CSE502 – Computer Architecture

Lecture 17
Vector Computers

Chapter 4 and Appendix G of H&P CAQA 5th Ed.

Dr. Ilchul Yoon (ickeyoon@sunykorea.ac.kr)
Slides adapted from:
Larry Wittie, SBU,
Krste Asanovic of UCB and David Patterson of UCB

Outline

- Vector Processing Overview
- Vector Metrics, Terms
- Greater Efficiency than SuperScalar Processors
- Examples
  - CRAY-1 (1976, 1979) 1st vector-register supercomputer
  - Multimedia extensions to high-performance PC processors
  - Modern multi-vector-processor supercomputer – NEC ESS
- Design Features of Vector Supercomputers
- Conclusions

- Next Reading Assignment: MultiProcessors

Vector Programming Model

Vector Code Example

# Scalar Code
LI    R4, 64
loop:
L.D   F0, 0(R1)
L.D   F2, 0(R2)
ADD.D F4, F2, F0
S.D   F4, 0(R3)
DADDIU R1, 8
DADDIU R2, 8
DADDIU R3, 8
DSUBIU R4, 8
BNEZ   R4, loop

# C code
for (i=0; i<64; i++)
C[i] = A[i] + B[i];

# Vector Code
LI     VLR, 64
LV     V1, R1
ADDV.D V3, V1, V2
SV     V3, R3

# Vector Code
LI     VLR, 64
LV     V1, R1
ADDV.D V3, V1, V2
SV     V3, R3

# Scalar Code
LI     R4, 64
loop:
L.D   F0, 0(R1)
L.D   F2, 0(R2)
ADD.D F4, F2, F0
S.D   F4, 0(R3)
DADDIU R1, 8
DADDIU R2, 8
DADDIU R3, 8
DSUBIU R4, 8
BNEZ   R4, loop
Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)

$$V_3 \leftarrow v_1 \times v_2$$

Six stage multiply pipeline

Vector Instruction Set Advantages

- Compact
  - one short instruction encodes N operations => N*FLOp
- Bandwidth
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in the same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - OR access memory in a known pattern (strided load/store)
- Scalable
  - can run same object code on more parallel pipelines or lanes

Properties of Vector Processors

- Each result is independent of previous result
  => long pipeline, compiler ensures no dependencies
  => high clock rate
- Vector instructions access memory with known pattern
  => highly interleaved memory
  => amortize memory latency of 64-plus elements
  => no (data) caches required! (but use instruction cache)
- Reduces branches and branch problems in pipelines
- Single vector instruction implies lots of work (≈ loop)
  => fewer instruction fetches

Operation & Instruction Counts: RISC vs. Vector Processor

<table>
<thead>
<tr>
<th>Program</th>
<th>RISC Operations (Millions)</th>
<th>Vector Operations (Millions)</th>
<th>R / V</th>
<th>RISC Instructions (M)</th>
<th>Vector Instructions (M)</th>
<th>R / V</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim256</td>
<td>115</td>
<td>95</td>
<td>1.1x</td>
<td>115</td>
<td>0.8</td>
<td>142x</td>
</tr>
<tr>
<td>hydro2d</td>
<td>58</td>
<td>40</td>
<td>1.4x</td>
<td>58</td>
<td>0.8</td>
<td>71x</td>
</tr>
<tr>
<td>nasa7</td>
<td>69</td>
<td>41</td>
<td>1.7x</td>
<td>69</td>
<td>2.2</td>
<td>31x</td>
</tr>
<tr>
<td>su2cor</td>
<td>51</td>
<td>35</td>
<td>1.4x</td>
<td>51</td>
<td>1.8</td>
<td>29x</td>
</tr>
<tr>
<td>tomcatv</td>
<td>15</td>
<td>10</td>
<td>1.4x</td>
<td>15</td>
<td>1.3</td>
<td>11x</td>
</tr>
<tr>
<td>wave5</td>
<td>27</td>
<td>25</td>
<td>1.1x</td>
<td>27</td>
<td>7.2</td>
<td>4x</td>
</tr>
<tr>
<td>mdljdp2</td>
<td>32</td>
<td>52</td>
<td>0.6x</td>
<td>32</td>
<td>15.8</td>
<td>2x</td>
</tr>
</tbody>
</table>

(from F. Quintana, U. Barcelona)

Vector reduces ops by 1.2X, instructions by 41X
Common Vector Metrics

- **R_∞**: MFLOPS rate on an infinite-length vector
  - vector “speed of light”
  - Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger
  - (R_n is the MFLOPS rate for a vector of length n)
- **N_{1/2}**: The vector length needed to reach one-half of R_∞
  - a good measure of the impact of start-up
- **N_V**: Minimum vector length for vector mode faster than scalar mode
  - measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit

Vector Execution Time

- **Time** = \( f(\text{vector length, data dependencies, structural hazards}) \)
- **Initiation rate**: rate that FU consumes vector elements
  (= number of lanes; usually 1 or 2 on Cray T-90)
- **Convoy**: set of vector instructions that can begin execution on same clock (if no structural or data hazards)
- **Chime**: unit of time taken to execute one convoy
  - \( m \) convoys take \( m \) chimes; if each vector length is \( n \), then they take approx. \( m \times n \) clock cycles if no chaining (assume one lane; ignores overhead; good approximation for long vectors) and as little as \( m + n - 1 \) cycles, if fully chained.

Memory Operations

- **Load/store operations move groups of data between registers and memory**
- **Three types of addressing**
  - **Unit stride**
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - **Non-unit (constant) stride**
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth (Duncan Lawrie patent)
  - **Indexed (gather-scatter)**
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize

Interleaved Memory Layout

- **Great for unit stride:**
  - Contiguous elements in different DRAMs
  - Startup time for vector operation is latency of single read
- **What about non-unit stride?**
  - Banks above are good for strides that are relatively prime to 8
  - Bad for: 2, 4
  - Better: prime number of banks...!
How Get Full Bandwidth if Unit Stride?

- Memory system must sustain (# lanes x word) / clock
- *Num. memory banks > memory latency to avoid stalls*
  - M banks ⇒ M words per memory latency L in clocks
  - if M < L, then “gap” in memory pipeline:
    - clock: 0 ... L L+1 L+2 ... L+M-1 L+M ... 2L
    - word:  - ... 0 1 2 ... M-1 - ... M
- may have 1024 banks in SRAM
- If desired throughput is greater than one word per cycle
  - Either more banks (and start multiple requests simultaneously)
  - Or wider DRAMS. Only good for unit stride or large data types
- More banks & weird (prime) numbers of banks good to support more strides at full bandwidth

Vectors Are Inexpensive

**Superscalar**
- N ops per cycle
  - ⇒ O(N^2) circuitry
- HP PA-8000
  - 4-way issue
  - reorder buffer alone: 850K transistors
    - incl. 6,720 5-bit register number comparators
**Vector**
- N ops per cycle
  - ⇒ O(N + n^2) circuitry
- UCB-T0 Integer vector µP
  - 24 ops per cycle
  - 730K transistors total
    - only 23 5-bit register number comparators
  - Integer, no floating point

Vectors Lower Power

**Single-issue Scalar**
- One instruction fetch, decode, dispatch per operation
- Arbitrary register accesses, adds area and power
- Loop unrolling and software pipelining for high performance increases instruction cache footprint
- All data passes through cache; waste power if no temporal locality
- One TLB lookup per load or store
- Off-chip access is via whole cache lines
**Vector**
- One instruction fetch, decode, dispatch per vector
- Structured register accesses
- Smaller code for high performance, less power in instruction cache misses
  - Bypass cache
- One TLB lookup per group of loads or stores
  - Move only necessary data across chip boundary

Superscalar Energy Efficiency Even Worse

**Superscalar**
- Control logic grows quadratically with issue width (non hazard checks)
- Control logic consumes energy regardless of available parallelism
- Speculation to increase visible parallelism wastes energy
**Vector**
- Control logic grows linearly with issue width
- Vector unit switches off when not in use
- Vector instructions expose parallelism without speculation
- Software control of speculation when desired:
  - Whether to use vector mask or compress/expand for conditionals
Older Vector Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>Clock</th>
<th>Regs</th>
<th>Elements</th>
<th>FUs</th>
<th>LSUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray 1</td>
<td>1976</td>
<td>80 MHz</td>
<td>8</td>
<td>64</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Cray XMP</td>
<td>1983</td>
<td>120 MHz</td>
<td>8</td>
<td>64</td>
<td>2 L, 1 S</td>
<td></td>
</tr>
<tr>
<td>Cray YMP</td>
<td>1988</td>
<td>166 MHz</td>
<td>8</td>
<td>64</td>
<td>2 L, 1 S</td>
<td></td>
</tr>
<tr>
<td>Cray C-90</td>
<td>1991</td>
<td>240 MHz</td>
<td>8</td>
<td>128</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Cray T-90</td>
<td>1996</td>
<td>455 MHz</td>
<td>8</td>
<td>128</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Convex C-1</td>
<td>1984</td>
<td>10 MHz</td>
<td>8</td>
<td>128</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Convex C-4</td>
<td>1994</td>
<td>133 MHz</td>
<td>16</td>
<td>128</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Fuji. VP200</td>
<td>1982</td>
<td>133 MHz</td>
<td>8-256</td>
<td>32-1024</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Fuji. VP300</td>
<td>1996</td>
<td>100 MHz</td>
<td>8-256</td>
<td>32-1024</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>NEC SX/2</td>
<td>1984</td>
<td>160 MHz</td>
<td>8+8K</td>
<td>256+var</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>NEC SX/3</td>
<td>1995</td>
<td>400 MHz</td>
<td>8+8K</td>
<td>256+var</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

Supercomputers

- “Definitions” of a supercomputer
- Fastest machine in world at the given task
- Any computer costing more than $30M
- Any 1966-89 machine designed by Seymour Cray
  - (Cray, born 1925, died in a 1996 Pike’s Peak wreck.)
- A device to turn a compute-bound problem into an I/O-bound problem :-)

- The Control Data CDC6600 (designer: Cray, 1964) is regarded to be the first supercomputer.

- In 1966-89, Supercomputer = Vector Machine

Vector Supercomputers

- Epitomized by Cray-1, 1976 (from icy Minnesota):
  - Scalar Unit + Vector Extensions
  - Load/Store Architecture*
  - Vector Registers
  - Vector Instructions
  - Hardwired Control
  - Highly Pipelined Functional Units*
  - Interleaved Memory System
  - No Data Caches
  - No Virtual Memory
  - 1976 80 M FLOp/sec (1979 160 MFlops)
    (2008 IBM BlueGene: 120,000,000 MFLOPS)
  - * 2 features of modern instruction-pipeline CPUs

Cray-1 (1976)
### Vector Memory System

- **Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency**
  - **bank busy time**: cycles between accesses to same bank

![Diagram of Vector Memory System](image)

### Vector Memory-Memory versus Vector Register Machines

- **Vector memory-memory architectures (VMMA)** require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
  - VMMA make it difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
  - VMMA incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar break even point was around 2 elements
  - Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures
    (we ignore vector memory-memory from now on)

### Example Source Code

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

### VMIPS Double-Precision Vector Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV D</td>
<td>FL2, FL3</td>
<td>Add element of FL2 and FL3, then put result in FL3</td>
</tr>
<tr>
<td>ADDV D</td>
<td>FL2, FL3, FV</td>
<td>Add FV to each element of FL2, then put result in FL3</td>
</tr>
<tr>
<td>SUBV D</td>
<td>FL2, FL3</td>
<td>Subtract each element of FL2 from element of FL3, then put result in FL3</td>
</tr>
<tr>
<td>SUBV D</td>
<td>FL2, FL3, FV</td>
<td>Subtract FV from each element of FL2, then put result in FL3</td>
</tr>
<tr>
<td>MULVD</td>
<td>FL2, FL3</td>
<td>Multiply each element of FL2 and FL3, then put result in FL3</td>
</tr>
<tr>
<td>DIVV D</td>
<td>FL2, FL3</td>
<td>Divide each element of FL2 by element of FL3, then put result in FL3</td>
</tr>
<tr>
<td>DIVV D</td>
<td>FL2, FL3, FV</td>
<td>Divide each element of FL2 by element of FL3, then put result in FL3</td>
</tr>
<tr>
<td>LV</td>
<td>FL3</td>
<td>Load element from memory starting at address FL3</td>
</tr>
<tr>
<td>LV</td>
<td>FL3, FV</td>
<td>Load element from memory starting at address FL3, then put result in FV</td>
</tr>
<tr>
<td>LV</td>
<td>FL3, FV, FV</td>
<td>Load element from memory starting at address FL3, then put result in FV, FV</td>
</tr>
</tbody>
</table>

![Diagram of VMIPS Double-Precision Vector Instructions](image)

### VMIPS Double-Precision Vector Code

- **Example VMIPS Double-Precision Vector Code**
  - "ADDV D, FL2, FL3"
  - "SUBV D, FL2, FL3"
  - "DIVV D, FL2, FL3"

- **Example Source Code**
  - "ADDV C, A, B"
  - "SUBV D, A, B"
  - "DIVV D, A, B"

### VMIPS Double-Precision Vector Code

- **Example VMIPS Double-Precision Vector Code**
  - "ADDV C, A, B"
  - "SUBV D, A, B"
  - "DIVV D, A, B"

- **Example Source Code**
  - "ADDV C, A, B"
  - "SUBV D, A, B"
  - "DIVV D, A, B"

- CMOS Technology
  - Each 500 MHz CPU fits on single chip
  - SDRAM main memory (up to 64GB)
- Scalar unit in each CPU
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache
- Vector unit in each CPU
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 1 multiply unit, 1 divide unit, 1 add/shift unit, 1 logical unit, 1 mask unit
  - 8 lanes (8 GFLOPS peak, 16 FLOPS/cycle)
  - 32 GB/s memory bandwidth per processor
- SMP (Symmetric Multi-Processor) structure
  - 8 CPUs connected to memory through crossbar
  - 256 GB/s shared memory bandwidth (4096 interleaved banks)

NEC ESS EarthSimSys (2002)

General Purpose Supercomputer – Configuration

- Processor Nodes (PN): Total number of processor nodes is 640. Each processor node consists of eight vector processors of 8 GFLOPS and 16GB shared memories. Therefore, total numbers of processors is 5,120 and total peak performance and main memory of the system are 40 TFLOPS (= 8*8*640 GFLOPS) and 10 TB (= 640*16GB), respectively. Two nodes are installed into one cabinet, which size is 40" x 56" x 80". 16 nodes are in a cluster. Power consumption per cabinet is approximately 20 KVA [total power for all 320 cabinets is 6.4 MW (megawatts)].
- Interconnection Network (IN): The nodes are coupled together with more than 83,000 copper cables via single-stage crossbar switches of 16GB/s x 2 (Load + Store). The total length of the cables is approximately 3,000 km.
- Hard Disk: Raid disks are used for the system. The capacities are 450 TB for the systems operations and 250 TB for users.
- Mass Storage system: 12 Automatic Cartridge Systems (STK PowderHorn9310); total storage capacity is approximately 1.6 PetaBytes (PB).
- Fastest computer in world, 2002-04. {NY Blue at SBU/BNL: 100+ TF}

From Horst D. Simon, NERSC/LBNL, 15May02, "ESS Rapid Response Meeting"
Recent Multimedia Extensions for PCs

- Very short vectors added to existing ISAs for micros
- Usually 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit registers (Altivec, SSE2)
  - Pentium 4 SSE2: Streaming SIMD Extensions 2

- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary

- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure

- Trend towards fuller vector support in microprocessors
**Automatic Code Vectorization**

for (i=0; i < N; i++)
C[i] = A[i] + B[i];

**Vector Stripmining**

- Problem: Vector registers have finite length (64)
- Solution: Break longer (than 64) loops into pieces that fit into vector registers, “Stripmining”

for (i=0; i<N; i++)
C[i] = A[i]+B[i];

Vector Instruction Parallelism

- Chain to overlap execution of multiple vector instructions
- Example machine has 32 elements per vector register and 8 lanes

**Vector Chaining**

- Vector version of register bypassing
- First in revised Cray-1 ’79, Rpeak: 80 MFlops in ’76 => 160 MFlops in ’79
Vector Chaining Advantage

- 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

Vector Startup

- Two components of vector startup penalty
  - functional unit latency (time through pipeline)
  - dead time or recovery time (time before another vector instruction can start down pipeline)

Dead Time and Short Vectors

- UC-B T0, Eight lanes
  - No dead time
  - 100% efficiency with 8 element (integer) vectors

- Cray C90, two lanes,
  - 4 cycle dead time,
  - Maximum efficiency 94% (64/68) with 128 element vectors

Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

\[
\text{for } (i=0; i<N; i++) \\
\quad A[i] = B[i] + C[D[i]]
\]

Indexed load instruction (Gather)

- \( \text{LV } v_D, r_D \) # Load indices in D vector
- \( \text{LVI } v_C, (r_C+v_D) \) # Load indirect from rC base
- \( \text{LV } v_B, r_B \) # Load B vector
- \( \text{ADDV.D } v_A, v_B, v_C \) # Do add
- \( \text{SV } v_A, r_A \) # Store result
Vector Scatter/Gather

Scatter example:

\[
\text{for (i=0; i<N; i++)} \\
\text{A[B[i]]++;} \\
\]

Is this code a correct translation? No!!

LV vB, rB  # Load indices in B vector
LVI vA, (rA+vB)  # Gather initial A values
ADDV vA, vA, F1  # Increase A values by F1=1.
SVI vA, (rA+vB)  # Scatter incremented values

Vector Conditional Execution

- Problem: Want to vectorize loops with conditional code:
  
  \[
  \text{for (i=0; i<N; i++)} \\
  \text{if (A[i]>0) then} \\
  \text{A[i] = B[i];} \\
  \]

- Solution: Add vector mask (or flag) registers
- vector version of predicate registers, 1 bit per element
- …and maskable vector instructions
- vector operation becomes NOOP at elements where mask bit is clear (0)

- Code example:
  
  CVM  # Turn on all elements
  LV vA, rA  # Load entire A vector
  SGTVS.D vA, F0  # Set bits in mask register where A>0
  LV vA, rB  # Load B vector into A under mask
  SV vA, rA  # Store A back to memory under mask

Vector Scatter/Gather

Scatter example:

\[
\text{for (i=0; i<N; i++)} \\
\text{A[B[i]]++;} \\
\]

Is this code a correct translation? Now it is!!

DADDI F1, F0, 1  # Integer 1 in F1
CVT.W.D F1, F1  # Convert 32-bit 1 → double 1.0
LV vB, rB  # Load indices in B vector
LVI vA, (rA+vB)  # Gather initial A values
ADDVS vA, vA, F1  # Increase A values by F1=1.
SVI vA, (rA+vB)  # Scatter incremented values

Masked Vector Instruction Implementations

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

Write data port

Write Disable
Compress/Expand Operations

- **Compresses** (packs) non-masked elements from one vector register contiguously at start of destination vector register
  - population count of mask vector gives packed vector length
- Expand performs inverse operation

![Compress/Expand Diagram](image)

Used for density-time conditionals and for general selection operations

Vector Reductions (vector values → one result)

- **Problem**: Loop-carried dependence on reduction variables
  
  \[
  \text{sum} = 0; \\
  \text{for} \ i=0; \ i<N; \ i++ \ \\
  \text{sum} += A[i]; \quad \# \text{loop-carried dependence on sum}
  \]

- **Solution**: Re-associate operations if possible, use binary tree to perform reduction
  
  # Rearrange as:
  \[
  \text{sum}[0:VL-1] = 0 \quad \# \text{vector of VL partial sums} \\
  \text{for}(i=0; \ i<N; \ i+=VL) \ \\
  \text{sum}[0:VL-1] += A[i:i+VL-1]; \quad \# \text{vector sum} \\
  \]  
  # Now have VL partial sums in one vector register
  
  do {
  \[
  \text{VL} = \text{VL}/2; \quad \# \text{halve vector length} \\
  \text{sum}[0:VL-1] += \text{sum}[\text{VL}:2*VL-1] \quad \# \text{halve no. of partials} \\
  \]  
  while (VL>1)

Vector Summary

- 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
  - Especially with virtual address translation and caching
- Will multimedia popularity revive vector architectures?

And in Conclusion [Vector Processing]

- One instruction operates on vectors of data
- Vector loads get data from memory into big register files, operate, and then vector store
- Have indexed load, store for sparse matrices
- Easy to add vectors to commodity instruction sets
  - E.g., Morph SIMD into vector processing
- Vector is a very efficient architecture for vectorizable codes, including multimedia and many scientific matrix applications