Data-Level Parallelism in SIMD and Vector Architectures
Current Trends in Architecture

• Cannot continue to leverage Instruction-Level parallelism (ILP)

• Beyond ILP: new models for managing parallelism:
  – Data-level parallelism (DLP)
  – Thread-level parallelism (TLP)
  – Request-level parallelism (RLP)
ILP, DLP

• Instead of going in the direction of complex out-of-order ILP processor

• An in-order vector processor can achieve the same performance, or more, by exploiting DLP (Data Level Parallelism)
  – With more energy efficiency
Flynn’s Taxonomy

• **SISD** Single instruction stream, single data stream
  - uniprocessors (including ILP processors)

• **SIMD** Single instruction stream, multiple data streams
  - Vector architectures
  - Multimedia extensions
  - Graphics processor units

• **MISD** Multiple instruction streams, single data stream
  - No commercial implementation

• **MIMD** Multiple instruction streams, multiple data streams
  - Tightly-coupled MIMD (thread-level parallelism)
  - Loosely-coupled MIMD (request-level parallelism)
SIMD and Vector Architectures
Introduction to SIMD

- SIMD architectures can exploit significant data-level parallelism for:
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially (compared to MIMD) and achieve parallel speedups.
SIMD Architecture

• Central controller broadcasts instructions to multiple processing elements (PEs)

✓ Only requires one controller for whole array
✓ Only requires storage for one copy of program
✓ All computations fully synchronized
Three variations of SIMD Machines

1. Vector architectures
2. SIMD extensions:
   - x86 multimedia SIMD extensions: MMX 1996, SSE (Streaming SIMD Extension), AVX (Advanced Vector Extension)
3. Graphics Processor Units (GPUs)
Vector Architectures

• Basic idea:
  – Load sets of data elements into “vector registers”
  – Operate on those registers
  – Disperse the results back into memory

• A single instruction operates on vectors of data
  – Synchronized units: single Program Counter
  – Which results in dozens of register-to-register operations
  – Used to hide memory latency (memory latency occurs one per vector load/store vs. one per element load/store).
  – Leverage memory bandwidth
Vector Architectures

From Cray-1, 1976:

Scalar Unit + Vector Extensions
• Load/Store Architecture
• Vector Registers
• Vector Instructions
• Hardwired Control
• Highly Pipelined Functional Units
• Interleaved Memory System
• No Data Caches
• No Virtual Memory
Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"
Properties of Vector Processors

• Each result independent of previous result
  => long pipeline, compiler ensures no dependencies
  => high clock rate

• Vector instructions access memory with known pattern
  => highly interleaved memory
  => amortize memory latency of over 64 elements
  => no (data) caches required! (Do use instruction cache)

• Reduces branches and branch problems in pipelines

• Single vector instruction implies lots of work (loop)
  => fewer instruction fetches
Styles of Vector Architectures

- **Memory-memory vector processors:** all vector operations are memory to memory
- **Vector-register processors:** all vector operations between vector registers (except load and store)
  - Vector equivalent of load-store scalar architectures
  - Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC
Components of Vector Processors

• **Vector Register**: fixed length bank holding a single vector
  – has at least 2 read and 1 write ports
  – typically 8-32 vector registers, each holding 64-128 bit elements

• **Vector Functional Units (FUs)**: fully pipelined, start new operation every clock
  – typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit

• **Vector Load-Store Units (LSUs)**: fully pipelined unit to load or store a vector; may have multiple LSUs

• **Scalar Registers**: single element for FP scalar or address

• **Cross-bar** to connect FUs, LSUs, registers
Scalar Registers vs Vector Registers

16 Scalar Registers:
- each register holds a 32-bit element

16 Vector Registers:
- each vector register holds VLRMAX elements,
  32-bit per element
Vector Arithmetic Instructions

ADDV v3, v1, v2
Example architecture: VMIPS

- Loosely based on Cray-1
- **Vector registers**
  - 8 registers. Each register holds a 64-element, 64 bits/element vector
  - Register file has (at least) 16 read ports and 8 write ports
- **Vector functional units**
  - Fully pipelined so they can start a new operation every cycle
- **Vector load-store unit**
  - Fully pipelined, one word per clock cycle after initial memory latency
- **Scalar registers**
  - 32 general-purpose registers
  - 32 floating-point registers
• 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.

• The vector and scalar registers have a significant number of read and write ports.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV.D</td>
<td>V1, V2, V3</td>
<td>Add elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>ADDVS.D</td>
<td>V1, V2, F0</td>
<td>Add F0 to each element of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBV.D</td>
<td>V1, V2, V3</td>
<td>Subtract elements of V3 from V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBVS.D</td>
<td>V1, V2, F0</td>
<td>Subtract F0 from elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBSV.D</td>
<td>V1, F0, V2</td>
<td>Subtract elements of V2 from F0, then put each result in V1.</td>
</tr>
<tr>
<td>MULV.D</td>
<td>V1, V2, V3</td>
<td>Multiply elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>MULVS.D</td>
<td>V1, V2, F0</td>
<td>Multiply each element of V2 by F0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVV.D</td>
<td>V1, V2, V3</td>
<td>Divide elements of V2 by V3, then put each result in V1.</td>
</tr>
<tr>
<td>DIVVS.D</td>
<td>V1, V2, F0</td>
<td>Divide elements of V2 by F0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVSV.D</td>
<td>V1, F0, V2</td>
<td>Divide F0 by elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
<td>Load vector register V1 from memory starting at address R1.</td>
</tr>
<tr>
<td>SV</td>
<td>R1, V1</td>
<td>Store vector register V1 into memory starting at address R1.</td>
</tr>
<tr>
<td>LVWS</td>
<td>V1, (R1, R2)</td>
<td>Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).</td>
</tr>
<tr>
<td>SVWS</td>
<td>(R1, R2), V1</td>
<td>Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).</td>
</tr>
<tr>
<td>LVI</td>
<td>V1, (R1 + V2)</td>
<td>Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).</td>
</tr>
<tr>
<td>SVI</td>
<td>(R1 + V2), V1</td>
<td>Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).</td>
</tr>
<tr>
<td>CVI</td>
<td>V1, R1</td>
<td>Create an index vector by storing the values 0, 1 × R1, 2 × R1, …, 63 × R1 into V1.</td>
</tr>
<tr>
<td>S--VV.D</td>
<td>V1, V2</td>
<td>Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.</td>
</tr>
<tr>
<td>S--VS.D</td>
<td>V1, F0</td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>R1, VM</td>
<td>Count the 1s in vector-mask register VM and store count in R1.</td>
</tr>
<tr>
<td>CVM</td>
<td></td>
<td>Set the vector-mask register to all 1s.</td>
</tr>
<tr>
<td>MTC1</td>
<td>VLR, R1</td>
<td>Move contents of R1 to vector-length register VLR.</td>
</tr>
<tr>
<td>MFC1</td>
<td>R1, VLR</td>
<td>Move the contents of vector-length register VLR to R1.</td>
</tr>
<tr>
<td>MVTM</td>
<td>VM, F0</td>
<td>Move contents of F0 to vector-mask register VM.</td>
</tr>
<tr>
<td>MVFM</td>
<td>F0, VM</td>
<td>Move contents of vector-mask register VM to F0.</td>
</tr>
</tbody>
</table>

**Figure 4.3** The VMIPS vector instructions, showing only the double-precision floating-point operations. In
Accommodating varying data sizes

- Vector processors are good for several applications (scientific applications, but also media applications)
  - Because they can adapt to several width: a vector size can be seen as 64 64-bit elements, or 128 32-bit elements etc.
DAXPY operation

• DAXPY operation, in scalar vs vector MIPS
• DAXPY stands for: double precision a X plus Y
  – i.e. \( Y = a \times X + Y \)

```c
for (i=0; i<64, i++)
{
    Y[i]=a*X[i]+Y[i];
}
```
Scalar version of DAXPY
(assume that Rx and Ry are holding the addresses of X and Y)

L.D          F0, a            ; load scalar a
DADDIU       R4, Rx, #512    ; last address to load
Loop:
  L.D          F2, 0(Rx)      ; load X[i]
  MUL.D        F2, F2, F0     ; a * X[i]
  L.D          F4, 0(Rx)      ; load Y[i]
  ADD.D        F4, F4, F2     ; a * X[i] + Y[i]
  S.D          F4, 0(Rx)      ; store into Y[i]
  DADDIU       Rx, Rx, #8     ; increment index to X
  DSUBU        R20, R4, Rx    ; compute bound
  BNEZ         R20, Loop      ; check if done
Scalar version of DAXPY

(assume that Rx and Ry are holding the addresses of X and Y)

L.D   F0, a ; load scalar a
DADDIU R4, Rx, #512 ; last address to load
Loop: L.D   F2, 0(Rx) ; load X[i]
      MUL.D F2, F2, F0
      L.D   F4, 0(Rx)
      ADD.D F4, F4, F2
      S.D   F4, 0(Rx)
      DADDIU Rx, Rx, #8 ; increment index to X
      DSUBU R20, R4, Rx ; compute bound
      BNEZ R20, Loop ; check if done

Scalar version of DAXPY:
8 instructions per iteration
=> (64 x 8) + 2 = 514 instructions per loop
plus stalls
Analysis of scalar version (1)

Loop:

- **L.D** F2, 0(Rx) ; load X[i]
- **MUL.D** F2, F2, F0 ; a * X[i]
- **L.D** F4, 0(Rx) ; load Y[i]
- **ADD.D** F4, F4, F2 ; a * X[i] + Y[i]
- **S.D** F4, 0(Rx) ; store into Y[i]
- **DADDIU** Rx, Rx, #8 ; increment index to X
- **DSUBU** R20, R4, Rx ; compute bound
- **BNEZ** R20, Loop ; check if done

Computation overhead 25
Analysis of scalar version (2)

Scalar version stalls at EVERY iteration
VMIPS Instructions

- **ADDVV.D**: add two vectors
- **MULVS.D**: multiply vector to a scalar
- **LV/SV**: vector load and vector store from memory address
- Vector processor version of DAXPY

```
L.D F0,a ; load scalar a
LV V1,Rx ; load vector X
MULVS.D V2,V1,F0 ; vector-scalar multiply
LV V3,Ry ; load vector Y
ADDVV.D V4,V2,V3 ; add two vectors
SV Ry,V4 ; store the result
```

- Requires 6 instructions per loop vs. almost 600 for MIPS: greatly decreased!
Analysis of advantages vs. scalar version

- Requires 6 instructions per loop vs. almost 600 instructions per loop for MIPS: greatly decreased!
- No branches anymore!
- The scalar version can try to get a similar effect by loop unrolling, but it cannot get the instruction count decrease.
- Pipeline stalls greatly decreased
  - Vector version: must stall ONLY for THE FIRST vector element, after that, results can come out every clock cycle
Operation Chaining

• Results from FU forwarded to next FU in the chain
• Concept of forwarding extended to vector registers:
  – A vector operation can start as soon as the individual elements of its vector source operand become available
  – Even though a pair of operations depend on one another, chaining allows the operations to proceed in parallel on separate elements of the vector
Operation Chaining (2)

• Without chaining: must wait for last element of result to be written before starting dependent instruction
• With chaining: a dependent operation can start as soon as the individual elements of its vector source operand become available

Load → Mul → Add

LV v1
MULV v3, v1, v2
ADDV v5, v3, v4

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies

- VMIPS functional units consume one element per clock cycle
  - So, the execution time of one vector instruction is approximately the vector length
Convoys

- Simplification: to introduce the notion of *convoy*
  - Set of vector instructions that could potentially execute together (no structural hazards)
- Sequences with read-after-write dependency hazards can be in the same convoy via *chaining*
Chimes

- **Chime** is a timing metric corresponding to the unit of time to execute one convoy
  - *m convoys* execute in *m chimes*
  - Simply stated: for a vector length of *n*, and *m* convoys in a program, *n x m* clock cycles are required
  - Chime approximation ignores some processor-specific overheads
Example

LV V1,Rx ; load vector X
MULVS.D V2,V1,F0 ; vector-scalar multiply
LV V3,Ry ; load vector Y
ADDVV.D V4,V2,V3 ; add two vectors
SV Ry,V4 ; store the sum

Convoys:
1 LV MULVS.D (chaining)
2 LV ADDVV.D (chaining)
3 SV (struct. hazards with 2nd LV)

3 convoys (3 chimes); 2 FP ops per result; 1.5 cycles per FLOP
For 64 element vectors, requires 64 x 3 = 192 clock cycles
Multiple Lanes

- Instead of generating an element per cycle in one lane, spread the elements of the two vector operands into multiple lanes to improve vector performance.

**SINGLE ADD PIPELINE:**
- 1 add per cycle
- 64 cycles for a vector of 64 elements

**FOUR ADDs PIPELINE:**
- 4 adds per cycle
- 16 cycles for a vector of 64 elements
Multiple Lanes

Vector Unit with 4 lanes

Vector registers are divided across 4 lanes
Vector Length Control

• The *Maximum Vector Length (MVL)* is the physical length of vector registers in a machine (64 in our VMIPS example)

• What do you do when the vector length in a program is not exactly 64?
  – Vector length *smaller* than 64
  – Vector length *unknown* at compile time and maybe *greater* than MVL
Vector length smaller than 64

for (i=0; i<63; i=i+1)
  Y[i]=a * X[i] + Y[i];

for (i=0; i<31; i=i+1)
  Y[i]=a * X[i] + Y[i];

• There is a special register, called vector-length register (VLR)
• The VLR controls the length of any vector operation (including vector load/store).
• It can be set to any value smaller than the MVL (64)
Restructure the code using a technique called *strip mining*:
- Sort of loop unrolling where the length of first segment is the remainder and all subsequent segments are of length MVL.
Vector length unknown at compile time

for (i=0; i<n; i=i+1)
    Y[i]=a * X[i] + Y[i];

Restructure the code using a technique called \textit{strip mining}:
- Code generation technique such that each vector operation is done for a size less than or equal to MVL
- Sort of loop unrolling where the length of the first segment is \((n \mod \text{MVL})\) and all subsequent segments are of length MVL
Vector length unknown at compile time

For (i=0; i<n; i=i+1)
Y[i] = a * X[i] + Y[i];

Restructure the code using a technique called *strip mining*

low = 0;
VL = (n % MVL);
for (j = 0; j <= (n/MVL); j=j+1) {
  for (i = low; i < (low+VL); i=i+1)
    Y[i] = a * X[i] + Y[i];
  low = low + VL;
  VL = MVL;
}

/* find odd-size piece using modulo op % */
/* outer loop */
/* runs for length VL */
/* main operation */
/* start of next vector */
/* reset the length to maximum vector length */

<table>
<thead>
<tr>
<th>Value of j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>...</th>
<th>n/MVL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt; m &gt;</td>
<td>&lt; MVL &gt;</td>
<td>&lt; MVL &gt;</td>
<td>&lt; MVL &gt;</td>
<td>&lt; MVL &gt;</td>
<td>&lt; MVL &gt;</td>
<td>&lt; MVL &gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Range of i</th>
<th>0</th>
<th>m</th>
<th>(m+MVL)</th>
<th>(m+2×MVL)</th>
<th>...</th>
<th>...</th>
<th>(n−MVL)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(m−1)</td>
<td>(m−1)</td>
<td>...</td>
<td>(m−1)</td>
<td>(n−1)</td>
</tr>
</tbody>
</table>
Vector Mask Registers

for (i = 0; i < 64; i=i+1)
    if (X[i] != 0)
        X[i] = X[i] – Y[i];

Control Dependence in a loop
This loop cannot normally be vectorized because of the if clause inside it

• Use vector mask register to “disable” some elements:
• The vector-mask control uses a Boolean vector of length MVL to control the execution of a vector instruction
• When vector mask registers are enabled, any vector instruction operates ONLY on the vector elements whose corresponding masks bits are set to 1
Vector Mask Registers

for (i = 0; i < 64; i=i+1)
  if (X[i] != 0)
    X[i] = X[i] – Y[i];

This loop cannot normally be vectorized because of the if clause inside it

Use vector mask register to “disable” elements:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV V1,Rx</td>
<td>load vector X into V1</td>
</tr>
<tr>
<td>LV V2,Ry</td>
<td>load vector Y into V2</td>
</tr>
<tr>
<td>L.D F0,#0</td>
<td>load FP zero into F0</td>
</tr>
<tr>
<td>SNEVS.D V1,F0</td>
<td>sets VM(i) to 1 if V1(i)!=F0</td>
</tr>
<tr>
<td>SUBVV.D V1,V1,V2</td>
<td>subtract under vector mask</td>
</tr>
<tr>
<td>SV Rx,V1</td>
<td>store the result in X</td>
</tr>
</tbody>
</table>

The cycles for non-executed operation elements are lost
But the loop can still be vectorized!
Stride

- How do you do with non-adjacent memory elements?
- The *stride* is the distance separating memory elements that are to be gathered into a single register.

When a matrix is allocated in memory, it is linearized and laid out in row-major order in C => the elements in the columns are not-adjacent in memory.
Stride

- When the elements of a matrix in the inner loop are accessed by column => they are separated in memory by a stride equal to the row size times 8 bytes per entry
- We need an instruction LVWS to load elements of a vector that are non-adjacent in memory from address R1 with stride R2:
  \[ \text{LVWS V1, (R1, R2)} \quad ; \quad \text{V1 <= M[R1 + i*R2]} \]
- Example:  \[ \text{LVWS V1, (C, 100)} \quad ; \quad \text{V1 <= M[C + i*100]} \]
  while \[ \text{LV V2, B} \quad ; \quad \text{V2 <= M[B]} \]
Stride

- Consider:
  for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
      A[i][j] = 0.0;
      for (k = 0; k < 100; k=k+1)
    }

- Must vectorize multiplication of rows of B with columns of D
- Use *non-unit stride for columns of D*
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride,#banks) < bank busy time
Scatter-Gather

- Primary mechanism to support sparse matrices using index vectors. Consider:
  
  ```
  for (i = 0; i < n; i=i+1)
      A[K[i]] = A[K[i]] + C[M[i]];
  ```

- Use index vector K and M to indicate the nonzero elements of A and C (A and C must have the same number of nonzero elements).

```plaintext
 LV    Vk, Rk ;load K
LVI    Va, (Ra+Vk) ;load A[K[]]
 LV    Vm, Rm ;load M
LVI    Vc, (Rc+Vm) ;load C[M[]]
ADDVVV.D Va, Va, Vc ;add them
SVI   (Ra+Vk), Va ;store A[K[]]
```
SIMD Instruction Set Extensions

• Multimedia applications operate on data types narrower than the native word size
  – Example: disconnect carry chains to “partition” adder

• Limitations, compared to vector instructions:
  – Number of data operands encoded into op code
  – No sophisticated addressing modes (strided, scatter-gather)
  – No mask registers
SIMD Implementations

• Implementations:
  – Intel MMX (1996)
    • Eight 8-bit integer ops or four 16-bit integer ops
  – Streaming SIMD Extensions (SSE) (1999)
    • Eight 16-bit integer ops
    • Four 32-bit integer/float ops or two 64-bit integer/float ops
  – Advanced Vector Extensions (2010)
    • Four 64-bit integer/float ops
  – Operands must be consecutive and aligned memory locations