Vector Processors

Kavitha Chandrasekar
Sreesudhan Ramkumar

Agenda

- Why Vector processors
- Basic Vector Architecture
- Vector Execution time
- Vector load - store units and Vector memory systems
- Vector length Control
- Vector stride

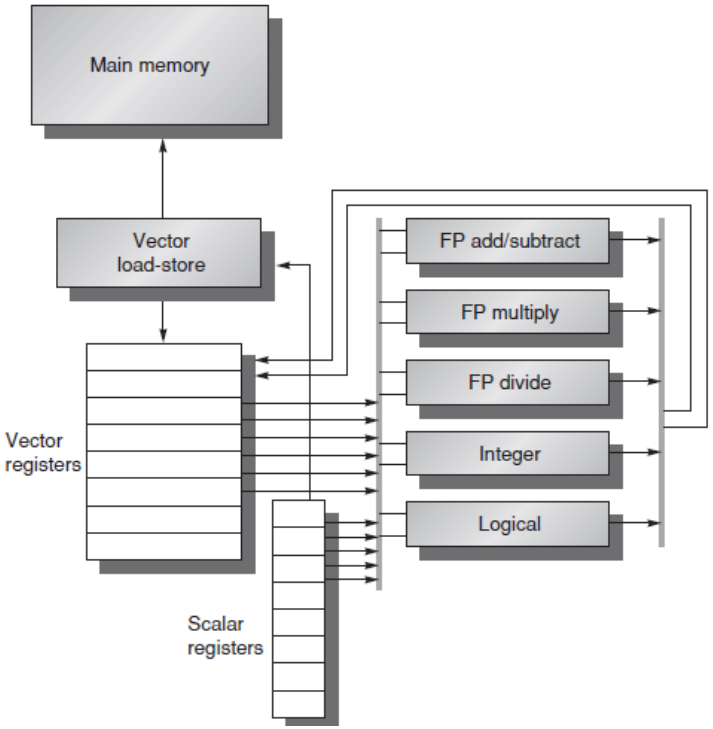
Limitations of ILP

- ILP:
  - Increase in instruction width (superscalar)
  - Increase in machine pipeline depth
  - Hence, Increase in number of in-flight instructions
- Need for increase in hardware structures like ROB, rename register files
- Need to increase logic to track dependences
- Even in VLIW, increase in hardware and logic is required

Vector Processor

- Work on linear arrays of numbers(vectors)
- Each iteration of a loop becomes one element of the vector
- Overcoming limitations of ILP:
  - Dramatic reduction in fetch and decode bandwidth.
  - No data hazard between elements of the same vector.
  - Data hazard logic is required only between two vector instructions
  - Heavily interleaved memory banks. Hence latency of initiating memory access versus cache access is amortized.
  - Since loops are reduced to vector instructions, there are no control hazards
  - Good performance for poor locality

Basic Architecture



- Vector and Scalar units
- Types:
  - Vector-register processors
  - Memory-memory Vector processors
- Vector Units
  - Vector registers (with 2 read and 1 write ports)
  - Vector functional units (fully pipelined)
  - Vector Load Store unit(fully pipelined)
  - Set of scalar registers

VMIPS vector instructions

Instruction	Operands	Function
ADDV.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.
SUBV.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.
MULV.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.
DIVV.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.
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 × R2.
SVWS	(R1,R2),V1	Store V1 from address at R1 with stride in R2, i.e., R1+i × 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 × R1, 2 × R1,...,63 × R1 into V1.
S--V.D S--VS.D	V1,V2 V1,F0	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.
POP	R1,VM	Count the 1s in the vector-mask register and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1 MFC1	VLR,R1 R1,VLR	Move contents of R1 to the vector-length register. Move the contents of the vector-length register to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to the vector-mask register. Move contents of vector-mask register to F0.

MIPS vs VMIPS (DAXPY loop)

$$Y = a \times X + Y$$

```
        L.D     F0,a        ;load scalar a
        DADDIU  R4,Rx,#512  ;last address to load
Loop:   L.D     F2,0(Rx)    ;load X(i)
        MUL.D   F2,F2,F0    ;a × X(i)
        L.D     F4,0(Ry)    ;load Y(i)
        ADD.D   F4,F4,F2    ;a × X(i) + Y(i)
        S.D     0(Ry),F4    ;store into Y(i)
        DADDIU  Rx,Rx,#8    ;increment index to X
        DADDIU  Ry,Ry,#8    ;increment index to Y
        DSUBU   R20,R4,Rx   ;compute bound
        BNEZ    R20,Loop    ;check if done
```

Here is the VMIPS code for DAXPY.

```
        L.D     F0,a        ;load scalar a
        LV      V1,Rx       ;load vector X
        MULVS.D V2,V1,F0    ;vector-scalar multiply
        LV      V3,Ry       ;load vector Y
        ADDV.D  V4,V2,V3    ;add
        SV      Ry,V4       ;store the result
```

Execution time of vector instructions

- Factors:
  - length of operand vectors
  - structural hazards among operations
  - data dependences
- Overhead:
  - initiating multiple vector instructions in a clock cycle
  - Start-up overhead (more details soon)

Vector Execution time (contd.)

- Terms:
  - Convoy:
  - set of vector instructions that can begin execution together in one clock period
  - Instructions in a convoy must not contain any structural or data hazards
  - Analogous to placing scalar instructions in VLIW
  - One convoy must finish before another begins
  - Chime: Unit of time taken to execute one convoy
- Hence for vector sequence *m* convoys executes in *m chimes*
- Hence for vector length of n, time=m × n clock cycles

Example

```
LV        V1,Rx      ;load vector X
MULVS.D   V2,V1,F0   ;vector-scalar multiply
LV        V3,Ry      ;load vector Y
ADDV.D    V4,V2,V3   ;add
SV        Ry,V4      ;store the result
```

Convoy

1. LV
2. MULVS.D LV
3. ADDV.D
4. SV

Start-up overhead

- *Startup time:* Time between initialization of the instruction and time the first result emerges from pipeline
- Once pipeline is full, result is produced every cycle.
- If vector lengths were infinite, startup overhead is amortized
- But for finite vector lengths, it adds significant overhead

Startup overhead-example

Unit	Start-up overhead (cycles)
Load and store unit	12
Multiply unit	7
Add unit	6

Figure F.4 Start-up overhead.

Convoy	Starting time	First-result time	Last-result time
1. LV	0	12	$11 + n$
2. MULVS.D LV	$12 + n$	$12 + n + 12$	$23 + 2n$
3. ADDV.D	$24 + 2n$	$24 + 2n + 6$	$29 + 3n$
4. SV	$30 + 3n$	$30 + 3n + 12$	$41 + 4n$

Figure F.5 Starting times and first- and last-result times for convoys 1 through 4. The vector length is n.

Operation	Start-up penalty
Vector add	6
Vector multiply	7
Vector divide	20
Vector load	12

Figure F.6 Start-up penalties on VMIPS. These are the start-up penalties in clock cycles for VMIPS vector operations.

Vector Load-Store Units and Vector Memory Systems

- Start-up time: Time to get first word from memory into a register
- To produce results every clock multiple memory banks are used
- Need for multiple memory banks in vector processors:
  - Many vector processors allow multiple loads and stores per clock cycle
  - Support for nonsequential access
  - Support for sharing of system memory by multiple processors

Example

- Number of memory banks required:

	Bank							
Cycle no.	0	1	2	3	4	5	6	7
0		136						
1		busy	144					
2		busy	busy	152				
3		busy	busy	busy	160			
4		busy	busy	busy	busy	168		
5		busy	busy	busy	busy	busy	176	
6			busy	busy	busy	busy	busy	184
7	192			busy	busy	busy	busy	busy
8	busy	200			busy	busy	busy	busy
9	busy	busy	208			busy	busy	busy
10	busy	busy	busy	216			busy	busy
11	busy	busy	busy	busy	224			busy
12	busy	busy	busy	busy	busy	232		
13		busy	busy	busy	busy	busy	240	
14			busy	busy	busy	busy	busy	248
15	256			busy	busy	busy	busy	busy
16	busy	264			busy	busy	busy	busy

Real world issues

- Vector length in a program is not always fixed(say 64)
- Need to access non adjacent elements from memory
- Solutions:
  - Vector length Control
  - Vector Stride

Vector Length Control

- Example:

```
      do 10 i = 1,n
10    Y(i) = a * X(i) + Y(i)
```

- Here value of ‘n’ might be known only during runtime.
- In case of parameters to procedure, it changes even during runtime
- Hence, VLR (Vector Length Register) is used to control the length of a vector operation during runtime
- MVL (Maximum Vector Length) holds the maximum length of a vector operation (processor dependent)

Vector Length Control(contd.)

- Strip mining:
  - When vector operation is longer than MVL, this concept is used

```
      low = 1
      VL = (n mod MVL) /*find the odd-size piece*/
      do 1 j = 0,(n / MVL) /*outer loop*/
           do 10 i = low, low + VL - 1 /*runs for length VL*/
               Y(i) = a * X(i) + Y(i) /*main operation*/
10         continue
           low = low + VL /*start of next vector*/
           VL = MVL /*reset the length to max*/
1     continue
```

Execution time due to strip mining

- Key factors that contribute to the running time of a strip-mined loop consisting of a sequence of convoys:

1. Number of convoys in the loop, which determines the number of chimes.
2. Overhead for each strip-mined sequence of convoys. This overhead consists of the cost of executing the scalar code for strip-mining each block, plus the vector start-up cost for each convoy.

- Total running time for a vector sequence operating on a vector of length *n,Tn:*

$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}}$$

Example

```
        DADDUI   R2,R0,#1600  ;total # bytes in vector
        DADDU    R2,R2,Ra     ;address of the end of A vector
        DADDUI   R1,R0,#8     ;loads length of 1st segment
        MTC1     VLR,R1       ;load vector length in VLR
        DADDUI   R1,R0,#64    ;length in bytes of 1st segment
        DADDUI   R3,R0,#64    ;vector length of other segments
Loop:   LV       V1,Rb        ;load B
        MULVS.D  V2,V1,Fs     ;vector * scalar
        SV       Ra,V2        ;store A
        DADDU    Ra,Ra,R1     ;address of next segment of A
        DADDU    Rb,Rb,R1     ;address of next segment of B
        DADDUI   R1,R0,#512   ;load byte offset next segment
        MTC1     VLR,R3       ;set length to 64 elements
        DSUBU    R4,R2,Ra     ;at the end of A?
        BNEZ     R4,Loop      ;if not, go back
```

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

$$T_{start} = 12 + 7 + 12 = 31$$

$$T_{200} = 660 + 4 \times 31 = 784$$

Vector Stride

- To overcome access to nonadjacent elements in memory
- Example:

```
      do 10 i = 1,100
         do 10 j = 1,100
            A(i,j) = 0.0
            do 10 k = 1,100
10             A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

- This loop can be strip-mined as a vector multiplication
- Each row of B would be first operand and each column of C would be second operand
- For memory organization as column major order, B's elements would be non-adjacent
- Stride is distance(uniform) between the non-adjacent elements.
- Allows access of nonsequential memory elements

Vector processors - Contd.

Agenda

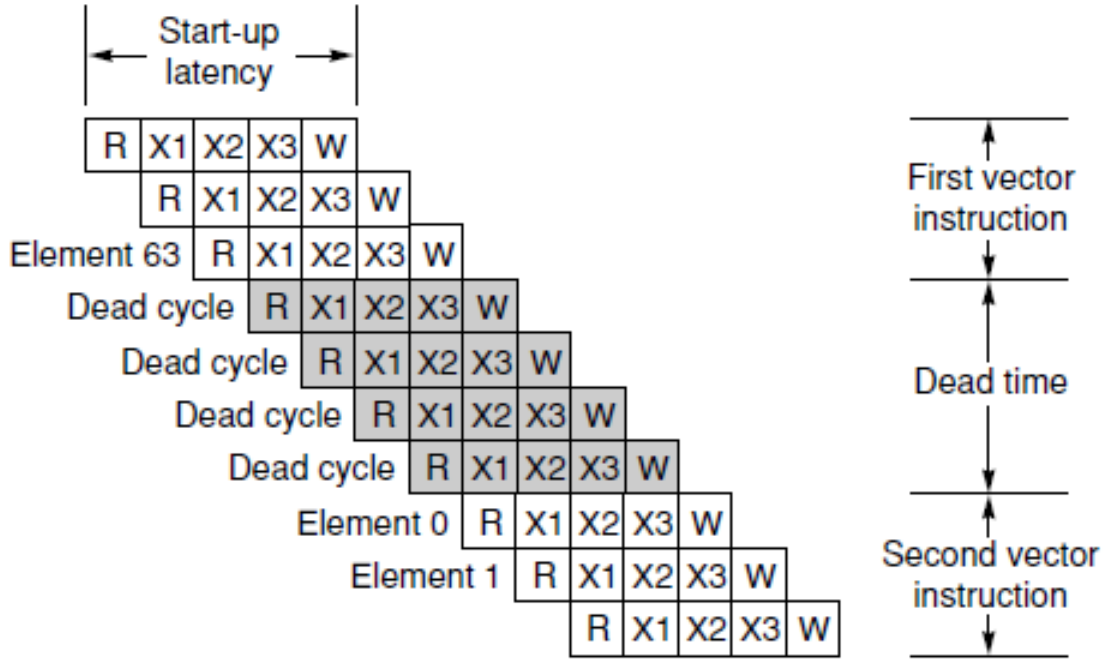
- Enhancing Vector performance
- Measuring Vector performance
- SSE Instruction set and Applications
- A case study - Intel Larrabee vector processor
- Pitfalls and Fallacies

Enhancing Vector performance

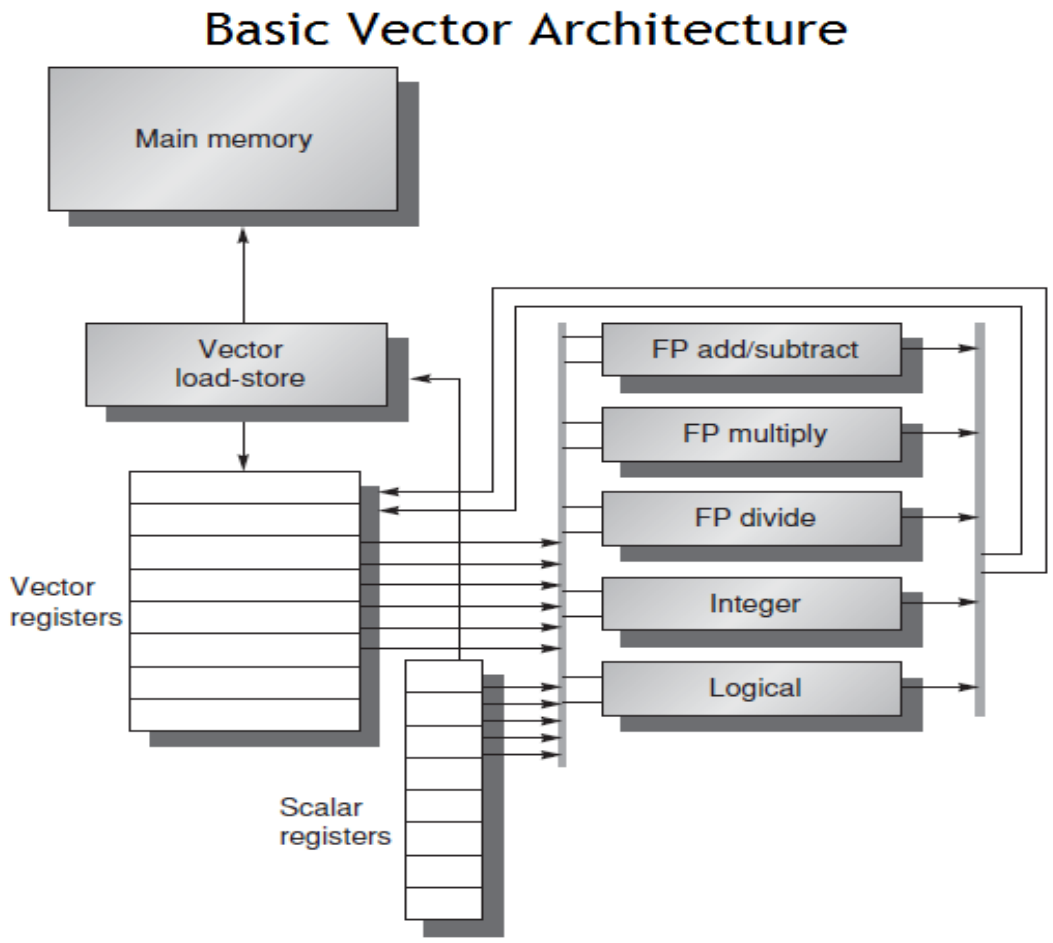
- General
  - Pipelining individual operations of one instruction
  - Reducing Startup latency
- Addressing following hazards effectively
  - Structural hazards
  - Data hazards
  - Control hazards

Pipelining & reducing Startup latency

```
ADDV.D V1,V2,V3
ADDV.D V4,V5,V6
```

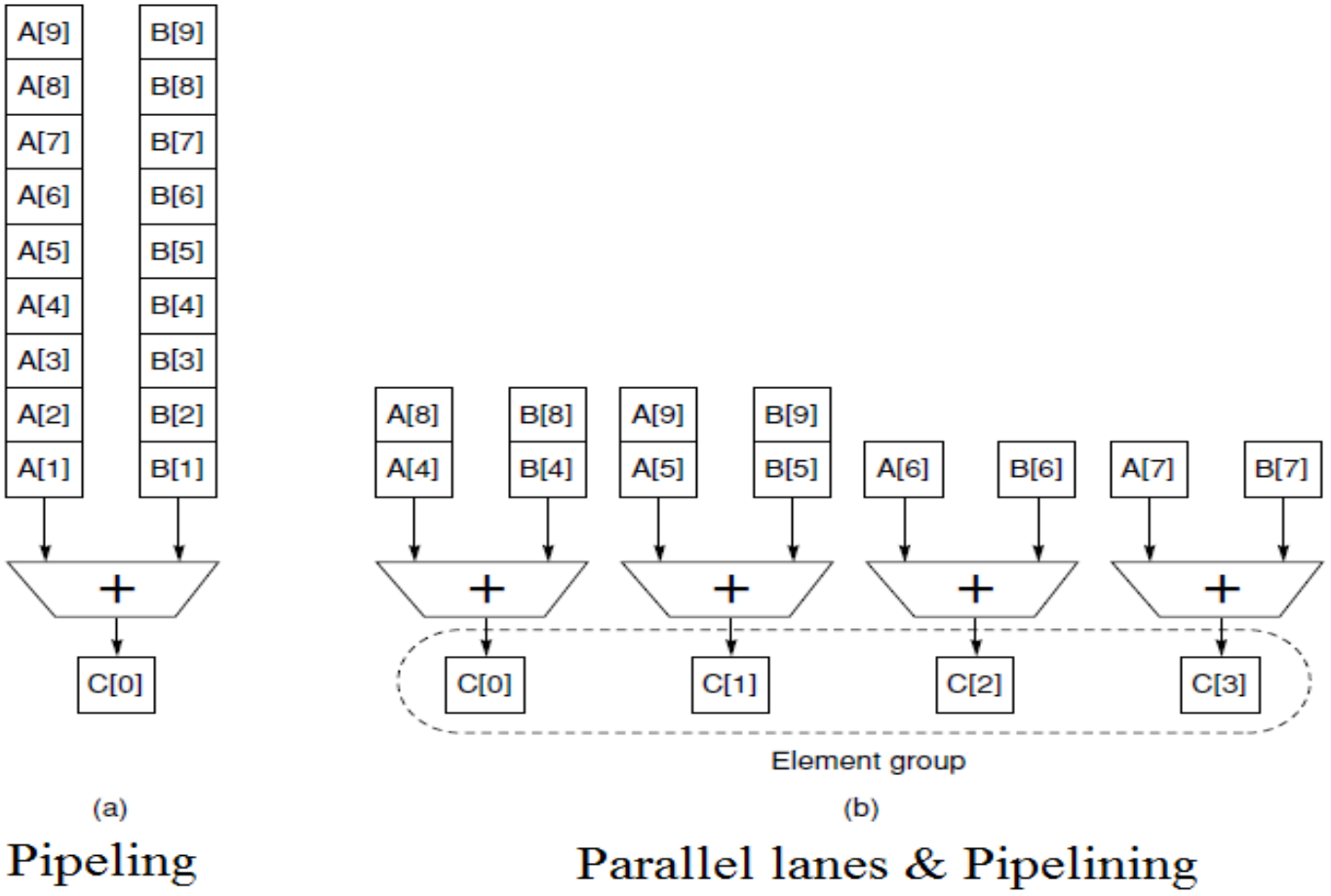


Addressing Structural hazards - Multiple Lanes



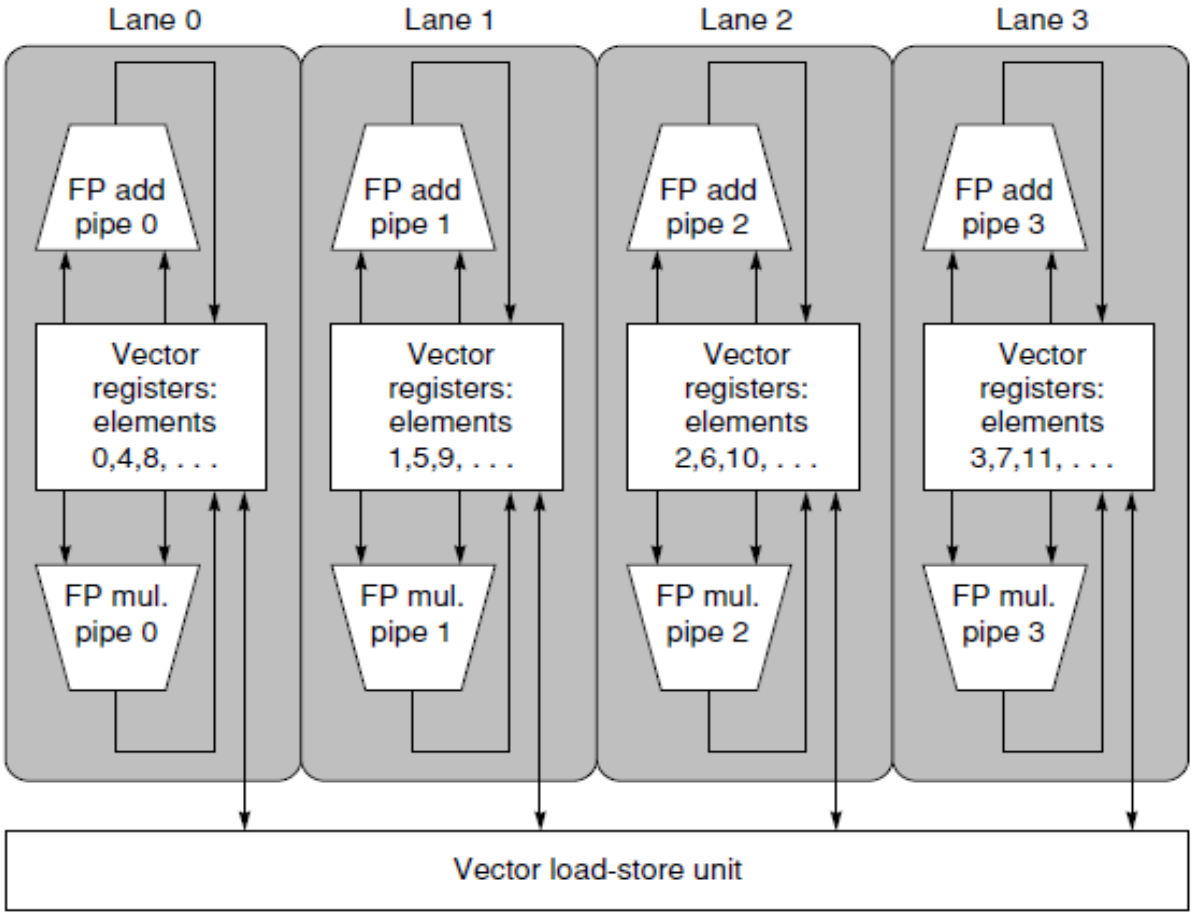
Addressing Structural hazards - Multiple Lanes

- Addressed using pipelining and parallel lanes



Multiple Lanes - Contd.

- Registers & Floating point units are localized within lanes

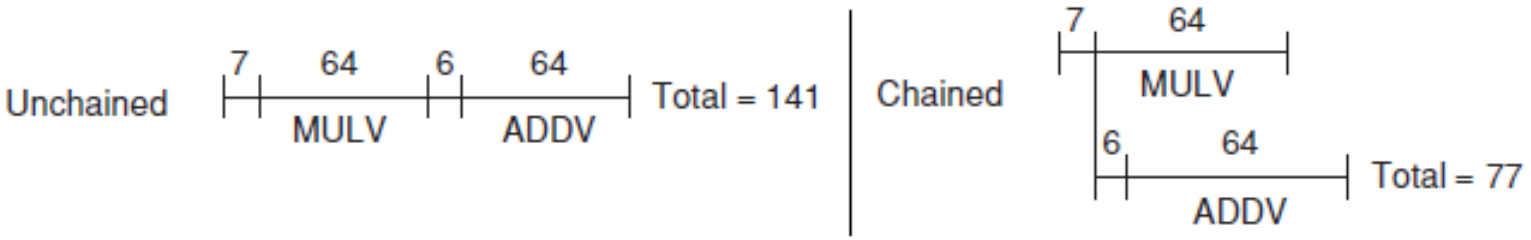


Addressing Data hazards - Flexible chaining

- Similar to Forwarding
- Chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available
- Example:

Instruction	Startup time (cycles)	Vector length (units)
MULV.D V1, V2, V3	7	64
ADDV.D V4, V1, V5	6	64

Flexible Chaining - Contd.



MULV.D V1, V2, V3 *ADDV.D V4, V1, V5*	Unchained	Chained
Time (cycles)	VLM + VLA +STM + STA = 141	VLM/A + STM + STA = 77
cycles / result	141 / 64 = 2.2	77 / 64 = 1.2
FLOPS / clock cycle	128 / 141 = 0.9	128 / 77 = 1.7

Addressing Control hazards - Vector mask

- Instructions involving control statement can't run in vector mode
- Solution:
  - Convert control dependence into data dependence by executing control statement and updating vector mask register
  - Run data dependent instructions in vector mode based on value in value mask register

Vector mask - Contd.

```
      do 100 i = 1, 64
            if (A(i).ne. 0) then
                  A(i) = A(i) – B(i)
            endif
100         continue
```

```
LV          V1,Ra         ;load vector A into V1
LV          V2,Rb         ;load vector B
L.D         F0,#0         ;load FP zero into F0
SNEVS.D     V1,F0         ;sets VM(i) to 1 if V1(i)!=F0
SUBV.D      V1,V1,V2      ;subtract under vector mask
CVM                       ;set the vector mask to all 1s
SV          Ra,V1         ;store the result in A
```

Total time = 5n + c1

Improving Vector mask - Scatter & Gather method

- Step 1: Set VM to 1 based on control condition
- Step 2: Create CVI - Create Vector Index based on VM
  - Create an index vector which points to addresses of valid contents
- Step 3: LVI - Load Vector Index (GATHER)
  - Load valid operands based on step 2
- Step 4: Execute arithemetic operation on compressed vector
- Step 5: SVI - Store Vector Index (SCATTER)
  - Store valid output based on step 2

Scatter & Gather - Contd.

```
            do 100 i = 1, 64
                   if (A(i).ne. 0) then
                          A(i) = A(i) – B(i)
                   endif
      100   continue
```

```
LV          V1,Ra         ;load vector A into V1
L.D         F0,#0         ;load FP zero into F0
SNEVS.D     V1,F0         ;sets the VM to 1 if V1(i)!=F0
CVI         V2,#8         ;generates indices in V2
POP         R1,VM         ;find the number of 1's in VM
MTC1        VLR,R1        ;load vector-length register
CVM                       ;clears the mask
LVI         V3,(Ra+V2)    ;load the nonzero A elements
LVI         V4,(Rb+V2)    ;load corresponding B elements
SUBV.D      V3,V3,V4      ;do the subtract
SVI         (Ra+V2),V3    ;store A back
```

Time = 4n + 4fn + c2

Comparison - Basic vector mask & Scatter - Gather

$$\text{Time}_1 = 5(n)$$

$$\text{Time}_2 = 4n + 4fn$$

We want $\text{Time}_1 > \text{Time}_2$, so

$$5n > 4n + 4fn$$

$$\frac{1}{4} > f$$

- Conclusion: Scatter & Gather will run faster if less than one-quarter of elements are non zero

Enhancing Vector performance - Summary

- General
  - Pipelining individual operations of one instruction
  - Reducing Startup latency
- Structural hazards
  - Multiple Lanes
- Data hazards
  - Flexible chaining
- Control hazards
  - Basic vector mask
  - Scatter & Gather

Measuring Vector Performance - Total execution time

Scale for measuring performance:

- Total execution time of the vector loop - Tn
  - Used to compare performance of different instructions on processor

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

  - Unit - clock cycles
  - n - vector length
  - MVL - maximum vector length
  - Tloop - Loop overhead
  - Tstart - startup overhead
  - Tchime - unit of convoys

Measuring Vector Performance - MFLOPS

- MFLOPS - Millions of FLoating point Operations Per Second
  - Used to compare performance of two different processors
- MFLOPS - Rn

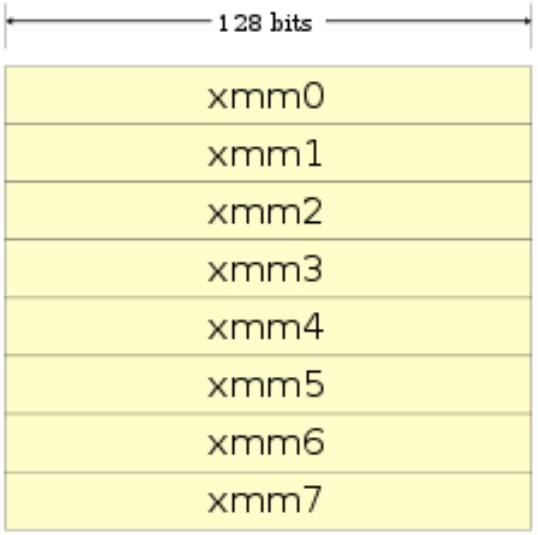
$$Rn = \frac{\text{Number of FLOPS per iteration (operation per iteration) * clock rate (cycles per second)}}{\text{(Tn / n) (cycles per iteration)}}$$

- MFLOPS - Rinfinity (theoritical / peak performance)

$$R_{\infty} = \lim_{n \to \infty} \left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

SSE Instructions

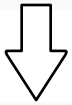
- Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture
- Streaming SIMD Extensions are similar to vector instructions.
- SSE originally added eight new 128-bit registers known as XMM0 through XMM7
- Each register packs together:
  - four 32-bit single - precision floating point numbers or
  - two 64-bit double - precision floating point numbers or
  - two 64-bit integers or
  - four 32-bit integers or
  - eight 16-bit short integers or
  - sixteen 8-bit bytes or characters.



SSE Instruction set & Applications

- Sample instruction set for floating point operations
  - Scalar – ADDSS, SUBSS, MULSS, DIVSS
  - Packed – ADDPS, SUBPS, MULPS, DIVPS
- Example

```
vec_res.x = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```



- ```
  movaps xmm0,address-of-v1          ;xmm0=v1.w | v1.z | v1.y | v1.x
  addps xmm0,address-of-v2           ;xmm0=v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x
  movaps address-of-vec_res,xmm0
  ```
  ```

A Case study - Intel Larrabee Architecture

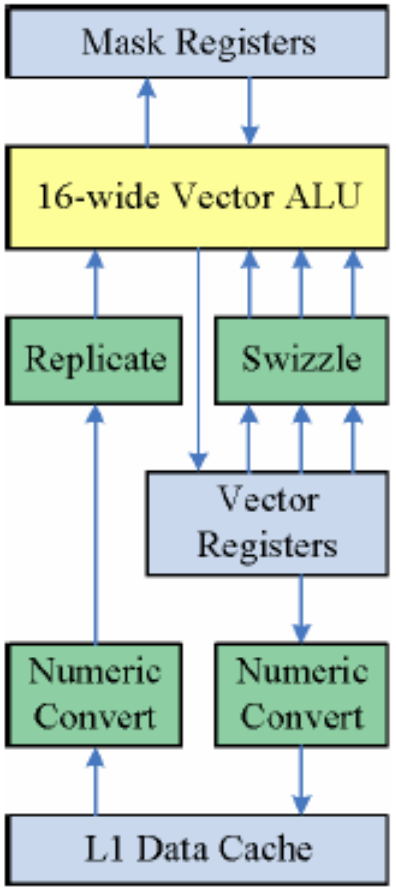
- a many-core visual computing architecture code
- Intel's new approach to a GPU
- Considered to be a hybrid between a multi-core CPU and a GPU
- Combines functions of a multi-core CPU with the functions of a GPU

Larrabee - The Big picture

- in order execution (Execution is also more deterministic so instruction and task scheduling can be done by the compiler)
- Each Larrabee core contains a 512-bit vector processing unit, able to process 16 single precision floating point numbers at a time.
- uses extended x86 architecture set with additional features like scatter / gather instructions and a mask register designed to make using the vector unit easier and more efficient.

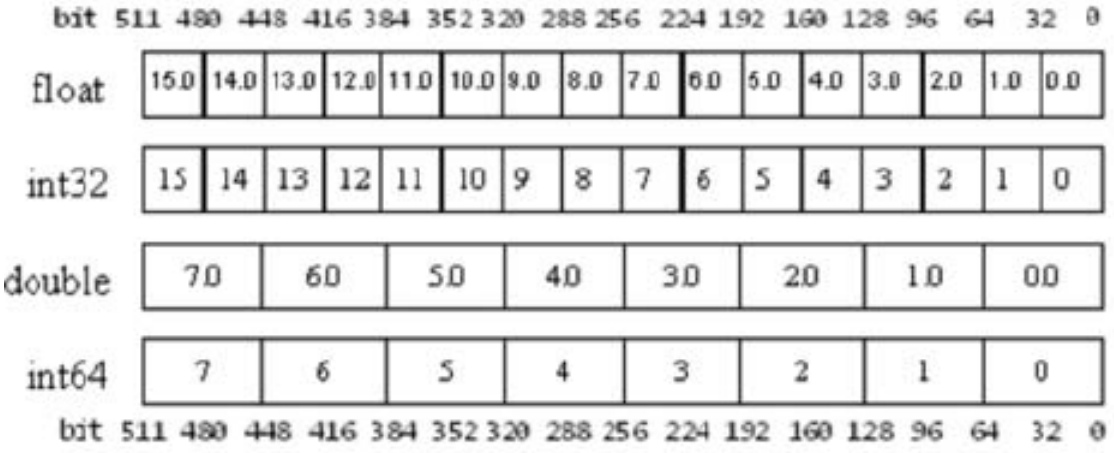
Larrabee VPU Architecture

- 16 wide vector ALU in one core
- executes interger, single precision, float and double precision float instructions
- choice of 16 - Tradeoff between increased computational density and difficulty of high utilization with wider one
- suports swizzling and replication
- Mask register and index register operations



Larrabee Data types

- 32 512-bit vector registers & 8 16-bit vector mask registers
- Each element of vector register can be
  - 8 wide - to store 16 float 32's or 16 int 32's
  - 16 wide - to store 8 float 64's or 8 int 64's



Larrabee Instruction set

- vector arithmetic, logic and shift
- vector mask generation
- vector load / store
- swizzling

> Vector multiply - add, multiply - sub instructions

```
vmadd132p{ds}: v1 = (v1 * v3) + v2
vmadd213p{ds}: v1 = (v2 * v1) + v3
vmadd231p{dis}: v1 = (v2 * v3) + v1
vmadd233p{is}: v1 = (v2 * ExtractScaleElement (v3)) + ExtractOffsetElement (v3)
vmaddn132p{ds}: v1 = -((v1 * v3) + v2)
vmaddn213p{ds}: v1 = -((v2 * v1) + v3)
vmaddn231p{ds}: v1 = -((v2 * v3) + v1)
vmsub132p{ds}: v1 = (v1 * v3) – v2
vmsub213p{ds}: v1 = (v2 * v1) – v3
vmsub231p{ds}: v1 = (v2 * v3) – v1
vmsubr132p{ds}: v1 = v2 - (v1 * v3)
vmsubr213p{ds}: v1 = v3 - (v2 * v1)
vmsubr231p{ds}: v1 = v1 - (v2 * v3)
vmsubr23c1p{ds}: v1 = 1.0 - (v2 * v3)
```

Past, Present & Future of Vector processors

- Past
  - Cray X1
  - Earth simulator
- Present
  - Cray Jaguar
  - Larrabee
- Future: AVE (Advanced Vector Extensions)
  - Sandy Bridge (Intel)
  - Bulldozer (AMD)

Pitfalls and Fallacies

- Pitfalls:
  - Concentrating on peak performance and ignoring start up overhead (on memory-memory vector architecture)
  - Increasing Vector performance, without comparable increase in scalar performance
- Fallacy
  - You can get vector performance without providing memory bandwidth (by reusing vector registers)

Recap

- Why Vector processors
- Basic Vector Architecture
- Vector Execution time
- Vector load - store units and Vector memory systems
- Vector length - VLR
- Vector stride
- Enhancing Vector performance
- Measuring Vector performance
- SSE Instruction set and Applications
- A case study - Intel Larrabee vector processor
- Pitfalls and Fallacies

References

- Computer Architecture - A quantitative approach 4th edition (Appendix A, F & G, chapter 2 & 3)
- Cray X1 http://www.supercomp.org/sc2003/paperpdfs/pap183.pdf
- Larrabee official page on intel http://software.intel.com/en-us/articles/larrabee/
- Larrabee http://www.gpucomputing.org/drdobbs_042909_final.pdf
- http://www.vizworld.com/2009/05/new-whitepapers-from-intel-about-larrabee/

Thank you.