CS 152 Computer Architecture and Engineering

Lecture 15: Vector Computers

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.cs.berkeley.edu/~cs152

Last Time Lecture 14: Multithreading

3/21/2013
CS152, Spring 2013
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, 1964) regarded as first supercomputer

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 reg.
  8  60-bit data registers (X)
  8  18-bit address registers (A)
  8  18-bit index registers (B)

• All arithmetic and logic instructions are reg-to-reg

\[
\text{opcode} \begin{array}{c|c|c|c} i & j & k \end{array} \quad \text{Ri} \leftarrow (\text{Rj}) \text{ op (Rk)}
\]

• Only Load and Store instructions refer to memory!

\[
\text{opcode} \begin{array}{c|c|c} i & j & \text{disp} \end{array} \quad \text{Ri} \leftarrow \text{M[}(\text{Rj}) + \text{disp}] 
\]

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 data register (Xr) simplifies providing multiple outstanding memory accesses
  - Software can schedule load of address register before use of value
  - Can interleave independent instructions inbetween
- 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
  A0 ← B0 + a0 load X0
  A1 ← B0 + b0 load X1
  X6 ← X0 + X1
  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

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

Scalar Registers
- r15
- r0

Vector Registers
- v15
- v0
- [0] [1] [2] [VLRMAX-1]

Vector Arithmetic Instructions
- ADDV v3, v1, v2

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

Vector Length Register
- VLR

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
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

3/21/2013

CS152, Spring 2013

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 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 registers in same pattern as previous instructions
  - 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!)

\[ V_3 \leftarrow v_1 \times v_2 \]

*Six stage multiply pipeline*
Vector Instruction Execution

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
Vector Unit Structure

Functional Unit

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

Lane

3/21/2013 CS152, Spring 2013

CS152 Handout Page 10
**Vector Instruction Parallelism**

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

**Diagram:**

- Load Unit
- Multiply Unit
- Add Unit

- Complete 24 operations/cycle while issuing 1 short instruction/cycle

---

**CS152 Administrivia**

- Spring Break week
  - No classes!
Vector Chaining

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

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

- Cray C90, Two lanes
  - 4 cycle dead time
  - Maximum efficiency 94%
    with 128 element vectors

- T0, Eight lanes
  - No dead time
  - 100% efficiency with 8 element vectors
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

```
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)
Automatic Code Vectorization

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

Scalar Sequential Code

\[
\begin{align*}
\text{load} & \quad \text{load} \\
\text{add} & \quad \text{add} \\
\text{store} & \quad \text{store}
\end{align*}
\]

Vectorized Code

\[
\begin{align*}
\text{load} & \quad \text{load} \\
\text{add} & \quad \text{add} \\
\text{store} & \quad \text{store}
\end{align*}
\]

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”

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

\[
\begin{align*}
\text{ANDI R1, N, 63} & \quad \text{# N mod 64} \\
\text{MTC1 VLR, R1} & \quad \text{# Do remainder loop:} \\
\text{LV V1, RA} \\
\text{DSLL R2, R1, 3} & \quad \text{# Multiply by 8} \\
\text{DADDU RA, RA, R2} & \quad \text{# Bump pointer} \\
\text{LV V2, RB} \\
\text{DADDU RB, RB, R2} \\
\text{ADDV.D V3, V1, V2} \\
\text{SV V3, RC} \\
\text{DADDU RC, RC, R2} \\
\text{DSUBU N, N, R1} & \quad \text{# Subtract elements} \\
\text{LI R1, 64} \\
\text{MTC1 VLR, R1} & \quad \text{# Reset full length} \\
\text{BGTZ N, loop} & \quad \text{# Any more to do?}
\end{align*}
\]
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 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
```

Masked Vector Instructions

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

[Diagram of masked vector instruction example]
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

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

Indexed load instruction *(Gather)*
```
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
```
Vector Scatter/Gather

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

Is following a correct translation?
LV vB, rB  # Load indices in B vector
LVI vA, rA, vB  # Gather initial A values
ADDS 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)

- 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

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
- 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)
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