arm

SVE Architecture & Optimization examples

SVE-Riken Summer School

Paris June 14th 2019.

Jelena Milanovic,
Arm Architecture & Technology

arm

Scalable Vector Extension (SVE)

Scalable Vector Extension

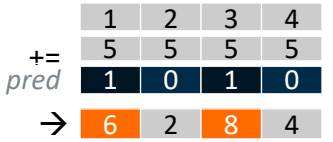
- **SVE is Vector Length Agnostic (VLA)**
  - Vector Length (VL) is a hardware implementation choice from 128 up to 2048 bits.
  - New programming model allows software to scale dynamically to available vector length.
  - No need to define a new ISA, rewrite or recompile for new vector lengths.
- **SVE is not an extension of Advanced SIMD (*aka* Neon)**
  - A separate, optional extension with a new set of instruction encodings.
  - Initial focus is HPC and general-purpose server, not media/image processing.
- **SVE begins to tackle traditional barriers to auto-vectorization**
  - Software-managed speculative vectorization allows uncounted loops to be vectorized.
  - In-vector serialised inner loop permits outer loop vectorization in spite of dependencies.

Other SVE features

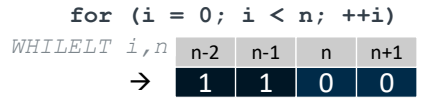
A SIMD vector extension to the Armv8-A architecture with important new features

- **Gather-load and scatter-store**
  - Loads a single vector register from non-contiguous memory locations.

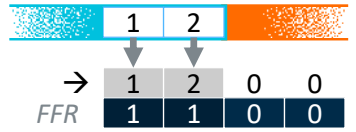
- **Per-lane predication**
  - Operate on individual lanes of vector controlled by of a governing predicate register.



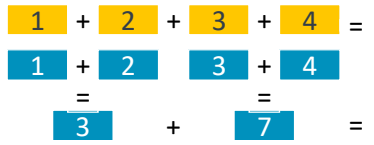
- **Predicate-driven loop control and management**
  - Eliminate loop heads and tails and other overhead by processing partial vectors.



- **Vector partitioning for software-managed speculation**
  - First-fault vector load instructions allow vector accesses to cross into invalid pages.



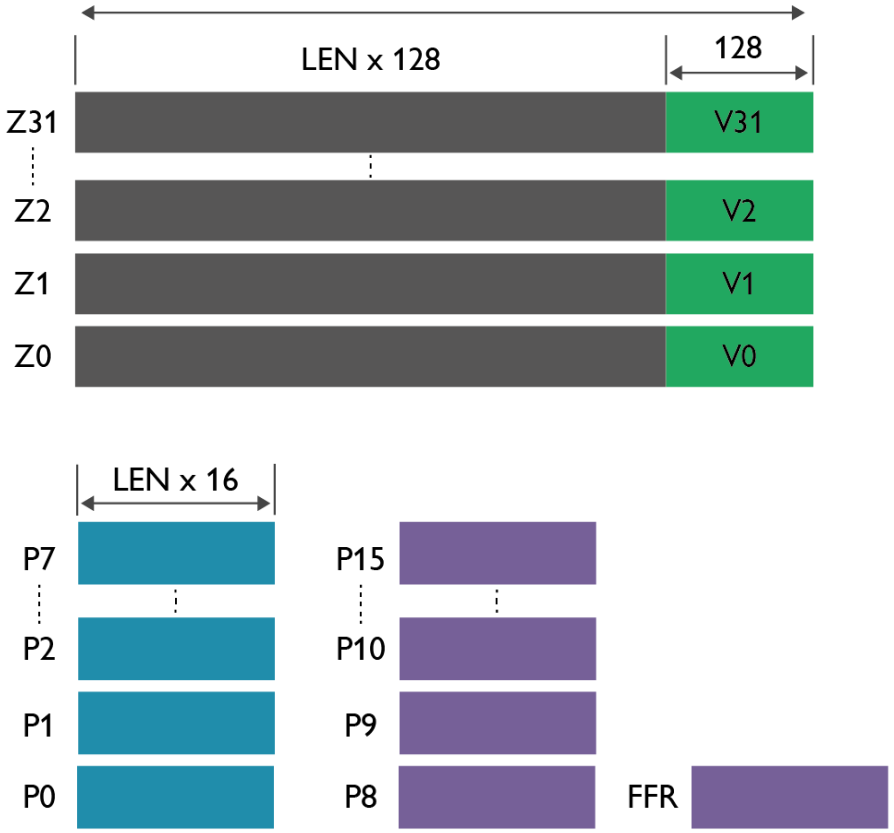
- **Extended floating-point and bitwise horizontal reductions**
  - In-order or tree-based floating-point sum, trade-off repeatability vs performance.



arm

SVE Registers

- **Scalable vector registers**
  - `Z0-Z31` extending NEON's 128-bit `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 software speculation.

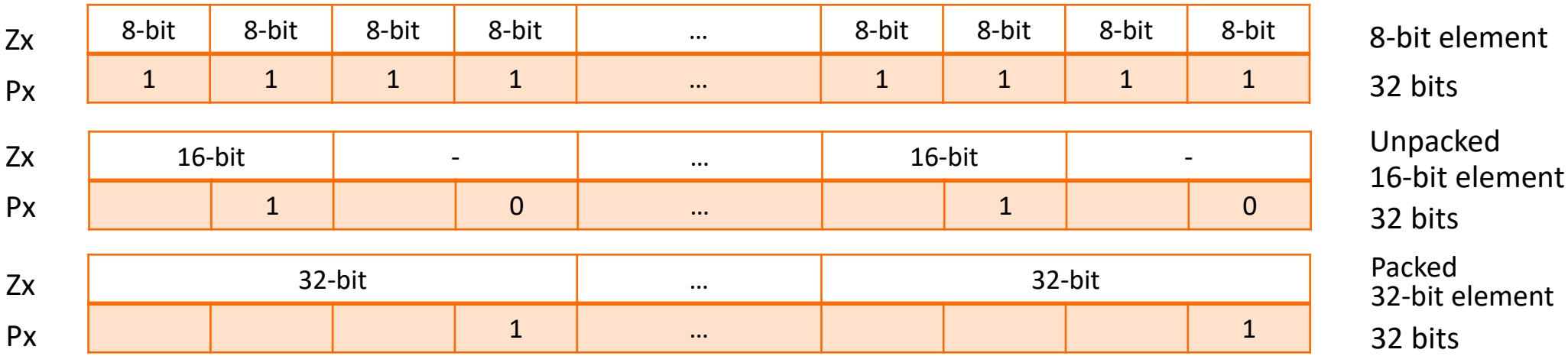


arm

Predicators

- **16 Scalable Predicate Registers P0-P15:**
  - Have 1/8th of a vector register's length. 1 bit of predicate register is mapped to 1 byte of vector register
  - Each predicate element size is 1/8th of a vector element's length. Only the lowest bit of each predicate element is significant: 1=active 0=inactive. Other bits are ignored in read and set to zero write
  - Predicate registers can be updated by operation status, and initialized by `PTRUE` and `PFALSE` instructions with one of the patterns: fixed length, power of 2, multiple of 3 or, `#uimm5`, or default all elements
  - Supporting Zeroing(`/Z`) or Merging(`/M`) to the inactive vector elements

- **Vector-predicate registers mapping examples** (Vector length 256 bits):



arm

SVE Predicate condition flags

SVE is a *predicate-centric* architecture

- Predicates are central, not an afterthought
- Support complex nested conditions and loops.
- Predicate generation also sets condition flags.
- Reduces vector loop management overhead.

Overloading the A64 NZCV condition flags

Flag	SVE	Condition
N	First	Set if first active element is true
Z	None	Set if no active element is true
C	!Last	Set if last active element is false
V		Scalarized loop state, else zero

Reuses the A64 conditional instructions

- Conditional branches B.EQ → B.NONE
- Conditional select, set, increment, etc.

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

arm

Vector Length Agnostic (VLA) instructions

- Vectors cannot be initialised from compile-time constant in memory, so…
  - `INDEX Zd.S,#1,#4` : `Zd = [ 1, 5, 9, 13, 17, 21, 25, 29 ]`
- Predicates also cannot be initialised 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 64-bit dwords in vector
  - `WHILELT Pd.D,Xi,Xe` : next iteration predicate `Pd = [ while i++ < e ]`
- Vector register spill & fill must adjust to vector length, so…
  - `ADDVL SP,SP,#-4` : decrement stack pointer by (4*VL)
  - `STR Z1,[SP,#3,MUL VL]` : store vector Z1 to address (SP + 3*VL)

arm

SVE Vector partitioning

- **Vector partitioning allows software-managed speculative vectorisation**
  - Predicates create sub-vectors (partitions) in response to data and dynamic faults.
- **First-fault load allows vector 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 (do...while, if…break, etc.)**
  - Load data using first-fault load and create a *before-fault* partition from FFR.
  - Test for break condition and create a *before-break* partition from condition predicate.
  - Process data within partition, then exit loop if break condition was found.
- **Vector length agnosticism**
  - Just a special case of vector partitioning where partition is determined by the current vector length.

arm

SVE2
Arm New technology

SVE2 goals and novelties

Write in your subtitle here

- Improve applicability of SVE to a broader range of domains than HPC
- Build on SVE foundations to achieve scalable performance
  - With parity at 128-bit for traditional Neon media & DSP workloads
  - No reason to prefer Neon over SVE2 for new software development
  - Improve competitiveness of general-purpose ARM processors vs proprietary DSP solutions
- Optimize for emerging applications
  - ML, CV, baseband networking, genomics, database, server/enterprise, etc
  - “Good enough” performance for a general-purpose processor w/o hardware acceleration
- Extend benefits of SVE auto-vectorization to such applications
  - Reducing s/w development and deployment cost

SVE2 novelties

Write in your subtitle here

- SVE2 instructions set adds:
  - Thorough support for fixed-point DSP arithmetic (traditional Neon DSP/Media processing, complex numbers arithmetic for LTE),
  - Multiprecision arithmetic (bignum, crypto),
  - Non-temporal gather/scatter (HPC, sort),
  - Enhanced permute and bitwise permute instructions (CV, FIR, FFT, LTE, ML, genomics, cryptanalysis),
  - Histogram acceleration support (CV, HPC, sort),
  - String processing acceleration support (parsers),
  - (optional) Cryptography support instructions for AES, SM4, SHA standards (encryption).

Top-Bottom approach (1)

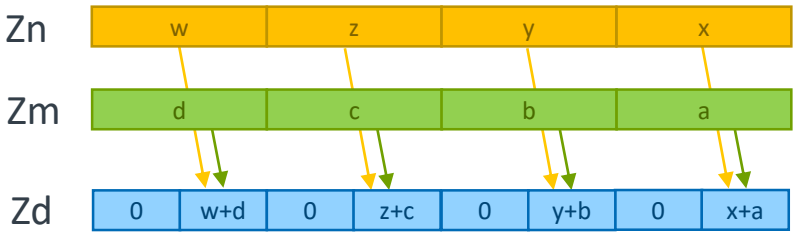
Write in your subtitle here

- Moving across lanes is costly for wider vectors (VL > 128b)
- Widening instructions
  - SVE2 operates on even (Bottom instructions) or odd (Top instructions) elements and widens “in lane”.
  - Widening instruction deinterleaves elements.
- Narrowing instructions
  - SVE2 produces even (Bottom instructions) or odd (Top instructions) results and narrows “in lane”.
  - Narrowing instruction reinterleaves elements.

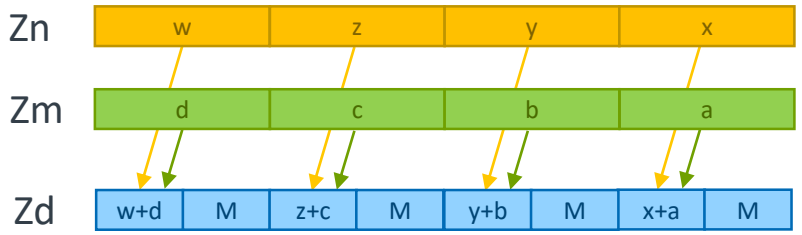
Top-Bottom approach (2)

Narrowing instruction

ADDHNB Zd.T, Zn.Tb, Zm.Tb



ADDHNT Zd.T, Zn.Tb, Zm.Tb

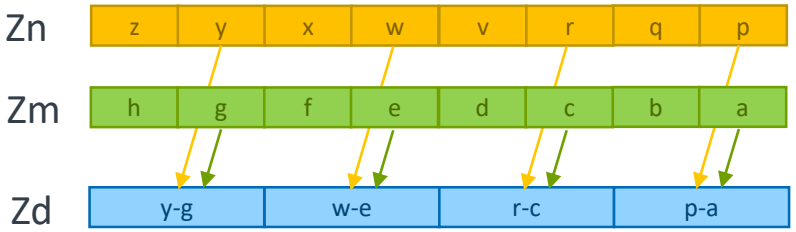


arm

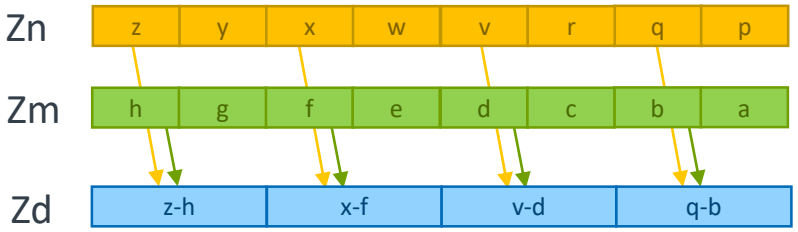
Top-Bottom approach (3)

Widening instructions

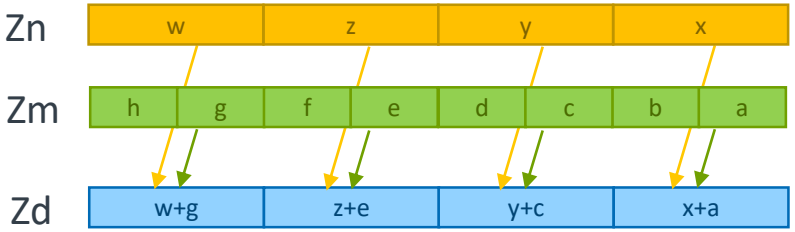
USUBLB Zd.T, Zn.Tb, Zm.Tb



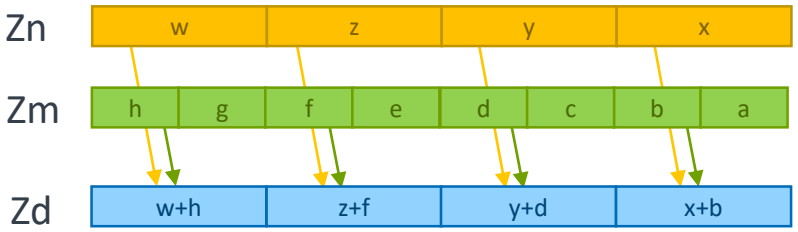
USUBLT Zd.T, Zn.Tb, Zm.Tb



SADDWB Zd.T, Zn.T, Zm.Tb



SADDWT Zd.T, Zn.T, Zm.Tb



arm

arm

SVE Examples
daxpy

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
```


arm

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
```

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

daxpy (scalar)

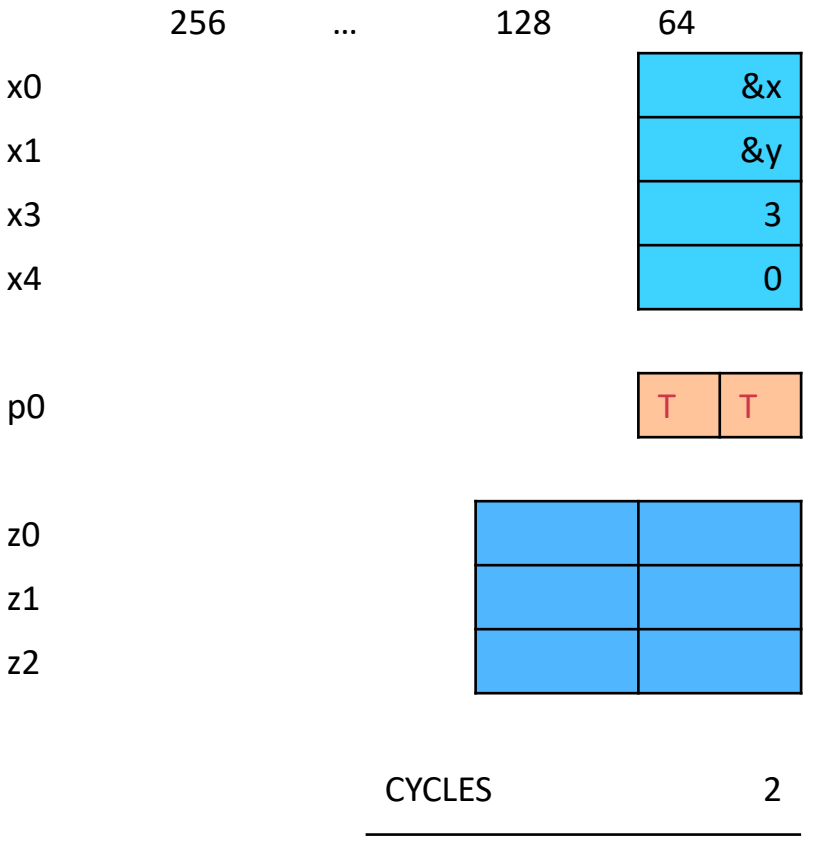
```
daxpy_:
        ldrsw           x3, [x3]
        mov             x4, #0
        ldr             d0, [x2]
        b               .latch
.loop:
        ldr             d1, [x0, x4, lsl #3]
        ldr             d2, [x1, x4, lsl #3]
        fmadd           d2, d1, d0, d2
        str             d2,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


arm

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

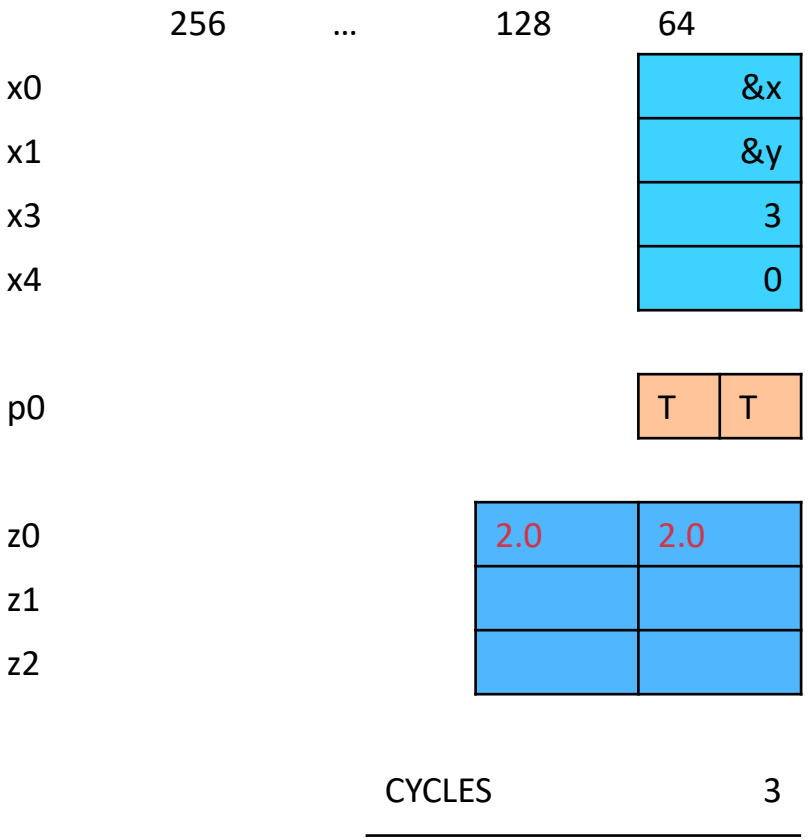


arm

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

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				

CYCLES 4

arm

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

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				0
p0				T | T
z0			2.0	2.0
z1			1.0	0.0
z2			2.0	0.0

CYCLES 7

arm

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

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

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 10

arm

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			2.0	0.0

CYCLES 11

arm

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

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	4.0

CYCLES	13

arm

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

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 | T
z0			2.0	2.0
z1			0.0	2.0
z2			0.0	4.0

CYCLES	15

arm

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

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

arm

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

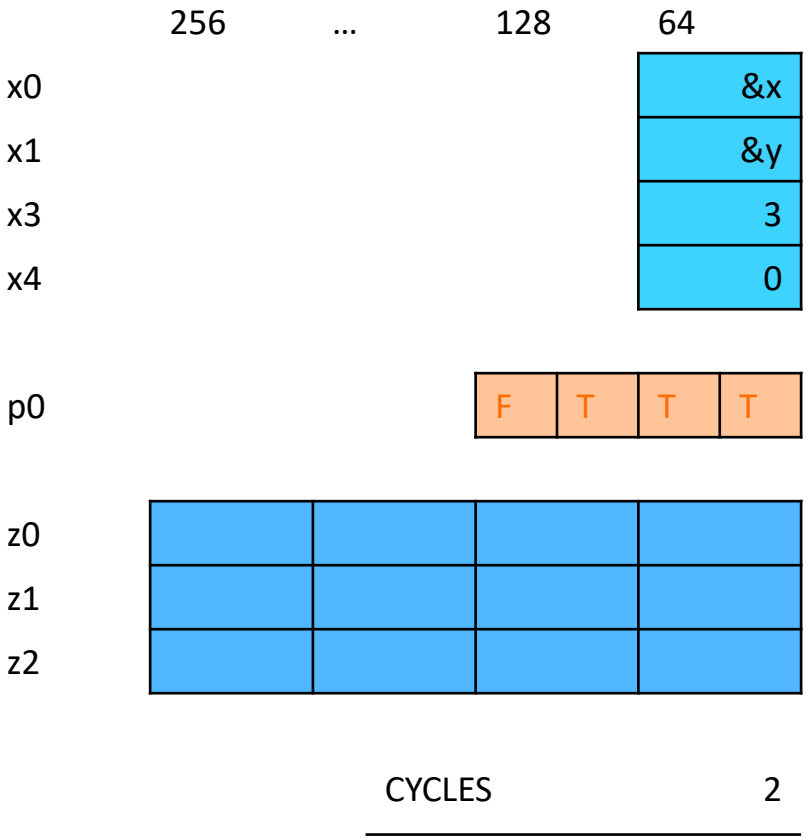
CYCLES 18

arm

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

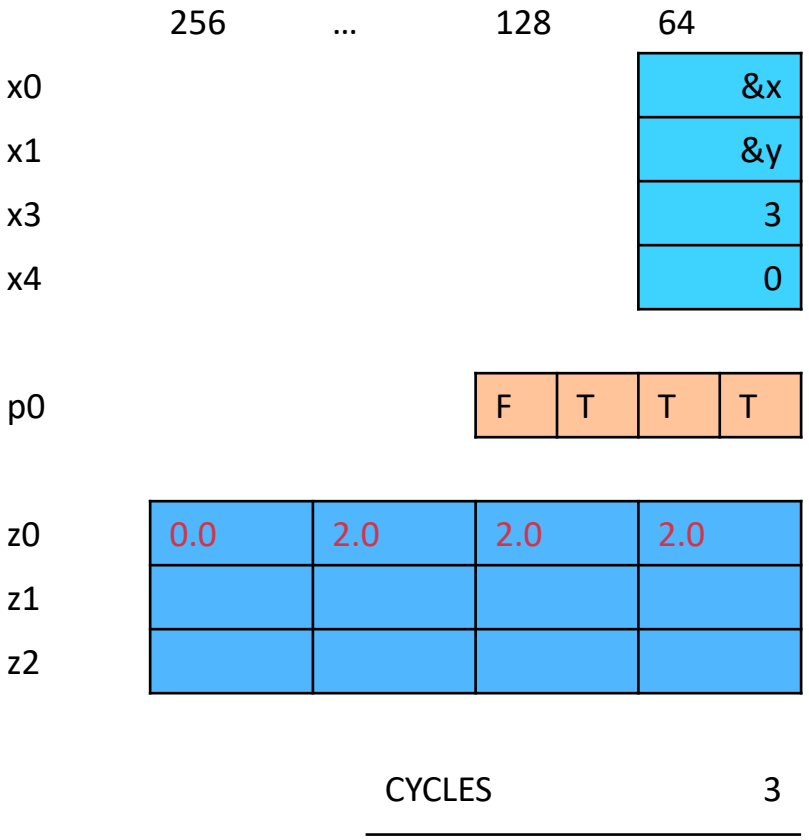


arm

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

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				

CYCLES 4

arm

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

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

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

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

arm

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

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

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

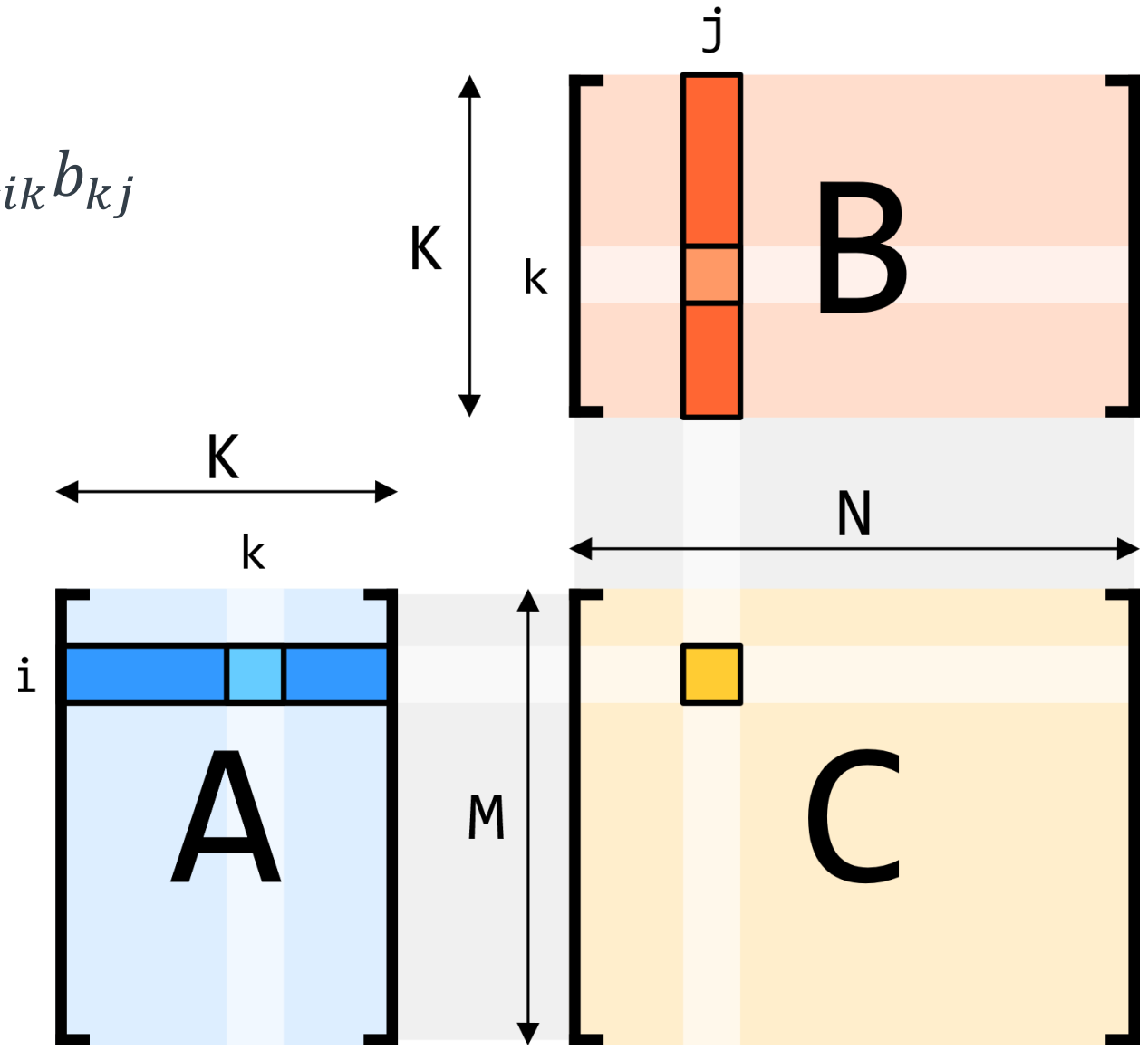
arm

arm

SVE Examples dgemm

GEMM

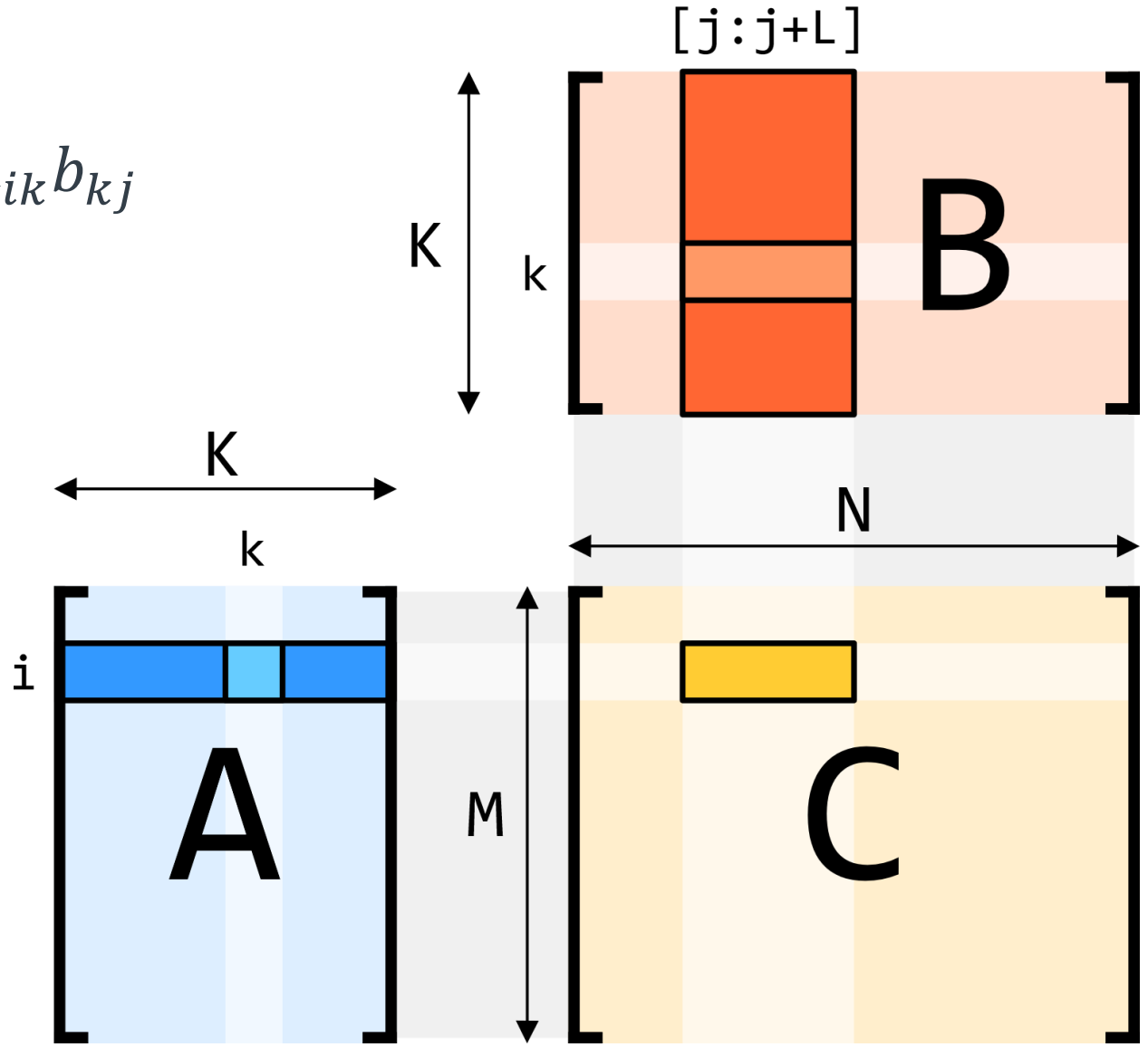
$$c_{ij} \mathrel{+}= \sum_{k=0}^{K-1} a_{ik} b_{kj}$$



arm

GEMM – step 1 – vectorize along the columns

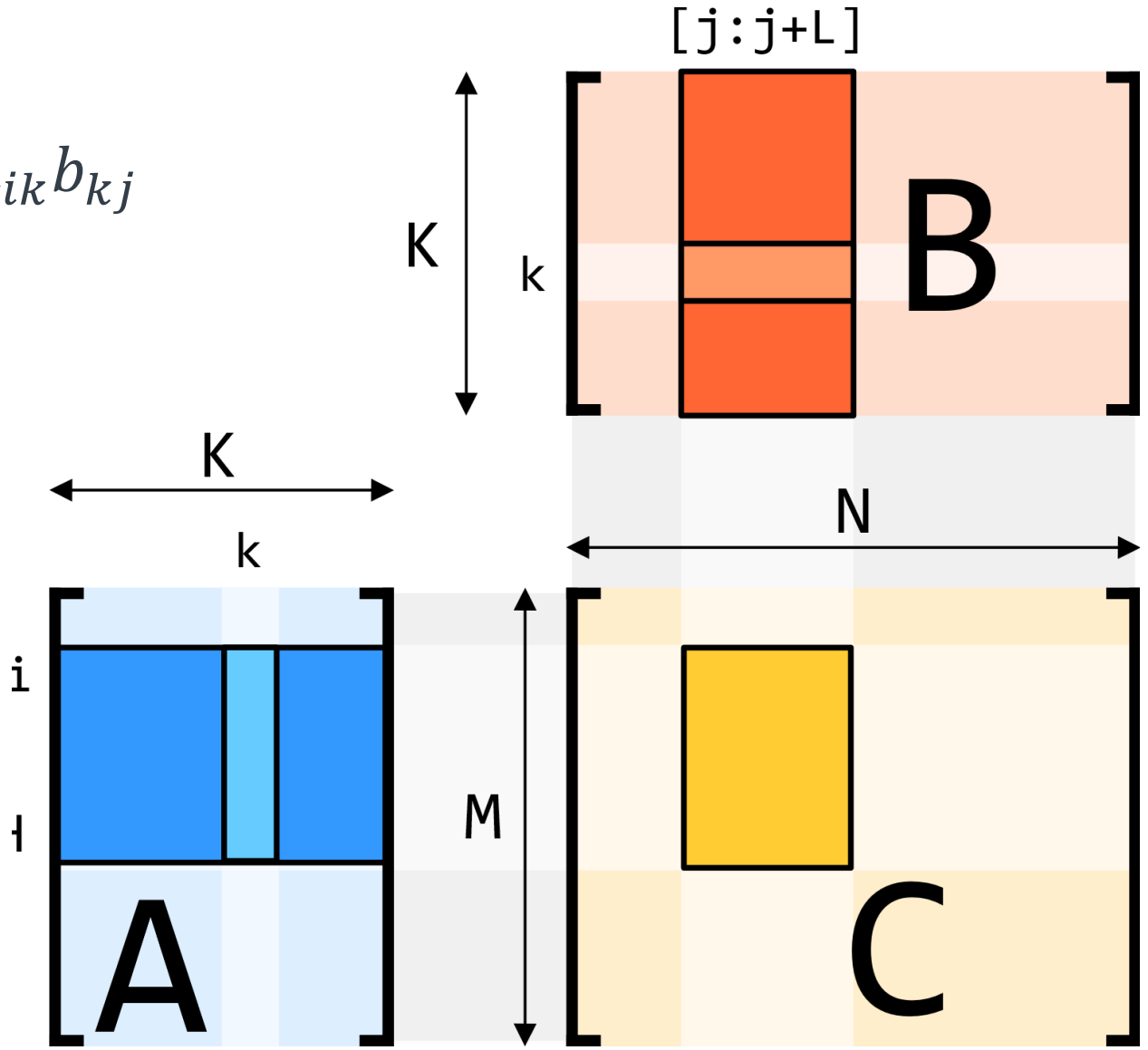
$$c_{ij} \mathrel{+}= \sum_{k=0}^{K-1} a_{ik} b_{kj}$$



arm

GEMM – step 2 – unroll along rows

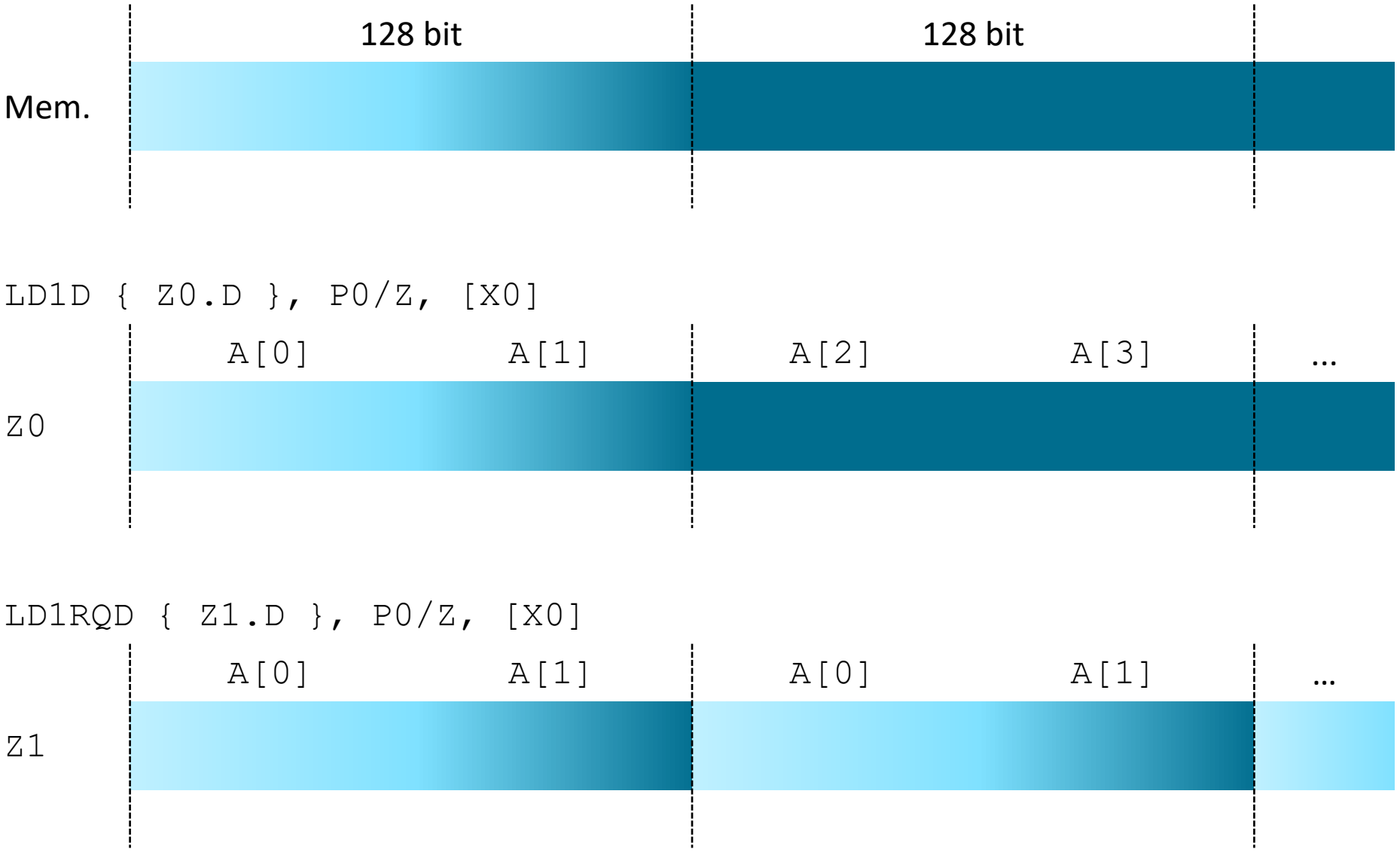
$$c_{ij} \mathrel{+}= \sum_{k=0}^{K-1} a_{ik} b_{kj}$$



arm

SVE load Replicate Quadword instructions: LD1RQ[BHWD]

```
double *A;
```



arm

FP64 inLeft[M, K] * inRight[K, N] = out[M, N]

- The following SVE vectorization assumptions are taken:
- Minimum matrix dimension is 32.
- Matrix dimensions are multiple of 16.

```
void matmul_f64_C( uint64_t M, uint64_t K, uint64_t N,
                   float64_t * inLeft, float64_t * inRight, float64_t * out) {
  uint64_t x, y, z;
  float64_t acc;
  for (x=0; x<M; x++) {
    for (y=0; y<N; y++) {
      acc = 0.0;
      for (z=0; z<K; z++) {
        acc += inLeft[x*K + z] * inRight[z*N + y];
      }
      out[x*N + y] = acc;
} } }
```

```
void matmul_f64( uint64_t M, uint64_t K, uint64_t N,
                 float64_t * inLeft, float64_t * inRight, float64_t * out) {
  uint64_t x, y, z;
  svbool_t p64_all = svptrue_b64();
  uint64_t vl = svcntd();
  uint64_t offsetIN_1, offsetIN_2, offsetIN_3;
  uint64_t offsetOUT_1, offsetOUT_2, offsetOUT_3;
  float64_t *ptrIN_left;
  float64_t *ptrIN_right;
  float64_t *ptrOUT;
  svfloat64_t acc0, acc1, acc2, acc3;
  svfloat64_t inR_0, inR_1;
  svfloat64_t inL_0, inL_1, inL_2, inL_3;
  offsetIN_1 = K;
  offsetIN_2 = 2*K;
  offsetIN_3 = 3*K;
  offsetOUT_1 = N;
  offsetOUT_2 = 2*N;
  offsetOUT_3 = 3*N;
  for (x=0; x<M; x+=4) {
    ptrOUT = &out[x*N];
    for (y=0; y<N; y+=vl) {
      acc0 = acc1 = acc2 = acc3 = svdup_f64(0.0);
      ptrIN_left = &inLeft[x*K];
      ptrIN_right = &inRight[y];
      for (z=0; z<K; z+=2) {
        inR_0 = svld1(p64_all, ptrIN_right);
        inR_1 = svld1(p64_all, &ptrIN_right[offsetOUT_1]);
        inL_0 = svld1rq(p64_all, ptrIN_left);
        inL_1 = svld1rq(p64_all, &ptrIN_left[offsetIN_1]);
        inL_2 = svld1rq(p64_all, &ptrIN_left[offsetIN_2]);
        inL_3 = svld1rq(p64_all, &ptrIN_left[offsetIN_3]);
        acc0 = svmla_lane(acc0, inR_0, inL_0, 0);
        acc0 = svmla_lane(acc0, inR_1, inL_0, 1);
        acc1 = svmla_lane(acc1, inR_0, inL_1, 0);
        acc1 = svmla_lane(acc1, inR_1, inL_1, 1);
        acc2 = svmla_lane(acc2, inR_0, inL_2, 0);
        acc2 = svmla_lane(acc2, inR_1, inL_2, 1);
        acc3 = svmla_lane(acc3, inR_0, inL_3, 0);
        acc3 = svmla_lane(acc3, inR_1, inL_3, 1);
        ptrIN_right += 2*N;
        ptrIN_left += 2;
      }
      svst1(p64_all, ptrOUT, acc0);
      svst1(p64_all, &ptrOUT[offsetOUT_1], acc1);
      svst1(p64_all, &ptrOUT[offsetOUT_2], acc2);
      svst1(p64_all, &ptrOUT[offsetOUT_3], acc3);
      ptrOUT += vl;
} } }
```


arm

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