SC19

A Closer Look at Arm SVE

SAXPY

```
subroutine saxpy(x,y,a,n)
real*4 x(n),y(n),a
do i = 1,n
    y(i) = a*x(i) + y(i)
enddo
```

Scalar [–march=armv8–a]

```
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy_:
    ldrsw    x3, [x3]                // x3=*n
    mov      x4, #0                  // x4=i=0
    ldr      s0, [x2]                // d0=*a
    b        .latch
.loop:
    ldr      s1, [x0,x4,lsl 2]       // s1=x[i]
    ldr      s2, [x1,x4,lsl 2]       // s2=y[i]
    fmadd    s2, s1, s0, s2          // s2+=x[i]*a
    str      s2, [x1,x4,lsl 2]       // y[i]=s2
    add      x4, x4, #1              // i+=1
.latch:
    cmp      x4, x3                  // i < n
    b.lt     .loop                   // more to do?
    ret
```

Key Operations

- `whilelt` constructs a **predicate** (p0) to dynamically map vector operations to vector data
- `incw` increments a scalar register (x4) by the number of float elements that fit in a vector register
- No drain loop! Predication handles the remainder

SVE [–march=armv8–a+sve]

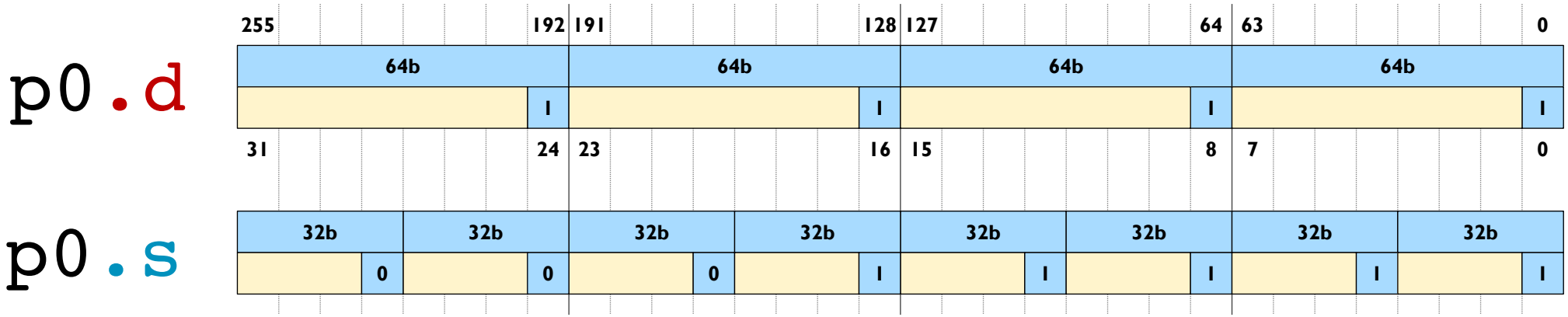
```
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy_:
    ldrsw    x3, [x3]                // x3=*n
    mov      x4, #0                  // x4=i=0
    whilelt  p0.s, x4, x3            // p0=while(i++<n)
    ld1rw    z0.s, p0/z, [x2]        // p0:z0=bcast(*a)
.loop:
    ld1w     z1.s, p0/z, [x0,x4,lsl 2]  // p0:z1=x[i]
    ld1w     z2.s, p0/z, [x1,x4,lsl 2]  // p0:z2=y[i]
    fmla     z2.s, p0/m, z1.s, z0.s     // p0?z2+=x[i]*a
    st1w     z2.s, p0, [x1,x4,lsl 2]    // p0?y[i]=z2
    incw     x4                      // i+=(VL/32)
.latch:
    whilelt  p0.s, x4, x3            // p0=while(i++<n)
    b.first  .loop                   // more to do?
    ret
```


arm

Predicates: Active Lanes vs Inactive Lanes

Predicate registers track lane activity

- 16 predicate registers (P0-P15)
- 1 predicate bit per 8 vector bits (lowest predicate bit per lane is significant)
- On **load**, active elements update the destination
- On **store**, inactive lanes leave destination unchanged (p0/m) or set to 0's (p0/z)



arm

Predicate Condition Flags

SVE is a *predicate-centric* architecture

- Predicates support complex nested conditions and loops.
- Predicate generation also sets condition flags.
- Reduces vector loop management overhead.

Overloading the A64 NZCV condition flags

Condition Test	A64 Name	SVE Alias	SVE Interpretation
Z=1	EQ	NONE	No active elements are true
Z=0	NE	ANY	Any active element is true
C=1	CS	NLAST	Last active element is not true
C=0	CC	LAST	Last active element is true
N=1	MI	FIRST	First active element is true
N=0	PL	NFRST	First active element is not true
C=1 & Z=0	HI	PMORE	More partitions: some active elements are true but not the last one
C=0 | Z=1	LS	PLAST	Last partition: last active element is true or none are true
N=V	GE	TCONT	Continue scalar loop
N!=V	LT	TSTOP	Stop scalar loop

How do you count by vector width?

No need for multi-versioning: one increment to rule all vector sizes

```
ld1w z1.s, p0/z, [x0,x4,lsl 2]      // p0:z1=x[i]
ld1w z2.s, p0/z, [x1,x4,lsl 2]      // p0:z2=y[i]
fmla z2.s, p0/m, z1.s, z0.s         // p0?z2+=x[i]*a
st1w z2.s, p0, [x1,x4,lsl 2]        // p0?y[i]=z2
incw x4                             // i+=(VL/32)
```

"Increment `x4` by the number of 32-bit lanes (`w`) that fit in a VL."

arm

Vector Length Agnostic (VLA) instructions

Increments and counting instructions.

```
incb x0, mul3, mul #2     x0 += 2 x (largest mul3 <= VL.b)
incd z0.d, pow2           each lane += largest pow2 <= VL.d
incp x0, p0.s             x0 += # active lanes
incp z0.h,  p0            each vector lane += # active lanes of p

cntw x0                   x0 = VL.s
cntp x0, p0, p1.s.        x0 = # active lanes of (p0 && p1.s)
```

Initialization when vector length is unknown

- Vectors cannot be initialized from compile-time constant, so…
  - `INDEX Zd.S,#1,#4` : `Zd = [ 1, 5, 9, 13, 17, 21, 25, 29 ]`
- Predicates cannot be initialized from memory, so…
  - `PTRUE Pd.S, MUL3` : `Pd = [ T, T, T , T, T, T , F, F ]`
- Vector loop increment and trip count are unknown at compile-time, so…
  - `INCD Xi` : increment scalar Xi by # of 64b dwords in vector
  - `WHILELT Pd.D,Xi,Xe` : next iteration predicate `Pd = [ while i++ < e ]`
- Vectors stores to stack must be dynamically allocated and indexed, so…
  - `ADDVL SP,SP,#-4` : decrement stack pointer by (4*VL)
  - `STR Zi, [SP,#3,MUL VL]` : store vector Z1 to address (SP+3*VL)

arm

Portability

Is it really possible to run a vectorized application anywhere?

Write once: can my code ***compile*** for machines with different VL?


Code that is auto-vectorized by the compiler


Hand written assembly


Hand written C intrinsics

Compile once: Can I take my executable and ***run it*** on machines with different VL?


Self contained programs with no external dependencies


But what about programs that depend on external libraries? ... (spoiler: )

arm

Auto-vectorize external calls: `libm` example.

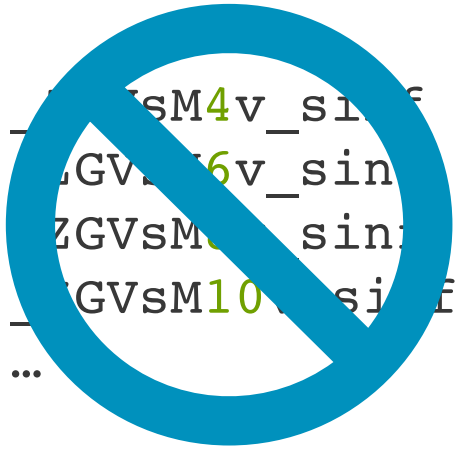
```
float sinf(float);
```

NEON

- Neon has 128-bit and 64-bit register split.
- The library has to provide at least 2 symbols, because it doesn't know where the auto-vec code comes from:
  - `_ZGVnN2v_sinf`
  - `_ZGVnN4v_sinf`

SVE

- Does `libm` need to provide a symbol for each VL?
  - …



- One symbol!

`_ZGVsMxv_sinf`

arm

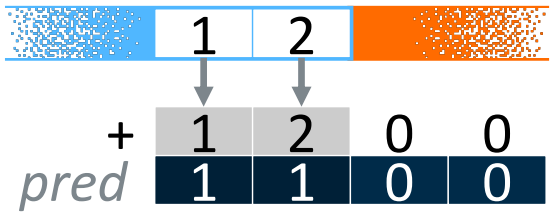
arm

SVE Features for Improving Vectorization

- Vector Partitioning
- Non-Contiguous Memory Access
- Horizontal Operations

Vector Partitioning: when a vector spans a protected region

With VLA, we don’t always know what data we may touch



- **Software-managed speculative vectorisation**
  - Create sub-vectors (partitions) in response to data and dynamic faults
- **First-fault load allows access to safely cross a page boundary**
  - First element is mandatory but others are a “speculative prefetch”
  - Dedicated FFR predicate register indicates successfully loaded elements
- **Allows uncounted loops with break conditions**
  - Load data using first-fault load
  - Create a *before-fault* partition from FFR
  - Test for break condition
  - Create a *before-break* partition from condition predicate
  - Process data within partition
  - Exit loop if break condition was found.

arm

Vectorizing `strlen`

A vector load could result in a segfault if the vector spans protected memory.

Source Code

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

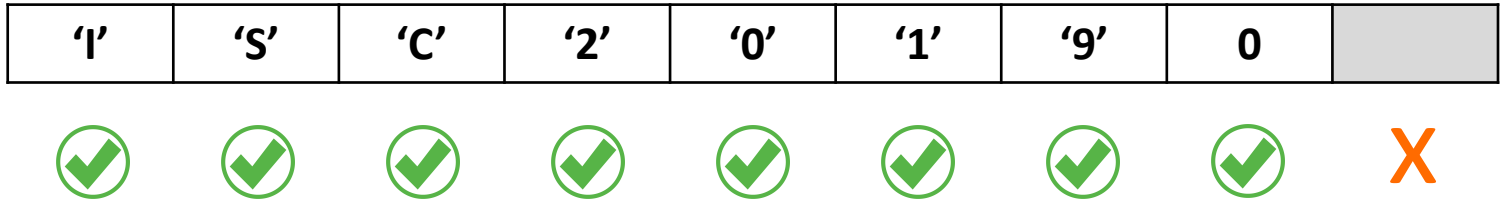
Scalar [-march=armv8-a]

```
// 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
```


arm

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

arm

Vectorizing `strlen`

Partitioning off protected memory

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

SVE [-march=armv8-a+sve]

```
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
```

Scalar [-march=armv8-a]

```
// x0 = s
strlen:
    mov     x1, x0
.loop:
    ldrb    x2, [x1],#1
    cbnz    x2, .loop
.done:
    sub     x0, x1, x0
    sub     x0, x0, #1
    ret
```


arm

Optimized 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
```

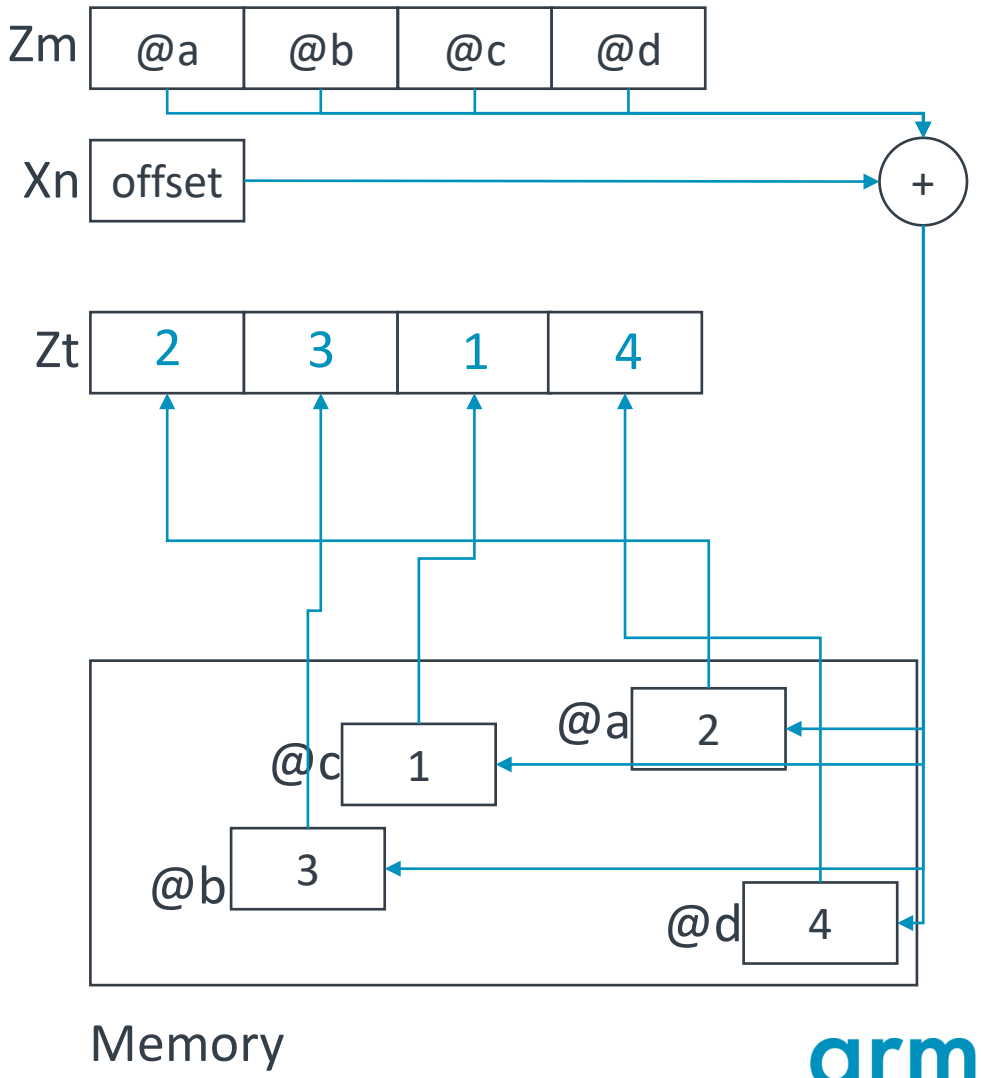
Suboptimal implementation

```
    setffr              /* initialize FFR */
    ptrue p2.b          /* all ones; loop invariant */
    mov x1, 0           /* initialize length */

    /* Read a vector's worth of bytes, stopping on first fault. */
0:  ldff1b z0.b, p2/z, [x0, x1]
    rdffrs p0.b, p2/z
    b.nlast 2f

    /* First fault did not fail: the whole vector is valid. Avoid
    depending on the contents of FFR beyond the branch. */
    incb x1, all                    /* speculate increment */
    cmpeq p1.b, p2/z, z0.b, 0       /* loop if no zeros */
    b.none 0b
    decb x1, all                    /* undo speculate */

    /* Zero found. Select the bytes before the first and count them. */
1:  brkb p0.b, p2/z, p1.b
    incp x1, p0.b
    mov x0, x1
    ret

    /* First fault failed: only some of the vector is valid. Perform the
    comparison only on the valid bytes. */
2:  cmpeq p1.b, p0/z, z0.b, 0
    b.any 1b

    /* No zero found. Re-init FFR, increment, and loop. */
    setffr
    incp x1, p0.b
    b 0b
```


arm

Gather/Scatter Operations are Good and Evil

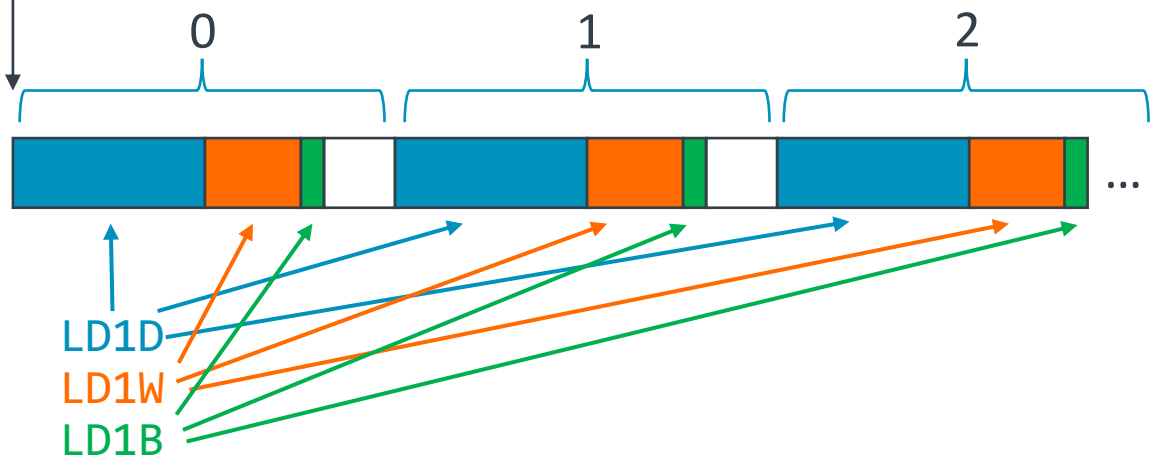
- Enable vectorization of codes with non-adjacent accesses on adjacent lanes
- Examples:
  - Outer loop vectorization
  - Strided accesses (larger than +1)
  - Random accesses
- Performance implementation dependent
  - Worst case one separate access per element
- `LD1D <Zt>.D, Ps/Z [<Xn>, <Zm>.D]`



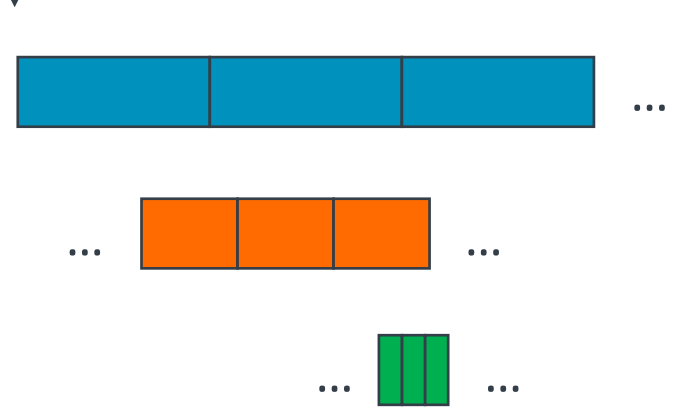
arm

Array of Structures vs. Structure of Arrays

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

Programmer_t programmers[N];
```



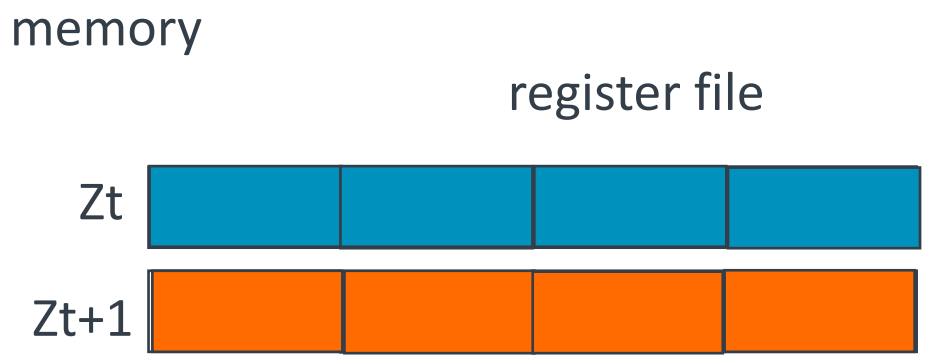

```
typedef struct {
    uin64_t num_projects[N];
    float caffeine[N];
    bool vim_nemacs[N];
} Programmer_t;

Programmer_t programmers;
```



arm

Contiguous Multi-Register Structure Loads/Stores

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

Researcher_t researchers[N];
```



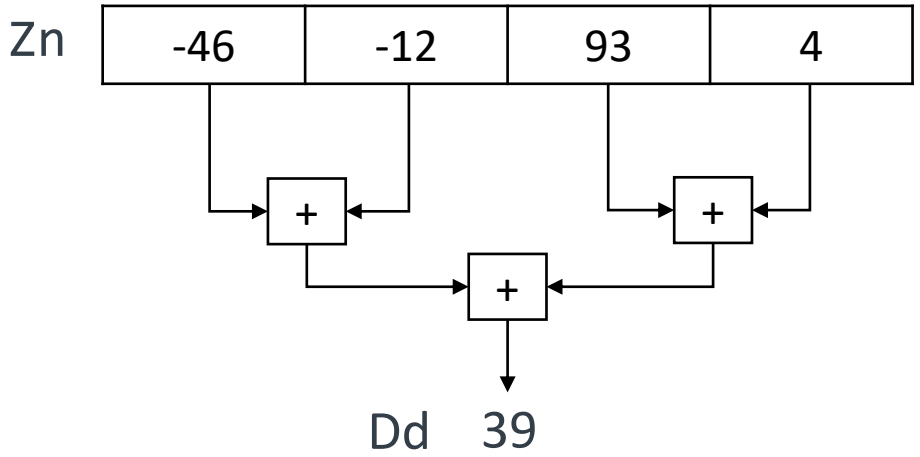
arm

A rich set of horizontal operations

The operation happens across the lanes of a vector

- Addition, maximum, minimum, bitwise AND, OR, XOR …

```
SADDV <Dd>, <Pg>, <Zn.T>
```



arm

arm

Arm C Language Extensions for SVE

Compiler Intrinsics for SVE

Arm C Language Extensions

Intrinsics and other features for supporting Arm features in C and C++

- ACLE extends C/C++ with Arm-specific features
  - Predefined macros: __ARM_ARCH_ISA_A64, __ARM_BIG_ENDIAN, etc.
  - Intrinsic functions: __clz(uint32_t x), __cls(uint32_t x), etc.
  - Data types: SVE, NEON and FP16 data types
- **ACLE for SVE enables VLA programming with ACLE**
  - Nearly one intrinsic per SVE instruction
  - Data types to represent the size-less vectors used for SVE intrinsics
- Intended for users that…
  - Want to hand-tune SVE code
  - Want to adapt or hand-optimize applications and libraries
  - Need low-level access to Arm targets

How to use ACLE

- Include the headers you need
  - `arm_acle.h` → for core ACLE
  - `arm_fp16.h` → to add scalar FP16 arithmetic
  - `arm_neon.h` → to add NEON intrinsics and data types
  - `arm_sve.h` → to add SVE intrinsics and data types

- Each of those require certain features at the compilation target
  - `arm_fp16.h` → Your target platform needs to support FP16 (-march=armv8-a+fp16)
  - `arm_neon.h` → Your target platform need to support NEON (-march=armv8-a+simd)
  - `arm_sve.h` → Your target platform need to support SVE (-march=armv8-a+sve)

SVE ACLE

SVE Arm C Language Extensions – aka *C intrinsics*

```
#include <arm_sve.h>
```

- VLA Data types:
  - `svfloat64_t`, `svfloat16_t`, `svuint32_t`,…
- Predication:
  - Merging: `_m`
  - Zeroing: `_z`
  - Don't care: `_x`
  - Predicate type: `svbool_t`
- Use *C11 generics* for function overloading.
- Intrinsics are **not 1-1 with the ISA**.

Examples

```
svfloat32_t
svadd[_n_f32]_z(svbool_t pg,
                svfloat32_t op1,
                float32_t op2);


svfloat16_t
svsqrt_m(svfloat16_t inactive,
         svbool_t pg,
         svfloat16_t op)
```


arm

Vectorizing a scalar loop with ACLE

`a[:] = 2.0 * a[:]`

Original Code

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

128-bit NEON vectorization with ACLE

```
int i;

// vector loop
for (i=0; (i<N-3) && (N&~3); i+=4) {
  float32x4_t va = vld1q_f32(&a[i]);
  va = vmulq_n_f32(va, 2.0);
  vst1q_f32(&a[i], va)
}
// drain loop
for (; i < N; ++i)
  a[i] = 2.0 * a[i];
```

This is NEON, ***not*** SVE!

arm

Vectorizing a scalar loop with ACLE

a[:] = 2.0 * a[:]

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

SVE vectorization

```
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);
}
```

128-bit NEON vectorization

```
int i;

// vector loop
for (i=0; (i<N-3) && (N&~3); i+=4) {
  float32x4_t va = vld1q_f32(&a[i]);
  va = vmulq_n_f32(va, 2.0);
  vst1q_f32(&a[i], va)
}
// drain loop
for (; i < N; ++i)
  a[i] = 2.0 * a[i];
```


arm

Vectorizing a scalar loop with ACLE

a[:] = 2.0 * a[:]

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

SVE vectorization

```
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 vectorization with fewer branches

```
svbool_t all = svptrue_b32();
svbool_t Pg;
for (int i=0;
     svptest_first(all,
         Pg=svwhilelt_b32(i, N));
     i += svcntw())
{
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```


arm

ACLE for SVE Cheat Sheet

- Vector types
  - `sv<datatype><datasize>_t`
    - `svfloat32_t`
    - `svint8_t`
    - `svuint16_t`
- Predicate types
  - `svbool_t`

- Functions
  - `sv`*`base`*`[`*`disambiguator`*`][`*`type0`*`][`*`type1`*`]…[`*`predication`*`]`
  - base is the lower-case name of an SVE instruction
  - *`disambiguator`* distinguishes between different forms of a function
  - *`typeN`* lists the types of vectors and predicates
  - *`predication`* describes the inactive elements in the result of a predicated operation

```
svfloat64_t svld1_f64(svbool_t pg, const float64_t *base)
svbool_t svwhilelt_b8(int64_t op1, int64_t op2)
svuint32_t svmla_u32_z(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint32_t svmla_u32_m(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3)
```

C code without intrinsics

Just compile with `-march=armv8-a+sve`

```
double ddot (double *a, double *b, int n)
{
    double sum = 0.0;
    for ( int i = 0; i < n; i++ ) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

- Accumulates in a scalar
- No horizontal reductions

```
cmp     w2, #1
b.lt    #52 <ddot+0x38>
mov     w9, w2
mov     x8, xzr
whilelo p0.d, xzr, x9
fmov    d0, xzr
ld1d    { z1.d }, p0/z, [x0, x8, lsl #3]
ld1d    { z2.d }, p0/z, [x1, x8, lsl #3]
incd    x8
fmul    z1.d, z1.d, z2.d
fadda   d0, p0, d0, z1.d
whilelo p0.d, x8, x9
b.mi    #-24 <ddot+0x18>
ret
fmov    d0, xzr
ret
```


arm

C code with SVE intrinsics

Vector accumulator; horizonal reduction

```
#include <arm_sve.h>

double ddot (double *a, double *b, int n) {
  svfloat64_t svsum = svdup_f64(0.0);
  svbool_t pg;
  svfloat64_t sva, svb, svsum;
  for (int i = 0; i < n; i += svcntd()) {
    pg = svwhilelt_b64(i, n);
    sva = svld1_f64(pg, &a[i]);
    svb = svld1_f64(pg, &b[i]);
    svsum = svmla_f64_m(pg, svsum, sva, svb);
  }
  return svaddv_f64(svptrue_b64(), svsum);
}
```

```
cmp     w2, #1
b.lt    #60 <ddot+0x40>
mov     w8, wzr
cntd    x9
mov     z0.d, #0
whilelt p0.d, w8, w2
sxtw    x10, w8
ld1d    { z1.d }, p0/z, [x0, x10, lsl #3]
ld1d    { z2.d }, p0/z, [x1, x10, lsl #3]
add     w8, w8, w9
cmp     w8, w2
fmla    z0.d, p0/m, z1.d, z2.d
b.lt    #-28 <ddot+0x14>
ptrue   p0.d
faddv   d0, p0, z0.d
ret
mov     z0.d, #0
ptrue   p0.d
faddv   d0, p0, z0.d
ret
```


arm

arm

Resources

SVE Resources

http://developer.arm.com/hpc

- **Porting and Optimizing Guides**
  - For SVE: https://developer.arm.com/docs/101726/0110
  - For Arm in general: https://developer.arm.com/docs/101725/0110

- **The SVE Specification**
  - Arm Architecture Reference Manual Supplement, SVE for ARMv8-A

- **ACLE References and Examples**
  - ACLE: https://developer.arm.com/docs/101028/latest
  - ACLE for SVE: https://developer.arm.com/docs/100987/latest
  - Worked examples: A Sneak Peek Into SVE and VLA Programming
  - Optimized machine learning: Arm SVE and Applications to Machine Learning

arm

arm

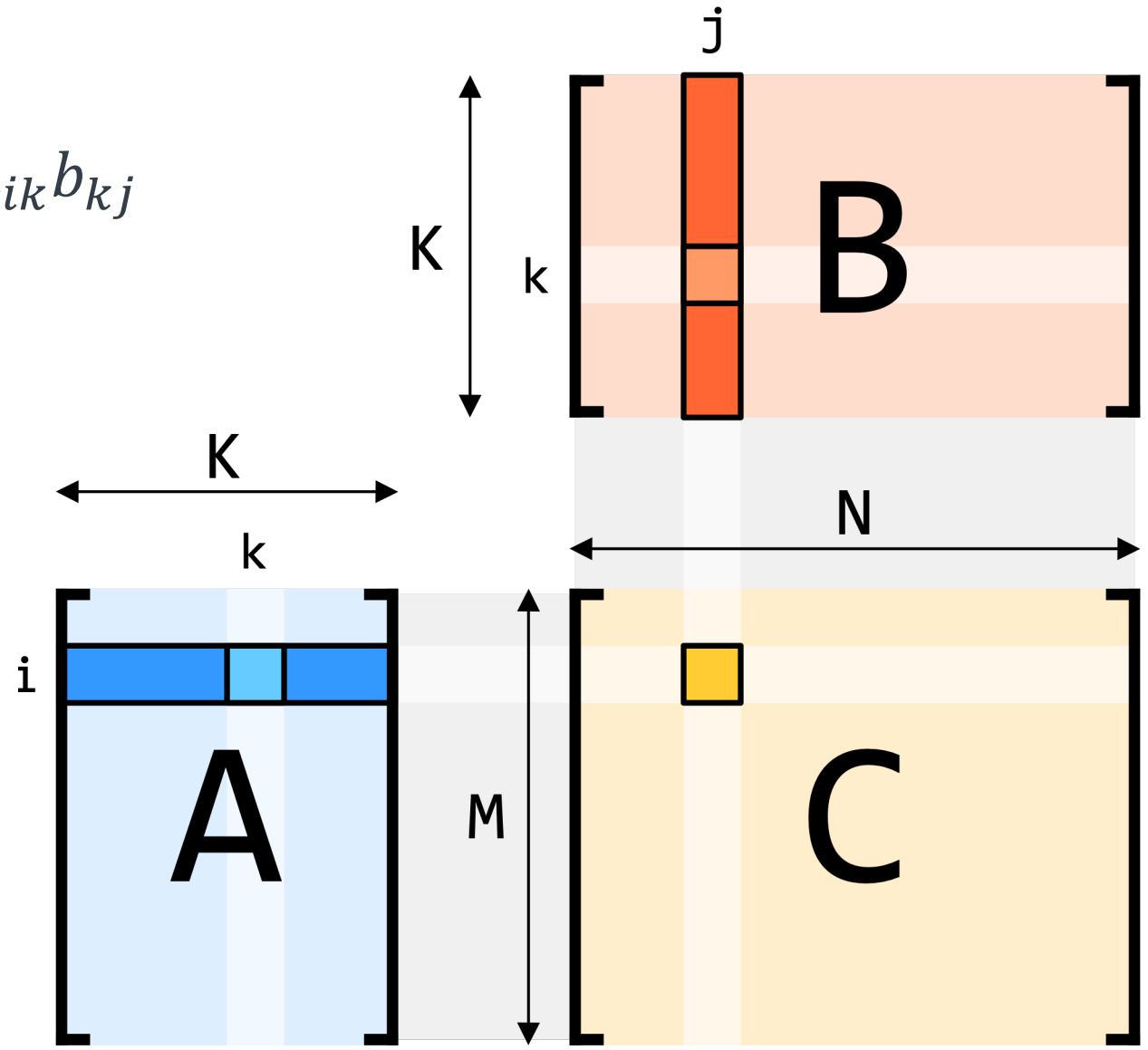
Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה

arm

SVE Load Replicate and GEMM

GEMM

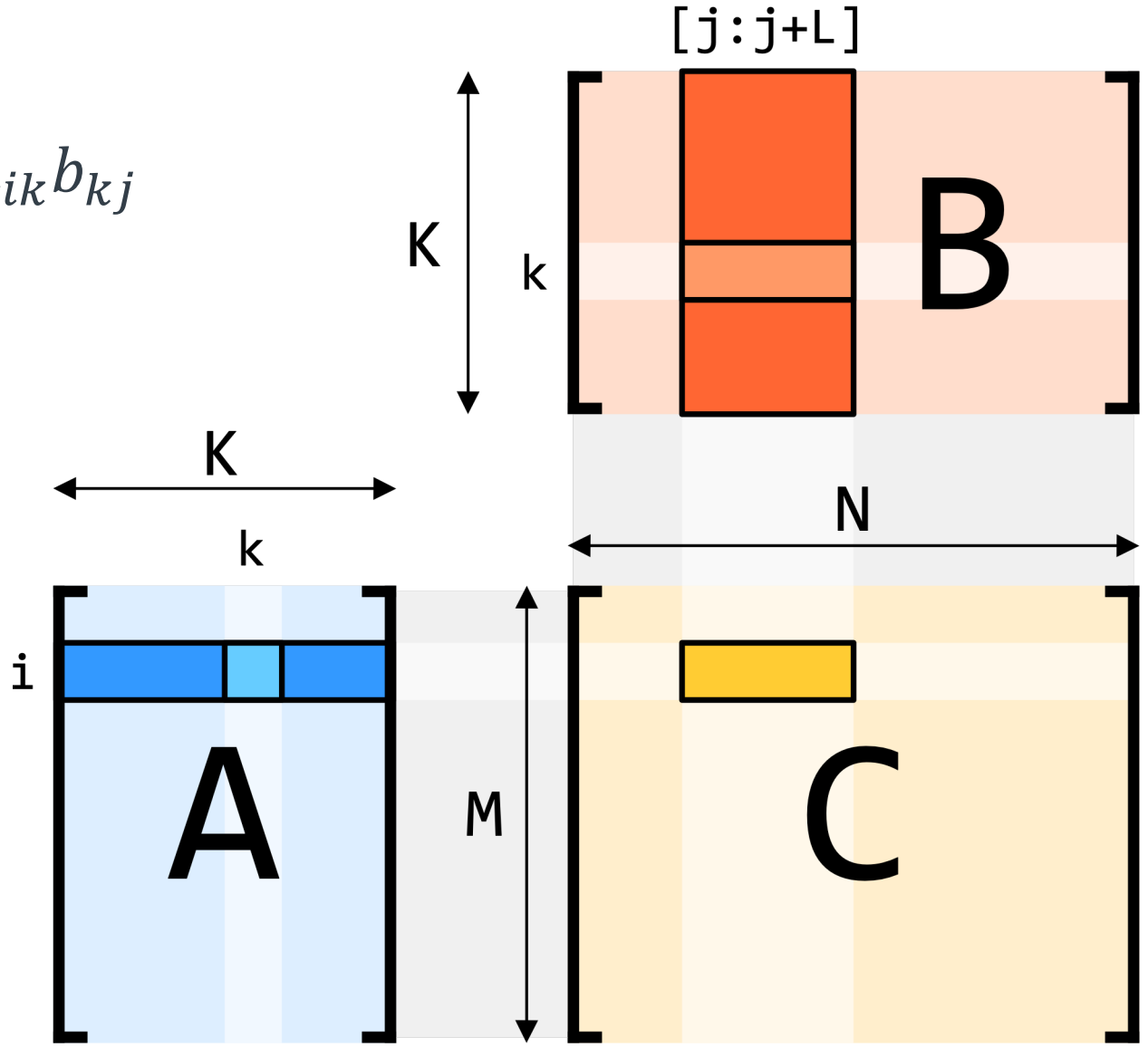
$$c_{ij} \mathrel{+}= \sum_{k=0}^{K-1} a_{ik} b_{kj}$$



arm

GEMM – step 1 – vectorize along the columns

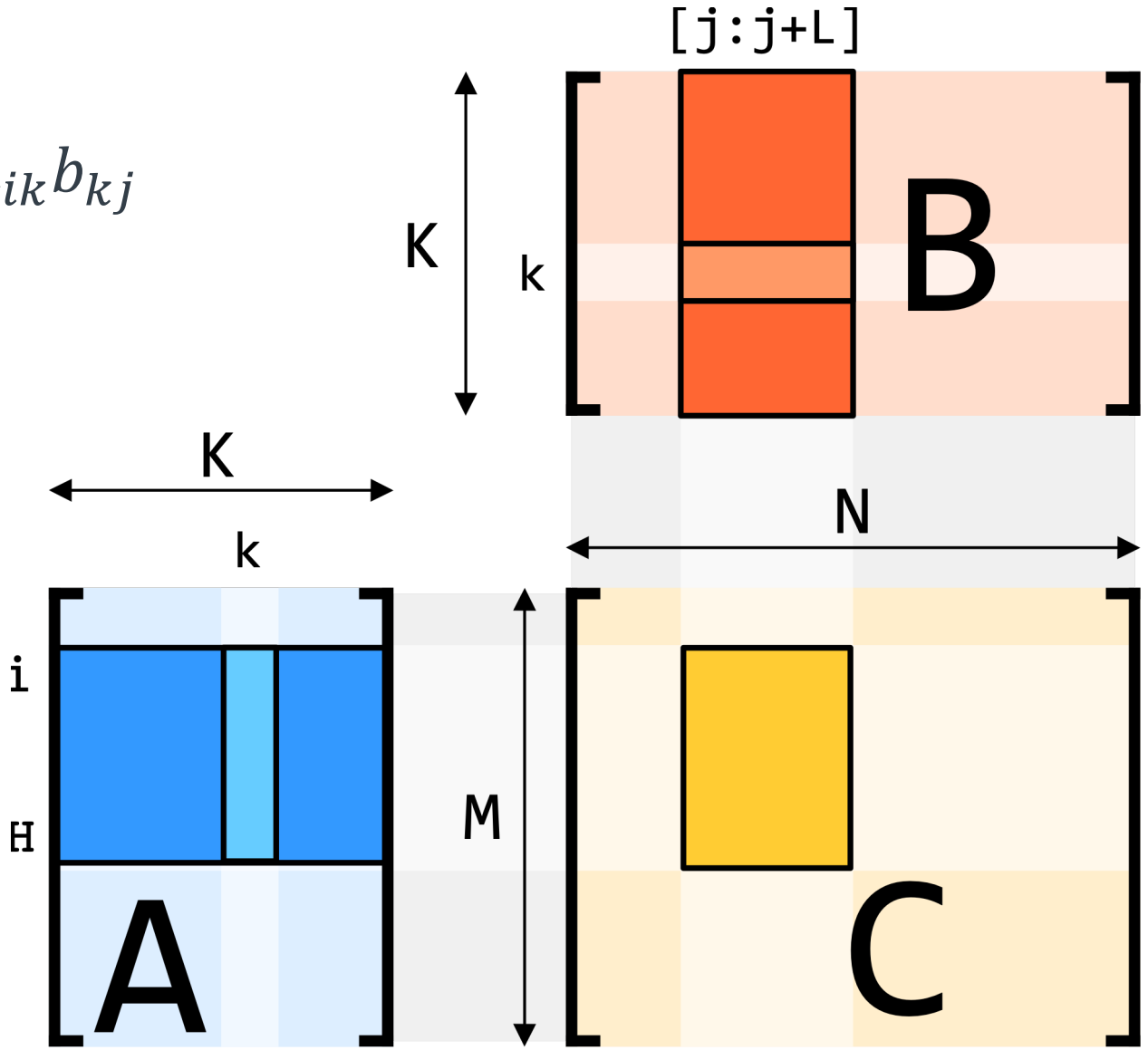
$$c_{ij} \mathrel{+}= \sum_{k=0}^{K-1} a_{ik} b_{kj}$$



arm

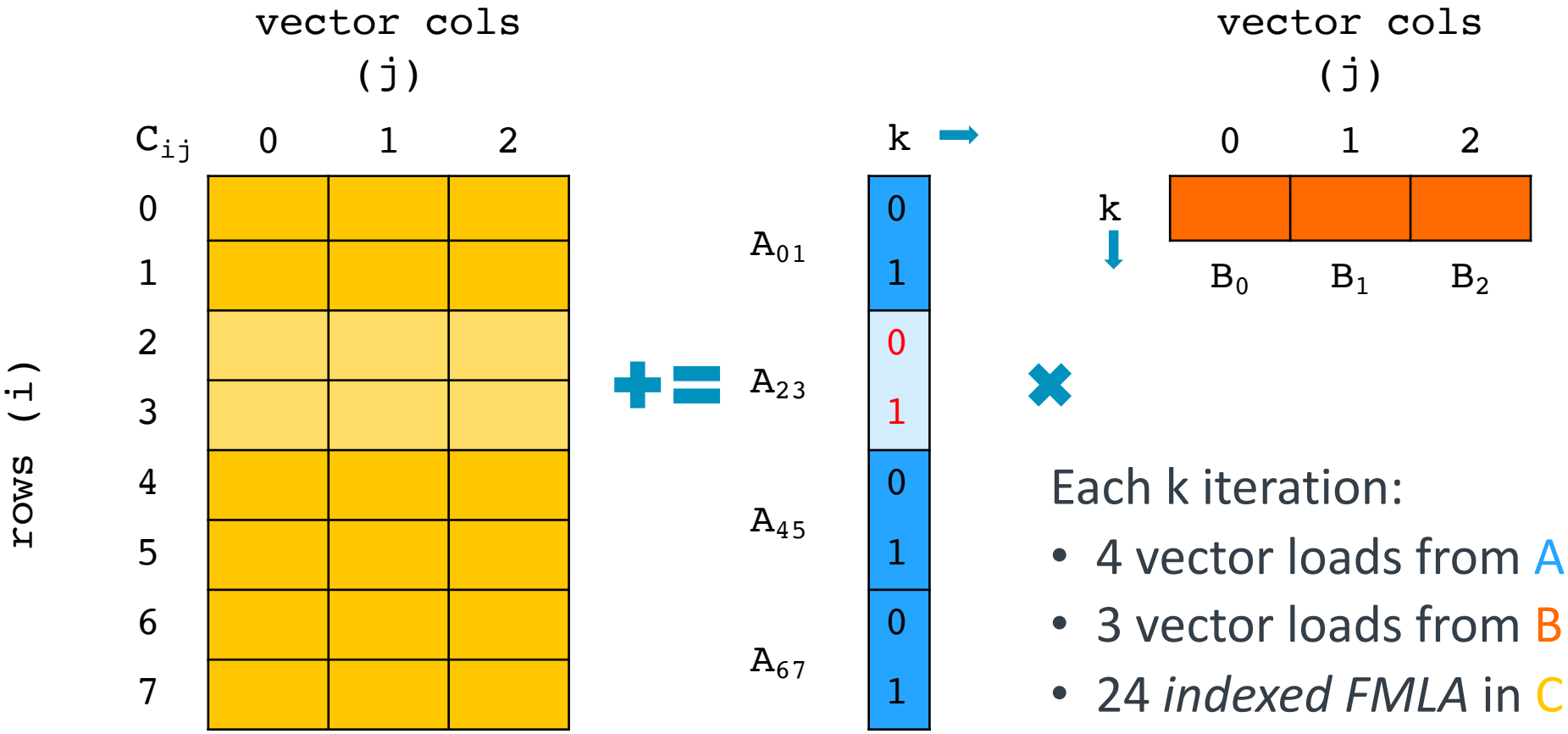
GEMM – step 2 – unroll along rows

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



arm

DGEMM kernel (NEON, 128-bit, 24 accumulators + 4 + 3)



Each k iteration:

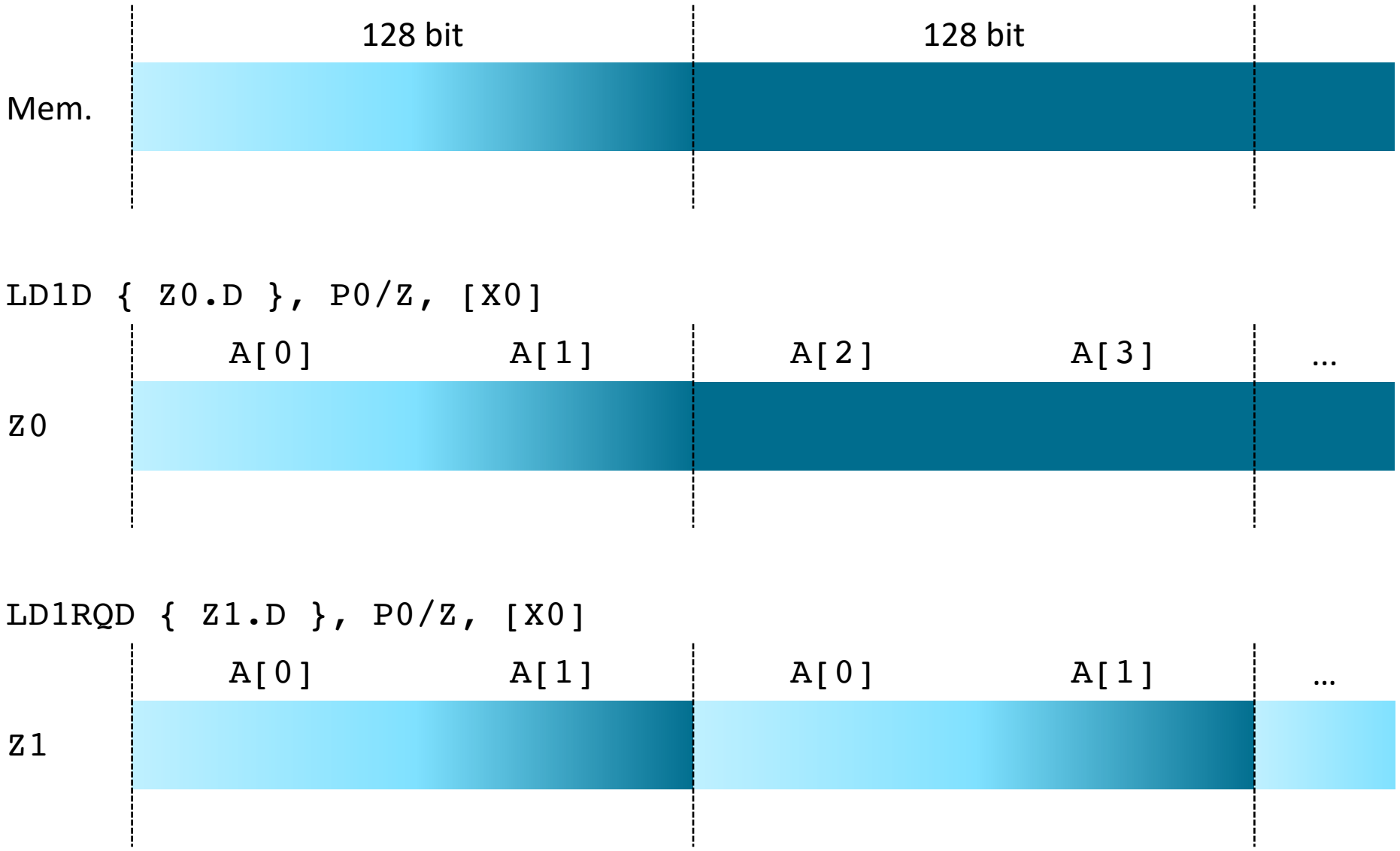
- 4 vector loads from A
- 3 vector loads from B
- 24 *indexed FMLA* in C

```
fmla C20.2d, B0.2d, A23.d[0] fmla C30.2d, B0.2d, A23.d[1]
fmla C21.2d, B1.2d, A23.d[0] fmla C31.2d, B1.2d, A23.d[1]
fmla C22.2d, B2.2d, A23.d[0] fmla C32.2d, B2.2d, A23.d[1]
```


arm

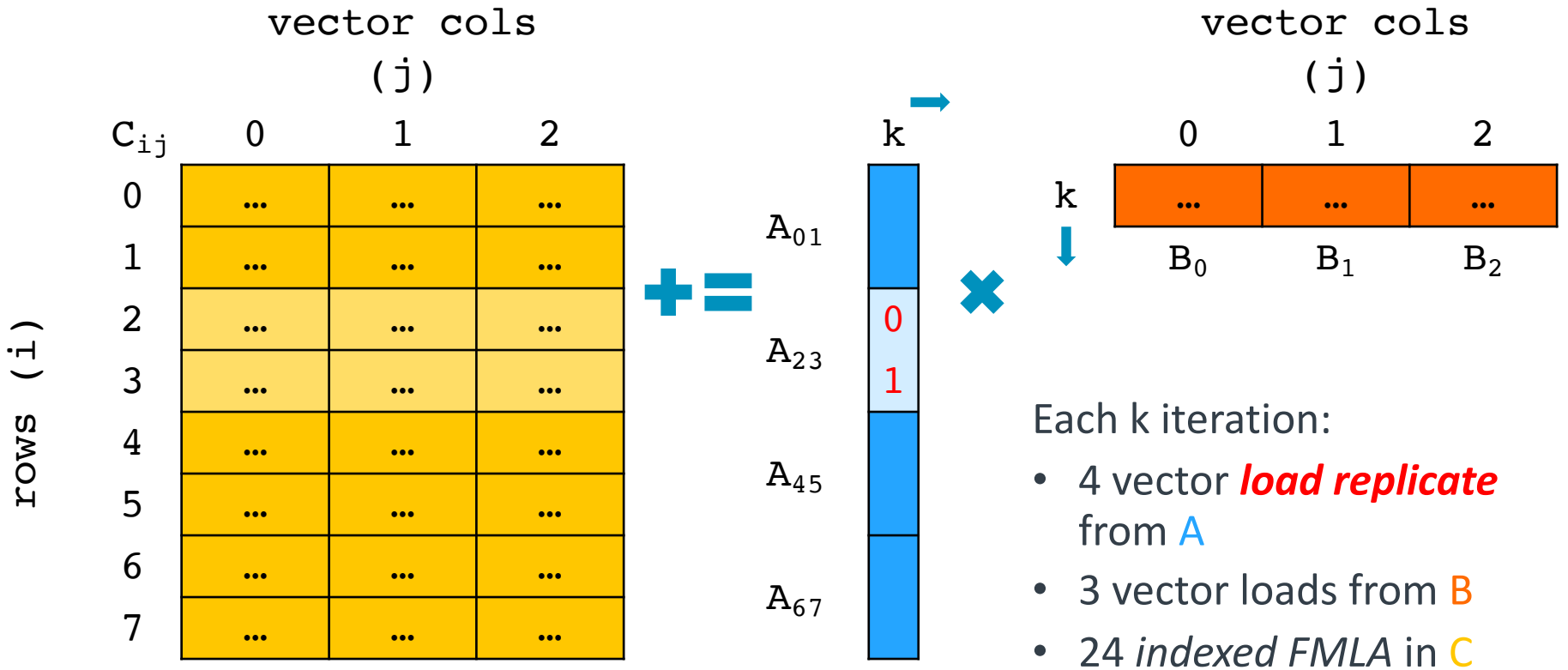
SVE load Replicate Quadword instructions: LD1RQ[BHWD]

```
double *A;
```



arm

DGEMM kernel (SVE, ***LEN x 128-bit***, 24 accumulators + 4 + 3)



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]
```

```
fmla C30.d, B0.d, A23.d[1]
fmla C31.d, B1.d, A23.d[1]
fmla C32.d, B2.d, A23.d[1]
```

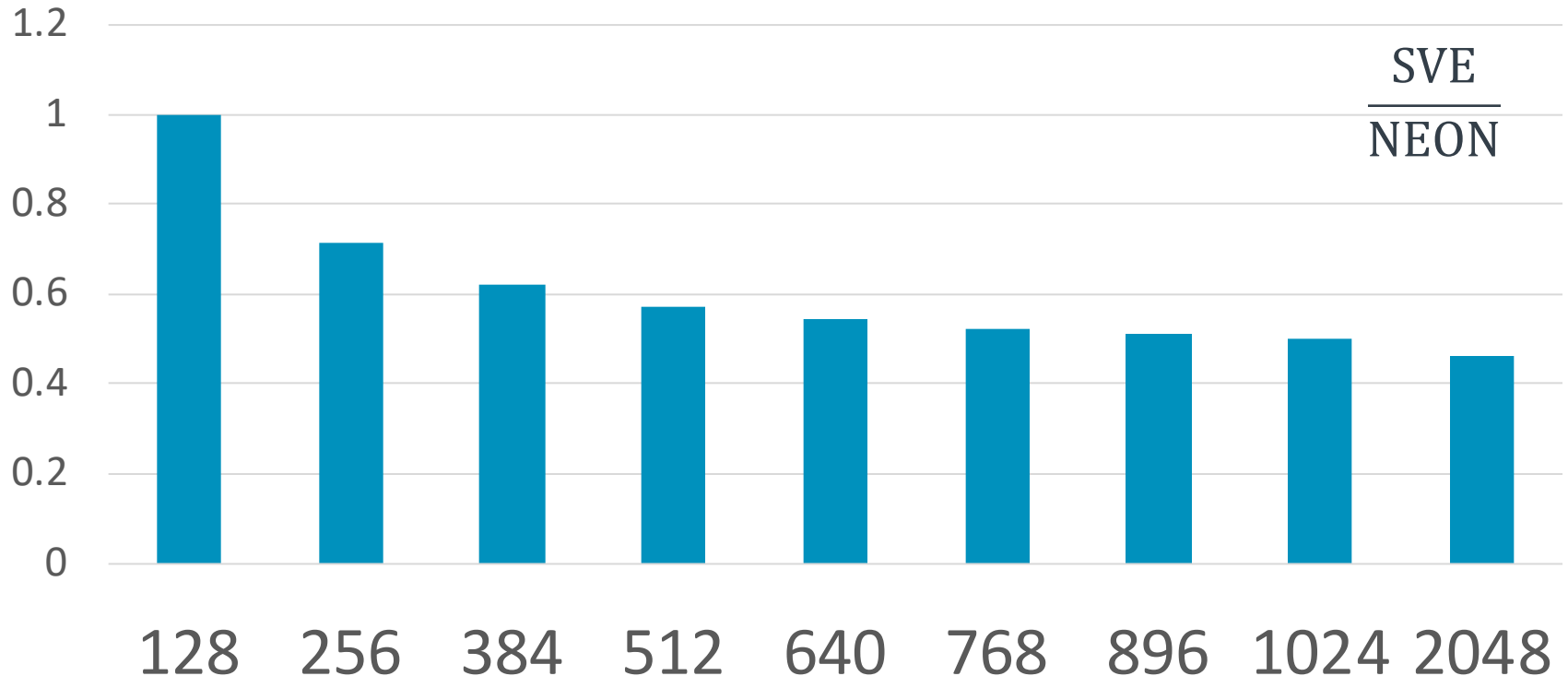

arm

DGEMM: SVE vs NEON

	Each k iteration	Bytes in one k iteration	Total C area computed	SVE/NEON A and B data reads for same C area
SVE	• 4 vector ***load replicate*** from A • 3 vector loads from B	4 x 128b + 3 x LEN x 128b	24 x 128b x LEN	$\frac{\text{SVE}}{\text{NEON}} = \frac{4 + 3\times\text{LEN}}{7\times\text{LEN}}$
NEON	• 4 vector loads from A • 3 vector loads from B	7 x 128b	24 x 128b	

arm

DGEMM: SVE more memory efficient than LEN times NEON



arm