significant)
- Active elements update destination
  - inactive lanes leave destination unchanged or set to 0's
  - power-saving opportunities
- ≈6% SVE instructions act on and return predicates → "Predicate ALU"



arm