Lecture 15 – Vectors

John Wawrzynek
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~johnw
http://inst.eecs.berkeley.edu/~cs152
Last Time Lecture 14: Multithreading

<table>
<thead>
<tr>
<th>Time (processor cycle)</th>
<th>Superscalar</th>
<th>Fine-Grained</th>
<th>Coarse-Grained</th>
<th>Multiprocessing</th>
<th>Simultaneous Multithreading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idle slot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- Exploit ILP (instruction level parallelism)
  - Super-scalar OoO: dynamic instruction scheduling
  - VLIW: static (compile-time) instruction scheduling
- Exploit TLP (thread level parallelism)
  - fine-grained, coarse-grained threading, SMT
- Exploit DLP (data level parallelism)
  - vector architectures
Supercomputer Applications

- Typical application areas
  - Military research (nuclear weapons, cryptography)
  - Scientific research
  - Weather forecasting
  - Oil exploration
  - Industrial design (car crash simulation)
  - Bioinformatics
  - Cryptography

- All involve huge computations on large data set

- Supercomputers: CDC6600, CDC7600, Cray-1, ...

- In 70s-80s, Supercomputer ≡ Vector Machine
Vector Supercomputers

- Epitomized by Cray-1, 1976:
  - Scalar Unit
    - Load/Store Architecture
  - Vector Extension
    - Vector Registers
    - Vector Instructions
  - Implementation
    - Hardwired Control
    - Highly Pipelined Functional Units
    - Interleaved Memory System
    - No Data Caches
    - No Virtual Memory

[©Cray Research, 1976]
Fig. 1. Physical organization of mainframe.

- Dimensions
  - Base: 103.5 inches diameter by 19 inches high
  - Columns: 56.5 inches diameter by 77 inches high including height of base

- 24 chassis
- 1662 modules; 113 module types
- Each module contains up to 288 IC packages per module
- Power consumption approximately 115 kw input for maximum memory size
- Freon cooled with Freon/water heat exchange
- Three memory options
- Weight 10,500 lbs (maximum memory size)
- Three basic chip types
  - 5/4 NAND gates
  - Memory chips
  - Register chips
Vector Programming Model

Scalar Registers
- x31
- x0

Vector Registers
- v31
- v0
- [0] [1] [2] [VLMAX-1]

Vector Length Register
- vl

Vector Arithmetic Instructions
- vadd v3, v1, v2

Vector Load and Store Instructions
- vls v1, (x1), x2

Base, x1
Stride, x2
Memory

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

# Scalar Code
# li x4, 64
loop:
    fld f1, 0(x1)
    fld f2, 0(x2)
    fadd.d f3, f1, f2
    fsd f3, 0(x3)
    addi x1, x1, 8
    addi x2, x2, 8
    addi x3, x3, 8
    subi x4, x4, 1
    bnez x4, loop

# Vector Code
# li x4, 64
vsetvl x4
vld v1, (x1)
vld v2, (x2)
vadd v3, v1, v2
vst v3, (x3)
### “DLXV” Vector Instructions

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Operands</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV</td>
<td>v1, v2, v3</td>
<td>V1 = V2 + V3</td>
<td>vector + vector</td>
</tr>
<tr>
<td>ADDSV</td>
<td>v1, F0, v2</td>
<td>V1 = F0 + V2</td>
<td>scalar + vector</td>
</tr>
<tr>
<td>MULTV</td>
<td>v1, v2, v3</td>
<td>V1 = V2 × V3</td>
<td>vector × vector</td>
</tr>
<tr>
<td>MULSV</td>
<td>v1, F0, v2</td>
<td>V1 = F0 × V2</td>
<td>scalar × 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+V2i, i=0..63]</td>
<td>indir. (&quot;gather&quot;)</td>
</tr>
<tr>
<td>CeqV</td>
<td>VM, v1, v2</td>
<td>VMASKi = (V1i=V2i)?</td>
<td>comp. setmask</td>
</tr>
<tr>
<td>MOV</td>
<td>VLR, r1</td>
<td>Vec. Len. Reg. = R1</td>
<td>set vector length</td>
</tr>
<tr>
<td>MOV</td>
<td>VM, r1</td>
<td>Vec. Mask = R1</td>
<td>set vector mask</td>
</tr>
<tr>
<td>Instruction</td>
<td>Operands</td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>ADDV.D</td>
<td>V1, V2, V3</td>
<td>Add elements of V2 and V3, then put each result in V1.</td>
<td></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>
<td></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>
<td></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>
<td></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>
<td></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>
<td></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>
<td></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>
<td></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>
<td></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>
<td></td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
<td>Load vector register V1 from memory starting at address R1.</td>
<td></td>
</tr>
<tr>
<td>SV</td>
<td>R1, V1</td>
<td>Store vector register V1 into memory starting at address R1.</td>
<td></td>
</tr>
<tr>
<td>LVWS</td>
<td>V1, (R1, R2)</td>
<td>Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).</td>
<td></td>
</tr>
<tr>
<td>SVWS</td>
<td>(R1, R2), V1</td>
<td>Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).</td>
<td></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>
<td></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>
<td></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>
<td></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>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>R1, VM</td>
<td>Count the 1s in vector-mask register VM and store count in R1.</td>
<td></td>
</tr>
<tr>
<td>CVM</td>
<td></td>
<td>Set the vector-mask register to all 1s.</td>
<td></td>
</tr>
<tr>
<td>MTC1</td>
<td>VLR, R1</td>
<td>Move contents of R1 to vector-length register VL.</td>
<td></td>
</tr>
<tr>
<td>MFC1</td>
<td>R1, VLR</td>
<td>Move the contents of vector-length register VL to R1.</td>
<td></td>
</tr>
<tr>
<td>MVTM</td>
<td>VM, F0</td>
<td>Move contents of F0 to vector-mask register VM.</td>
<td></td>
</tr>
<tr>
<td>MVFM</td>
<td>F0, VM</td>
<td>Move contents of vector-mask register VM to F0.</td>
<td></td>
</tr>
</tbody>
</table>
# Example 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>Conv. 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>Conv. 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>Fuj. 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>Fuj. 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>
Vector Instruction Set Advantages

- **Compact**
  - one short instruction encodes N operations

- **Expressive, tells hardware that these N operations:**
  - are independent
  - can use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous operation
  - access a contiguous block of memory
    (unit-stride load/store)
  - access memory in a known pattern
    (strided load/store)

- **Scalable**
  - can run same code on more parallel pipelines (lanes) - trade speed for complexity
Vector Memory-Memory versus Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- 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

```assembly
vadd (C), (A), (B)
vsub (D), (A), (B)
```

Vector Register Code

```assembly
vld V1, (A)
vld V2, (B)
vadd V3, V1, V2
vst V3, (C)
vsub V4, V1, V2
vst 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 2-4 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)
Cray-1 (1976)

Single-Port Memory

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

80MW/sec data load/store

320MW/sec instruction buffer refill

memory bank cycle 50 ns  processor cycle 12.5 ns (80MHz)
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!)

```
v1  v2  v3
```

*Six-stage multiply pipeline*

```
v3 <- v1 * v2
```
Interleaved Vector Memory System

- Sequentially addressed words reside in sequential banks.
- Cray-1, 16 banks, 4 cycle bank busy time* (50ns)
- 1 word per cycle for vector load/store
- 4 words read per cycle for instructions

*Bank busy time: Time before bank ready to accept next request
Vector Chaining

- Vector version of register bypassing
  - introduced with Cray-1

\[
\begin{align*}
\text{v1d} & \rightarrow v1 \\
\text{vmul} & \rightarrow v3, v1, v2 \\
\text{vadd} & \rightarrow v5, v3, v4
\end{align*}
\]
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 Instruction Execution

vadd vc, va, vb

Execution using one pipelined functional unit


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,

Memory Subsystem
T0 Vector Microprocessor (UCB/ICSI, 1995)

Vector register elements striped over lanes
Vector Instruction Parallelism

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

Complete 24 operations/cycle while issuing 1 short instruction/cycle
CS152 Administrivia

- PS 4, Lab 3 out, due Tue March 29th

- No lectures or sections next week!
  - Spring Break (March 21-25)
CS252 Administrivia

- Summaries due, discussion begins tomorrow on Readings on Branch Prediction
- Summaries due on Cray-1, VLIW Friday April 1

- Work on project! Progress report due after spring break.
Automatic Code Vectorization

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

Vectorization is a massive compile-time reordering of operation sequencing
→ requires extensive loop-dependence analysis
Vector Stripmining

**Problem:** Vector registers have finite length

**Solution:** Break loops into pieces that fit in registers, “Stripmining”

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

and x1, xN, 63 # N mod 64
vsetvl x1     # Do remainder

loop:
vld v1, (xA)
slli x2, x1, 3 # Multiply by 8
add xA, xA, x2 # Bump pointer
vld v2, (xB)
add xB, xB, x2
vadd v3, v1, v2
vst v3, (xC)
add xC, xC, x2
sub xN, xN, x1 # Subtract elements
li x1, 64
vsetvl x1     # Reset full length
bgtz xN, loop # Any more to do?
```
Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:
   
   ```
   for (i=0; i<N; i++)
       if (A[i]>0) then
           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 bubble (“NOP”) at elements
     where mask bit is clear

Code example:

```
cvm                 # Turn on all elements
vld vA, (xA)        # Load entire A vector
vgt vA, f0          # Set bits in mask register where A>0
vld vA, (xB)        # Load B vector into A under mask
vst vA, (xA)        # Store A back to memory under mask
```
Masked Vector Instructions

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*M[7]=1*  
*M[6]=0*  
*M[5]=1*  
*M[4]=1*  
*M[3]=0*  
*M[2]=0*  
*M[1]=1*  
*M[0]=0*  

*Write Enable*  
*Write data port*

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>C[2]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>C[1]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>C[0]</td>
<td></td>
</tr>
</tbody>
</table>

*M[7]=1*  
*M[6]=0*  
*M[5]=1*  
*M[4]=1*  
*M[3]=0*  
*M[2]=0*  
*M[1]=1*  
*M[0]=0*  

*Write data port*
Vector Reductions

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. partials
} while (VL>1)
```
Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```assembly
vld vD, (xD)       # Load indices in D vector
vlx vC, (xC), vD   # Load indexed from xC base
vld vB, (xB)       # Load B vector
vadd vA,vB,vC      # Do add
vst vA, (xA)       # Store result
```
Vector Memory Models

- Some vector machines have a very relaxed memory model, e.g.
  
  ```
  vst v1, (x1)  # Store vector to x1
  vld v2, (x1)  # Load vector from x1
  ```

  - No guarantee that elements of v2 will have value of elements of v1 even when store and load execute by same processor!

- So require explicit memory barrier or fence
  
  ```
  vst v1, (x1)  # Store vector to x1
  fence         # Enforce ordering s->l
  vld v2, (x1)  # Load vector from x1
  ```

Vector machines support highly parallel memory systems (multiple lanes and multiple load and store units) with long latency (100+ clock cycles)

- hardware coherence checks would be prohibitively expensive
- vectorizing compiler can eliminate most dependencies
Packed SIMD Extensions

- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b/512b for Intel AVX
- Single instruction operates on all elements within register
Packed SIMD versus Vectors

- Traditionally, 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
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b) (latest 2022 AVX-512 includes gather/scatter)
  - ARM Scalable Vector Extensions (SVE)
Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Krste Asanovic (UCB)
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)