ECE 552 / CPS 550
Advanced Computer Architecture I

Lecture 17
Vectors

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall11.html
13 November – Homework #4 Due

Project Status
- Plan on having preliminary data or infrastructure

8 November – Class Discussion

Roughly one reading per class. Do not wait until the day before!

1. Mudge, “Power: A first-class architectural design constraint”
2. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs”
4. Tullsen et al. “Simultaneous multithreading: Maximizing on-chip parallelism”
### Last Time

<table>
<thead>
<tr>
<th>Superscalar</th>
<th>Fine-Grained</th>
<th>Coarse-Grained</th>
<th>Multiprocessing</th>
<th>Simultaneous Multithreading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Thread 2</td>
<td>Thread 3</td>
<td>Thread 4</td>
<td>Thread 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thread 2</td>
<td>Thread 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thread 4</td>
<td>Thread 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Idle slot</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time (processor cycle)
Vectors and Data-level Parallelism

**SCALAR**
(1 operation)

```
r1
  +
  |
  v
```

r2

```
r3
```

add r3, r1, r2

**VECTOR**
(N operations)

```
v1
v2
  +
  |
  v
```

```
v3
```

vadd.vv v3, v1, v2
Data-level Parallelism

Vectors effective for data-level parallelism (DLP)

-- Vectors are most efficient way to exploit DLP
-- Superscalar (e.g., DLP as instruction-level parallelism) is less efficient
-- Multiprocessor (e.g., DLP as thread-level parallelism) is less efficient

Scientific Computing

-- Weather forecasting, car-crash simulation, biological modeling
-- Vector processors were invented for supercomputing, but fell out of favor after the advent of multiprocessors

Multimedia Computing

-- Identical ops on streams or arrays of sound samples, pixels, video frames
-- Vector processors were revived for multimedia computing
Vectors widely used for supercomputing (1970s-1990s)
  -- Cray, CDC, Convex, TI, IBM

Transition away from vectors (1980s-1990s)
  -- Fitting a vector processor into a single chip was difficult
  -- Building supercomputers from commodity components was easier

Vectors are re-emerging as SIMD
  -- SIMD – single instruction multiple data
  -- SIMD provides short vectors in all ISAs
  -- Provides multimedia acceleration
Parts of a Vector Processor

Scalar processor
-- Scalar register file
-- Example: 32 registers, each with 1 32-bit element
-- Scalar functional units (arithmetic, load/store, etc...)

Vector register file
-- Each register is an array of elements
-- Example: 32 registers, each with 32 64-bit elements
-- MVL – maximum vector length = max # of elements per register

Vector functional units
-- Integer, floating-point, load/store, etc...
-- Some datapaths (e.g., ALUs) shared by vector, scalar units
Parts of a Vector Processor

- Main memory
- Vector load-store
- Vector registers
- Scalar registers
- FP add/subtract
- FP multiply
- FP divide
- Integer
- Logical
Vector Supercomputers

Cray-1, 1976

Scalar Unit
- Load/Store architecture

Vector Extension
- Vector registers
- Vector instructions

Implementation
- Hardwired control (no microcode)
- Pipelined functional units
- Interleaved memory system
- No data caches
- No virtual memory
Vector Programming Model

Scalar Registers
- `r15`
- `r0`

Vector Registers
- `v15`
- `v0`
- `v1`
- `v2`
- `v3`

Vector Arithmetic Instructions
- `ADDV v3, v1, v2`

Vector Length Register
- `VLR`

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

Base, `r1`
Stride, `r2`
Memory

Vector Register
Vector ISA Benefits

Compact – single instruction defines N operations
-- also fewer branches

Parallel – N operations are (data) parallel
-- no dependencies between vector elements
-- like VLIW, no complex hardware for dynamic scheduling
-- scalable; additional functional units give additional performance

Expressive – memory ops describe access patterns
-- vector memory ops exhibit continuous or regular access patterns
-- vector memory ops can prefetch and/or effectively use memory banks

-- amortize high latency for 1\textsuperscript{st} element over large sequential pattern
(bursts of data transfer…1\textsuperscript{st} element incurs a long latency….subsequent elements are pipelined to produce a new element per cycle)
**Basic Vector Instructions**

Suppose 64-element vectors

<table>
<thead>
<tr>
<th>Instr</th>
<th>Operands</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADD.VV</td>
<td>V1, V2, V3</td>
<td>V1 = V2 + V3</td>
<td>vector + vector</td>
</tr>
<tr>
<td>VADD.SV</td>
<td>V1, R0, V2</td>
<td>V1 = R0 + V2</td>
<td>scalar + vector</td>
</tr>
<tr>
<td>VMUL.VV</td>
<td>V1, V2, V3</td>
<td>V1 = V2 * V3</td>
<td>vector x vector</td>
</tr>
<tr>
<td>VMUL.SV</td>
<td>V1, R0, V2</td>
<td>V1 = R0 * V2</td>
<td>scalar x vector</td>
</tr>
<tr>
<td>VLD</td>
<td>V1, R1</td>
<td>V1 = M[R1,…R1+63]</td>
<td>load, stride=1</td>
</tr>
<tr>
<td>VLDS</td>
<td>V1, R1, R2</td>
<td>V1 = M[R1,…R1+63*R2]</td>
<td>load, stride=R2</td>
</tr>
<tr>
<td>VLDX</td>
<td>V1, R1, V2</td>
<td>V1 = M[R1+V2(i), i=0 to 63]</td>
<td>indexed gather</td>
</tr>
<tr>
<td>VST</td>
<td>V1, R1</td>
<td>M[R1…R1+63] = V1</td>
<td>store, stride=1</td>
</tr>
<tr>
<td>VSTS</td>
<td>V1, R1, R2</td>
<td>M[R1,…R1+63*R2] = V1</td>
<td>store, stride=R2</td>
</tr>
<tr>
<td>VSTX</td>
<td>V1, R1, V2</td>
<td>M[R1+V2(i), i=0 to 63] = V1</td>
<td>indexed scatter</td>
</tr>
</tbody>
</table>
Vector Code Example

# C code
for (i=64 ; i>0 ; 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
VLD V1, R1
VLD V2, R2
ADD.VV V3, V1, V2
VST V3, R3

-- Load immediate (LI) with length of vector (64)
-- Vector length register (VLR)
Vector Length

Vector register holds a max number of elements

-- MVL: Maximum vector length (e.g., 64)
-- But application vector lengths may not match MVL

Vector length register

-- VL: controls length of any vector operation (add, multiply, load, store)
-- Example: vadd.vv with VL10 is equivalent to:
  for(i=0; i<10; i++) {V1[i] = V2[i] + V3[i]}
-- Before sequence of vector instructions, VL set to number <= to MVL

How can we code applications where the vector length is not known until run-time?
Strip Mining

-- Suppose application VL > MVL
-- Generate loop that handles MVL elements per iteration
-- Translate each loop iteration into a single vector instruction

Example: AX+Y

-- First loop for (N mod MVL) elements. Remaining loops for MVL elements

VL = (N mod MVL);
for (i=0 ; i<VL ; i++)
    Y[i] = A*X[i] + Y[i];
low = (N mod MVL)
VL = MVL
for (i=low ; i<N ; i++)
    Y[i] = A * X[i] + Y[i];

# set VL to be a smaller vector
# 1st-loop translates into a single set
# of vector instructions
# low – strips off beginning elements
# set VL to be max vector length
# 2nd-loop translates into N/MVL sets
# of vector instructions
Vector Instruction Execution

Use deep pipeline (fast clock) to execute operations for each vector element.

Simplify pipeline control because elements in vector are independent $\rightarrow$ no hazards.

\[
V3[i] \leftarrow V1[i] \times V2[i]
\]
Consider the following code with vector length of 32

\[
\begin{align*}
\text{vmul.vv} & \quad \text{V1, V2, V3} \\
\text{vadd.vv} & \quad \text{V4, V1, V5} \quad \text{# very long RAW hazard}
\end{align*}
\]

**Chaining**

-- V1 is not a single entity, but a vector of individual elements
-- Pipeline forwarding can work for individual elements

**Flexible Chaining**

-- Chain any vector to any other active vector operation
-- Requires more read/write ports in the vector register file
Opt 2 – Multiple Datapaths

ADDV C,A,B

Execution using one pipelined datapath

1 adder → 1 element / cycle
N cycles

Execution using four pipelined datapaths

4 adders → 4 elements / cycle
N/4 cycles
Opt 2+: Multiple Lanes

-- Vector elements interleaved across lanes
-- Example: V[0, 4, 8, …] on Lane 1, V[1, 5, 9, …] on Lane 2, etc.

-- Compute for multiple elements per cycle
-- Example: Lane 1 computes on V[0] and V[4] in one cycle

-- Modular, scalable design
-- No inter-lane communication needed for most vector instructions
Suppose you want to vectorize this code:

```c
for (i=0 ; i<N ; i++) {
    if(A[i] != B[i]) {A[i] -= B[i]; } }
```

Solution: vector conditional execution

-- Add **vector flag registers**, single-bit mask per vector element
-- Use vector-compare to set the vector flag register
-- Use vector flag register to control vector-sub
-- Vector op executed only if corresponding flag element is set

```
vld V1, Ra
vld V2, Rb
vcmp.neq.vv M0, V1, V2   # vector compare for mask
vsub.vv V3, V2, V1, M0   # conditional vadd
vst V3, Ra
```
Vector Memory

Multiple, interleaved memory banks (e.g., 16)
Provides memory-level parallelism when filling vector registers
Support narrow data types
-- Allow each vector register to store 16-, 32-, or 64-bit elements
-- Use a control register to indicate width of register elements

Support fixed-point arithmetic
-- Minor modification to functional units

Support element permutations for vector reductions
-- for(i=0 ; i<N ; i++) \{ S += A[i] \}
-- Rewrite as:
  for(i=0 ; i<N ; i+=VL) \{ S[0:VL-1] += A[i:i+VL-1]; \} # S[...], A[...] are
  for(i=0 ; i<VL ; i++) \{ S+=[S[i]]; \} # vectors of VL elements
-- First loop trivially vectorizable
-- Second loop vectorizable by splitting vector register S into two vector
  registers. Take a binary-tree approach to reduction
SIMD in Superscalar Processors

SIMD extends conventional ISA
-- SIMD – single instruction, multiple data
-- MMX, SSE, SSE-2, SSE-3, 3D-Now, Altivec, VIS

Objective: Accelerate multimedia processing
-- Define vectors of 16-, 32-bit elements in regular registers
-- A logical vector register may span multiple physical registers
-- Apply SIMD arithmetic on these vectors

Advantages
-- No vector register file, which would require additional area
-- Simple extensions (new opcodes, modified datapath)
SIMD Challenges

SIMD vectors are short with fixed size
-- Cannot capture data parallelism wider than 64 bits
-- Recent shift from 64-bit to 128-bit vectors (SSE, Altivec)

SIMD does not support vector memory accesses
-- Strided or indexed access require equivalent multi-instruction sequences
-- Without vector memory accesses, much lower benefits in performance and code density
SIMD versus Vectors

-- QCIF and CIF numbers are in clock cycles per frame
-- All other numbers are in clock cycles per pixel
-- MMX results assume no first-level cache misses
-- Courtesy: Christos Kozyrakis, Stanford

<table>
<thead>
<tr>
<th></th>
<th>Vector</th>
<th>MMX</th>
</tr>
</thead>
<tbody>
<tr>
<td>iDCT</td>
<td>0.75</td>
<td>3.75 (5.0x)</td>
</tr>
<tr>
<td>Color Conversion</td>
<td>0.78</td>
<td>8.00 (10.2x)</td>
</tr>
<tr>
<td>Image Convolution</td>
<td>1.23</td>
<td>5.49 (4.5x)</td>
</tr>
<tr>
<td>QCIF (176x144)</td>
<td>7.1M</td>
<td>33M (4.6x)</td>
</tr>
<tr>
<td>CIF (352x288)</td>
<td>28M</td>
<td>140M (5.0x)</td>
</tr>
</tbody>
</table>
**Vector Multiprocessor**
- 2-way superscalar, 4-way multi-threaded, in-order cores with vectors
- Cores communicate on a wide ring bus
- L2 cache is partitioned among the cores
  - Provides high aggregate bandwidth
  - Allows data replication and sharing
Larrabee x86 Core

-- separate scalar, vector units with separate registers
-- scalar unit: in-order x86 core
-- vector unit: 16 32-bit ops/clock

-- short execution pipelines
-- fast access to L1 cache
-- direct connection to L2 cache subset

-- instructions support prefetch into L1 and L2 caches
Larrabee Vector Unit

**Vector Instruction Set**
- 32 vector registers (512 bits each)
- vector load/store with scatter/gather
- 8 mask registers for conditional exec.
- mask registers select lanes for an instruction
- mask registers allow separate execution kernels in each lane

**Vector Instruction Support**
- Fast read from L1 cache
- Numeric type conversion and replication in memory path
Vector Power Efficiency

Power and Parallelism

-- Power(1-lane) = [capacitance] x [voltage]^2 x [frequency]
-- If we double number of lanes, we double peak performance
-- Then, if we halve frequency, we return to original peak performance.
-- But, halving frequency allows us to halve voltage
-- Power (2-lane) = [2 x capacitance] x [voltage/2]^2 x [frequency/2]
-- Power (2-lane) = Power(1-lane)/4 @ same peak performance

Simpler Logic

-- Replicate control logic for all lanes
-- Avoid logic for multiple instruction issue or dynamic out-of-order execution

Clock Gating

-- Turn-off clock when hardware is unused
-- Vector of given length uses specific resources for specific # of cycles
-- Conditional execution (masks) further exposes unused resources
Summary

Vector Processors
   -- Express and exploit data-level parallelism (DLP)

SIMD Extensions
   -- Extensions for short vectors in superscalar (ILP) processors
   -- Provide some advantages of vector processing at less cost