




Data-Level Parallelism in Vector and SIMD Architectures



Flynn Taxonomy of Computer Architectures (1972)

It is based on parallelism of instruction streams and data streams

- SISD
 - ▣ single instruction stream, single data stream • microprocessors
- SIMD
 - ▣ single instruction stream, multiple data streams
 - ▣ vector processors; principle behind multimedia extensions
 - ▣ graphic processing units (GPUs)
- MISD
 - ▣ multiple instruction streams, single data stream
 - ▣ not commercial processors (yet)
- MIMD
 - ▣ multiple instruction streams, multiple data streams
 - ▣ each processor fetches its own instruction and operates on its own data

SISD architecture

- Le SISD architectures sono quelle classiche nelle quali non è previsto **nessun grado di parallelismo** né tra le istruzioni né tra i dati.

MISD architecture

- MISD è una architettura abbastanza inusuale nella quale più istruzioni concorrenti operano sullo stesso flusso di dati.
- Un campo di applicazione possono ad esempio essere i **sistemi ridondanti**, come i sistemi di controllo degli aeroplani nei quali se uno dei processori si guasta l'elaborazione dei dati deve continuare ugualmente.

SIMD architecture

- This form of parallel processing has existed since the 1960s
- The idea is rather than executing array operations by loop, we execute all of the array operations in parallel on different processing elements (ALUs)
 - we convert `for(i=0;i<n;i++) a[i]++;` into a single operation, say `A=A+1`
- Not only do we get a speedup from the parallelism, we also get to remove the looping operation (incrementing `i`, the comparison and conditional branch)
- These technologies are often applied in the field of audio / video codecs and video games.
 - *For example, if a polygon is moved, it is necessary to translate all its vertices by adding to each of them the same value.*

MIMD

- Solitamente nella categoria MIMD si fanno rientrare i **sistemi distribuiti**, nei quali più processori autonomi operano in parallelo su dati differenti.

SIMD vs MIMD

- 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

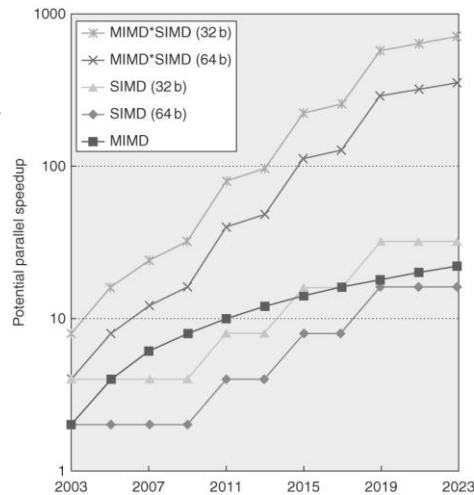
- SIMD allows programmer to continue to think sequentially

SIMD parallelism

- Vector architectures
- Multimedia SIMD instruction set extensions
- Graphics Processor Units (GPUs)

Potential speedup via parallelism over time for x86 computers.

- For x86 processors:
 - Expect two additional cores per chip per year
 - SIMD width to double every four years
 - Potential speedup from SIMD to be twice that from MIMD!



Vector Architectures

- Basic idea:
 - Read sets of data elements into “vector registers”
 - Operate on those registers
 - Disperse the results back into memory
- Registers are controlled by compiler
 - Used to hide memory latency
 - Leverage memory bandwidth

Vector Architectures

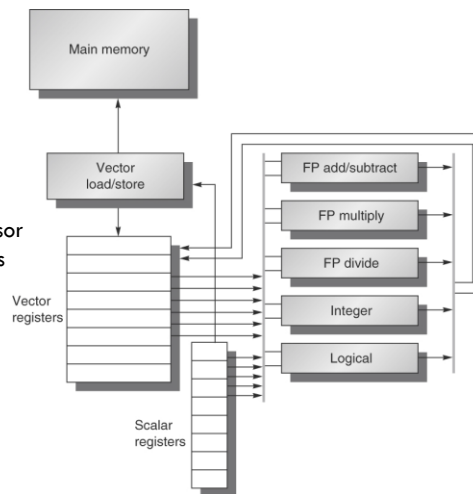
- provide high-level operations that work on vectors (linear arrays of numbers)
 - e.g. add two 64-element vectors in 1 step, instead of using a loop
- reduce IF, ID bandwidth
 - instruction represent many operations
- reduce HW complexity to support ILP
 - the computation on each element does not depend on the others
 - check hazards once for vector operand
 - since a loop is replaced by an instruction, loop branch, control hazards disappear
 - improve memory access
 - deeply-pipelined vector load/store unit a single access is initiated for the entire vector (bandwidth of one word per clock cycle after initial latency)

VMIPS

- Example architecture: VMIPS
 - Loosely based on Cray-1
 - Vector registers
 - Each register holds a 64-element, 64 bits/element vector
 - Register file has 16 read ports and 8 write ports
 - Vector functional units
 - Fully pipelined
 - Data and control hazards are detected
 - Vector load-store unit
 - Fully pipelined
 - One word per clock cycle after initial latency
 - Scalar registers
 - 32 general-purpose registers
 - 32 floating-point registers

Structure of VMIPS Vector Processor

The VMIPS 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 figure shows vector units for logical and integer operations so that VMIPS looks like a classic vector processor (Cray 1). The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. Crossbar switches (thick gray lines) connects these ports to the inputs and outputs of the vector functional units.



VMIPS Instruction Set

- Aside from the ordinary MIPS instructions (scalar operations), we enhance MIPS with the following:
 - LV, SV – load vector, store vector
 - LV V1, R1 – load vector register V1 with the data starting at the memory location stored in R1
 - also LVI/SVI for using indexed addressing mode, and LVWS and SVWS for using scaled addressing mode
 - ADDVV.D V1, V2, V3 ($V1 \leftarrow V2 + V3$)
 - ADDVS.D V1, V2, F0 (scalar addition)
 - similarly for SUB, MUL and DIV

VMIPS Instruction Set

- S--VV.D V1, V2 and S--VS.D V1, F0 to compare pairwise elements in V1 and V2 or V1 and F0
 - -- is one of EQ, NE, GT, LT, GE, LE
 - result of comparison is a set of boolean values placed into the bit vector register VM which we can then use to implement if statements
- POP R1, VM – count number of 1s in the VM and store in R1
 - this is only a partial list of instructions, and only the FP operations, see figure 4.3 for more detail, missing are any integer based operations

VMPIS instruction Set

| Instruction | Operands | Function |
|-------------|--------------|--|
| ADDVV.D | V1, V2, V3 | Add elements of V2 and V3, then put each result in V1. |
| ADDVS.D | V1, V2, F0 | Add F0 to each element of V2, then put each result in V1. |
| SUBVV.D | V1, V2, V3 | Subtract elements of V3 from V2, then put each result in V1. |
| SUBVS.D | V1, V2, F0 | Subtract F0 from elements of V2, then put each result in V1. |
| SUBSV.D | V1, F0, V2 | Subtract elements of V2 from F0, then put each result in V1. |
| MULVV.D | V1, V2, V3 | Multiply elements of V2 and V3, then put each result in V1. |
| MULVS.D | V1, V2, F0 | Multiply each element of V2 by F0, then put each result in V1. |
| DIVVV.D | V1, V2, V3 | Divide elements of V2 by V3, then put each result in V1. |
| DIVVS.D | V1, V2, F0 | Divide elements of V2 by F0, then put each result in V1. |
| DIVSV.D | V1, F0, V2 | Divide F0 by elements of V2, then put each result in V1. |
| LV | V1, R1 | Load vector register V1 from memory starting at address R1. |
| SV | R1, V1 | Store vector register V1 into memory starting at address R1. |
| LVWS | V1, (R1, R2) | Load V1 from address at R1 with stride in R2 (i.e., $R1 + i \times R2$). |
| SVWS | (R1, R2), V1 | Store V1 to address at R1 with stride in R2 (i.e., $R1 + i \times R2$). |
| LVI | V1, (R1+V2) | Load V1 with vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index). |
| SVI | (R1+V2), V1 | Store V1 to vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index). |

VMPIS instruction Set

| | | |
|---------|---------|---|
| CVI | V1, R1 | Create an index vector by storing the values $0, 1 \times R1, 2 \times R1, \dots, 63 \times R1$ into V1. |
| S--VV.D | V1, V2 | 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. |
| S--VS.D | V1, F0 | |
| POP | R1, VM | Count the 1s in vector-mask register VM and store count in R1. |
| CVM | | Set the vector-mask register to all 1s. |
| MTC1 | VLR, R1 | Move contents of R1 to vector-length register VL. |
| MFC1 | R1, VLR | Move the contents of vector-length register VL to R1. |
| MVTM | VM, F0 | Move contents of F0 to vector-mask register VM. |
| MVFM* | F0, VM | Move contents of vector-mask register VM to F0. |

Example

- Let's look at a typical vector processing problem, computing $Y = a * X + Y$
 - Where X & Y are vectors and a is a scalar (e.g., $y[i]=y[i]+a*x[i]$)
- The MIPS code is on the left and the VMIPS code is on the right

| | |
|---|---|
| <pre> Loop: L.D F0, a DADDIR4, Rx, #512 L.D F2, 0(Rx) MUL.DF2, F2, F0 L.D F4, 0(Ry) ADD.DF4, F4, F2 S.D F4, 0(Ry) DADDIRx, Rx, #8 DADDIRy, Ry, #8 DSUB R20, R4, Rx BNEZ R20, Loop </pre> | <pre> L.D F0, a LV V1, Rx MULVS.D V2, V1, F0 LV V3, Ry ADDVV.D V4, V2, V3 SV V4, Ry </pre> |
|---|---|

In MIPS, we execute 578 instructions whereas in VMIPS, only 6 (there are 64 elements in the array to process, each is 8 bytes long) and there are no RAW hazards or control hazards to deal with

VMIPS Instructions

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address

- Example: DAXPY

| | | |
|---------|----------|--------------------------|
| LD | 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 | V4,V2,V3 | ; add |
| SV | Ry,V4 | ; store the result |
- Requires 6 instructions vs. 578 for MIPS

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
 - ▣ Execution time is approximately the vector length

Convoy

- A convoy is a set of sequential vector operations that can be issued together without a structural hazard
 - Because we are operating on vectors in a pipeline, the execution of these operations can be overlapped
 - e.g., L.V V1, Rx followed by ADDVV.D V3, V1, V2 would allow us to retrieve the first element of V1 and then start the addition while retrieving the second element of V1

Chimes

- A chime is the amount of time it takes to execute a convoy
 - We will assume that there are no stalls in executing the convoy, so the chime will take $n + x - 1$ cycles where x is the length of the convoy and n is the number of data in the vector
 - A program of m convoys will take m chimes, or $m * (n + x - 1)$ cycles (again, assuming no stalls)
 - The chime time ignores pipeline overhead, and so architects prefer to discuss performance in chimes

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 |
| 2 | LV | ADDVV.D |
| 3 | SV | |

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires $64 \times 3 = 192$ clock cycles

Vector Chaining

- Vector version of register bypassing
- Without chaining, must wait for last element of result to be written before starting dependent instruction
- With chaining, can start dependent instruction as soon as first result appears

Challenges

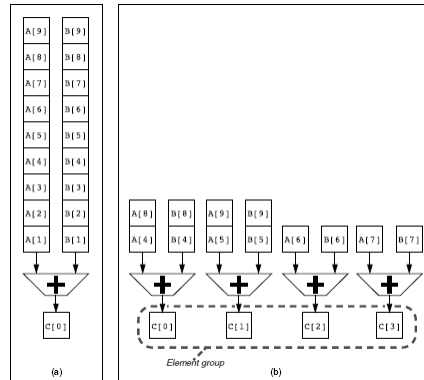
- Start up time
 - Latency of vector functional unit
 - Assume the same as Cray-1
 - Floating-point add => 6 clock cycles
 - Floating-point multiply => 7 clock cycles
 - Floating-point divide => 20 clock cycles
 - Vector load => 12 clock cycles

Challenges

- Improvements:
 - > 1 element per clock cycle
 - Non-64 wide vectors
 - IF statements in vector code
 - Memory system optimizations to support vector processors
 - Multiple dimensional matrices
 - Sparse matrices
 - Programming a vector computer

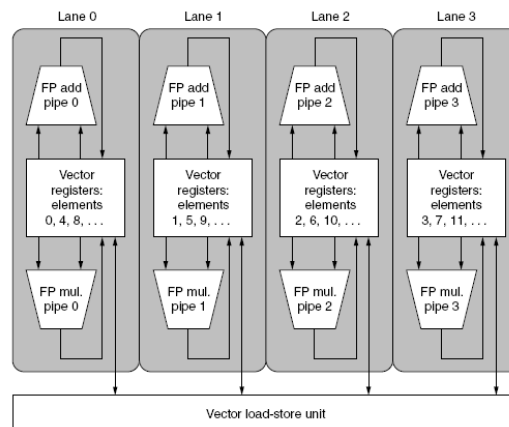
Multiple Lanes

- Element n of vector register A is “hardwired” to element n of vector register B
 - ▣ Allows for multiple hardware lanes



Multiple Lanes

- Each line contains a portion of vector register file and one execution pipeline from each vector functional unit



Vector Length Register

- Vector length not known at compile time?

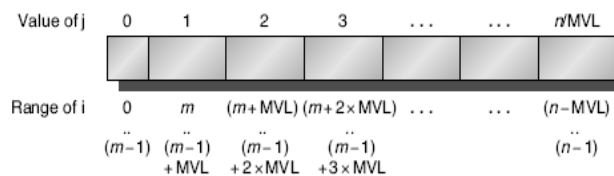

```
for ( i =0; i<n; i++)
  Y[i]=Y[i]+a*X[i];
```

 - n is know at run time
- Use Vector Length Register (VLR)
 - VLR cannot be greater than the size of the vector registers, the maximum vector length (MVL)
 - MVL determines the number of data in a vector

Vector Length Register

- Use strip mining for vectors over the maximum length:

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
  for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
    Y[i] = a * X[i] + Y[i]; /*main operation*/
  low = low + VL; /*start of next vector*/
  VL = MVL; /*reset the length to maximum vector length*/
}
```



Vector Mask Registers

- Consider:
 - for ($i = 0; i < 64; i=i+1$)
 - if ($X[i] \neq 0$)
 - $X[i] = X[i] - Y[i];$
- This loop cannot be normally vectorized
- Iteration can be vectorized for items for which $X[i] \neq 0$
- Use vector mask register (VM) to “disable” elements:
 - SNEVS.D V1,F0
 - This instruction sets $VM(i)$ to 1 if $V1(i) \neq F0$
- When VM register is enabled, vector instructions operate only on the elements with $VM(i)$ equal to one
- Clearing VM, vector instructions operate on all elements

Vector Mask Registers

| | | |
|---------|----------|---------------------------------------|
| LV | V1,Rx | ;load vector X into V1 |
| LV | V2,Ry | ;load vector Y |
| L.D | F0,#0 | ;load FP zero into F0 |
| SNEVS.D | V1,F0 | ;sets $VM(i)$ to 1 if $V1(i) \neq F0$ |
| SUBVV.D | V1,V1,V2 | ;subtract under vector mask |
| SV | Rx,V1 | ;store the result in X |

GFLOPS rate decreases!

Memory Banks

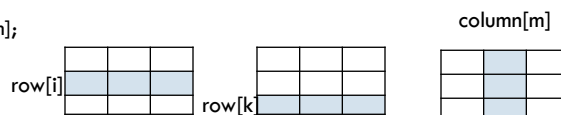
- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
 - ▣ Control bank addresses independently
 - ▣ Load or store non sequential words
 - ▣ Support multiple vector processors sharing the same memory
- Example:
 - ▣ 32 processors, each generating 4 loads and 2 stores/cycle
 - ▣ Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
 - ▣ How many memory banks needed?

Stride

- Consider:

```

for (j = 0; j < 64; j=j+1)
  A[i][j] = B[k][j] * D[j][m];
}
  
```



```

LV      V1, RB          ; RB contains address of row B[k]
LVWS   V2, (RD,R2)     ; RD contains address of D[0][m] and R2 contains row size
MULTW  V3,V1,V2
SW     RA, V3          ; RA contains address of row B[k]
  
```

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

Scatter-Gather

- Consider:

for ($i = 0; i < n; i=i+1$)

$A[K[i]] = A[K[i]] + C[M[i]];$

- Use index vector:

| | | |
|---------|-------------|---------------|
| LV | Vk, Rk | ;load K |
| LVI | Va, (Ra+Vk) | ;load A[K[]] |
| LV | Vm, Rm | ;load M |
| LVI | Vc, (Rc+Vm) | ;load C[M[]] |
| ADDVV.D | Va, Va, Vc | ;add them |
| SVI | (Ra+Vk), Va | ;store A[K[]] |

Programming Vec. Architectures

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler

| Benchmark name | Operations executed in vector mode, compiler-optimized | Operations executed in vector mode, with programmer aid | Speedup from hint optimization |
|----------------|--|---|--------------------------------|
| BDNA | 96.1% | 97.2% | 1.52 |
| MG3D | 95.1% | 94.5% | 1.00 |
| FLO52 | 91.5% | 88.7% | N/A |
| ARC3D | 91.1% | 92.0% | 1.01 |
| SPEC77 | 90.3% | 90.4% | 1.07 |
| MDG | 87.7% | 94.2% | 1.49 |
| TRFD | 69.8% | 73.7% | 1.67 |
| DYFESM | 68.8% | 65.6% | N/A |
| ADM | 42.9% | 59.6% | 3.60 |
| OCEAN | 42.8% | 91.2% | 3.92 |
| TRACK | 14.4% | 54.6% | 2.52 |
| SPICE | 11.5% | 79.9% | 4.06 |
| QCD | 4.2% | 75.1% | 2.15 |

SIMD Extensions

- Media 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

Esempi di estensioni di tipo SIMD

- Le più diffuse sono:
 - Apple/IBM/Freescale AltiVec
 - Intel MMX/SSE/SSE2/SSE3/SSSE3
 - AMD 3DNow!
 - SPARC VIS
 - ARM Neon/VFP
 - MIPS MDMX/MIPS-3D

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/fp ops or two 64-bit integer/fp ops
 - Advanced Vector Extensions (2010)
 - Four 64-bit integer/fp ops

- Operands must be consecutive and aligned memory locations