Advanced Parallel Architecture

Annalisa Massini - 2014/2015

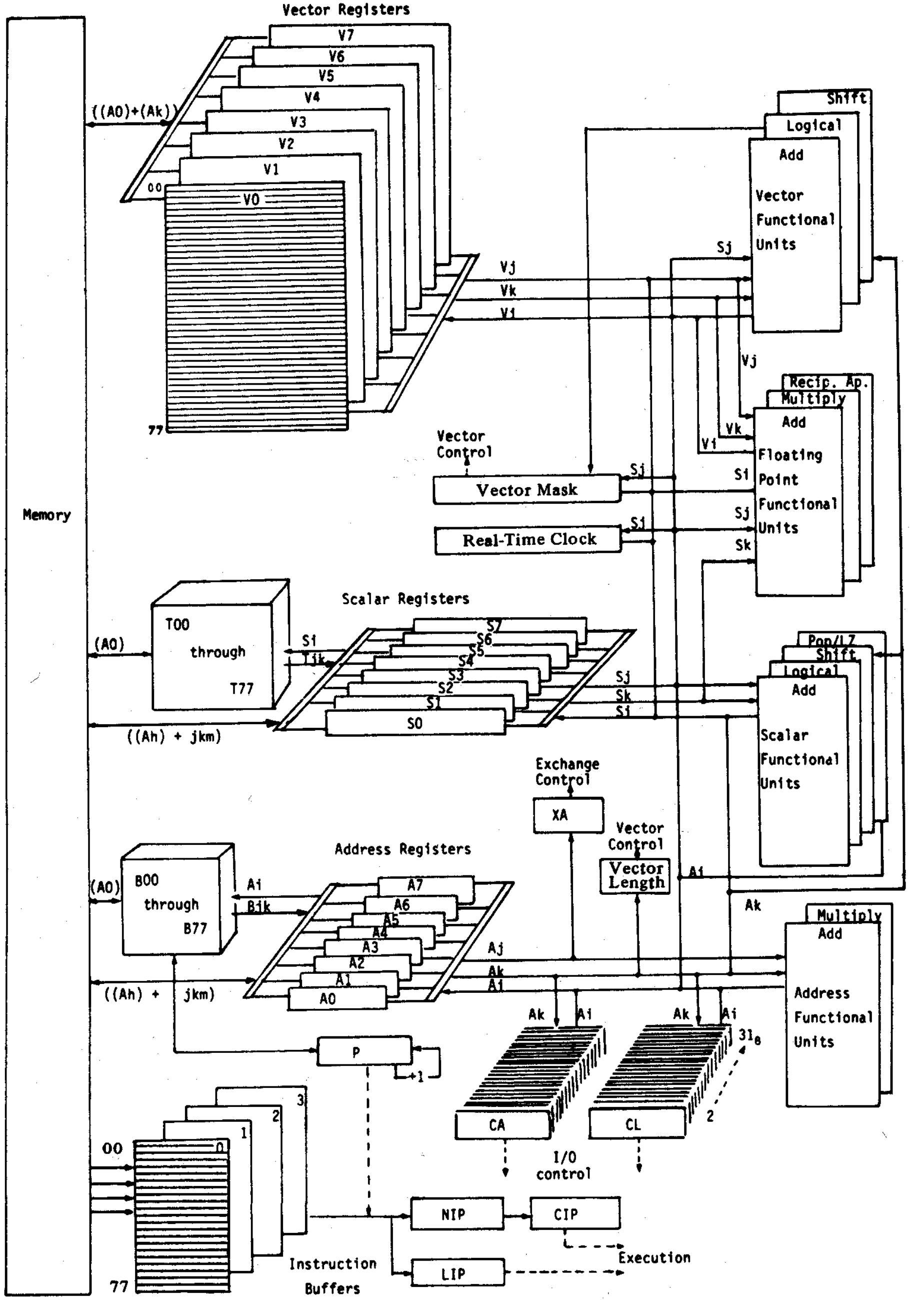
Vector architecture

Computer Architecture - A Quantitative Approach, Fifth Edition

Hennessy Patterson

- Chapter 4 - Data-Level Parallelism in Vector, SIMD, and GPU Architectures
  - **Section 4.2** - **Vector Architecture**

Cray 1


Vector Registers
V7
V6
V5
V4
V3
V2
V1
V0
00
77
((A0)+(Ak))
Memory
Vj
Vk
Vi
Shift
Logical
Add
Vector
Functional
Units
Sj
Vj
Recip. Ap.
Multiply
Add
Floating
Point
Functional
Units
Vi
Vk
Si
Sj
Sk
Vector
Control
Vector Mask
Real-Time Clock
Sj
Si
Scalar Registers
T00
through
T77
(A0)
Si
Tjk
S7
S6
S5
S4
S3
S2
S1
S0
((Ah) + jkm)
Sj
Sk
Si
Pop/LZ
Shift
Logical
Add
Scalar
Functional
Units
Exchange
Control
XA
Vector
Control
Vector
Length
Address Registers
B00
through
B77
(A0)
Ai
Bjk
A7
A6
A5
A4
A3
A2
A1
A0
((Ah) + jkm)
Aj
Ak
Ai
Ai
Ak
Multiply
Add
Address
Functional
Units
P
+1
Ak
Ai
Ak
Ai
31_8
CA
CL
2
I/O
control
00
0
1
2
3
77
Instruction
Buffers
NIP
CIP
LIP
Execution

Vector Architectures

- Basic idea:
  - Read sets of data elements scattered about memory
  - Place them into **vector registers**
  - Operate on those registers
  - Disperse the results back into memory

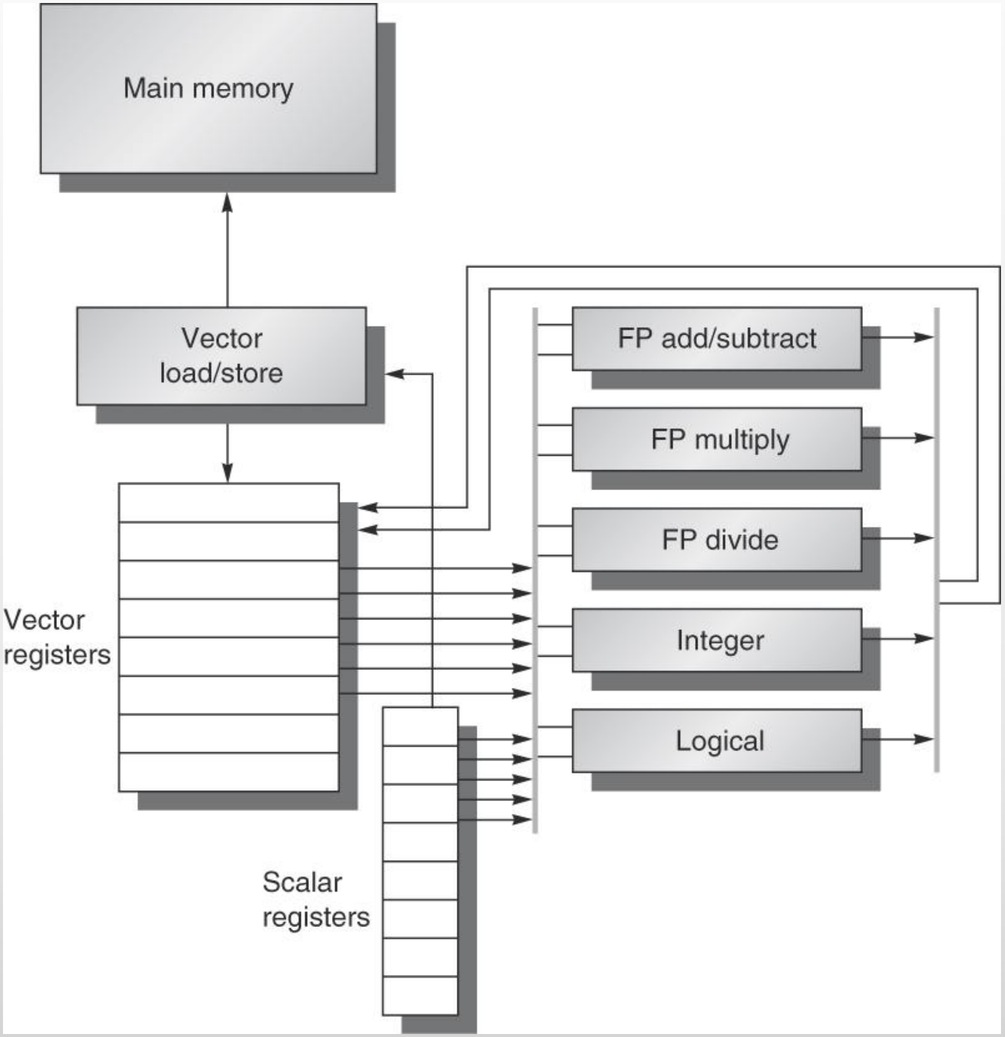
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

- Since vector loads and stores are deeply pipelined, the program pays the long memory latency only once per vector load or store versus once per element, thus amortizing the 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

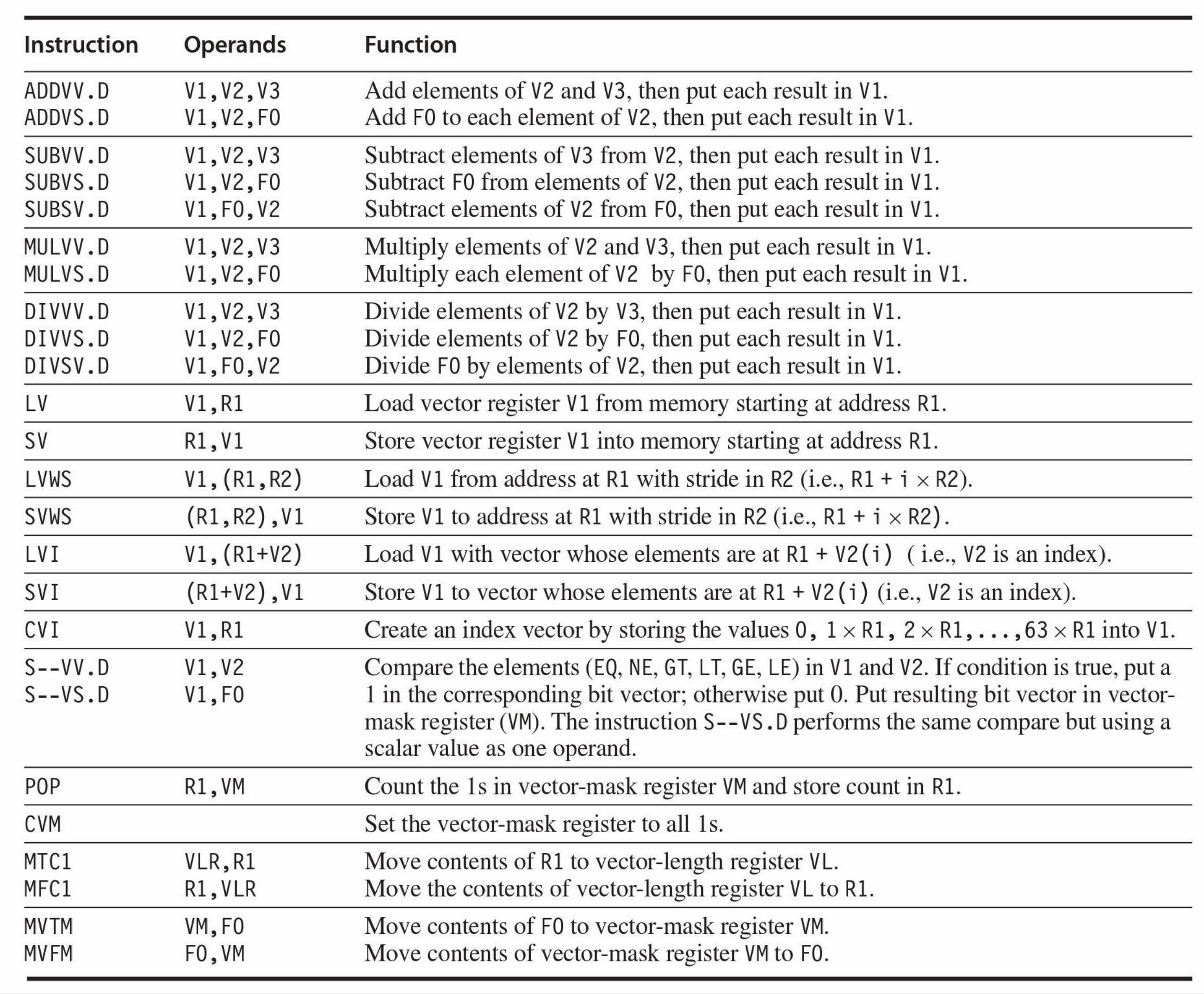
VMIPS



Basic structure of VMIPS vector architecture :

- scalar architecture just like MIPS
- eight 64-element vector
- all the functional units are vector functional units
- vector units for logical and integer operations
- 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

VMIPS Instructions



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 × R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., R1 + i × 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).
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--VV.D S--VS.D	V1,V2 V1,F0	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.
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.

DAXPY in MIPS Instructions

Example: DAXPY (double precision a*X+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 X[i]
        L.D       F4,0(Ry)      ; load Y[i]
        ADD.D     F4,F2,F2      ; a x X[i] + Y[i]
        S.D       F4,9(Ry)      ; store into Y[i]
        DADDIU    Rx,Rx,#8      ; increment index to X
        DADDIU    Ry,Ry,#8      ; increment index to Y
        SUBBU     R20,R4,Rx     ; compute bound
        BNEZ      R20,Loop      ; check if done
```

- Requires almost 600 MIPS ops

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 (double precision a*X+Y)

```
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      V4,V2,V3    ; add
SV         Ry,V4       ; store the result
```

- Requires 6 instructions

Vector Execution Time

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

- We can compute the time for a single vector instruction given
  - the vector length and
  - the *initiation rate,* rate at which a vector unit consumes new operands and produces new results
- Assuming initiation rate of one element per clock cycle for individual operations we obtain that **the execution time is approximately the vector length**

Convoy

- To discuss vector execution and vector performance, we use the notion of *Convoy*
  - Set of vector instructions that could potentially execute together

- We can estimate performance of a section of code by counting the number of convoys
  - the instructions in a convoy *must not contain any structural hazards;*
  - if such hazards were present, the instructions would need to be serialized and initiated in different Convoys
  - to simplify, we assume that a convoy of instructions must complete execution before any other instructions (scalar or vector) can begin execution

Chaining

- Sequences with read-after-write dependency hazards can be in the same convey via *chaining*

- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
  - The results from the first functional unit in the chain are “forwarded” to the second functional unit
  - Early implementations of chaining worked just like forwarding in scalar pipelining
  - Recent implementations use ***flexible chaining,*** which allows a vector instruction to chain to any other active vector instruction, assuming we don’t generate a structural hazard

Chimes

- To turn convoys into execution time we need a timing metric to estimate the time for a convoy: *Chime*
  - Unit of time to execute one convoy
  - A vector sequence that consists of *m* convoys executes in *m* chimes
  - For vector length of *n*, requires approximately *m* x *n* clock cycles
  - The chime approximation ignores some processor-specific overheads, many of which are dependent on vector length.
  - Measuring time in chimes is a better approximation for long vectors than for short ones

Chime

- The most important source of overhead ignored by the chime model is vector *start-up time*
- *Start-up time* is determined by the pipelining latency of vector functional unit
  - Assuming the same pipeline depths 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

Optimizations

- *Multiple Lanes*: > 1 element per clock cycle
- *Vector Length Registers*: Non-64 wide vectors
- *Vector Mask Registers*: IF statements in vector code
- *Memory Banks*: Memory system optimizations to support vector processors
- *Stride*: Multiple dimensional matrices
- *Scatter-Gather*: Sparse matrices
- *Programming Vector Architectures*: Program structures affecting performance

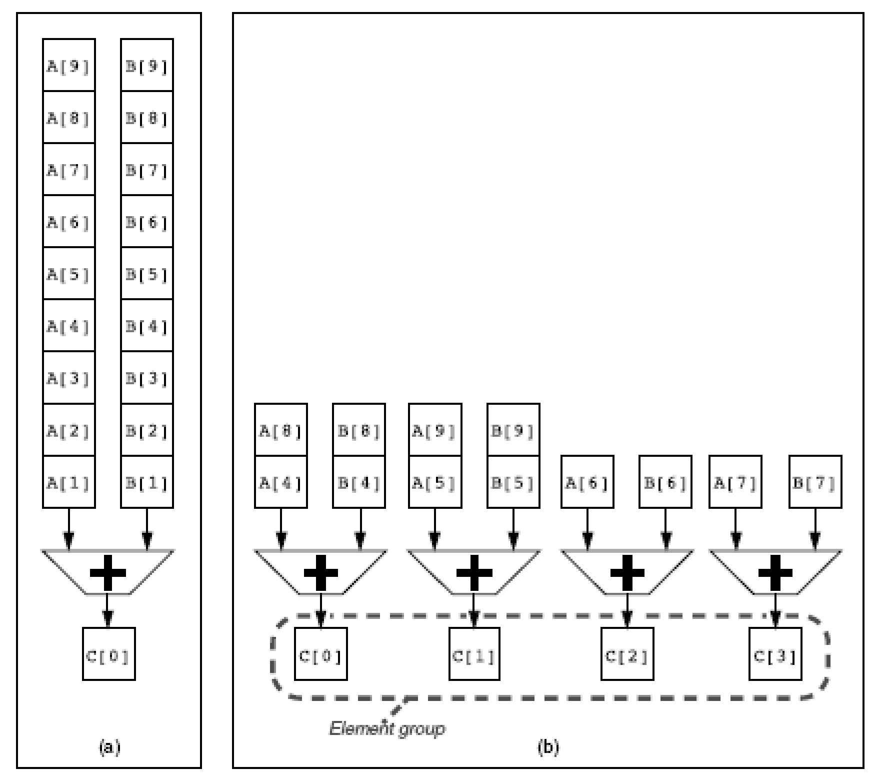
Multiple Lanes

- The advantage of a vector instruction set is that it allows software to pass a large amount of parallel work to hardware using only **a single short instruction**

- The parallel semantics of a vector instruction allow an implementation to execute these elemental operations using:
  - a **deeply pipelined functional unit**
  - an **array of parallel functional units**
  - a **combination** of parallel and pipelined functional units

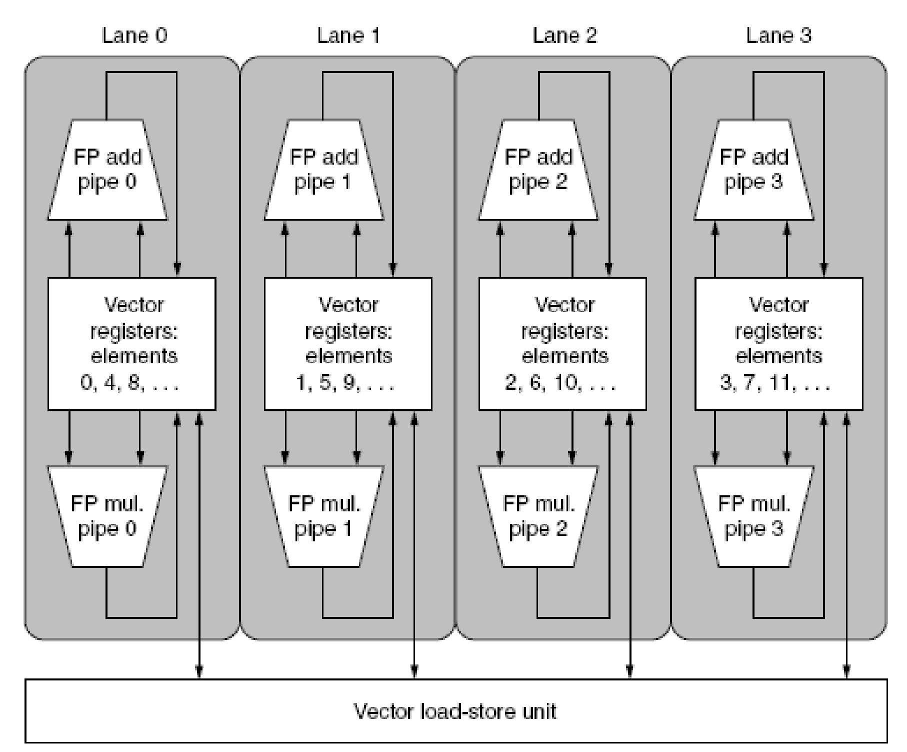
Multiple Lanes

- Figure illustrates how to improve vector performance by using **parallel pipelines** to execute a **vector add instruction**



Multiple Lanes

- In the VMIPS instruction set, all vector arithmetic instructions only allow element *N* of one vector register to take part in operations with element *N* from other vector registers
- A parallel vector unit can be build by multiple parallel *lanes*


Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, . . .
Vector registers: elements 1, 5, 9, . . .
Vector registers: elements 2, 6, 10, . . .
Vector registers: elements 3, 7, 11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

Vector Length Registers Handling Loops Not Equal to 64

- Real vector length ***n*** in a program is unlikely to match VMIPS vector length (which is 64)
- Vector length is not known at compile time
- The solution is to create a ***vector-length register (VLR):***
  - controls the length of any vector operation, including a vector load or store.
  - **but** the value in the VLR cannot be greater than the length of the vector registers
- Then we use also the ***maximum vector length (MVL):***
  - *determines the number of* data elements in a vector of an architecture

Vector Length Registers Handling Loops Not Equal to 64

- If the value of *n* is greater than the MVL, a technique called ***strip mining** is used:*
  - Generation of code such that each vector operation is done for a size less than or equal to the MVL:
  - one loop that handles any number of iterations that is a multiple of the MVL
  - another loop that handles any remaining iterations and must be less than the MVL
- In practice, compilers usually create a single strip-mined loop that is parameterized to handle both portions by changing the length.

Vector Length Registers Handling Loops Not Equal to 64

- For example, consider the code for DAXPY:

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

- the strip-mined version of the DAXPY loop in C:

```
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*/
}
```

- The length of the first segment is (n % MVL), and all subsequent segments are of length MVL

Vector Mask Registers IF Statements in Vector Loops

- The presence of conditionals (**IF statements**) inside loops introduce **control dependencies** into the loop
- Consider:

```
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 conditional execution of the body
- If the inner loop could be run for the **iterations for which X[i]≠0**, then the subtraction could be vectorized

Vector Mask Registers IF Statements in Vector Loops

- The solution is ***vector-mask control***
- **Mask registers** provide conditional execution of vector instruction
- When the **vector-mask register is enabled**, any vector instructions operate only on the vector elements whose corresponding entries in the vector-mask register are 1
- Use vector mask register to “disable” elements (*if conversion*):

```
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)!=F0
SUBVV.D     V1,V1,V2      ;subtract under vector mask
SV          Rx,V1         ;store the result in X
```

- GFLOPS rate decreases!

Memory Banks Bandwidth for Vector Load/Store Units

- Memory system must be designed to support high bandwidth for vector loads and stores

- **Spreading accesses** across multiple independent **memory banks** usually delivers the desired rate
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory

Stride *Handling Multidimensional Arrays in Vector Architectures*

- The position in memory of adjacent elements in a vector may not be sequential
- 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)
               A[i][j] = A[i][j] + B[i][k] * D[k][j];
      }
```

- Must vectorize multiplication of rows of B with columns of D
- An array in memory is linearized in either row-major (as in C) or column-major (as in Fortran) order, then either the elements in the row or in the column **are not adjacent in memory**

Stride *Handling Multidimensional Arrays in Vector Architectures*

- For vector processors, the technique to fetch elements of a vector that are not adjacent in memory exploits the **Stride**
  - The **distance separating elements** to be gathered into a single register
  - In our example, matrix D has a stride of 100 double words (800 bytes), and matrix B has a stride of 1 double word (8 bytes). For column-major order, the strides would be reversed.
- A vector processor can handle strides greater than one, called **non-unit strides**, ***using only vector load and vector store operations*** with stride capability.
- This ability to access nonsequential memory locations and to reshape them into a dense structure is one of the major advantages of a vector processor.

Stride *Handling Multidimensional Arrays in Vector Architectures*

- Example:

  8 memory banks with a bank busy time of 6 cycles and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

- **Answer:**
  - Stride of 1: number of banks is greater than the bank busy time, so it takes
    - 12+64 = 76 clock cycles → 1.2 cycle per element
  - Stride of 32: the worst case scenario happens when the stride value is a multiple of the number of banks, which this is! Every access to memory will collide with the previous one! Thus, the total time will be:
    - 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clock cycles per element!

Scatter-Gather Handling Sparse Matrices

- It is important to have techniques to allow programs with **sparse matrices** to execute in vector mode
- In a sparse matrix, the elements of a vector are usually stored in some compacted form and then accessed indirectly
- Consider sparse vectors A and C, and vector indices K and M, where A and C have the same number (n) of non-zeros:

```
for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

- The primary mechanism for supporting sparse matrices is ***gather-scatter operations*** using index vectors
- Such operations support moving between a compressed representation and normal representation of a sparse matrix

Scatter-Gather Handling Sparse Matrices

- A ***gather operation*** takes an index vector and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector.

  The result is a dense vector in a vector register

- After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a ***scatter store,*** using the same index vector.

- Hardware support for such operations is called *gather-scatter*

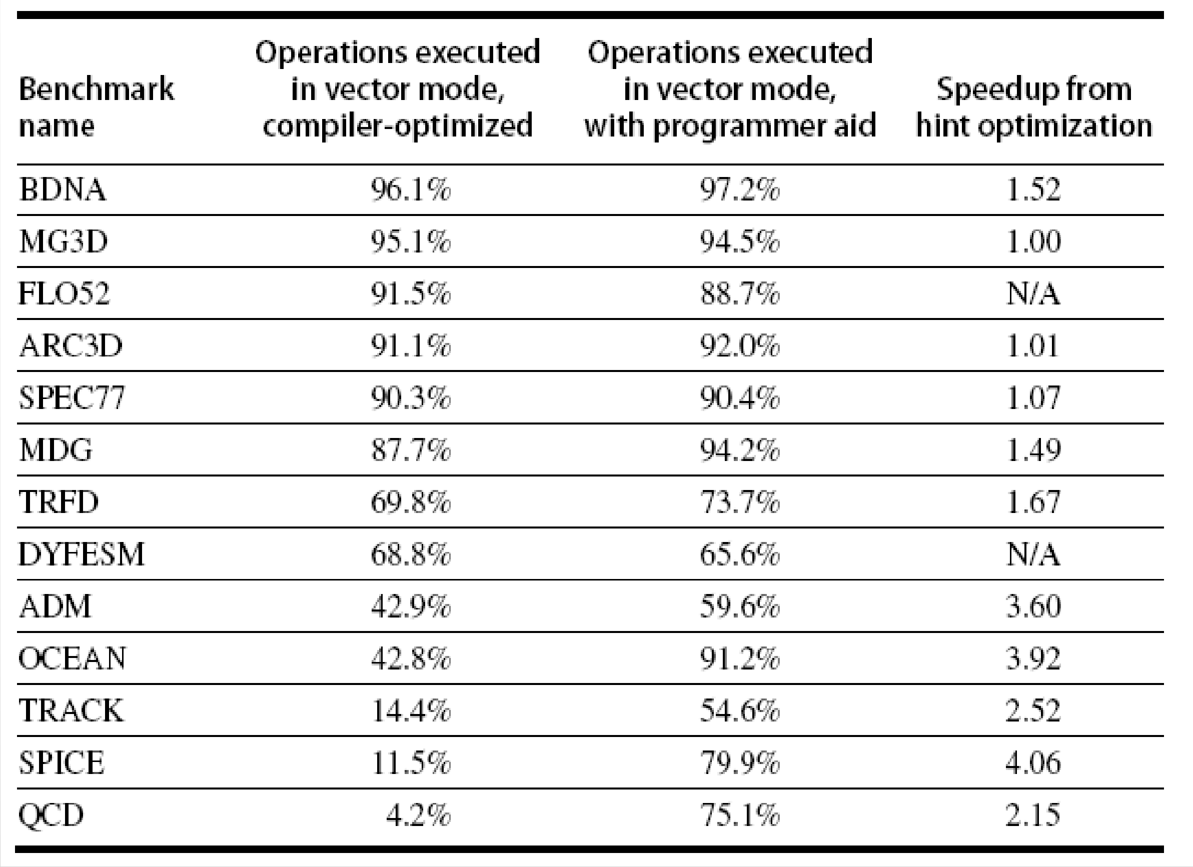
Scatter-Gather Handling Sparse Matrices

- This technique allows code with sparse matrices to run in vector mode
- Ra, Rc, Rk and Rm the starting addresses of vectors

```
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 Vector 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

Level of vectorization among the Perfect Club benchmarks executed on the Cray Y-MP [Vajapeyam 1991].

The first column shows the vectorization level obtained with the compiler without hints

The second column shows the results after the codes have been improved with hints from a team of Cray Research programmers.