Vector Processors Part 2

Performance

- Vector Execution Time
- Enhancing Performance
- Compiler Vectorization
- Performance of Vector Processors
- Fallacies and Pitfalls

Vector Execution Time

- *Convoys*

  Set of vector instructions that coud potentially begin execution together in one clock period. Cannot contain structural or data hazards. One convoy must finish execution before another begins (do not overlap in time).

- *Chime*

  Unit of time taken to execute a convoy. Approximate measure, ignores some overheads like issue limitations.

  m convoys of *n* length vectors execute in *m* chimes, approximately *m* x *n* cycles.

- *Startup costs*

  Pipeline latency

- Example: D = aX + Y

Unit	Start-up overhead (cycles)
Load and store unit	12
Multiply	7
Add unit	6

Convoy	Starting Time	First-result time	Last-result time
1. LV	0	12	11 + n
2. MULVS.D LV	12 + n	12 + n + 12	23 + 2n
3. ADDV.D	24 + 2n	24 + 2n + 6	29 + 3n
4. SV	30 + 3n	30 + 3n + 12	41 + 4n

Strip-mining loop

```
      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 */
      continue
```

Total running time of loop

$T_n = ceil[n/MVL]x(T_{loop} + T_{start}) + n \times T_{chime}$

Enhancing Performance

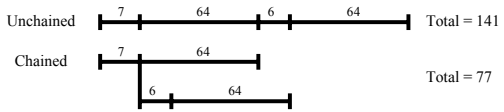
- **Chaining**

MULV.D **V1**,V2,V3
ADDV.D V4,**V1**,V5

Instructions must execute in 2 convoys because of dependancies.

Chaining treats vector registers as a collection of individual registers. Forward individual registers.

MULV.D **V1**,V2,V3
ADDV.D V4,**V1**,V5



Flexible Chaining

Allows a vector instruction to chain to any other active vector instruction

Conditionally Executed Statements

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

Cannot vectorize loop because of conditional execution

Vector-mask control

Boolean vector of length MVL to control the execution of a vector instruction

When a *vector-mask register* is enabled, any vector instructions executed operate only on vector elements whose corresponding entries in the vector-mask register are 1. Entries corresponding to a 0 in the vector-mask are unaffacted

The previous loop can be vectorized using a vector-mask.

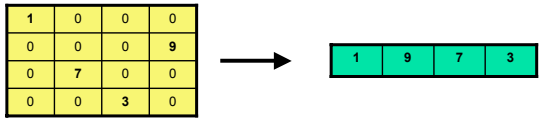
```
LV          V1,Ra          ;load vector A into V1
LV          V2,Rb          ;load vector B
L.D         F0,#0          ;load FP zero into F0
SNEVS.D     V1,F0          ;sets VM(i) to 1 if V1(i) != F0
SUBV.D      V1,V1,V2       ;subtract under vector mask
CVM                        ;set the vector mask to all 1s
SV          Ra,V1          ;store the result in A
```

Vector instructions executed with vector-masks still take execution time for elements where the vector-mask is 0

However, even with many 0s in the mask performance is often better than scalar mode

- **Sparse Matrices**

  Only non-zero elements of matrix are stored.



  How do we access vectors in such a structure?

Scatter-gather operations

Allow retrieval and storage of vectors from sparse data structures

Gather operation

Takes an *index vector* and a *base address*. Fetches vector whose elements are at address given by adding the base address to the offsets in the index vector

LVI Va, (Ra+Vk)

Scatter operation

Stores a vector in sparse form using an index vector and a base address

SVI (Ra+Vk),Va

Most vector processors provide support for computing index vectors. In VMIPS we have an instruction CVI

```
SNEVS.D   V1,F0
CVI       V2,#8
```

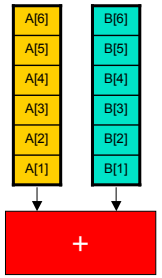
Sparse matrices cannot be automatically vectorized by simple compilers. Compiler cannot tell if elements of index vector are distinct values and that no dependancies exist. Requires programmer directives.

Example: Cache (1993) vs. Vector (1988)

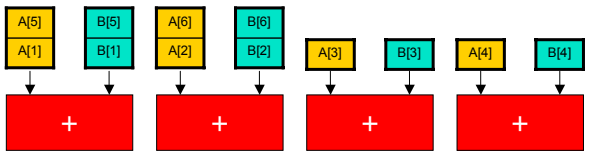
	IBM RS6000	Cray YMP
Clock	72 MHz	167 MHz
Cache	256 KB	0.25 KB
Linpack	140 MFLOPS	160 (1.1)
Sparse Matrix (Cholesky Blocked)	17 MFLOPS	125 (7.3)

- **Multiple Lanes**

Single lane



Can improve performance using multiple lanes



Each lane operates on its elements independantly of the others, so no communication between lanes needed

Adding extra lanes increases peak performance, but does not change start-up latency. As the number of lane increases, start-up costs become more significant.

- **Pipelined Instruction Start-Up**

  Allows the overlapping of vector instructions. Reduces start-up costs.

Effectiveness of Compiler Vectorization

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, hand-optimized	Speedup from hand 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%	92.4%	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 not sufficient by itself to determine performance. Alternate vectorization techniques can lead to better performance. BDNA has similar levels of vectorizations, but hand-optimized code over 50% faster.

Performance of Vector Processors

- R_{inf}: MFLOPS rate on an infinite-length vector
  - upper bound
  - Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger
  - (R_n is the MFLOPS rate for a vector of length n)
- $N_{1/2}$: The vector length needed to reach one-half of R_{inf}
  - a good measure of the impact of start-up
- N_V: The vector length needed to make vector mode faster than scalar mode
  - measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit

- Example: R_{inf}

DAXPY loop: D = aX + Y

$T_n = ceil[n/MVL]x(T_{loop} + T_{start}) + n \times T_{chime}$

Assuming chaining, we can execute loop in 3 chimes

1. LV V1,Rx MULVS.D V2,V1,F0
2. LV V3,Ry ADDV.D V4,V2,V3
3. SV Ry,V4

Assume MVL=64, T_{loop}=15,T_{start}=49,T_{chime}=3 and a 500MHz processor

$T_n = ceil[n/64]x(15 + 49) + n \times 3$

$T_n <= (n + 64) + 3n$

$= 4n + 64$

$$R_{inf} = \lim_{n\to\infty} \frac{\text{(Operations per iteration x Clock rate)}}{\text{Clock cycles per iteration}}$$

$$= \frac{\text{(Operations per iteration x Clock rate)}}{\lim_{n\to\infty} \text{(Clock cycles per iteration)}}$$

$\lim_{n\to\infty}$ (Clock cycles per iteration) $= \lim (T_n/n)$

$= \lim ((4n + 64)/n)$

$= 4$

R_{inf} = 2 x 500MHz / 4 = 250 MFLOPS

- Example: $N_{1/2}$

$$\text{MFLOPS} = \frac{\text{FLOPS executed in } N_{1/2} \text{ iterations}}{\text{Clock cycles to execute } N_{1/2} \text{ iterations}} \times \frac{\text{Clock cycles}}{\text{second}} \times 10^{-6}$$

$$125 = \frac{2 \times N_{1/2}}{T_{N1/2}} \times 500$$

Simplifying this and then assuming $N_{1/2} <= 64$, so that $T_{n<=64} = 1 \times 64 + 3 \times n$, yields

$T_{n<=64} = 8 \times N_{1/2}$

$1 \times 64 + 3 \times N_{1/2} = 8 \times N_{1/2}$

$5 \times N_{1/2} = 64$

$N_{1/2} = 12.8$

$N_{1/2} = 13$

- Example: N_v

Estimated time to do one iteration in scalar mode is 59 clocks.

$64 + 3N_v = 59N_v$

$N_v = ceil[64/56]$

$= 2$

Fallacies and Pitfalls

- Pitfall: Concentrating on peak performance and ignoring start-up overheads. Can lead to large $N_v > 100$!
- Pitfall: Increasing vector performance, without comparable increases in scalar performance (Amdahl's Law)
- Pitfall: You can get vector performance without providing memory bandwidth

- Where to now?