**Motivation**

Recall SIMD from Chapter 5

---

**Data Parallel Architectures - SIMD**

**Motivation**

Vectors  
Multimedia SIMD  
GPUs

---

**Vector Processors**

Figure 4.2 The basic structure of a vector architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. This chapter defines special vector instructions for both arithmetic and memory accesses. The figure shows vector units for logical and integer operations so that VMIPS looks like a standard vector processor that usually includes these units; however, we will not be discussing these units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches (thick gray lines) connects these ports to the inputs and outputs of the vector functional units.

---

**What are Vector Instructions?**

A vector is a one-dimensional array of numbers

\[
\text{float } A[64], B[64], C[64]
\]

Original motivation: Many scientific programs operate on vectors of floating point data

\[
\text{for (i=0; i<64; i++)}
\]

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

Multimedia, graphics, other emerging apps also operate on vectors of data

A vector instruction performs an operation on each vector element

\[
\text{ADDVV } C, A, B
\]

---

**Copyright © 2011, Elsevier Inc. All rights Reserved.**
### Why Vector Instructions?

Want deeper pipelines, BUT

### Vector Architectures

<table>
<thead>
<tr>
<th>Vector-Register Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/store architecture</td>
</tr>
<tr>
<td>All vector operations use registers (except load/store)</td>
</tr>
<tr>
<td>Multiple ports are cheaper</td>
</tr>
<tr>
<td>Optimized for small vectors</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory-Memory Vector Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>All vectors reside in memory</td>
</tr>
<tr>
<td>Long startup latency</td>
</tr>
<tr>
<td>Multiple ports are expensive</td>
</tr>
<tr>
<td>Optimized for long vectors</td>
</tr>
</tbody>
</table>

Often vectors are short

- Early machines were memory-memory (TI ASC, CDC STAR)
- Later machines use vector registers

### VMIPS Architecture

- Strongly based on Cray
- Extend MIPS with vector instructions
- Scalar unit
  - Eight vector registers (V0-V7)
  - Each is 64 elements, 64 bits wide
- Five Vector Functional Units
  - FP+, FP*, FP/, integer & logical
  - Fully pipelined
- Vector Load/Store Units
  - Fully pipelined

### VMIPS Architecture, cont.

- Vector-Vector Instructions
  - Operate on two vectors
  - Produce a third vector
    - for (i=0; i<64; i++)
      - \( V1[i] = V2[i] + V3[i] \)
    - ADDVV.D V1, V2, V3
- Vector-Scalar Instructions
  - Operate on one vector, one scalar
  - Produce a third vector
    - for (i=0; i<64; i++)
      - \( V1[i] = F0 + V3[i] \)
    - ADDVS.D V1, V3, F0
Vector Load/Store Instructions

Load/Store a vector from memory into a vector register

Operates on contiguous addresses

- $\text{LV} V1, R1$ ; $V1[i] = M[R1 + i]$
- $\text{SV} R1, V1$ ; $M[R1 + i] = V1[i]$

Load/Store Vector with Stride

Vectors not always contiguous in memory

Add non-unit stride on each access

- $\text{LVWS} V1, (R1, R2)$ ; $V1[i] = M[R1 + i*R2]$
- $\text{SVWS} (R1, R2), V1$ ; $M[R1 + i*R2] = V1[i]$

Vector Load/Store Indexed

Indirect accesses through an index vector

- $\text{LVI} V1, (R1+V2)$ ; $V1[i] = M[R1 + V2[i]]$
- $\text{SVI} (R1+V2), V1$ ; $M[R1 + V2[i]] = V1[i]$

Double-precision $A*X$ Plus Y (DAXPY):

for (i=0; i<64; i++)

- $Y[i] = a * X[i] + Y[i]$

- $\text{L.D} F0, a$
- $\text{LV} V1, Rx$
- $\text{MULVS.D} V2, V1, F0$
- $\text{LV} V3, Ry$
- $\text{ADDVV.D} V4, V2, V3$
- $\text{SV} Ry, V4$

6 instructions instead of 600!

Remember: MIPS means “Meaningless Indicator of Performance”

Not All Vectors are 64 Elements Long

Vector length register (VLR)

Controls length of vector operations

$0 \leq \text{VLR} \leq \text{MVL} = 64$

for (i=0; i<100; i++)

- $X[i] = a * X[i]$

- $\text{LD} FS, a$
- $\text{MTC1} \text{VLR}, 36$ /* 100 - 64 */
- $\text{LV} V1, Rx$
- $\text{MULVS} V2, V1, F0$
- $\text{SV} \text{Rx}, V2$
- $\text{ADD} \text{Rx}, \text{Rx}, 36$
- $\text{MTC1} \text{VLR}, 64$
- $\text{LV} V1, Rx$
- $\text{MULVS} V2, V1, F0$
- $\text{SV} \text{Rx}, V2$

Strip Mining for $i = 1, n$

Strip Mining

General case: Parameter n

- $\text{DO} 10 I = 1, n$

- $X[i] = a * X[i]$

10 $\text{CONTINUE}$

Strip-mined version (pseudocode)

- $\text{low} = 1$

- $\text{VL} = (n \text{ mod } \text{MVL})$ /* Odd sized piece */

- $\text{DO} 1 j = 0, (n \text{ / } \text{MVL})$ /* Outer loop */

- $\text{DO} 10 I = \text{low}, \text{low}+\text{VL1}$ /* Length */

- $X[\text{i}] = a * X[\text{i}]$

10 $\text{CONTINUE}$

- $\text{low} = \text{low} + \text{VL}$ /* Base of next chunk */

- $\text{VL} = \text{MVL}$ /* Reset length to MAX */

1 $\text{CONTINUE}$
Old Vector Machines Did Not Have Caches

Caches
- Vectorizable codes often have poor locality
- Large vectors don’t fit in cache
- Large vectors flush other data from the cache
- Cannot exploit known access patterns
- Unpredictability hurts
- Degrades cycle time

Vector Registers (like all registers)
- Very fast
- Predictable
- Short id
- Multiple ports easier

More Options

Use vector mask register for vectorizing

```fortran
DO 10 i = 1, 64
  if A(i) /= 0.0 then A(i) = A(i)
10 CONTINUE
```

Use chaining (vector register bypass) for RAWs

```fortran
MULTV V1, ,
ADDV , V1,
```

Use gather/scatter for sparse matrices

```fortran
DO 10 i = 1, 64
  A(K(i)) = A(K(i)) + C(M(i))
10 CONTINUE
```

FINAL WARNING: Make scalar unit fast!

Amdahl’s law

CRAY 1 was the fastest scalar computer

Compiler Technology

- Must detect vectorizable loops
- Must detect dependences that prevent vectorization
- Data, anti, output dependences
- Only data (or true) dependences important, others can be eliminated with renaming

Multimedia Instructions

- Recent general-purpose processors include multimedia instructions
- Multimedia data derived from sampling analog input
- Correctness dictated by human perception
- Smaller data types - 8-bit, 16-bit
- Compare with 32 and 64 bit processor data paths
- Significant levels of data parallelism
- Large collection of small data elements
- Identical processing of similar elements
- e.g. Image Addition
- For I = 1 to 1024
- For J = 1 to 1024
  dest[I,J] = src1[I,J] + src2[I,J]
**Multimedia - Packed Data Types**

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>16 bits</th>
<th>Operand 2</th>
<th>+</th>
<th>Result</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

48 bits are wasted!

*Can we use them in any way?*

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>16 bits</th>
<th>Operand 2</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>Result</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 operations in 1 cycle

**SPEEDUP: 4X??**

Called SIMD:

*single-instruction multiple-data parallelism*

**Other Multimedia Extensions**

**Saturation arithmetic**
Example: image addition

For $I = 1$ to 1024
For $J = 1$ to 1024
$dest[i,j] = src1[i,j]+src2[i,j]$

If $(dest > 255)$
$dest = 255$;
If $(dest < 0)$
$dest = 0$;

Saturation ensures clamping of values

---

**Other Multimedia Extensions (Cont.)**

**Sub-word Rearrangement**

How do we go from unpacked data types to packed data types?

- Provide ISA support for pack, unpack, expand, align, ...
- Support for other types of sub-word rearrangement
  - Shift, rotate, permute, ...
  - E.g., for FFT butterfly algorithm

Many others

- Conditional execution, memory instructions, special-purpose instructions, ...

---

**Example: Intel MMX ISA Extensions (~1996)**

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>PMADDUBSW, PMADDWD, PMULDQ, PMULUDQ, PMULUSW, PMULUSW2, PMULL, PMULLW, PMULLQ, PMULLQ2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison</td>
<td>PCMPGTB, PCMPGTE, PCMPGTW, PCMPGTEQ, PCMPGTQ, PCMPGTWQ, PCMPGTEQW, PCMPGTEQWQ</td>
</tr>
<tr>
<td>Logical</td>
<td>PAND, PANDN, POPFD, POPFDPEP, POPFDNUM, POPFDNUMP</td>
</tr>
<tr>
<td>FP and MMX state mgmt</td>
<td>EMMS</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>MOVQ, MOVQ2</td>
</tr>
</tbody>
</table>

57 new instructions

- Use FP registers, 32-bit data path, SIMD, saturation, ...
Example: Intel SSE ISA Extensions (~1999)

<table>
<thead>
<tr>
<th>Data movement</th>
<th>MOV, MOVAPS, MOVPS, MOVUPS, MOVHPS, MOVHPFS, MOVLPFS, MOVLPFS, MOVSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shuffle</td>
<td>SHUFPS, UNPCKHPS, UNPCKLPS</td>
</tr>
<tr>
<td>State</td>
<td>FXSAVE, FXRSTOR, SYMCSR, LDMXCSR</td>
</tr>
<tr>
<td>MMX Tech:</td>
<td>PINVFX, PROTXR, PULAXH, PSEEUXR, PAVG, PAVD, PAVG, PAVD, RMAX</td>
</tr>
<tr>
<td>Enhancements</td>
<td>RAX, RMXOR, MOVNTQ, MOVTPS, PREFETCH, SFENCE</td>
</tr>
<tr>
<td>Streaming/prefetching</td>
<td>MASKMOVQ, MOVNTD, MOVTPS, PREFETCH, SFENCE</td>
</tr>
<tr>
<td>Conversions</td>
<td>CVTSS2SI, CVTTSS2SI, CVTSI2SS, CVTPI2PS, CVTPS2PI, CVTTTPS2PI</td>
</tr>
</tbody>
</table>

70 instructions
Separate register state, 128-bit data path

Later Versions

2001/04/07: SSE2/3/4: double precision floating point, instructions to accelerate specific functions

2010: Advanced vector extensions (AVX)
- 256 bits, three operands
- Relaxed alignment
- Fused multiply-add (FMA) (A = A*B + C)

Most recent: AVX-512: 512 bits (1024 bits in the future)