Chapter 4

Data-Level Parallelism

Data-Level Parallelism Types

- Vector processors
- SIMD Instructions
- GPUs

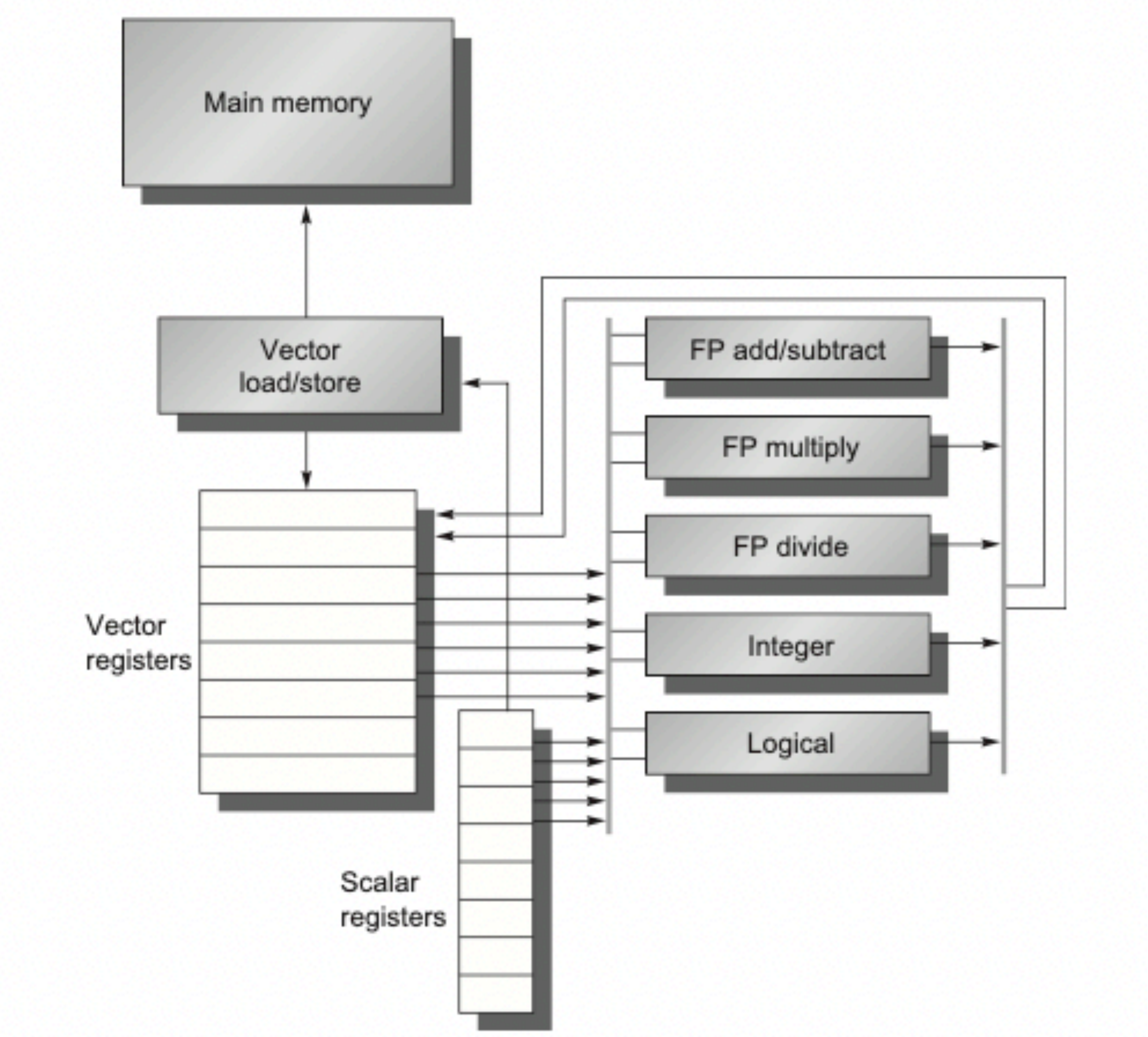


Figure 4.1 The basic structure of a vector architecture, RV64V, which includes a RISC-V scalar architecture. There are also 32 vector registers, and all the functional units

Mnemonic	Name	Description
vadd	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
vsub	SUBtract	Subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
vmul	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
vdiv	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
vrem	REMainder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
vsqrt	SQuare RooT	Take square root of elements of V[rs1], then put each result in V[rd]
vsll	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
vsrl	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
vsra	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
vxor	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vor	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vand	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
vsgnj	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
vsgnjn	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
vsgnjx	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]
vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vpeq	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpne	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vplt	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpxor	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpor	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpand	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
setvl	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source regsiter

Figure 4.2 The RV64V vector instructions. All use the R instruction format. Each vector operation with two operands

Example Show the code for RV64G and RV64V for the DAXPY loop. For this example, assume that X and Y have 32 elements and the starting addresses of X and Y are in x5 and x6, respectively. (A subsequent example covers when they do not have 32 elements.)

Answer Here is the RISC-V code:

```
      fld    f0,a          # Load scalar a
      addi   x28,x5,#256   # Last address to load
Loop: fld    f1,0(x5)      # Load X[i]
      fmul.d f1,f1,f0      # a × X[i]
      fld    f2,0(x6)      # Load Y[i]
      fadd.d f2,f2,f1      # a × X[i] + Y[i]
      fsd    f2,0(x6)      # Store into Y[i]
      addi   x5,x5,#8      # Increment index to X
      addi   x6,x6,#8      # Increment index to Y
      bne    x28,x5,Loop   # Check if done
```

Here is the RV64V code for DAXPY:

```
vsetdcfg  4*FP64      # Enable 4 DP FP vregs
fld       f0,a        # Load scalar a
vld       v0,x5       # Load vector X
vmul      v1,v0,f0    # Vector-scalar mult
vld       v2,x6       # Load vector Y
vadd      v3,v1,v2    # Vector-vector add
vst       v3,x6       # Store the sum
vdisable              # Disable vector regs
```

Note that the assembler determines which version of the vector operations to generate. Because the multiply has a scalar operand, it generates vmul.vs, whereas the add doesn't, so it generates vadd.vv.

Example A common use of multiply-accumulate operations is to multiply using narrow data and to accumulate at a wider size to increase the accuracy of a sum of products. Show how the preceding code would change if X and a were single-precision instead of a double-precision floating point. Next, show the changes to this code if we switch X, Y, and a from floating-point type to integers.

Answer The changes are underlined in the following code. Amazingly, the same code works with two small changes: the configuration instruction includes one single-precision vector, and the scalar load is now single-precision:

```
vsetdcfg 1*FP32,3*FP64  # 1 32b, 3 64b vregs
flw      f0,a           # Load scalar a
vld      v0,x5          # Load vector X
vmul     v1,v0,f0       # Vector-scalar mult
vld      v2,x6          # Load vector Y
vadd     v3,v1,v2       # Vector-vector add
vst      v3,x6          # Store the sum
vdisable                # Disable vector regs
```

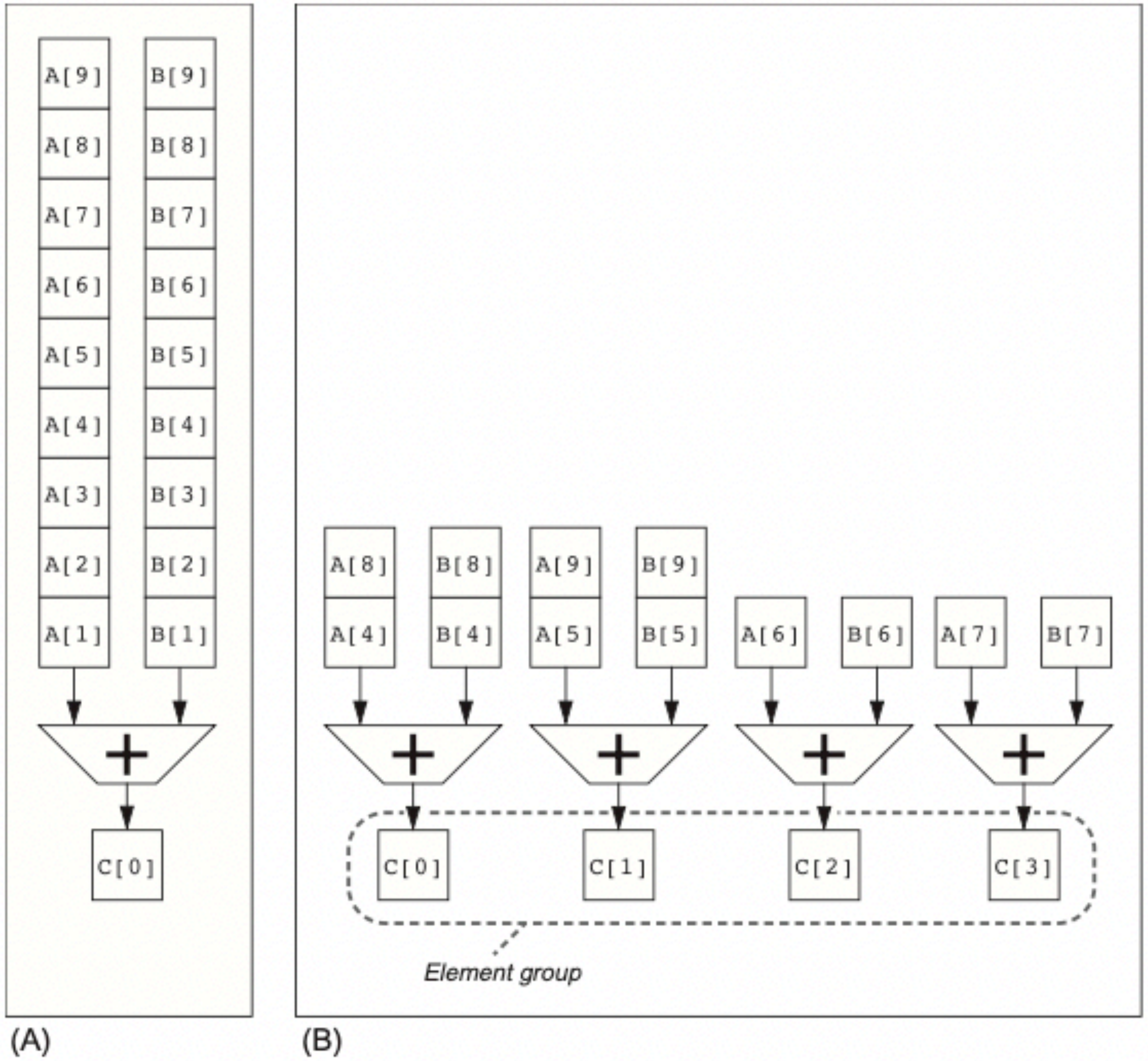


Figure 4.4 Using multiple functional units to improve the performance of a single vector add instruction,

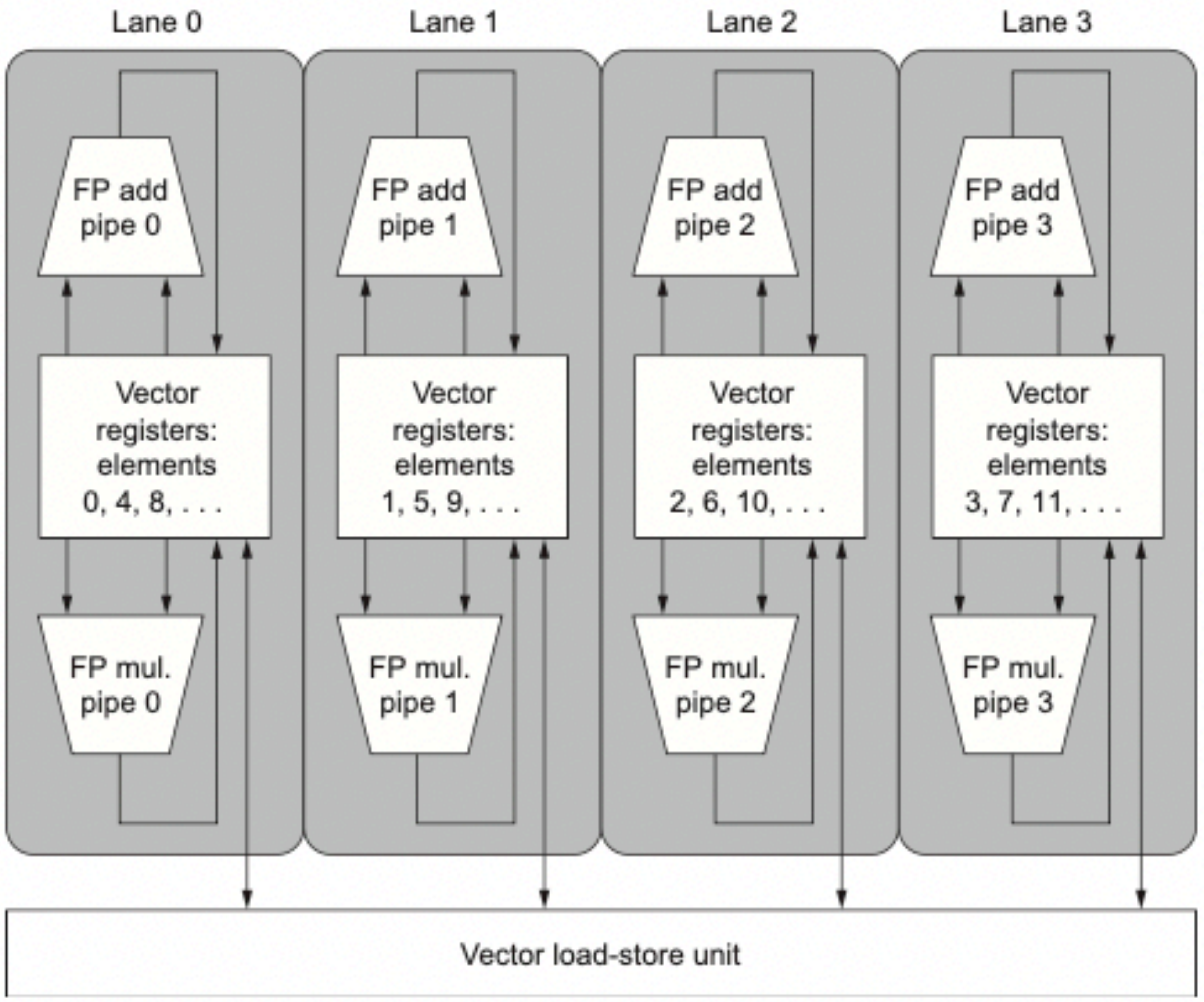


Figure 4.5 Structure of a vector unit containing four lanes. The vector register mem-

Vector Length Register

```
      vsetdcfg  2 DP FP     # Enable 2 64b Fl.Pt. registers
      fld       f0,a        # Load scalar a
loop: setvl     t0,a0       # vl = t0 = min(mvl,n)
      vld       v0,x5       # Load vector X
      slli      t1,t0,3     # t1 = vl * 8 (in bytes)
      add       x5,x5,t1    # Increment pointer to X by vl*8
      vmul      v0,v0,f0    # Vector-scalar mult
      vld       v1,x6       # Load vector Y
      vadd      v1,v0,v1    # Vector-vector add
      sub       a0,a0,t0    # n -= vl (t0)
      vst       v1,x6       # Store the sum into Y
      add       x6,x6,t1    # Increment pointer to Y by vl*8
      bnez      a0,loop     # Repeat if n != 0
      vdisable              # Disable vector regs
```

Vector Register Example

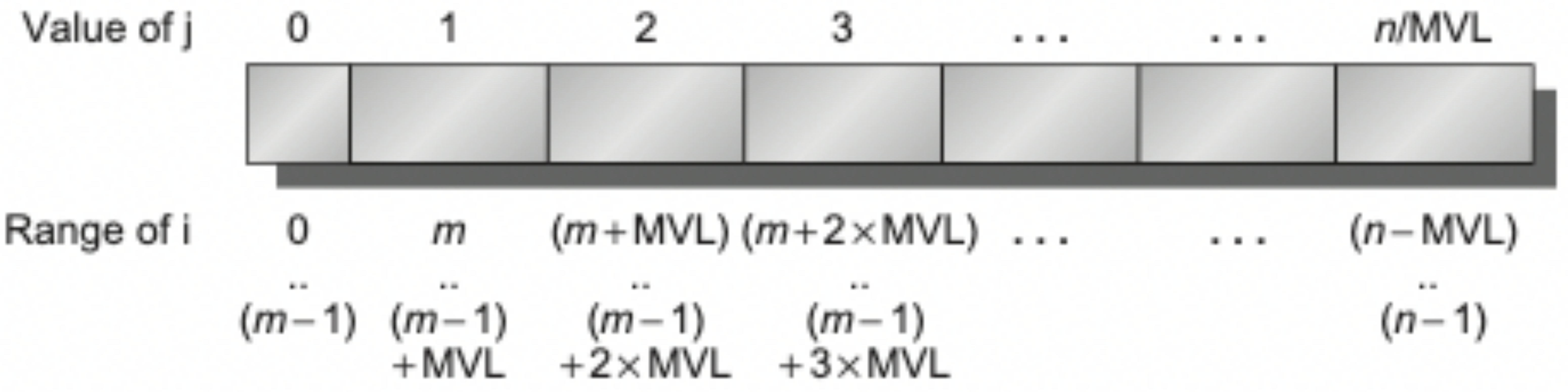


Figure 4.6 A vector of arbitrary length processed with strip mining. All blocks but the

Predicate Register

Original code

```
for (i = 0; i < 64;   i=i+1)
  if (X[i] != 0)
    X[i] = X[i] - Y[i];
```

Vector code (with Predicate)

```
vsetdcfg    2*FP64      # Enable 2 64b FP vector regs
vsetpcfgi   1           # Enable 1 predicate register
vld         v0,x5       # Load vector X into v0
vld         v1,x6       # Load vector Y into v1
fmv.d.x     f0,x0       # Put (FP) zero into f0
vpne        p0,v0,f0    # Set p0(i) to 1 if v0(i)!=f0
vsub        v0,v0,v1    # Subtract under vector mask
vst         v0,x5       # Store the result in X
vdisable                # Disable vector registers
vpdisable               # Disable predicate registers
```

Multidimensional Arrays

Stride - distance between elements in rows (or columns) in a multidimensional array

- RV64V uses `VLDS` instruction to set stride
    - Load V[rd] starting at addr R[rs1] with stride specified in R[rs2

Sparse Matrices - Gather-Scatter

Original Code

```
for (i = 0; i < n;  i=i+1)
  A[K[i]] = A[K[i]] + C[M[i]];
```

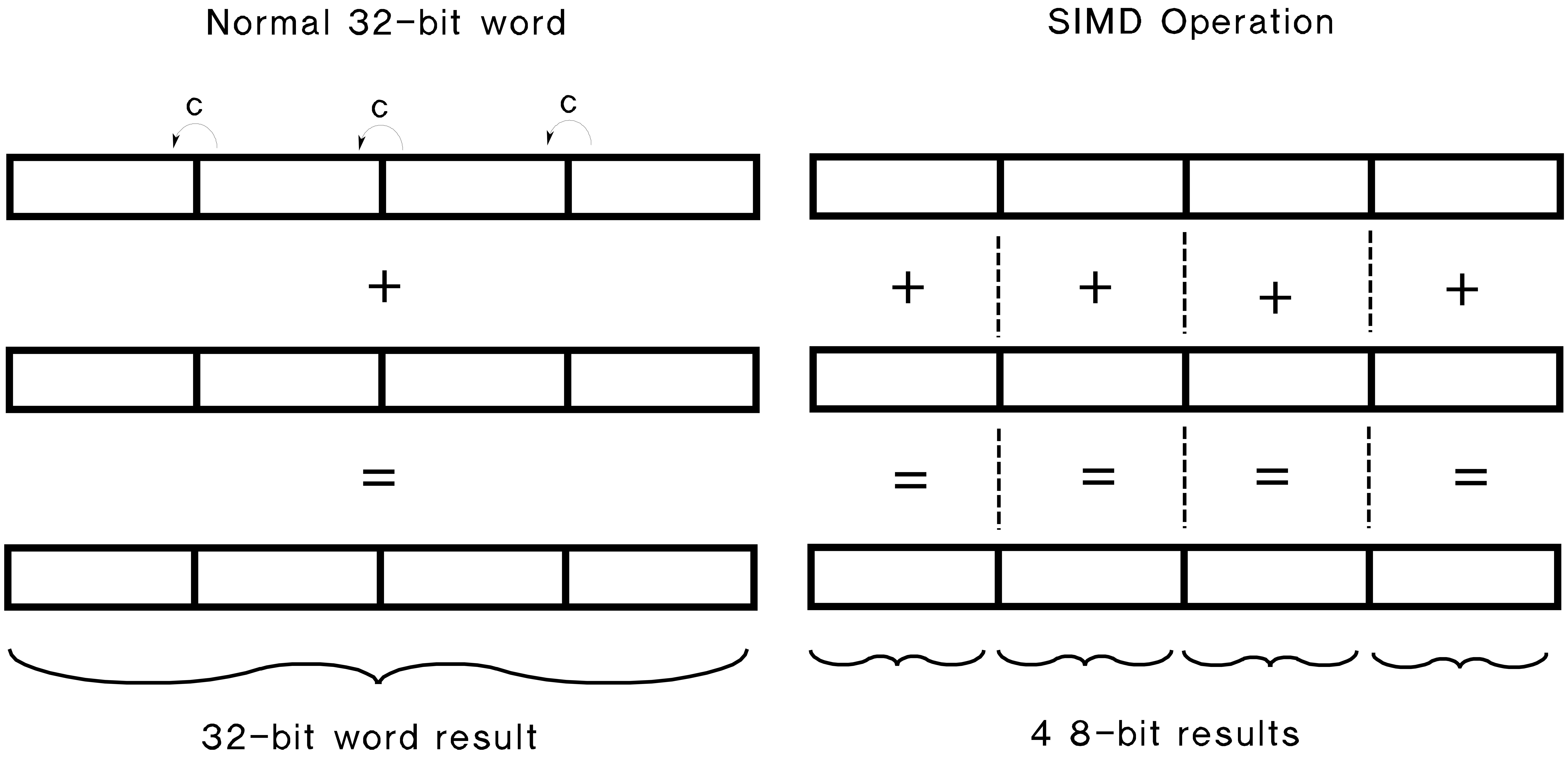
Vector Code

```
vsetdcfg  4*FP64        # 4  64b  FP vector registers
vld       v0, x7        # Load K[]
vldx      v1, x5, v0)   # Load A[K[]]
vld       v2, x28       # Load M[]
vldi      v3, x6, v2)   # Load C[M[]]
vadd      v1, v1, v3    # Add them
vstx      v1, x5, v0)   # Store A[K[]]
vdisable                # Disable vector registers
```

Flynn’s Taxonomy (1966)

- SISD (Single Instruction, Single Data)
- SIMD (Single Instruction, Multiple Data)
- MISD (Multiple Instruction, Single Data)
- MIMD (Multiple Instruction, Multiple Data)

SIMD ADD Operation



Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

Figure 4.8 Summary of typical SIMD multimedia support for 256-bit-wide operations. Note that the IEEE 754-2008 floating-point standard added half-precision (16-bit) and quad-precision (128-bit) floating-point operations.

AVX instruction	Description
VADDPD	Add four packed double-precision operands
VSUBPD	Subtract four packed double-precision operands
VMULPD	Multiply four packed double-precision operands
VDIVPD	Divide four packed double-precision operands
VFMADDPD	Multiply and add four packed double-precision operands
VFMSUBPD	Multiply and subtract four packed double-precision operands
VCMPxx	Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, ...
VMOVAPD	Move aligned four packed double-precision operands
VBROADCASTSD	Broadcast one double-precision operand to four locations in a 256-bit register

Figure 4.9 AVX instructions for x86 architecture useful in double-precision floating-point programs. Packed-

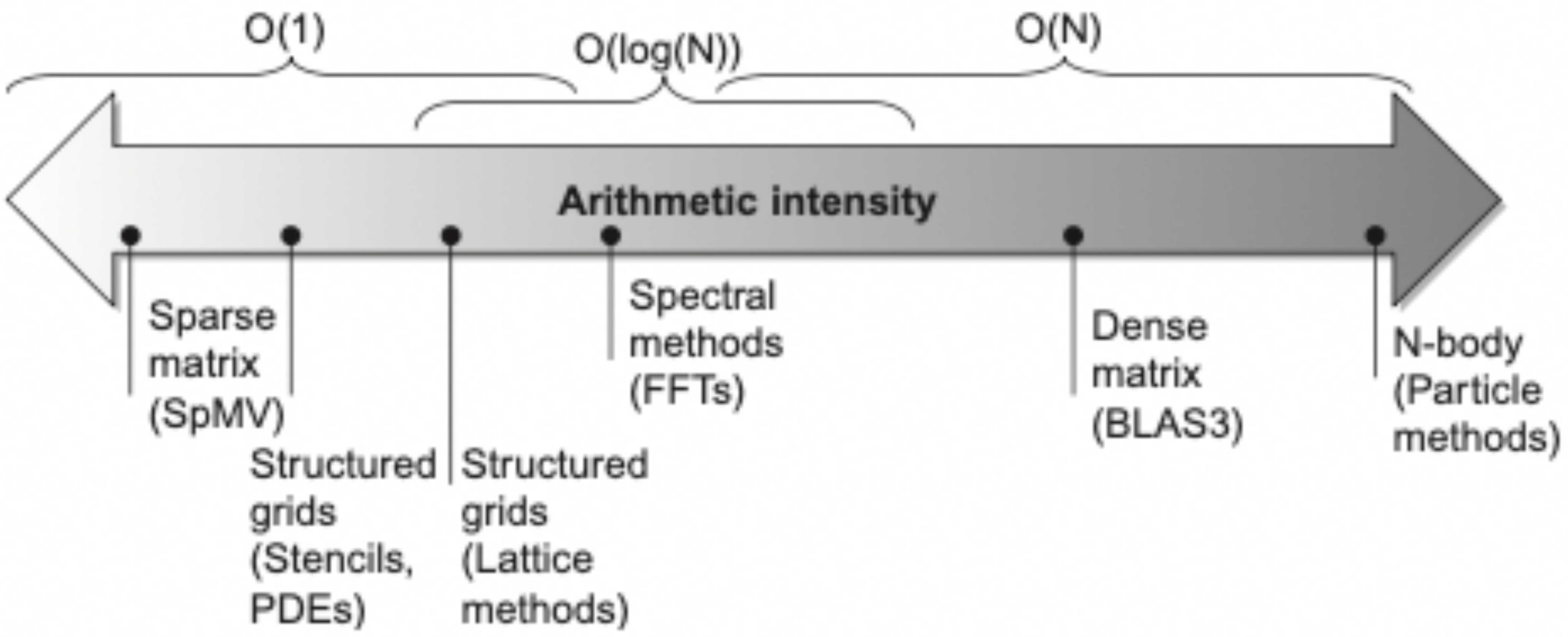


Figure 4.10 **Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory**

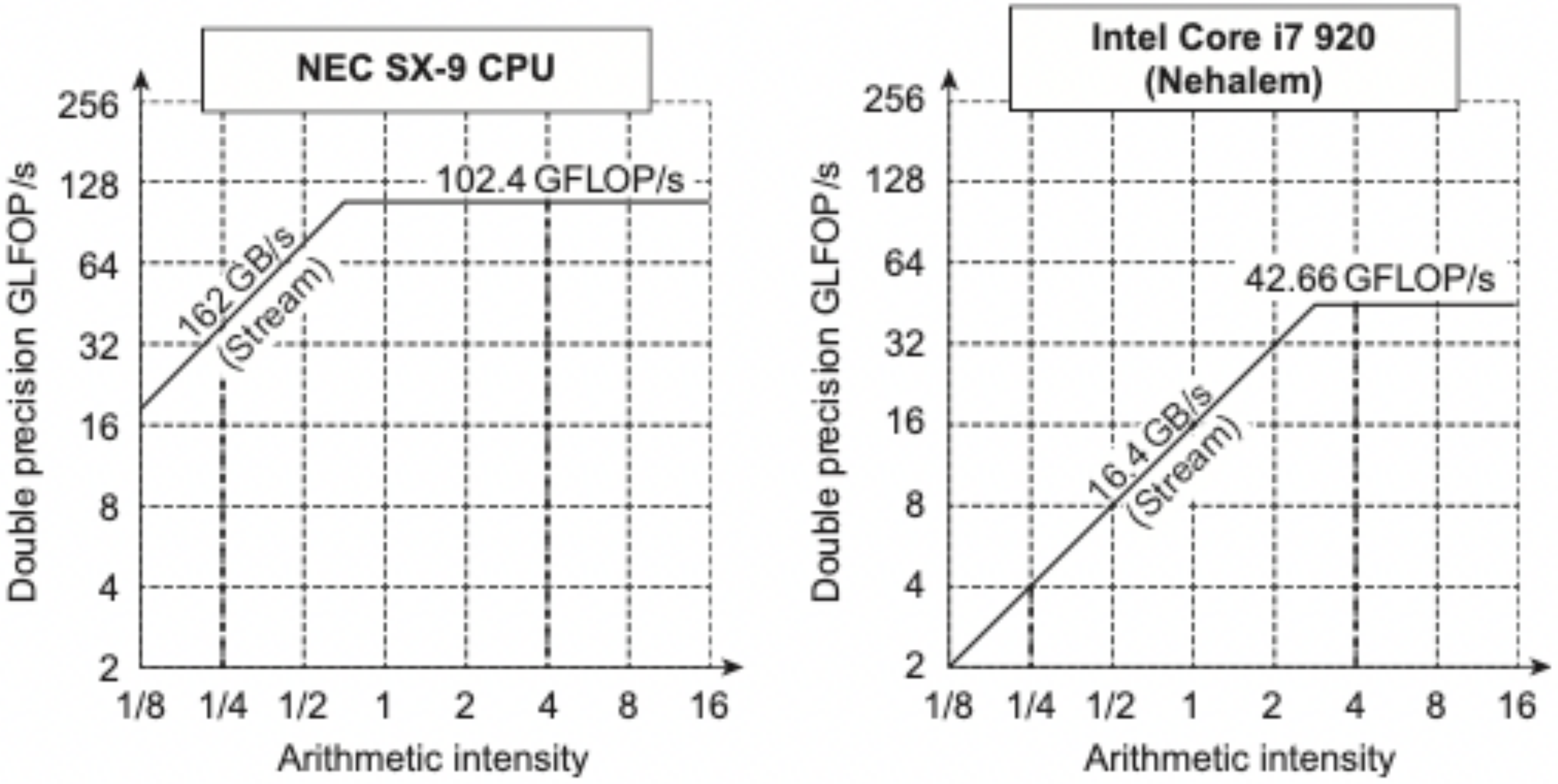


Figure 4.11 Roofline model for one NEC SX-9 vector processor on the left and the Intel Core i7 920 multicore computer with SIMD extensions on the right (Williams et al., 2009). This Roofline is for unit-stride memory accesses