CSE 502 Graduate Computer Architecture

Lec 13-15 – Vector Computers

Larry Wittie
Computer Science, StonyBrook University

http://www.cs.sunysb.edu/~cse502 and ~lw

Slides adapted from
Krste Asanovic of MIT and David Patterson of UCB, UC-Berkeley cs252-s06
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: Chapter 4 MultiProcessors
Vector Programming Model

**Scalar Registers**
- r15
- r0

**Vector Registers**
- v15
- v0

**Vector Load and Store Instructions**
- LV v1, r1, r2

**Vector Arithmetic Instructions**
- ADDV v3, v1, v2

**Vector Length Register**
- VLR

**Vector Register**
## Vector Code Example

### C code

```c
for (i=0; i<64; i++)
    C[i] = A[i] + B[i];
```

### 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, 1
    BNEZ   R4, loop
```

### Vector Code

```
LI     VLR, 64
LV     V1, R1
LV     V2, R2
ADDV.D V3, V1, V2
SV     V3, R3
```
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 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>Operations (Millions)</th>
<th>Instructions (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RISC</td>
<td>Vector</td>
</tr>
<tr>
<td>swim256</td>
<td>115</td>
<td>95</td>
</tr>
<tr>
<td>hydro2d</td>
<td>58</td>
<td>40</td>
</tr>
<tr>
<td>nasa7</td>
<td>69</td>
<td>41</td>
</tr>
<tr>
<td>su2cor</td>
<td>51</td>
<td>35</td>
</tr>
<tr>
<td>tomcatv</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>wave5</td>
<td>27</td>
<td>25</td>
</tr>
<tr>
<td>mdljdp2</td>
<td>32</td>
<td>52</td>
</tr>
</tbody>
</table>

(from F. Quintana, U. Barcelona)

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

• $R_\infty$: 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 $R_\infty$
  – 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(vector length, data dependencies, struct. 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**: approximate time for a vector operation
- **m convoys take m chimes**: if each vector length is n, then they take approx. $m \times n$ clock cycles if no chaining (ignores overhead; good approximation for long vectors) and as little as $m + n - 1$ cycles, if fully chained.

4 convoys, 1 lane, VL=64

=> $4 \times 64 = 256$ clocks
(or 4 clocks per result)

1: LV V1,Rx ;load vector X
2: MULV V2,F0,V1 ;vector-scalar mult.
   LV V3,Ry ;load vector Y
3: ADDV V4,V2,V3 ;add
4: SV Ry,V4 ;store the result
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 $\Rightarrow M$ words per memory latency $L$ in clocks
  – if $M < L$, then “gap” in memory pipeline:
    
    \[
    \begin{array}{cccccccc}
    \text{clock:} & 0 & \ldots & L & L+1 & L+2 & \ldots & L+M-1 & L+M & \ldots & 2L \\
    \text{word:} & -- & \ldots & 0 & 1 & 2 & \ldots & M-1 & -- & \ldots & M \\
    \end{array}
    \]

  – may have 1024 banks in SRAM

• If desired throughput 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

Multiscalar
- N ops per cycle
  \( \Rightarrow O(N^2) \) 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
  \( \Rightarrow O(N + \varepsilon N^2) \) circuitry
- UCB-T0 Integer vector \( \mu P \)
  - 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 inst 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 ($n \times n$ hazard chks)
- 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
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: Chapter 4 MultiProcessors
# 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>8</td>
<td>2 L, 1 S</td>
</tr>
<tr>
<td>Cray YMP</td>
<td>1988</td>
<td>166 MHz</td>
<td>8</td>
<td>64</td>
<td>8</td>
<td>2 L, 1 S</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>

(floatling) (load/store)
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 ('79 160 MFlops) "World’s most costly warm loveseat"

(2008 SBU/BNL NY IBM BlueGene: 120,000,000 MFLOPS)

* 2 features of modern instruction-pipeline CPUs
Cray-1 (1976)

Single Ported Memory

16 banks of 64-bit words + 8-bit SECDED

Single Error Correct Double Error Detect

80MW/sec data load/store

320MW/sec instruction buffer refill

64-bitx16

4 Instruction Buffers

Current Instruction Parcel (CIP) register issues 16-bit instructions; CIP + LIP (LowerIP) => 32-bit instructions.

memory bank cycle 50 ns processor cycle 12.5 ns (80MHz)
Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- Bank busy time: Cycles between accesses to same bank

Vector Registers

Memory Banks

Address Generator

Base Stride

0 1 2 3 4 5 6 7 8 9 A B C D E F
Vector Memory-Memory versus Vector Register Machines

- Vector memory-memory instructions held all vector operands in main memory
- Only the first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
- Cray-1 ('76) was first vector register machine

Example Source Code

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

Vector Memory-Memory Code

```
ADDV C, A, B
SUBV D, A, B
```

Vector Register Code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```
Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory

- VMMAs make it difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses

- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven 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)
Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make it difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven 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)
### 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>V1, V2, V3</td>
<td>Add elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>ADDVS.D</td>
<td>V1, V2, F0</td>
<td>Add F0 to each element of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBV.D</td>
<td>V1, V2, V3</td>
<td>Subtract elements of V3 from V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBVS.D</td>
<td>V1, V2, F0</td>
<td>Subtract F0 from elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBSV.D</td>
<td>V1, F0, V2</td>
<td>Subtract elements of V2 from F0, then put each result in V1.</td>
</tr>
<tr>
<td>MULV.D</td>
<td>V1, V2, V3</td>
<td>Multiply elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>MULVS.D</td>
<td>V1, V2, F0</td>
<td>Multiply each element of V2 by F0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVV.D</td>
<td>V1, V2, V3</td>
<td>Divide elements of V2 by V3, then put each result in V1.</td>
</tr>
<tr>
<td>DIVVS.D</td>
<td>V1, V2, F0</td>
<td>Divide elements of V2 by F0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVSV.D</td>
<td>V1, F0, V2</td>
<td>Divide F0 by elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
<td>Load vector register V1 from memory starting at address R1.</td>
</tr>
<tr>
<td>SV</td>
<td>R1, V1</td>
<td>Store vector register V1 into memory starting at address R1.</td>
</tr>
<tr>
<td>LWS</td>
<td>V1, (R1, R2)</td>
<td>Load V1 from address at R1 with stride in R2, i.e., R1+i×R2.</td>
</tr>
<tr>
<td>SVS</td>
<td>(R1, R2), V1</td>
<td>Store V1 from address at R1 with stride in R2, i.e., R1+i×R2.</td>
</tr>
<tr>
<td>LVI</td>
<td>V1, (R1+V2)</td>
<td>Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.</td>
</tr>
<tr>
<td>SVI</td>
<td>(R1+V2), V1</td>
<td>Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.</td>
</tr>
<tr>
<td>CVI</td>
<td>V1, R1</td>
<td>Create an index vector by storing the values 0, 1×R1, 2×R1, ..., 63×R1 into V1.</td>
</tr>
<tr>
<td>S--V.D</td>
<td>V1, V2</td>
<td>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.</td>
</tr>
<tr>
<td>POP</td>
<td>R1, VM</td>
<td>Count the 1s in the vector-mask register and store count in R1.</td>
</tr>
<tr>
<td>CVM</td>
<td></td>
<td>Set the vector-mask register to all 1s.</td>
</tr>
<tr>
<td>MTC1</td>
<td>VLR, R1</td>
<td>Move contents of R1 to the vector-length register.</td>
</tr>
<tr>
<td>MFC1</td>
<td>R1, VLR</td>
<td>Move the contents of the vector-length register to R1.</td>
</tr>
<tr>
<td>MVTM</td>
<td>VM, F0</td>
<td>Move contents of F0 to the vector-mask register.</td>
</tr>
<tr>
<td>MVFM</td>
<td>F0, VM</td>
<td>Move contents of vector-mask register to F0.</td>
</tr>
</tbody>
</table>

---

Figure F.3 The VMIPS vector instructions. Only the double-precision FP operations are shown. In addition to the vector registers, there are two special registers, VLR (discussed in Section F.3) and VM (discussed in Section F.4). These special registers are assumed to live in the MIPS coprocessor 1 space along with the FPU registers. The operations with stride are explained in Section F.3, and the uses of the index creation and indexed load-store operations are explained in Section F.4. (From page F-8 Appendix F Vector Processors of CAQA4e)

- **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)
  - 1 load & store unit (32x8 byte accesses/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

1. 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 and 10 TB, respectively. Two nodes are installed into one cabinet, which size is 40”x56”x80”. 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)].

2) Interconnection Network (IN): The nodes are coupled together with more than 83,000 copper cables via single-stage crossbar switches of 16GB/s x2 (Load + Store). The total length of the cables is approximately 3,000 km.

3) Hard Disk. Raid disks are used for the system. The capacities are 450 TB for the systems operations and 250 TB for users.

4) Mass Storage system: 12 Automatic Cartridge Systems (STK PowderHorn9310); total storage capacity is approximately 1.6 PetaBytes (PB). From Horst D. Simon, NERSC/LBNL, 15May02, “ESS Rapid Response Meeting”

5) Fastest computer in world, 2002-04. {SBU/BNL NY Blue: 100+ TF}
NEC ESS - Earth Simulator System (ne' European Supercomputer System)
Earth Simulator Building (210 x 160ft) (Complete system installed 4/1/2002)
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
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: Chapter 4 MultiProcessors
Vector Instruction Execution

ADDV C, A, B

Execution using one pipelined functional unit


Four-lane execution using four pipelined functional units

Vector Unit Structure

Vector Registers

Elements 0, 4, 8, ...

Elements 1, 5, 9, ...

Elements 2, 6, 10, ...

Elements 3, 7, 11, ...

Functional Unit

Memory Subsystem

Lane
Automatic Code Vectorization

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

Scalar Sequential Code

Vectorized Code

Vectorization is a massive compile-time reordering of operation sequencing \(\Rightarrow\) requires extensive loop dependence analysis
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];

A

B

C

64 elements

64 elements

Remainder

Remainder

ANDI R1, N, 63  # N mod 64
MTC1 VLR, R1    # Do remainder

LV    V1, RA    # Vector load A
DSLL  R2, R1, 3 # Multiply N%64 *8
DADDU RA, RA, R2 # Bump RA pointer
LV    V2, RB    # Vector load B
DADDU RB, RB, R2 # Bump RB pointer
ADDV.D V3, V1, V2 # Vector add
SV    V3, RC    # Vector store C
DADDU RC, RC, R2 # Bump RC pointer
DSUBU N, N, R1  # R1 elements done
LI    R1, 64    # Vector length is
MTC1 VLR, R1    # Set to full 64
BGTZ  N, loop   # Any more to do?
Vector Instruction Parallelism

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

Load Unit
Multiply Unit
Add Unit

Cycle
1
2
3
4
5
6
7
8
9
10

6 issues of instruction

Complete 24 operations/cycle but issue 1 short instruction/cycle
Vector Chaining

- Vector version of register bypassing
  - First in revised Cray-1 '79, \( R_{\text{peak}} \): 80 MFlops in '76 => 160 MFlops in '79

```
LV v1
MULV v3, v1, v2
ADDV v5, v3, v4
```
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)

Functional Unit Latency

R=Read regs
X=execute
W=Write reg

Dead Time
If FU not pipelined

First Vector Instruction

Second Vector Instruction
Dead Time and Short Vectors

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

Krste Asanovic's PhD: UC-B T0, Eight lanes No dead time 100% efficiency with 8 element (integer) vectors

64 cycles active

4 cycles dead time

No dead time
Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (Gather)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV</td>
<td>vD, rD # Load indices in D vector</td>
</tr>
<tr>
<td>LVI</td>
<td>vC, (rC+vD) # Load indirect from rC base</td>
</tr>
<tr>
<td>LV</td>
<td>vB, rB # Load B vector</td>
</tr>
<tr>
<td>ADDV.D</td>
<td>vA, vB, vC # Do add</td>
</tr>
<tr>
<td>SV</td>
<td>vA, rA # Store result</td>
</tr>
</tbody>
</table>
Vector Scatter/Gather

Scatter example:

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

Is this code a correct translation?  No!

.L.

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 Scatter/Gather

 Scatter example:

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

Is this code a correct translation? Now it is!

. DADDI F1,F0,1    # Integer 1 into 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
Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

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

Solution: Add vector \textit{mask} (or \textit{flag}) registers

- vector version of predicate registers, 1 bit per element

...and \textit{maskable} vector instructions

- vector operation becomes NOOP at elements where mask bit is clear (0)

Code example:

- \text{CVM} \quad \# \text{Turn on all elements}
- \text{LV vA, rA} \quad \# \text{Load entire A vector}
- \text{SGTVS.D vA, F0} \quad \# \text{Set bits in mask register where A>0}
- \text{LV vA, rB} \quad \# \text{Load B vector into A under mask}
- \text{SV vA, rA} \quad \# \text{Store A back to memory under mask}
Masked Vector Instruction Implementations

**Simple Implementation**
- execute all N operations, turn off result writeback according to mask

\[
\begin{align*}
M[2] &= 0 \\
M[1] &= 1 \\
M[0] &= 0
\end{align*}
\]

\[\text{Write Disable} \quad \text{Write data port}\]

**Density-Time Implementation**
- scan mask vector and only execute elements with non-zero masks

\[
\begin{align*}
M[7] &= 1 \\
M[6] &= 0 \\
M[5] &= 1 \\
M[4] &= 1 \\
M[3] &= 0 \\
M[2] &= 0 \\
M[1] &= 1 \\
M[0] &= 0
\end{align*}
\]

\[\text{Write data port}\]
Compress/Expand Operations

- Compress packs non-masked elements from one vector register contiguously at start of destination vector reg.
  - Population count of mask vector gives packed vector length
- Expand performs inverse operation

Used for density-time conditionals and for general selection operations
Vector Reductions (vector values => one result)

Problem: Loop-carried dependence on reduction variables

```c
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  // Loop-carried dependence on sum
```

Solution: Re-associate operations if possible, use binary tree to perform reduction

```c
# Rearrange as:
sum[0:VL-1] = 0  // Vector of VL partial sums
for(i=0; i<N; i+=VL)  // Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1]; // Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2;  // Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1]  // 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