Data Parallel Architectures - SIMD

Motivation
Vectors
Multimedia SIMD
GPUs
Recall SIMD from Chapter 5
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

```
float A[64], B[64], C[64]
```

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

```
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

```
ADDVV C, A, B
```
Why Vector Instructions?

Want deeper pipelines, BUT
Vector Architectures

Vector-Register Machines
  Load/store architecture
  All vector operations use registers (except load/store)
  Multiple ports are cheaper
  Optimized for small vectors

Memory-Memory Vector Machines
  All vectors reside in memory
  Long startup latency
  Multiple ports are expensive
  Optimized for long vectors

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
Vector-Vector Instructions
Operate on two vectors
Produce a third vector

\[
\text{for (}\ i=0;\ i<64;\ i++\text{)}
\text{ V1}[i] = V2[i] + V3[i]
\]

\text{ADDVV.D V1, V2, V3}

Vector-Scalar Instructions
Operate on one vector, one scalar
Produce a third vector

\[
\text{for (}\ i=0;\ i<64;\ i++\text{)}
\text{ V1}[i] = F0 + V3[i]
\]

\text{ADDSV.D V1, V3, F0}
VMIPS Architecture, cont.

Vector Load/Store Instructions

Load/Store a vector from memory into a vector register

Operates on contiguous addresses

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

Load/Store Vector with Stride

Vectors not always contiguous in memory

Add \textit{non-unit stride} on each access

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

Vector Load/Store Indexed

Indirect accesses through an index vector

\[ \text{LVI } \text{V1, (R1+V2) ; V1}[i] = \text{M}[\text{R1} + \text{V2}[i]] \]
\[ \text{SVI (R1+V2), V1 ; M}[\text{R1} + \text{V2}[i]] = \text{V1}[i] \]
Double-precision A*X Plus Y (DAXPY):
   for (i=0; i<64; i++)
       Y[i] = a * X[i] + Y[i]

L.D      F0, a
LV       V1, Rx
MULVS.D  V2, V1, F0
LV       V3, Ry
ADDVV.D  V4, V2, V3
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 < VLR \leq MVL = 64

for (i=0; i<100; i++)
    X[i] = a \times X[i]

LD F0, a
MTC1 VLR, 36 /* 100 - 64 */
LV V1, Rx
MULVS V2, V1, F0
SV Rx, V2
ADD Rx, Rx, 36
MTC1 VLR, 64
LV V1, Rx
MULVS V2, V1, F0
SV Rx, V2

Strip Mining for i = 1, n
Strip Mining

General case: Parameter $n$

\[
\text{DO 10 I = 1, n} \\
\quad X(i) = a \times X(i) \\
10 \text{ CONTINUE}
\]

Strip-mined version (pseudocode)

\[
\text{low = 1} \\
\text{VL = (n mod MVL) /* Odd sized piece */} \\
\text{DO 1 j = 0, (n / MVL) /* Outer loop */} \\
\quad \text{DO 10 i = low, low+VL1 /* Length */} \\
\quad \quad X(i) = a \times X(i) \\
10 \quad \text{CONTINUE} \\
\quad \text{low = low + VL /* Base of next chunk */} \\
\quad \text{VL = MVL /* Reset length to MAX */} \\
1 \quad \text{CONTINUE}
\]
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

```plaintext
DO 10 i = 1, 64
   if A(i) ! 0.0 then A(i) = A(i)
10 CONTINUE
```

Use chaining (vector register bypass) for RAWs

```plaintext
MULTV V1, ,
ADDV , V1,
```

Use gather/scatter for sparse matrices

```plaintext
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

CRAY1 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
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

48 bits are wasted!

Can we use them in any way?

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
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>Category</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>PADD[B,W,D], PADDSS[B,W], PADDUS[B,W], PSUB[B,W,D], PSUBS[B,W,D], PSUBUS[B,W], PMULHW, PMULLW, PMADDWD</td>
</tr>
<tr>
<td>Comparison</td>
<td>PCMPEQ[B,W,D], PCMPGT[B,W,D]</td>
</tr>
<tr>
<td>Conversion</td>
<td>PACKUSWB, PACKSS[WB,DW], PUNPCKH[WB,WD,DQ], PUNPCKL[BW,WD,DQ]</td>
</tr>
<tr>
<td>Logical</td>
<td>PAND, PANDN, POR, PXOR</td>
</tr>
<tr>
<td>Shift</td>
<td>PSLL[W,D,Q], PSRL[W,D,Q], PSRA[W,D]</td>
</tr>
<tr>
<td>FP and MMX state mgt</td>
<td>EMMS</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>MOV[D,Q]</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>Category</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data movement</td>
<td>MOV, MOVUPS, MOVLPS, MOVLHPS, MOVHPS, MOVHLPS, MOVMSKPS, MOVSS</td>
</tr>
<tr>
<td>Shuffle</td>
<td>SHUFPS, UNPCKHPS, UNPCKLPS</td>
</tr>
<tr>
<td>State</td>
<td>FXSAVE, FXRSTOR, STMXCSR, LDMXCSR</td>
</tr>
<tr>
<td>MMX Tech Enhancements</td>
<td>PINSRW, PEXTRW, PMULXHU, PSHUFW, PMOVMSKRB, PSAD, PAVG, PMIN, PMAX</td>
</tr>
<tr>
<td>Streaming/prefetching</td>
<td>MASKMOVQ, MOVNTQ, MOVTPS, PREFETCH, SFENCE</td>
</tr>
<tr>
<td>Conversions</td>
<td>CVTSS2SI, CVTTSS2SI, CVTSI2SS, CVTPI2PS, CVTPS2PI, CVTTPS2PI</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)