COSC 6385
Computer Architecture
- Vector Processors (I)

Edgar Gabriel

Fall 2006

Vector Processors

- Chapter G of the book
  - Only available online at http://www.mkp.com/ca3
  - Anybody having problems to find it should contact me
- Vector processors big in ‘70 and ‘80
- Still used today
  - Vector machines: Earth Simulator, NEC SX8, Cray X1
  - Graphics cards
  - MMX, SSE, SSE2 are to some extent ‘vector units’

Main concepts

- Vector processors abstract operations on vectors, e.g. replace the following loop

```
for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
}
```

by

```
a = b + c;   ⟶   ADDV.D V10, V8, V6
```

- Some languages offer high-level support for these operations (e.g. Fortran90 or newer)

Main concepts (II)

- Advantages of vector instructions
  - A single instruction specifies a great deal of work
  - Since each loop iteration must not contain data dependence to other loop iterations
    - No need to check for data hazards between loop iterations
    - Only one check required between two vector instructions
    - Loop branches eliminated

Basic vector architecture

- A modern vector processor contains
  - Regular, pipelined scalar units
  - Regular scalar registers
  - Vector units – (inventors of pipelining!)
  - Vector register: can hold a fixed number of entries (e.g. 64)
  - Vector load-store units

Comparison MIPS code vs. vector code

Example: Y=aX+Y for 64 elements

```
   L.D       F0, a              /* load scalar a*/
   DADDIU    R4, Rx, #512       /* last address */
L: L.D       F2, 0(Rx)          /* load X(i) */
   MUL.D     F2, F2, F0         /* calc. a times X(i)*/
   L.D       F4, 0(Ry)          /* load Y(i) */
   ADD.D     F4, F4, F2         /* aX(I) + Y(i) */
   S.D       F4, 0(Ry)          /* store Y(i) */
   DADDIU    Rx, Rx, #8         /* increment X*/
   DADDIU    Ry, Ry, #8         /* increment Y */
   DSUBU     R20, R4, Rx        /* compute bound */
   BNEZ      R20, L
```

Comparison MIPS code vs. vector code (II)

Example: Y=aX+Y for 64 elements

```
L.D       F0, a              /* load scalar a*/
LV        V1, 0(Rx)          /* load vector X */
MULVS.D   V2, V1, F0         /* vector scalar mult*/
LV        V3, 0(Ry)          /* load vector Y */
ADDV.D    V4, V2, V3         /* vector add */
SV        V4, 0(Ry)          /* store vector Y */
```

Vector execution time

- *Convoy*: set of vector instructions that could potentially begin execution in one clock cycle
  - A convoy must not contain structural or data hazards
  - Similar to VLIW
  - Initial assumption: a convoy must complete before another convoy can start execution
- *Chime*: time unit to execute a convoy
  - Independent of the vector length
  - A vector sequence consisting of *m* convoys executes in *m* chimes
  - A vector sequence consisting of *m* convoys and vector length *n* takes approximately *mxn* clock cycles

Example

```
LV        V1, 0(Rx)          /* load vector X */
MULVS.D   V2, V1, F0         /* vector scalar
mult*/
LV        V3, 0(Ry)          /* load vector Y */
ADDV.D    V4, V2, V3         /* vector add */
SV        V4, 0(Ry)          /* store vector Y */
```

- Convoys of the above code sequence:

```
1. LV
2. MULVS.D      LV
3. ADDV.D
4. SV
```

4 convoys ⇒ 4 chimes to execute

Overhead

- Start-up overhead of a pipeline: how many cycles does it take to fill the pipeline before the first result is available?

Unit	Start-up
Load/store	12
Multiply	7
Add	6

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

Pipelining – Metrics (I)

T_c Clocktime, time to finish one segment/sub-operation

m number of stages of the pipeline

n length of the vector

S Startup time in clocks, time after which the first result is available, $S = M$

$N_{\frac{1}{2}}$ length of the loop to achieve half of the maximum speed

Assuming a simple loop such as:

```
for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
}
```

Pipelining – Metrics (II)

op Number of operations per loop iteration

op_{total} total number of operations for the loop, with $op_{total} = op * n$

Speed of the loop is

$$F = \frac{op_{total}}{time} = \frac{op * n}{T_c(m + (n-1))} = \frac{op}{T_c(\frac{m-1}{n} + 1)}$$

For $n \to \infty$ we get

$$F_{\max} = \frac{op}{T_c}$$

Pipelining – Metrics (III)

Because of the Definition of $N_{\frac{1}{2}}$ we now have

$$\frac{op}{T_c(\frac{m-1}{N_{\frac{1}{2}}}+1)} = \frac{1}{2}F_{\max} = \frac{1}{2}\frac{op}{T_c}$$

or $$\frac{m-1}{N_{\frac{1}{2}}}+1 = 2$$

and $$N_{\frac{1}{2}} = m-1$$

➔length of the loop required to achieve half of the theoretical peak performance of a pipeline is equal to the number of segments (stages) of the pipeline

Pipelining – Metrics (IV)

More general: N_α is defined through

$$\frac{op}{T_c(\frac{m-1}{N_\alpha}+1)} = \alpha \frac{op}{T_c}$$

and leads to $N_\alpha = \frac{m-1}{\frac{1}{\alpha}-1}$

E.g. for $\alpha = \frac{3}{4}$ you get $N_{\frac{3}{4}} = 3(m-1) \approx 3m$

➔ the closer you would like to get to the maximum performance of your pipeline, the larger the iteration counter of your loop has to be

Vector length control

- What happens if the length is not matching the length of the vector registers?
- A vector-length register (VLR) contains the number of elements used within a vector register
- *Strip mining*: split a large loop into loops less or equal the maximum vector length (MVL)

Vector length control (II)

```
low =0;
VL  = (n mod MVL);
for (j=0; j < n/MVL; j++ ) {
   for (i=low; i < VL; i++ ) {
        Y(i) = a * X(i) + Y(i);
   }
   low += VL;
   VL   = MVL;
}
```

Overhead of strip mining

- Strip mining introduces
  - a scalar code section (non-vector code)
  - Overhead by having to setup the vector sequence
- Thus, estimated execution time of vector operations is

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

Vector stride

- Memory on vector machines typically organized in multiple banks
  - Allow for independent management of different memory addresses
  - Memory bank time an order of magnitude larger than CPU clock cycle
- Example: assume 8 memory banks and 6 cycles of memory bank time to deliver a data item
  - Overlapping of multiple data requests by the hardware

Cycle	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7	Bank 8
0	X(0)							
1	Busy	X(1)						
2	Busy	Busy	X(2)					
3	Busy	Busy	Busy	X(3)				
4	Busy	Busy	Busy	Busy	X(4)			
5	Busy	Busy	Busy	Busy	Busy	X(5)		
6		Busy	Busy	Busy	Busy	Busy	X(6)	
7			Busy	Busy	Busy	Busy	Busy	X(7)
8	X(8)			Busy	Busy	Busy	Busy	Busy
9	Busy	X(9)			Busy	Busy	Busy	Busy
10	Busy	Busy	X(10)			Busy	Busy	Busy
11	Busy	Busy	Busy	X(11)			Busy	Busy
12	Busy	Busy	Busy	Busy	X(12)			Busy
13	Busy	Busy	Busy	Busy	Busy	X(13)		
14		Busy	Busy	Busy	Busy	Busy	X(14)	

Vector stride (II)

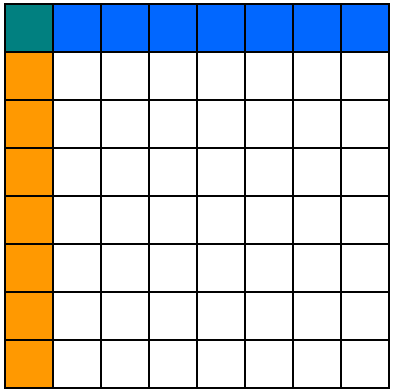
- What happens if the code does not access subsequent elements of the vector

```
for (i=0; i<n; i+=2) {
    a[i] = b[i] + c[i];
}
```

  - Vector load ‘compacts’ the data items in the vector register (gather)
    - No affect on the execution of the loop
    - You might however use only a subset of the memory banks -> longer load time
    - Worst case: stride is a multiple of the number of memory banks

Memory layout of multi-dimensional arrays

- Logical view of the matrix



- Layout in memory of the same matrix (in C)

- Layout in memory of the same matrix (in FORTRAN)