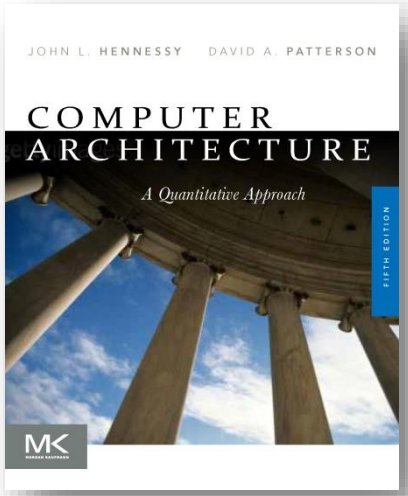


Chapter 4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures

Soontae Kim
Spring 2017
School of Computing, KAIST

Announcements

- Homework assignment #3
  - Posted on class web site
  - Due on May 26 (today)
- Term project
  - Final presentations on June 7 and 9
  - Final report due on June 16

Introduction

- 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
- SIMD extensions
- Graphics Processor Units (GPUs)

We call these algorithms *data parallel* algorithms because their parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control.

W. Daniel Hillis and Guy L. Steele
"Data Parallel Algorithms," *Comm. ACM* (1986)

If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?

Seymour Cray, Father of the Supercomputer
(arguing for two powerful vector processors versus many simple processors)

Vector Architectures

Vector architectures grab sets of data elements scattered about memory, place them into large, sequential register files, operate on data in those register files, and then disperse the results back into memory. A single instruction operates on vectors of data, which results in dozens of register–register operations on independent data elements.

These large register files act as compiler-controlled buffers, both to hide memory latency and to 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 over, say, 64 elements. Indeed, vector programs strive to keep memory busy.

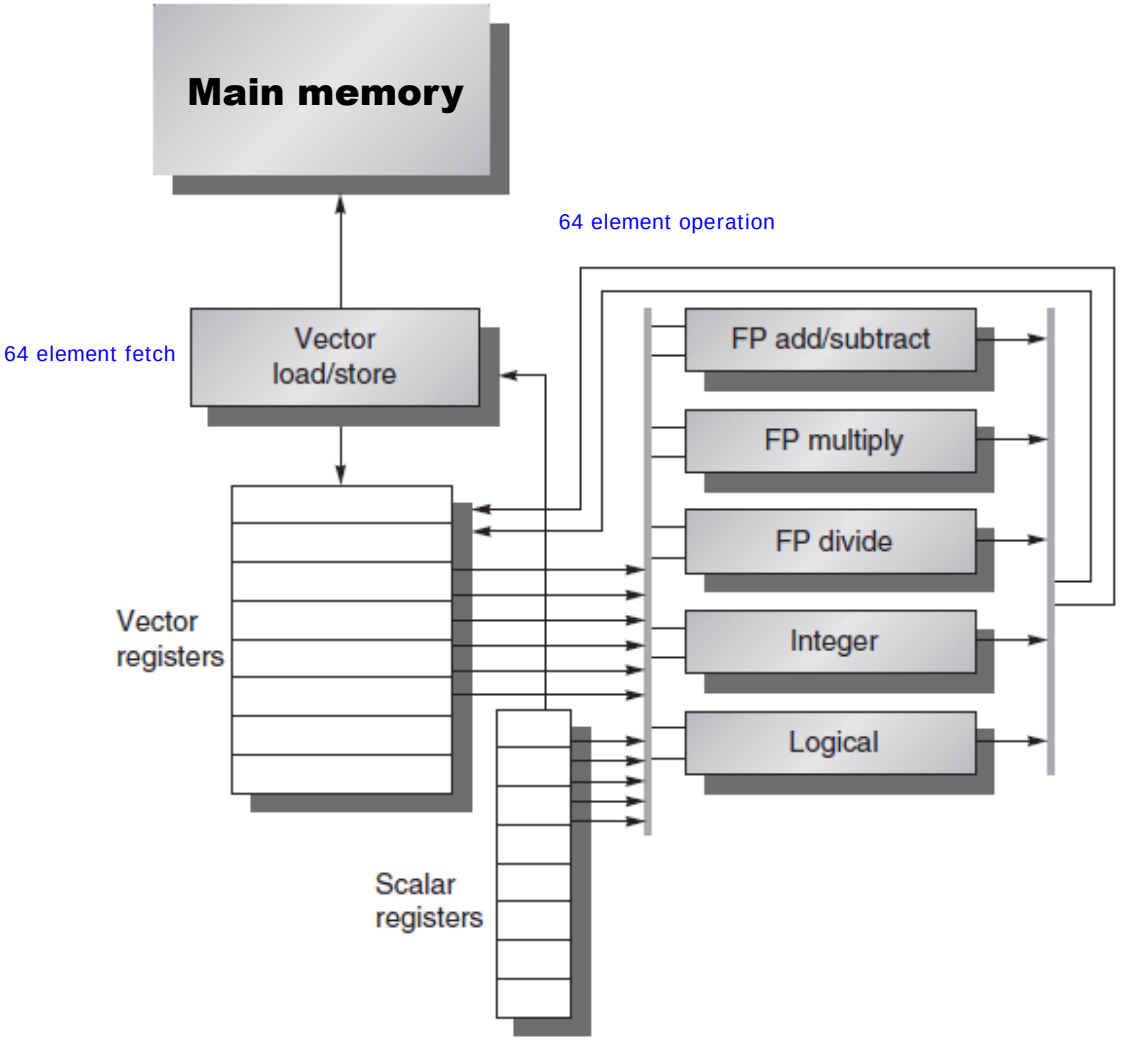


Figure 4.2 The basic structure of a vector architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. This chapter defines special vector

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

ADDVV.D
source source double
vector vector precision

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 MFC1	VLR,R1 R1,VLR	Move contents of R1 to vector-length register VL. Move the contents of vector-length register VL to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to vector-mask register VM. Move contents of vector-mask register VM to F0.

Figure 4.3 The VMIPS vector instructions, showing only the double-precision floating-point operations. In

Example: Y=a * X + Y (DAXPY)

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

Here is the VMIPS code for 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
        SV      V4,Ry           ;store the result
```

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 for simplicity
  - Execution time is approximately the vector length
- *Convoy*
  - Set of vector instructions that could potentially execute together
  - No structural hazards
  - No RAW hazards

Chimes

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

- *Chaining* data forwarding
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available, similar to forwarding in MIPS

- *Chime*
  - Unit of time to execute one convoy
  - *m* convoys execute in *m* chimes
  - For vector length of *n*, requires *m* x *n* clock cycles

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        V4,Ry      ;store the sum
```

Convoys:

1	LV	MULVS.D
2	LV	ADDVV.D
3	SV	

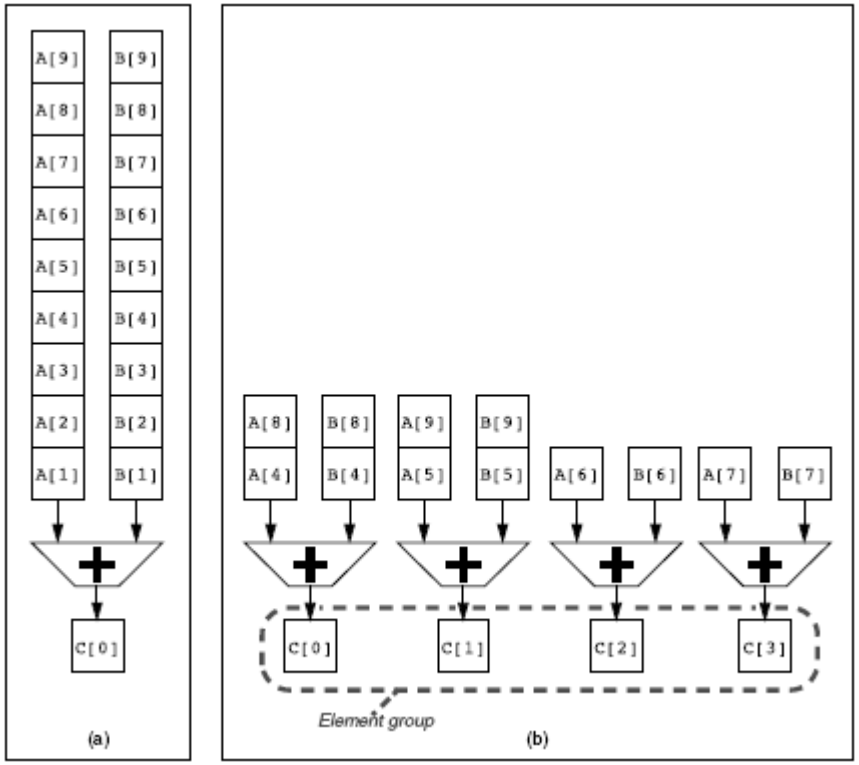
For 64-element vectors, requires 64 x 3 = 192 clock cycles

Challenges

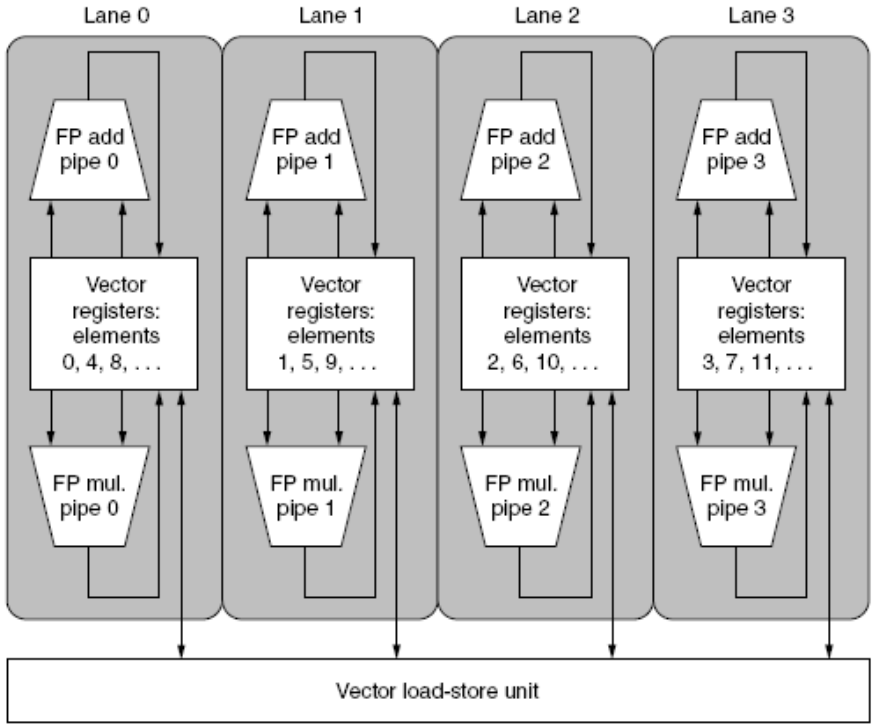
- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Improvements
  - > 1 element per clock cycle more functional unit
  - 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* lane마다 저장될 수 있는 특정 element index가 정해져 있기 때문에
  - Allows for multiple hardware lanes



Using multiple functional units



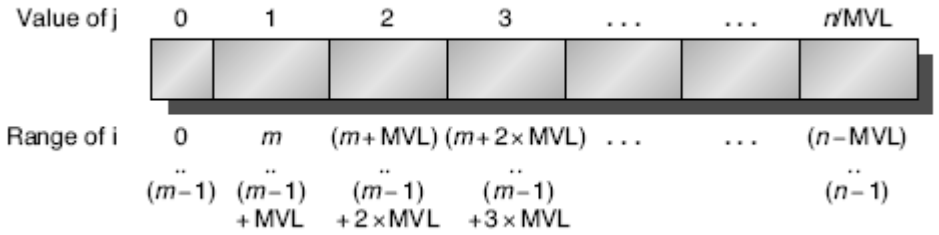
Using multiple lanes

Vector Length Register

- Vector length not known at compile time? unknwon
- Use Vector Length Register (VLR)
- 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*/
}
```

MVL (maximum vector length)



Vector Mask Registers

- Consider:

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

- Use vector mask register to “disable” elements:

```
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          V1, Rx      ;store the result in X
```

- GFLOPS rate decreases!

FLOPS : floating point operations per second

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
  - Use memory banks to allow independent accesses rather than simple interleaving
- Spread accesses across multiple independent banks
  - Control addresses to banks 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?
    - 32*6= 192 , 15/2.167= 6.92
    - 192*7 = 1344

each bank can support 1 memory access
-> how many bank we need?
32 processor * 6 memory access = 192 access/cycle

sram access time = 7 cycle

7개의 access가 sram에 simultaneous하게 이루어질 수 있기 때문에
모두 처리하기 위해서 7*192개만큼 bank 필요

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