

EEL 6764 Principles of Computer Architecture

Data-Level Parallelism in Vector, SIMD and GPU Architectures

Dr Hao Zheng
Dept of Comp Sci & Eng
U of South Florida

Flynn's Classification

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instructions	Not Exists	MIMD

Data Parallelism

- Concurrency arises from performing the **same operations on different pieces of data**
 - Single instruction multiple data (SIMD)
 - E.g., dot product of two vectors
- Contrast with data flow
 - Concurrency arises from executing different operations in parallel (in a data driven manner)
- Contrast with thread (“control”) parallelism
 - Concurrency arises from executing different threads of control in parallel

Introduction

- SIMD architectures can exploit significant data-level parallelism for:
 - matrix-oriented scientific computing
 - media-oriented image and sound processing
- SIMD is more energy efficient than MIMD
 - Fetch one instruction for many data operation
 - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially
 - Unlike MIMD

SIMD Processing

- Single instruction operates on multiple data elements
 - In time – PEs are pipelined
 - In space – Multiples PEs
- Time-Space Duality
 - **Array processor** – instruction operates on multiple data at the same time, needs duplicates of PEs
 - **Vector processor** – instruction operates on multiple data in consecutive time steps, PEs needs to be pipelined

Vector Architecture

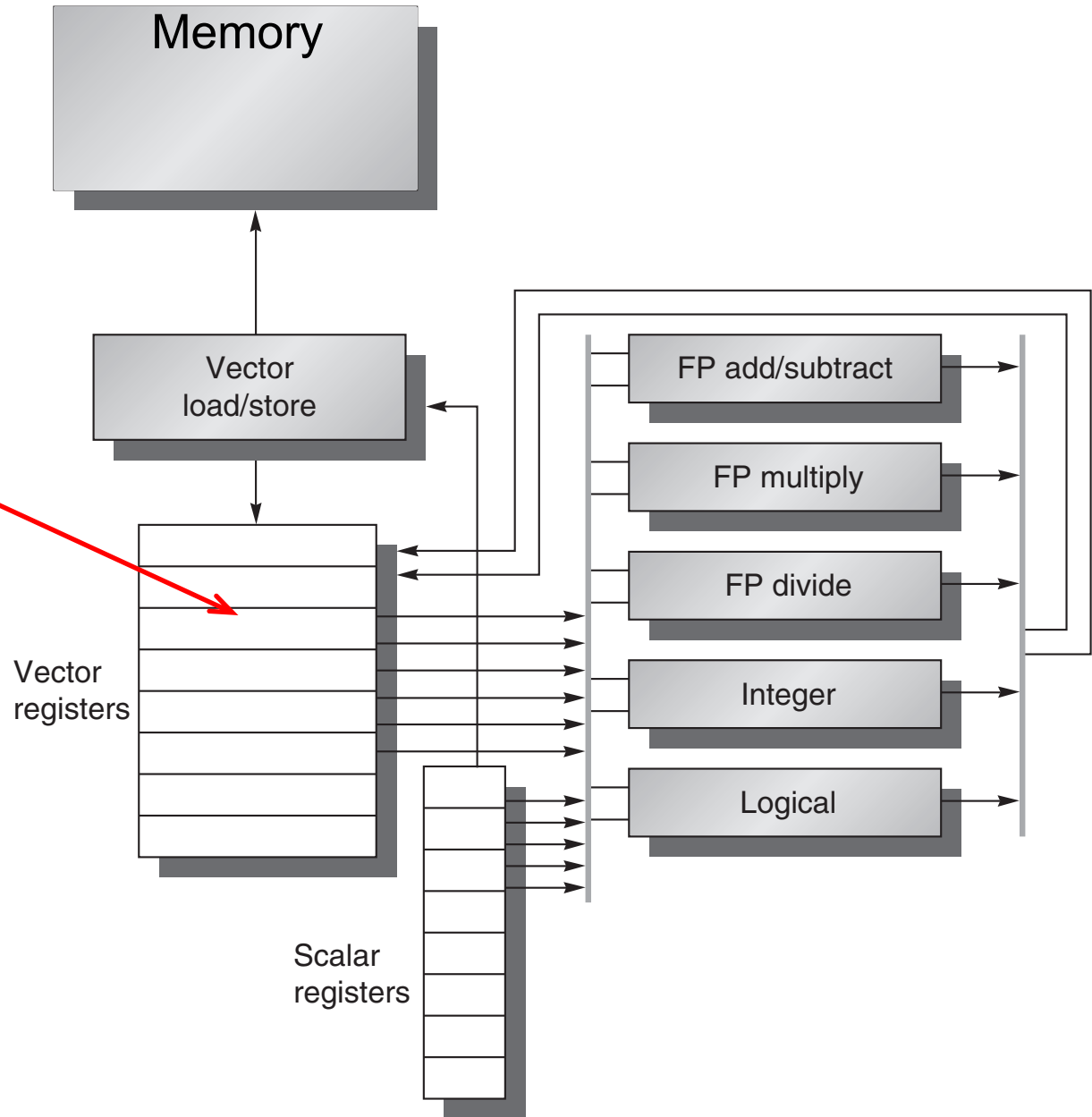
- A vector is a one-dimensional array of numbers
 - Many scientific/commercial applications use vector
- An instruction operates on vectors, instead of scalar values
 - Each instruction generates a lot of work
- Basic idea:
 - Read vectors of data elements into **vector registers**
 - Vector FUs operate on those registers in a pipelined manner one element at a time
 - Disperse the results back into memory

Vector Architecture

- FUs are deeply pipelined
 - No intra-vector dependence – no HW interlocking within a vector
 - No control-flow within a vector
- Registers are controlled by compiler
 - Used to hide memory latency
 - Leverage memory bandwidth
- Deliver high performance without energy/design complexity of out-of-order superscalar processor
 - Increase performance of in-order simple scalar

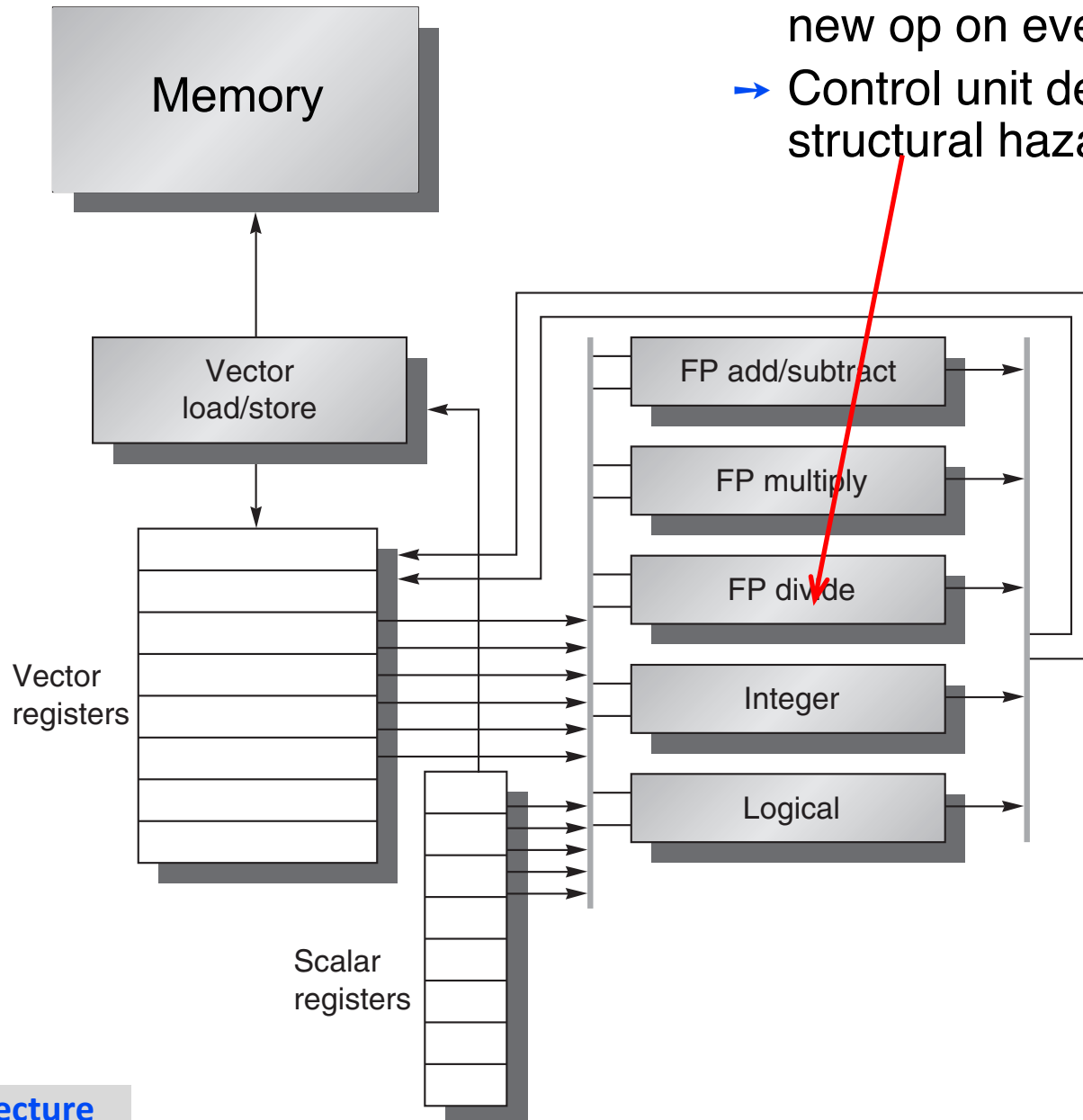
RISC-V Vector Architecture

- Each register holds a 64-element, 64 bits/element vector
- Register file has 16 read ports and 8 write ports
- Other special purpose registers, i.e VLR/MVL.



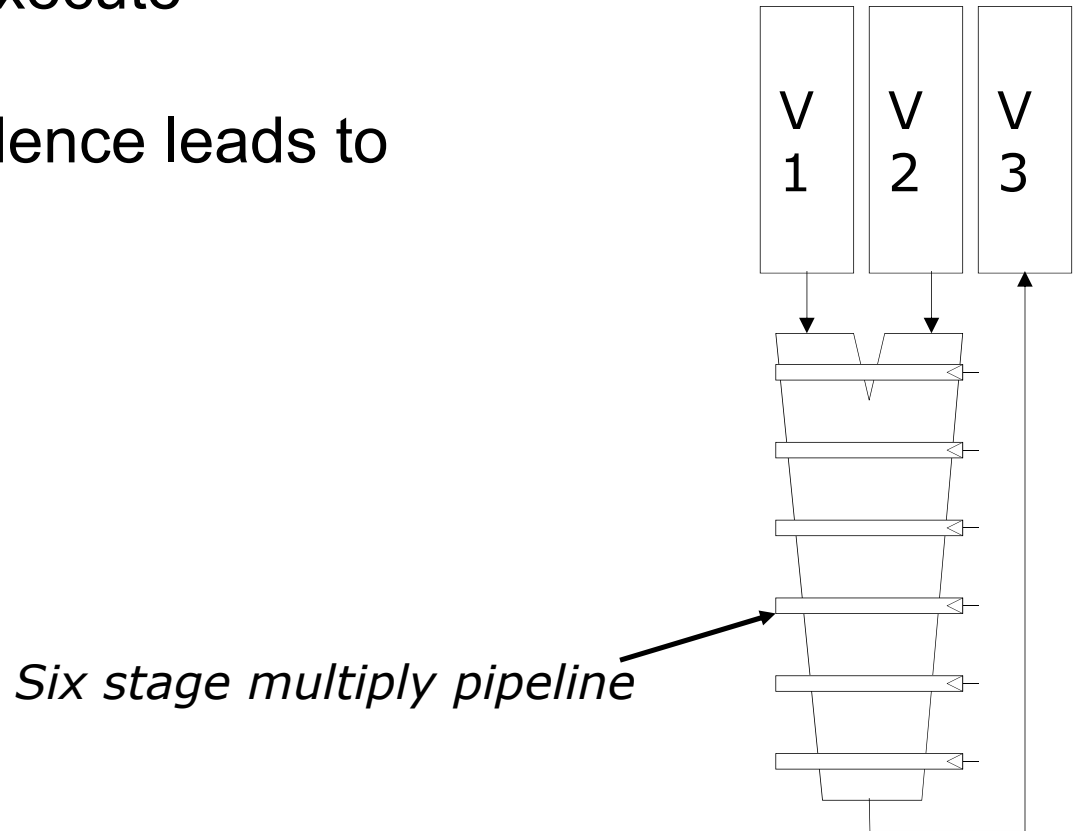
RISC-V Vector Architecture

- Fully pipelined – start a new op on every cycle
- Control unit detects data & structural hazards



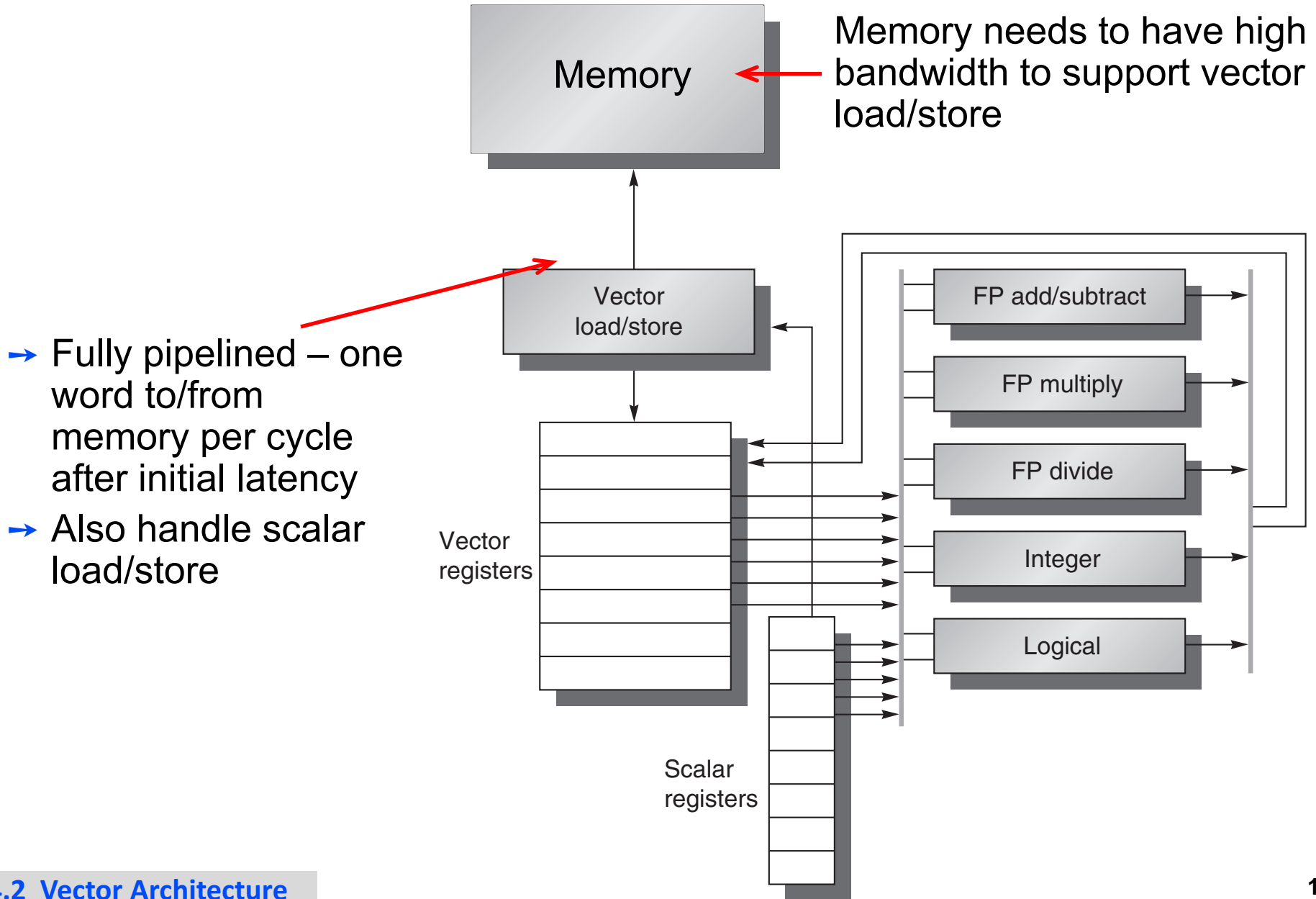
Vector Functional Units

- Use deep pipeline to execute elements at fast speed
- No intra-vector dependence leads to simple pipeline control

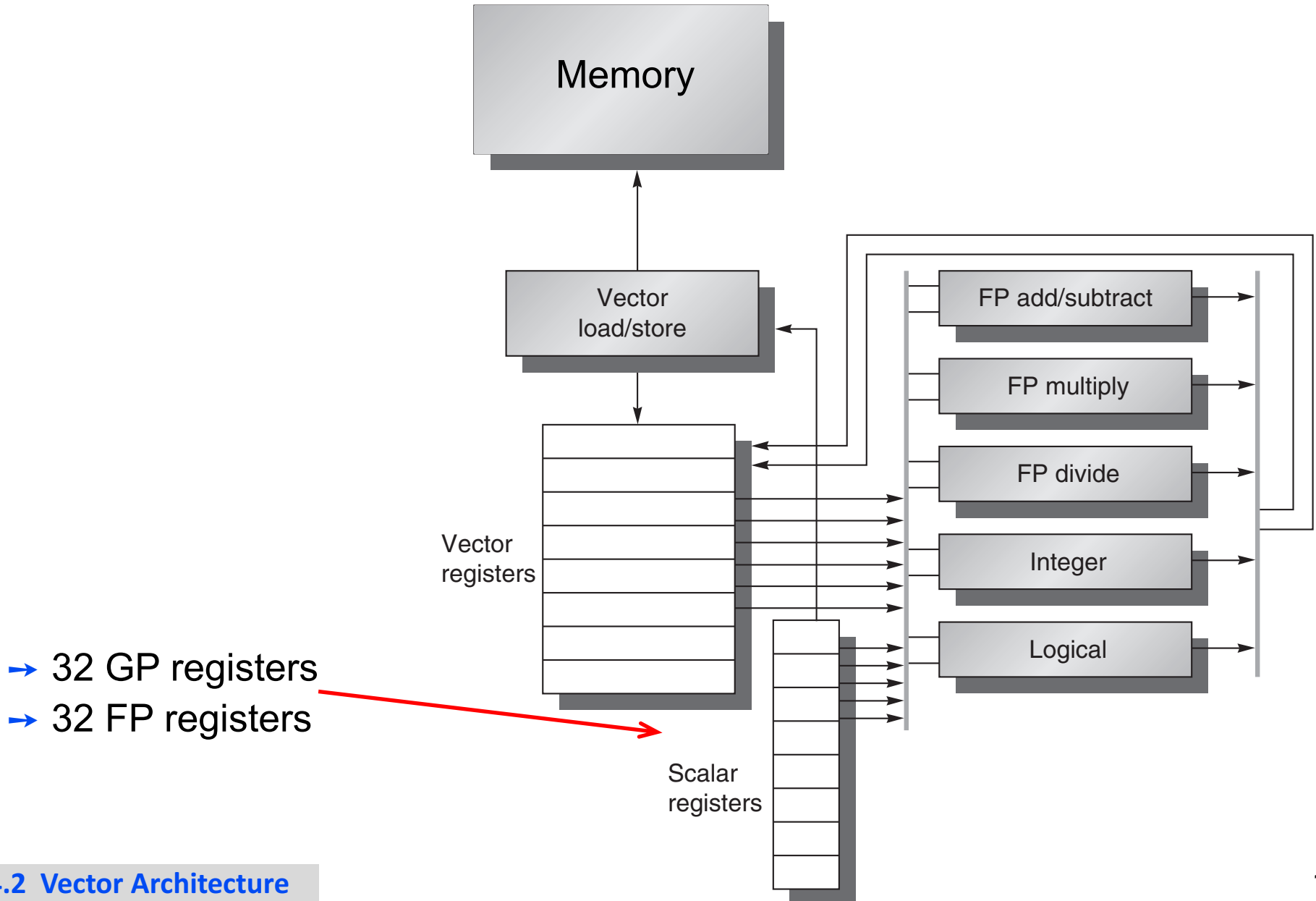


$$V3 \leftarrow v1 * v2$$

RISC-V Vector Architecture



RISC-V Vector Architecture



RISC-V Vector Instructions

- **vadd**: add two vectors
- **vld/vst**: vector load and vector store from address
- Example: $\text{DAXPY} - \mathbf{Y} = a * \mathbf{X} + \mathbf{Y}$; vector length = 64

vsetdcfg	4*FP64	# Enable 4 DP FP vregs
fld	f0, a	# Load scalar a
vld	v0, x5	# Load vector X
vmul	v1, v0, f0	# Vector-scalar mult
vld	v2, x6	# Load vector Y
vadd	v3, v1, v2	# Vector-vector add
vst	v3, x6	# Store the sum
vdisable		# Disable vector regs

- Requires 6 instructions vs. almost 600 for MIPS

VMIPS Vector Instructions

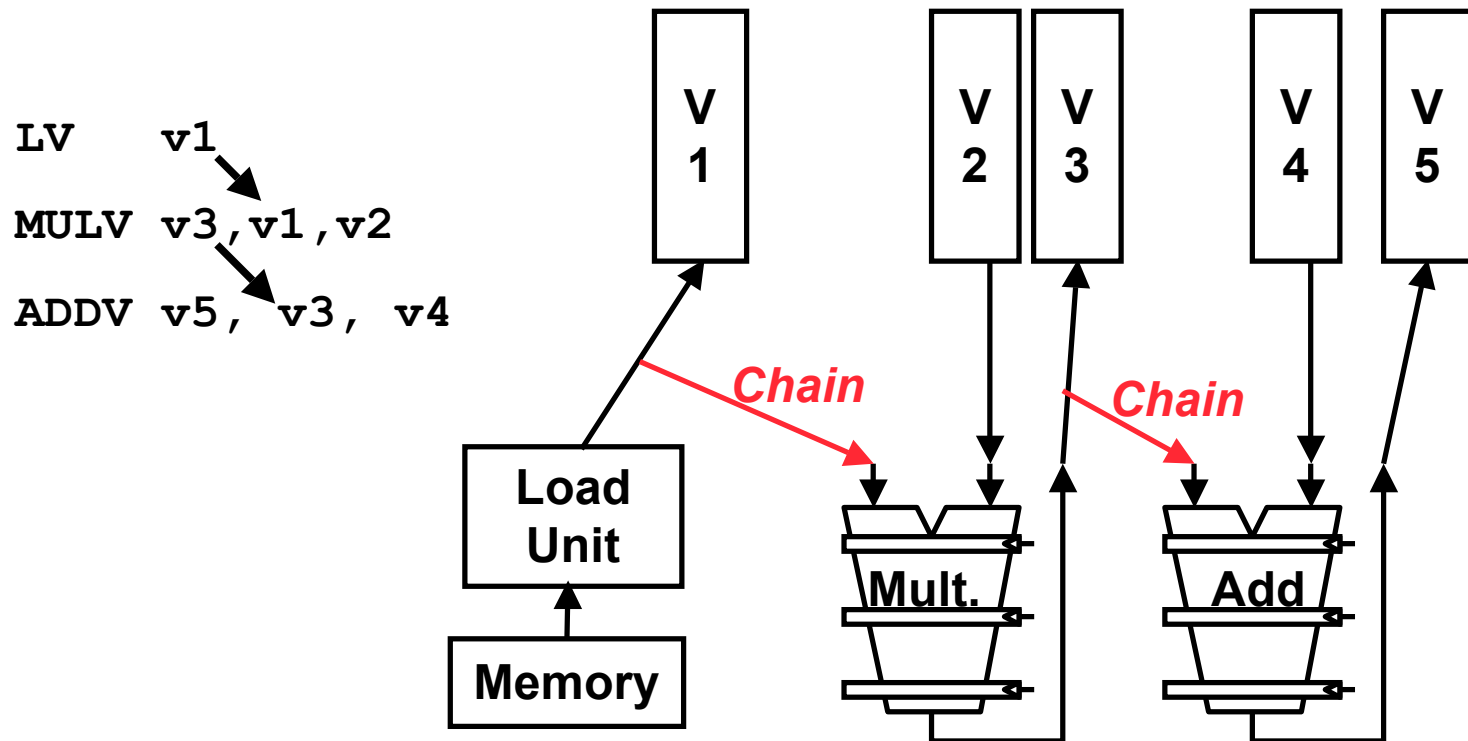
→ Example: DAXPY – $Y = a * X + Y$

```
        fld      f0, a           # Load scalar a
        addi     x28, x5, 256    # Last address to load
Loop:   fld      f1, 0(x5)       # Load X[i]
        fmul.d   f1, f1, f0      # a × X[i]
        fld      f2, 0(x6)       # Load Y[i]
        fadd.d   f2, f2, f1      # a × X[i] + Y[i]
        fsd      f2, 0(x6)       # Store into Y[i]
        addi     x5, x5, 8       # Increment index to X
        addi     x6, x6, 8       # Increment index to Y
        bne      x28, x5, Loop   # Check if done
```

→ Almost 600 MIPS instructions

Vector Chaining – Handling D-Dependence

- Vector version of forwarding
- Next vector operation starts when the result from previous operation on the 1st element is finished.

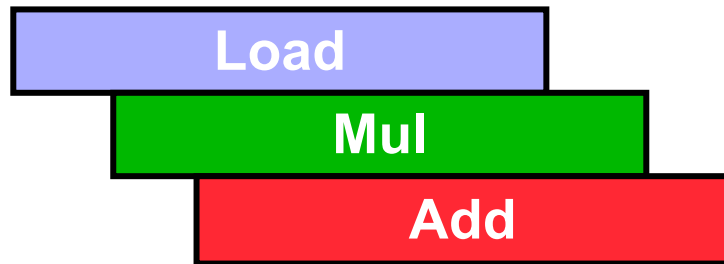


Vector Chaining

→ Without chaining



→ With Chaining



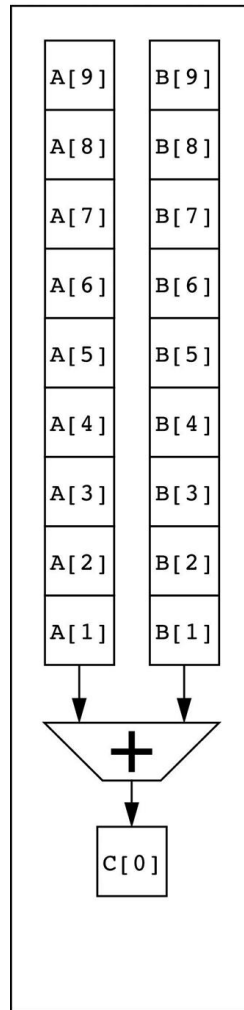
Multiple Lanes

- How can a vector processor execute a single vector faster than one element in a clock cycle?
- **Lane**: one copy of FU + a portion of vector register file

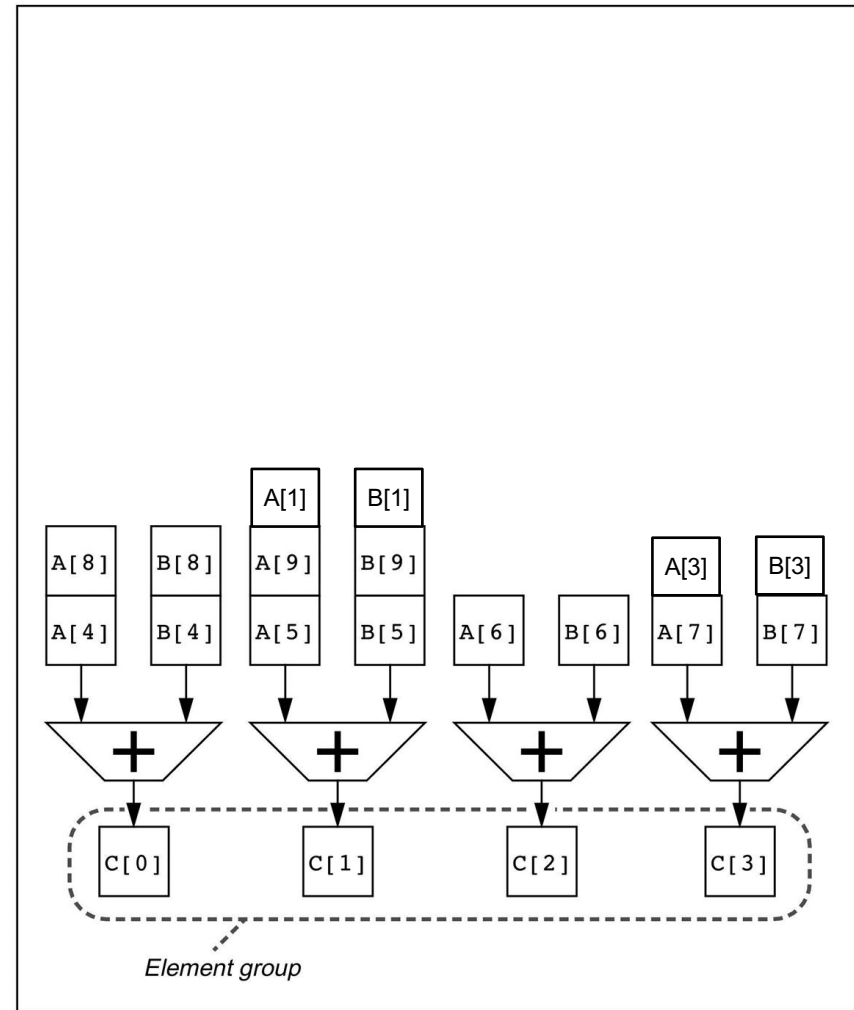
Multiple Lanes

→ Goal: Execute >1 element per cycle.

- Use multiple FUs to improve performance of a single vector add operation $\mathbf{C} = \mathbf{A} + \mathbf{B}$
- Elements are interleaved across FUs
- Increase performance to 4 elements per cycle from 1 per cycle

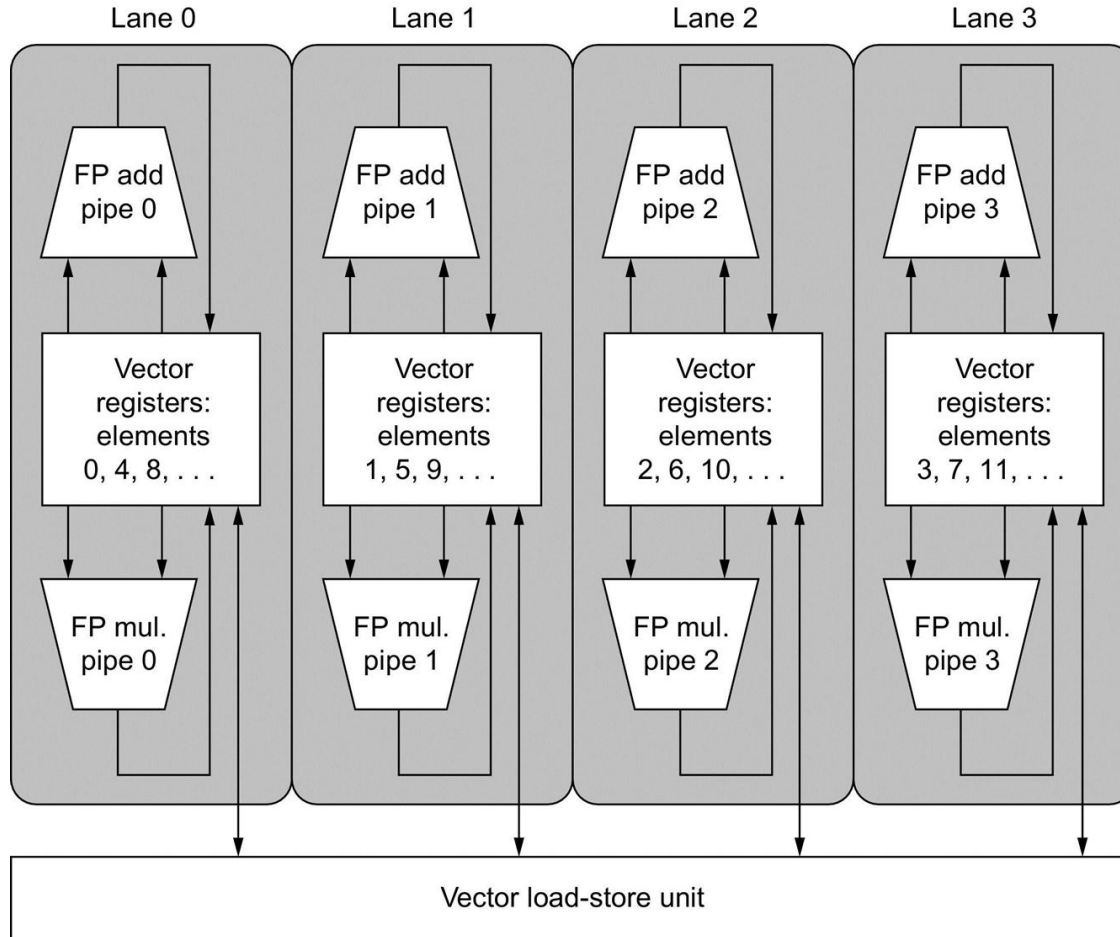


(A)



(B)

Multiple Lanes



Both applications and HW architecture must support long vectors to take advantage of the higher processing power of multiple lanes

Increase performance with the *same* power – by reducing clock rate by half, and doubling lanes


- Element *n* of vector register *A* is “hardwired” to element *n* of vector register *B*
- Each lane handles a portion of vector registers

Vector Length Register (VLR)

- What if vector length is different from the length of vector registers? Or
- The length is known only at runtime?

Known only at runtime

for (i=0; i < *n*; i = i + 1)
Y[i] = a * X[i] + Y[i]



VLR controls the length of every vector operation

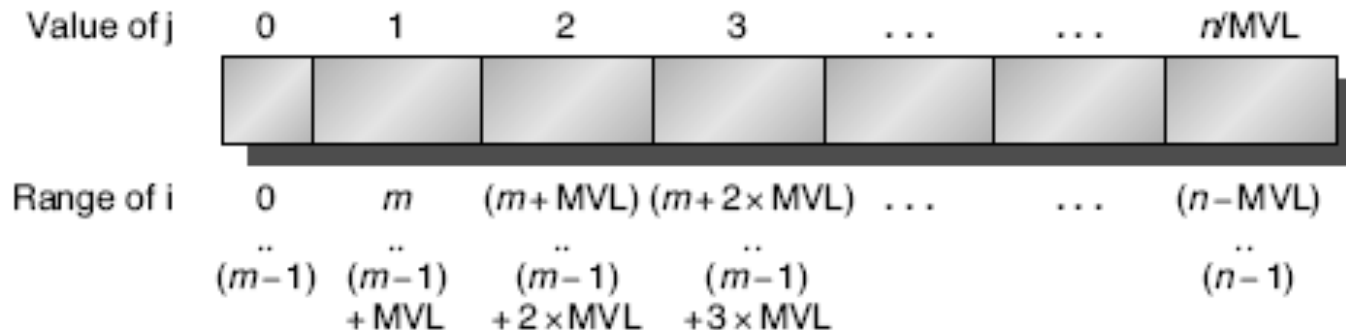
- Set by **setvl**

- What if *n* is not known at compile time, but may be larger than the *maximal vector length* (MVL)?

Strip Mining

```
for (i=0; i <  $n$ ; i = i + 1)
     $Y[i] = a * X[i] + Y[i]$ 
```

```
low = 0;
VL = (n % MVL);           /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) {           /*outer loop*/
    for (i = low; i < (low+VL); i=i+1)       /*runs for length VL*/
         $Y[i] = a * X[i] + Y[i]$ ;           /*main operation*/
    low = low + VL;                         /*start of next vector*/
    VL = MVL;                               /*reset the length to maximum vector length*/
}
```



Predicate Register

```
for (i = 0; i < 64; i=i+1)
  if (X[i] != 0)
    X[i] = X[i] - Y[i] ;
```

IF statement introduces control dependence, reduces the level of parallelism

```
vsetdcfg  2*FP64 # Enable 2 64b FP vector regs
vsetpcfgi 1      # Enable 1 predicate register
vld v0,x5       # Load vector X into v0
vld v1,x6       # Load vector Y into v1
fmv.d.x f0,x0   # Put (FP) zero into f0
vpne p0,v0,f0   # Set p0(i) to 1 if v0(i)!=f0
vsub v0,v0,v1   # Subtract under vector mask
vst v0,x5       # Store the result in X
vdisable       # Disable vector registers
vpdisable      # Disable predicate registers
```

Predicate Register

Simple Implementation

- execute all N operations, turn off result writeback according to mask

M[7]=1 A[7] B[7]

M[6]=0 A[6] B[6]

M[5]=1 A[5] B[5]

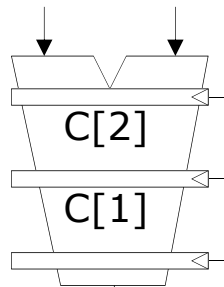
M[4]=1 A[4] B[4]

M[3]=0 A[3] B[3]

M[2]=0

M[1]=1

M[0]=0



Write Enable

Write data port

Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks

M[7]=1

M[6]=0

M[5]=1

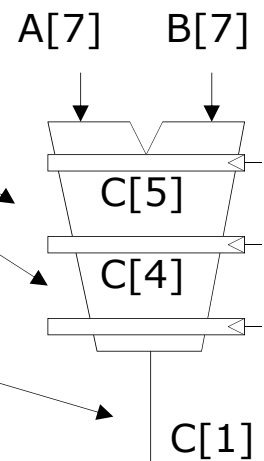
M[4]=1

M[3]=0

M[2]=0

M[1]=1

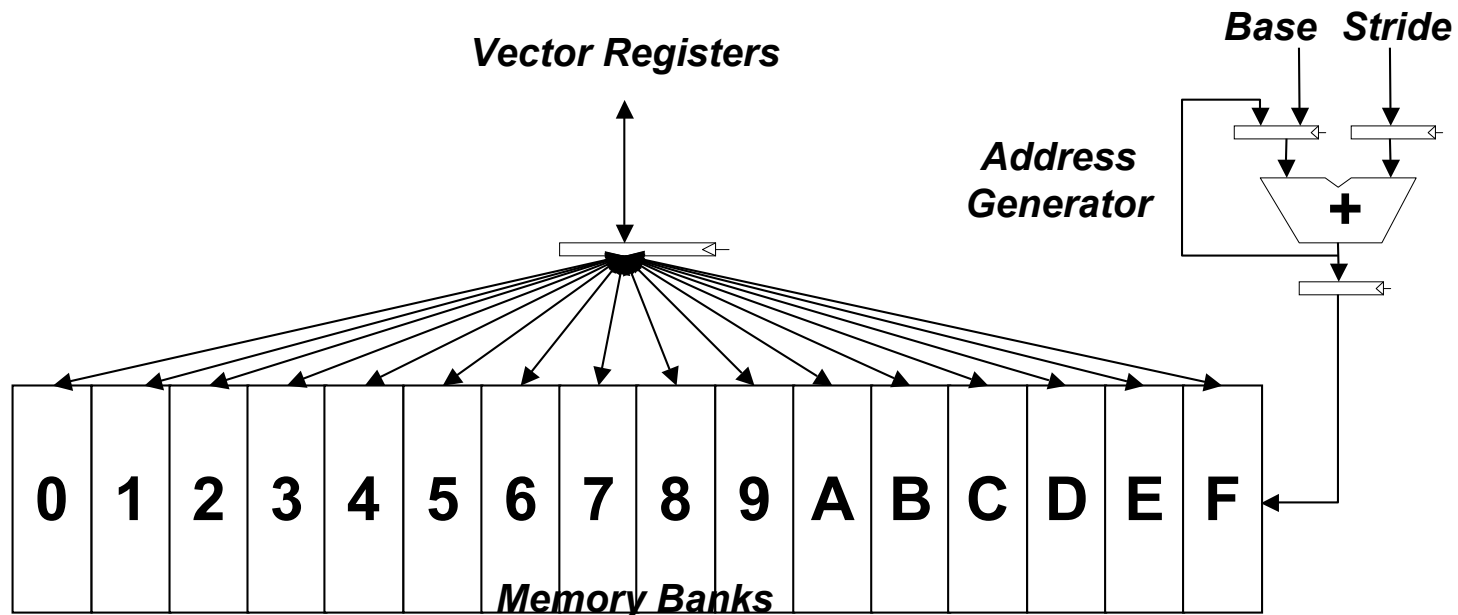
M[0]=0



Write data port

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
 - Control bank addresses independently
 - Load or store non sequential words
 - Support multiple vector processors sharing the same memory



Memory Banks – Example

Question: A vector machine has 32 processors, each can generate 4 loads and 2 stores per cycle. SRAM cycle time is 7 cycles. What is the minimum number of memory banks needed to allow all processors to run at full bandwidth?

Answer: the max number of memory references per cycle is $32 * 6 = 192$.

The min number of memory banks is $192 * 7 = 1344$.

Vector Stride – Handling Multi-D Arrays

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

→ Assume all A, B, D are 100x100 matrices

→ How are matrices stored in memory?

Memory Layout of Matrices

→ Consider $M[i][j]$

Row-major

$M[0][0]$
$M[0][1]$
...
$M[0][j-1]$
$M[1][0]$
...
$M[1][j-1]$
$M[2][0]$
...
$M[2][j-1]$
...

Column-major

$M[0][0]$
$M[1][0]$
...
$M[i-1][0]$
$M[0][1]$
...
$M[i-1][1]$
$M[0][2]$
...
$M[i-1][2]$
...

Vector Stride

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

- Assume all A, B, D are 100x100 matrices
- **Stride** = distance that separate elements in a array to be gathered into a single register.
- For D, in row-major layout, the stride is 100 double words, or 800 bytes

Vector Stride

- Vector base address and stride are stored in GP registers
 - Use **vlds/vsts** for memory access with strides
- Memory bank conflicts due to Mem access with stride
 - Stall memory accesses

Question: 8 Mem banks, with busy time of 6 cycles for each bank. Mem latency is 12 cycles. Time to access 64 data with stride of 1 and 32?

Answer: When stride = 1, access time = $12 + 64 = 76$ cycles.
When stride = 32, access time = $12 + 1 + 6 \cdot 63 = 391$ cycles

Gather and Scatter

- Sparse matrices are arrays where most elements are 0

1	0	0	3	0	0	0	0	0	6
---	---	---	---	---	---	---	---	---	---

- Sparse matrices are stored in a compact format, and accessed indirectly.

```
for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

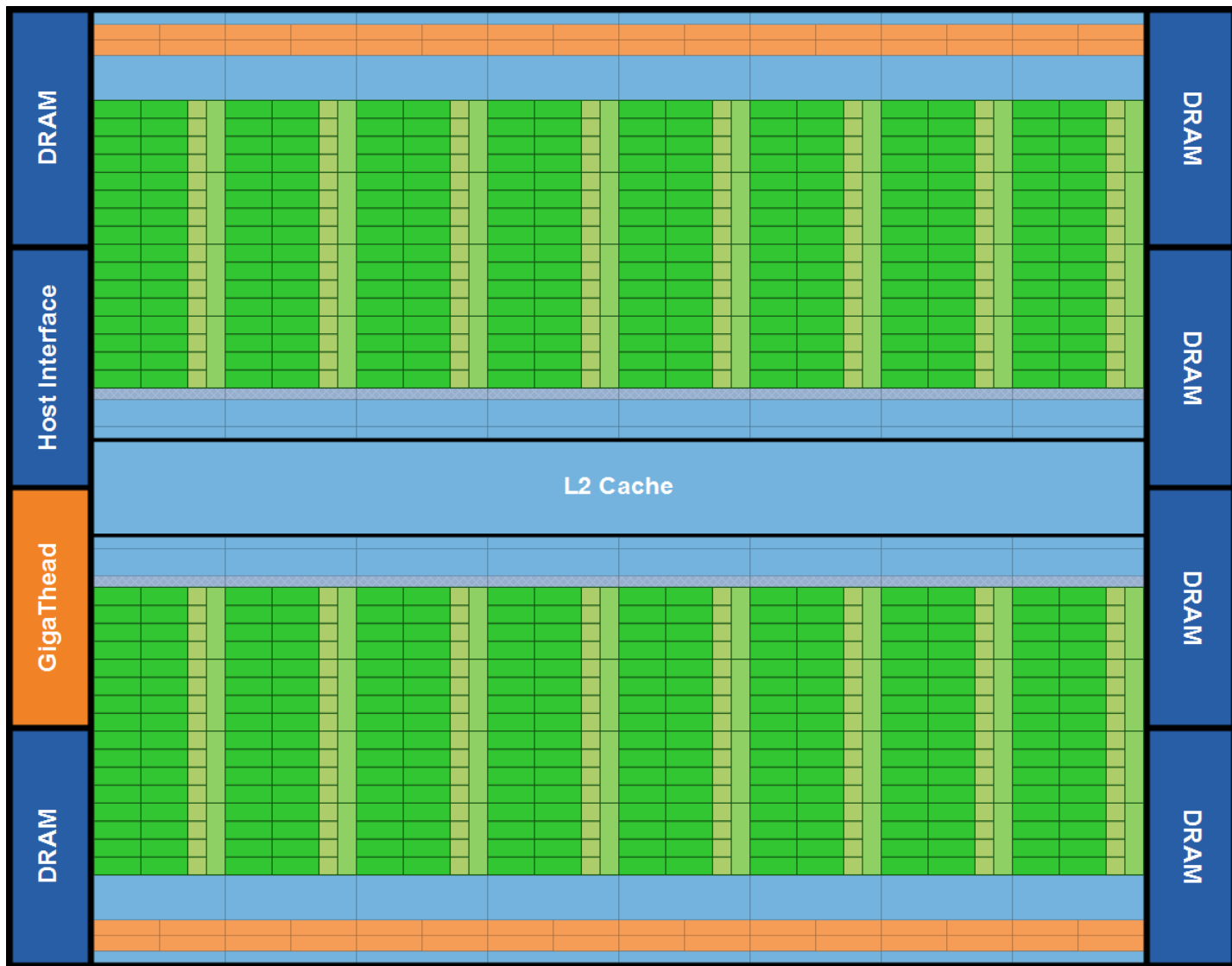
- A and C have the same number of non-zero elements
 - **Index vectors** are K and M are the same size.

Gather and Scatter

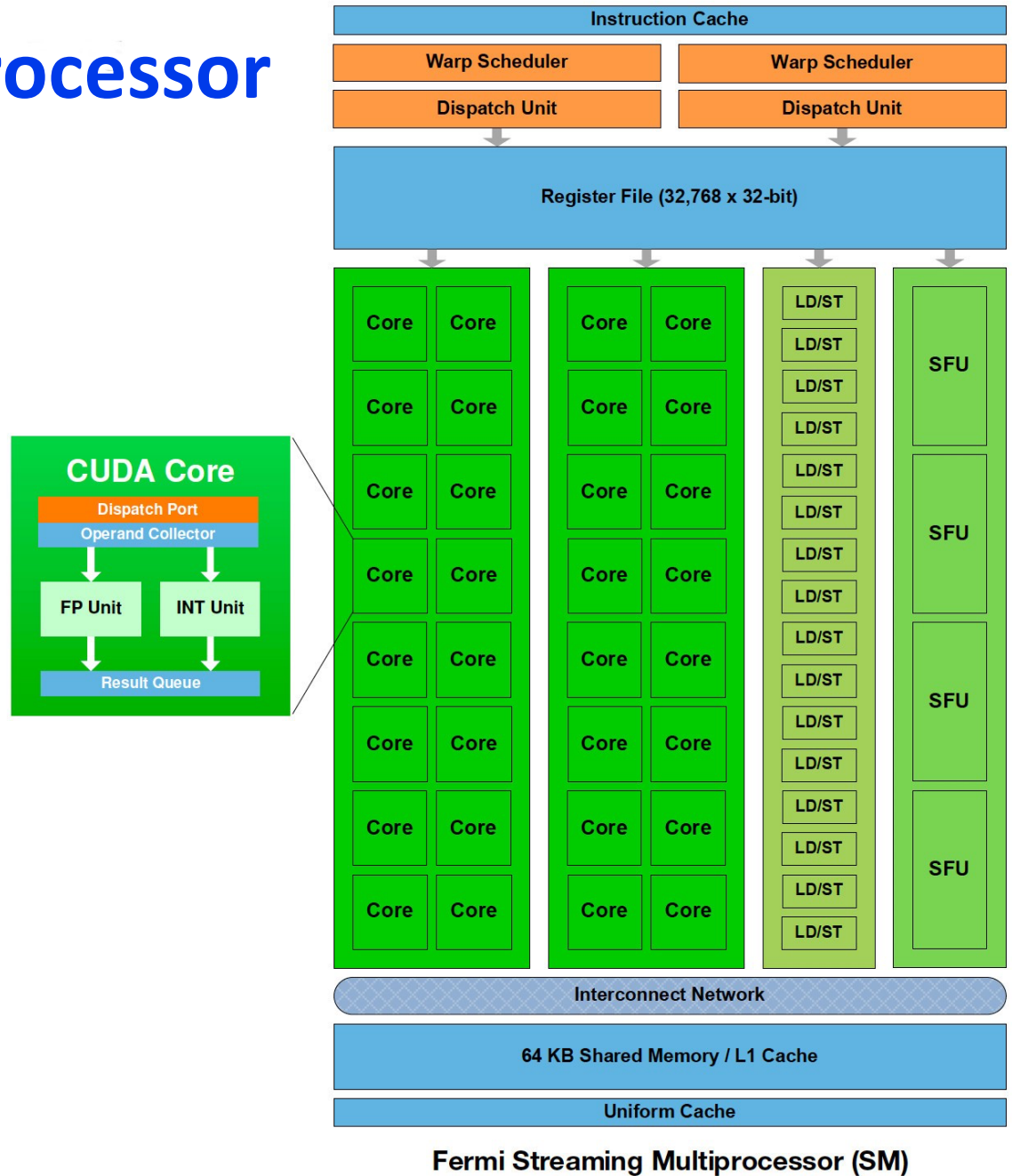
- Access sparse matrices using index vectors.
- **Gather load** takes an index vector and fetches vector elements whose addresses are $\text{base} + \text{offset}$
 - Offsets are stored in the index vector
- **Scatter store** values in vector registers to Mem addresses computed as above.
- Allows sparse matrix operations to run in vector mode

Graphics Processing Units

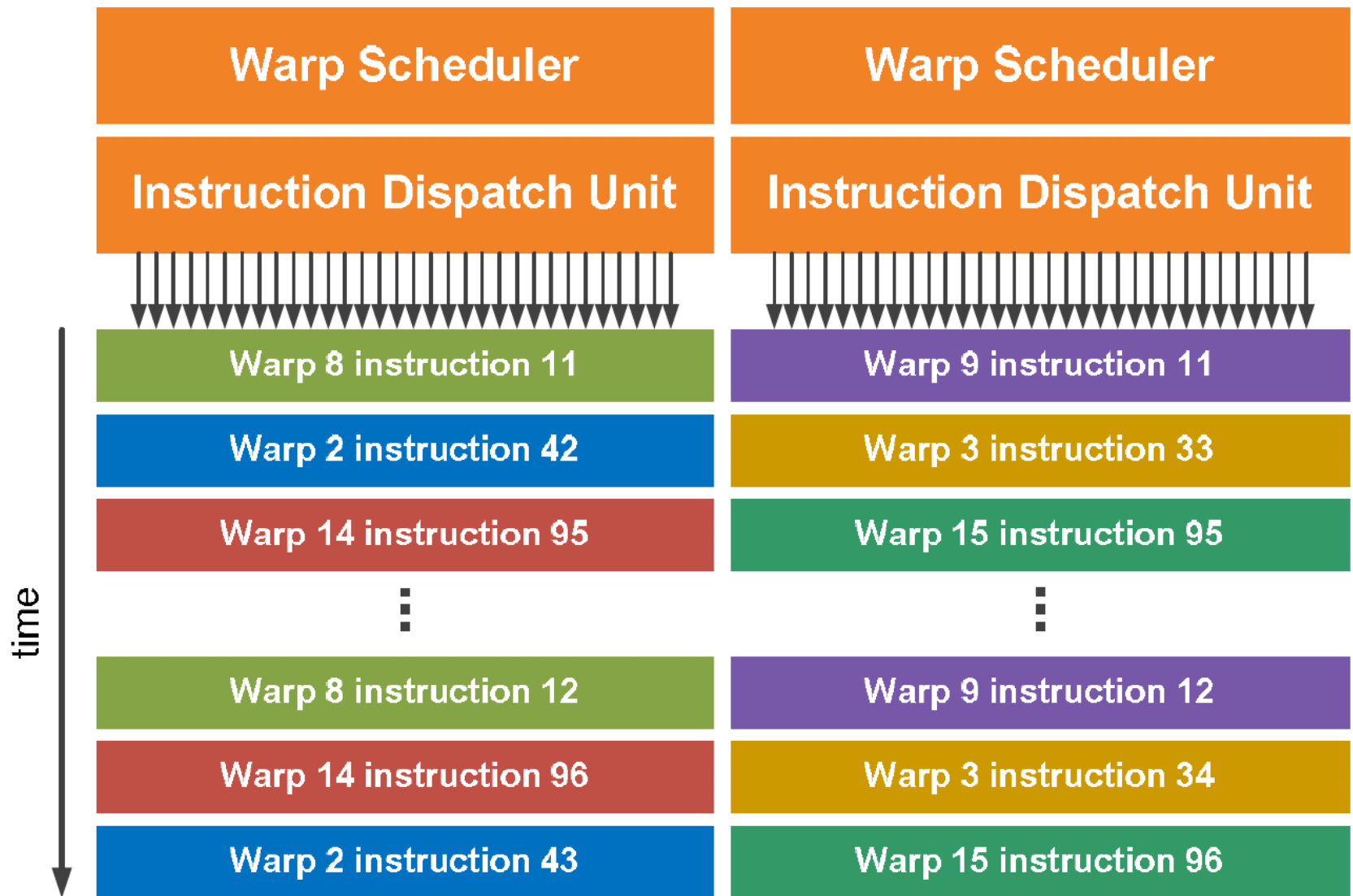
CUDA Architecture



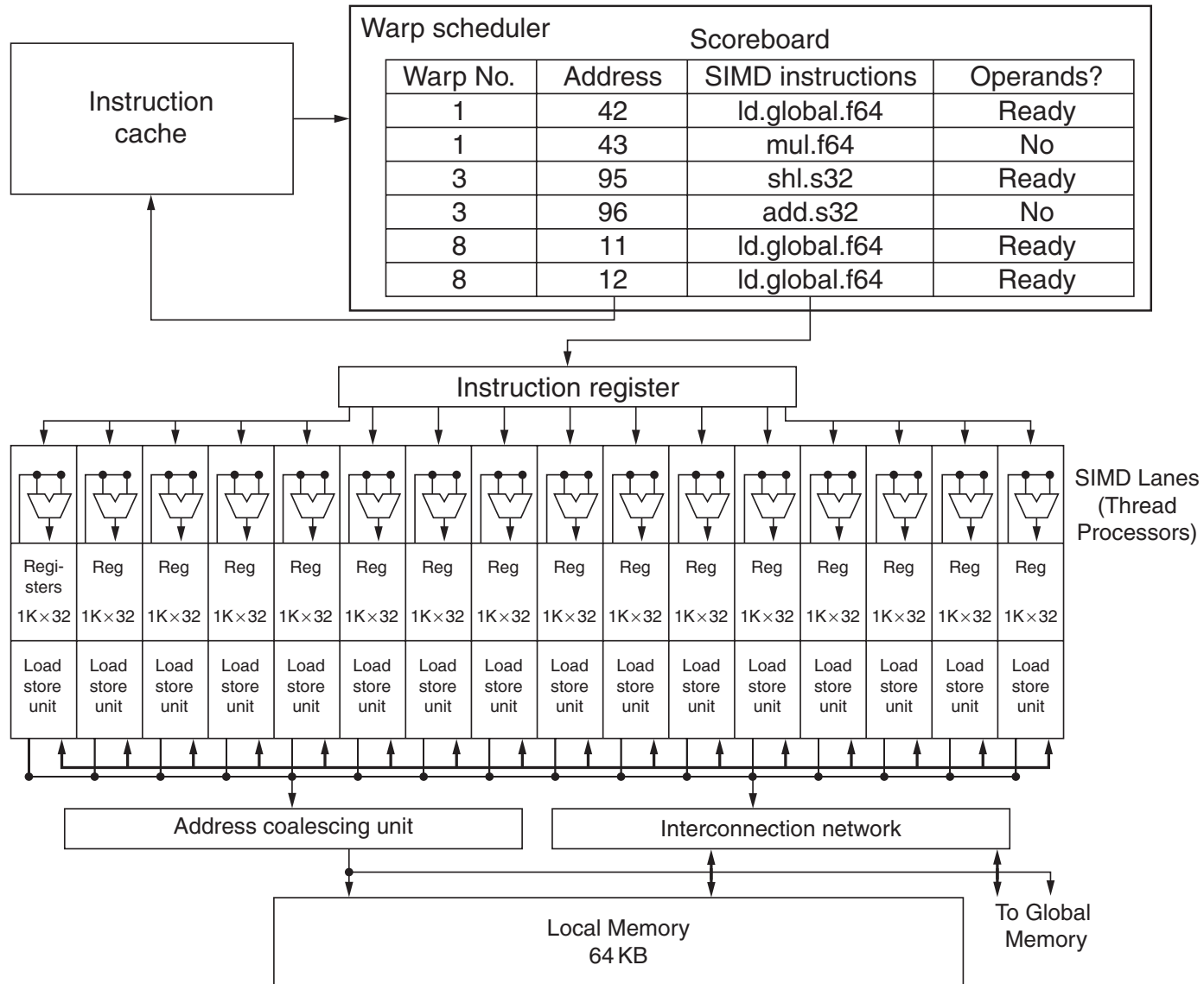
Stream Multiprocessor



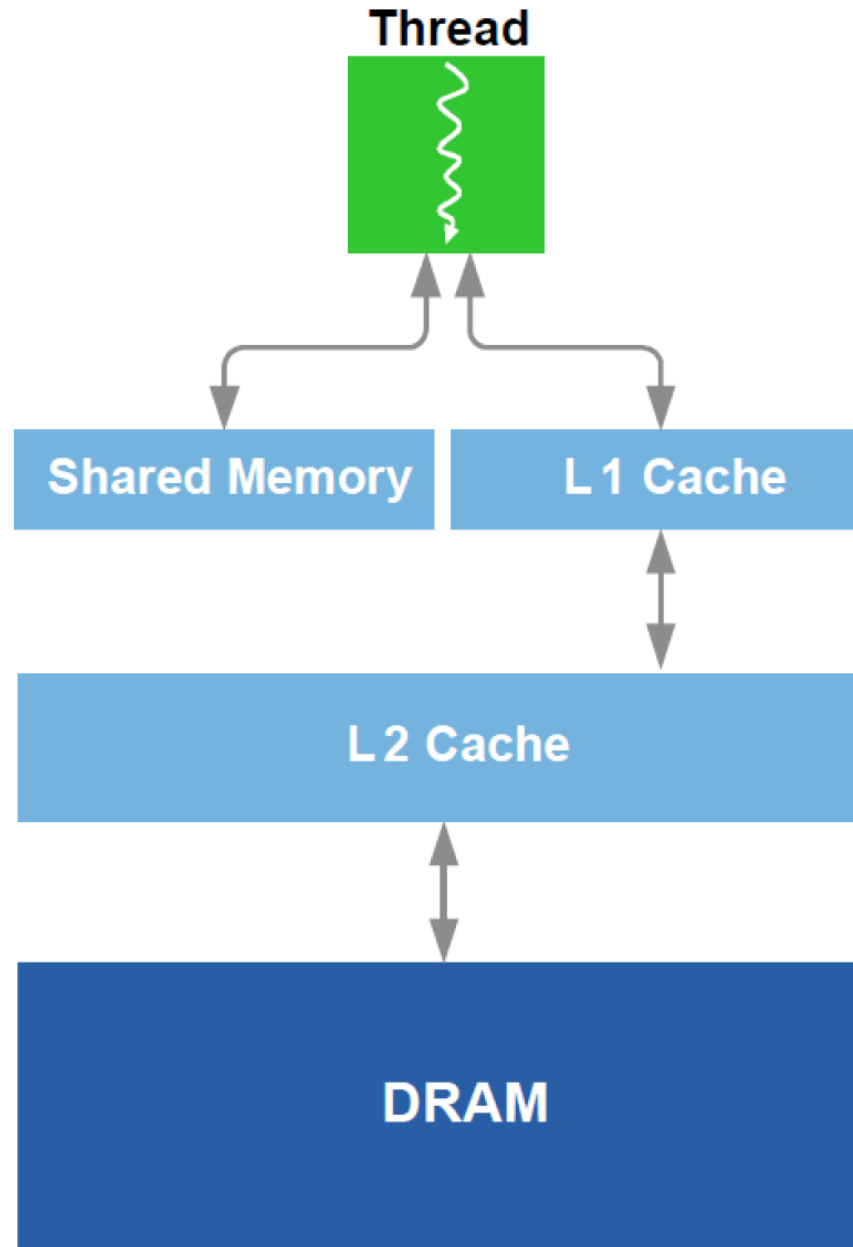
Thread Execution



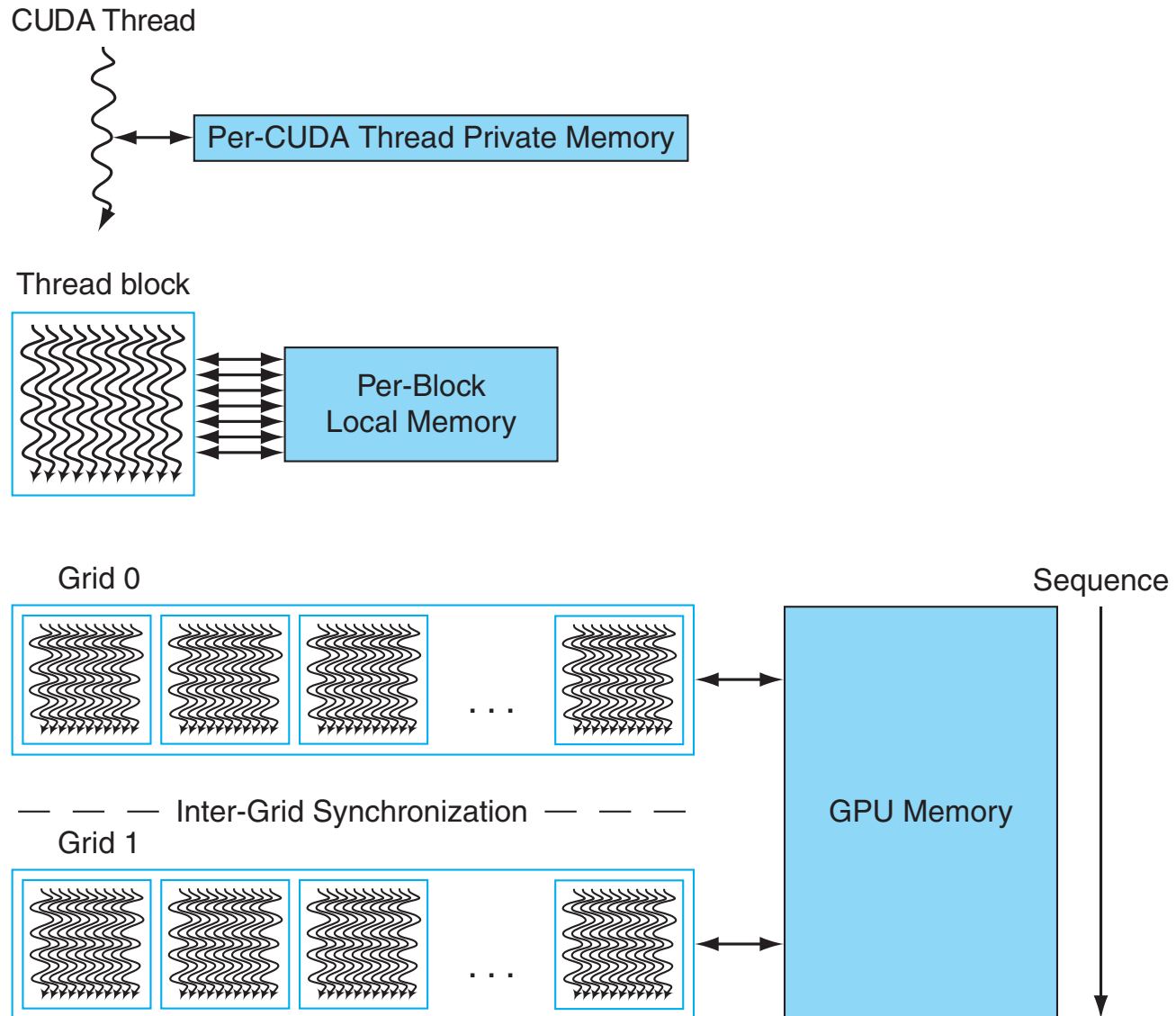
Thread Execution



Fermi Memory Hierarchy



Programming Model



Programming Model

- GPU executes grids of threads
- A stream multiprocessor executes one or more thread blocks
- A CUDA core executes threads

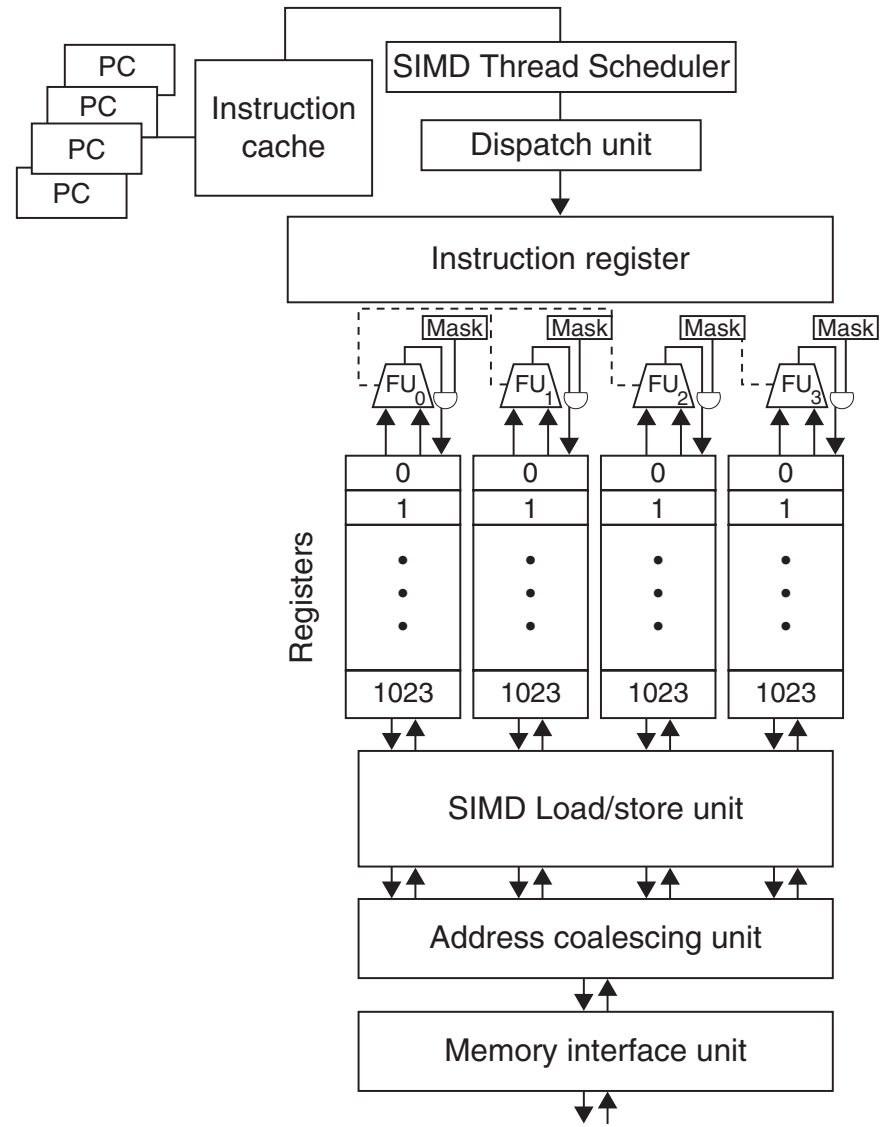
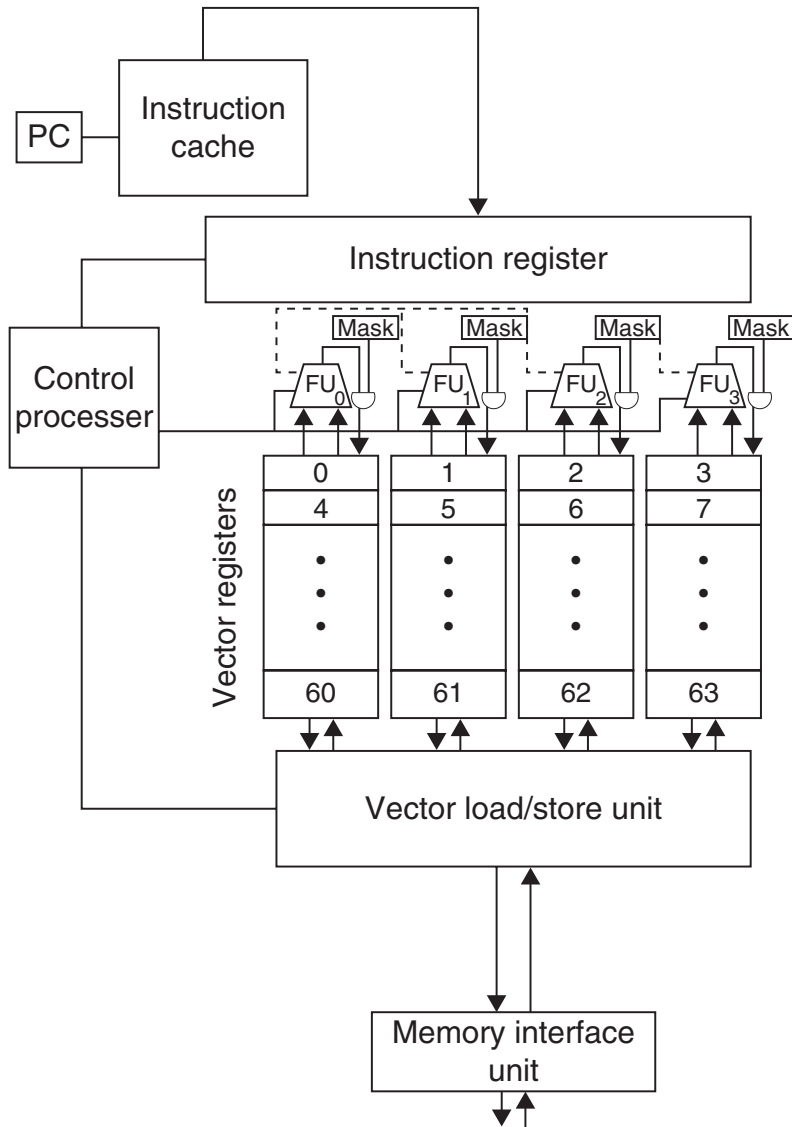
Thread Block 0	SIMD Thread0	A[0] = B [0] * C[0]
		A[1] = B [1] * C[1]
	
		A[31] = B [31] * C[31]
	SIMD Thread1	A[32] = B [32] * C[32]
		A[33] = B [33] * C[33]
	
		A[63] = B [63] * C[63]
	SIMD Thread1 5	A[64] = B [64] * C[64]
	
		A[479] = B [479] * C[479]
		A[480] = B [480] * C[480]
		A[481] = B [481] * C[481]
	
		A[511] = B [511] * C[511]
		A[512] = B [512] * C[512]
		A[7679] = B [7679] * C[7679]
		A[7680] = B [7680] * C[7680]
		A[7681] = B [7681] * C[7681]
	
A[7711] = B [7711] * C[7711]		
SIMD Thread1	A[7712] = B [7712] * C[7712]	
	A[7713] = B [7713] * C[7713]	
	
	A[7743] = B [7743] * C[7743]	
SIMD Thread1 5	A[7744] = B [7744] * C[7744]	
	
	A[8159] = B [8159] * C[8159]	
	A[8160] = B [8160] * C[8160]	
	A[8161] = B [8161] * C[8161]	
	
	A[8191] = B [8191] * C[8191]	

Grid

...
A[7679] = B [7679] * C[7679]						

Thread Block 15	SIMD Thread0	A[7680] = B [7680] * C[7680]
		A[7681] = B [7681] * C[7681]
	
		A[7711] = B [7711] * C[7711]
	SIMD Thread1	A[7712] = B [7712] * C[7712]
		A[7713] = B [7713] * C[7713]
	
		A[7743] = B [7743] * C[7743]
	SIMD Thread1 5	A[7744] = B [7744] * C[7744]
	
		A[8159] = B [8159] * C[8159]
		A[8160] = B [8160] * C[8160]
		A[8161] = B [8161] * C[8161]
	
		A[8191] = B [8191] * C[8191]

Vector vs GPU



Vector vs GPU

→ Vector:

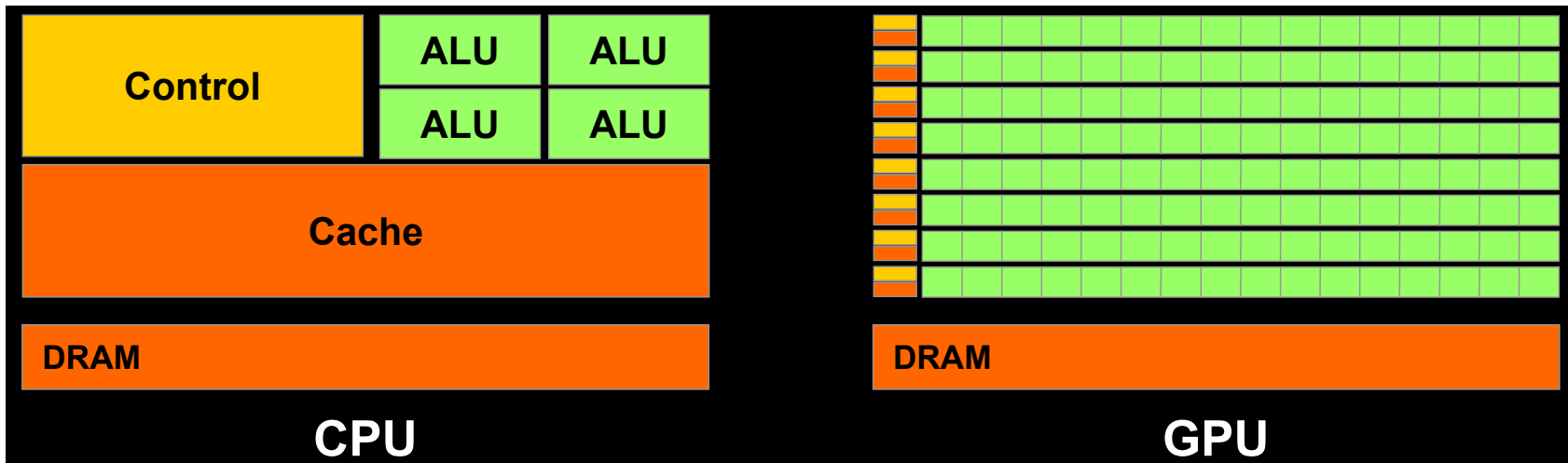
- no multithreading
- hide mem latency by deep pipelining

→ GPU:

- stream multiprocessors --> multithreading
- hide mem latency by multithreading

CPU vs GPU

- CPU: low latency, low throughput
 - Cache, out-of-order execution, speculation, ...
- GPU: high latency, high throughput



CPU vs GPU - Parallelism

→ CPU: task/instruction parallelism

- Multiple tasks map to multiple threads
- tasks run different instructions
- 10s of relatively heavyweight threads run on 1-s of cores
- each thread managed and scheduled explicitly
- each thread individually programmed

→ GPU: data parallelism

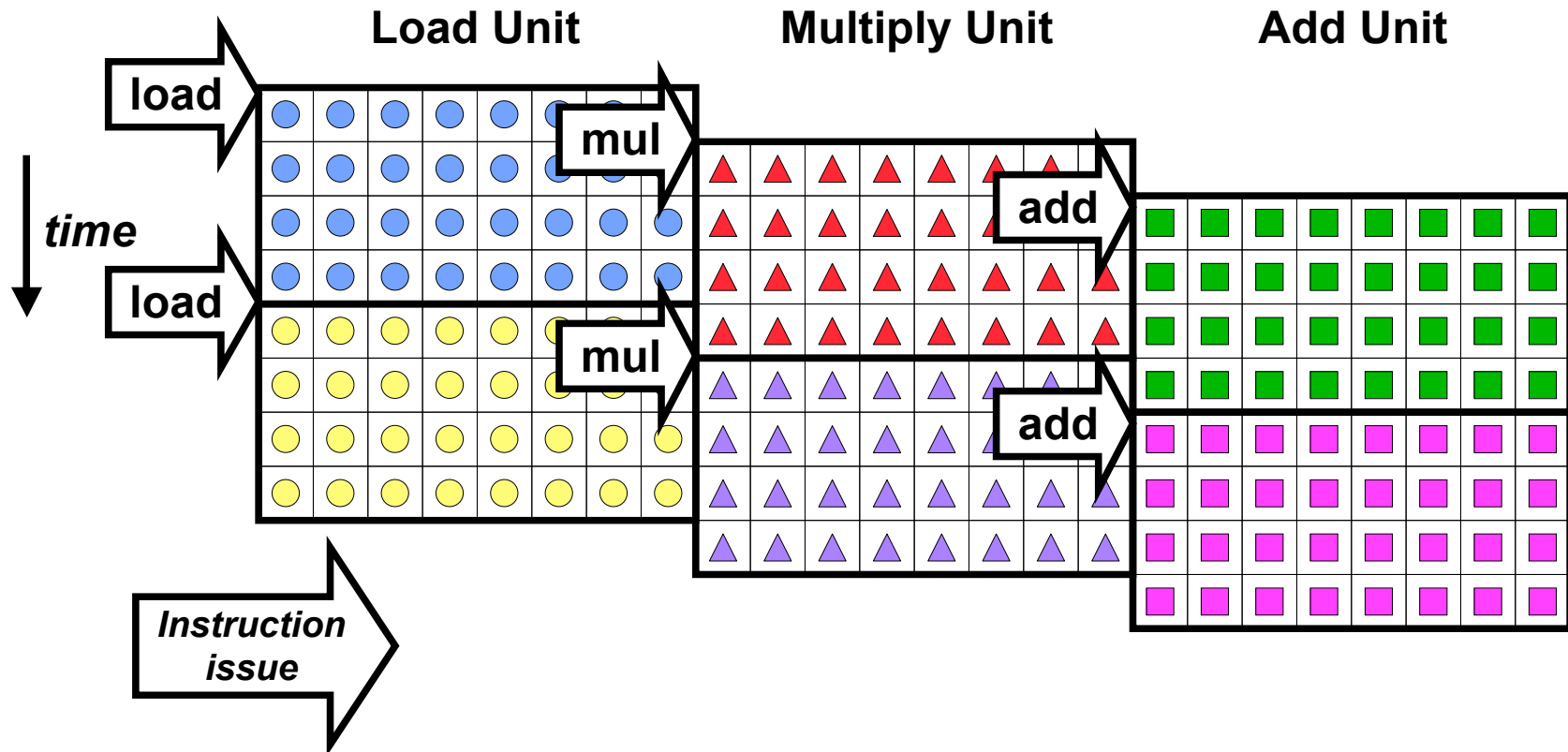
- SIMD execution model
- same instruction operates on different data
- 10,000s of lightweight threads on 100s of cores
- threads managed and scheduled by hardware
- programming done for batches of threads (program on vectors of data)

Backup

Vector Instruction Parallelism

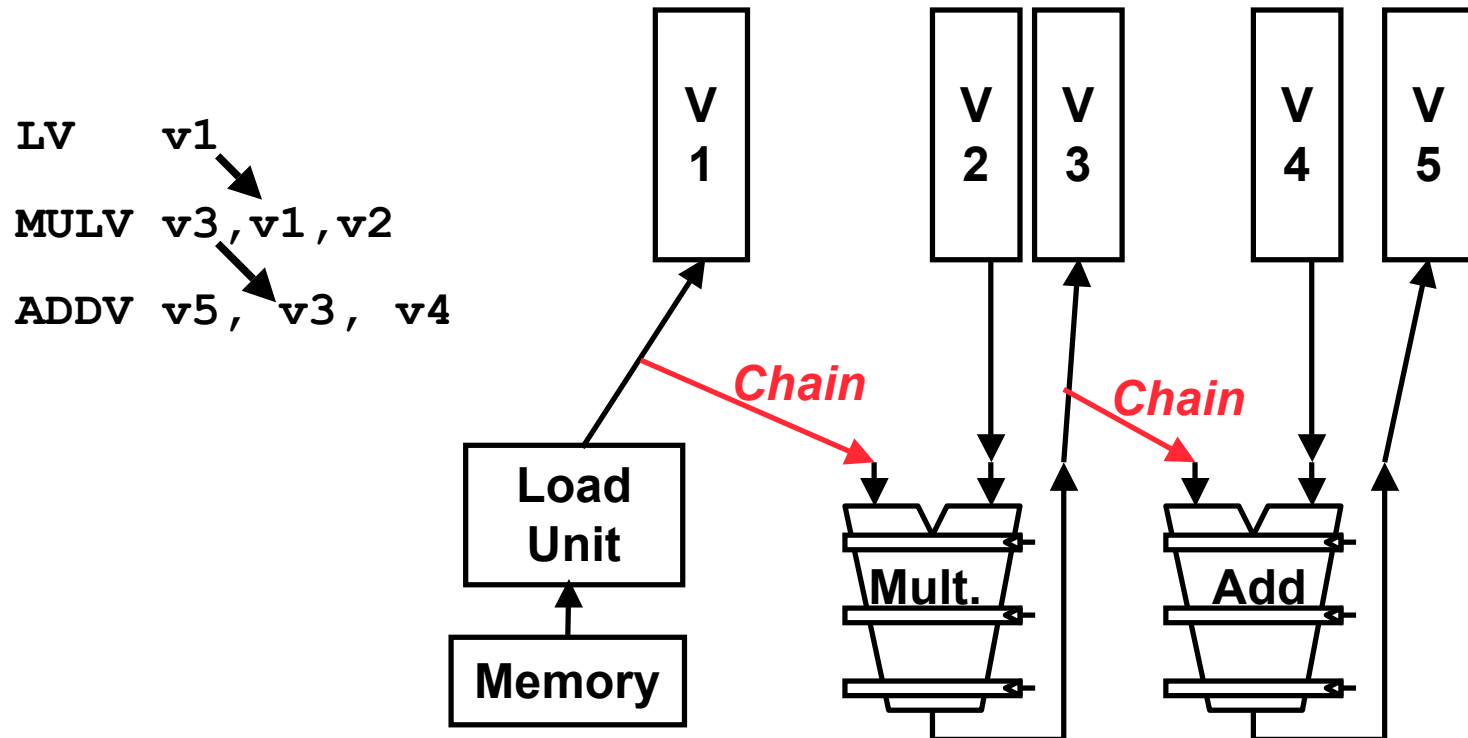
- Can overlap execution of multiple instructions

$$A[i] = B[i] * C[i]$$



Vector Chaining

- Vector version of register bypassing
 - First appeared in Cray-1

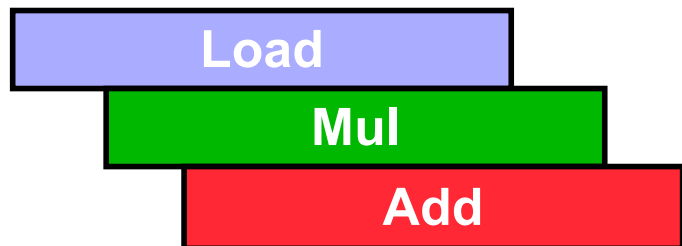


Vector Chaining

- Without chaining, an dependent instruction must wait for the results from a previous instruction to be written into register



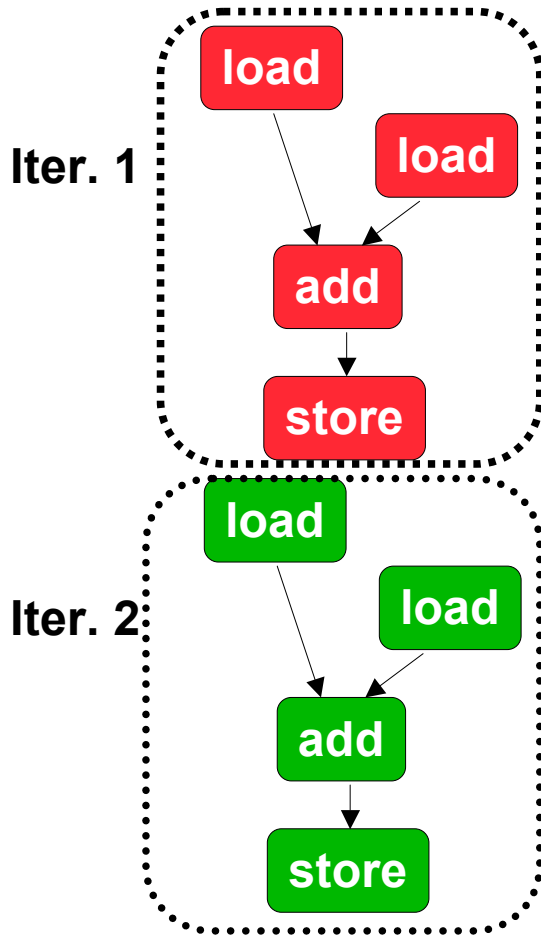
- With chaining, an dependent instruction can start as soon as results from a previous instruction become available



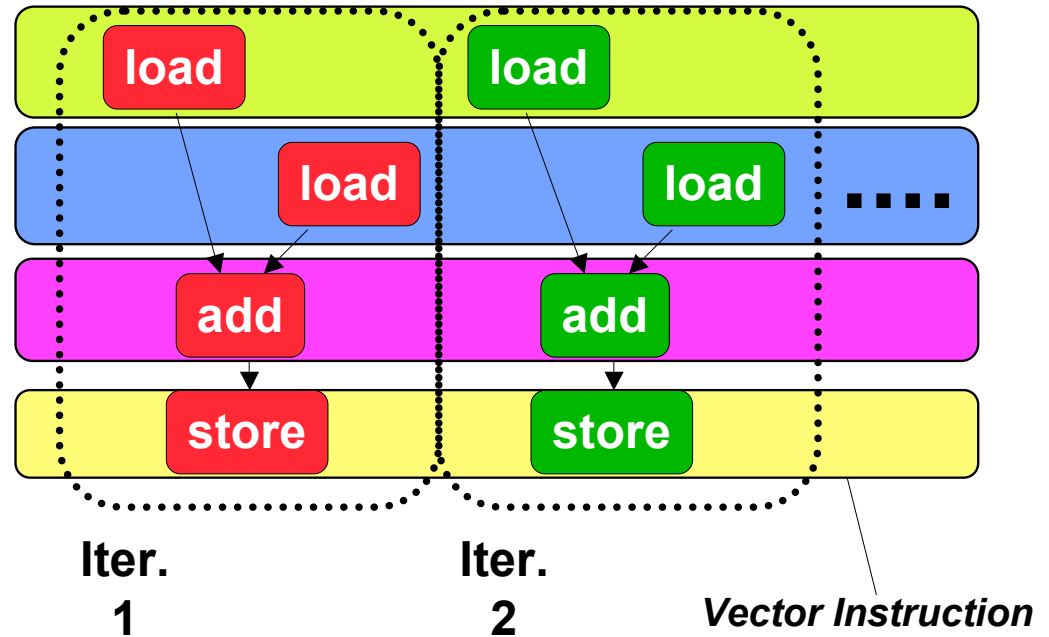
Loop Vectorization

```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```

Scalar Sequential Code



Vectorized Code



**Vectorization is a massive compile-time
reordering of operation sequencing
⇒ requires extensive loop dependence
analysis**