Vector Introduction

Lecture 10

2/21/96

Outline

Motivation

Vector components and example machines

Vector instructions & vector program

Vector Execution

Vector Load/Store Units

Vector Length, Stride, Strip Mining

Vector Optimizations: Chaining, Gather/Scatter, Conditional

Vector Metrics

Vector Pitfalls

Instructon Level Parallelism

High speed execution based on *instruction level parallelism* (ilp): potential of short instruction sequences to execute in parallel

High-speed microprocessors exploit ILP by:

1) pipelined execution: overlap instructions

2) superscalar execution: issue and execute multiple instructions per clock cycle

3) Out-of-order execution: complete out of order

Memory accesses for high-speed microprocessor?

For cache hits

Problems with conventional approach

Limits to conventional exploitation of ILP:

1) *pipelined clock rate*: at some point, each increase in clock rate has corresponding CPI increase

2) *instruction fetch and decode*: at some point, its hard to fetch and decode more instructions per clock cycle

3) *cache hit rate*: some long-running (scientific) programs have very large data sets accessed with poor locality

Vector Processors

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

e.g., A = BxC, where A, B, C are 64-element vectors of 64-bit floating point numbers

Properties of vectors:

Each result independent of previous result => long pipeline, compiler ensures no dependencies

single vector instruction implies lots of work (≈ loop)

vector instructions access memory with known pattern => highly interleaved memory => amortize memory latency of over ≈ 64 elements => no caches required!

reduces branches and branch problems in pipelines

Styles of Vector Architectures

vector-register processors: all vector operations between vector registers (except load and store)

Vector equivalent of load-store architectures

Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC

memory-memory vector processors: all vector operations are memory to memory

Components of Vector Processor

Vector Register: fixed length bank holding a single vector

has at least 2 read and 1 write ports

typically 8-16 vector registers, each holding 64-128 64-bit elements

Vector Functional Units (FUs): fully pipelined, start new operation every clock

typically 4 to 8: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift

Vector Load-Store Units (LSUs): fully pipelined unit to load or store a vector

Scalar registers: single element for FP scalar or address

Cross-bar to connect FUs , LSUs, registers together

Example Vector Machines

Machine	*Year*	*Clock*	*Regs*	*Elements*	*FUs*	*LSUs*
Cray 1	1976	80 MHz	8	64	6	1
Cray YMP	1988	166 MHz	8	64	8	3
Cray C-90	1991	240 MHz	8	128	8	4
Cray T-90	1995	450 MHz	8	128	8	4
Conv. C-1	1984	10 MHz	8	128	4	1
Conv. C-4	1994	133 MHz	16	128	3	1
Fuj. VP200	1982	133 MHz	8-256	32-1024	3	2
Fuj. VP500	1994	100 MHz	8-256	32-1024	16?	2?
NEC SX/2	1984	160 MHz	8+8K	256+var	16	8
NEC SX/2	1995	400 MHz	8+8K	256+var	16	8

Vector Linpack Performance

Machine	*Year*	*Clock*	*100x100*	*1kx1k*	*any(Procs)*
Cray 1	1976	80 MHz	27 MF	110 MF	160 (1)
Cray YMP	1988	166 MHz	161 MF	324 MF	2,667 (8)
Cray C-90	1991	240 MHz	387 MF	902 MF	15,238 (16)
Cray T-90	1995	450 MHz	522 MF	1577 MF	57,600 (32)
Conv. C-1	1984	10 MHz	3 MF	--	20 (1)
Conv. C-4	1994	133 MHz	99 MF	705 MF	3,240 (4)
Fuj. VP200	1982	133 MHz	18 MF	422 MF	533 (1)
Fuj. VP500	1994	100 MHz	206 MF	1490 MF	1600 (1)
NEC SX-1	1984	160 MHz	27 MF	110 MF	160 (1)
NEC SX-3	1995	400 MHz	368 MF	2757 MF	25,600 (4)

Vector Instructions

Instr.	Operands	Operation	Comment
`ADDV`	V1,V2,V3	V1=V2+V3	vector + vector
`ADDS`	V1,F0,V2	V1=F0+V2	scalar + vector
`MULV`	V1,V2,V3	V1=V2xV3	vector x vector
`MULS`	V1,F0,V2	V1=F0xV2	scalar x vector
`LV`	V1,R1	V1=M[R1..R1+63]	load, stride=1
`LVWS`	V1,R1,R2	V1=M[R1..R1+63*R2]	load, stride=R2
`LVI`	V1,R1,V2	V1=M[R1+V2i,i=0..63]	indirect("gather")
`CeqV`	VM,V1,V2	VMASKi = (V1i=V2i)?	compare, setmask
`MOV`	VLR,R1	Vec. Len. Reg. = R1	set vector length
`MOV`	VM,R1	Vec. Mask = R1	set vector mask

DAXPY (Y = a x X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector

```
      LD     F0,a          ;load scalar a
      LV     V1,Rx         ;load vector X
      MULS   V2,F0,V1      ;vector-scalar mult.
      LV     V3,Ry         ;load vector Y
      ADDV   V4,V2,V3      ;add
      SV     Ry,V4         ;store the result
```

```
      LD     F0,a
      ADDI   R4,Rx,#512    ;last address to load
loop: LD     F2,0(Rx)      ;load X(i)
      MULTD  F2,F0,F2      ;a*X(i)
      LD     F4,0(Ry)      ;load Y(i)
      ADDD   F4,F2,F4      ;a*X(i) + Y(i)
      SD     F4,0(Ry)      ;store into Y(i)
      ADDI   Rx,Rx,#8      ;increment index to X
      ADDI   Ry,Ry,#8      ;increment index to Y
      SUB    R20,R4,Rx     ;compute bound
      BNZ    R20,loop      ;check if done
```

578 (2+9*64) or vs. 6 instructions:

64 operation vectors+

no loop overhead

also fewer pipeline hazards

Vector Execution Time

Time = f(vector length, data dependicies, hazards)

Initiation rate: rate that FU consumes vector elements (usually 1, 2 on T-90)

Convey: set of vector instructions that can begin execution in same clock (no hazards)

Chime: approx. time for a vector operation

m convoys take *m* chimes; if each vector length is n, then they take approx. *m* x *n* clock cycles (ignores overhead)

```
1: LV    V1,Rx       ;load vector X
2: MULS  V2,F0,V1    ;vector-scalar mult.
   LV    V3,Ry       ;load vector Y
3: ADDV  V4,V2,V3    ;add
4: SV    Ry,V4       ;store the result
```

4 conveys
=> 4 x 64 ≈ 256 clocks

Start-up Time

Start-up time: pipeline latency time (depth of FU pipeline)

Operation	*Start-up penalty*
Vector load/store	12
Vector multply	7
Vector add	6

Assumes convoys don't overlap; vector length = n

Convoy	*Start*	*1st result*	*last result*
1. LV	0	12	11+n
2. MULV, LV	12+n	12+n+12	24+2n
3. ADDV	25+2n	25+2n+6	31+3n
4. SV	32+3n	32+3n+12	42+4n

Vector Load/Store Units & Memories

Start-up overheads usually longer fo LSUs

Memory system must sustain 1 word/clock cycle

Many Vector Procs. use banks vs. simple interleaving:

1) support multiple loads/stores per cycle => multiple banks & address banks independently

2) support non-sequential accesses

Note: No. memory banks > memory latency to avoid stalls

Vector Length

What to do when the vector length is not exactly 64?

vector-length register **(VLR) controls the length of any vector operation, including a vector load or store. (cannot be > the length of vector registers)**

```
        do 10 i = 1, n
10      Y(i) = a * X(i) + Y(i)
```

Don't know `n` until runtime! > Max. Vector Length (MVL)?

Strip mining**: generation of code such that each vector operation is done for a size ≤ to the MVL**

Strip Mining

Suppose Vector Length > Max. Vector Length (MVL)?

Strip mining**: generation of code such that each vector operation is done for a size ≤ to the MVL**

1st loop do short piece (n mod MVL), rest VL = MVL

```
   low = 1
   VL = (n mod MVL)  /*find the odd size piece*/
   do 1 j = 0,(n / MVL)  /*outer loop*/
        do 10 i = low,low+VL-1  /*runs for length VL*/
               Y(i) = a*X(i) + Y(i)  /*main operation*/
10      continue
        low = low+VL  /*start of next vector*/
        VL = MVL  /*reset the length to max*/
1  continue
```

Vector Stride

Suppose adjacent elements not sequential in memory?

```
    do 10 i = 1,100
          do 10 j = 1,100
                A(i,j) = 0.0
                do 10 k = 1,100
10              A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

Either B or C accesses not adjacent (800 bytes between)

stride: distance separating elements that are to be merged into a single vector (caches do unit stride) => `LVWS`

Strides => can cause bank conflicts (e.g., stride = 32 and 16 banks)

Compiler Vectorization on Cray XMP

Benchmark	*%FP*	*%FP in vector*
ADM	23%	68%
DYFESM	26%	95%
FLO52	41%	100%
MDG	28%27%	
MG3D	31%	86%
OCEAN	28%	58%
QCD	14%	1%
SPICE	16%	7%
TRACK	9%	23%
TRFD	22%	10%

Vector Opt #1: Chaining

Suppose:

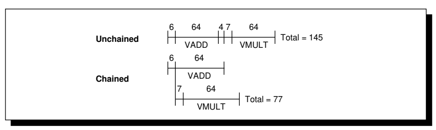
MULV **V1,V2,V3**

ADDV **V4,*V1*,V5** **; separate convoy?**

***chaining*: vector register (V1) is not as a single entity but as a group of individual registers, then pipeline forwarding can work on individual elements of a vector**

***Flexible chaining*: allow vector to chain to any other active vector operation => more read/write port**

As long as enough HW, increases convoy size



Vector Opt #2: Sparse Matrices

Suppose:

```
      do   100 i = 1,n
100        A(K(i)) = A(K(i)) + C(M(i))
```

***gather* (`LVI`) 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 => a nonsparse 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 (`SVI`), using the same index vector

Can't be done by compiler since can't know Ki elements distinct, no dependencies; by compiler directive

Use `CVI` to create index 0,m, 2m, ..., 63m

Vector Opt #3: Conditional Execution

Suppose:

```
      do 100 i = 1, 64
           if (A(i) .ne. 0) then
               A(i) = A(i) – B(i)
           endif
  100 continue
```

vector-mask control takes a Boolean vector: when *vector-mask register* is loaded from vector test, vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1.

Still requires clock even if result not stored; if still performs operation, what about divide by 0?

Vector Metrics

R∞—The MFLOPS rate on an infinite-length vector. Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger. (Rn is the MFLOPS rate for a vector of length n.)

N1/2—The vector length needed to reach one-half of r∞. This is a good measure of the impact of start-up.

Nv—The vector length needed to make vector mode faster than scalar mode. This measures both start-up and the speed of scalars relative to vectors.

Vector Pitfalls

Pitfall: Concentrating on peak performance and ignoring start-up overhead (Nv > 100!)

Pitfall: Increasing vector performance, without comparable increases in scalar performance (Amdahl's Law)

Pitfall: Good processor vector performance without providing good memory bandwidth.