HY425 Lecture 10: Vector processors

Dimitrios S. Nikolopoulos

University of Crete and FORTH-ICS

November 5, 2010

Limitations of ILP

ILP walls

- Hard to exploit higher degrees of ILP
- Deeper pipelines, wider instruction issue
  - Increased hardware complexity with small performance gain
  - Heavy burden on software on statically scheduled processors

Vector processors

Vector instructions

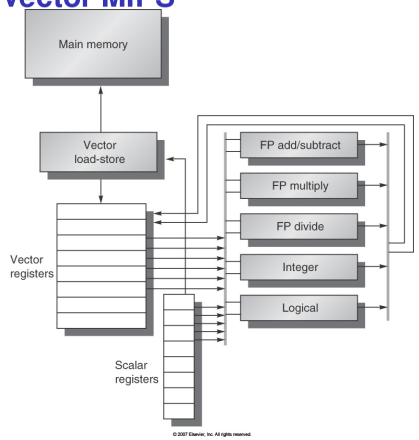
- a loop in an instruction
- explicit parallelism, programmer guarantees independence
- hazard check between blocks of operations
- exploit memory parallelism
- less loop overhead, control hazards

Benefits

- Scientific, engineering, multimedia applications
  - Computation on arrays dominates, data parallelism

Vector architecture

Vector MIPS



Features compared to scalar

- Vector registers (wide)
- Vector FUs, load/store units
- Parallel pipelines or lanes
- Vector ISA

Example: SAXPY, DAXPY

MIPS code

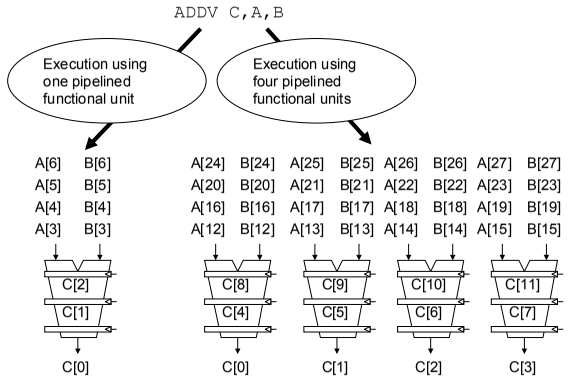
```
       LD F0,a         ;load scalar a
       ADDI R4,Rx,#512 ;last address to load
Loop:  LD F2,0(Rx)     ;load X(i)
       MULD F2,F2,F0   ;a ? X(i)
       LD F4,0(Ry)     ;load Y(i)
       ADDD F4,F4,F2   ;a ? X(i) + Y(i)
       SD 0(Ry),F4     ;store into Y(i)
       ADDIU Rx,Rx,#8  ;increment index to X
       ADDIU Ry,Ry,#8  ;increment index to Y
       SUBU R20,R4,Rx  ;compute bound
       BNEZ R20,Loop ;check if done
```

VMIPS code

```
       LD F0,a         ;load scalar a
       LV V1,Rx        ;load vector X
       MULVSD V2,V1,F0 ;vector-scalar multiply
       LV V3,Ry        ;load vector Y
       ADDVD V4,V2,V3  ;add
       SV Ry,V4        ;store the result
```

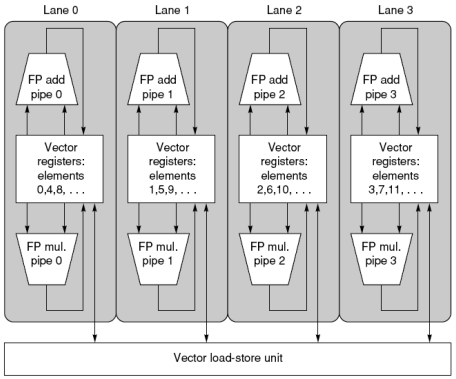

Vector instruction execution

One versus multiple pipelined units



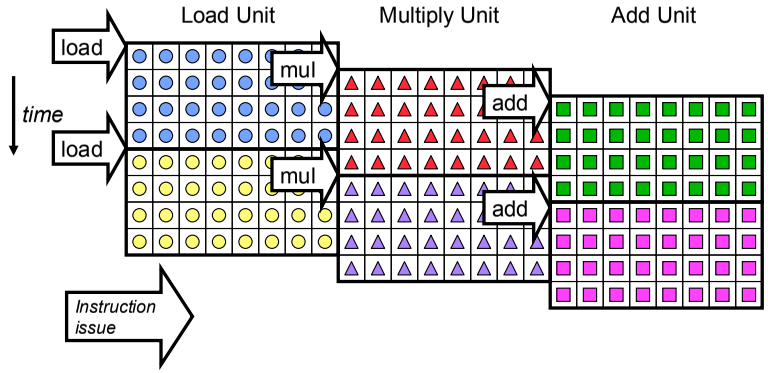
Vector instruction-level parallel execution

Vector processor with four lanes



Multi-lane vector unit

Example



Vector processor performance factors

- Length of vector registers
- Vector operation initiation rate (>= 1 per cycle)
- Number of lanes (parallel pipelines)
- Vector instructions executing in same cycle – convoy
- Time to execute a single convoy – chime

```
1: LV V1,Rx        ;load vector X
2: MULVSD V2,V1,F0 ;vector-scalar multiply
3: LV V3,Ry        ;load vector Y
4: ADDVD V4,V2,V3  ;add
5: SV Ry,V4        ;store the result
```


Vector processor performance measurement

Vector instruction latency

- Pipelined vector FUs
- Convoys do not overlap due to dependences
- Vector startup time:
  - time until first result out from pipelined FU
- Rest of results come out one per cycle
- $S + VL - 1$, S: startup time, VL: vector length

Example: DAXPY

4 convoys

Functional unit	Latency
load/store unit	12
multiply unit	7
add unit	6

Convoy	Start time	First result	Last result
1: LD	0	12	11+n
2: MULVSD, LV	12+n	24+n	23+2n
3: ADDVD	24+2n	30+2n	29+3n
4: SV	30+3n	42+3n	41+4n

- Latency of convoy depends on slowest instruction in convoy
- Latency of second convoy is latency of slower LV instruction
- Shorter vector length implies more pipeline restarts in FUs

Vector load/store unit

Requirements from memory system

- Load/store units need word/cycle bandwidth from memory
- Hard to meet demand, even with advanced memory systems
  - Multiple memory banks, interleaving
  - High memory bank cycle time
  - Multiple loads/stores per clock need to be supported
- How many banks are needed to sustain throughput to load/store units?

Example

6-cycle bank latency, starting addrress=136

- double-word (8-byte) bank interleaving

	Bank							
cycle	0	1	2	3	4	5	6	7
0		136						
1		busy	144					
2		busy	busy	152				
3		busy	busy	busy	160			
4		busy	busy	busy	busy	168		
5		busy	busy	busy	busy	busy	176	
6			busy	busy	busy	busy	busy	184
7	192			busy	busy	busy	busy	busy
8	busy	200			busy	busy	busy	busy
9	busy	busy	208			busy	busy	busy
10	busy	busy	busy	216			busy	busy
11	busy	busy	busy	busy	224			busy
12	busy	busy	busy	busy	busy	232		
13		busy	busy	busy	busy	busy	240	
14			busy	busy	busy	busy	busy	248
15	256			busy	busy	busy	busy	busy
16	busy	264			busy	busy	busy	busy

Vector processor performance metrics

- FLOPS: Floating point operations per second
- R_n: FLOPS with $MVL = n$
- T_n: Time with $MVL = n$
- R_∞: FLOPS with $MVL = \infty$
- $N_{1/2}$: Vector length to achieve $\frac{R_\infty}{2}$ FLOPS
- N_v: Vector length to achieve more FLOPS than scalar

Stripmining

Vector length

- Typically shorter than real vector sizes (2–64 elements)
- Loop transformation to vectorize code
- Partition loop into vector instructions

Example

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

/* Vectorized code */
low = 1;
VL = (n mod MVL); /*find the odd-size piece*/
for j = 0; j < n/MVL; j++) {
  for (i=low; i <= low+VL-1; i++)
     Y(i) = a * X(i) + Y(i); /*main operation*/
   low = low + VL; /*start of next vector*/
   VL = MVL; /*reset the length to max*/
}
```

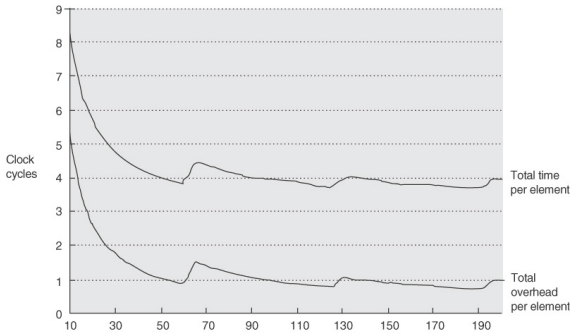

Stripmining performance

- T_{start}: vector pipeline startup cost for loop body
- T_{loop}: scalar code per outer loop iteration
- *chimes*: number of chimes needed to execute convoys in the loop
- Pipelined vector units, no overlap of convoys

$$T_n = \lceil \frac{n}{MVL} \rceil \times (T_{loop} + T_{start}) + n \times chimes \quad (1)$$

Stripmining performance (cont.)

- Overhead more than 50% of total time for short vectors
- Jumps indicate iterations of outer loop after stripmining



Vector strides

Matrix-matrix multiplication

```
for (i=0; i<100; i++)
  for (j=0; j<100; j++) {
    A[i][j] = 0.0;
    for (k=0; k<100; k++)
      A[i][j] = A[i][j] + B[i][k]*C[k][j]
  }
```

Vectorization considerations

- Innermost loop over row of B and column of C
- Row-major array allocation in C
- Vector with non-adjacent elements for k-th column of C
- Vector registers pack data with non-unit strides in memory
  - Compare to caches where blocks store only unit-stride data

Vector strides

Instructions for strided data access

- LVWS: load vector with stride
- SVWS: store vector with stride
- Also known as gather/scatter operations

Vector strides (cont.)

Implications for memory system

- Non-unit strides may increase conflicts in memory banks
- Example: 100×100 matrix-matrix multiplication, double-word (8-byte) interleaving, array type double, starting address of C = 136

	Bank							
cycle	0	1	2	3	4	5	6	7
0		136						
1		busy				936		
2		busy				busy		
3		busy				busy		
4		busy				busy		
5		busy				busy		
6		1736				busy		
7		busy				2536		
8		busy				busy		
9		busy				busy		
10		busy				busy		
11		busy				busy		
12		3336				busy		
13		busy				4136		
14		busy				busy		
15		busy				busy		
16		busy				busy		

Vector strides (cont.)

Implications for memory system

- Example: 128 × 128 matrix-matrix multiplication, double-word (8-byte) interleaving, array type double, starting address of C = 136

	Bank							
cycle	0	1	2	3	4	5	6	7
0		136						
1		busy						
2		busy						
3		busy						
4		busy						
5		busy						
6		1160						
7		busy						
8		busy						
9		busy						
10		busy						
11		busy						
12		2184						
13		busy						
14		busy						
15		busy						
16		busy						

Vector strides (cont.)

Evaluating the impact of conflicts

- Conflicts occur when stride is multiple of $nbanks \times element\_size$
- Stalls occur if requests to same bank come more frequently than bank busy time

$$\frac{nbanks}{LCM(stride, nbanks)} < bank\_busy\_time \quad (2)$$

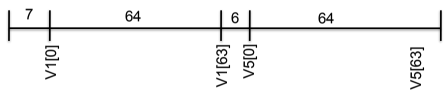
- Conflicts avoided when strides and number of banks are prime relative to each other
- Increasing number of banks decreases frequency of conflicts

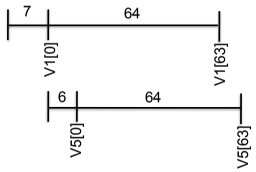
Chaining

Forwarding results between vector instructions

- Vector instruction starts as soon as first element becomes available
- Simultaneous read/write of vector register for different elements
- Vector register also read by multiple dependent instructions simultaneously

```
MULV V1,V2,V3
ADDV V5,V1,V4
```





Conditional execution

Conditional vector instructions

- Convert control dependence to data dependence
- Vector mask register marks vector elements to operate
  - Set vector mask bits for elements for which conditional is true

Example

```
for (i=0; i<64; i++)
   if (A[i] != 0)
      A[i] = A[i] - B[i]
   }
```

VMIPS conditional instructions

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

Scatter-gather operations

Indirection arrays

- Sparse matrix codes access arrays indirectly (e.g. A[B[i]])
- Indirection arrays implemented with index vector registers
- Gather: Collect elements pointed to by index vector
- Scatter: Distribute elements to locations pointed to by index vector

Example

```
for (i=0; i<64; i++)
  A[K[i]]=A[K[i]]-C[M[i]]
```

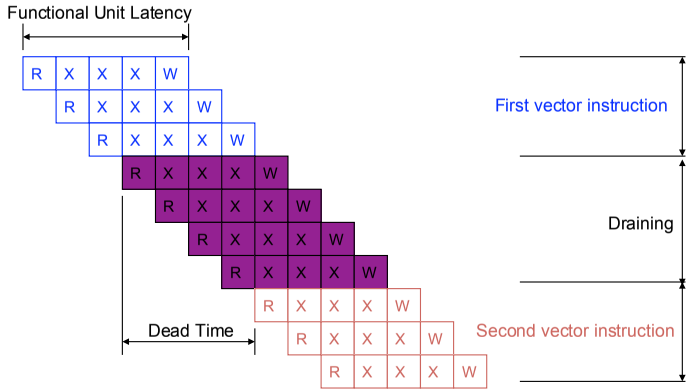
VMIPS gather (LVI), scatter (SVI) instructions

Assume Ra, Rk, Rc Rm have starting addresses of A, K, C, M

```
LV Vk,Rk        ;load K
LVI Va,(Ra+Vk)  ;load A(K(I))
LV Vm,Rm        ;load M
LVI Vc,(Rc+Vm)  ;load C(M(I))
ADDV.D Va,Va,Vc ;add them
SVI (Ra+Vk),Va  ;store A(K(I))
```


Pipeline startup and draining latency

Two independent instructions using same FU



Vector processor performance metrics

- FLOPS: Floating point operations per second
- R_n: FLOPS with $MVL = n$
- T_n: Time with $MVL = n$
- R_∞: FLOPS with $MVL = \infty$
- $N_{1/2}$: Vector length to achieve $\frac{R_\infty}{2}$ FLOPS
- N_v: Vector length to achieve more FLOPS than scalar

$$T_n = \lceil \frac{n}{MVL} \rceil \times (T_{loop} + T_{start}) + n \times T_{chime} \quad (3)$$

DAXPY example

Execution with 500 MHz dual-issue vector processor

LV V1,Rx	MULVS V2,V1,F0	Convoy 1: Chained load and multiply
LV V3,Ry	ADDVD V4,V2,V3	Convoy 2: Chained load and add
SV V4,Ry		Convoy 3: Store result

Assumptions

- MVL=64, LV/SV FU latency=12, ADDVD FU latency=6, MULVS FU latency=7, $T_{loop} = 15$

$$T_{start} = 12 + 7 + 12 + 6 + 12 = 49 \quad (4)$$

$$T_{loop} = 15 \quad (5)$$

$$T_{chime} = 3 \quad (6)$$

$$T_n = \lceil \frac{n}{64} \rceil (49 + 15) + 3n \quad (7)$$

$$T_n \leq 4n + 64 \quad (8)$$

DAXPY example (cont.)

Execution with 500 MHz, dual-issue vector processor

LV V1,Rx	MULVS V2,V1,F0	Convoy 1: Chained load and multiply
LV V3,Ry	ADDVD V4,V2,V3	Convoy 2: Chained load and add
SV V4,Ry		Convoy 3: Store result

Peak and realistic performance

$$R_\infty = \lim_{n\to\infty}\left(\frac{\text{operations per iteration} \times \text{clock rate}}{\text{clock cycles per iteration}}\right) \quad (9)$$

$$\lim_{n\to\infty}(\text{clock cycles per iteration}) = \lim_{n\to\infty}\left(\frac{4n+64}{n}\right) = 4 \quad (10)$$

$$R_\infty = \left(\frac{2\times 500\text{MHz}}{4}\right) = 250\text{MFLOPS} \quad (11)$$

$$R_{66} = \frac{2\times 500\text{MHz}}{\frac{T_{66}}{66}} = \frac{2\times 66\times 500}{2\times(49+15)+3\times 66}\text{MFLOPS} = 202\text{MFLOPS} \quad (12)$$

DAXPY example (cont.)

Execution with 500 MHz, dual-issue vector processor

LV V1,Rx	MULVS V2,V1,F0	Convoy 1: Chained load and multiply
LV V3,Ry	ADDVD V4,V2,V3	Convoy 2: Chained load and add
SV V4,Ry		Convoy 3: Store result

Peak and realistic performance

Assume three instead of one pipeline for memory operations
Instructions fit in one convoy (with chaining)

$$T_{66} = \lceil\frac{66}{64}\rceil(15+49)+66\times 1 = 194 \quad (13)$$

(14)

Strip-mining overhead overlapped

$$T_{66} = (15+49)+66\times 1 = 130, R_\infty = 2\times\text{clock rate} \quad (15)$$

(16)

SIMD extensions to superscalar processors

Motivation

- Add SIMD vector instruction execution capabilities with minimal extensions to a pipelined processor
  - Wide registers, typically 128-bit
  - Additional short vector execution units, typically 128-bit
- Extend ISA with vector instructions
  - MMX, SSE, SSE2, SSE3, SSE4, AltiVec, ...

Adding SIMD extensions to pipelined superscalar processor

Pentium 4 microarchitecture

Borrowed from Intel Technology Journal

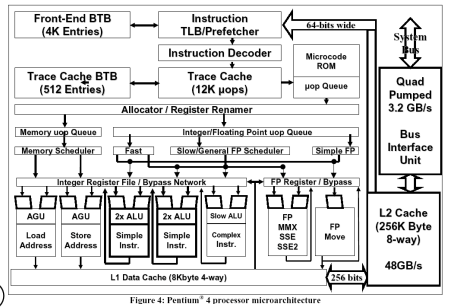


Figure 4: Pentium® 4 processor microarchitecture

Intel SSE extensions

Pentium II onwards

- 8–16 128-bit registers (XMM registers), plus 8–16 64-bit MMX registers
- Registers enable packed data (2 double-words, 4 words, 8 half-words, 16 chars) operations and/or scalar operations
- Single-cycle SSE ALU operations
- SIMD memory-register load/store operations
  - prefetching
  - streaming stores

Intel SSE extensions

Example: inner product

```
typedef float v4sf __attribute__ ((mode(V4SF))); // floating point vector type
float x[k]; float y[k]; // operand vectors of length k
float inner_product = 0.0, temp[4];
v4sf acc, X, Y; // 4x32-bit float registers
acc = __builtin_ia32_xorps(acc, acc); // zero the accumulator
for (int i = 0; i < (k - 3); i += 4) {
  X = __builtin_ia32_loadups(&x[i]); // load groups of four floats
  Y = __builtin_ia32_loadups(&y[i]);
  acc = __builtin_ia32_addps(acc, __builtin_ia32_mulps(X, Y));
}
__builtin_ia32_storeups(temp, acc); // add the accumulated values
inner_product = temp[0] + temp[1] + temp[2] + temp[3];
for (; i < k; i++) // add up the remaining floats
  inner_product += x[i] * y[i];
```

xorps: bit-wise xor of single-precision floating point values
loadups: move four unaligned packed single-precision floats from memory
mulps,addps: multiply, add single-precision floats