Outline

- Properties of Vector Processing
- Components of a Vector Processor
- Vector Execution Time
- Real-world Problems: Vector Length and Stride
- Vector Optimizations: Chaining, Conditional Execution, Sparse Matrices

Why Vector Processors?

- Instruction level parallelism (Ch 3&4)
  - Deeper pipeline and wider superscalar machines to extract more parallelism
  - More register file ports, more registers, more hazard interlock logic
  - In dynamically scheduled machines instruction window, reorder buffer, rename register files must grow to have enough capacity to keep relevant information about in-flight instructions
- Difficult to build machines supporting large number of in-flight instructions => limit the issue width and pipeline depths => limit the amount parallelism you can extract
- Commercial versions long before ILP machines

Vector Processing Definitions

- Vector - a set of scalar data items, all of the same type, stored in memory
- Vector processor - an ensemble of hardware resources, including vector registers, functional pipelines, processing elements, and register counters for performing vector operations
- Vector processing occurs when arithmetic or logical operations are applied to vectors
Properties of Vector Processors

1) Single vector instruction specifies lots of work
equivalent to executing an entire loop
fewer instructions to fetch and decode
2) Computation of each result in the vector is independent of
the computation of other results in the same vector
deep pipeline without data hazards; high clock rate
3) Hw checks for data hazards only between vector
instructions (once per vector, not per vector element)
4) Access memory with known pattern
elements are all adjacent in memory => highly interleaved memory banks provides high bandw.
access is initiated for entire vector => high memory latency is amortised (no data caches are needed)
5) Control hazards from the loop branches are reduced
nonexistent for one vector instruction

Properties of Vector Processors (cont’d)

Vector operations: arithmetic (add, sub, mul, div),
memory accesses, effective address calculations
Multiple vector instructions can be in progress
at the same time => more parallelism
Applications to benefit
• Large scientific and engineering applications
(car crash simulations, whether forecasting, …)
• Multimedia applications

Basic Vector Architectures

Vector processor: ordinary pipelined scalar unit + vector unit
Types of vector processors
• Memory-memory: processors: all vector operations are
memory-to-memory (CDC)
• Vector-register: processors: all vector operations
except load and store are among the vector registers
(CRAY-1, CRAY-2, X-MP, Y-MP, NEX SX/2(3), Fujitsu)
  VMIPS – Vector processor as
  an extension of the 5-stage MIPS processor

Components of a vector-register processor

Vector Registers: each vector register
is a fixed length bank holding a single vector
has at least 2 read and 1 write ports
typically 8-32 vector registers, each holding 64-128 64 bit
elements
VMIPS: 8 vector registers, each holding 64 elements
(16 Rd ports, 8 Wr ports)
Vector Functional Units (FUs): fully pipelined,
start new operation every clock
typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X),
integer add, logical, shift;
may have multiple of same unit
VMIPS: 5 FUs (FP add/sub, FP mul, FP div, FP integer, FP logical)
Components of a vector-register processor (cont’d)

- **Vector Load-Store Units (LSUs)**
  - fully pipelined unit to load or store a vector; may have multiple LSUs
  - VMIPS: 1 VLSU, bandwidth is 1 word per cycle after initial delay
- **Scalar registers**
  - single element for FP scalar or address
  - VMIPS: 32 GPR, 32 FPRs they are read out and latched at one input of the FUs
- **Cross-bar** to connect FUs, LSUs, registers
- **Cross-bar** to connect Rd/Wr ports and FUs

VMIPS: Basic Structure

- 8 64-element vector registers
- 5 FUs; each unit is fully pipelined, can start a new operation on every clock cycle
- Load/store unit - fully pipelined
- Scalar registers

VMIPS Vector Instructions

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Operands</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV.D</td>
<td>V1,V2,V3</td>
<td>V1=V2+V3</td>
<td>vector + vector</td>
</tr>
<tr>
<td>ADDSV.D</td>
<td>V1,F0,V2</td>
<td>V1=F0+V2</td>
<td>scalar + vector</td>
</tr>
<tr>
<td>MULV.D</td>
<td>V1,V2,V3</td>
<td>V1=V2xV3</td>
<td>vector x vector</td>
</tr>
<tr>
<td>MVLSV.D</td>
<td>V1,F0,V2</td>
<td>V1=F0xV2</td>
<td>scalar x vector</td>
</tr>
<tr>
<td>LV</td>
<td>V1,R1</td>
<td>V1=M[R1..R1+63] load, stride=1</td>
<td></td>
</tr>
<tr>
<td>LVMS</td>
<td>V1,R1,R2</td>
<td>V1=M[R1..R1+63*R2] load, stride=R2</td>
<td></td>
</tr>
<tr>
<td>LIV</td>
<td>V1,R1,R2</td>
<td>V1=M[R1+V2(i),i=0..63] indir.(“gather”)</td>
<td></td>
</tr>
<tr>
<td>CeqV.D</td>
<td>VM,V1,V2</td>
<td>VMASKi = (V1i=V2i)? comp. setmask</td>
<td></td>
</tr>
<tr>
<td>MTC1</td>
<td>VLR,R1</td>
<td>Vec. Len. Reg. = R1</td>
<td>set vector length</td>
</tr>
<tr>
<td>MFC1</td>
<td>VM,R1</td>
<td>R1 = Vec. Mask</td>
<td>set vector mask</td>
</tr>
</tbody>
</table>

See table G3 for the VMIPS vector instructions.

VMIPS Vector Instructions (cont’d)

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Operands</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBV.D</td>
<td>V1,V2,V3</td>
<td>V1=V2-V3</td>
<td>vector - vector</td>
</tr>
<tr>
<td>SUBSV.D</td>
<td>V1,F0,V2</td>
<td>V1=F0-V2</td>
<td>scalar - vector</td>
</tr>
<tr>
<td>SUBV.S.D</td>
<td>V1,V2,F0</td>
<td>V1=V2-F0</td>
<td>vector - scalar</td>
</tr>
<tr>
<td>DIVV.D</td>
<td>V1,V2,V3</td>
<td>V1=V2/V3</td>
<td>vector / vector</td>
</tr>
<tr>
<td>DIVSV.D</td>
<td>V1,F0,V2</td>
<td>V1=F0/V2</td>
<td>scalar / vector</td>
</tr>
<tr>
<td>DIVVS.D</td>
<td>V1,V2,F0</td>
<td>V1=V2/F0</td>
<td>vector / scalar</td>
</tr>
<tr>
<td>POP</td>
<td>R1, M</td>
<td>Count the 1s in the VM register</td>
<td></td>
</tr>
<tr>
<td>CVM</td>
<td></td>
<td>Set the vector-mask register to all 1s</td>
<td></td>
</tr>
</tbody>
</table>

See table G3 for the VMIPS vector instructions.
**DAXPY: Double \( aX + Y \)**

Assuming vectors \( X, Y \) are length 64

Scalar vs. Vector

<table>
<thead>
<tr>
<th>Operation</th>
<th>Scalar Code</th>
<th>Vector Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD ( F0.a )</td>
<td>load scalar ( a )</td>
<td>( \text{L} )D ( V1,Rx ) load vector ( X )</td>
</tr>
<tr>
<td>LV ( V2,V1,F0 )</td>
<td>vector-scalar mult.</td>
<td>( \text{MULVS} ) ( V2,V1,F0 )</td>
</tr>
<tr>
<td>LV ( V3,Ry )</td>
<td>load vector ( Y )</td>
<td>( \text{LV} ) ( V3,Ry )</td>
</tr>
<tr>
<td>ADDV.D ( V4,V2,V3 )</td>
<td>add</td>
<td>( \text{ADDV.D} ) ( V4,V2,V3 )</td>
</tr>
<tr>
<td>SD ( Ry, V4 )</td>
<td>store the result</td>
<td>( \text{SV} ) ( Ry,V4 )</td>
</tr>
</tbody>
</table>

L.D \( F0,a \)
DADDIU \( R4,Rx,#512 \); last address to load

**Loop:**

- \( \text{L.D} \) \( F2,0(Rx) \) load \( X(i) \)
- \( \text{MULT.D} \) \( F2,F0,F2 \)
- \( \text{ADD.D} \) \( F4,F2,F4 \)
- \( \text{DSUBU} \) \( R20,R4,Rx \) compute bound
- \( \text{DADDIU} \) \( R4,Rx,#8 \) increment index to \( X \)
- \( \text{DADDIU} \) \( R4,Ry,#8 \) increment index to \( Y \)
- BNEZ \( R20,\text{loop} \) check if done

**Assuming vectors \( X, Y \) are length 64**

**Vector Execution Time**

- **Time** = (vector length, data dependencies, struct. hazards)
- **Initiation rate**: rate at which a FU consumes vector elements (= number of lanes; usually 1 or 2 on Cray T-90)
- **Convoy**: set of vector instructions that can begin execution in same clock (no struct. or data hazards)
- **Chime**: approx. time to execute a convoy
  - \( m \) convoys take \( m \) chimes; if each vector length is \( n \), then they take \( m \times n \) clock cycles (ignores overhead; good approximation for long vectors)

**VMIPS Start-up Time**

- **Start-up time**: pipeline latency time (depth of FU pipeline); another sources of overhead

<table>
<thead>
<tr>
<th>Operation</th>
<th>Start-up penalty (from CRAY-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/store</td>
<td>12</td>
</tr>
<tr>
<td>Multiply</td>
<td>7</td>
</tr>
<tr>
<td>Add</td>
<td>6</td>
</tr>
</tbody>
</table>

Assume convoys don’t overlap; vector length = \( n \):

<table>
<thead>
<tr>
<th>Convoy</th>
<th>Start</th>
<th>1st result</th>
<th>Last result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( \text{LV} )</td>
<td>( 0 )</td>
<td>12</td>
<td>12 \times (12-1\times n)</td>
</tr>
<tr>
<td>2: ( \text{MULVS.D, LV} )</td>
<td>( 12\times n )</td>
<td>12\times (n+12)</td>
<td>23\times n + load start-up</td>
</tr>
<tr>
<td>3: ( \text{ADDV.D} )</td>
<td>( 24\times 2\times n )</td>
<td>24\times (n+6)</td>
<td>29\times n + wait convoy 2</td>
</tr>
<tr>
<td>4: ( \text{SV} )</td>
<td>( 30\times 3\times n )</td>
<td>30\times 3\times (n+12)</td>
<td>41\times n + wait convoy 3</td>
</tr>
</tbody>
</table>

**VMIPS Execution Time**

1: \( \text{LV} \) \( V1,Rx \)
2: \( \text{MULVS} \) \( V5,F0,V1 \)
3: \( \text{ADDV.D} \) \( V4,V5,V3 \)
4: \( \text{SV} \) \( Ry,V4 \)
Vector Load/Store Units & Memories

- Start-up overheads usually longer for LSUs
- Memory system must sustain
  \( (# \text{ lanes} \times \text{word}) / \text{clock cycle} \)
- Many Vector Procs. use banks (vs. simple interleaving):
  - support multiple loads/stores per cycle
    \( \Rightarrow \) multiple banks & address banks independently
  - support non-sequential accesses
- Note: No. memory banks > memory latency to avoid stalls
  - \( m \) banks \( \Rightarrow m \) words per memory latency \( l \) clocks
    if \( m < l \), then gap in memory pipeline:
  - may have 1024 banks in SRAM

Real-World Issues: Vector Length

- What to do when vector length is not exactly 64?

  \[
  \text{for}(i=0; i<n, i++)
  \{ \text{Y}(i)=a*\text{X}(i)+\text{Y}(i) \}
  \]

- N can be unknown at compile time?
- Vector-Length Register (VLR): controls the length of any vector operation, including a vector load or store (cannot be > the length of vector registers)
- What if \( n > \text{Max. Vector Length (MVL)} \)?
  \( \Rightarrow \) Strip mining

Strip Mining

- Strip mining: generation of code such that each vector operation is done for a size to the MVL
- 1st loop do short piece \( (n \mod \text{MVL}) \), rest \( \text{VL} = \text{MVL} \)

  \[
  \begin{align*}
  i & = 0; \\
  \text{VL} & = n \mod \text{MVL}; \\
  \text{for} \ (j=0; j<n/\text{MVL}; j++)
  & \{ \\
  & \text{for}(i<\text{VL}; i++)
  & \{ \text{Y}(i)=a*\text{X}(i)+\text{Y}(i) \}
  & \text{VL} = \text{MVL}; \\
  & \} \\
  \end{align*}
  \]

- Overhead of executing strip-mined loop?

Vector Stride

- Suppose adjacent elements not sequential in memory (e.g., matrix multiplication)

  \[
  \begin{align*}
  \text{for}(i=0; i<n; i++)
  & \{ \\
  & \text{for}(j=0; j<n/\text{MVL}; j++)
  & \{ \\
  & \text{A}(i,j)=0.0; \\
  & \text{for}(k=0; k<n; k++)
  & \{ \\
  & \text{A}(i,j)=\text{A}(i,j)+\text{B}(i,k)\cdot\text{C}(k,j); \\
  & \} \\
  & \} \\
  \} \\
  \end{align*}
  \]

- Matrix C accesses are not adjacent (800 bytes between)
- Stride: distance separating elements that are to be merged into a single vector
  \( \Rightarrow \) LVWS (load vector with stride) instruction
- Strides can cause bank conflicts (e.g., stride=32 and 16 banks)
## Vector Opt #1: Chaining

- Suppose:
  - MUL.D V1, V2, V3
  - ADDV.D V4, V1, V5 ; separate convoy?
- Chaining: vector register (V1) is not as a single entity but as a group of individual registers, then pipeline forwarding can work on individual elements of a vector
- Flexible chaining: allow vector to chain to any other active vector operation => more read/write port
- As long as enough HW, increases convoy size

### DAXPY Chaining: CRAY-1

- CRAY-1 has one memory access pipe either for load or store (not for both at the same time)
- 3 chains
  - Chain 1: LV V3
  - Chain 2: LV V1 + MULV V2,F0,V1 + ADDV V4,V2,V3
  - Chain 3: SV V4

### DAXPY Chaining: CRAY X-MP

- CRAY X-MP has 3 memory access pipes, two for vector load and one for vector store
- 1 chain: LV V3, LV V1 + MULV V2,F0,V1 + ADDV V4,V2,V3 + SV V4
One Chain DAXPY for CRAY X-MP

Memory Access pipe
R port
W port

Access pipe
V1
V2
V3
V4

Multiply pipe
F0

Add pipe

Vector Opt #2: Conditional Execution

Consider:

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

- **Vector-mask control** takes a Boolean vector: when vector-mask register is loaded from vector test, vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1
- Requires clock even for the elements where the mask is 0
- Some VP use vector mask only to disable the storing of the result and the operation still occurs; zero division exception is possible? => mask operation

Vector Opt #3: Sparse Matrices

Sparse matrix: elements of a vector are usually stored in some compacted form and then accessed indirectly

Suppose:

```
do 100 i = 1, n
  A(K(i)) = A(K(i)) + C(M(i))
100 continue
```

- Mechanism to support sparse matrices: scatter-gather operations
- Gather (LVI) 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 => a nonsparse vector in a vector register
- After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a scatter store (SVI), using the same index vector
Sparse Matrices Example

```plaintext
   do 100 i = 1, n
100  A(K(i)) = A(K(i)) + C(M(i))
```

- Can’t be done by compiler since can’t know Ki elements distinct

```plaintext
   LV Vk, Rx  ; load K
   LVI Va,(Ra+Vk)  ; load A(K(i))
   LV Vm,Rm  ; load M
   LVI Vc,(Rc+Vm)  ; load C(M(i))
   ADDV.D Va,Va,Vc ; add them
   SVI (Ra+Vk),Va ; store A(K(i))
```

Sparse Matrices Example (cont’d)

```plaintext
   LV V1,Ra  ; load A into V1
   L.D F0,#0 ; load FP zero into F0
   SNESV.D F0,V1 ; sets VM to 1 if V1(i)<>F0
   CVI V2,#8 ; generates indices in V2
   POP R1,VM  ; find the number of 1s
   MTC1 VLR,R1 ; load vector-length reg.
   CVM  ; clears the mask
   LVI V3,(Ra+V2) ; load the nonzero As
   LVI V4,(Rb+V2) ; load the nonzero Bs
   SUBV.D V3,V3,V4 ; do the subtract
   SVX (Ra+V2),V3 ; store A back
```

- Use CVI to create index 0, 1xm, …. 63xm (compressed index vector whose entries correspond to the positions with a 1 in the mask register)

Things to Remember

- Properties of vector processing
  - Each result independent of previous result
  - Vector instructions access memory with known pattern
  - Reduces branches and branch problems in pipelines
    - Single vector instruction implies lots of work (loop)
  - Components of a vector processor: vector registers, functional units, load/store, crossbar....
- Strip mining technique for long vectors
- Optimisation techniques: chaining, conditional execution, sparse matrices