CPE 631 Lecture 24:
Vector Processing

Aleksandar Milenković, milenka@ece.uah.edu
Electrical and Computer Engineering
University of Alabama in Huntsville

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 bandwidth
   - 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, weather 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 cycle
  - 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>MULSV.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]</td>
<td>load, stride=1</td>
</tr>
<tr>
<td>LVWS</td>
<td>V1,R1,R2</td>
<td>V1=M[R1..R1+63*R2]</td>
<td>load, stride=R2</td>
</tr>
<tr>
<td>LVI</td>
<td>V1,R1,V2</td>
<td>V1=M[R1+V2(i),i=0..63] indir.(&quot;gather&quot;)</td>
<td></td>
</tr>
<tr>
<td>SeqV.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>SUBVS.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 $a \times X + Y$

Assuming vectors $X$, $Y$ are length 64

Scalar vs. Vector

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,a</td>
<td>load scalar $a$</td>
</tr>
<tr>
<td>LV V1,Rx</td>
<td>load vector $X$</td>
</tr>
<tr>
<td>MULVS V2,V1,F0</td>
<td>vector-scalar mult.</td>
</tr>
<tr>
<td>LV V3,Ry</td>
<td>load vector $Y$</td>
</tr>
<tr>
<td>ADDV.D V4,V2,V3</td>
<td>add</td>
</tr>
<tr>
<td>S.V Ry,V4</td>
<td>store the result</td>
</tr>
</tbody>
</table>

Operations: $578 \ (2+9 \times 64) \ vs. \ 321 \ (1+5 \times 64) \ (1.8 \times)$

Instructions: $578 \ (2+9 \times 64) \ vs. \ 6 \ instructions \ (96 \times)$

Hazards: 64X fewer pipeline hazards

---

Vector Execution Time

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

1. **LV V1,Rx**: load vector $X$
2. **MULVS D V2,V1,F0**: vector-scalar mult.
3. **ADDV.D V4,V2,V3**: add
4. **SV Ry,V4**: store the result

4 convoys, 1 lane, $VL=64$

=> $4 \times 64 = 256$ clocks (or 4 clocks per result)
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>Vector load/store</td>
<td>12</td>
</tr>
<tr>
<td>Vector multiply</td>
<td>7</td>
</tr>
<tr>
<td>Vector 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. LV</td>
<td>0</td>
<td>12</td>
<td>11+n (12-1+n)</td>
</tr>
<tr>
<td>2. MULVS.D, LV</td>
<td>12+n</td>
<td>12+n+12</td>
<td>23+2n wait start-up</td>
</tr>
<tr>
<td>3. ADDV.D</td>
<td>24+2n</td>
<td>24+2n+6</td>
<td>29+3n wait convoy 2</td>
</tr>
<tr>
<td>4. SV</td>
<td>30+3n</td>
<td>30+3n+12</td>
<td>41+4n wait convoy 3</td>
</tr>
</tbody>
</table>

VMIPS Execution Time

1: LV $v_1, r_x$
2: MULV $v_2, f_0, v_1$
   LV $v_3, r_y$
3: ADDV $v_4, v_2, v_3$
4: SV $r_y, v_4$

Time

11/04/2005 UAH-CPE631
Vector Load/Store Units & Memories

- Start-up overheads usually longer for LSUs
- Memory system must sustain 
  (# lanes x word) /clock cycle
- Many Vector Processors use banks (vs. simple interleaving):
  - support multiple loads/stores per cycle
    => multiple banks & address banks independently
  - support non-sequential accesses
- Note: No. memory banks > memory latency to avoid stalls
  - m banks => 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?

```c
for(i=0; i<n, i++)
{ Y(i)=a*X(i)+Y(i)
}
```

- Value of 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 > Maximum Vector Length (MVL)?
  => Strip mining
Strip Mining

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

```
  i = 0;
  VL = n \mod \text{MVL};
  for (j=0; j<n/\text{MVL}; j++) {
    for (i<VL; i++)
      \{ Y(i)=a*X(i)+Y(i) \}
    VL = \text{MVL};
  }
```

- Overhead of executing strip-mined loop?

Vector Stride

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

```
  for (i=0; i<100; i++)
    for (j=0; j<100; j++) {
      A(i,j)=0.0;
      for (k=0; k<100; k++)
        A(i,j)=A(i,j)+B(i,k)*C(k,j);
    }
```

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

- Suppose:
  MULV.D V1,V2,V3
  ADDV.D V4,V1,V5 ; separate convoy?
- Chaining: if vector register (V1) is not treated 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 ports
- 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

![Diagram of chains and time]
CPE 631 ©AM

**3 Chains DAXPY for CRAY-1**

```
Memory R/W port
V3:
Access pipe
```
```
Memory R/W port
V1:
Access pipe
```
```
Multiply pipe
V2:
```
```
```
V3:
```
```
Add pipe
V4:
```
```
```
V4:
```
```
```
Memory R/W port
V4:
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
`
One Chain DAXPY for CRAY X-MP

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 Mask Control

LV V1, Ra ; load A into V1
LV V2, Rb ; load B into V2
L.D F0, #0 ; load FP zero to F0
SNESV.D F0,V1 ; sets VM register if V1(i)<0
SUBV.D V1,V1,V2 ; subtract under VM
CVM ; set VM to all 1s
SV Ra,V1 ; store results in A

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
  100 A(K(i)) = A(K(i)) + C(M(i))
  ```

- 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

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

```
LV Vk, Rk ; load K
LVI Va, (Ra+Vk) ; load A(K(i))
LV Vm, Rm ; load M
LVI Vc, (Rc+Vm) ; load C(M(i))
ADVD V a, Va, Vc ; add them
SVI (Ra+Vk), Va ; store A(K(i))
```

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

Sparse Matrices Example (cont’d)

```
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
MTCl 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
SVI (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)
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