Last Time: 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
Vector Programming Model

Scalar Registers

Vector Registers

Vector Length Register

Vector Arithmetic Instructions

ADDV v3, v1, v2

Vector Load and Store Instructions

LV v1, r1, r2

Base, r1

Stride, r2

Memory

VLR
## Vector Code Example

<table>
<thead>
<tr>
<th>Vector Code Example</th>
<th>Scalar Code</th>
<th>Vector Code</th>
</tr>
</thead>
<tbody>
<tr>
<td># C code&lt;br&gt;for (i=0; i&lt;64; i++)&lt;br&gt;C[i] = A[i] + B[i];</td>
<td># Scalar Code&lt;br&gt;LI R4, 64&lt;br&gt;loop:&lt;br&gt;L.D F0, 0(R1)&lt;br&gt;L.D F2, 0(R2)&lt;br&gt;ADD.D F4, F2, F0&lt;br&gt;S.D F4, 0(R3)&lt;br&gt;DADDIU R1, 8&lt;br&gt;DADDIU R2, 8&lt;br&gt;DADDIU R3, 8&lt;br&gt;DSUBIU R4, 1&lt;br&gt;BNEZ R4, loop</td>
<td># Vector Code&lt;br&gt;LI VLR, 64&lt;br&gt;LV V1, R1&lt;br&gt;LV V2, R2&lt;br&gt;ADDV.D V3, V1, V2&lt;br&gt;SV V3, R3</td>
</tr>
</tbody>
</table>
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**

- A[0] B[0]

**Execution using four pipelined functional units**

- A[0] B[0]
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

Lane
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

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)
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 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)
CS152 Administrivia

• Quiz 4, Tue Apr 13
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 Instruction

Vectorized Code
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];

**Diagram:**
- A
- B
- C
  - Remainder
  - 64 elements

**Code:**
```
ANDI R1, N, 63   # N mod 64
MTC1 VLR, R1     # Do remainder

loop:
    LV V1, RA
    DSLL R2, R1, 3  # Multiply by 8
    DADDU RA, RA, R2 # Bump pointer
    LV V2, RB
    DADDU RB, RB, R2
    ADDV.D V3, V1, V2
    SV V3, RC
    DADDU RC, RC, R2
    DSUBU N, N, R1  # Subtract elements
    LI R1, 64
    MTC1 VLR, R1    # Reset full length
    BGTZ N, 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 NOP at elements where mask bit is clear

Code example:

```
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

M[1]=1  C[1]
M[0]=0  C[0]

Write Enable  Write Data Port

M[1]=1  C[1]
M[0]=0  Write Data Port
Vector Reductions

Problem: Loop-carried dependence on reduction variables

\[
\text{sum} = 0;
\]

for (i=0; i<N; i++)

\[
\text{sum} += A[i]; \quad \# \text{Loop-carried dependence on sum}
\]

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

\[
\# \text{Rearrange as:}
\]

\[
\text{sum}[0:VL-1] = 0 \quad \# \text{Vector of VL partial sums}
\]

for (i=0; i<N; i+=VL) \quad \# \text{Stripmine VL-sized chunks}

\[
\text{sum}[0:VL-1] += A[i:i+VL-1]; \quad \# \text{Vector sum}
\]

\# Now have VL partial sums in one vector register

do {

    VL = VL/2; \quad \# \text{Halve vector length}

    \text{sum}[0:VL-1] += \text{sum}[VL:2*VL-1] \quad \# \text{Halve no. of partials}

} while (VL>1)
Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

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

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

Scatter example:

\[
\begin{align*}
\text{for } (i=0; i<N; i++) & \\
A[B[i]] & ++;
\end{align*}
\]

Is following a correct translation?

\[
\begin{align*}
\text{LV vB, rB} & \quad \# \text{ Load indices in B vector} \\
\text{LVI vA, rA, vB} & \quad \# \text{ Gather initial A values} \\
\text{ADDV vA, vA, 1} & \quad \# \text{ Increment} \\
\text{SVI vA, rA, vB} & \quad \# \text{ Scatter incremented values}
\end{align*}
\]

- 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)

(See also Cray X1E in Appendix F)
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
  - This concept first used on Lincoln Labs TX-2 computer in 1957, with 36b datapath split into 2x18b or 4x9b
  - Newer designs have 128-bit registers (PowerPC Altivec, Intel SSE2/3/4)
- Single instruction operates on all elements within register
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)
Graphics Processing Units (GPUs)

• Original GPUs were dedicated fixed-function devices for generating 3D graphics

• More recently, GPUs have been made more programmable, so called “General-Purpose” GPUs or GP-GPUs.

• Base building block of modern GP-GPU is very similar to a vector machine
  – e.g., NVIDIA G80 series core (NVIDIA term is Streaming Multiprocessor, SM) has 8 “lanes” (NVIDIA term is Streaming Processor, SP). Vector length is 32 elements (NVIDIA calls this a “warp”).

• Currently machines are built with separate chips for CPU and GP-GPU, but future designs will merge onto one chip
  – Already happening for smartphones and tablet designs
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