

---

# Handout 5 SIMD: Vector, SIMD

# Introduction

---

- **Instruction level parallelism**
  - **Superscalar + OOO**
- **Thread level parallelism**
  - **Simultaneous multi-threading**
  - **Multi-core multi-threading**
- **Data level parallelism**
  - **SIMD**

# Introduction

---

- **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. One instruction stream**

# Architecture exploring SIMD Parallelism

---

- **Vector architectures**
- **SIMD extensions**
- **Graphics Processor Units (GPUs)**
  
- **For x86 processors:**
  - **Expect two additional cores per chip per year**
  - **SIMD width to double every four years**
  - **Potential speedup from SIMD to be twice that from MIMD!**

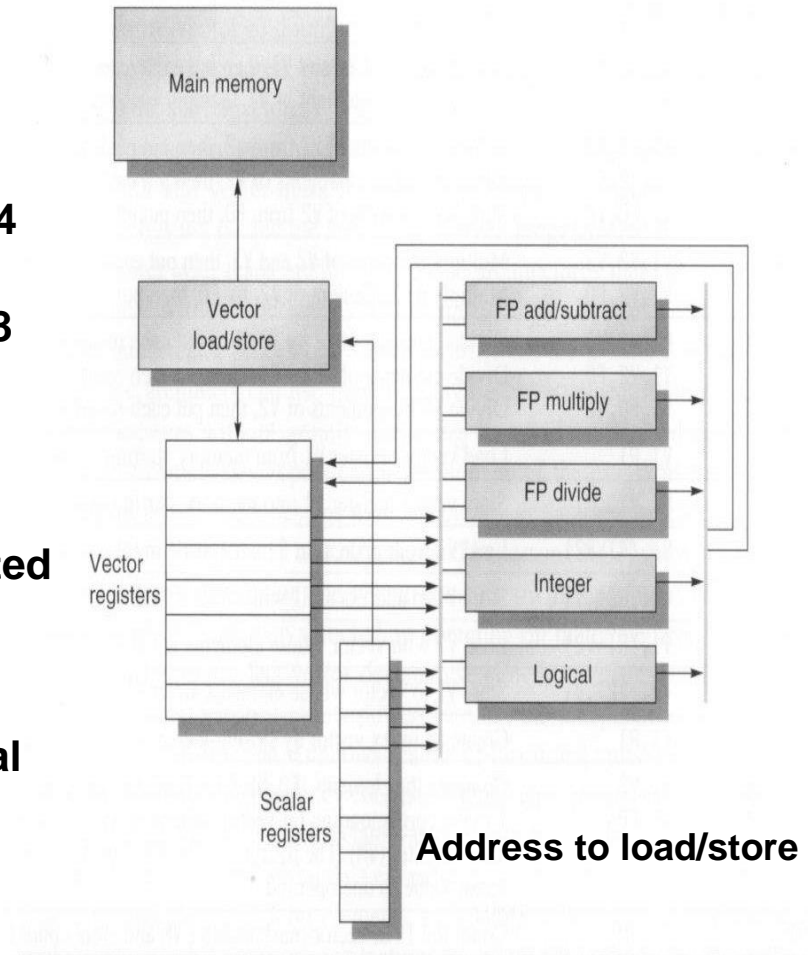
# Vector Architectures

---

- **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**
    - » **Vector load/store are deeply pipelined**

# V-MIPS

- **Example architecture: VMIPS**
  - **Loosely based on Cray-1**
  - **Vector registers**
    - » Each register holds a 64-element, 64 bits/element vector
    - » 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**
    - » 32 general-purpose registers
    - » 32 floating-point registers



# V-MIPS ISA

---

- **Vector instructions 1**
- **A vector V has 64 elements**
- **F is scalar**

---

ADDVV.D	V1, V2, V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1, V2, F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1, V2, V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1, V2, F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1, F0, V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1, V2, V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1, V2, F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1, V2, V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1, V2, F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1, F0, V2	Divide F0 by elements of V2, then put each result in V1.

---

# V-MIPS

- **Vector instructions 2**

LV	V1, R1	Load vector register V1 from memory starting at address R1.
SV	R1, V1	Store vector register V1 into memory starting at address R1.
LVWS	V1, (R1, R2)	Load V1 from address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
SVWS	(R1, R2), V1	Store V1 to address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
LVI	V1, (R1+V2)	Load V1 with vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).
SVI	(R1+V2), V1	Store V1 to vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).
CVI	V1, R1	Create an index vector by storing the values $0, 1 \times R1, 2 \times R1, \dots, 63 \times R1$ into V1.
S--VV.D	V1, V2	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
S--VS.D	V1, F0	
POP	R1, VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR, R1	Move contents of R1 to vector-length register VL.
MFC1	R1, VLR	Move the contents of vector-length register VL to R1.
MVTM	VM, F0	Move contents of F0 to vector-mask register VM.
MVFM	F0, VM	Move contents of vector-mask register VM to F0.



# VMIPS Instructions

---

- **ADDVV.D: add two vectors**
- **ADDVS.D: add vector to a scalar**
- **LV/SV: vector load and vector store from address**
  
- **Example:**

<b>L.D</b>	<b>F0,a</b>	<b>; load scalar a</b>
<b>LV</b>	<b>V1,Rx</b>	<b>; load vector X</b>
<b>MULVS.D</b>	<b>V2,V1,F0</b>	<b>; vector-scalar multiply</b>
<b>LV</b>	<b>V3,Ry</b>	<b>; load vector Y</b>
<b>ADDVV</b>	<b>V4,V2,V3</b>	<b>; add</b>
<b>SV</b>	<b>Ry,V4</b>	<b>; store the result</b>
- **Requires 6 instructions vs. almost 600 for MIPS**

# Vector Execution Time

---

- **Execution time depends on three factors:**
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- **VMIPS functional units consume one element per clock cycle**
  - Assume one lane
  - Execution time is approximately the vector length
- ***Convoy***
  - A set of vector instructions that could potentially execute together- called a convoy
    - » Instructions in a convoy contain no structural hazards
    - » By counting the number of convoys to estimate the performance

# Chimes

---

- Sequences with read-after-write dependency hazards can be in the same convoy via *chaining*
- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
  - The results from the first functional unit in the **chain** are forwarded to the second functional unit.
- *Chime*
  - Unit of time to execute one convey
  - $m$  convoys executes in  $m$  chimes
  - For vector length of  $n$ , requires  $m \times n$  clock cycles
    - » This however has ignored certain overheads within the vector execution, such as due to length difference.

# Vector problem: $Y = a \times X + Y$

---

LV	V1,Rx	;load vector X
MULVS.D	V2,V1,F0	;vector-scalar multiply
LV	V3,Ry	;load vector Y
ADDVV.D	V4,V2,V3	;add two vectors
SV	Ry,V4	;store the sum

## Convoys:

1	LV	MULVS.D // by chaining
2	LV	ADDVV.D // LV V3 on convoys 2 due to Struc. Haz.
3	SV	

**3** chimes, **2** FP ops per result, cycles per FLOP ( $3/2$ ) = 1.5  
(ignore pipeline start-up time) ( $192/128 = 1.5$ )

For 64 element vectors, requires  $64 \times 3 = 192$  clock cycles

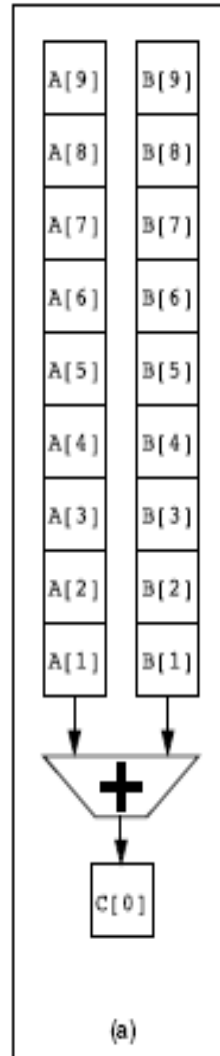
# Challenges

---

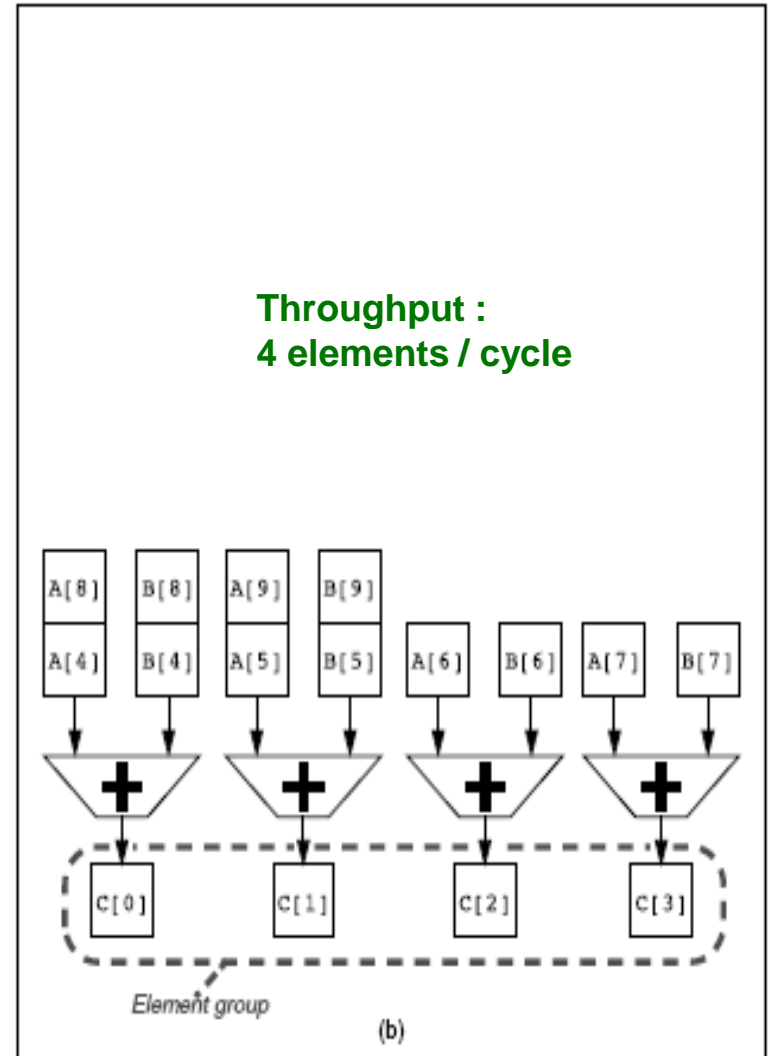
- **Start up time: pipelined functional unit**
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - » Floating-point add => 6 clock cycles
    - » Floating-point multiply => 7 clock cycles
    - » Floating-point divide => 20 clock cycles
    - » Vector load => 12 clock cycles
- **Improvements:**
  - > 1 element per clock cycle
  - Non-64 wide vectors (vector lengths are not the same as the vector registers)
  - IF statements in vector code (conditional statement)
  - Memory system optimizations to support vector processors
  - Multiple dimensional matrices
  - Sparse matrices
  - Programming a vector computer

# Multiple pipelines for a vector ADD

- $C = A + B$
- (a) single pipeline
- (b) four pipelines
- Element  $n$  of vector register  $A$  is “hardwired” to element  $n$  of vector register  $B$ 
  - Allows for multiple hardware lanes
  - Elements of  $A$  and  $B$  are interleaved across the four pipelines



(a)

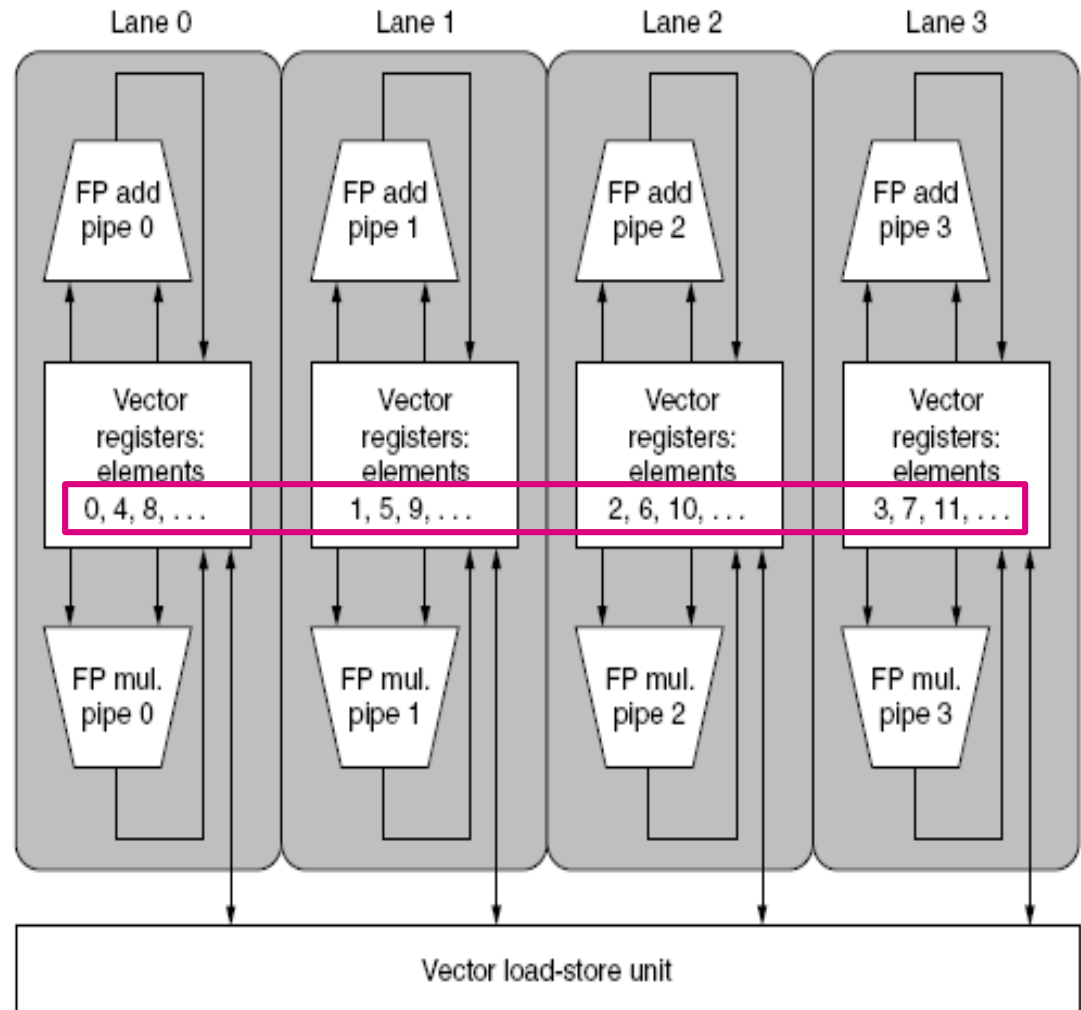


(b)

1 element / cycle

# Simply spread the elements of a vector register across the lanes

- A four lane
- First lane holds element 0 for all vector registers
- A 64-cycle Chime -> 16 cycles



# Create a Vector Length Register

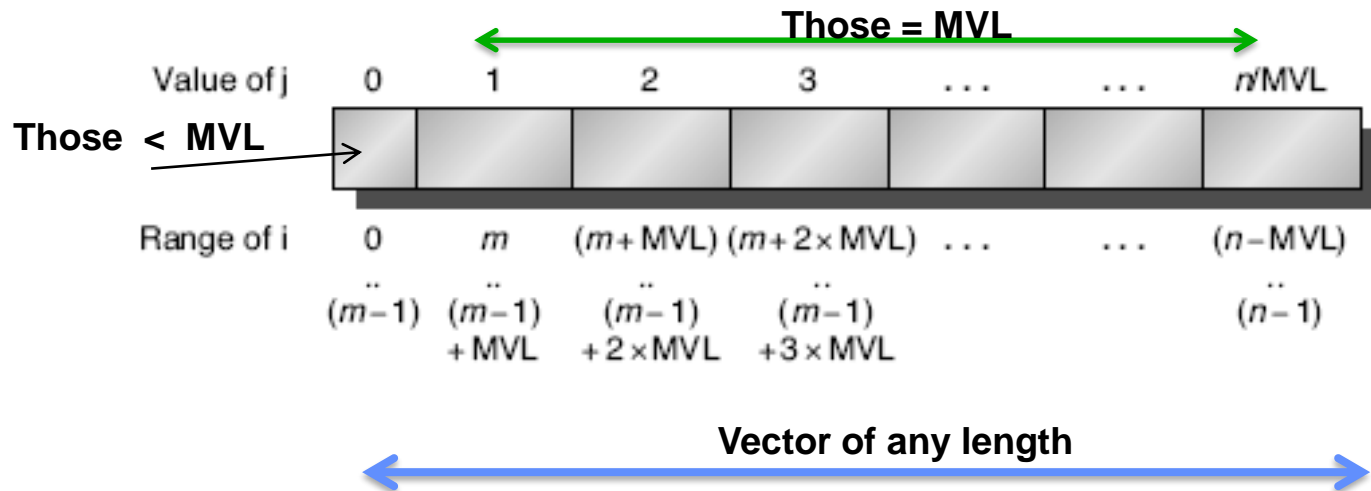
---

- **Vector length is often unknown at compile time**
- **Use Vector Length Register (VLR) to control the length of any vector operation, VLR length  $\leq$  maximum vector register length (MVL), (MVL depends on your hardware design)**
- **If vector size of an operation  $<$  MVL, just use length in VLR to control vector operation**
  - Specific:**
    - » **Move vector length to VLR**
    - » **VLR controls the corresponding vector functional unit**



# Create a Vector Length Register-2

- What if Vector Length  $\geq$  MVL
- Use a code generation scheme called strip mining for vectors over the maximum length:
  - Run those of which VL is  $<$  MVL
  - Then Run any number of iterations that are a multiple of MVL.



Like loop unrolling

# Handle IF: Vector Mask Registers

- Consider:

for (i = 0; i < 64; i=i+1)

if (X[i] != 0)

X[i] = X[i] – Y[i];

Mask register provides conditional execution of each element operation in a vector instruction .

In run time, enable the mask register when needed.  
1: execute, 0: disable

- Use vector mask register to “disable” elements:

LV	V1,Rx	;load vector X into V1
LV	V2,Ry	;load vector Y
L.D	F0,#0	;load FP zero into F0
<b>SNEVS.D</b>	<b>V1,F0</b>	<b>;sets VM(i) to 1 if V1(i)!=F0</b>
SUBVV.D	V1,V1,V2	;subtract under vector mask
SV	Rx,V1	;store the result in X

Compiler puts mask instruction



- GFLOPS rate decreases!

# Vector Load/Store

---

- **Start-up time of a load: the time to get the first word from memory into a register**
  - **Use cache to reduce start-up time**
  - **Then best one word per cycle afterwards**
- **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**
  - **Support multiple vector processors sharing the same memory**

# Need a large number of independent memory banks for vector load/store

- **Example: Allow all processors to run at full memory bandwidth**
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks are needed?

**32 x 6 = 192 memory references/CPU cycle**

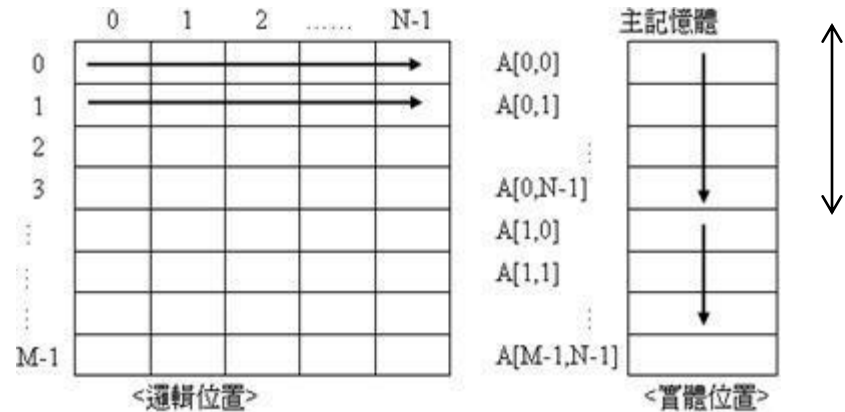
**Each SRAM bank is busy for  $15/2.167 = 7$  CPU clock cycles**

**during 7 CPU cycles,  $192 \times 7 = 1344$  (memory ref) memory banks**

# Stride

- Example: row major order (one entry = 8 bytes)
- Load

```
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];
  }
```



For D, since it is k changes first,

D[00] then D[1,0], so it has a stride of 800 bytes (8 bytes x 100)

For B, B[0,0], then B[0,1], so the stride is only 8 bytes

**Need a vector load with stride here: LVWS**

# V-MIPS

- **Vector instructions 2**

LV	V1, R1	Load vector register V1 from memory starting at address R1.
SV	R1, V1	Store vector register V1 into memory starting at address R1.
LVWS	V1, (R1, R2)	Load V1 from address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
SVWS	(R1, R2), V1	Store V1 to address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
LVI	V1, (R1+V2)	Load V1 with vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).
SVI	(R1+V2), V1	Store V1 to vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).
CVI	V1, R1	Create an index vector by storing the values $0, 1 \times R1, 2 \times R1, \dots, 63 \times R1$ into V1.
S--VV.D	V1, V2	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
S--VS.D	V1, F0	
POP	R1, VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR, R1	Move contents of R1 to vector-length register VL.
MFC1	R1, VLR	Move the contents of vector-length register VL to R1.
MVTM	VM, F0	Move contents of F0 to vector-mask register VM.
MVFM	F0, VM	Move contents of vector-mask register VM to F0.

# Representation of sparse matrices

---

- **Compressed representation (zeros not included)**
- **Normal representation (zeros are included)**
- **Gather operation**
  - Using an index vector and fetching the vector whose elements are at the addresses given by adding a base to the offsets given in the index vector –This is dense form in the vector register
  - **LVI (load vector indexed or gather)**
- **Scatter store**
  - Using the same index vector, the sparse vector can be stored in expanded form by a scatter store
  - **SVI (store vector indexed or scatter)**



# V-MIPS

- **Vector instructions 2**

LV	V1, R1	Load vector register V1 from memory starting at address R1.
SV	R1, V1	Store vector register V1 into memory starting at address R1.
LVWS	V1, (R1, R2)	Load V1 from address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
SVWS	(R1, R2), V1	Store V1 to address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
LVI	V1, (R1+V2)	Load V1 with vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).
SVI	(R1+V2), V1	Store V1 to vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).
CVI	V1, R1	Create an index vector by storing the values $0, 1 \times R1, 2 \times R1, \dots, 63 \times R1$ into V1.
S--VV.D	V1, V2	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
S--VS.D	V1, F0	
POP	R1, VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR, R1	Move contents of R1 to vector-length register VL.
MFC1	R1, VLR	Move the contents of vector-length register VL to R1.
MVTM	VM, F0	Move contents of F0 to vector-mask register VM.
MVFM	F0, VM	Move contents of vector-mask register VM to F0.



# Scatter-Gather

---

- Consider:

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

$$A[K[i]] = A[K[i]] + C[M[i]]; \text{ // } K[i] \text{ and } M[i] \text{ are index vectors of size } n$$

To do a sparse vector sum of array A and C

index vectors K and M designate the non-zero elements of A and C

base

	1		6	0	0	5	
		K	31				

K is an array (vector)  
of offsets : ( $i=0$ ) 0, ( $i=1$ ) 3, ....

# Scatter-Gather

---

- Consider:

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

$A[K[i]] = A[K[i]] + C[M[i]]$ ; // K[i] and M[i] are index vectors of same size n

- Use index vector: Ra, Rc, Rk Rm contains the starting addresses of the vectors:

- the inner loop

LV	Vk, Rk	;load K, vector Vk now has the offset vector
LVI	Va, (Ra+Vk)	;load A[K[]], gather, <b>base Ra</b>
LV	Vm, Rm	;load M, Vm is the offset vector
LVI	Vc, (Rc+Vm)	;load C[M[]]
ADDVV.D	Va, Va, Vc	;add them
SVI	(Ra+Vk), Va	;store A[K[]], scatter

# Programming Vector Architectures

- Compilers can provide feedback to programmers
- 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

Cray- Y-MP 1991

Without hint from programmers

# SIMD Extensions

---

- **Media applications operate on data types narrower than the native word size**
  - **Disconnect carry chains to “partition” adder is of little overheads**

## 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 Extensions v.s. Vector

---

- **Number of data operands encoded into op code, fixed operands in SIMD**
- **While in vector CPU, it is variable and specified in vector length register**
- **No sophisticated addressing modes (strided, scatter-gather) in SIMD**
- **Often no mask registers in SIMD (means no support of conditional execution)**
- **Vector is much more complicated than SIMD ext.**

# SIMD Implementations

- **Implementations:**

---

- Intel MMX (1996): **64-bit FP registers**
  - » Eight 8-bit integer ops or four 16-bit integer ops
- Streaming SIMD Extensions (SSE) (1999): **128-bit reg**
  - » Sixteen 8-bit ops, eight 16-bit integer ops
  - » Four 32-bit integer/fp ops or two 64-bit integer/fp ops
- Advanced Vector Extensions (2010): **256-bit reg**
  - » Four 64-bit integer/fp ops
- Intel® Advanced Vector Extensions 512 / Intel® AVX-512 (2013): **512-bit reg**
  - » Programs can pack eight double precision or sixteen single precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers within the 512-bit vectors.
- Operands must be consecutive and aligned memory locations, within page boundary

**Future => 1024 bits reg**

# Example SIMD Code, 256-bit reg

## 4D: operate on 4 double-precision operands

---

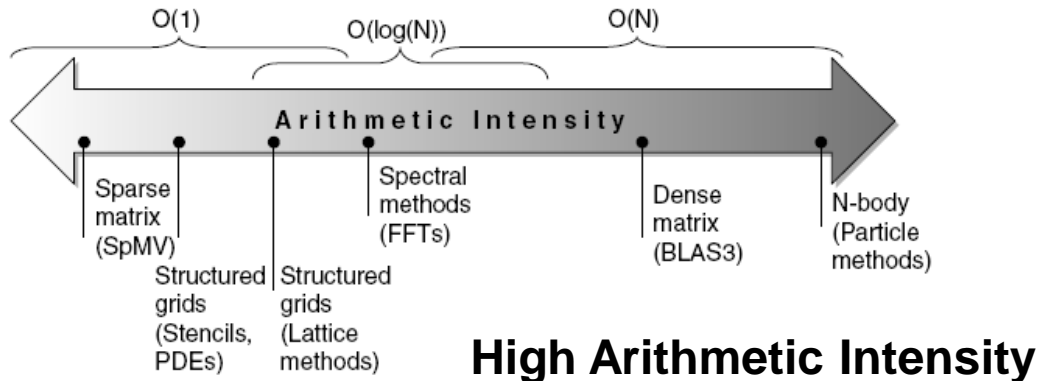
- Example DXPY:

```
L.D          F0,a          ;load scalar a
MOV          F1, F0        ;copy a into F1 for SIMD MUL
MOV          F2, F0        ;copy a into F2 for SIMD MUL
MOV          F3, F0        ;copy a into F3 for SIMD MUL
DADDIU      R4,Rx,#512     ;last address to load
Loop:       L.4D F4,0[Rx]   ;load X[i], X[i+1], X[i+2], X[i+3]
            MUL.4D F4,F4,F0 ;axX[i],axX[i+1],axX[i+2],axX[i+3]
            L.4D F8,0[Ry]   ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
            ADD.4D F8,F8,F4 ;axX[i]+Y[i], ..., axX[i+3]+Y[i+3]
            S.4D 0[Ry],F8   ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
            DADDIU Rx,Rx,#32 ;increment index to X
            DADDIU Ry,Ry,#32 ;increment index to Y
            DSUBU R20,R4,Rx ;compute bound
            BNEZR20,Loop    ;check if done
```

# Roofline Performance 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
- Arithmetic intensity
  - Floating-point operations per byte read:  
**FLOPs/byte**

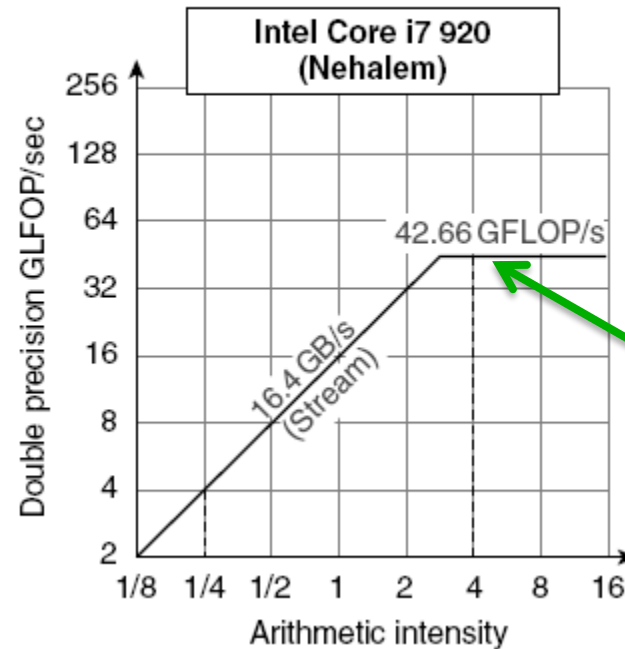
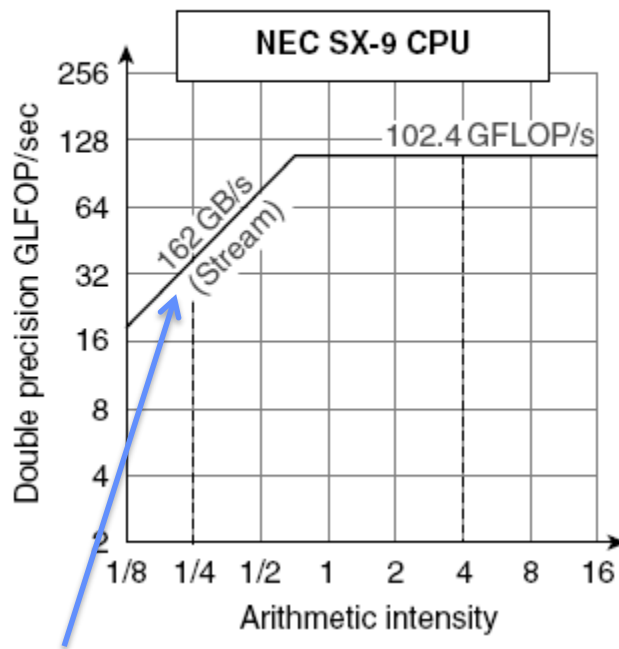




# Examples

- **Attainable GFLOPs/sec Min = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)**

FLOPs/byte



Hit the roof by computations

Hit the roof by  
by memory BW

3.2 GHz, with eight-way replicated vector pipes

# Summary

---

- **Vector processor**
  - **More flexible but complicated in context switching, vector ISA, memory systems**
- **SIMD extensions**
  - **Less expensive, fixed vector size, compatibility issues grow as more registers are added**