中山大學
SUN YAT-SEN UNIVERSITY

数据级并行：向量体系结构

主讲教师：胡淼

中山大学计算机学院
2024 年 秋季

教学内容

第 4 章 高性能处理器的并行计算技术

4.2 数据级并行

part 1 向量体系结构

课程内容

- **参考材料：**
  - Computer Architecture, A Quantitative Approach, 6th Ed.
  - Chapter 4. “Data-Level Parallelism in Vector, SIMD, and GPU Architectures”
    - 1 Introduction
    - 2 Vector Architecture
    - 3 SIMD Instruction Set Extensions for Multimedia
    - 4 Graphics Processing Units
    - 5 Detecting and Enhancing Loop-Level Parallelism

1 Introduction

Flynn's Taxonomy

- **Single instruction stream, single data stream (SISD)**
- **Single instruction stream, multiple data streams (SIMD)**
  - Vector architectures
  - Multimedia SIMD instruction set extensions
  - Graphics Processor Units (GPUs)
- **Multiple instruction streams, single data stream (MISD)**
  - No commercial implementation
- **Multiple instruction streams, multiple data streams (MIMD)**
  - Tightly-coupled MIMD
  - Loosely-coupled MIMD

SIMD

- **SIMD architectures can exploit significant data-level parallelism for:**
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors

- **SIMD is more energy efficient than MIMD**
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices

- **SIMD allows programmer to continue to think sequentially**

2 Vector Architecture

Vector Architectures

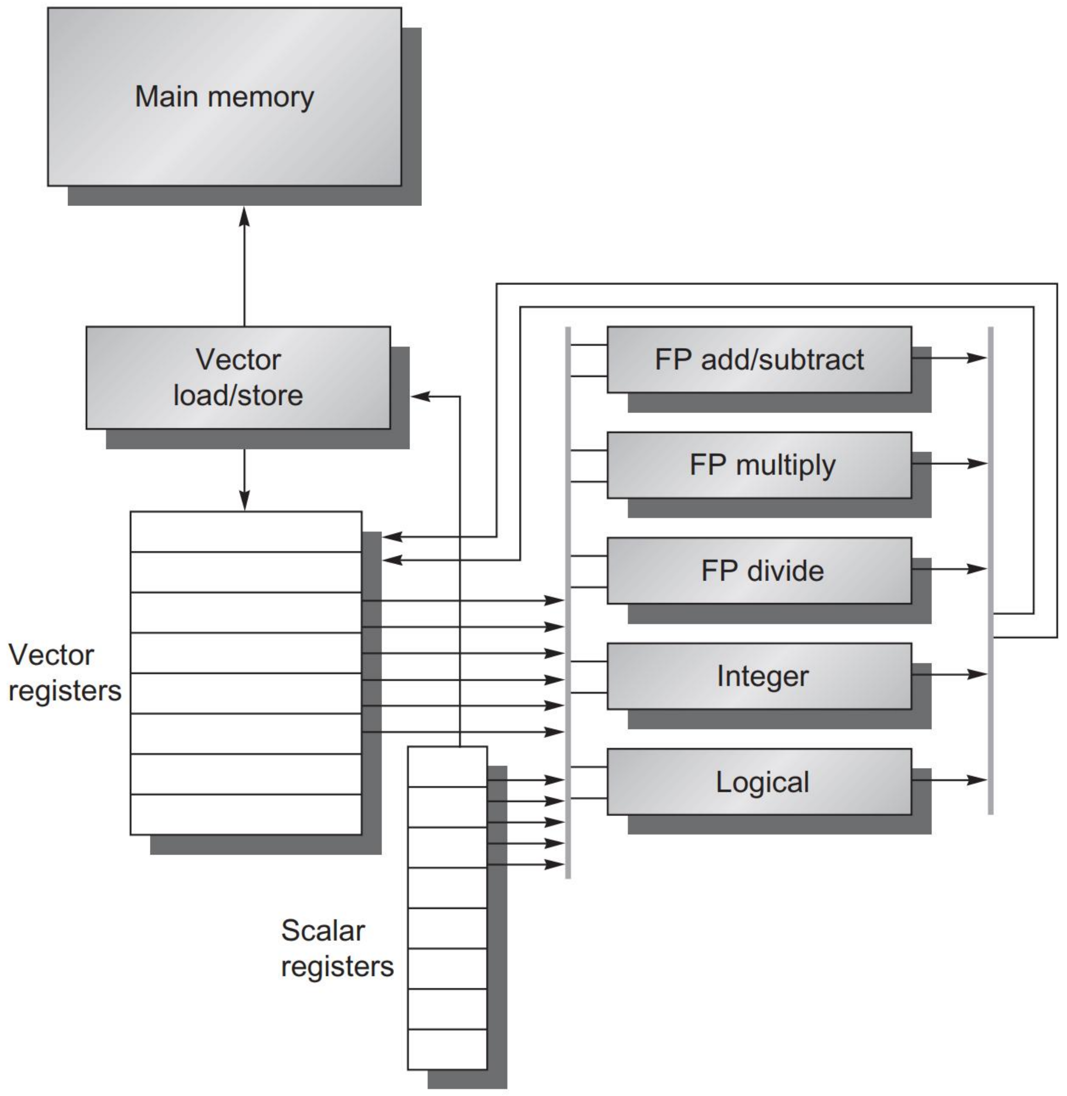
- **Basic idea:**
  - Read sets of data elements into "**vector registers**"
  - **Operate** on those registers
  - Disperse the results **back into memory**

- **Registers are controlled by compiler**
  - Used to hide memory latency
  - Leverage memory bandwidth

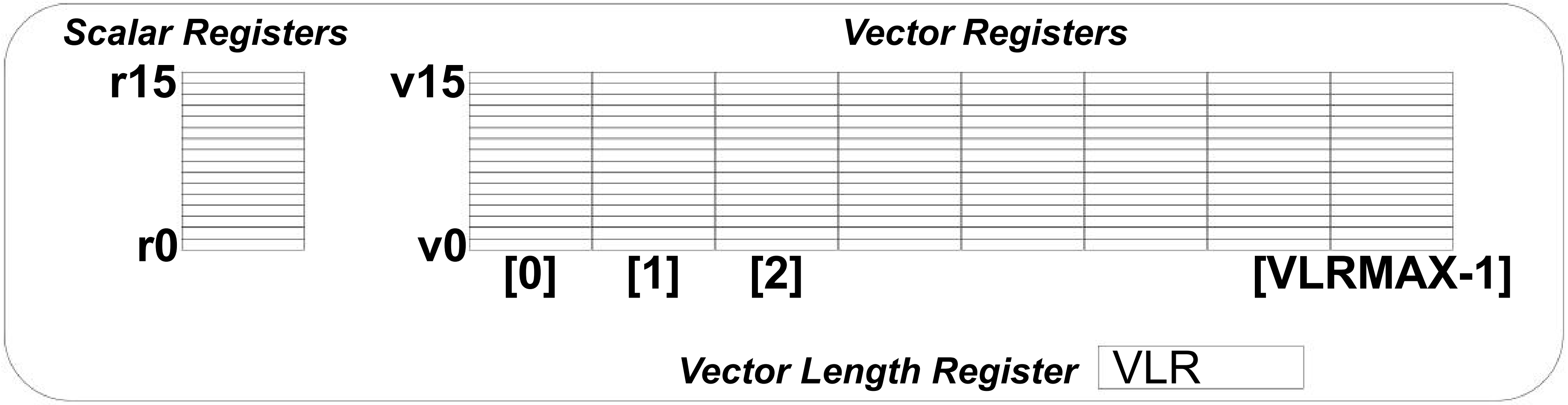
RV64V

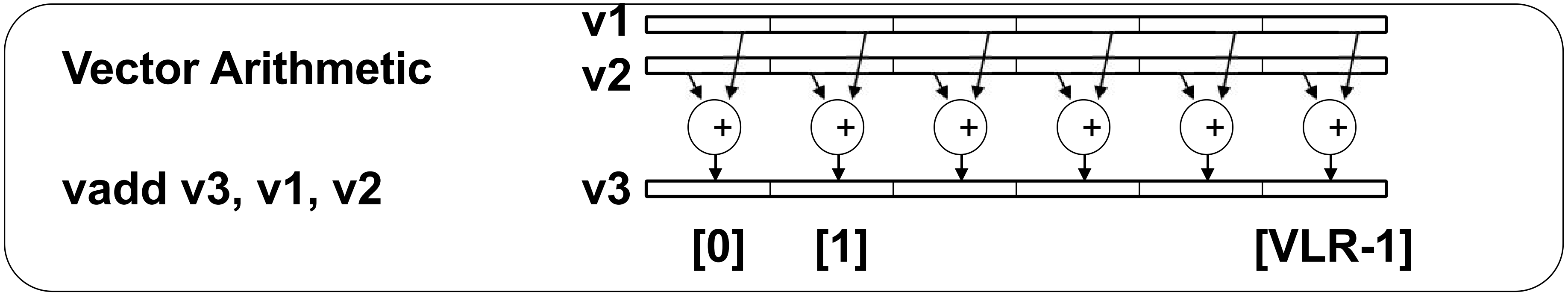
- **RV64V:** RISC-V based instructions + vector extension
  - 32 64-bit vector registers
    - Register file has 16 read ports and 8 write ports
  - Vector functional units
    - Fully pipelined
    - Data and control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Scalar registers
    - 31 general-purpose registers
    - 32 floating-point registers

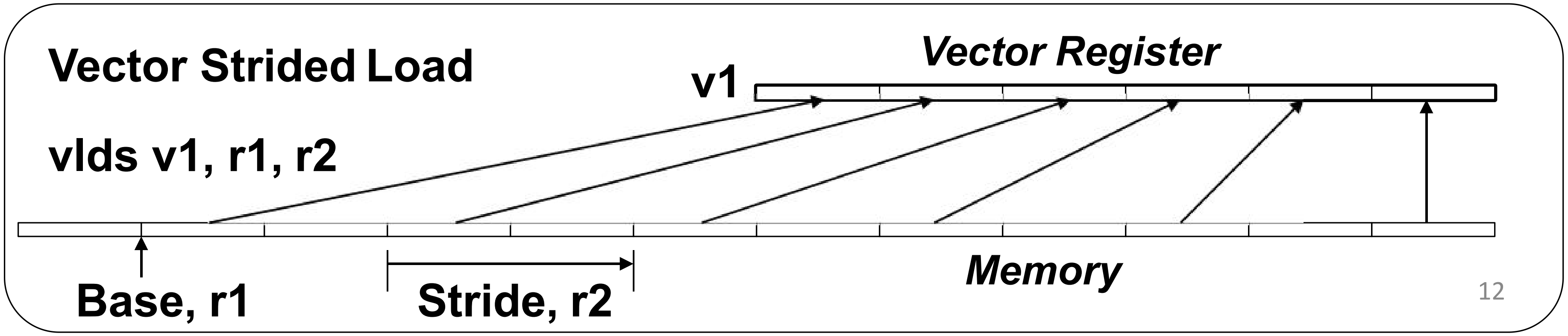
Basic Structure of RV64V



Vector Programming Model







RV64V Instructions

- **RV64V uses the suffix to indicate operand type**
  - **.vv**: two vector operands
  - **.vs** and **.sv**: vector and scalar operands
  - Example: **vsub.vv, vsub.vs**

How Vector Processors Work: An Example

- **Y = a * X + Y**
  - **DAXPY** for double precision a*X plus Y
- **RV64G 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
```

How Vector Processors Work: An Example

- **RV64V Code:**

```
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
```

Remarks

- RV64V vector instructions vs. RV64G scalar instructions
- In RV64G Code
  - Fadd.d must wait for fmul.d
  - fsd must wait for fadd.d
  - Lots of *pipeline stalls* are necessary for deeply pipelined architecture
- In RV64V Code
  - Stall once for the first vector element, subsequent elements will flow smoothly down the pipeline
  - Pipeline stalls are required only once per vector instruction, rather than once per vector element

- **Pipeline stall frequency on RV64G will be about 32 higher than it is on RV64V**

Vector Execution Time

- **Execution time depends on three factors:**
  - (1) Length of operand vectors
  - (2) Structural hazards among operations
  - (3) Data dependencies

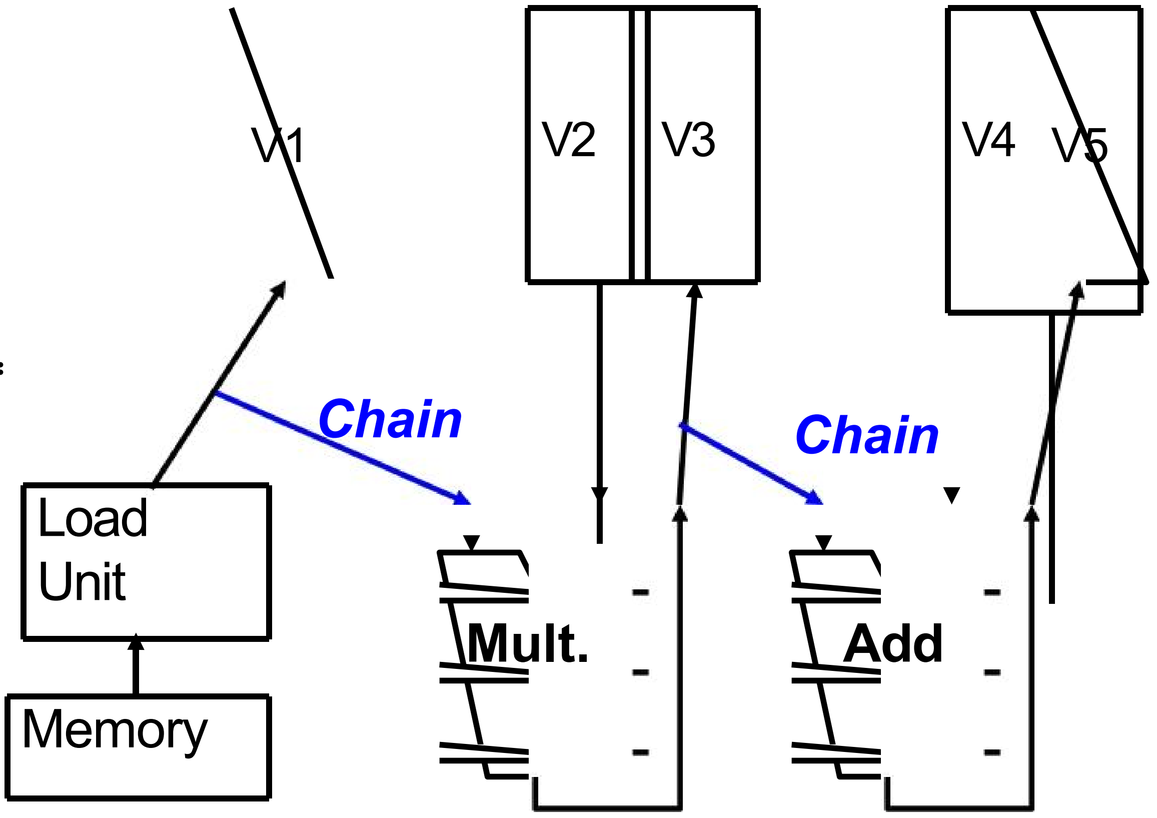
Convoy

- **Convoy (护航指令组)**
  - Set of vector instructions that could **potentially execute together**
- **The instructions in a convoy *must not* contain any structural hazards**
  - If such hazards were present, the instructions need to be initiated in different convoys
- **Sequences with read-after-write dependency hazards placed in same convoy via *chaining***
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available

Vector Chaining

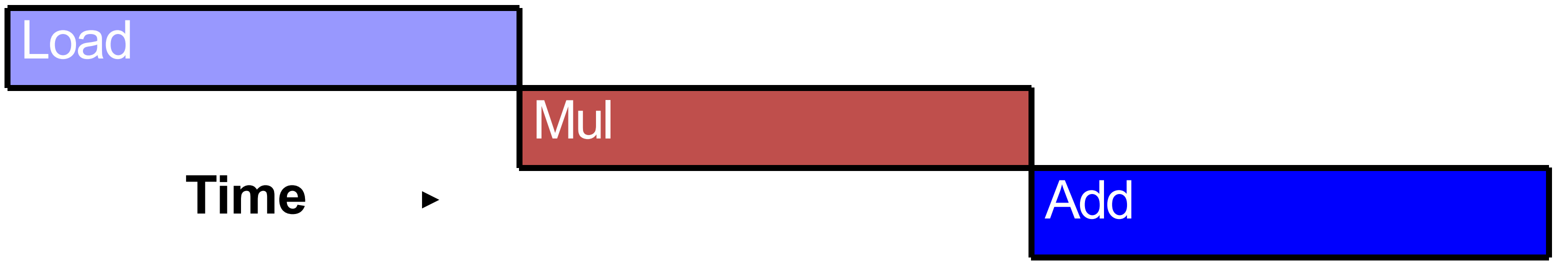
- Vector version of register bypassing
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available

```
LV   v1
MULV v3, v1,v2
ADDV v5, v3, v4
```

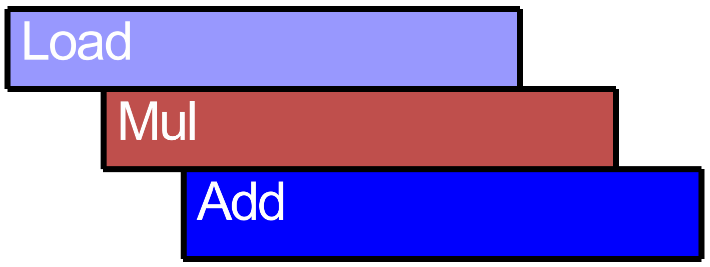


Advantages of Vector Chaining

- Without chaining, must wait for last element of result to be written before starting dependent instruction



- With chaining, can start dependent instruction as soon as first result appears



Example

```
vld     v0,x5       # Load vector X
vmul    v1,v0,f0    # Vector-scalar multiply
vld     v2,x6       # Load vector Y
vadd    v3,v1,v2    # Vector-vector add
vst     v3,x6       # Store the sum
```

Three Convoys:

```
1    vld    vmul
2    vld    vadd
3    vst
```

Chimes

- ***Chime:*** *unit of time to execute one convoy*
  - *m* convoys executes in *m* chimes
  - For a vector length of *n*, requires approximately *m* x *n* clock cycles

PS: a vector processor execute a single vector one element per clock cycle.

Improve the Performance of Vector Processor

Questions

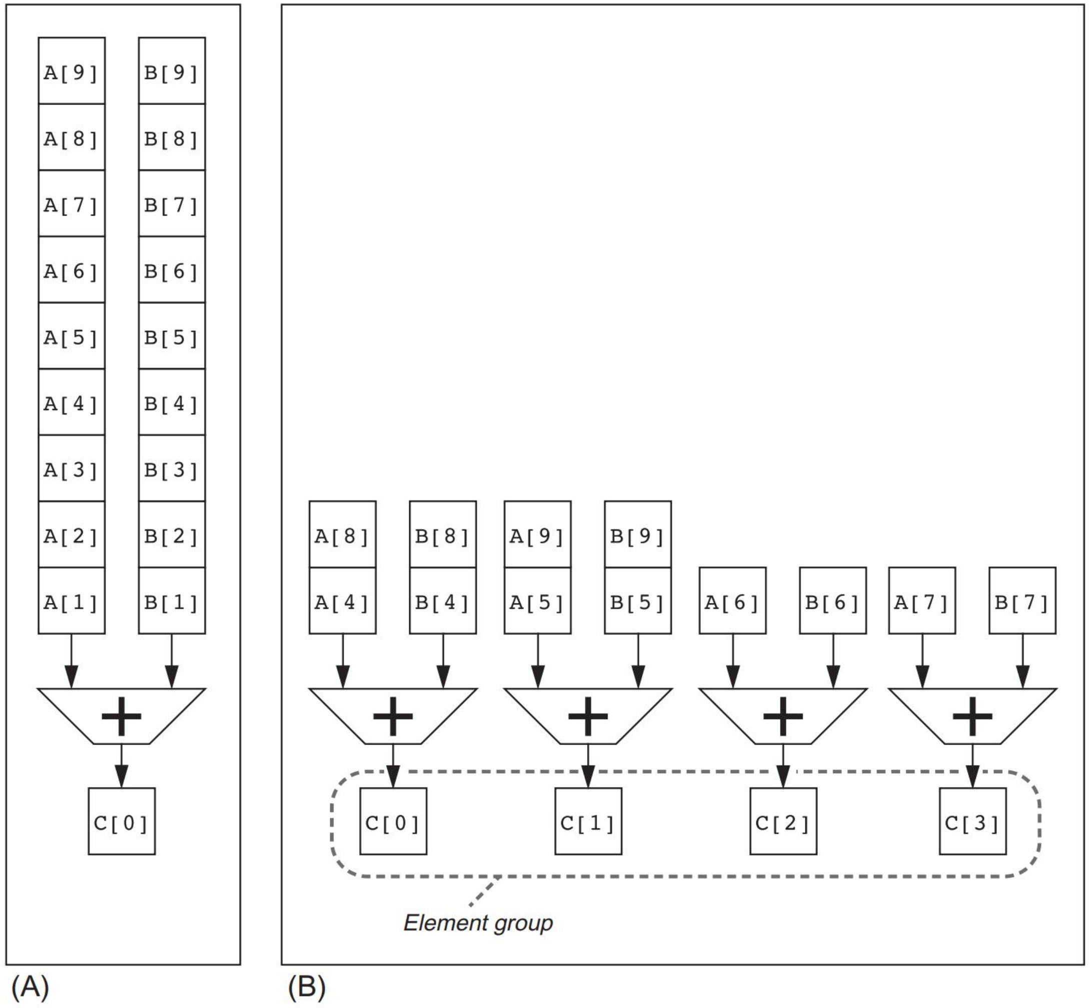
- **How can a vector processor execute a single vector faster than one element per clock cycle?**
  - Multiple elements per clock cycle improve performance
- **How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length (`mvl`)?**
  - Most application vectors don't match the architecture vector length
- **What happens when there is an IF statement inside the code to be vectorized?**
  - More code can vectorize if we can efficiently handle conditional statements

Questions

- **What does a vector processor need from the memory system?**
  - Without sufficient memory bandwidth, vector execution can be futile
- **How does a vector processor handle multiple dimensional matrices?**
  - This popular data structure must vectorize for vector architectures to do well
- **How does a vector processor handle sparse matrices?**
  - This popular data structure must vectorize also
- **How do you program a vector computer?**
  - Architectural innovations that are a mismatch to programming languages and their compilers may not get widespread use

Multiple Lanes

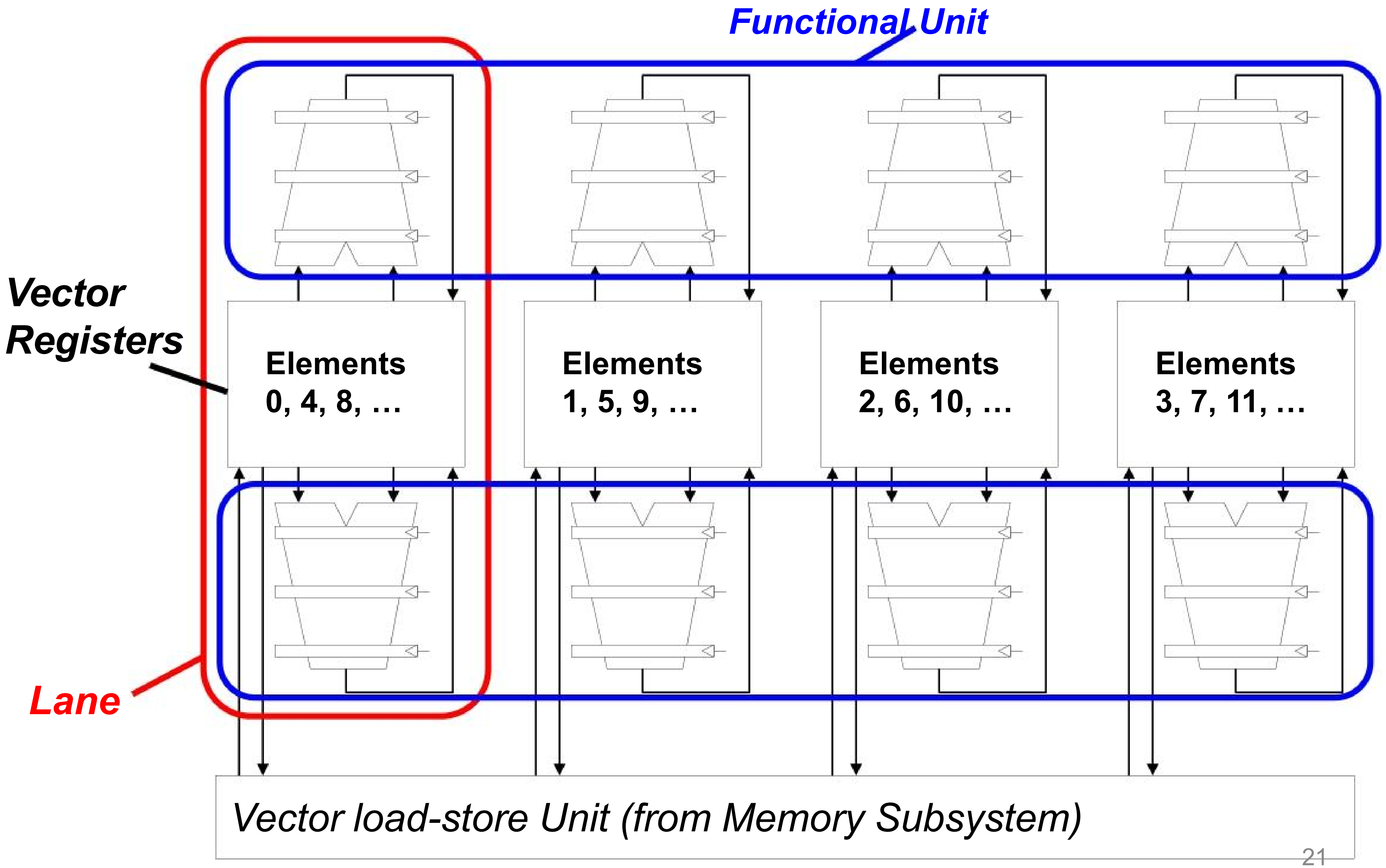
- Using multiple functional units to execute a vector add instruction



(A) on the left has a single add pipeline and can complete one addition per clock cycle

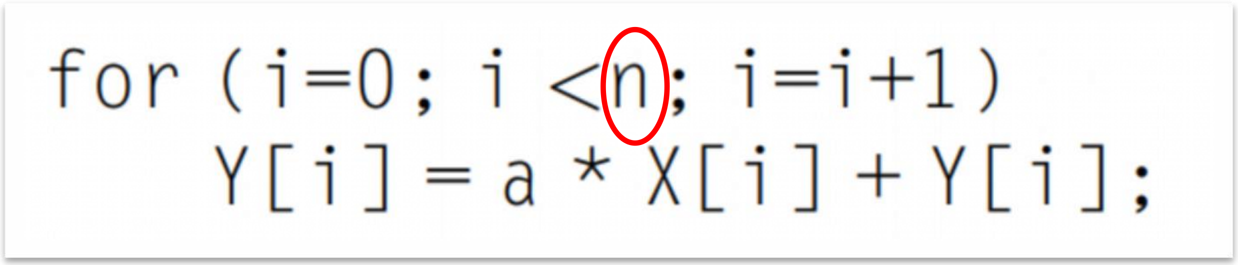
(B) on the right has four add pipelines and can complete four additions per clock cycle

Datapath for Multiple Lanes Structure



Vector Length Register

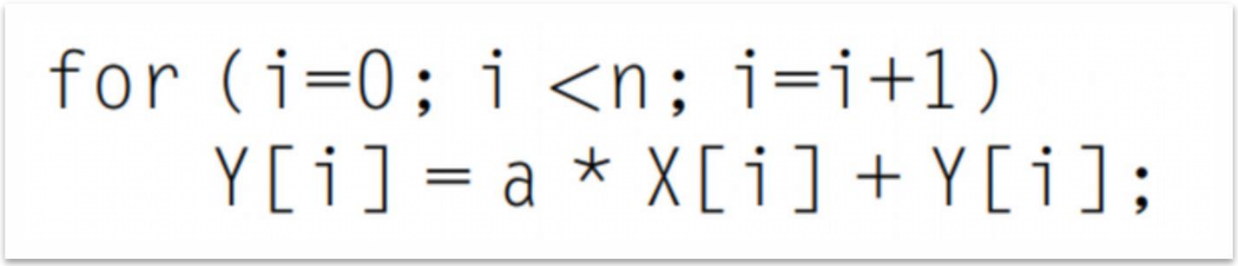
- **In a real program, the length of a particular vector operation is often unknown at compile time**
  - `n` might also be a parameter subject to change during execution



```
for (i=0; i <n; i=i+1)
    Y[i] = a * X[i] + Y[i];
```

Vector Length Register

- **The solution to these problems is to add a vector-length register (`vl`)**
  - `vl` controls the length of any vector operation
  - The value in the `vl` cannot be greater than the maximum vector length (`mvl`)



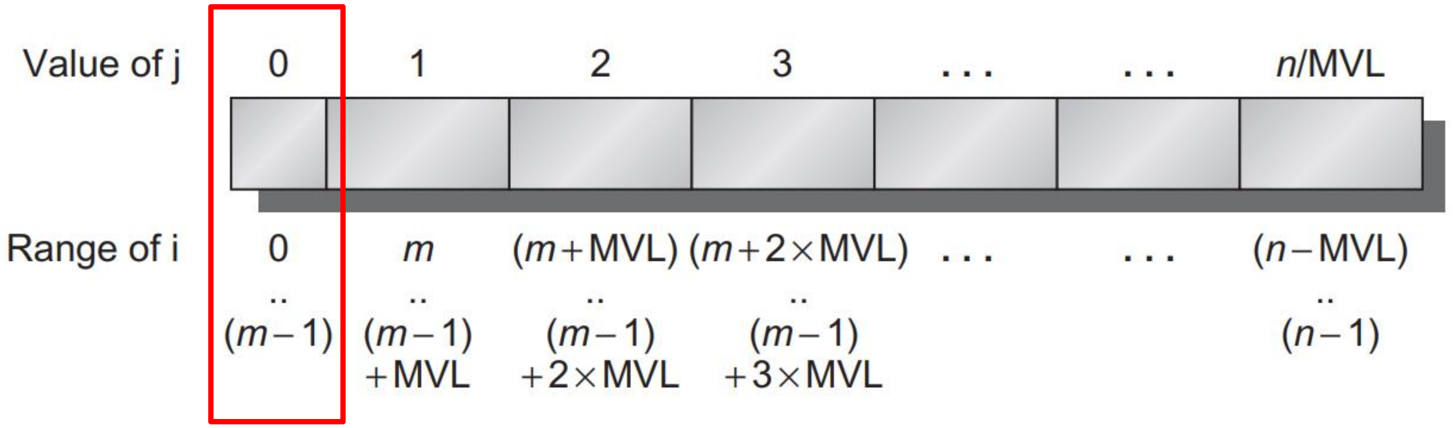
```
for (i=0; i <n; i=i+1)
    Y[i] = a * X[i] + Y[i];
```

Problem:

What if the value of n is not known at compile time and thus may be greater than mvl?

Strip Mining

- **Solution: Strip mining**
  - the generation of code such that each vector operation is done for a **`size <= mvl`**
  - One loop handles any number of iterations that is a multiple of the **`mvl`**
  - another loop that handles any remaining iterations and must be less than the **`mvl`**



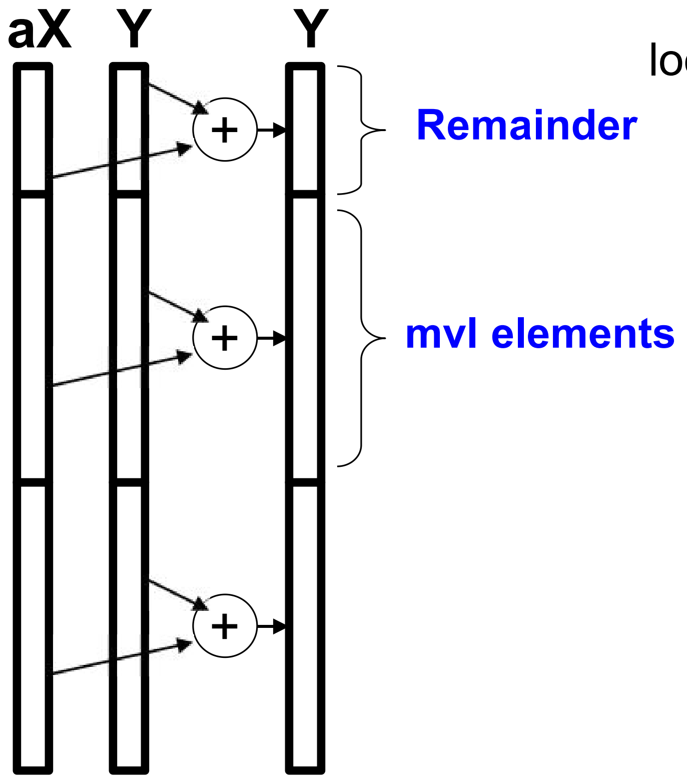
m = n % MVL

Vector Length Register

- **RISC-V has a better solution than a separate loop for strip mining**
  - The instruction **setvl** writes the **smaller** of the `mvl` and the loop variable `n` into `vl`
  - If `n` is greater than `mvl`, then the fastest the loop can compute is `mvl` values at time, so setvl sets `vl` to `mvl`
  - If `n` is smaller than `mvl`, it should compute only on the last `n` elements in this final iteration of the loop, so setvl sets `vl` to `n`

Vector Strip-Mining Example

```
for (i=0; i<n; i++)
    Y[i] = a * X[i] + Y[i];
```



```
      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}
```

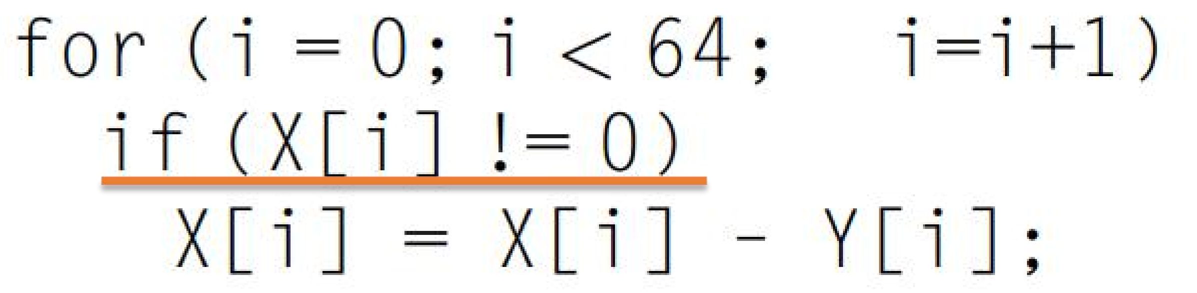
Vector Architecture

- From Amdahl's law, we know that the speedup on programs with low to moderate levels of vectorization will be very limited

- Two main reasons for lower levels of vectorization
  - the presence of ***conditionals (IF statements) inside loops***
  - the use of ***sparse matrices***

Vector Conditional Execution

- **Problem: Programs that contain IF statements in loops cannot be vectorized**
  - IF statements introduce control dependences into a loop
  - if the inner loop could be run for the iterations for which **`X[i]!=0`**, then the subtraction could be vectorized



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

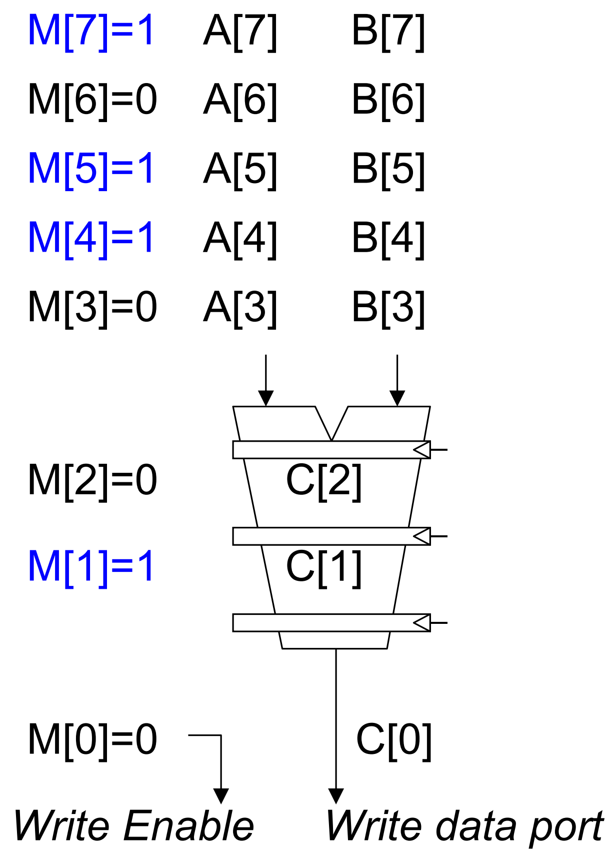
Vector Mask Registers

- **Vector-mask control**
  - **Predicate registers** hold the mask and essentially provide conditional execution of each element operation in a vector instruction
  - These registers use a **Boolean vector** to control the execution of a vector instruction
    - When the predicate register p0 is set, all following vector instructions operate only on the vector elements whose entries in the predicate register are 1

Masked Vector Instructions

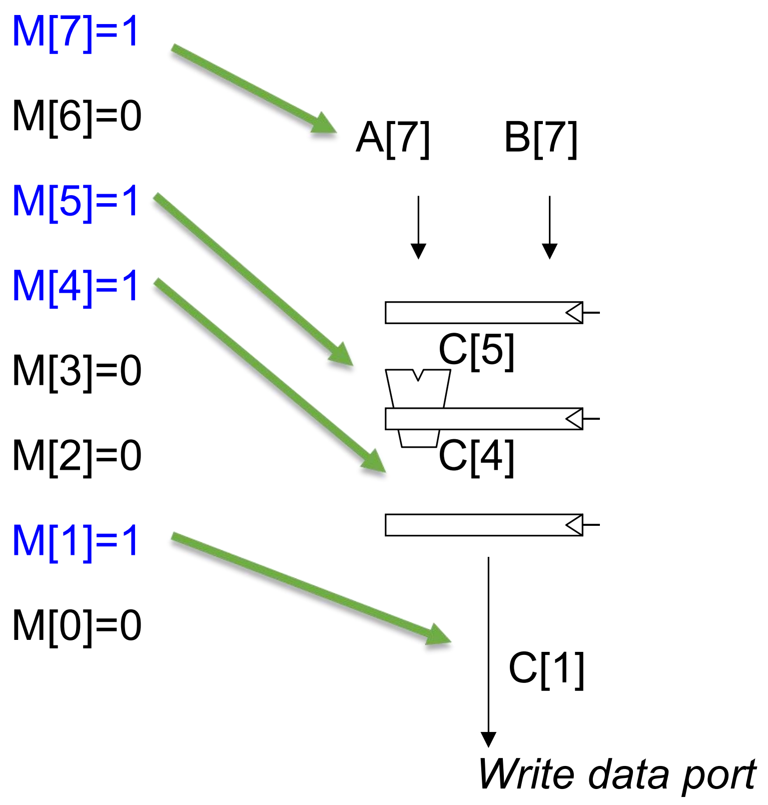
Simple Implementation

– execute all N operations, turn off result writeback according to mask



Density-Time Implementation

– scan mask vector and only execute elements with non-zero masks



Vector Mask Registers

- Use predicate register to "disable" elements:

```
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
```

transform an IF statement into a straight-line code sequence

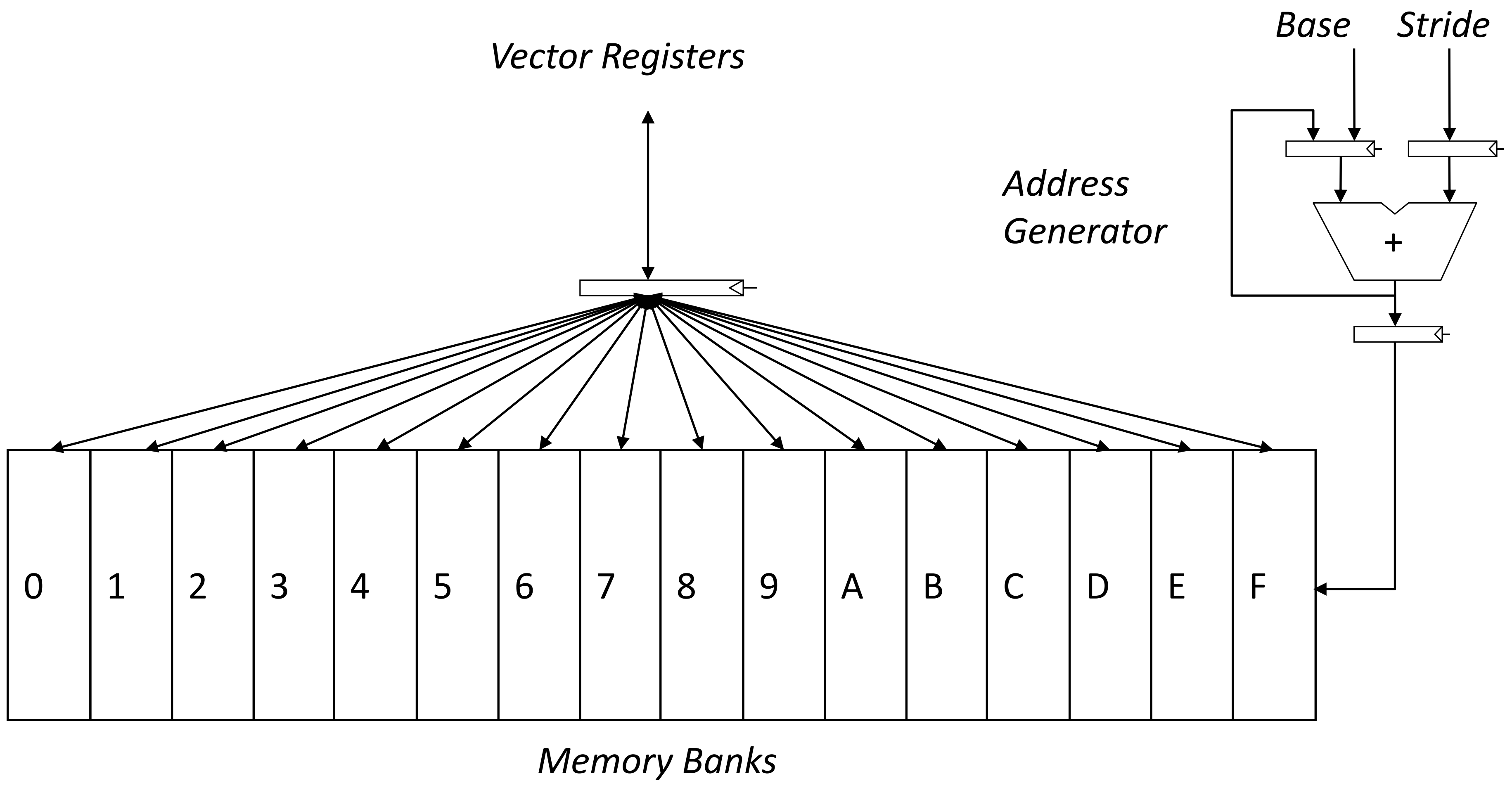
Vector Mask Registers

- **Vector-mask control**
  - Using a vector-mask register does have **overhead**
    - vector instructions executed with a vector mask still require execution time, even for the elements where the mask is zero
    - despite a significant number of zeros in the mask, using vector-mask control may still be significantly faster than using scalar mode

Memory Banks

- **Memory system must be designed to support high bandwidth for vector loads and stores**
- **Spread accesses across multiple banks**
  - Control bank addresses independently
  - Load or store non sequential words (need independent bank addressing)
  - Support multiple vector processors sharing the same memory

Vector Memory Subsystem

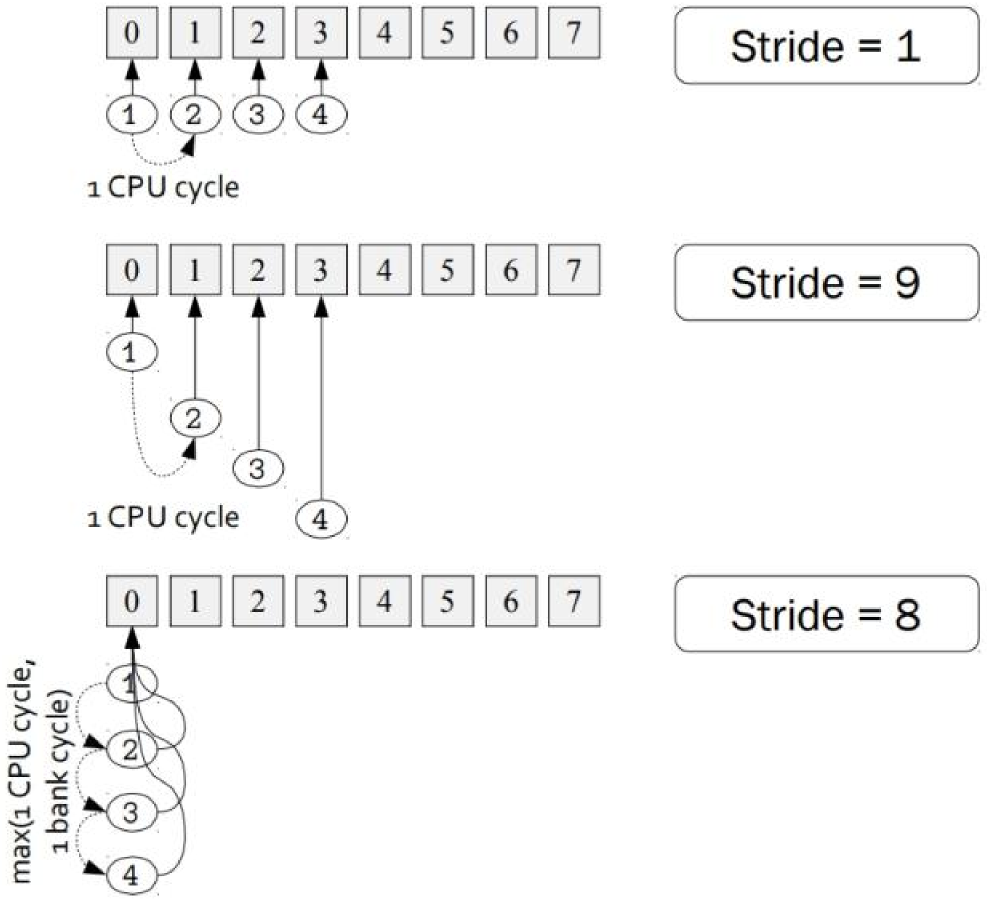


Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- Bank busy time: Cycles between accesses to same bank

Problem of Memory Bank Conflict

- **Bank busy time**: minimum period of time between successive accesses to the same bank
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:



Example

- Cray T90 (Cray T932) has 32 processors, each generating 4 loads and 2 stores/cycle
- Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - Each SRAM bank is busy for 15/2.167 = 6.92 clock cycles, which we round up to 7 processor clock cycles
- How many memory banks needed?
  - 32 x (4 + 2) x 7 = 1344 banks

The Cray T932 actually has 1024 memory banks

Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
  - The first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
  - Cray-1 ('76) was first vector register machine

Example Source Code

```
for (i=0; i<N; i++)
{
   C[i] = A[i] + B[i];
   D[i] = A[i] - B[i];
}
```

Vector Memory-Memory Code

```
ADDV C, A, B
SUBV D, A, B
```

Vector Register Code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

Vector Memory-Memory vs. Vector Register Machines

- **Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?**
  - All operands must be read in and out of memory
- **VMMAs make if difficult to overlap execution of multiple vector operations, why?**
  - Must check dependencies on memory addresses
- **VMMAs incur greater startup latency**

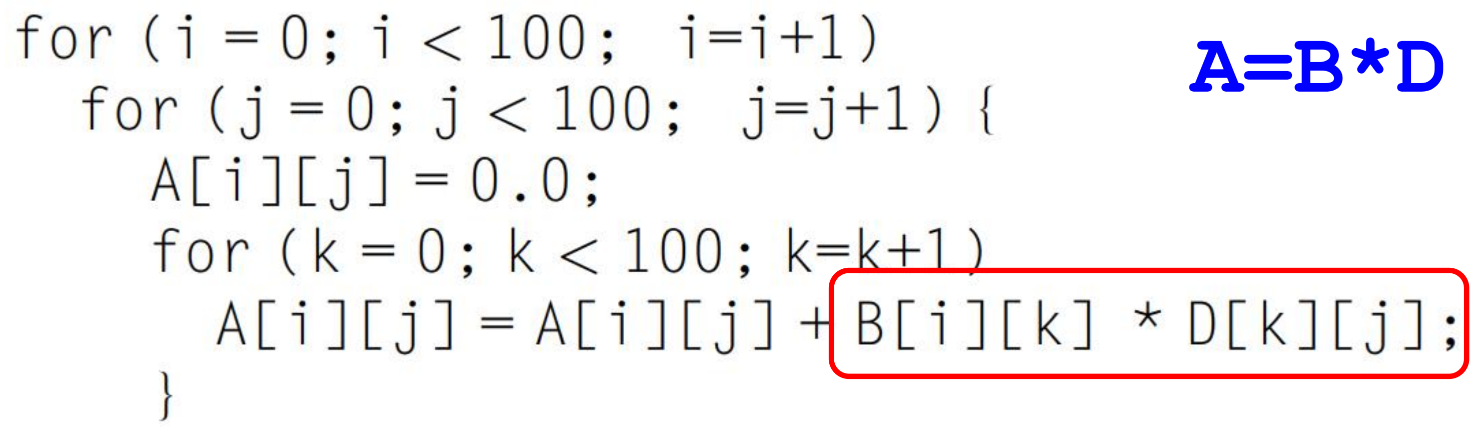
Vector Memory-Memory vs. Vector Register Machines

All major vector machines since Cray-1 have had vector register architectures

(we ignore vector memory-memory from now on)

Handling Multidimensional Arrays in Vector Architectures

- The position in memory of adjacent elements in a vector may **not be sequential**
- We could **vectorize** the multiplication of each row of B with each column of D
  - we must consider how to address adjacent elements in B and adjacent elements in D

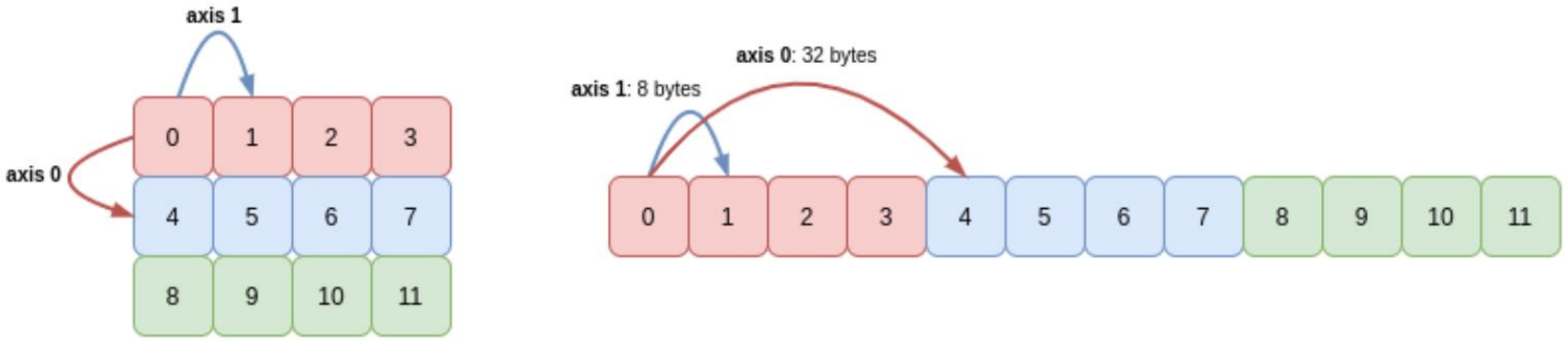


```
for (i = 0; i < 100;  i=i+1)
  for (j = 0; j < 100;  j=j+1) {
    A[i][j] = 0.0;
    for (k = 0; k < 100; k=k+1)
      A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

A=B*D

Stride (步幅)

- **Stride**: the distance separating elements to be gathered into a single vector register



Stride

- **Stride:** the distance separating elements to be gathered into a single vector register
  - matrix **D** has a stride of 100 double words (800 bytes)
  - matrix **B** has a stride of 1 double word (8 bytes)

```
for (i = 0; i < 100;  i=i+1)
   for (j = 0; j < 100;  j=j+1) {
      A[i][j] = 0.0;
      for (k = 0; k < 100; k=k+1)
         A[i][j] = A[i][j] + B[i][k] * D[k][j];
      }
```

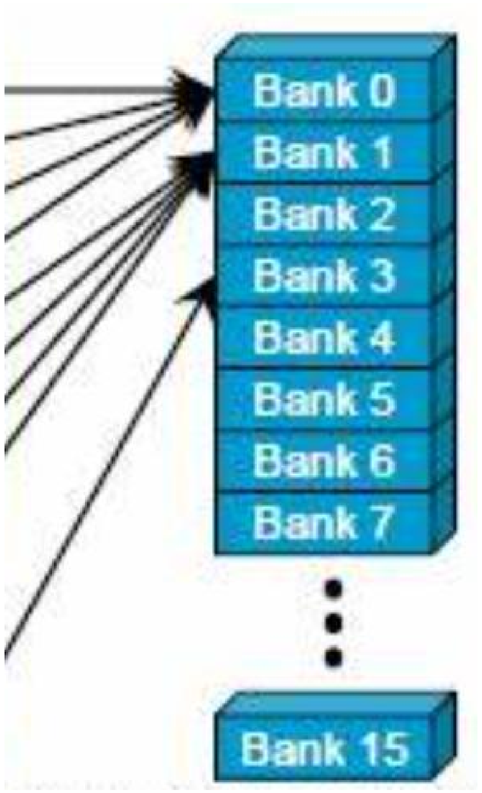
When an array is allocated memory, it is linearized and must be laid out in row-major order (as in C)

Stride

- **Non-unit stride：** strides greater than one
  - Once a vector is loaded into a vector register, it acts as if it had logically adjacent elements
- **A vector processor can handle non-unit strides using only vector load and vector store operations with stride capability**
  - The ability to access **nonsequential memory locations** and to **reshape them into a dense structure** is one of the major advantages of a vector architecture

Stride

- **Non-unit stride**
  - It is possible to request accesses from the same bank frequently
- **When multiple accesses contend for a bank, a memory bank conflict occurs**



- **Example**:
  - Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles.
  - How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?
- **Answer**
  - For a stride of 1
    - Because #banks is larger than the bank busy time, the load will take 12 + 64 = 76 clock cycles, or 1.2 clock cycles/element
  - For a stride of 32
    - Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock cycle bank busy time
    - The total time will be 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clock cycles/element

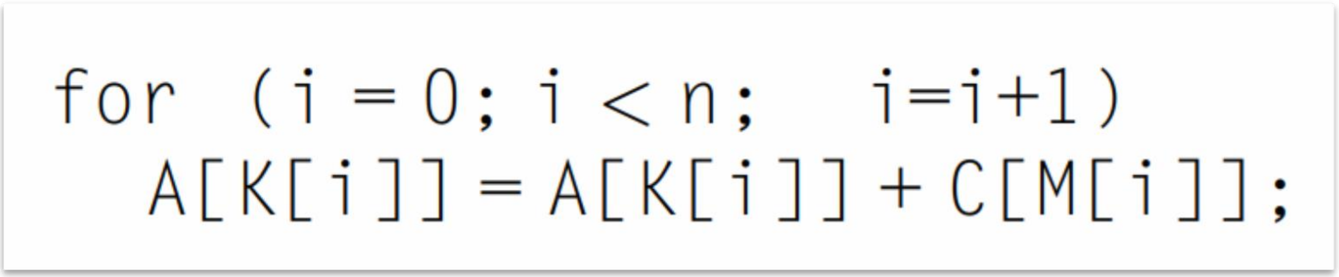
Handling Sparse Matrices in Vector Architectures

- **Sparse matrices are common**

$$\begin{bmatrix} 0 & 9 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Handling Sparse Matrices in Vector Architectures

- In a sparse matrix, the elements of a vector are usually stored in some compacted form and then accessed indirectly

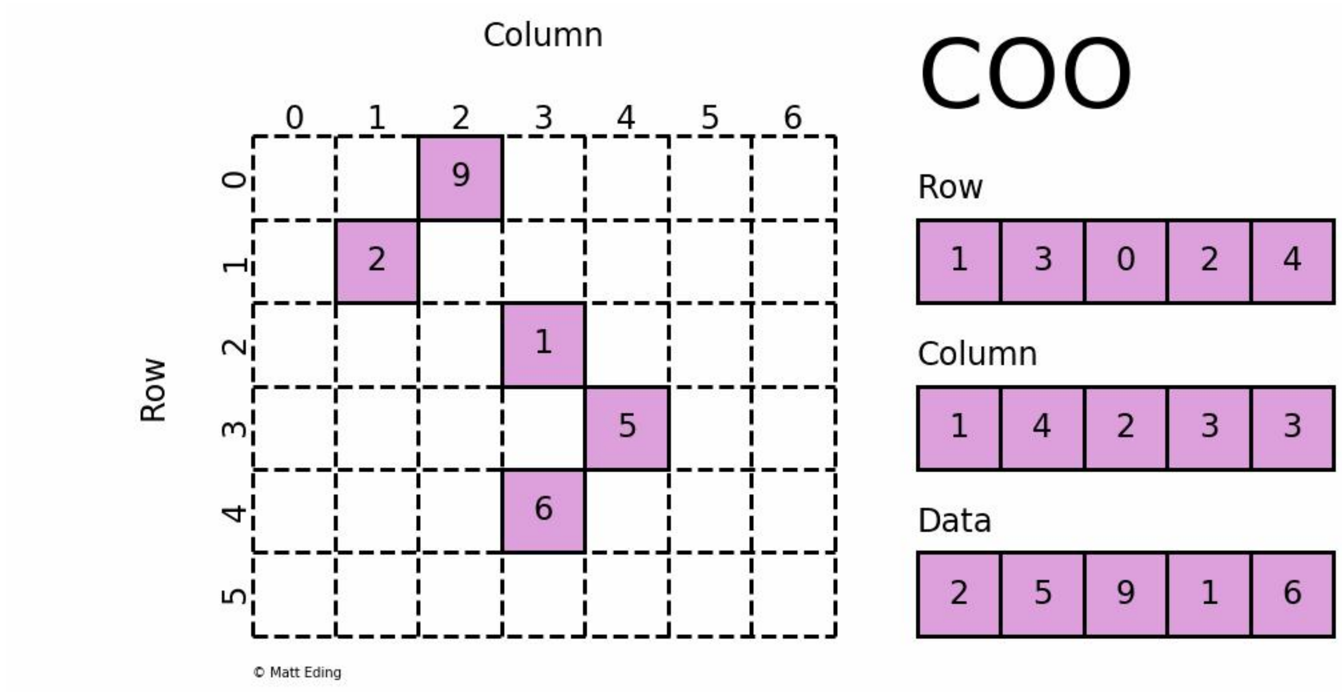


```
for (i = 0; i < n;   i=i+1)
   A[K[i]] = A[K[i]] + C[M[i]];
```

This code implements a sparse vector sum on the arrays A and C, using index vectors K and M to designate the nonzero elements of A and C

Handling Sparse Matrices in Vector Architectures

- In a sparse matrix, the elements of a vector are usually stored in some compacted form and then accessed indirectly



Coordinate list (COO) is a fast format for constructing sparse matrices

Gather-Scatter

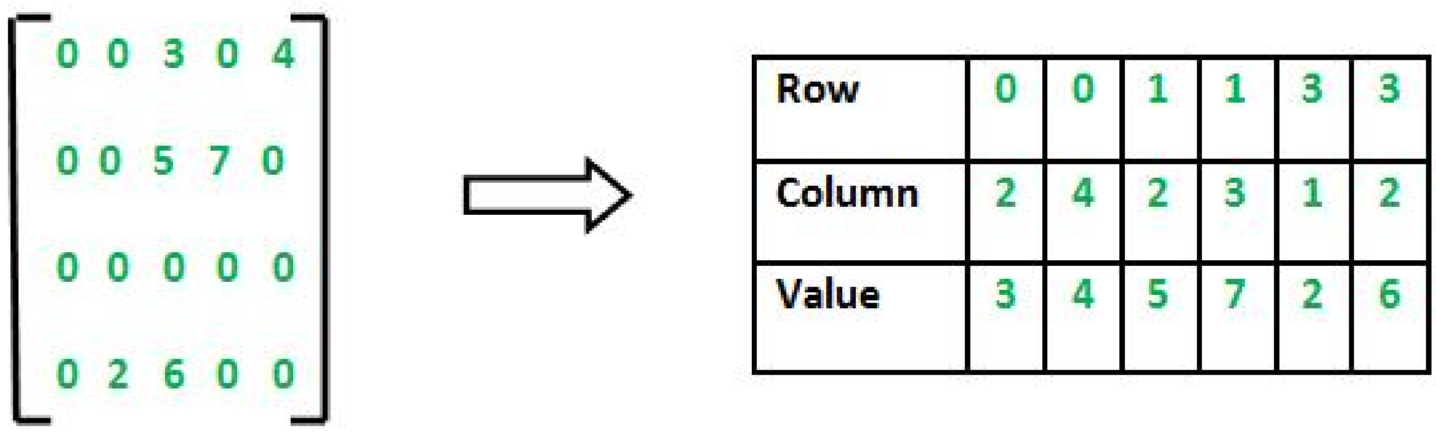
- **The primary mechanism for supporting sparse matrices is gather-scatter operations using index vectors**
  - **Goal**: support moving between a compressed representation (i.e., zeros are not included) and normal representation (i.e., the zeros are included) of a sparse matrix

Gather-Scatter

- **A gather operation takes an index vector and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector**
  - The result is a dense vector in a vector register
- **After these elements are operated on in a dense form, the sparse vector can be stored in an expanded form by a scatter store, using the same index vector**
- **Hardware support for such operations is called gather-scatter**

Gather-Scatter

- **The primary mechanism for supporting sparse matrices is gather-scatter operations using index vectors**
  - Support moving between a compressed representation (i.e., zeros are not included) and normal representation (i.e., zeros are included) of a sparse matrix

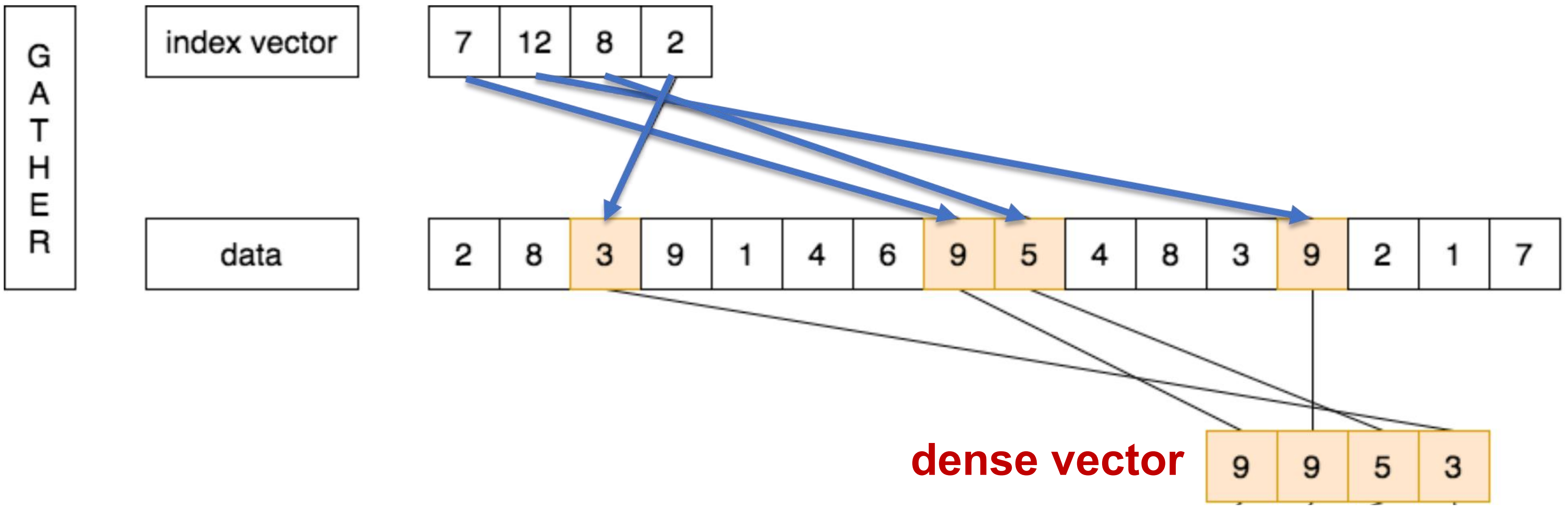


$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

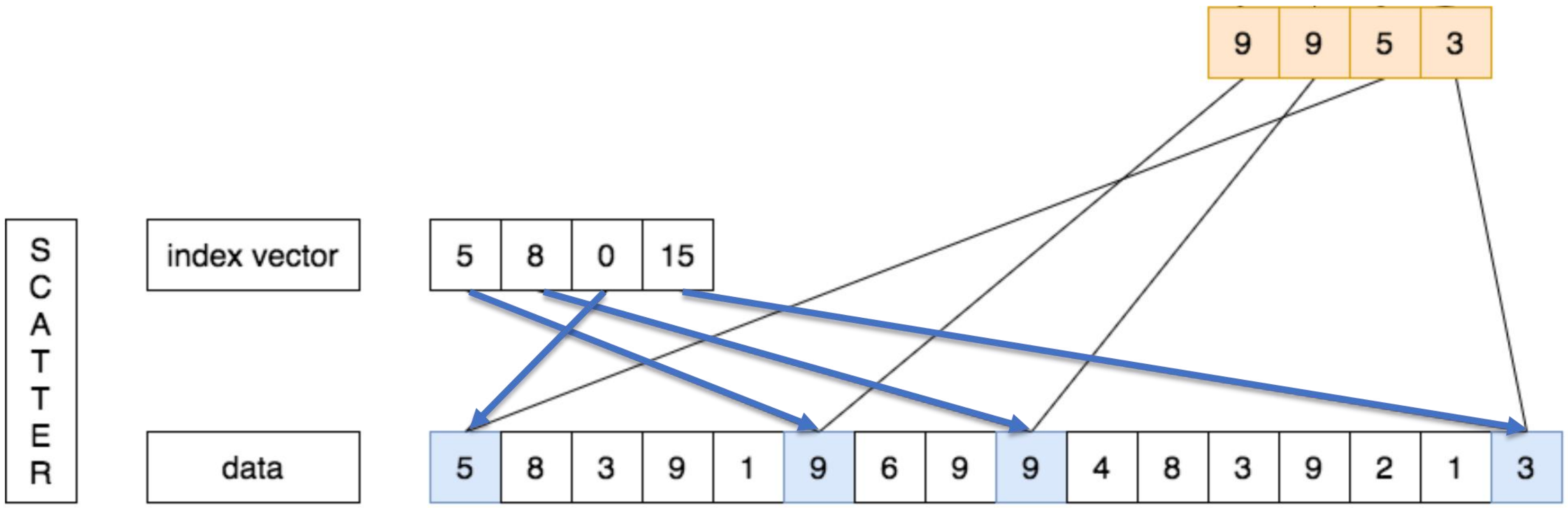
Gather

- **A gather operation takes an index vector and fetches the vector**
  - whose elements are at the addresses given by adding a base address to the offsets given in the index vector
  - The result is a **dense vector** in a vector register



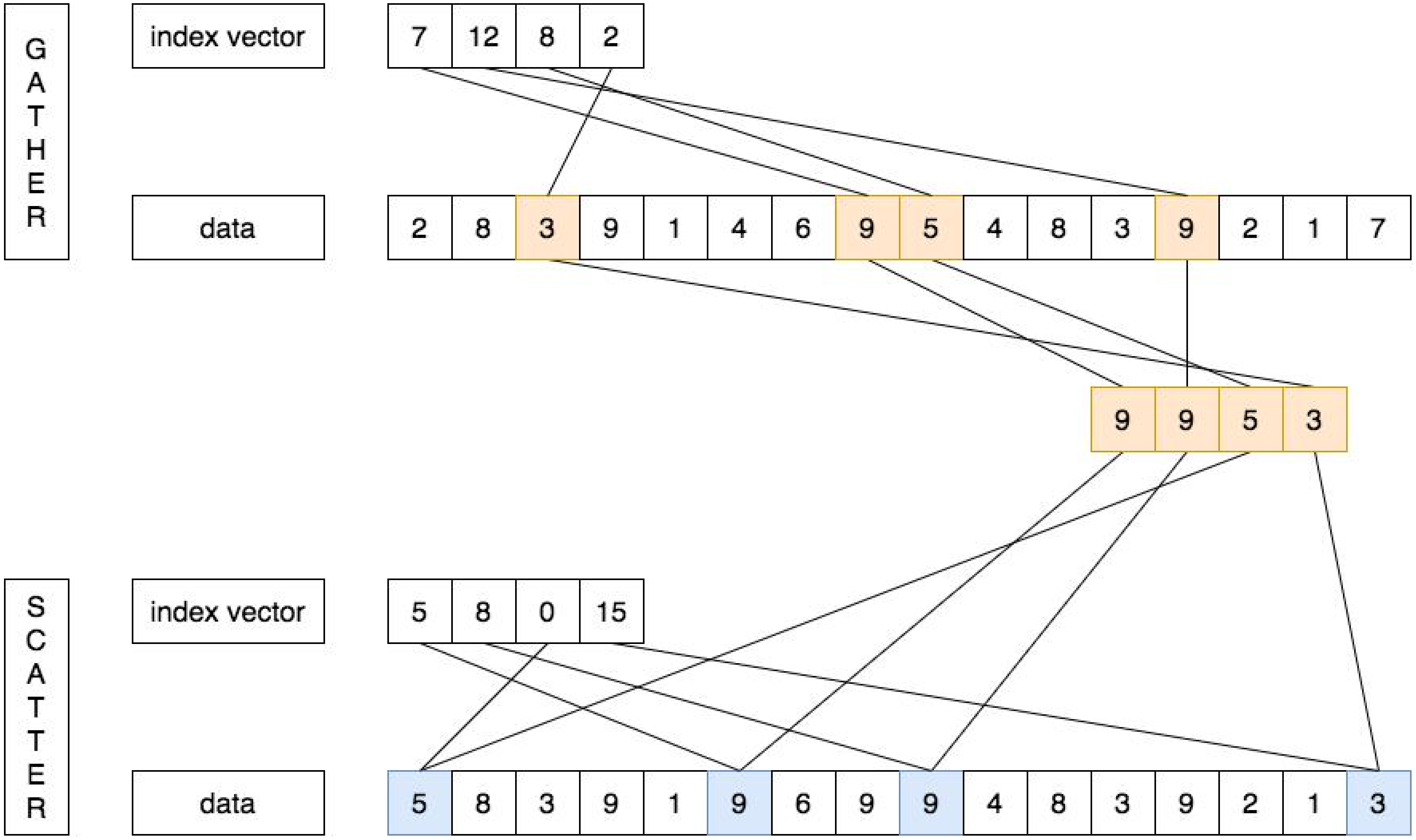
Scatter

- **Sparse vector can be stored in an expanded form by a scatter store**
  - using the same index vector



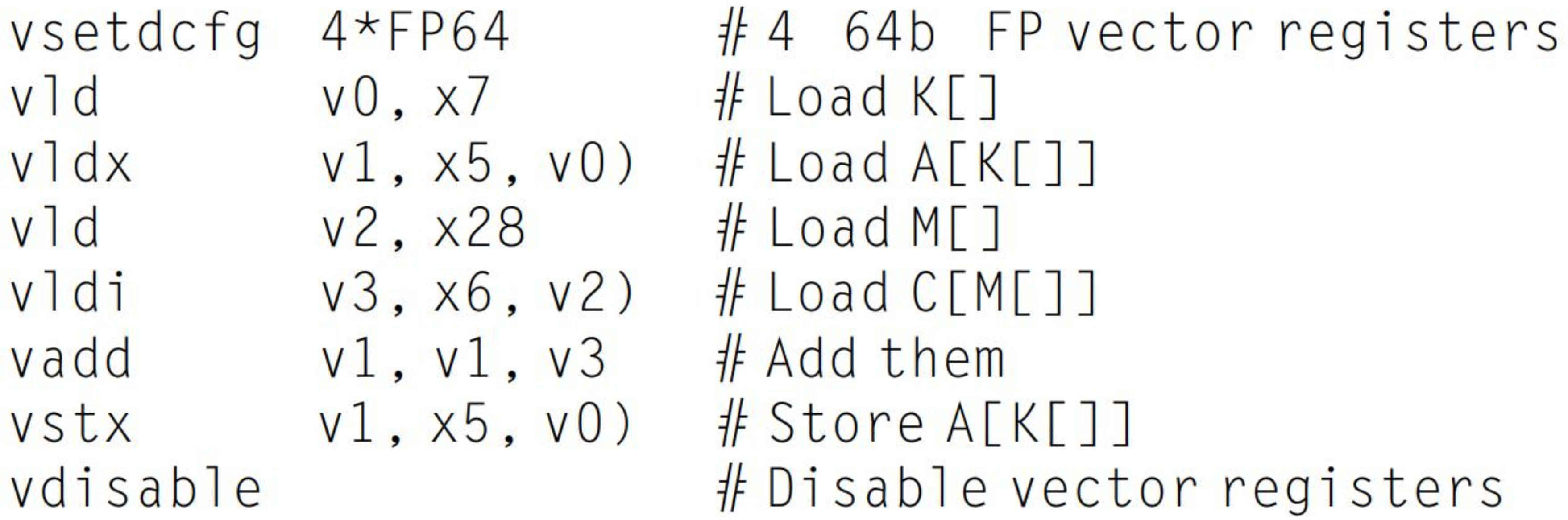
Gather-Scatter

- Hardware support for such operations is called **gather-scatter**



Gather-Scatter Example

- `vldi` (load vector indexed or **gather**)
- `vsti` (store vector indexed or **scatter**)



```
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
```

x5, x6, x7, and x28 contain the starting addresses of the vectors

Gather-Scatter Example

- for (i = 0; i < n; i=i+1)

      A[K[i]] = A[K[i]] + C[M[i]];

- Use index vector:

```
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
```

x5, x6, x7, and x28 contain the starting addresses of the vectors

Programming Vector Architectures

- Compilers can tell programmers at compile time whether a section of code will vectorize or not
- Programmers can provide hints to compiler

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Summary of Vector Architecture

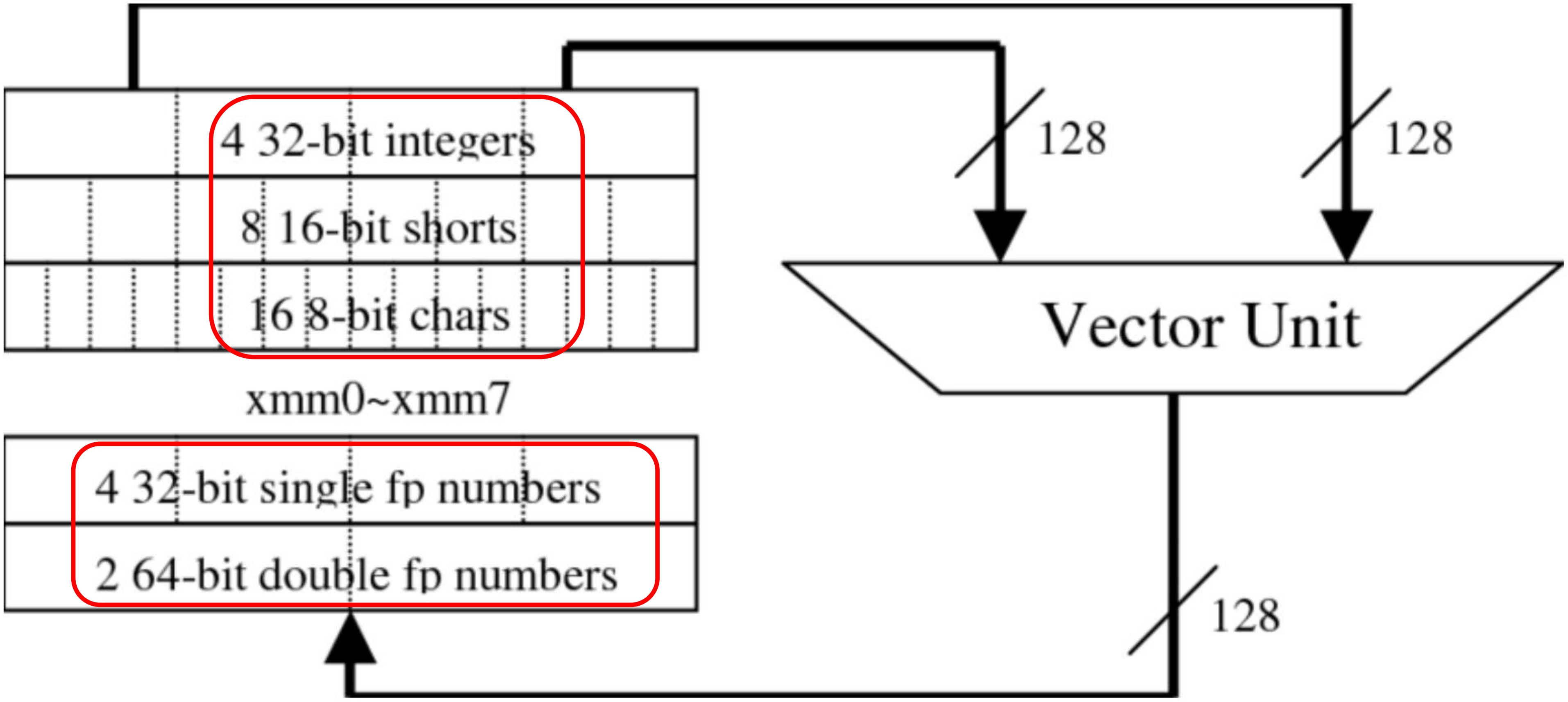
- **Vector is alternative model for exploiting ILP**
- **If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than out-of-order machines**
- **Design issues include:**
  - number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- **Fundamental design issue is memory bandwidth**

3 SIMD Instruction Set Extensions for Multimedia

SIMD Extensions

- **Many media applications operate on narrower data types than the 32-bit word size**
  - Graphics: 8-bit color
  - Audio samples: 8-16 bits
- **By partitioning the carry chains within a 256-bit adder, a processor could perform simultaneous operations on short vectors**
  - 32 8-bit operands, 16 16-bit operands, 8 32-bit operands

Example: 128-bit adder

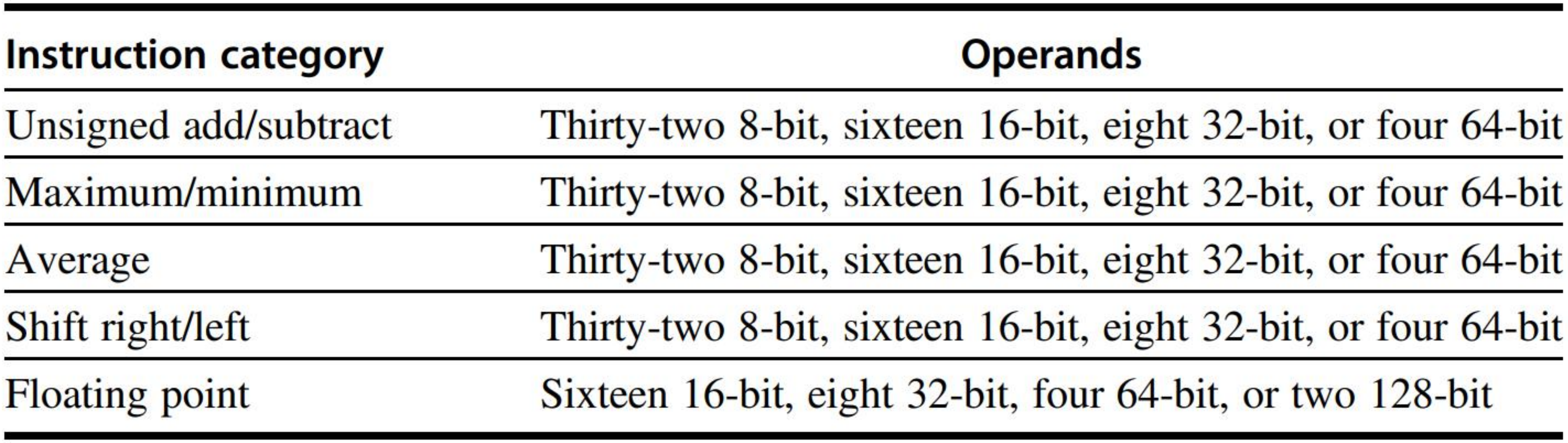


SIMD Extensions vs. Vector Arch.

- **Limitations, compared to vector instructions:**
  - **Fix # of data operands** encoded into op code
    - led to the addition of hundreds of instructions
  - **No sophisticated addressing modes**
    - Not support strided, scatter-gather
  - **No mask registers**
    - Not support conditional branching

Typical SIMD multimedia support

- Summary of typical SIMD multimedia support for 256-bit-wide operations



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

SIMD Implementations

- **Intel MMX (1996)**
  - Eight 8-bit integer ops or four 16-bit integer ops
- **Streaming SIMD Extensions (SSE) (1999)**
  - Eight 16-bit integer ops
  - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
- **Advanced Vector Extensions (2010)**
  - Four 64-bit integer/fp ops
- **AVX-512 (2017)**
  - Eight 64-bit integer/fp ops

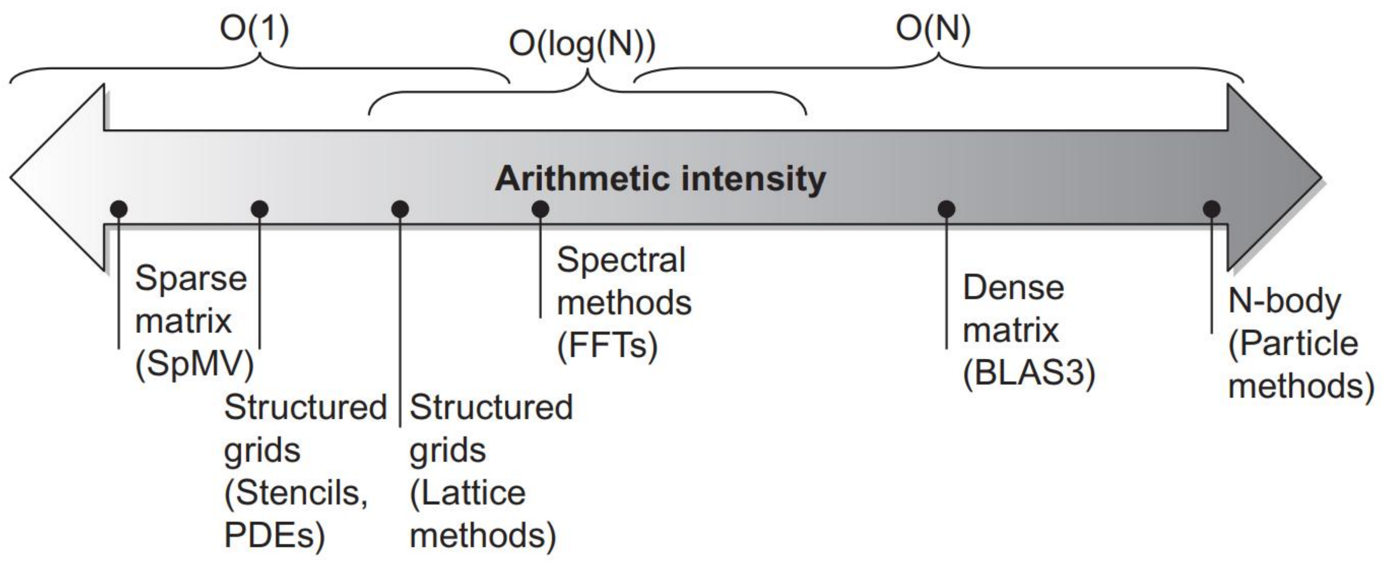
RISC-V SIMD code for DAXPY

```
       fld       f0,a           #Load scalar a
       splat.4D  f0,f0          #Make 4 copies of a
       addi      x28,x5,#256    #Last address to load
Loop:  fld.4D    f1,0(x5)       #Load X[i] ... X[i+3]
       fmul.4D   f1,f1,f0       #a×X[i] ... a×X[i+3]
       fld.4D    f2,0(x6)       #Load Y[i] ... Y[i+3]
       fadd.4D   f2,f2,f1       # a×X[i]+Y[i]...
                                # a×X[i+3]+Y[i+3]
       fsd.4D    f2,0(x6)       #Store Y[i]... Y[i+3]
       addi      x5,x5,#32      #Increment index to X
       addi      x6,x6,#32      #Increment index to Y
       bne       x28,x5,Loop    #Check if done
```

NOTE: We add the suffix "**4D**" on instructions that operate on **four double-precision operands** at once

Roofline Performance Model

- **Arithmetic intensity**
  - the ratio of floating-point operations per byte of memory accessed



Roofline Performance Model

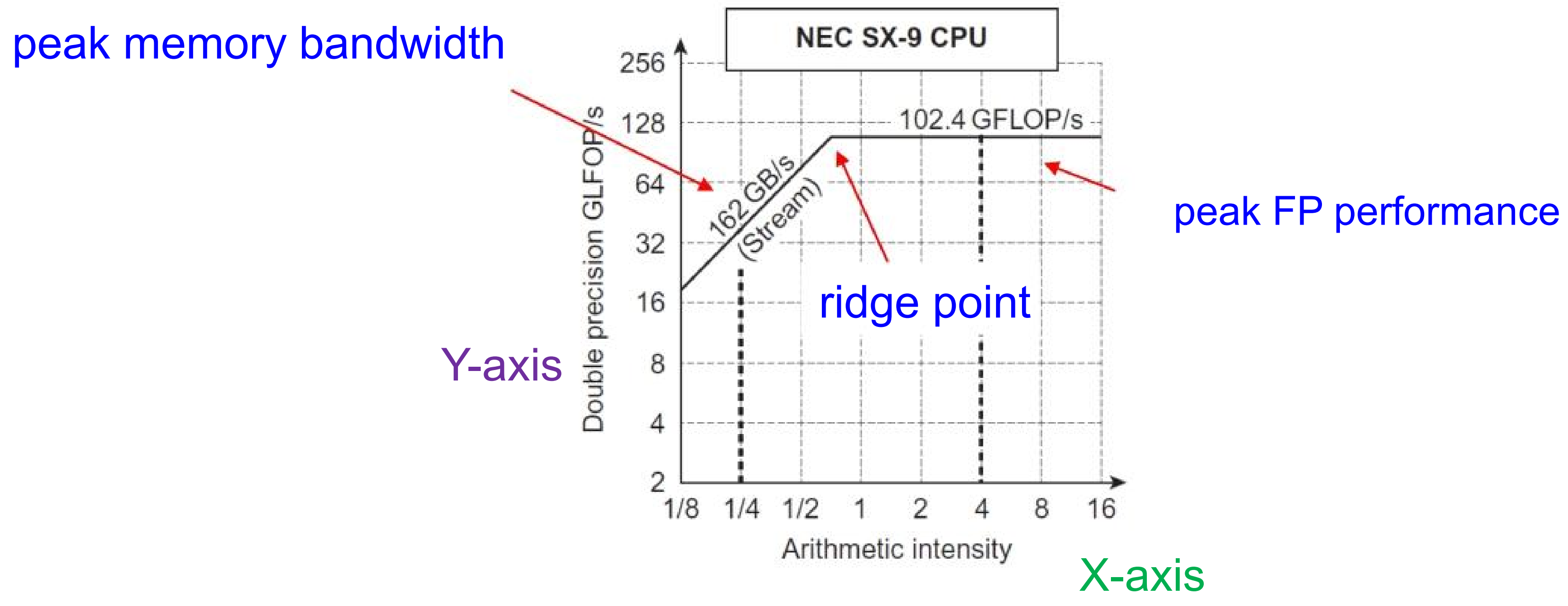
- One intuitive way to compare variations of SIMD architectures is the **Roofline model**

- **Basic idea:**
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine

Roofline Model Example

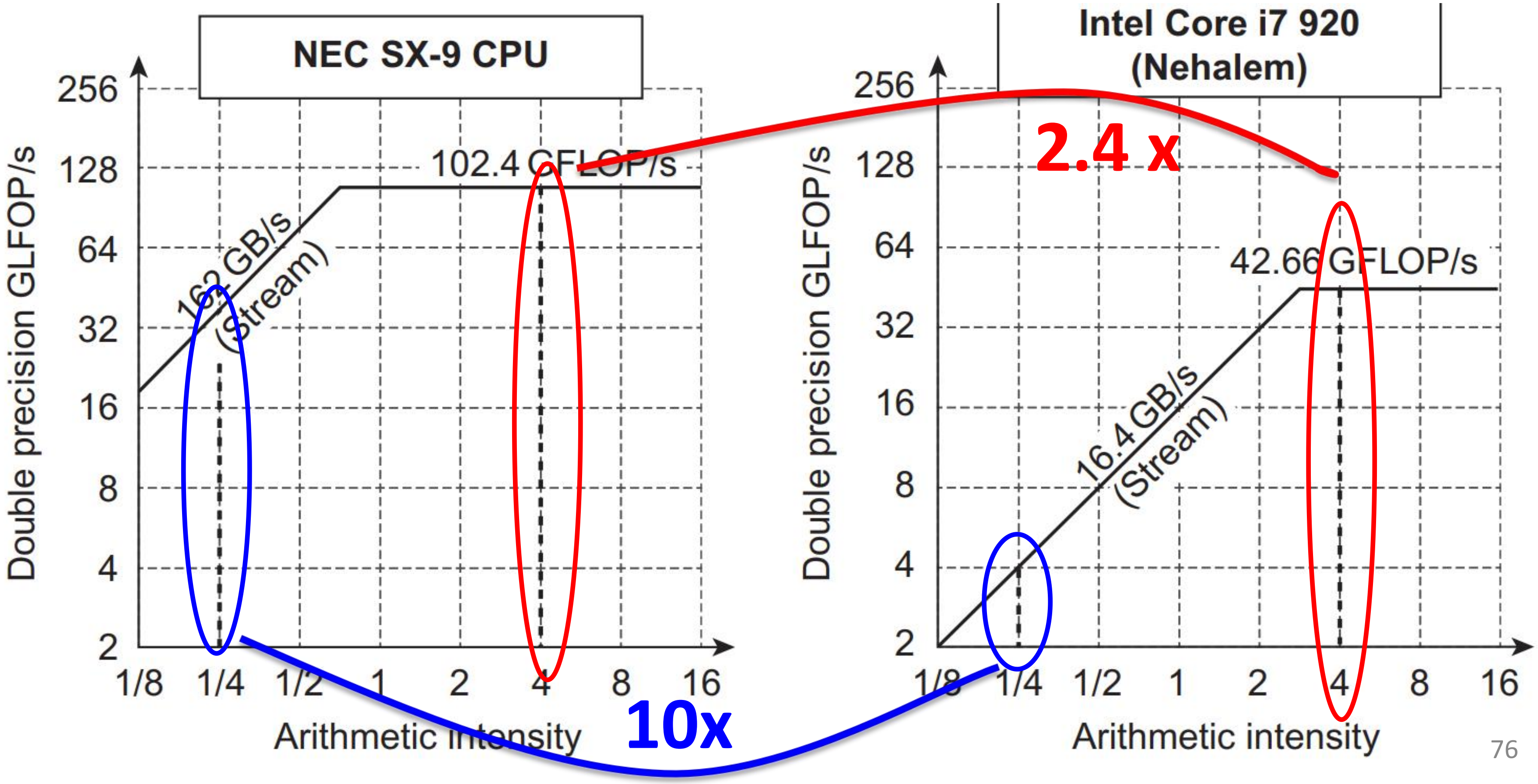
- The “Roofline” sets an upper bound on performance of a kernel depending on its arithmetic intensity
- Y-axis: attainable fp performance (GFLOPs/sec)

  Attainable GFLOPs/s = min(Peak Memory BW × Arithmetic Intensity, Peak FP Perf.)
- X-axis: arithmetic intensity (1/8-to 16 FLOP/DRAM byte accessed)



Roofline Model Example

$$\text{Attainable GFLOPs/s} = \text{Min}(\text{Peak Memory BW} \times \text{Arithmetic Intensity}, \text{ Peak Floating} - \text{Point Perf.})$$



4 Graphics Processing Units

5 Detecting and Enhancing Loop-Level Parallelism

Loop-Level Parallelism

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence
- Example 1:

```
  for (i=999; i>=0; i=i-1)
          x[i] = x[i] + s;
```

  - No loop-carried dependence

Loop-Level Parallelism

Example 2:

```
for (i=0; i<100; i=i+1) {
    A[i+1] = A[i] + C[i]; /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, because iteration i computes A[i+1], which is read in iteration i+1

- The same is true of S2 for B[i] and B[i+1]

2. S2 uses the value A[i+1] computed by S1 in the same iteration

Loop-Level Parallelism

- Example 3: **What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel**

```
for (i=0; i<100; i=i+1) {
     A[i] = A[i] + B[i]; /* S1 */
     B[i+1] = C[i] + D[i]; /* S2 */
}
```

S1 uses value computed by S2 in previous iteration

- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
     B[i+1] = C[i] + D[i];
     A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

The dependence is no longer loop-carried

Finding dependencies

- **Assume indices are affine（仿射的）：**
  - *$a * i + b$ (i is loop index)*
  - *Sparse array accesses: $x[y[i]]$ => nonaffine accesses*

- **Assume:**
  - Store to $a * i + b$, then load from $c * i + d$, i runs from m to n
  - **Dependence exists** if:
    - Given j, k such that $m \le j \le n$, $m \le k \le n$
    - Store to $a * j + b$, load from $c * k + d$, and
    - $a * j + b = c * k + d$

Finding dependencies

- **A simple and sufficient test for the absence of a dependence is the *greatest common divisor (GCD) test***
  - It is based on the observation that if a loop-carried dependence exists, then

GCD (c, a) must divide (d - b)

Finding dependencies

Example:

```
  for (i=0; i<100; i=i+1) {
      X[2*i+3] = X[2*i] * 5.0;
  }
```

a=2, b=3, c=2, d=0 ➔ GCD(a, c)=2, d-b=-3

Because 2 does not divide -3, no dependence is possible

Finding dependencies

- The GCD test is sufficient to guarantee that no dependence exists
- ***however**, there are cases where the GCD test succeeds but no dependence exists*
  - This can arise because the GCD test does not consider the loop bounds
- In general, determining whether a dependence actually exists is ***NP-complete***

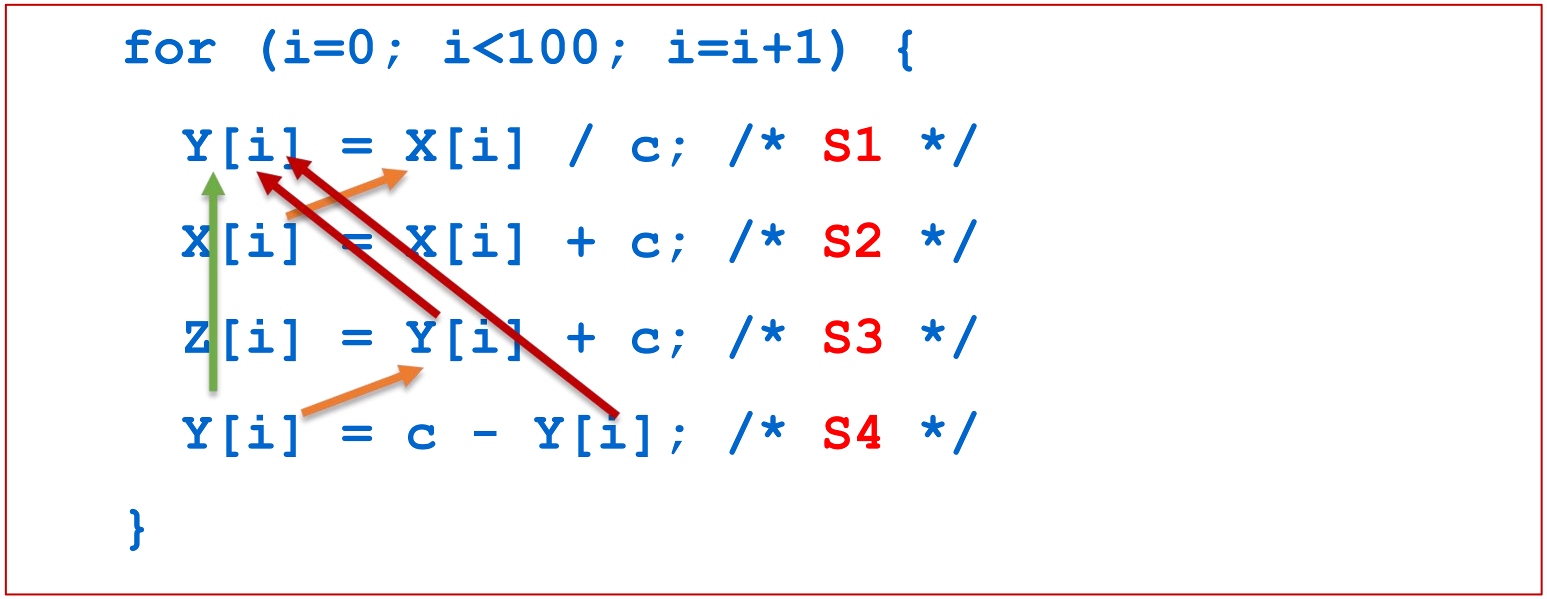
Finding dependencies

Example:

- The following loop has multiple types of dependences
  - ➢ Find all the true dependences, output dependences, and antidependences
  - ➢ eliminate the output dependences and antidependences by renaming

```
for (i=0; i<100; i=i+1) {

  Y[i] = X[i] / c; /* S1 */

  X[i] = X[i] + c; /* S2 */

  Z[i] = Y[i] + c; /* S3 */

  Y[i] = c - Y[i]; /* S4 */

}
```

Finding dependencies



```
for (i=0; i<100; i=i+1) {
  Y[i] = X[i] / c; /* S1 */
  X[i] = X[i] + c; /* S2 */
  Z[i] = Y[i] + c; /* S3 */
  Y[i] = c - Y[i]; /* S4 */
}
```

1. There are true dependences from S1 to S3 and from S1 to S4 because of Y[i]
   - These dependences will force S3 and S4 to wait for S1 to complete
2. There is an antidependence from S1 to S2, based on X[i]
3. There is an antidependence from S3 to S4 for Y[i]
4. There is an output dependence from S1 to S4, based on Y[i]

Eliminating dependencies

```
for (i=0; i<100; i=i+1) {
  Y[i] = X[i] / c; /* S1 */
  X[i] = X[i] + c; /* S2 */
  Z[i] = Y[i] + c; /* S3 */
  Y[i] = c - Y[i]; /* S4 */
}
```

```
for (i=0; i<100; i=i+1 {
T[i] = X[i] / c; /* Y renamed to T to remove output dependence */
X1[i] = X[i] + c;/* X renamed to X1 to remove antidependence */
Z[i] = T[i] + c;/* Y renamed to T to remove antidependence */
Y[i] = c - T[i];
}
```

Eliminating Dependent Computations

- **One of the most important forms of dependent computations is a recurrence**

```
for (i=9999; i>=0; i=i-1)
      sum = sum + x[i] * y[i];
```

This loop is not parallel because it has a loop-carried dependence on the variable sum

Eliminating Dependent Computations

- We can transform it to a set of loops, one of which is completely parallel and the other partly parallel

```
for (i=9999; i>=0; i=i-1)
      sum = sum + x[i] * y[i];
```

```
for (i=9999; i>=0; i=i-1)
      sum[i] = x[i] * y[i];

for (i=9999; i>=0; i=i-1)
      finalsum = finalsum + sum[i];
```

Reduction

Although this loop is not parallel, it has a very specific structure called a ***reduction***

```
for (i=9999; i>=0; i=i-1)
      finalsum = finalsum + sum[i];
```

Note: **Reductions are sometimes handled by special hardware** in a vector and SIMD architecture that allows the reduce step to be done much faster than it could be done in scalar mode

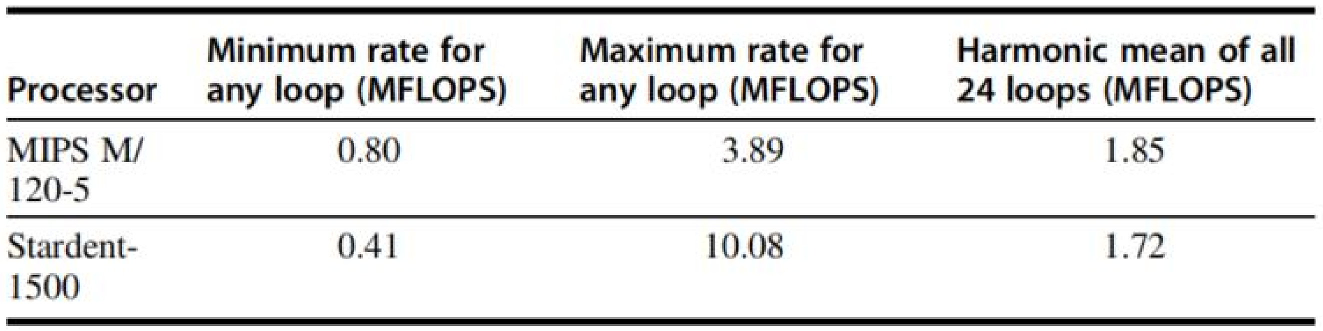
```
for (i=999; i>=0; i=i-1)
   finalsum[p] = finalsum[p] + sum[i+1000*p];
```

Processor Number

Fallacies and Pitfalls

Fallacies and Pitfalls

- **GPUs suffer from being coprocessors**
  - GPUs have flexibility to change ISA
- **Concentrating on peak performance in vector architectures and ignoring start-up overhead**
  - Overheads require long vector lengths to achieve speedup
- **Increasing vector performance without comparable increases in scalar performance**



Processor	Minimum rate for any loop (MFLOPS)	Maximum rate for any loop (MFLOPS)	Harmonic mean of all 24 loops (MFLOPS)
MIPS M/ 120-5	0.80	3.89	1.85
Stardent- 1500	0.41	10.08	1.72

Fallacies and Pitfalls

- **You can get good vector performance without providing memory bandwidth**
  - memory bandwidth is quite important to all SIMD architectures
    - DAXPY requires **1.5** memory references per floating-point operation
- **On GPUs, just add more threads if you don't have enough memory performance**
  - If memory accesses are not correlated among CUDA Threads, the memory system will get progressively slower in responding to each individual request
  - Eventually, even many threads will not cover latency
  - For the "more CUDA Threads" strategy to work, CUDA Threads must be well behaved in terms of locality of memory accesses

Summary

- Vector Architecture
- Graphics Processing Units
  - *hierarchical thread organization*
  - *hierarchical memory structure*
- Detecting and Enhancing Loop-Level Parallelism