16
Vector Architecture

18-548/15-548 Memory System Architecture
Philip Koopman
November 4, 1998

Required Reading: Cragon 11.0-11.2.2, 11.4-11.6.3
Supplemental Reading: Hennessy & Patterson B.1-B.5
Siewiorek & Koopman 5.0-5.9
Siewiorek, Bell & Newell Chapter 44 (Cray 1)

Assignments

- By next class read about vector performance:
  - Cragon 11.3-11.3.5, 11.7
  - Siewiorek & Koopman 7.4 (available on-line)
  - Supplemental Reading:
    - Hennessy & Patterson B.6-B.9
    - Palacharla & Kessler; ISCA 1994

- Homework #9 due November 11

- Lab #5 due November 20
Where Are We Now?

- Where we’ve been:
  - Main memory
  - Disk drives

- Where we’re going today:
  - Vector architecture
  - Vector chaining

- Where we’re going next:
  - Vector Performance
  - Buses

Preview

- Vector processing overview
  - Why was it a good idea in the 60s?
  - (Next lecture -- why it might be a good idea today…)

- Generic vector processor architecture
  - It’s about data pipelining instead of instruction pipelining

- Data pipelining within vector execution -- extended example
  - Vector loads
  - Vectors stores
  - Vector “chaining”
WHAT IS VECTOR PROCESSING?

**Vector Processing Definition**

- **A vector instruction operates on a set of data elements**
  - Typically elements in a data array separated by fixed stride
  - Typically implemented as a load/store register architecture (but, “registers” contain many data elements each)
    - Vector load: start address, stride, number of elements, register#
    - Vector store: start address, stride, number of elements, register#
    - Vector operation: source reg#1, source reg#2, dest reg

- **Traditional supercomputers are vector-based**
  - Scientific code typically operates on arrays and matrices
  - Provides hardware speedup for loop overhead and address generation

- **Vector techniques are useful, even outside of supercomputing**
  - Dealing with structured data that caches poorly, especially multimedia data streams
Classical Vector Processing Bypasses Cache

Historical View of Vector Processing

- **Scientific code speedup** *(Fortran heritage)*
  - Regular strided access patterns encouraged hardware speedup for loops
    - Single-issue machines didn’t have to spend clocks on loop overhead
    - Vector data loads and stores could exploit interleaved memory capabilities
  - When caches became available, they weren’t big enough to help

- **Reduced semantic gap between application and hardware** -- allows compilers to make assumptions about data structures
  - Assume no memory overlap on vector loads/stores (don’t stall reads waiting for writes to complete)
  - Eliminate branch mis-prediction penalties (loops replaced by vector length)
  - Provided optimized performance in an era of immature compiler technology
Vectorization As Pipelining

- Data pipelining, rather than deep instruction execution pipelining
  - Single instruction repeated multiple times on multiple data items

- Single-issue machine could have multiple vector instructions active
  - Even if one clock per result, vectors have many data elements
  - Concurrent instruction execution possible even with a single-issue machine

GENERIC VECTOR ARCHITECTURE

(Inspired by the Stardent Titan architecture; but this example is intentionally not balanced)
Generic Vector Architecture

- Vector Address Generators drive strided fetches/stores
- Interconnect/Bus supports pipelined+interleaved loads and stores
- Vector Register File holds collated data sets
- Functional units perform pipelined operations on vector register contents

Vector Register File

- High-speed memory accessible as **vector register set**
  - Each vector register holds multiple data elements forming a vector (e.g., 8 vector registers, each with 64 data elements)
  - Length register may permit holding fewer than maximum elements in a vector register (e.g., vector registers might have only 32 valid data elements, with the remainder of the vector register contents ignored)

- **Vector registers are a contiguous set of data**
  - Vector Registers always accessed in sequential data words (with “stride” of 1)
  - Memory arrays/vectors of stride > 1 are converted to/from contiguous data in VRF by load/store operations

- **Also need scalar registers**
  - Might be kept within VRF and accessed with special address form
  - Might be the scalar CPU floating point register set
Vector Address Generators

- Coordinates data placement for load and store operations
- Generates strided memory addresses
  - Seeded with vector information:
    - Start address
    - Stride
    - Number of elements
  - Generates addresses for memory load/store
- Generates vector register addresses
  - Vector register number
  - Offset within vector register
- One VAG needed for each concurrent vector load/store

Functional Units

- Pipelined arithmetic and logic units, typically:
  - Addition/logic functions
    1) Pre-normalization (align decimal points w.r.t. exponent values)
    2) Addition/subtraction
    3) Post-normalization (align resultant decimal point) & round
  - Multiplication
    1) Mantissa multiplication
    2) Partial product addition
    3) Post-normalization & round
  - Division/square root
- Data may enter and leave functional unit every clock cycle for high throughput
  - Vector execution model means software guarantees no data dependencies among vector elements
Vector Data Switch

- **Crossbar switch to connect system components**
  - Data rate must be one element per clock cycle per port
  - Number of ports determined by expected data flow rates
    - At least three VRF ports to support LOAD/LOAD/op/STORE functions
    - One or more memory ports to support concurrent Loads and Stores
    - At least three Functional Unit ports to support LOAD/LOAD/op/STORE functions

Multiple Memory Banks

- **Low end machine -- interleaved memory**
  - Memory banks take turns being connect to bus
  - Interleaved memory access improves available bandwidth and may reduce latency for concurrent accesses.

- **High end machine -- multiple concurrent banks**
  - Might use crossbar switch (instead of bus, not instead of VDS) to connect several memory banks to the VDS simultaneously
  - Might be interleaved and assume different subsets of banks connected each clock
### Vector Instruction Forms

<table>
<thead>
<tr>
<th>Result</th>
<th>Operand1</th>
<th>Operand2</th>
<th>Example</th>
<th>Data Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>Memory</td>
<td>---</td>
<td>Load A(i)</td>
<td>v</td>
</tr>
<tr>
<td>Memory</td>
<td>Vector</td>
<td>---</td>
<td>Store A(i)</td>
<td>v</td>
</tr>
<tr>
<td>Vector</td>
<td>Scalar</td>
<td>Vector</td>
<td>C(i) + B + A(i)</td>
<td>2v+1</td>
</tr>
<tr>
<td>Vector</td>
<td>Vector</td>
<td>---</td>
<td>C(i) + abs(B(i))</td>
<td>2v</td>
</tr>
<tr>
<td>Vector</td>
<td>Vector</td>
<td>Mask</td>
<td>C(i) + B(i) ≤ mask</td>
<td>up to 2v</td>
</tr>
<tr>
<td>Scalar</td>
<td>Vector</td>
<td>---</td>
<td>C + S B(i)</td>
<td>v+1</td>
</tr>
<tr>
<td>Scalar</td>
<td>Vector</td>
<td>Vector</td>
<td>C + S B(i) ≠ A(i)</td>
<td>2v+1</td>
</tr>
<tr>
<td>Vector</td>
<td>Scalar</td>
<td>Vector</td>
<td>C(i) + A B(i) + C(i)</td>
<td>3v+1</td>
</tr>
</tbody>
</table>

- **Key instruction:** **DAXPY**
  - Y = aX + Y ("double precision a-X plus Y")
  - Inner loop of LINPACK

### DAXPY -- A Key Operation

- **Used in Gaussian Elimination; “most used” supercomputer instruction**
  - LOAD -- LOAD -- OP -- STORE

- **for (i = 0; i < 64; i++) { Y(i) = a * X(i) + Y(i); }**

  /* assume 64-element vector registers */
  
  LD F0, a ; load scalar a  
  LV V1, x ; load vector register 1 with x  
  MULTSV V2, F0, V1 ; vector-scalar multiply F0 is a; V1 is X  
  LV V3, y ; load vector register 2 with y  
  ADDV V4, V2, V3 ; add vector new Y (V4) = aX + old Y  
  SV y, V4 ; Store result

- **Variations**
  - Vector length register could be set for varying array sizes
    - If vector exceeds vector register length, must use “strip mining”
  - Some systems have a DAXPY instruction: vLoad; vLoad; DAXPY; vStore
VECTOR EXAMPLE:
\[ Z(i) = X(i) + Y(i) \]
**Vector Load Operation  X(i)**

<table>
<thead>
<tr>
<th>Beat</th>
<th>VAG</th>
<th>ADDR</th>
<th>BUS</th>
<th>BANK 0</th>
<th>BANK 1</th>
<th>BANK 2</th>
<th>BANK 3</th>
<th>DATA BUS</th>
<th>DATA SWITCH</th>
<th>VRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X[1]</td>
<td>X[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>X[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 8-element vector load
  - 8 beat latency to first load
  - 8 elements loaded in 15 beats latency
  - BUT, could load other data starting at beat 9...

**Concurrent Vector Loads**

- VECTOR REGISTER FILE
- VECTOR ADDRESS GENERATORS
- FUNCTIONAL UNITS

- BANK 0
- BANK 1
- BANK 2
- BANK 3

4-WAY INTERLEAVED MEMORY
Concurrent Vector Load Operations  \(X(i)\)  \(Y(i)\)

<table>
<thead>
<tr>
<th>Beat</th>
<th>VAG0</th>
<th>ADDR</th>
<th>BUS</th>
<th>BK 0</th>
<th>BK 1</th>
<th>BK 2</th>
<th>BK 3</th>
<th>DATA BUS</th>
<th>DATA SWITCH</th>
<th>VRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X[1]</td>
<td>Y[0]</td>
<td>X[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Y[1]</td>
<td>Y[0]</td>
<td>rX[0]</td>
<td>rY[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td>cyc</td>
<td></td>
</tr>
</tbody>
</table>

Notes on Concurrent Load Example

- **Single bus results in data bandwidth limit**
  - Could provide 2 paths (2 buses, or 2 ports on a crossbar switch)
  - Could run bus at 2x vector clock cycle (2 beats per clock cycle)

- **Two VAGs compete for bus**
  - Need arbitration logic to assure fair access
  - VAGs aren’t strictly alternating in order to exploit available memory banks
    - VAGs with addresses waiting might go in round-robin order
    - But, a VAG might have to skip a turn or wait if the memory bank it needs is busy
Vector Store

Vector Store Operation  Z(i)

<table>
<thead>
<tr>
<th>Beat</th>
<th>VAG</th>
<th>ADDR BUS</th>
<th>VRF</th>
<th>DATA SWITCH</th>
<th>DATA BUS</th>
<th>BANK 0</th>
<th>BANK 1</th>
<th>BANK 2</th>
<th>BANK 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Z[0]</td>
<td>Z[0]</td>
<td>Z[0]</td>
<td>Z[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Z[1]</td>
<td>Z[1]</td>
<td>Z[0]</td>
<td>Z[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 8-element vector store
  - 4 beat latency to first store -- data transmission in parallel with addressing
  - Assume writing one clock faster than reading because don’t have to wait for data to get back out of DRAM -- address and data arrive concurrently
Vector Add Operation  registers:  \( Z = X + Y \)

<table>
<thead>
<tr>
<th>Beat</th>
<th>VRF</th>
<th>DATA SWITCH</th>
<th>ADDER (3 STAGES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X[0] Y[0]</td>
<td>X[0] X[0]</td>
<td>Z[0]</td>
</tr>
<tr>
<td>14</td>
<td>Z[7]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 8-element vector add
  - Data pipelining:
    - 7 beats to first result
    - 1 beat per result steady state
Vector Chaining

- Vector chaining is analogous to data forwarding in a scalar processor
  - Accomplished by hardware
    - Semantics of vector operations assume no implicit data hazards among vectors
  - Load/op chaining:
    - Vectors loaded via data switch from memory
    - Subsequent arithmetic operations fed from data switch as data arrives
  - Op/op chaining:
    - Outputs from arithmetic op fed back into functional units
  - Op/store chaining:
    - Outputs from arithmetic op fed straight to memory from data switch
Vector Chaining -- loads + operation

Vector Chaining -- loads + operation + store
Vector Chaining Analysis

**Example:** 8 adds in 30 beats

- 8 of 30 beats of ADDER consumed
- 24 of 30 beats of bus consumed (both address and data)
- 24 of 90 available beats of vector address generator consumed
- 88 of 120 available beats of memory bandwidth consumed

**Problem with this example architecture: lack of balance**

- Bus a bottleneck
- Memory bandwidth is close to being a bottleneck
- Vector address generators under-used
- Adder under-used

- Lack of bus bandwidth is a common architectural problem
  - Bus bandwidth is more expensive getting a faster CPU
  - Toy benchmarks don’t use the bandwidth, so there may not be “obvious” benefit
REVIEW

Vector Microprocessors Today?

◆ Vector computation model not as compelling as it once was
  • Multi-issue, latency-tolerant architectures reduce cost of loop overhead
    – Instruction concurrency is available, and can substitute for data concurrency
  • Improved compiler technology reduces value of programmer using vectors to give hints to hardware
    – Improved algorithms to exploit cache
    – Smart pre-fetching hardware, cache bypass, latency tolerance
  • Commodity networked computing can often achieve comparable performance to a supercomputer
    – Single-chip CPUs now have very high clock rates
    – Improved infrastructure for parallel computing makes it accessible

◆ But, desktop CPUs can benefit from supercomputer tricks
  • Strided prefetching to reduce latency and better use memory bandwidth
  • Selective bypassing of cache to avoid cache pollution
  • Intel i860 was an experiment in this direction; but it was a poor compiler target
### Review

- **Vector processing overview**
  - Exploits regular data access patterns to achieve data movement pipelining

- **Generic vector processor architecture**
  - VRF, VAG, VDS, functional units, memory banks

- **Data pipelining within vector execution**
  - Vector loads
  - Vectors stores
  - Vector chaining