Cray-1 and Graphics Processors

Last time — TM

modern implementations hide all side effects

speculate that there will be no conflicts

generalizing speculation

speculation — guess and check:

- branch prediction
- early loads
- ...

transaction mechanism is general way to support it

more opportunities:

- speculate that cached file is up-to-date
- check after getting reply from file server

Common questions

swizzling???

where does the Cray-1 speedup come from?

- startup times?
- versus loop unrolling?

what workloads?

swizzling

rearranging vectors:

/ into [Z, W, Y, X]

/ into [Z, Z, Z, W]

etc.

GPU : rearranging vectors

every instruction allows reordering vectors (“swizzling”):

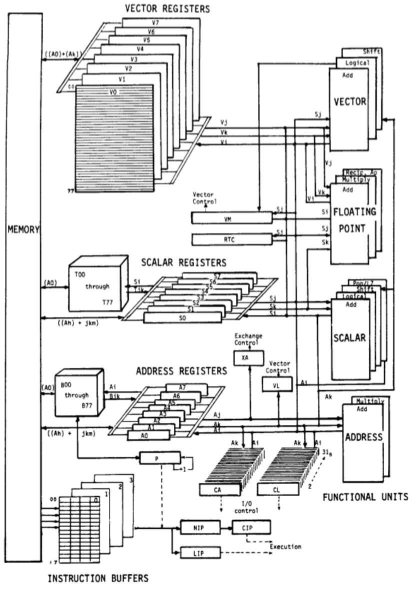
`R0.xyzw`, `R0.yyyy`, `R0.wzyx`, …

every instruction allows write masks:

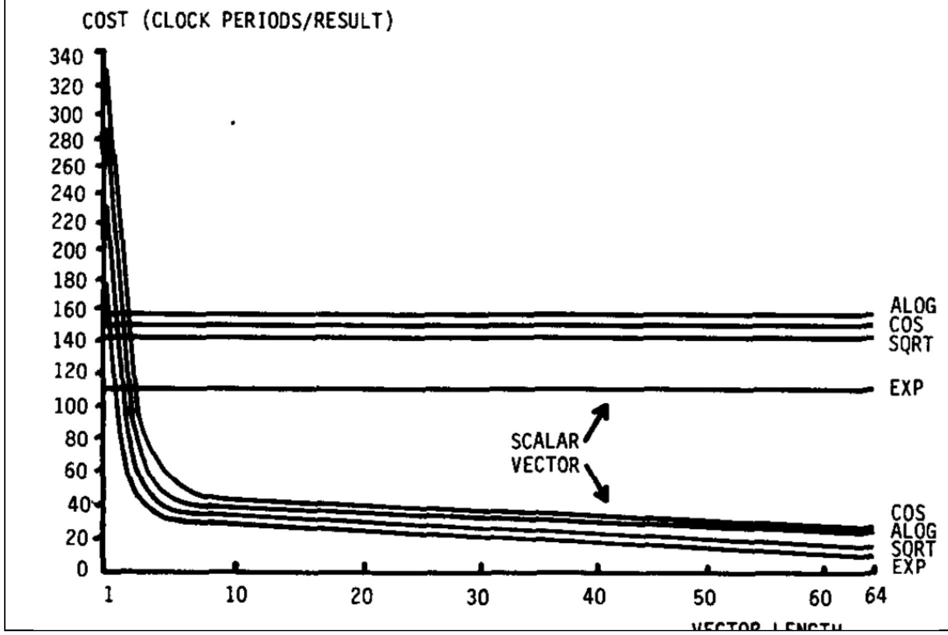
`MUL R0.x, R1, R2` — throw away R1.y * R2.y, etc.

scalar operations — produce vector with multiple copies of output

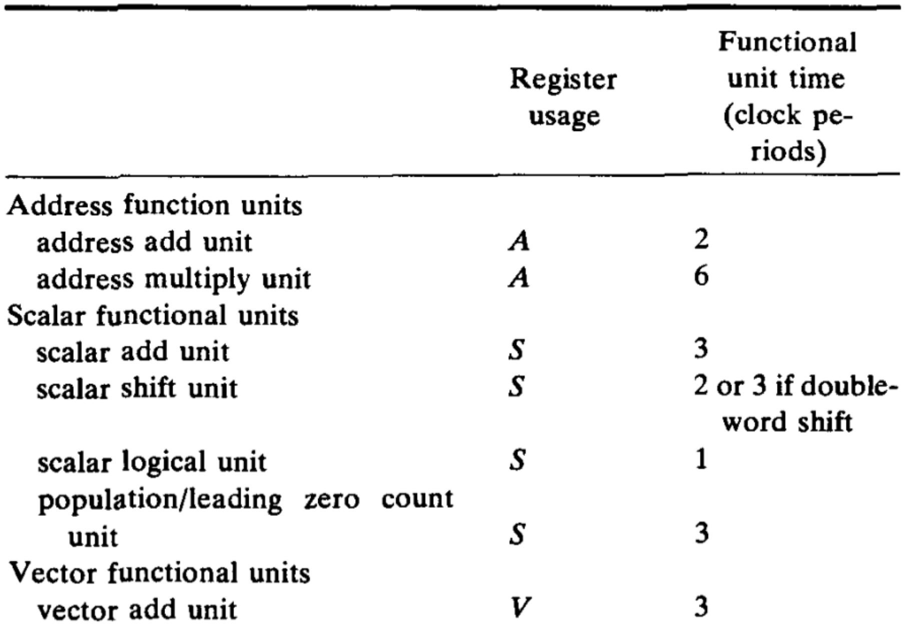
Cray Block Diagram



Cray Vector Performance

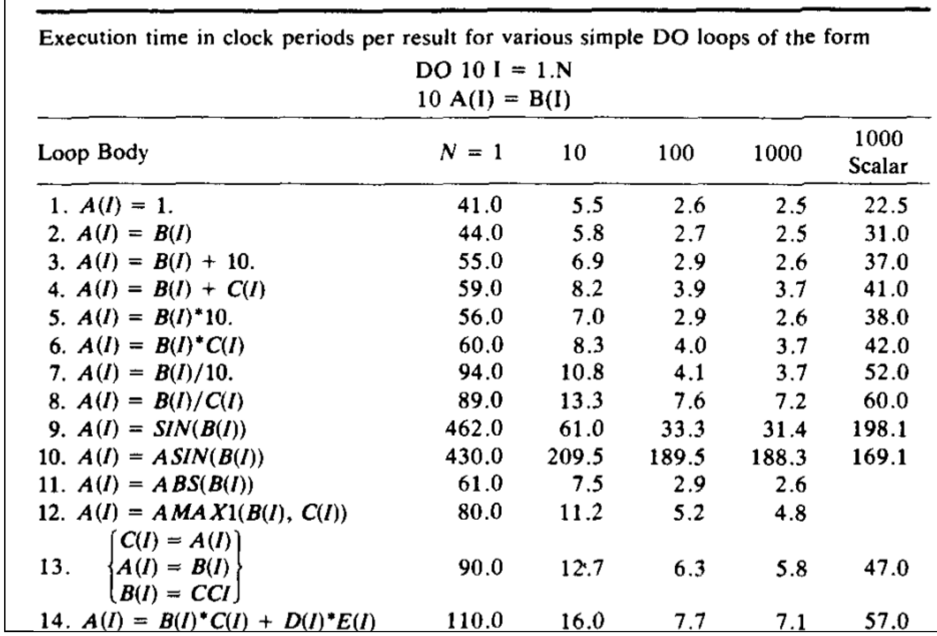


Cray Timing — functional unit



	Register usage	Functional unit time (clock periods)
Address function units		
address add unit	*A*	2
address multiply unit	*A*	6
Scalar functional units		
scalar add unit	*S*	3
scalar shift unit	*S*	2 or 3 if double-word shift
scalar logical unit	*S*	1
population/leading zero count unit	*S*	3
Vector functional units		
vector add unit	*V*	3

Cray Timing — actual



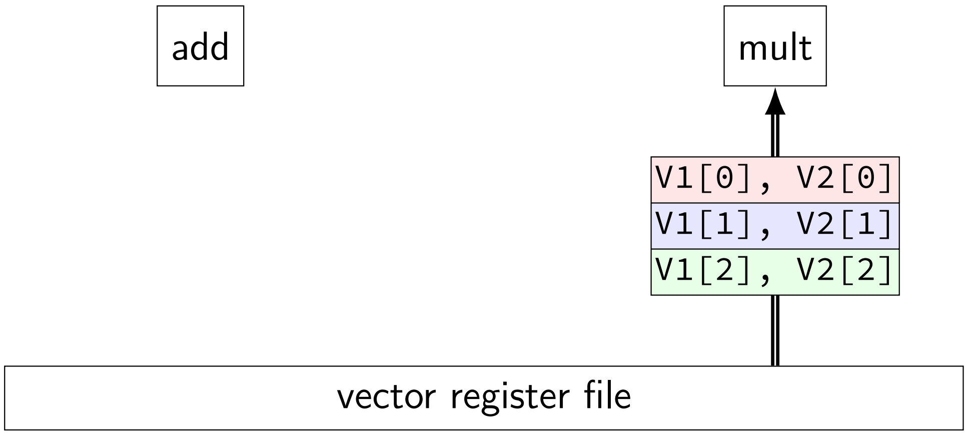
Execution time in clock periods per result for various simple DO loops of the form

DO 10 I = 1.N
10 A(I) = B(I)

Loop Body	*N* = 1	10	100	1000	1000 Scalar
1. *A*(*I*) = 1.	41.0	5.5	2.6	2.5	22.5
2. *A*(*I*) = *B*(*I*)	44.0	5.8	2.7	2.5	31.0
3. *A*(*I*) = *B*(*I*) + 10.	55.0	6.9	2.9	2.6	37.0
4. *A*(*I*) = *B*(*I*) + *C*(*I*)	59.0	8.2	3.9	3.7	41.0
5. *A*(*I*) = *B*(*I*)*10.	56.0	7.0	2.9	2.6	38.0
6. *A*(*I*) = *B*(*I*)**C*(*I*)	60.0	8.3	4.0	3.7	42.0
7. *A*(*I*) = *B*(*I*)/10.	94.0	10.8	4.1	3.7	52.0
8. *A*(*I*) = *B*(*I*)/*C*(*I*)	89.0	13.3	7.6	7.2	60.0
9. *A*(*I*) = *SIN*(*B*(*I*))	462.0	61.0	33.3	31.4	198.1
10. *A*(*I*) = *ASIN*(*B*(*I*))	430.0	209.5	189.5	188.3	169.1
11. *A*(*I*) = *ABS*(*B*(*I*))	61.0	7.5	2.9	2.6	
12. *A*(*I*) = *AMAX*1(*B*(*I*), *C*(*I*))	80.0	11.2	5.2	4.8	
13. {*C*(*I*) = *A*(*I*) *A*(*I*) = *B*(*I*) *B*(*I*) = *CCI*}	90.0	12.7	6.3	5.8	47.0
14. *A*(*I*) = *B*(*I*)**C*(*I*) + *D*(*I*)**E*(*I*)	110.0	16.0	7.7	7.1	57.0

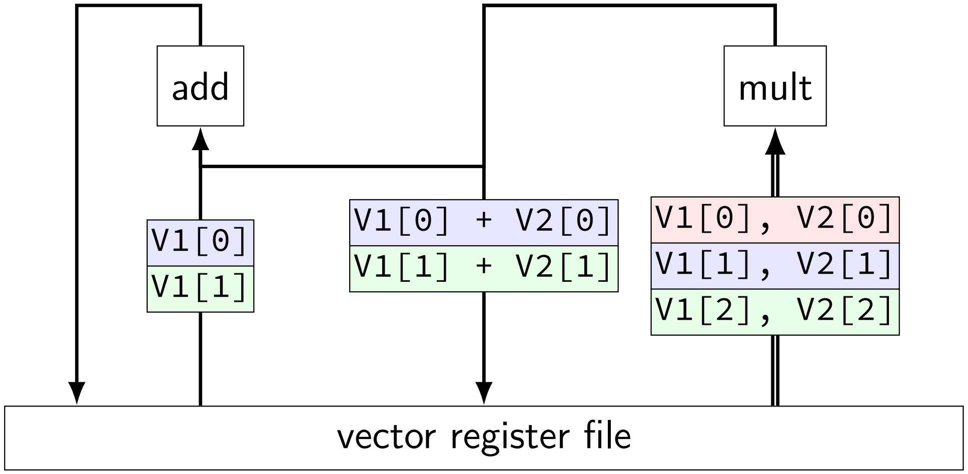
chaining

$$V3 := V1 \times V2$$
$$V0 := V1 + V3$$



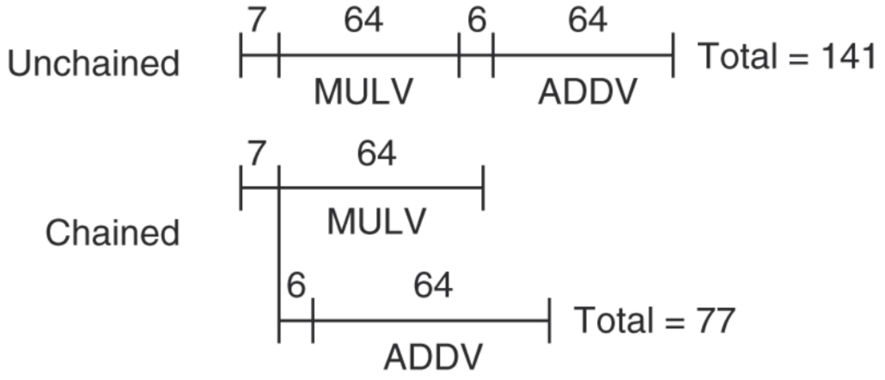
chaining

$$V3 := V1 \times V2$$
$$V0 := V1 + V3$$



chaining timing

7-cycle multiply latency, 6-cycle add latency,
64-element vector:



Unchained

7 64 6 64

MULV ADDV

Total = 141

Chained

7 64

MULV

6 64

ADDV

Total = 77

Hennessy and Patterson, Figure G.8

start-up overhead

time to first result

$7 + 6$ cycles in the chaining example

register read + functional unit latency

start-up overhead

time to first result

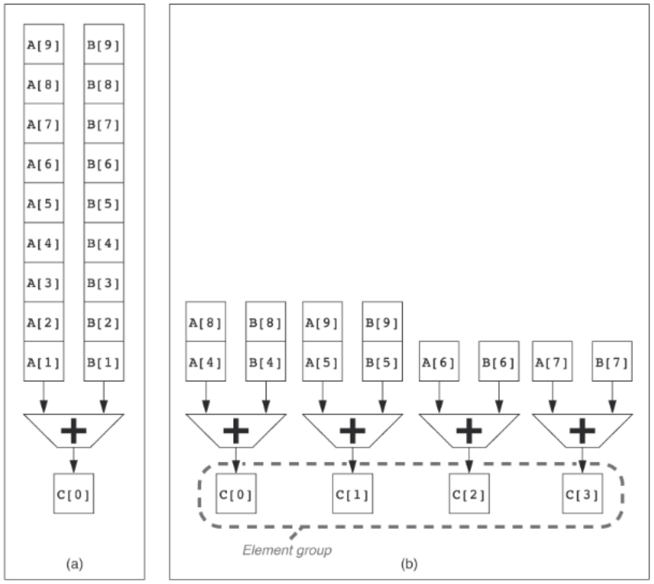
$7 + 6$ cycles in the chaining example

register read + functional unit latency

hidden with pipelining?

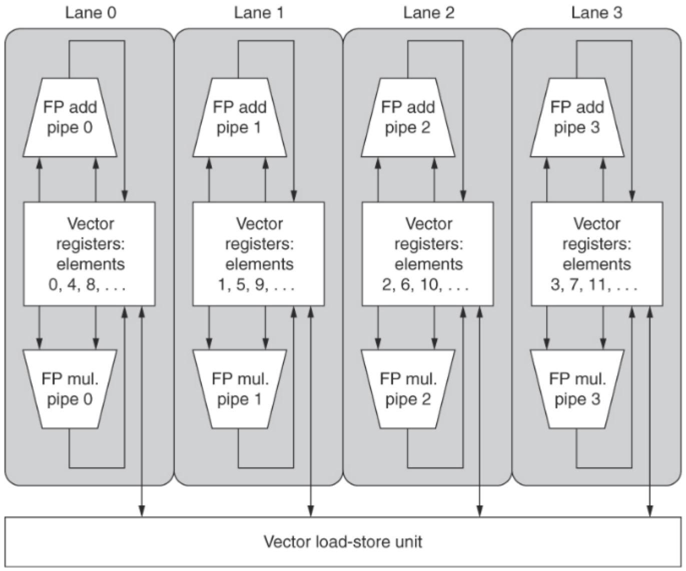
needs logic to overlap non-chained operations

doing multiple operations at once



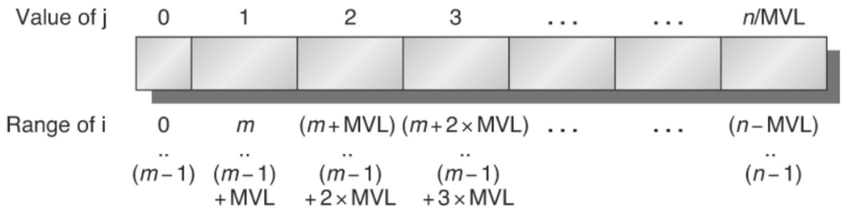
Hennessy and Patterson, Figure 4.4

lanes — spreading out vectors


Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, . . .
Vector registers: elements 1, 5, 9, . . .
Vector registers: elements 2, 6, 10, . . .
Vector registers: elements 3, 7, 11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

Hennessy and Patterson, Figure 4.5

diving up an array



Hennessy and Patterson, Figure 4.6

Vector length registers

Cray 1: vector register holds up to 64 values

VL — vector length register

indicates how many of 64 values are used

remaining elements unchanged

Dealing with branches

do nothing

vector mask register

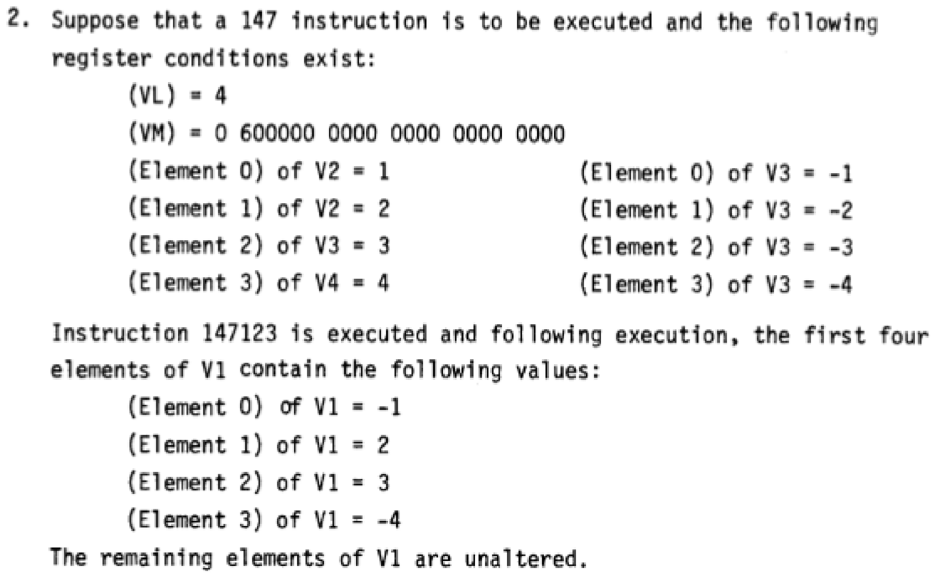
Cray-1 Vector Merge

Vector Mask = [1, 1, 1, 0, 0, 1, 1]

V3 = Merge(V1, V2):

V3[i] = V1[i] if Mask[i] == 1
V3[i] = V2[i] otherwise

Cray-1 Vector merge example



2. Suppose that a 147 instruction is to be executed and the following register conditions exist:

```
(VL) = 4
(VM) = 0 600000 0000 0000 0000 0000
(Element 0) of V2 = 1            (Element 0) of V3 = -1
(Element 1) of V2 = 2            (Element 1) of V3 = -2
(Element 2) of V3 = 3            (Element 2) of V3 = -3
(Element 3) of V4 = 4            (Element 3) of V3 = -4
```

Instruction 147123 is executed and following execution, the first four elements of V1 contain the following values:

```
(Element 0) of V1 = -1
(Element 1) of V1 = 2
(Element 2) of V1 = 3
(Element 3) of V1 = -4
```

The remaining elements of V1 are unaltered.

Cray-1 Hardware Reference Manual

Setting Vector Masks

Cray-1 has two options:

load integer register into vector mask

set based on vector register, bit i is 1 if element i of register is:

- zero
- nonzero
- negative
- positive

GPU branching

`SLT V3, V1, V2` (Set Less Than):

V3[i] = 1.0 if V1[i] < V2[i]
V3[i] = 0.0 otherwise

example: R3 = MIN(R1, R2)

```
SLT R4, R1, R2
MUL R4, R1, R4
SGE R5, R1, R2
MUL R5, R2, R5
ADD R3, R5, R4
```

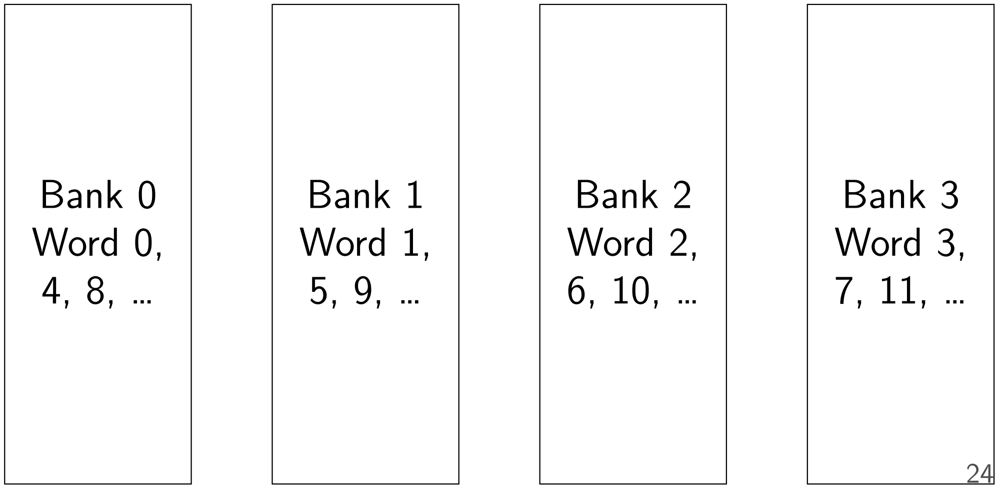
Cray Branching

```
/* V3 = MIN(V1, V2) */
/* pseudo—assembly */
VM <— LESS—THAN(V1, V2)
/* VM[x] = 1 if V1[x] < V2[x] */
V3 <— MERGE(V1, V2)
/* V3[x] = V1[x] if VM[x] = 1 */
```

Memory banks

want parallelism from loads/stores

trick: interleave memory



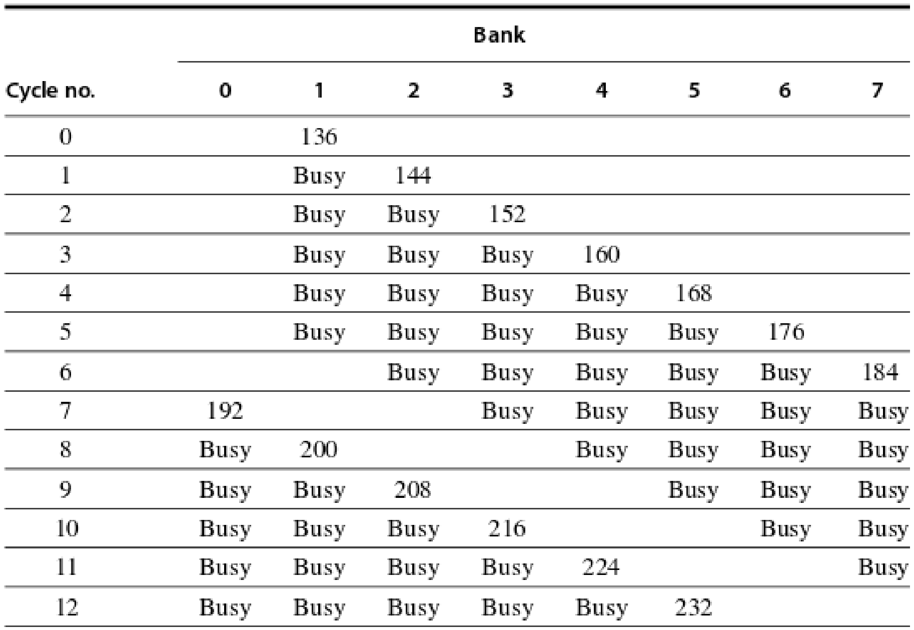
Bank 0 Word 0, 4, 8, …

Bank 1 Word 1, 5, 9, …

Bank 2 Word 2, 6, 10, …

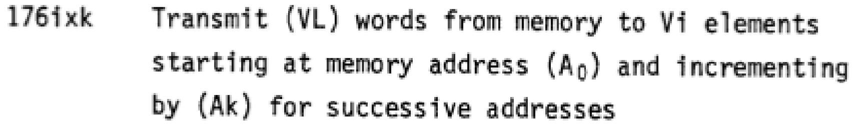
Bank 3 Word 3, 7, 11, …

Multiple banks: timeline



	Bank							
Cycle no.	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		

Cray-1 loading vectors



```
176ixk    Transmit (VL) words from memory to Vi elements
          starting at memory address (A0) and incrementing
          by (Ak) for successive addresses
```

load instruction

V1[0] = memory[A0]

V1[1] = memory[A0 + Ak]

V1[2] = memory[A0 + 2*Ak]

…

Strides

a matrix (logically):

$$\begin{array}{cccc} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ \dots & \dots & \dots & \dots \end{array}$$

Strides

typical memory layout:

0: A_{00}
1: A_{01}
2: A_{02}
3: A_{03}
4: A_{10}
5: A_{11}
6: A_{12}
7: A_{13}
8: A_{20}
...

a matrix (logically):

$$\begin{array}{cccc} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ \dots & \dots & \dots & \dots \end{array}$$

Strides

typical memory layout:

0: A_{00}
1: A_{01}
2: A_{02}
3: A_{03}
4: A_{10}
5: A_{11}
6: A_{12}
7: A_{13}
8: A_{20}
...

a matrix (logically):

$$\begin{array}{cccc} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ \dots & \dots & \dots & \dots \end{array}$$

access column 0 — stride 4

Vector loads/stores

bad strides create bank conflicts

latency of memory may be visible

GPU: sources of parallelism

```
MUL R0.xyzw, R1.xywz, R2.xywz
```

1 instruction, four multiplies:
R0.x = R1.x × R2.x
R0.y = R1.y × R2.y
...

hardware multithreading

like Tera machine — fixed latency makes simple
round-robin between threads

similar effect to chaining (since same program, no branches)

Cray-1-style machines: parallelism

convoys/chaining — overlap consecutive instructions

overlap fetch/setup with computation:

second element fetched while first computing

first can't overlap — “start-up time”

Vector versus Out-of-Order

both ways of making efficient use of functional units

ideal: every functional unit used every cycle
forward values as soon as they are ready

vector: much less complexity for processor

faster?
more space for functional units/registers?
multiple lanes instead of wider/slower register files?

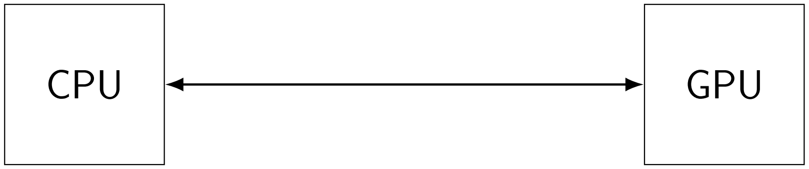
GPU: specialization

limited input and output and memory

special instructions for lighting computations

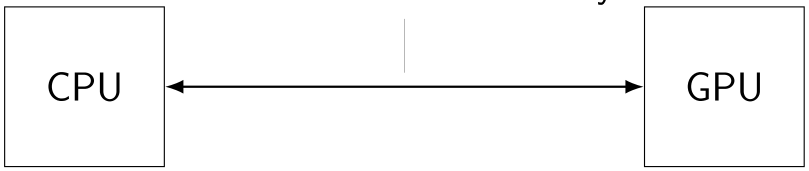
(almost) no integer operations

GPU and the CPU



GPU and the CPU

same bus used for memory?



communicating with the GPU (1)

typical CPU interface — talk to memory bus

GPU (and/or its controller) listens to memory reads/writes

write to memory special memory location — sends command

memory locations often called “registers”

(even if they aren’t really registers)

communicating with the GPU (2)

DMA — direct memory access

CPU: write values to memory (e.g. list of vertices)

CPU: send command to GPU with memory address

GPU: read values (e.g. list of vertices) from memory

CPU: do other computation while GPU is reading from memory