CS252 Graduate Computer Architecture
Fall 2015
Lecture 9: Vector Supercomputers

Krste Asanovic
krste@eecs.berkeley.edu
http://inst.eecs.berkeley.edu/~cs252/fa15
Last Time in Lecture 8

Overcoming the worst hazards in OoO superscalars:
- Branch prediction
- Load-Store Queues
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]
Vector Programming Model

Scalar Registers

<table>
<thead>
<tr>
<th>x31</th>
<th>x31</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>x0</td>
</tr>
</tbody>
</table>

Vector Registers

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

Vector Length Register

VLR

Vector Arithmetic Instructions

vadd v3, v1, v2

Vector Load and Store Instructions

vld v1, x1, x2

Base, x1  Stride, x2  Memory
## Vector Code Example

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

```c
# 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, 8
    addi x2, 8
    addi x3, 8
    subi x4, 1
    bnez x4, loop
```

```c
# Vector Code
li x4, 64
setvlr x4
vfld v1, x1
vfld v2, x2
vfadd.d v3,v1,v2
vfsd v3, x3
```
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

80MW/sec data load/store

320MW/sec instruction buffer refill

64 Element Vector Registers

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!)

\[
v3 \leftarrow v1 \times v2
\]
Vector Instruction Execution

vfadd.d vc, va, vb

Execution using one pipelined functional unit


C[1]

C[2]

C[0]

Execution using four pipelined functional units

A[22]  B[22]
A[27]  B[27]

C[0]

C[9]

C[1]

C[2]

C[3]

C[11]

C[7]

C[6]
Interleaved Vector Memory System

- Bank busy time: Time before bank ready to accept next request
- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency
Vector Unit Structure

- **Vector Registers**
  - Elements 0, 4, 8, ...
  - Elements 1, 5, 9, ...
  - Elements 2, 6, 10, ...
  - Elements 3, 7, 11, ...

- **Lane**

- **Functional Unit**

- **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
Vector Chaining

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

\[ \text{vld v1} \]
\[ \text{vfmul v3, v1, v2} \]
\[ \text{Vfadd v5, v3, v4} \]
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

First Vector Instruction

Dead Time

Second Vector Instruction
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

```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 2-4 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)
Automatic Code Vectorization

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

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

Vectorization is a massive compile-time reordering of operation sequencing
\[
\Rightarrow \text{requires extensive loop dependence analysis}
\]
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];
```

```
    andi x1, xN, 63  # N mod 64
    setvlr x1       # Do remainder

    loop:
        vld v1, xA
        sll x2, x1, 3  # Multiply by 8
        add xA, x2     # Bump pointer
        vld v2, xB
        add xB, x2
        vfadd.d v3, v1, v2
        vsd v3, xC
        add xC, x2
        sub xN, x1     # Subtract elements
        li x1, 64
        setvlr x1      # Reset full length
        bgtz xN, loop  # Any more to do?
```

- 64 elements
- Remainder
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
  vfsgts.d vA, f0 # Set bits in mask register where A>0
  vld vA, xB   # Load B vector into A under mask
  vsd 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

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

![Diagram of Simple Implementation](image1)
![Diagram of Density-Time Implementation](image2)
Compress/Expand Operations

- Compress 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

Used for density-time conditionals and also for general selection operations
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:

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

Indexed load instruction (\textit{Gather})

\[
\text{vld vD, xD} \quad \# \text{Load indices in D vector}
\]
\[
\text{vdli vC, xC, vD} \quad \# \text{Load indirect from } rC \text{ base}
\]
\[
\text{vld vB, xB} \quad \# \text{Load B vector}
\]
\[
\text{vfadd.d vA,vB,vC} \quad \# \text{Do add}
\]
\[
\text{vsd vA, xA} \quad \# \text{Store result}
\]
Vector Scatter/Gather

Histogram example:

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

Is following a correct translation?

```c
vld vB, xB  # Load indices in B vector
vldi vA, xA, vB  # Gather initial A values
vadd vA, vA, 1  # Increment
vsdi vA, xA, vB  # Scatter incremented values
```
Vector Memory Models

- Most vector machines have a very relaxed memory model, e.g.
  
  ```
  vsd 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!

- Requires explicit memory barrier or fence
  
  ```
  vsd v1, x1   # Store vector to x1
  fence.vs.vl  # 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

- 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

[©NEC]

- 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)

[New announcement SX-ACE, 4x16-lane vector CPUs on one chip]
Acknowledgements

This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:

- Arvind (MIT)
- Joel Emer (Intel/MIT)
- James Hoe (CMU)
- John Kubiatowicz (UCB)
- David Patterson (UCB)