CS 152 Computer Architecture and Engineering

Lecture 15: Vector Computers

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

http://www.eecs.berkeley.edu/~johnw
http://inst.cs.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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
Supercomputers

- Definition of a supercomputer:
- Fastest machine in world at given task
- A device to turn a compute-bound problem into an I/O bound problem
- Any machine costing $30M+
- Any machine designed by Seymour Cray
- CDC6600 (Cray, 1963) regarded as first supercomputer
# Supercomputer

[https://www.top500.org/lists/2016/06/](https://www.top500.org/lists/2016/06/)

## TOP 10 Sites for June 2016

For more information about the sites and systems in the list, click on the links or view the complete list.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Supercomputing Center in Wuxi</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>2</td>
<td>National Super Computer Center in Guangzhou</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>3</td>
<td>DOE/SC/Oak Ridge National Laboratory</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
</tbody>
</table>
CDC 6600  *Seymour Cray, 1963*

- A fast pipelined machine with 60-bit words
  - 128 Kword main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
  - Floating Point: adder, 2 multipliers, divider
  - Integer: adder, 2 incrementers, ...
- Hardwired control (no microcoding)
- *Scoreboard* for dynamic scheduling of instructions
- Ten Peripheral Processors for Input/Output
  - a fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling
- Fastest machine in world for 5 years (until 7600)
  - over 100 sold ($7-10M each)
IBM Memo on CDC6600

Thomas Watson Jr., IBM CEO, August 1963:

“Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer.”

To which Cray replied: “It seems like Mr. Watson has answered his own question.”
CDC 6600:
A Load/Store Architecture

- Separate instructions to manipulate three types of registers:
  - 8 60-bit data registers (X)
  - 8 18-bit address registers (A)
  - 8 18-bit index registers (B)

- All arithmetic and logic instructions are register-to-register.

<table>
<thead>
<tr>
<th>opcode</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xi ← (Xj) op (Xk)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- No Explicit Load and Store instructions!

  - $Ai ← (Ri) + \text{disp} \Rightarrow Xi ← M[Ai]$, for $0<i<6$
  - $Ai ← (Ri) + \text{disp} \Rightarrow M[Ai] ← Xi$, for $5<i<8$

- Touching address registers 1 to 5 initiates a load
  - 6 to 7 initiates a store

  - very useful for vector operations
CDC6600 ISA designed to simplify high-performance implementation

- Use of three-address, **register-register** ALU instructions simplifies pipelined implementation
  - No implicit dependencies between inputs and outputs

- **Decoupling** setting of address register (Ar) from retrieving value from/to data register (Xr) simplifies providing multiple outstanding memory accesses
  - Software can schedule load of address register before use of value
  - Can interleave independent instructions in-between

- CDC6600 has multiple parallel but unpipelined functional units
  - E.g., 2 separate multipliers

- Follow-on machine CDC7600 used pipelined functional units
  - Foreshadows later RISC designs
CDC6600: Vector Addition

B0 ← - n

loop:  JZE  B0, exit

A1 ← B0 + a0  # load X1
A2 ← B0 + b0  # load X2
X6 ← X1 + X2  # load X2
A6 ← B0 + c0  # store X6
B0 ← B0 + 1

jump loop

Ai = address register
Bi = index register
Xi = data register
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 sets, double precision floats

- In 70s-80s, Supercomputer ≡ Vector Machine
Vector Programming Model

Scalar Registers

Vector Registers

Vector Length Register [VLR]

Vector Arithmetic Instructions

ADDV v3, v1, v2

Vector Load and Store Instructions

LV v1, r1, r2

Scalar Registers

r15

v15

r0

v0

Vector Registers

[0] [1] [2] [VLRMAX-1]

Vector Load and Store Instructions

LV v1, r1, r2

Base, r1

Stride, r2

Memory

Vector Register

v1

v0

v1

v2

v3

[0] [1] [VLR-1]
Vector Code Example

# C code
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
# VMIPS vector instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV.V.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.V.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.V.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.V.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>LVWS</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>SVWS</td>
<td>(R1, R2), V1</td>
<td>Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).</td>
</tr>
<tr>
<td>LV1</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--VV.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>S--VS.D</td>
<td>V1, F0</td>
<td>Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and F0. 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 vector-mask register VM 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 vector-length register VL.</td>
</tr>
<tr>
<td>MFC1</td>
<td>R1, VLR</td>
<td>Move the contents of vector-length register VL to R1.</td>
</tr>
<tr>
<td>MVTM</td>
<td>VM, F0</td>
<td>Move contents of F0 to vector-mask register VM.</td>
</tr>
<tr>
<td>MVFM*</td>
<td>F0, VM</td>
<td>Move contents of vector-mask register VM to F0.</td>
</tr>
</tbody>
</table>

**Figure 4.3** The VMIPS vector instructions, showing only the double-precision floating-point operations. In addition to the vector registers, there are two special registers, VLR and VM, discussed below. These special registers are assumed to live in the MIPS coprocessor 1 space along with the FPU registers. The operations with stride and uses of the index creation and indexed load/store operations are explained later.
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

*Higher scalar-code performance than then state-of-the-art scalar machines!*
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

64 Element Vector Registers

64 T Regs

64 B Regs

64-bitx16

4 Instruction Buffers

memory bank cycle 50 ns  processor cycle 12.5 ns (80MHz)
Vector Instruction Set Advantages

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

- **Expressive, tells hardware that these N operations:**
  - are independent
  - use the same functional unit
  - access disjoint registers
  - 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)
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!)

\[ V3 \leftarrow v1 \times v2 \]

*Six stage multiply pipeline*
Vector Instruction Execution Flexibility

ADDV C, A, B

Execution using one pipelined functional unit


Execution using four pipelined functional units

Interleaved Vector Memory System

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

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

```
  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
```

**Vector Registers**

**Address Generator**

**Memory Banks**

- **Base**
- **Stride**
Vector Unit Structure

Vector Registers

Elements 0, 4, 8, ...

Elements 1, 5, 9, ...

Elements 2, 6, 10,

Elements 3, 7, 11,

Functional Unit

Lane

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

Vector register elements striped over lanes

Lane
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

- Discussion this week moved from Friday to Thursday
Vector Chaining

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

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 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

- `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 (no reg reuse)

- 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)
Automatic Code Vectorization

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

**Scalar Sequential Code**

Iter. 1

- load
- add
- store

Iter. 2

- load
- add
- store

**Vectorized Code**

Iter. 1

- load
- add
- store

Iter. 2

- load
- add
- store

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”

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

\[ C[i] = A[i]+B[i]; \]

\[ \text{loop:} \]

\[ \text{ANID R1, N, 63 } \# \text{ N mod 64} \]

\[ \text{MTC1 VLR, R1 } \# \text{ Do remainder} \]

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

\[ C[i] = A[i]+B[i]; \]

\[ \text{loop:} \]

\[ \text{ANID R1, N, 63 } \# \text{ N mod 64} \]

\[ \text{MTC1 VLR, R1 } \# \text{ Do remainder} \]
Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```c
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:

```c
CVM            # Turn off all elements
LV vA, rA      # Load entire A vector
L.D  F0, #0    # Load FP zero into F0
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
```
Masked Vector Instructions

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

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

Write Enable

Write data port

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

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

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. of 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)

```c
LV vD, rD         # Load indices in D vector
LVI vC, rC, vD    # Load indirect from rC base
LV vB, rB         # Load B vector
ADDV.D vA,vB,vC   # Do add
SV vA, rA         # Store result
```
Histogram example:

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

Is following a correct translation?

```assembly
LV vB, rB     # Load indices in B vector
LVI vA, rA, vB # Gather initial A values
ADDV vA, vA, 1 # Increment
SVI vA, rA, vB # Scatter incremented values
```

- 65nm CMOS technology

- Vector unit (3.2 GHz)
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 64-bit functional units: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
  - 8 lanes (32+ FLOPS/cycle, 100+ GFLOPS peak per CPU)
  - 1 load or store unit (8 x 8-byte accesses/cycle)

- Scalar unit (1.6 GHz)
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache

- Memory system provides 256GB/s DRAM bandwidth per CPU
- Up to 16 CPUs and up to 1TB DRAM form shared-memory node
  - total of 4TB/s bandwidth to shared DRAM memory
- Up to 512 nodes connected via 128GB/s network links (message passing between nodes)
Multimedia Extensions (aka SIMD extensions)

<table>
<thead>
<tr>
<th>64b</th>
<th>32b</th>
<th>32b</th>
</tr>
</thead>
<tbody>
<tr>
<td>16b</td>
<td>16b</td>
<td>16b</td>
</tr>
<tr>
<td>8b</td>
<td>8b</td>
<td>8b</td>
</tr>
<tr>
<td>8b</td>
<td>8b</td>
<td>8b</td>
</tr>
<tr>
<td>8b</td>
<td>8b</td>
<td>8b</td>
</tr>
<tr>
<td>8b</td>
<td>8b</td>
<td>8b</td>
</tr>
</tbody>
</table>

- 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 for Intel AVX

- Single instruction operates on all elements within register

```
16b  16b  16b  16b
+    +    +    +
16b  16b  16b  16b
```

4x16b adds
Multimedia Extensions versus Vectors

- **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

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

- **The trend is towards fuller vector support in microprocessors**
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)
Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252