Computer Architecture

Lecture 7: SIMD Processors and GPUs

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2018

10 October 2018

Last Week

- Main Memory and DRAM Fundamentals (Lecture 5)
  - Wrap-up Main Memory Challenges
  - Main Memory Fundamentals
  - DRAM Basics and Operation
  - Memory Controllers
  - Simulation
  - Memory Latency
- Research in DRAM
  - ChargeCache (Lecture 6a)
  - SoftMC (Lecture 6b)
  - REAPER: The Reach Profiler (Lecture 6c)
  - The DRAM Latency PUF (Lecture 6d)

Agenda for This Lecture

- SIMD Processing
  - Vector and Array Processors
- Graphics Processing Units (GPUs)

Exploiting Data Parallelism: SIMD Processors and GPUs

SIMD Processing:
Exploiting Regular (Data) Parallelism

Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- SISD: Single instruction operates on single data element
- SIMD: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- MISD: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

Data Parallelism

- Concurrency arises from performing the same operation 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
- SIMD exploits operation-level parallelism on different data
  - Same operation concurrently applied to different pieces of data
  - A form of ILP where instruction happens to be the same across data

SIMD Processing

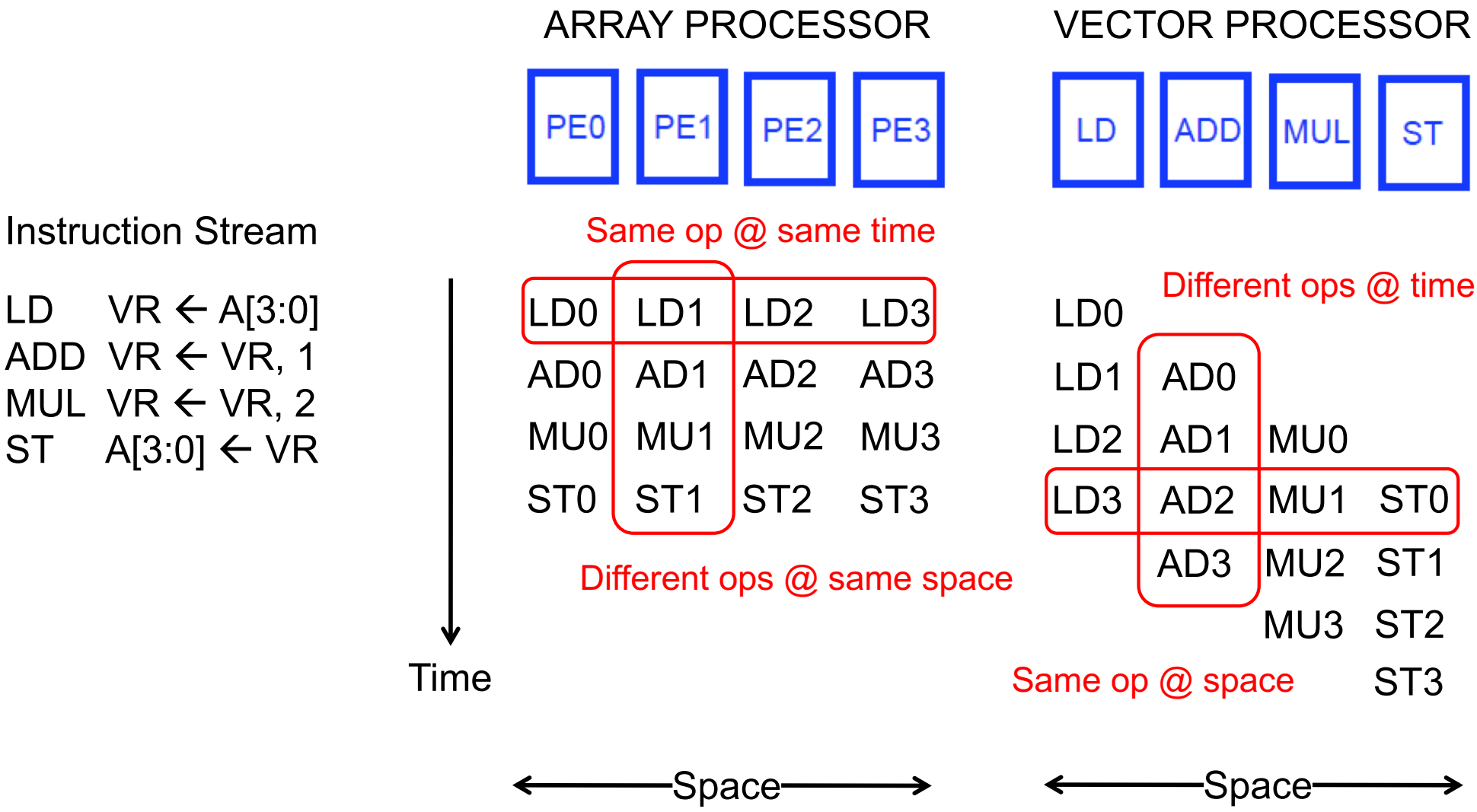
- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements

- Time-space duality

  - Array processor: Instruction operates on multiple data elements at the same time using different spaces

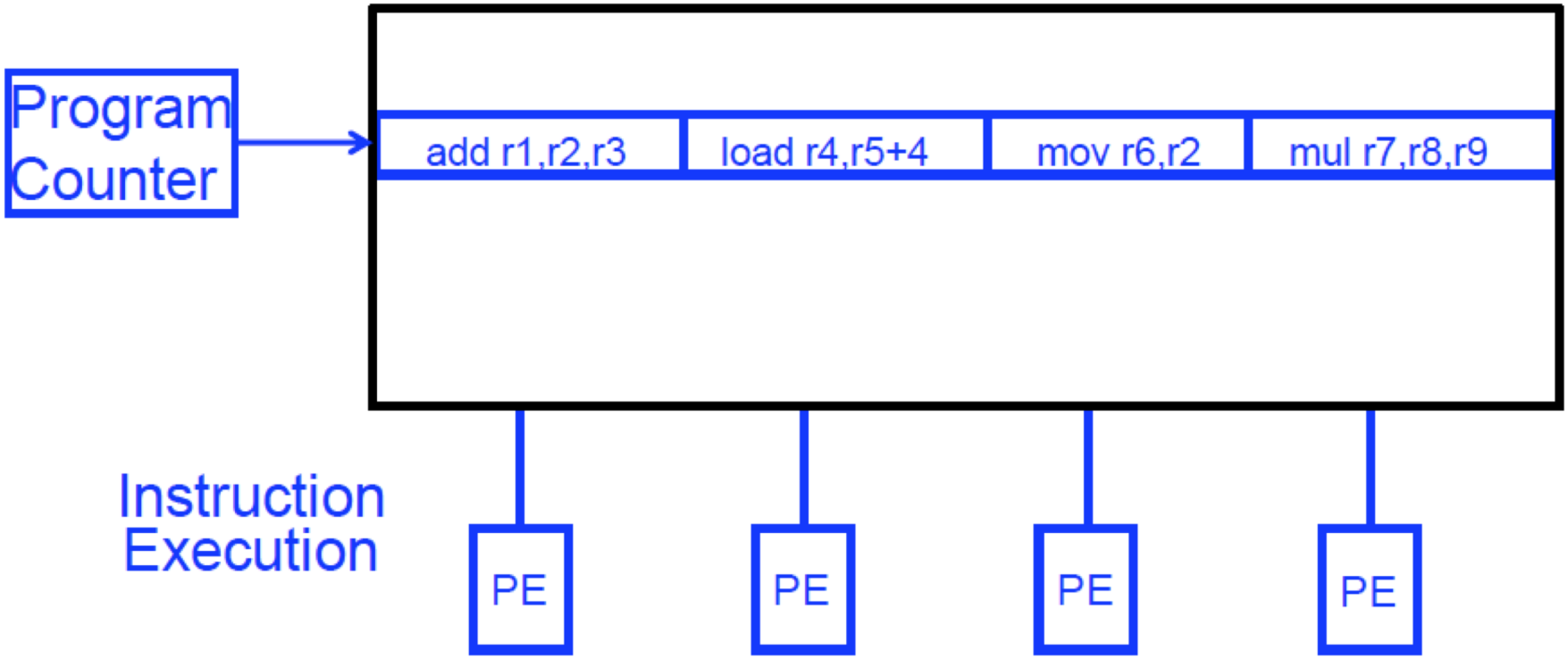
  - Vector processor: Instruction operates on multiple data elements in consecutive time steps using the same space

Array vs. Vector Processors



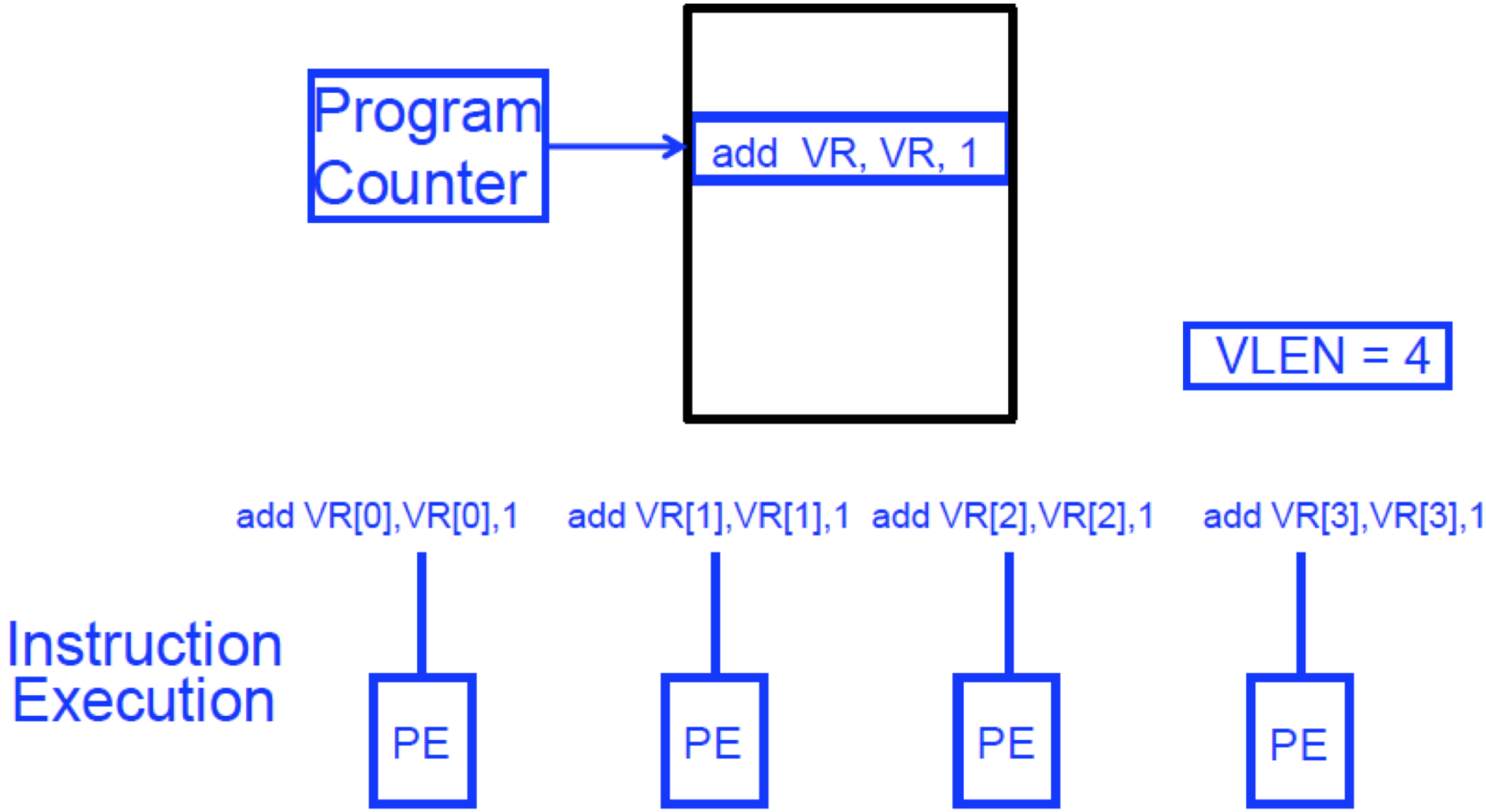
SIMD Array Processing vs. **VLIW**

- VLIW: Multiple independent operations packed together by the compiler



SIMD Array Processing vs. VLIW

- Array processor: Single operation on multiple (different) data elements



Vector Processors (I)

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors

```
for (i = 0; i<=49; i++)
    C[i] = (A[i] + B[i]) / 2
```

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- Basic requirements
  - Need to load/store vectors → vector registers (contain vectors)
  - Need to operate on vectors of different lengths → vector length register (VLEN)
  - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
    - Stride: distance in memory between two elements of a vector

Vector Processors (II)

- A vector instruction performs an operation on each element in consecutive cycles
  - Vector functional units are pipelined
  - Each pipeline stage operates on a different data element

- Vector instructions allow deeper pipelines
  - No intra-vector dependencies → no hardware interlocking needed within a vector
  - No control flow within a vector
  - Known stride allows easy address calculation for all vector elements
    - Enables prefetching of vectors into registers/cache/memory

Vector Processor Advantages

+ No dependencies within a vector
  - Pipelining & parallelization work really well
  - Can have very deep pipelines, no dependencies!

+ Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth requirements

+ Highly regular memory access pattern

+ No need to explicitly code loops
  - Fewer branches in the instruction sequence

Vector Processor Disadvantages

-- Works (only) if parallelism is regular (data/SIMD parallelism)

++ Vector operations

-- Very inefficient if parallelism is irregular

-- How about searching for a key in a linked list?

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

Vector Processor Limitations

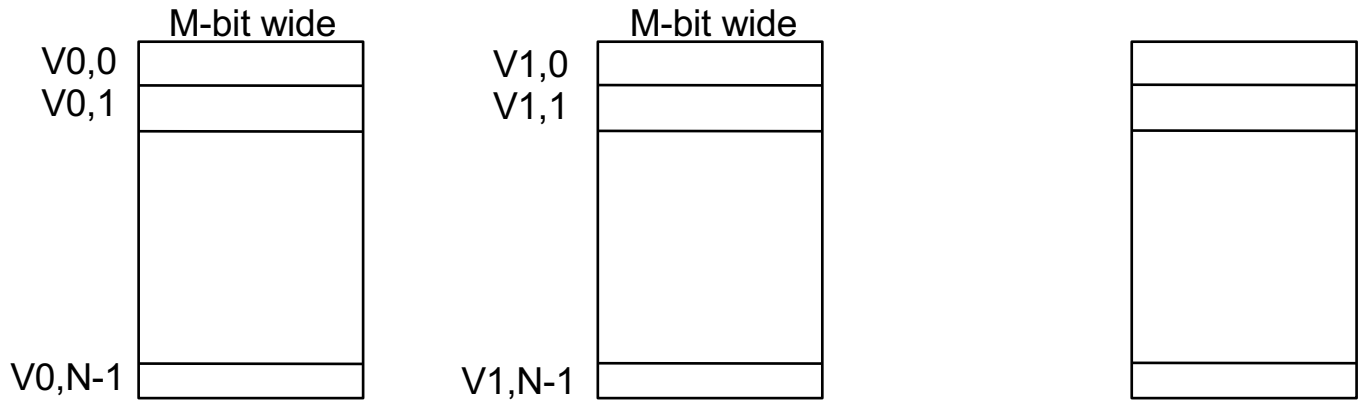
-- Memory (bandwidth) can easily become a bottleneck, especially if

1. compute/memory operation balance is not maintained
2. data is not mapped appropriately to memory banks

Vector Processing in More Depth

Vector Registers

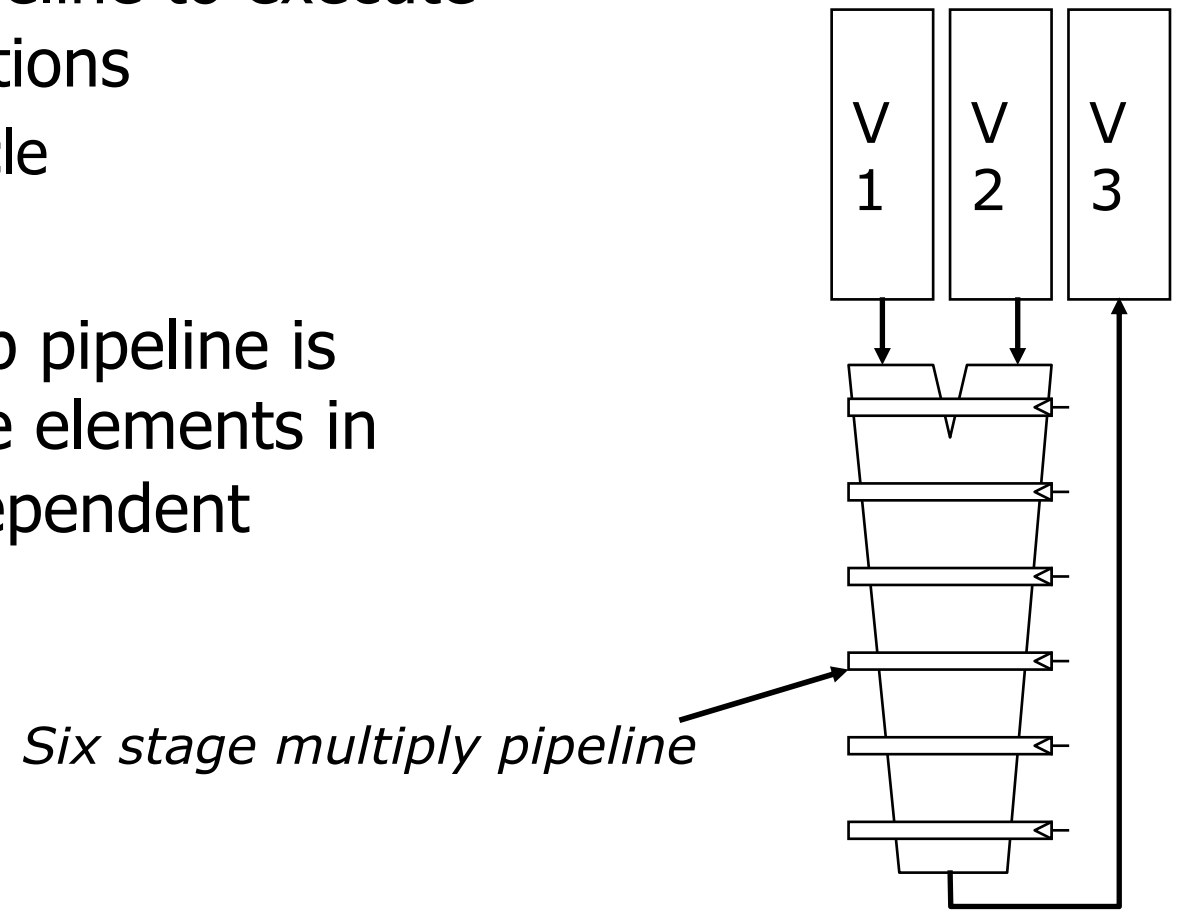
- Each vector data register holds N M-bit values
- Vector control registers: VLEN, VSTR, VMASK
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register
- Vector Mask Register (VMASK)
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g., VMASK[i] = (V_k[i] == 0)



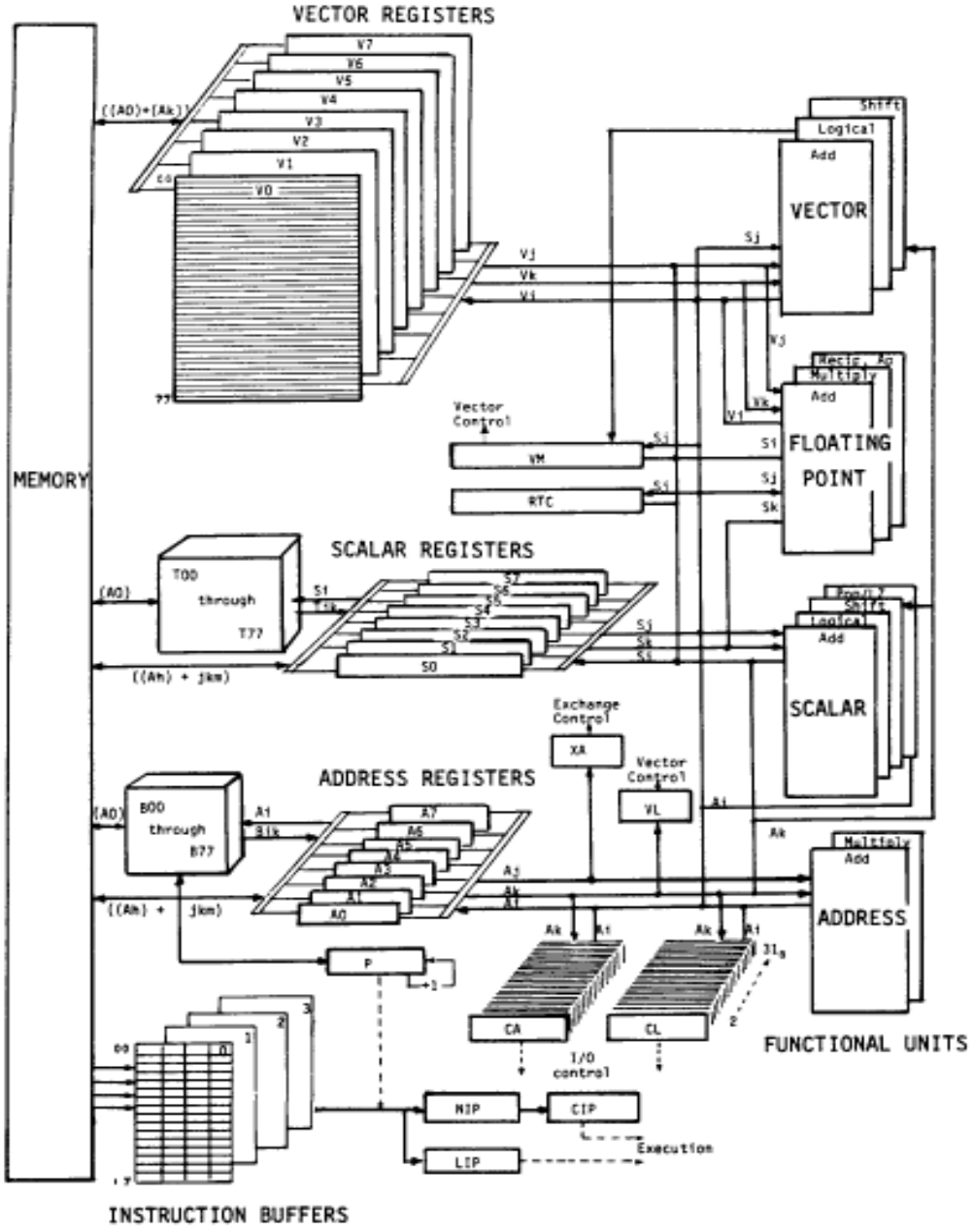
Vector Functional Units

- Use a deep pipeline to execute element operations
  → fast clock cycle

- Control of deep pipeline is simple because elements in vector are independent



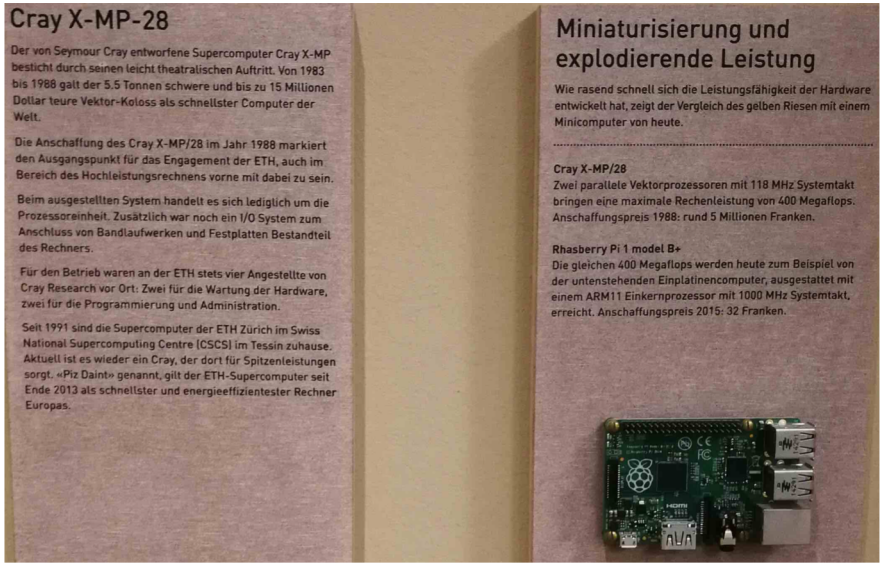
Vector Machine Organization (CRAY-1)



- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.

- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers

CRAY X-MP-28 @ ETH (CAB, E Floor)



Cray X-MP-28

Der von Seymour Cray entworfene Supercomputer Cray X-MP besticht durch seinen leicht theatralischen Auftritt. Von 1983 bis 1988 galt der 5,5 Tonnen schwere und bis zu 15 Millionen Dollar teure Vektor-Koloss als schnellster Computer der Welt.

Die Anschaffung des Cray X-MP/28 im Jahr 1988 markiert den Ausgangspunkt für das Engagement der ETH, auch im Bereich des Hochleistungsrechnens vorne mit dabei zu sein.

Beim ausgestellten System handelt es sich lediglich um die Prozessoreinheit. Zusätzlich war noch ein I/O System zum Anschluss von Bandlaufwerken und Festplatten Bestandteil des Rechners.

Für den Betrieb waren an der ETH stets vier Angestellte von Cray Research vor Ort: Zwei für die Wartung der Hardware, zwei für die Programmierung und Administration.

Seit 1991 sind die Supercomputer der ETH Zürich im Swiss National Supercomputing Centre (CSCS) im Tessin zuhause. Aktuell ist es wieder ein Cray, der dort für Spitzenleistungen sorgt. «Piz Daint» genannt, gilt der ETH-Supercomputer seit Ende 2013 als schnellster und energieeffizientester Rechner Europas.

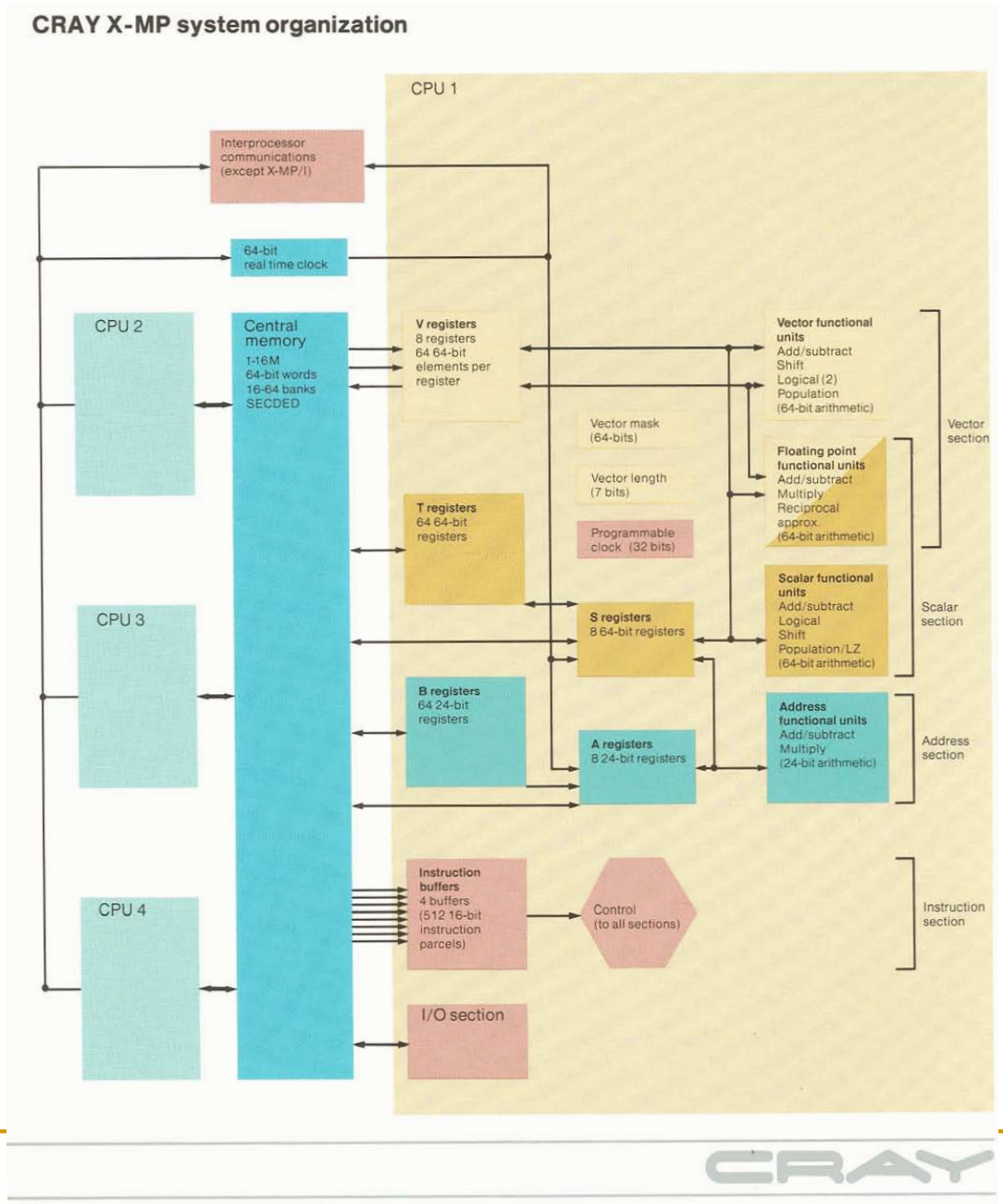
Miniaturisierung und explodierende Leistung

Wie rasend schnell sich die Leistungsfähigkeit der Hardware entwickelt hat, zeigt der Vergleich des gelben Riesen mit einem Minicomputer von heute.

Cray X-MP/28
Zwei parallele Vektorprozessoren mit 118 MHz Systemtakt bringen eine maximale Rechenleistung von 400 Megaflops. Anschaffungspreis 1988: rund 5 Millionen Franken.

Rhasberry Pi 1 model B+
Die gleichen 400 Megaflops werden heute zum Beispiel von der untenstehenden Einplatinencomputer, ausgestattet mit einem ARM11 Einkernprozessor mit 1000 MHz Systemtakt, erreicht. Anschaffungspreis 2015: 32 Franken.

CRAY X-MP System Organization



Cray Research Inc., "The CRAY X-MP Series of Computer Systems," 1985

CRAY X-MP Design Detail

CRAY X-MP design detail

Mainframe

CRAY X-MP single- and multiprocessor systems are designed to offer users outstanding performance on large-scale, compute-intensive and I/O-bound jobs.

CRAY X-MP mainframes consist of six (X-MP/1), eight (X-MP/2) or twelve (X-MP/4) vertical columns arranged in an arc. Power supplies and cooling are clustered around the base and extend outward.

Model	Number of CPUs	Memory size (millions of 64-bit words)	Number of banks
CRAY X-MP/416	4	16	64
CRAY X-MP/48	4	8	32
CRAY X-MP/216	2	16	32
CRAY X-MP/28	2	8	32
CRAY X-MP/24	2	4	16
CRAY X-MP/18	1	8	32
CRAY X-MP/14	1	4	16
CRAY X-MP/12	1	2	16
CRAY X-MP/11	1	1	16

Hardware features:

- 9.5 nsec clock
- One, two or four CPUs, each with its own computation and control sections
- Large multiport central memory
- Memory bank cycle time of 38 nsec on X-MP/4 systems, 76 nsec on X-MP/1 and X-MP/2 models
- Memory bandwidth of 25-100 gigabits, depending on model
- I/O section
- Proven cooling and packaging technologies

A description of the major system components and their functions follows.

CPU computation section

Within the computation section of each CPU are operating registers, functional units and an instruction control network — hardware elements that cooperate in executing sequences of instructions. The instruction control network makes all decisions related to instruction issue as well as coordinating the three types of processing within each CPU: vector, scalar and address. Each of the processing modes has its associated registers and functional units.

The block diagram of a CRAY X-MP/4 (opposite page) illustrates the relationship of the registers to the functional units, instruction buffers, I/O channel control registers, interprocessor communications section and memory. For multiple-processor CRAY X-MP models, the interprocessor communications section coordinates processing between CPUs, and central memory is shared.

Registers

The basic set of programmable registers is composed of:

Eight 24-bit address (A) registers
Sixty-four 24-bit intermediate address (B) registers
Eight 64-bit scalar (S) registers
Sixty-four 64-bit scalar-save (T) registers
Eight 64-element (4096-bit) vector (V) registers with 64 bits per element

The 24-bit A registers are generally used for addressing and counting operations. Associated with them are 64 B registers, also 24 bits wide. Since the transfer between an A and a B register takes only one clock period, the B registers assume the role of data cache, storing information for fast access without tying up the A registers for relatively long periods.

Cray Research Inc., "The CRAY X-MP Series of Computer Systems," 1985

CRAY X-MP CPU Functional Units

CRAY X-MP CPU functional units

	Register usage	Time in clock periods
Address functional units		
Addition	A	2
Multiplication	A	4
Scalar functional units		
Addition	S	3
Shift-single	S	2
Shift-double	S	3
Logical	S	1
Population, parity and leading zero	S	3 or 4
Vector functional units		
Addition	V	3
Shift	V	3 or 4
Full vector logical	V	2
Second vector logical	V	4
Population, parity	V	5
Floating-point functional units		
Addition	S and V	6
Multiplication	S and V	7
Reciprocal approximation	S and V	14

Cray Research Inc., "The CRAY X-MP Series of Computer Systems," 1985

CRAY X-MP System Configuration

System configuration options

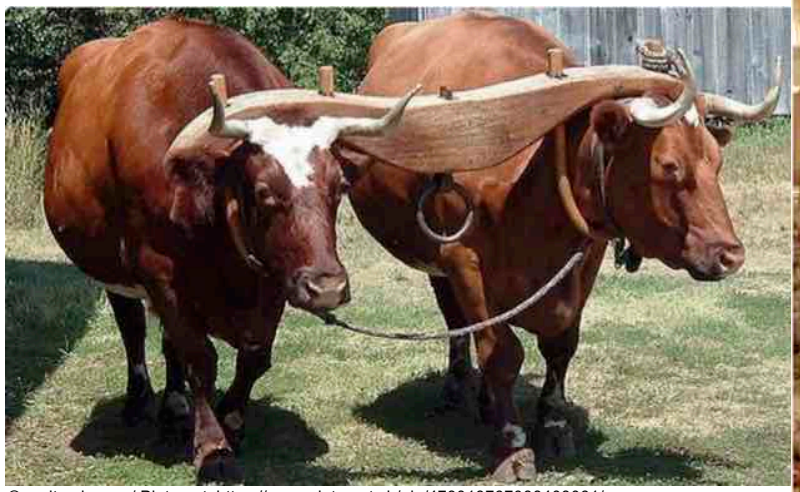
	X-MP/1	X-MP/2	X-MP/4
Mainframe			
CPUs	1	2	4
Bipolar memory (64-bit words)	N/A	N/A	8 or 16M
MOS memory (64-bit words)	1, 2, 4 or 8M	4, 8 or 16M	N/A
6-Mbyte channels	2 or 4	4	4
100-Mbyte channels	1 or 2	2	4
1000-Mbyte channels	1	1	2
I/O Subsystem			
I/O processors	2, 3 or 4	2, 3 or 4	4
Disk storage units	2-32	2-32	2-32
Magnetic tape channels	1-8	1-8	1-8
Front-end interfaces	1-7	1-7	1-7
Buffer memory (Mbytes)	8, 32 or 64	8, 32 or 64	64
Solid-state Storage Device			
Memory size (Mbytes)	256, 512 or 1024	256, 512 or 1024	256, 512 or 1024

N/A signifies option is not available on the model

Cray Research Inc., "The CRAY X-MP Series of Computer Systems," 1985

Seymour Cray, the Father of Supercomputers

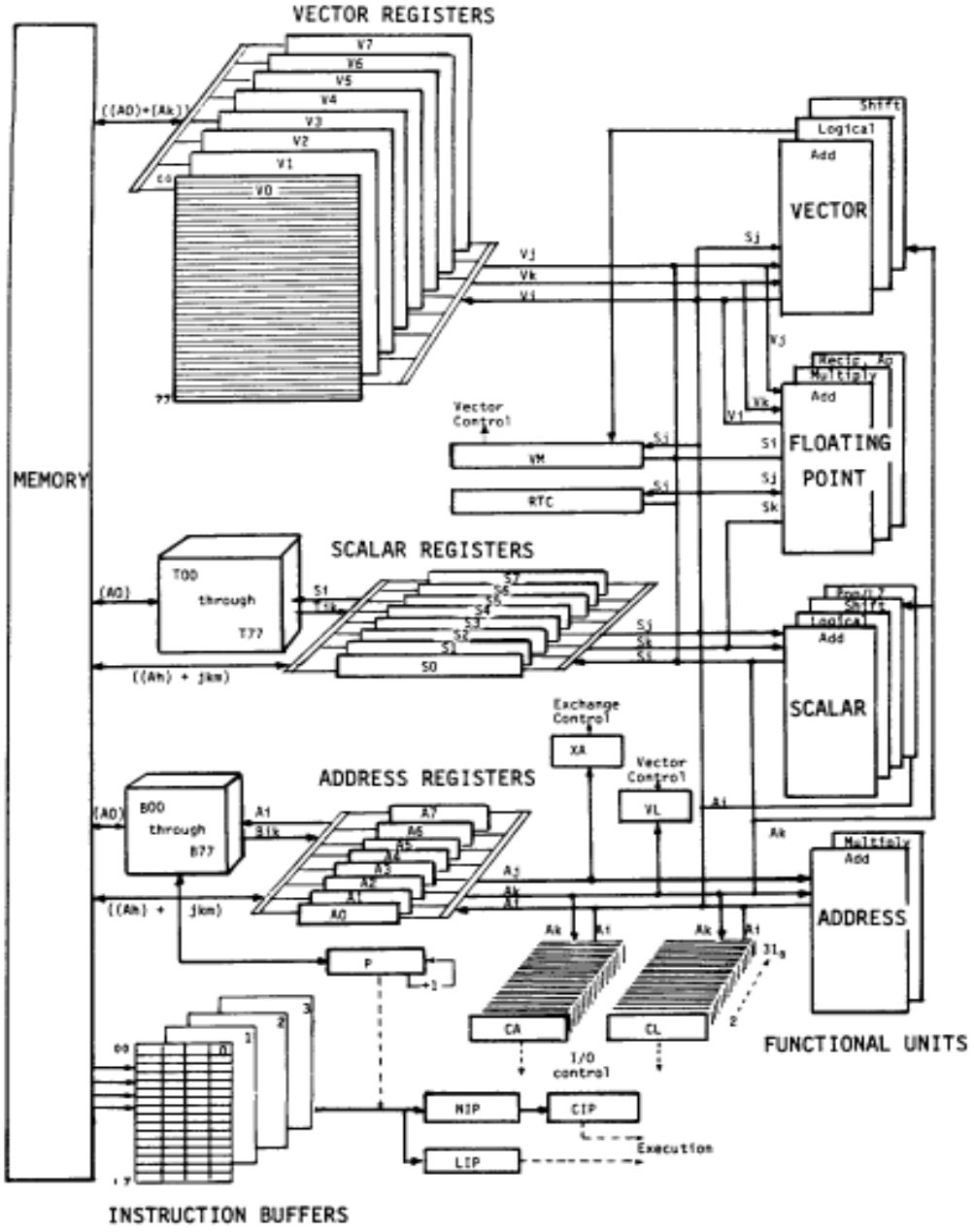
"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"



© amityrebecca / Pinterest. https://www.pinterest.ch/pin/473018767088408061/

© Scott Sinklier / Corbis. http://america.aljazeera.com/articles/2015/2/20/the-short-brutal-life-of-male-chickens.html

Vector Machine Organization (CRAY-1)



- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.

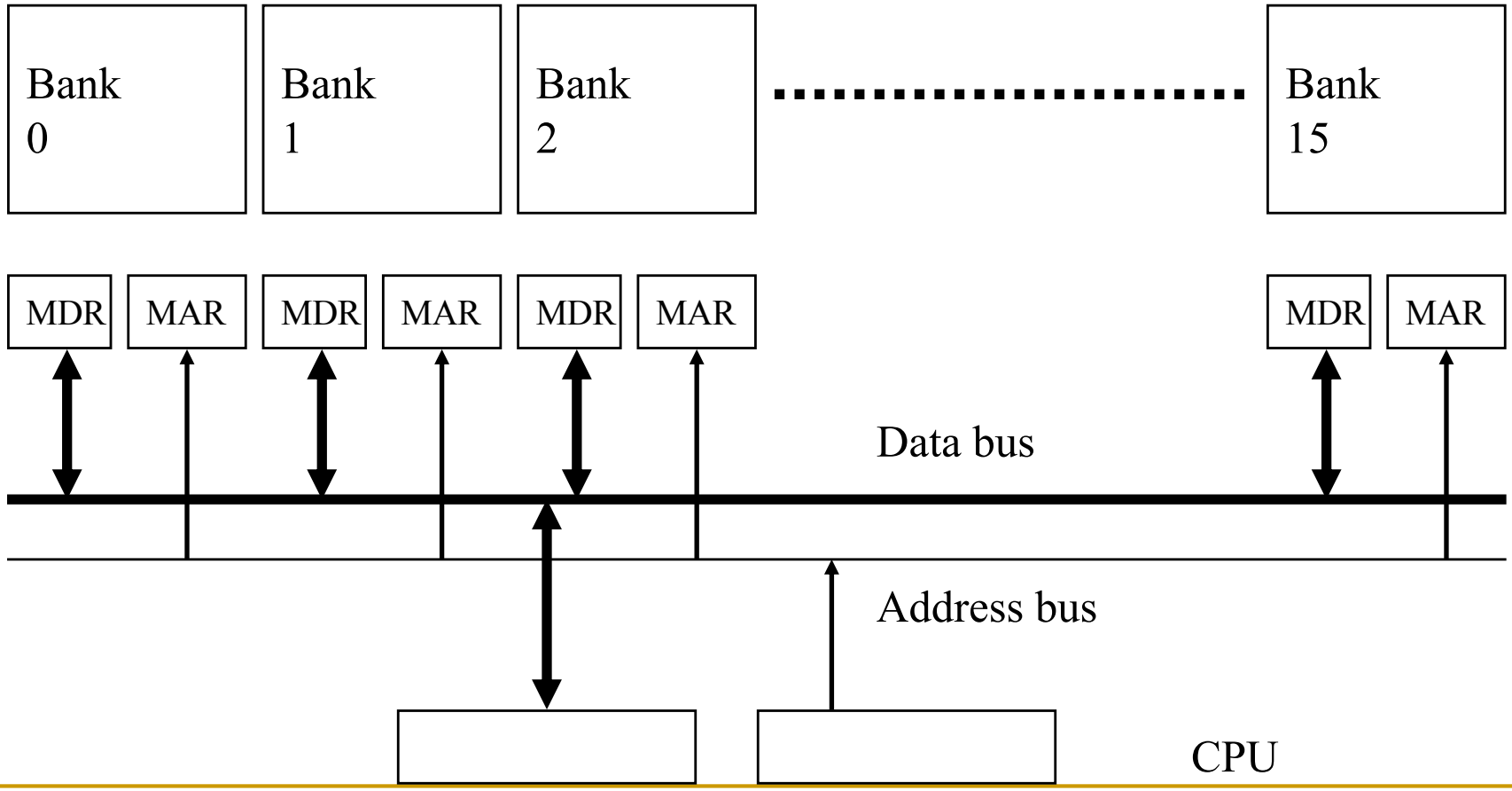
- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers

Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements
- Elements separated from each other by a constant distance (stride)
  - Assume stride = 1 for now
- Elements can be loaded in consecutive cycles if we can start the load of one element per cycle
  - Can sustain a throughput of one element per cycle
- Question: How do we achieve this with a memory that takes more than 1 cycle to access?
- Answer: Bank the memory; interleave the elements across banks

Memory Banking

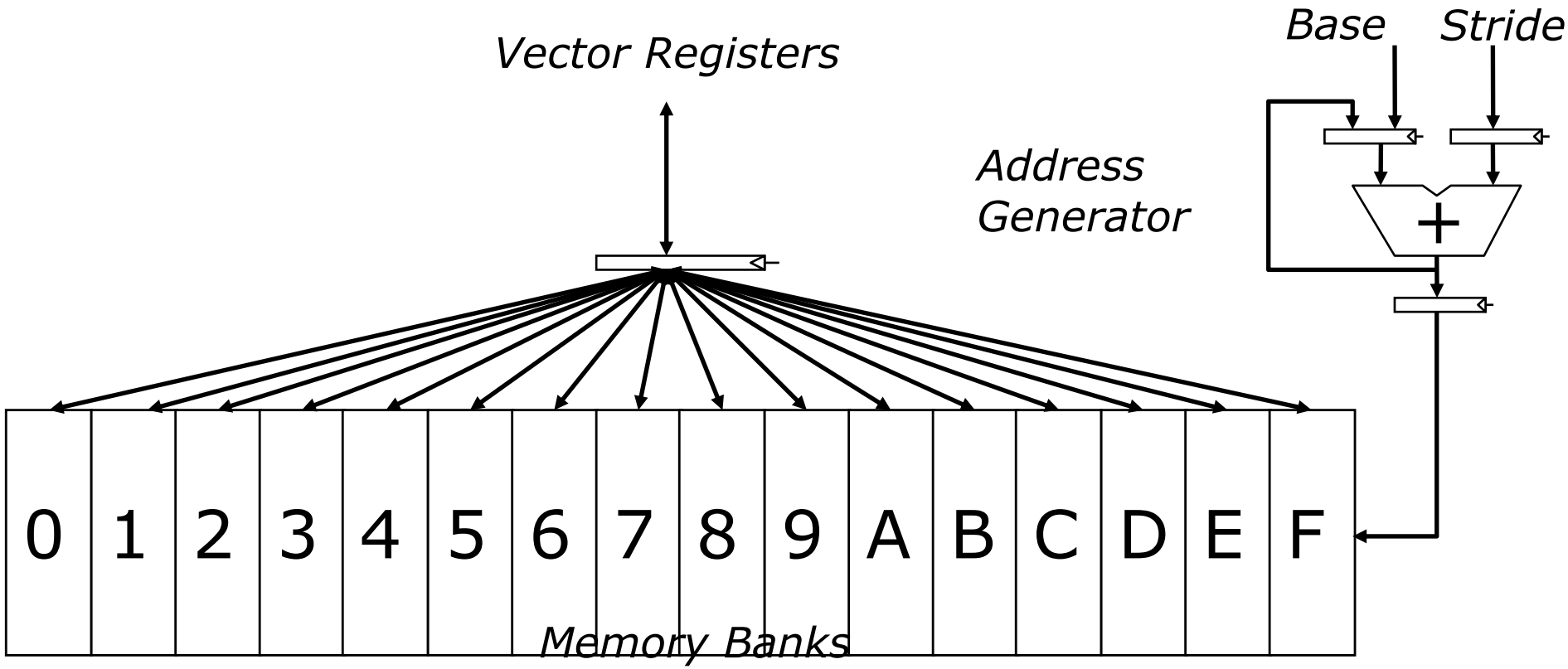
- Memory is divided into banks that can be accessed independently; banks share address and data buses (to minimize pin cost)
- Can start and complete one bank access per cycle
- Can sustain N parallel accesses if all N go to different banks



Picture credit: Derek Chiou

Vector Memory System

- Next address = Previous address + Stride
- If (stride == 1) && (consecutive elements interleaved across banks) && (number of banks >= bank latency), then
  - we can sustain 1 element/cycle throughput



Picture credit: Krste Asanovic

Scalar Code Example: Element-Wise Avg.

- For I = 0 to 49
  - C[i] = (A[i] + B[i]) / 2

- Scalar code (instruction and its latency)

```
    MOVI R0 = 50                1
    MOVA R1 = A                 1
    MOVA R2 = B                 1
    MOVA R3 = C                 1
X:  LD R4 = MEM[R1++]           11  ;autoincrement addressing
    LD R5 = MEM[R2++]           11
    ADD R6 = R4 + R5            4
    SHFR R7 = R6 >> 1           1
    ST MEM[R3++] = R7           11
    DECBNZ R0, X                2   ;decrement and branch if NZ
```

304 dynamic instructions

Scalar Code Execution Time (In Order)

- Scalar execution time on an in-order processor with 1 bank
  - First two loads in the loop cannot be pipelined: 2*11 cycles
  - 4 + 50*40 = 2004 cycles

- Scalar execution time on an in-order processor with 16 banks (word-interleaved: consecutive words are stored in consecutive banks)
  - First two loads in the loop can be pipelined
  - 4 + 50*30 = 1504 cycles

- Why 16 banks?
  - 11-cycle memory access latency
  - Having 16 (>11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency

Vectorizable Loops

- A loop is vectorizable if each iteration is independent of any other

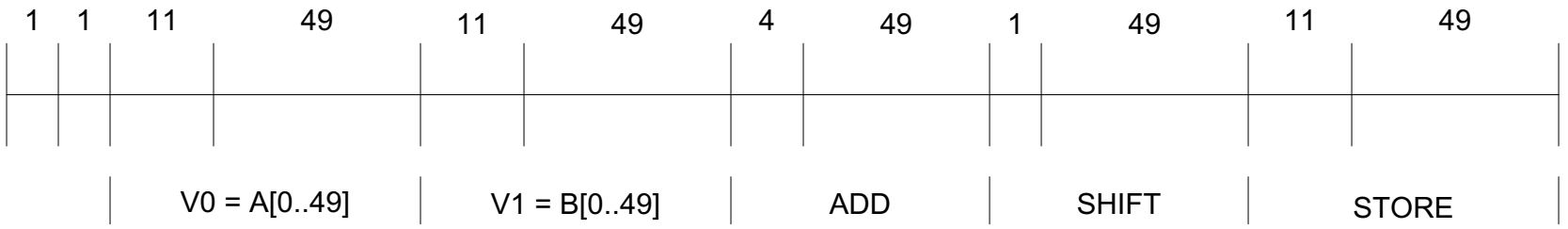
- For I = 0 to 49
  - C[i] = (A[i] + B[i]) / 2
- Vectorized loop (each instruction and its latency):

MOVI VLEN = 50	1
MOVI VSTR = 1	1
VLD V0 = A	11 + VLEN – 1
VLD V1 = B	11 + VLEN – 1
VADD V2 = V0 + V1	4 + VLEN – 1
VSHFR V3 = V2 >> 1	1 + VLEN – 1
VST C = V3	11 + VLEN – 1

7 dynamic instructions

Basic Vector Code Performance

- Assume no chaining (no vector data forwarding)
  - i.e., output of a vector functional unit cannot be used as the direct input of another
  - The entire vector register needs to be ready before any element of it can be used as part of another operation
- One memory port (one address generator)
- 16 memory banks (word-interleaved)

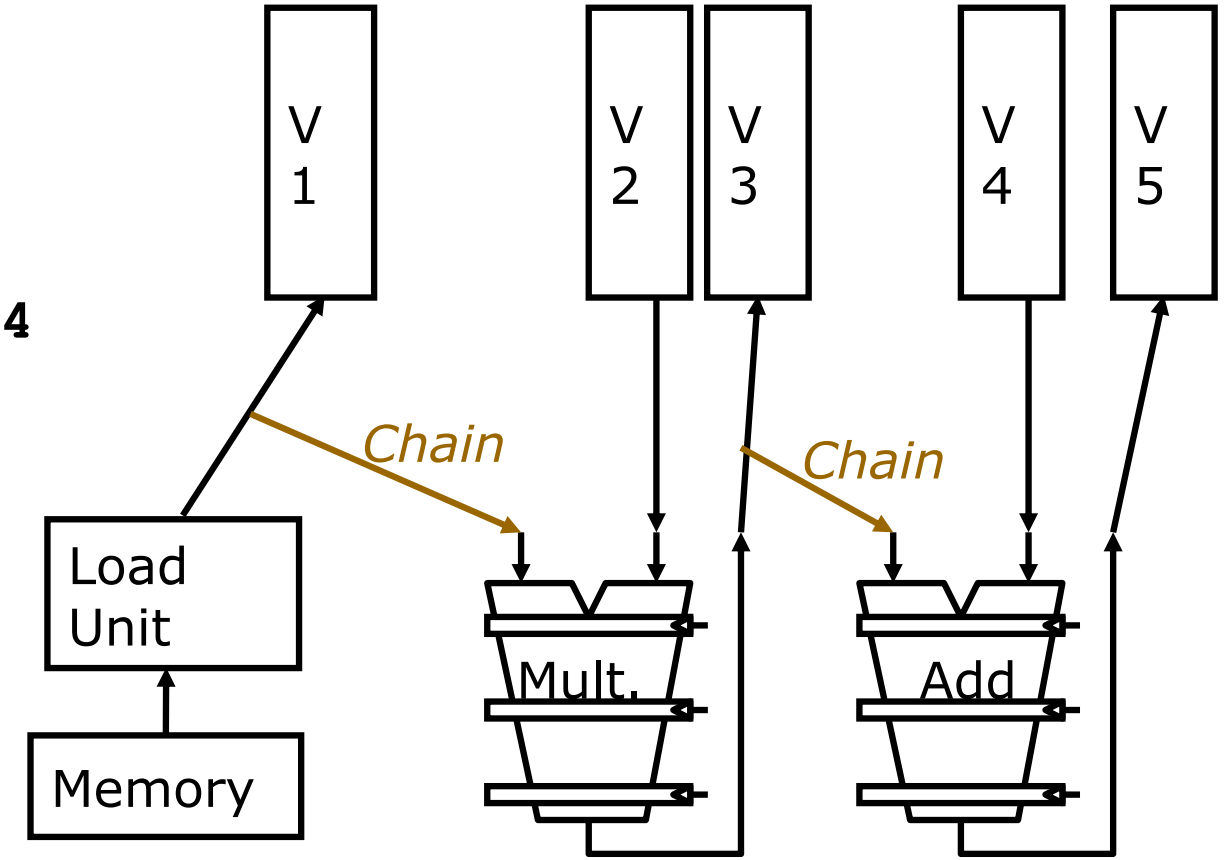


- 285 cycles

Vector Chaining

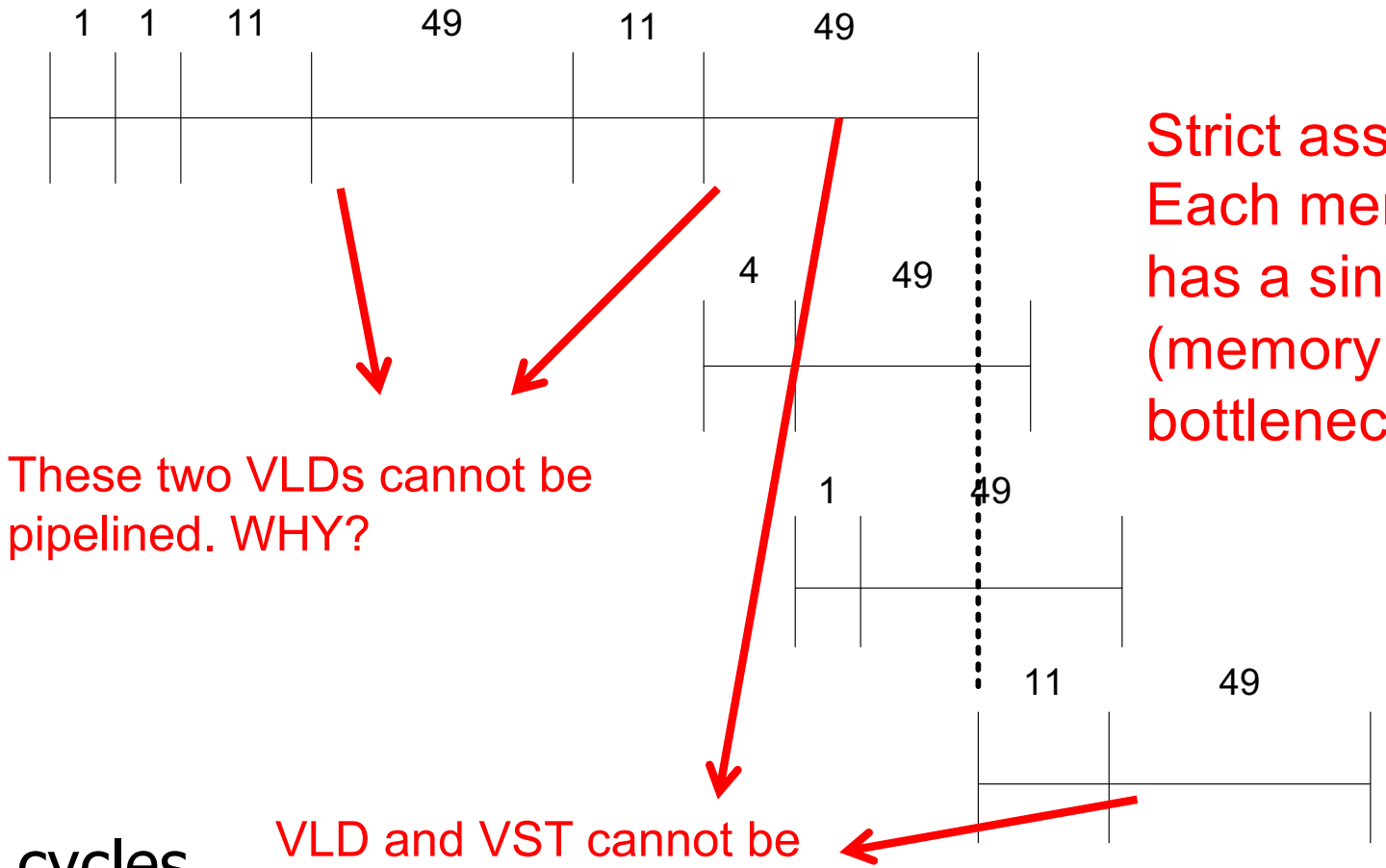
- Vector chaining: Data forwarding from one vector functional unit to another

```
LV   v1
MULV v3,v1,v2
ADDV v5,v3, v4
```



Vector Code Performance - Chaining

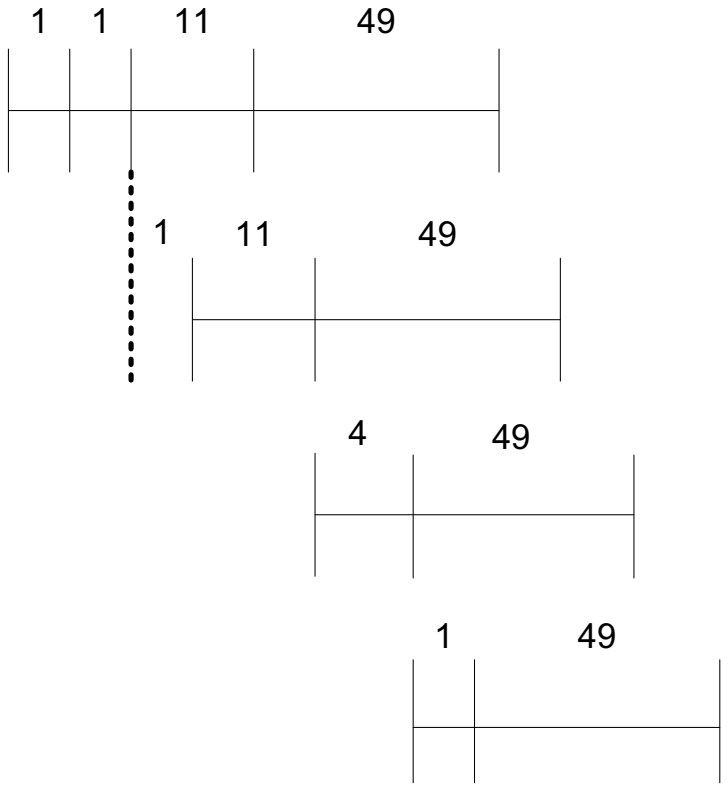
- Vector chaining: Data forwarding from one vector functional unit to another



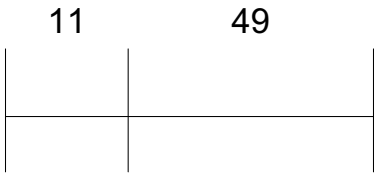
- 182 cycles

Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank



- 79 cycles
- 19X perf. improvement!



Questions (I)

- What if # data elements > # elements in a vector register?
  - Idea: Break loops so that each iteration operates on # elements in a vector register
    - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
  - Called vector stripmining

(Vector) Stripmining

Surface mining, including **strip mining**, open-pit mining and mountaintop removal mining, is a broad category of mining in which soil and rock overlying the mineral deposit (the overburden) are removed, in contrast to underground mining, in which the overlying rock is left in place, and the mineral removed through shafts or tunnels.

Surface mining began in the mid-sixteenth century[1] and is practiced throughout the world, although the majority of surface coal mining occurs in North America.[2] It gained

Coal strip mine in Wyoming

Source: https://en.wikipedia.org/wiki/Surface_mining

Recall: Questions (II)

- What if vector data is not stored in a strided fashion in memory? (irregular memory access to a vector)
  - Idea: Use indirection to combine/pack elements into vector registers
  - Called scatter/gather operations

Gather/Scatter Operations

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```
LV vD, rD          # Load indices in D vector
LVI vC, rC, vD     # Load indirect from rC base
LV vB, rB          # Load B vector
ADDV.D vA,vB,vC    # Do add
SV vA, rA          # Store result
```

Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle sparse vectors (matrices)
- Vector loads and stores use an index vector which is added to the base register to generate the addresses

- *Scatter* example

Index Vector	Data Vector (to Store)	Stored Vector (in Memory)	
0	3.14	Base+0	3.14
2	6.5	Base+1	X
6	71.2	Base+2	6.5
7	2.71	Base+3	X
		Base+4	X
		Base+5	X
		Base+6	71.2
		Base+7	2.71

Conditional Operations in a Loop

- What if some operations should not be executed on a vector (based on a dynamically-determined condition)?

```
loop:       for (i=0; i<N; i++)
                if (a[i] != 0) then b[i]=a[i]*b[i]
```

- Idea: Masked operations
  - VMASK register is a bit mask determining which data element should not be acted upon

```
VLD V0 = A
VLD V1 = B
VMASK = (V0 != 0)
VMUL V1 = V0 * V1
VST B = V1
```

  - This is predicated execution. Execution is *predicated* on mask bit.

Another Example with Masking

```
for (i = 0; i < 64; ++i)
    if (a[i] >= b[i])
        c[i] = a[i]
    else
        c[i] = b[i]
```

A	B	VMASK
1	2	0
2	2	1
3	2	1
4	10	0
-5	-4	0
0	-3	1
6	5	1
-7	-8	1

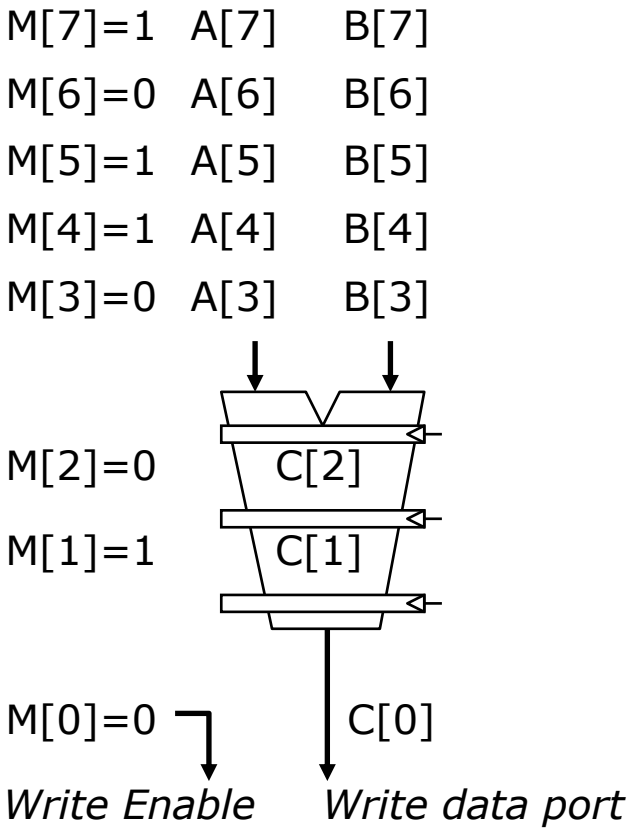
Steps to execute the loop in SIMD code

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

Masked Vector Instructions

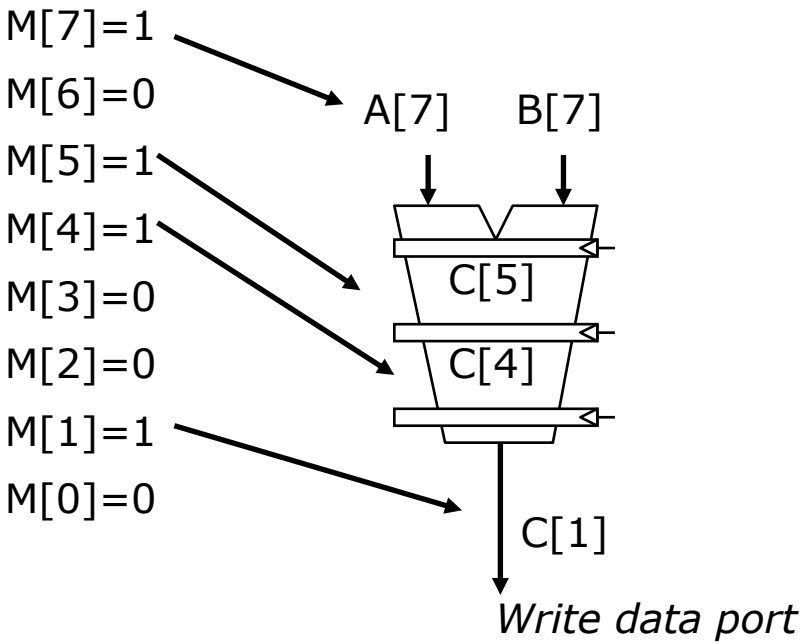
Simple Implementation

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



Density-Time Implementation

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



Which one is better?

Tradeoffs?

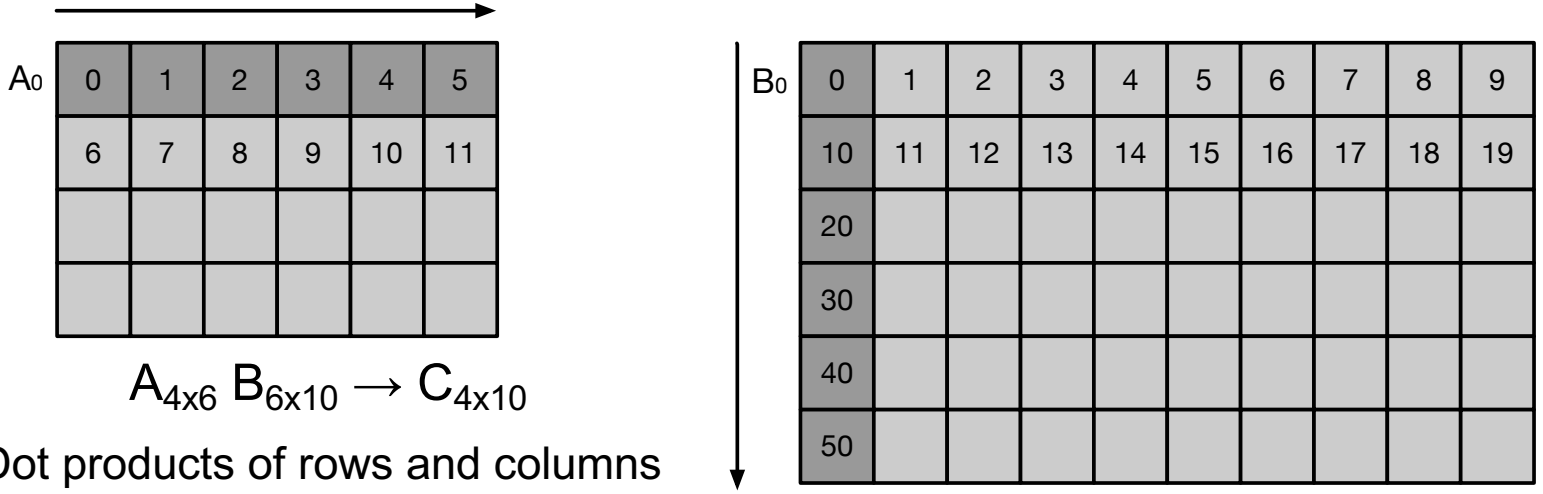
Some Issues

- Stride and banking
  - As long as they are *relatively prime* to each other and there are enough banks to cover bank access latency, we can sustain 1 element/cycle throughput

- Storage of a matrix
  - Row major: Consecutive elements in a row are laid out consecutively in memory
  - Column major: Consecutive elements in a column are laid out consecutively in memory
  - You need to change the stride when accessing a row versus column

Matrix Multiplication

- A and B, both in row-major order



$A_{4x6}\ B_{6x10} \rightarrow C_{4x10}$

Dot products of rows and columns of A and B

- A: Load A_0 into vector register V_1
  - Each time, increment address by one to access the next column
  - Accesses have a stride of 1
- B: Load B_0 into vector register V_2
  - Each time, increment address by 10
  - Accesses have a stride of 10

Different strides can lead to bank conflicts

How do we minimize them?

Minimizing Bank Conflicts

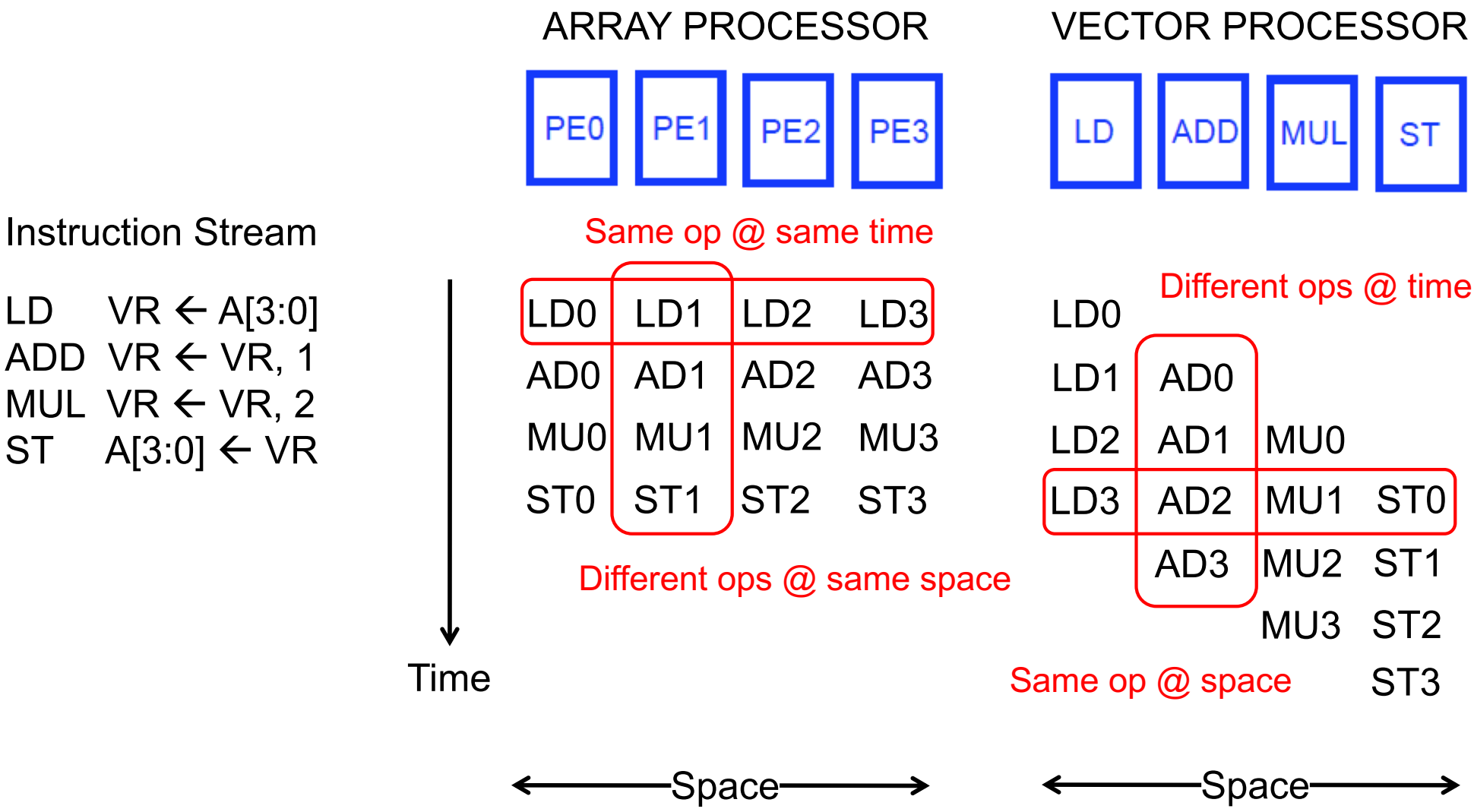
- More banks
- Better data layout to match the access pattern
  - Is this always possible?
- Better mapping of address to bank
  - E.g., randomized mapping
  - Rau, "Pseudo-randomly interleaved memory," ISCA 1991.

Array vs. Vector Processors, Revisited

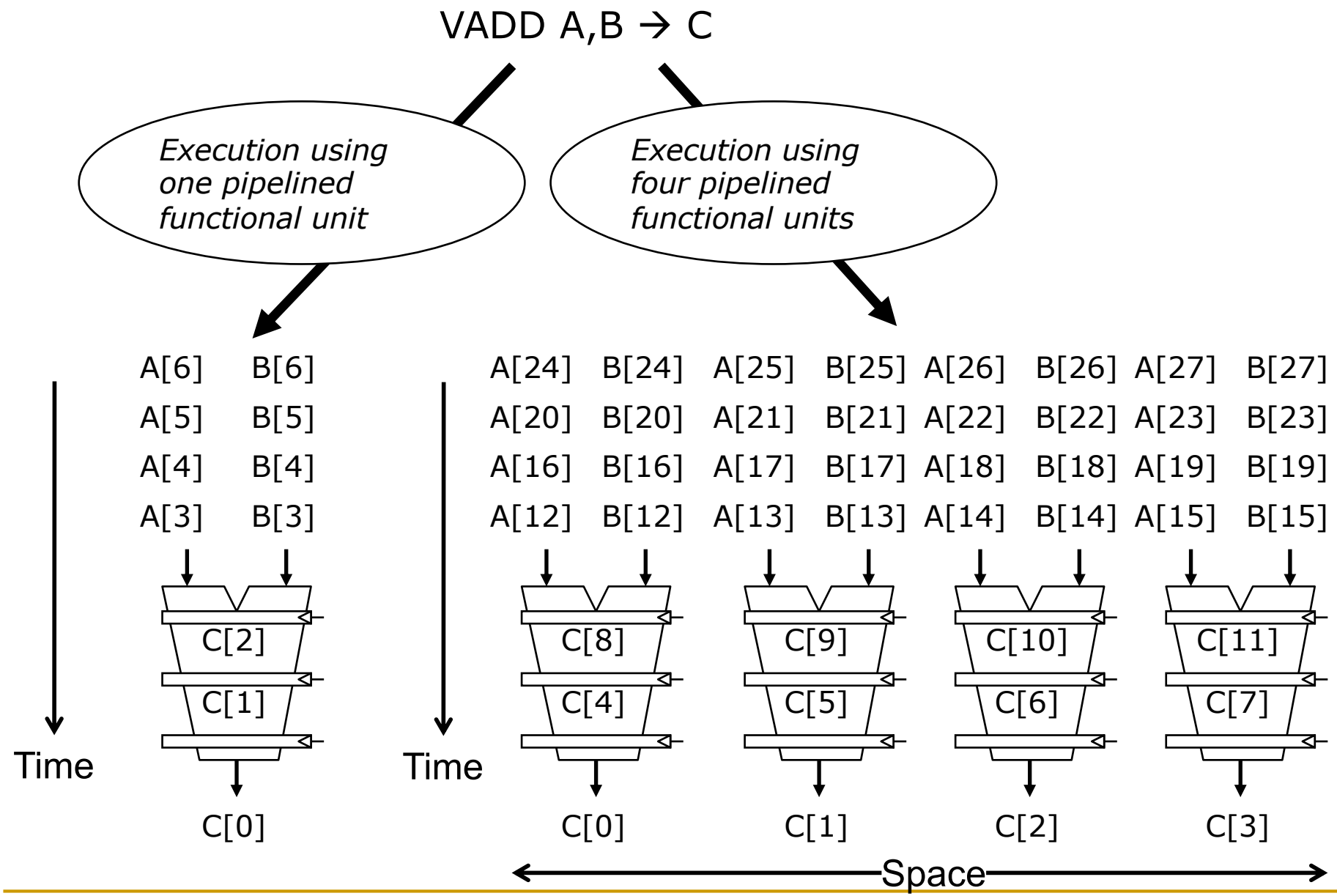
- Array vs. vector processor distinction is a "purist's" distinction

- Most "modern" SIMD processors are a combination of both
  - They exploit data parallelism in both time and space
  - GPUs are a prime example we will cover in a bit more detail

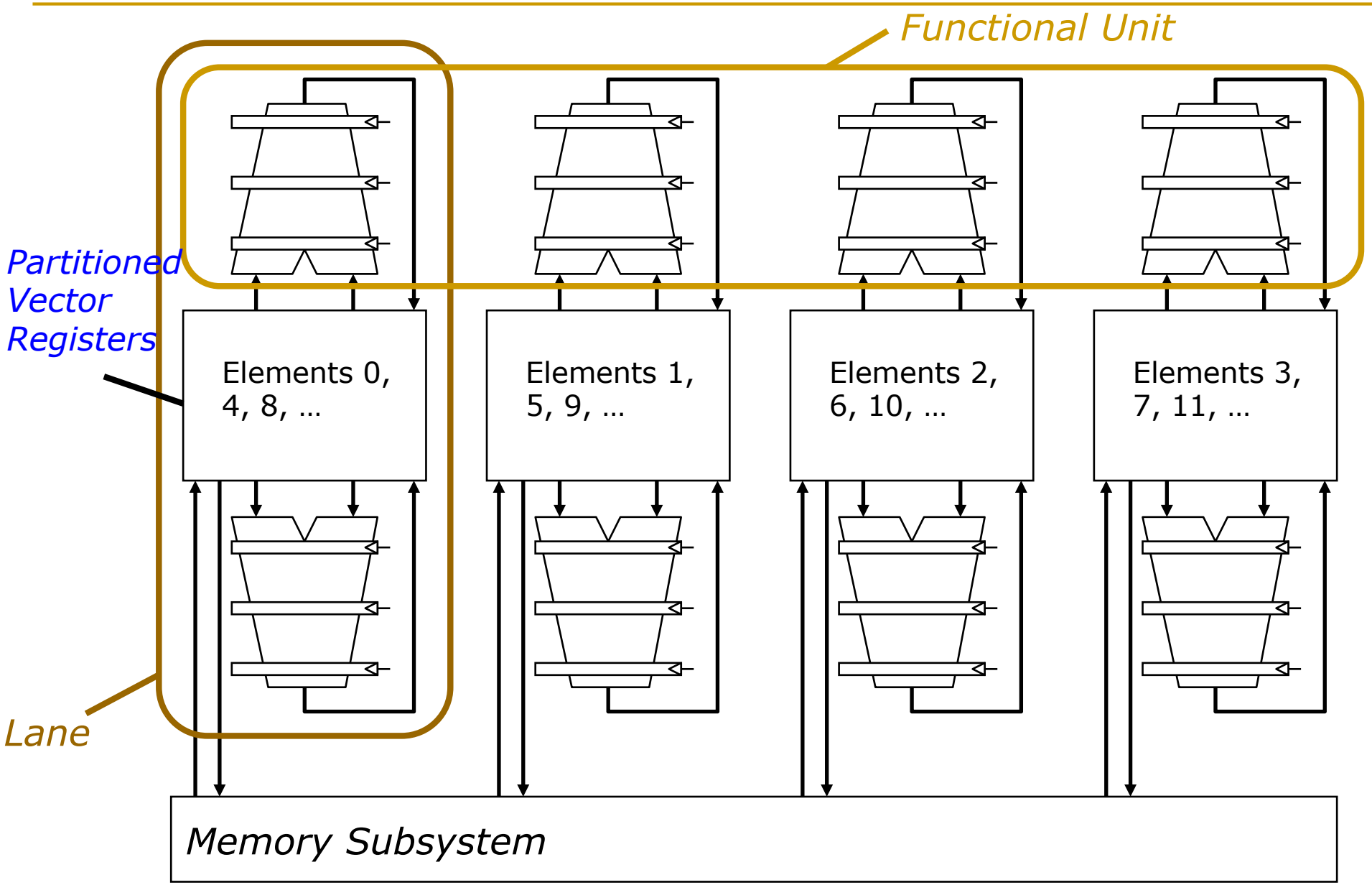
Recall: Array vs. Vector Processors



Vector Instruction Execution



Vector Unit Structure

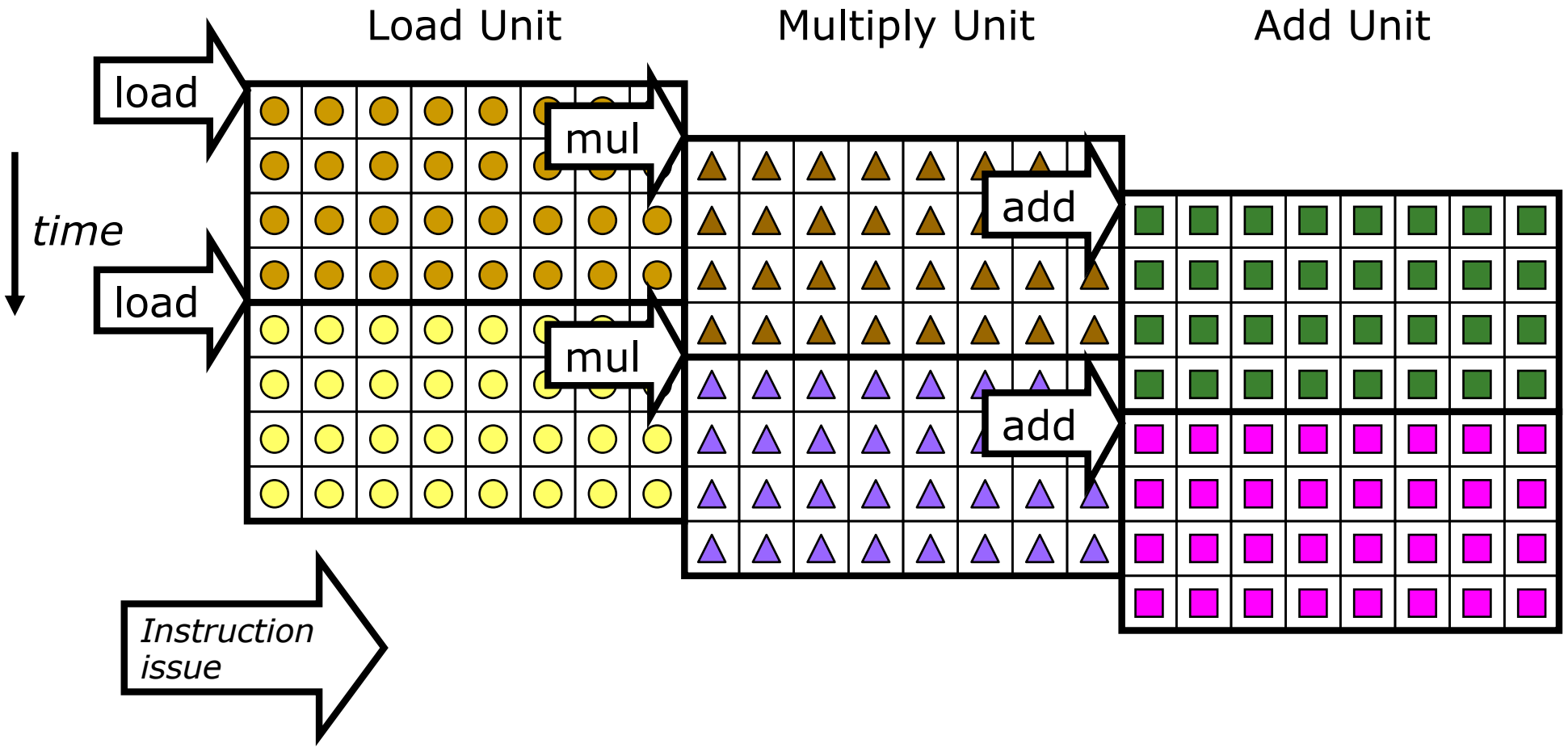


Slide credit: Krste Asanovic

Vector Instruction Level Parallelism

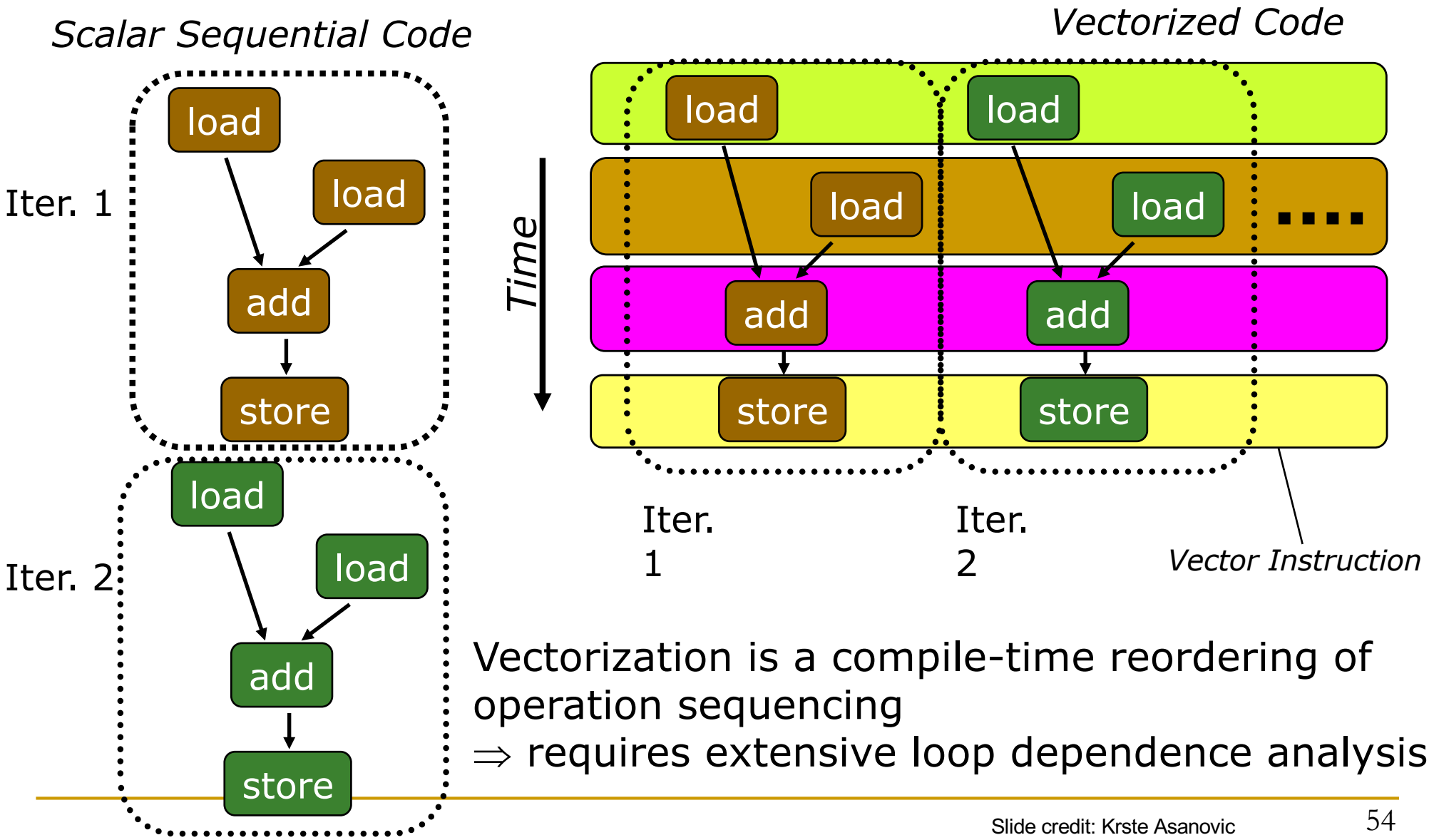
Can overlap execution of multiple vector instructions

- Example machine has 32 elements per vector register and 8 lanes
- Completes 24 operations/cycle while issuing 1 vector instruction/cycle



Automatic Code Vectorization

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



Vectorization is a compile-time reordering of operation sequencing

⇒ requires extensive loop dependence analysis

Slide credit: Krste Asanovic

Vector/SIMD Processing Summary

- Vector/SIMD machines are good at exploiting regular data-level parallelism
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)

- Performance improvement limited by vectorizability of code
  - Scalar operations limit vector machine performance
  - Remember Amdahl's Law
  - CRAY-1 was the fastest SCALAR machine at its time!

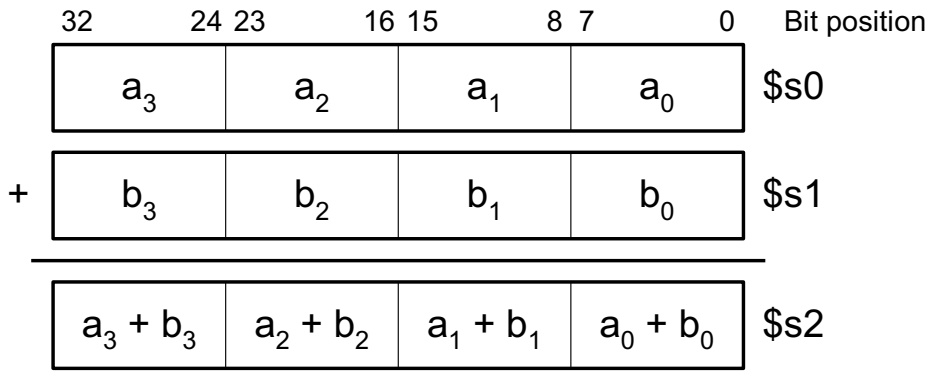
- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

SIMD Operations in Modern ISAs

SIMD ISA Extensions

- Single Instruction Multiple Data (SIMD) extension instructions
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called *packed arithmetic*)
- For example: add four 8-bit numbers
- Must modify ALU to eliminate carries between 8-bit values

```
padd8 $s2, $s0, $s1
```



Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements simultaneously
  - *À la* array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind

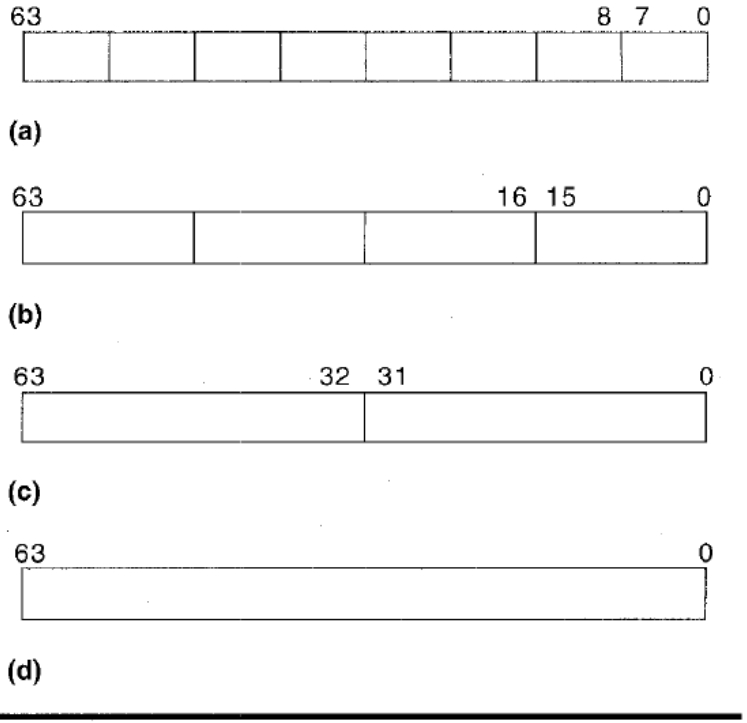


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

No VLEN register
Opcode determines data type:
8 8-bit bytes
4 16-bit words
2 32-bit doublewords
1 64-bit quadword

Stride is always equal to 1.

Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

MMX Example: Image Overlaying (I)

- Goal: Overlay the human in image 1 on top of the background in image 2

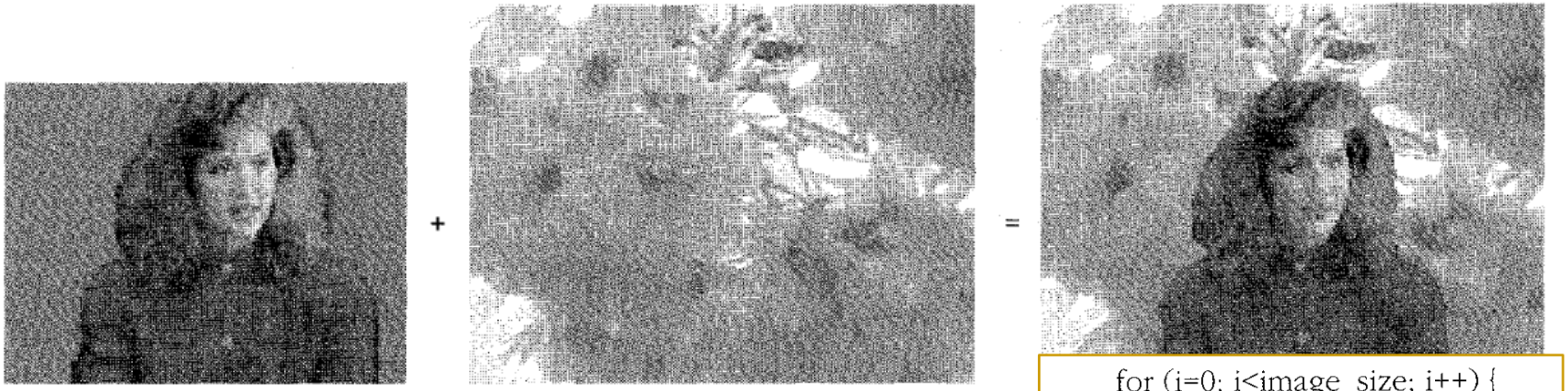


Figure 8. Chroma keying: image overlay using a background color.

```
for (i=0; i<image_size; i++) {
if (x[i] == Blue)  new_image[i] =y[i];
        else new_image[i] = x[i];
}
```

PCMPEQB MM1, MM3

MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF

Bitmask

Figure 9. Generating the selection bit mask.

Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

MMX Example: Image Overlaying (II)

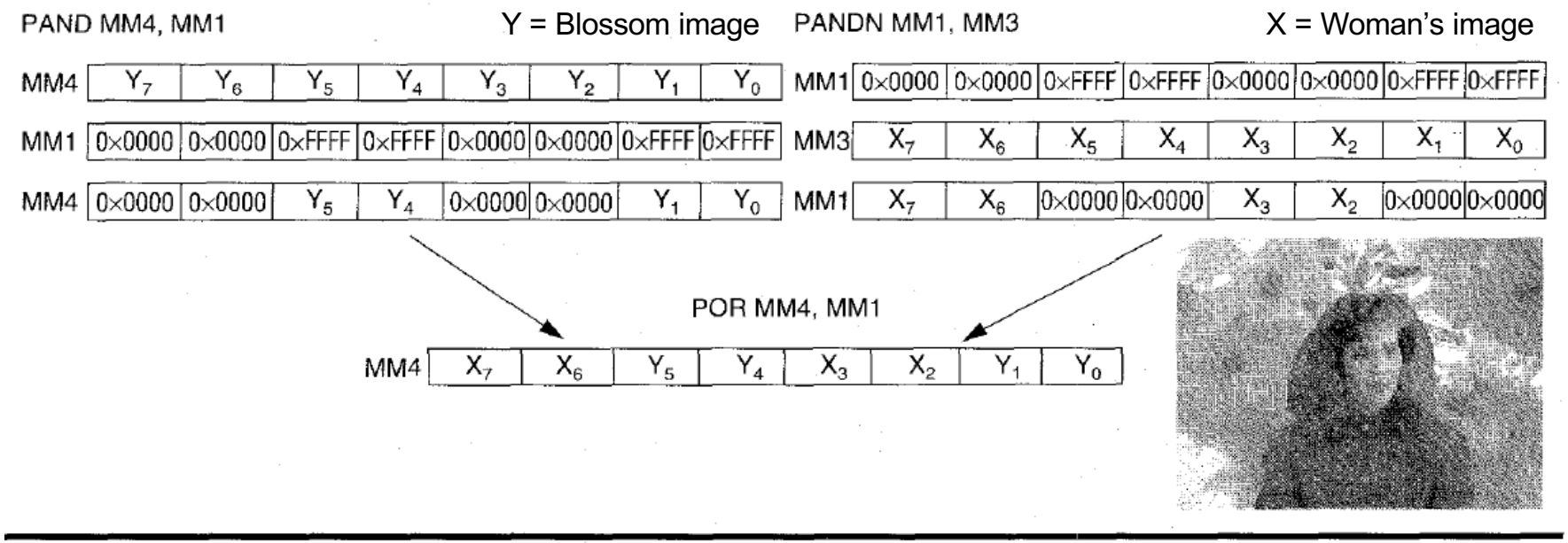


Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
Movq     mm3, mem1    /* Load eight pixels from
                         woman's image
Movq     mm4, mem2    /* Load eight pixels from the
                         blossom image
Pcmpeqb  mm1, mm3
Pand     mm4, mm1
Pandn    mm1, mm3
Por      mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.

Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

GPUs (Graphics Processing Units)

GPUs are SIMD Engines Underneath

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)
- However, the programming is done using threads, NOT SIMD instructions
- To understand this, let's go back to our parallelizable code example
- But, before that, let's distinguish between
  - Programming Model (Software)

    vs.
  - Execution Model (Hardware)

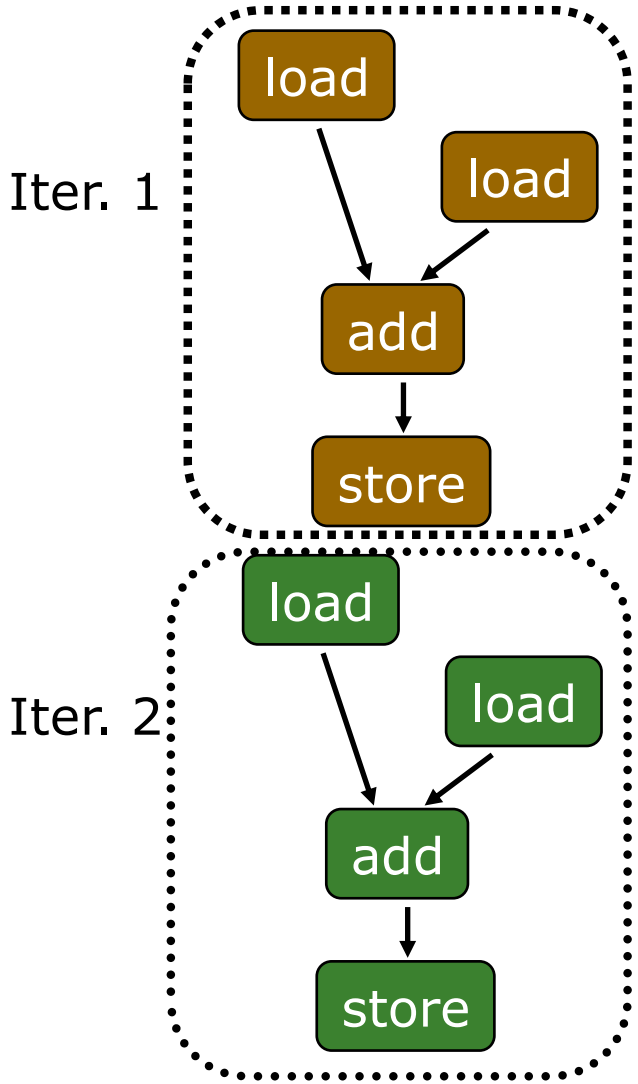
Programming Model vs. Hardware Execution Model

- Programming Model refers to how the programmer expresses the code
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), …
- Execution Model refers to how the hardware executes the code underneath
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, …
- Execution Model can be very different from the Programming Model
  - E.g., von Neumann model implemented by an OoO processor
  - E.g., SPMD model implemented by a SIMD processor (a GPU)

How Can You Exploit Parallelism Here?

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

Scalar Sequential Code



Let's examine three programming options to exploit instruction-level parallelism present in this sequential code:

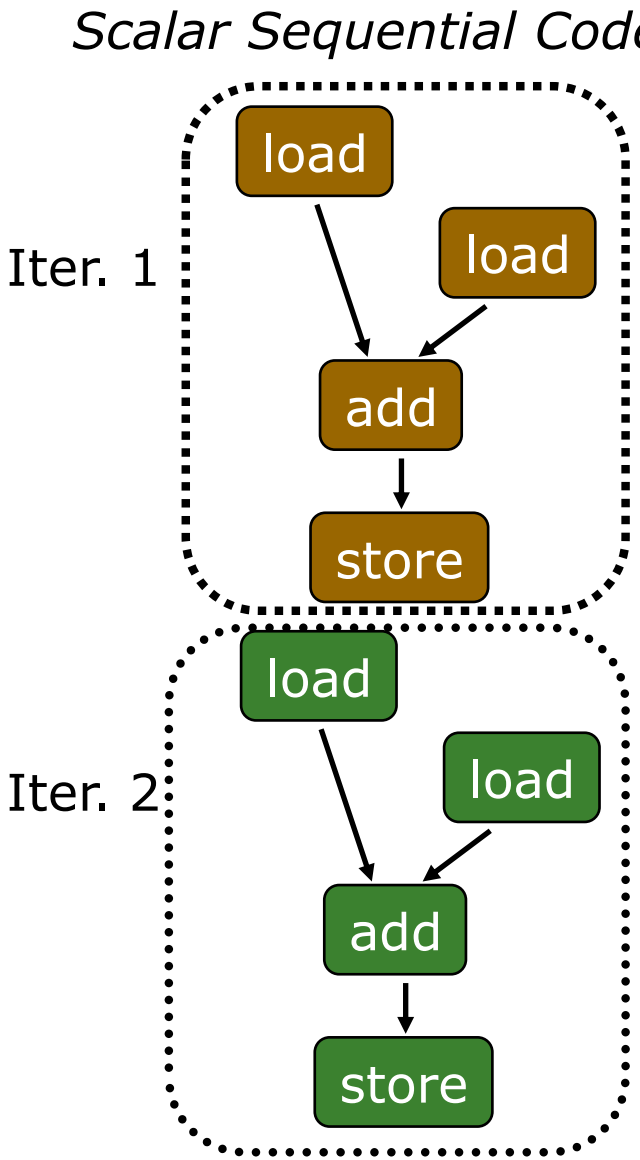
1. Sequential (SISD)

2. Data-Parallel (SIMD)

3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

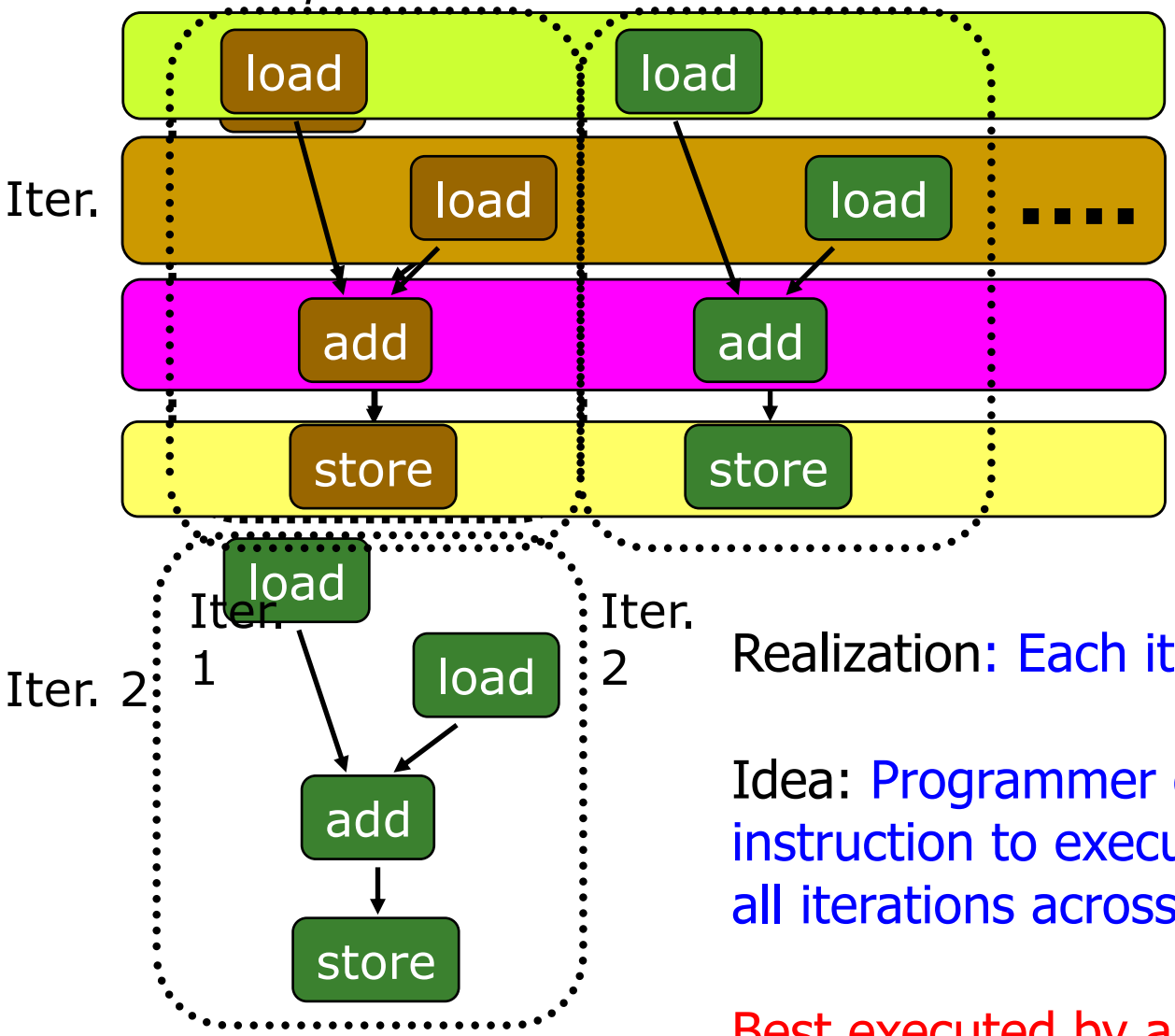
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



- Can be executed on a:
- Pipelined processor
- Out-of-order execution processor
  - Independent instructions executed when ready
  - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
  - In other words, the loop is dynamically unrolled by the hardware
- Superscalar or VLIW processor
  - Can fetch and execute multiple instructions per cycle

Prog. Model 2: Data Parallel (SIMD)

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



Vectorized Code

VLD A → V1

VLD B → V2

VADD V1 + V2 → V3

VST V3 → C

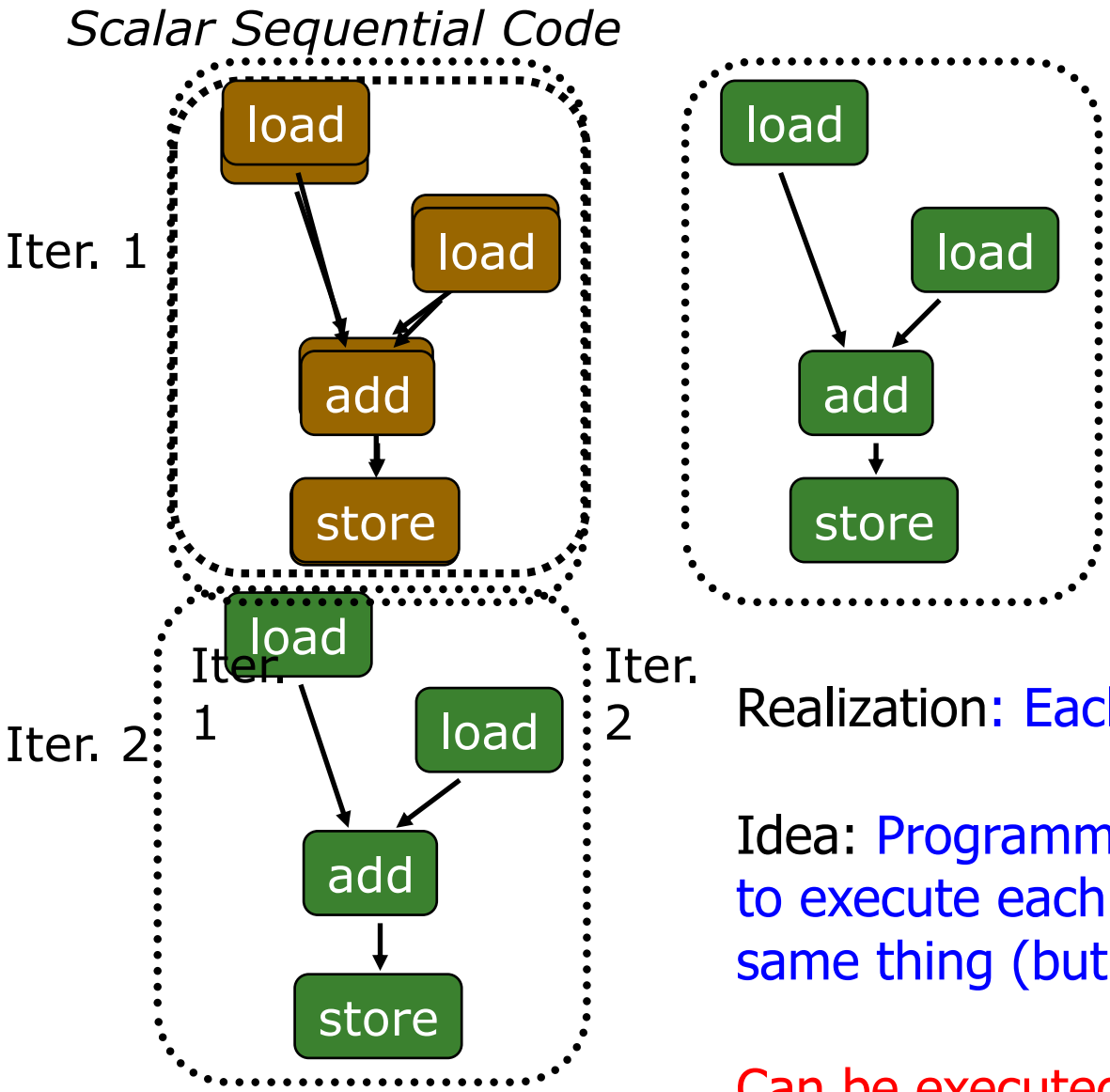
Realization: Each iteration is independent

Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Best executed by a SIMD processor (vector, array)

Prog. Model 3: Multithreaded

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



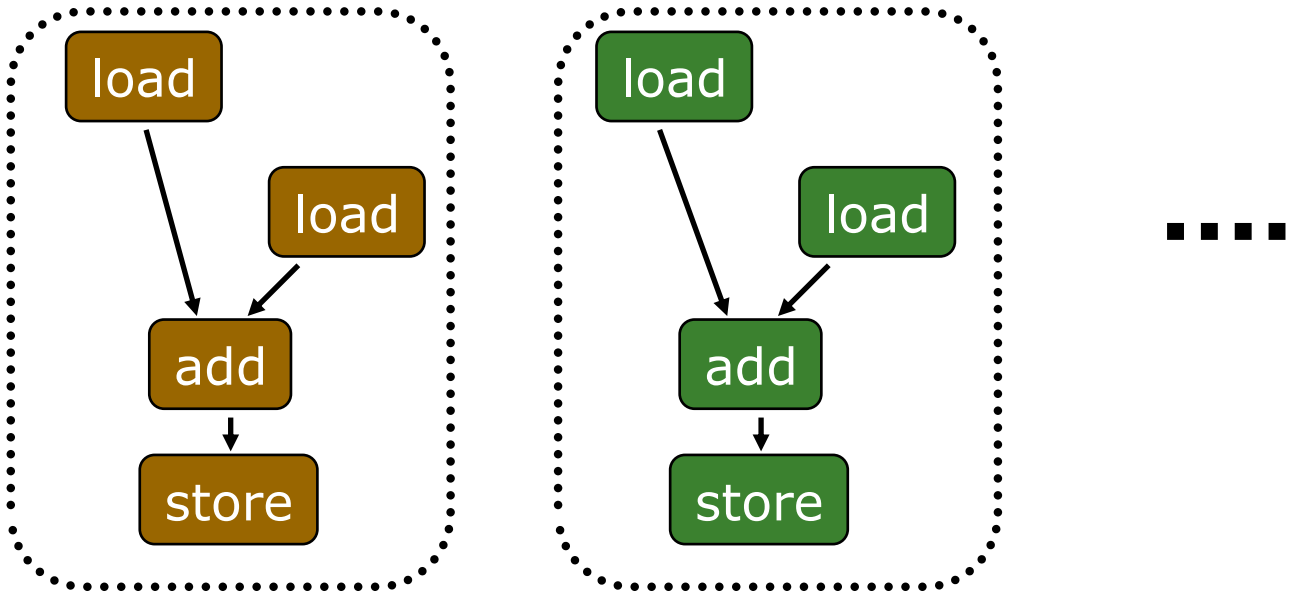
Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Can be executed on a MIMD machine

Prog. Model 3: Multithreaded

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



Iter. 1

Iter. 2

Realization: Each iteration is independent

This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread

A GPU is a SIMD (SIMT) Machine

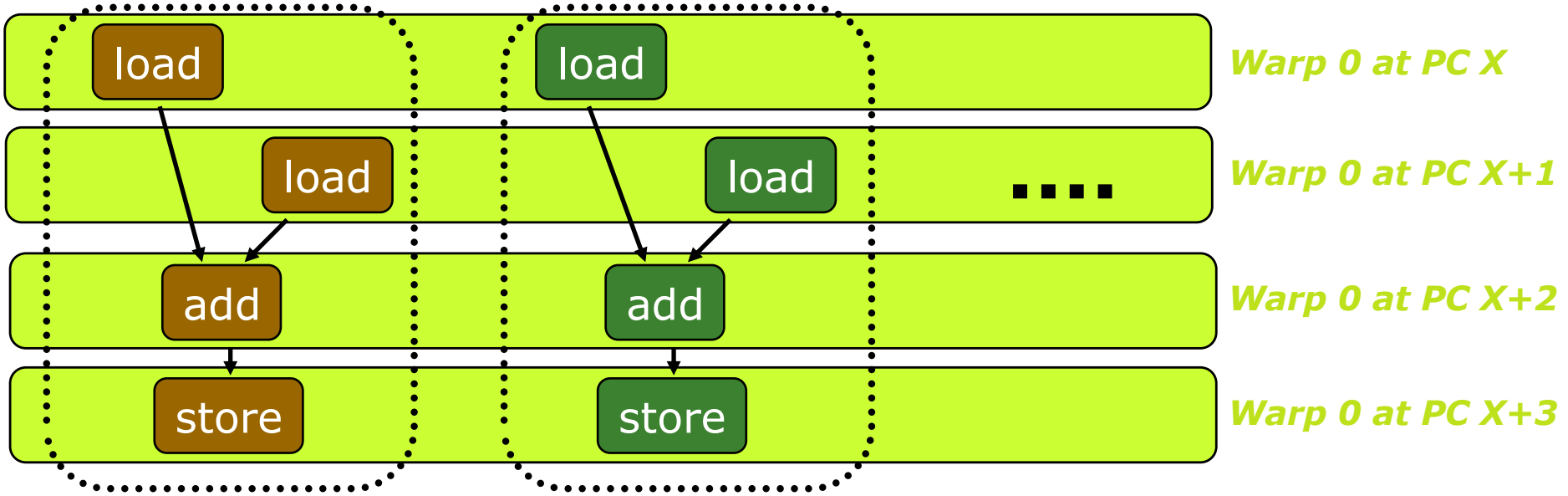
- Except it is **not** programmed using SIMD instructions

- It is programmed using threads (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

- A set of threads executing the same instruction are dynamically grouped into a **warp (wavefront)** by the hardware
  - A warp is essentially a SIMD operation formed by hardware!

SPMD on SIMT Machine

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



Warp: A set of threads that execute the same instruction (i.e., at the same PC)

This particular model is also called:

SPMD: Single Program Multiple Data

A GPU executes it using the SIMT model:
Single Instruction Multiple Thread

Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)

SIMD vs. SIMT Execution Model

- SIMD: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

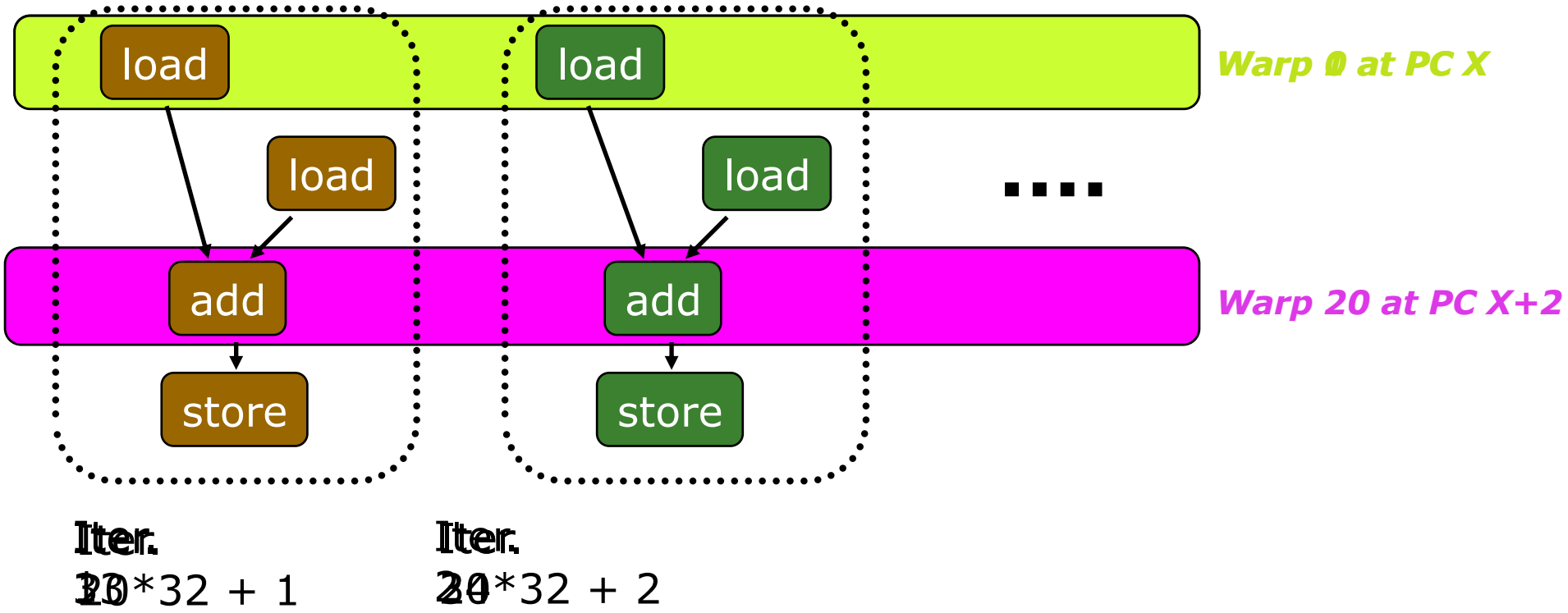
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently (on any type of scalar pipeline) → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

Multithreading of Warps

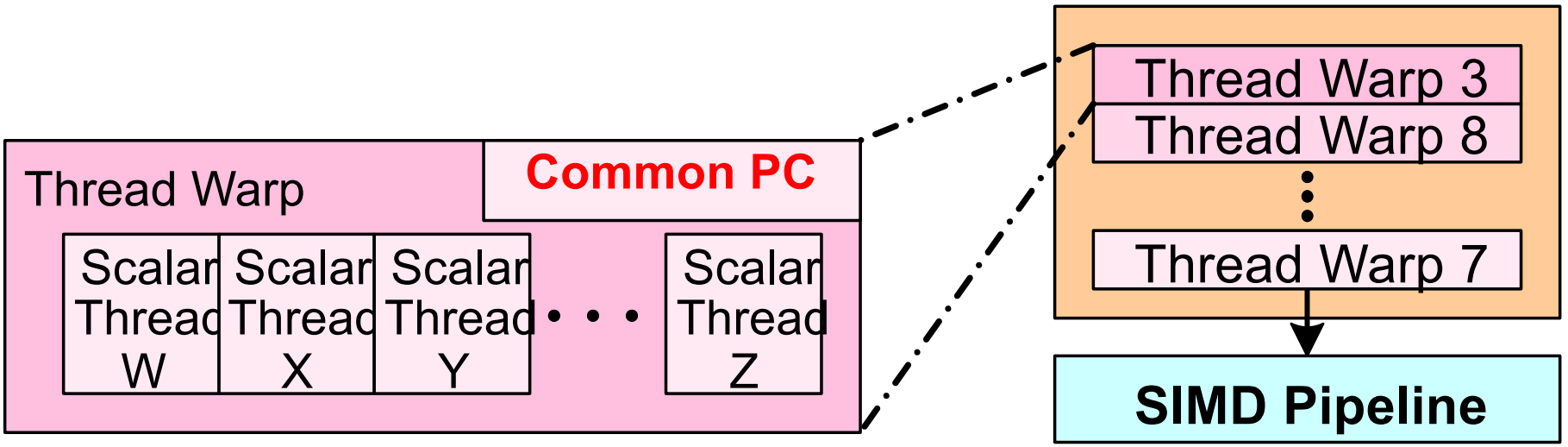
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

- Assume a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading of warps



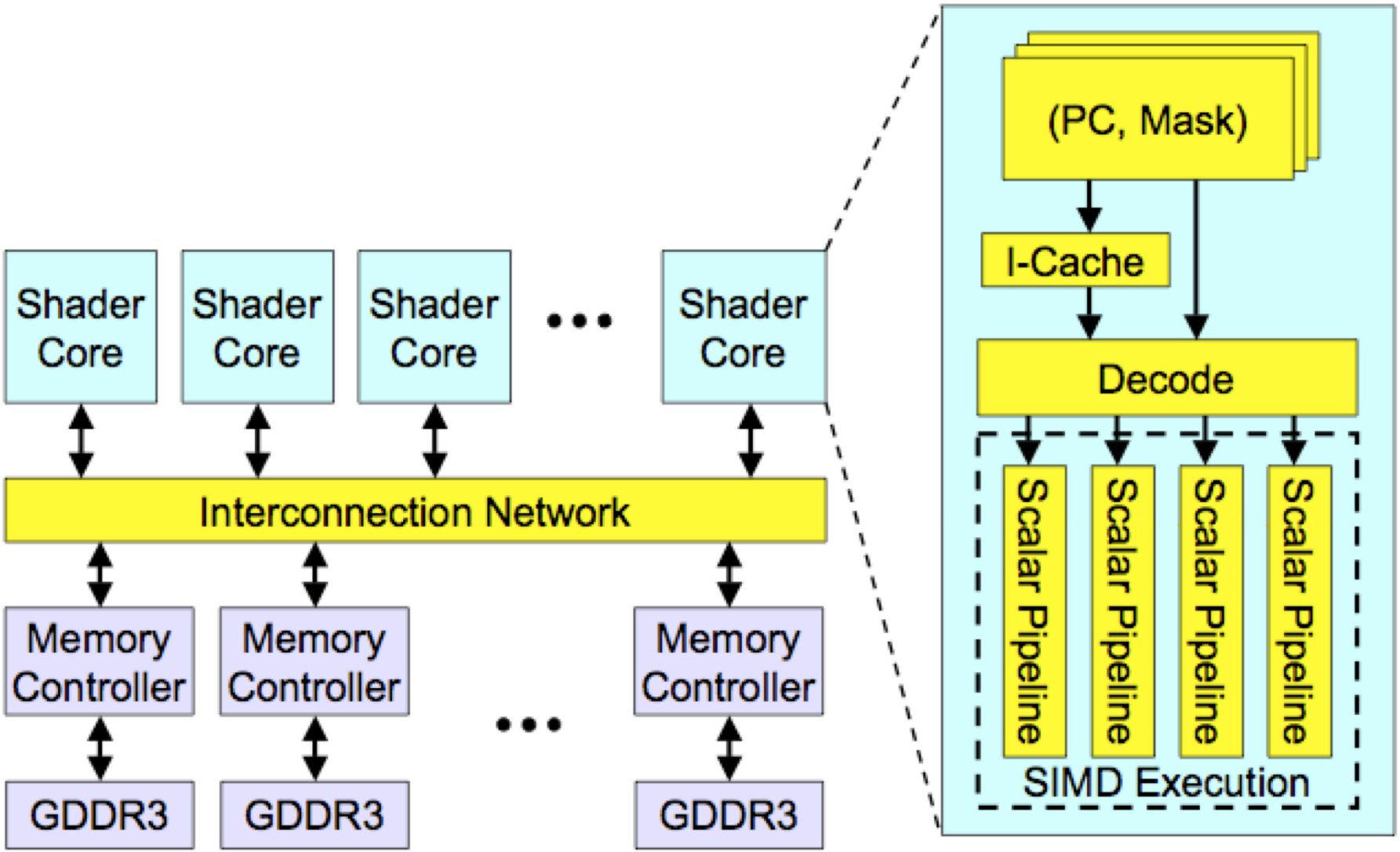
Warps and Warp-Level FGMT

- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same code
- Warp: The threads that run lengthwise in a woven fabric …



Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

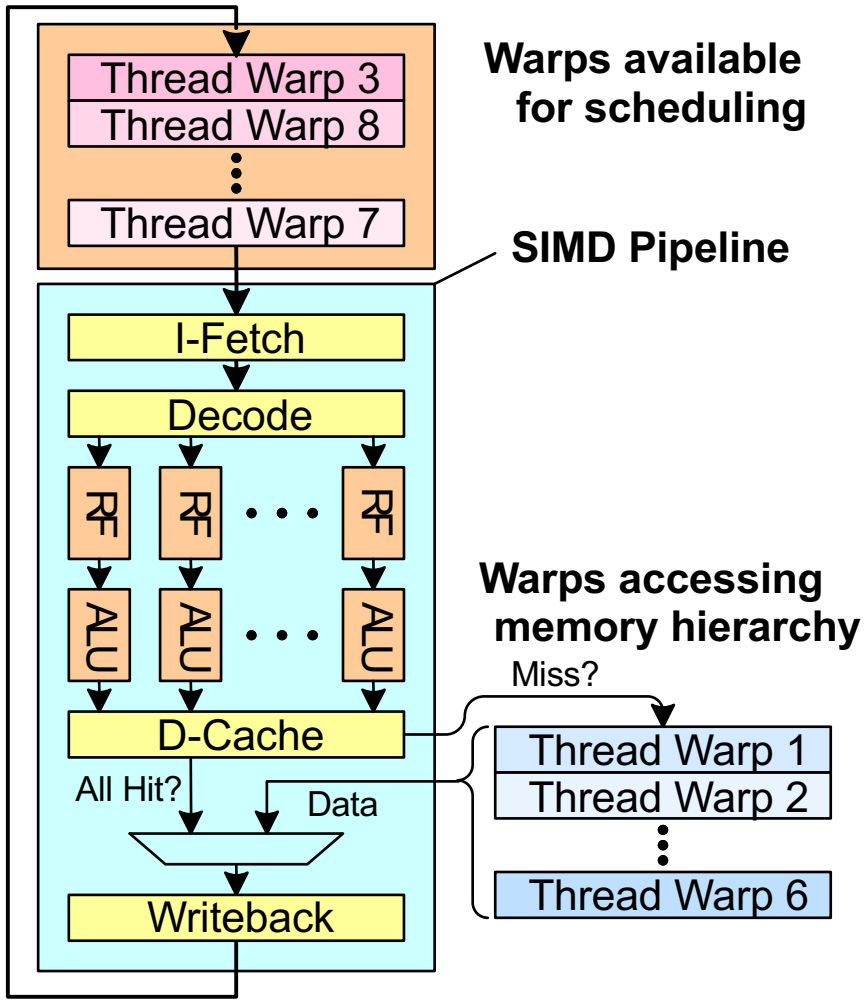
High-Level View of a GPU



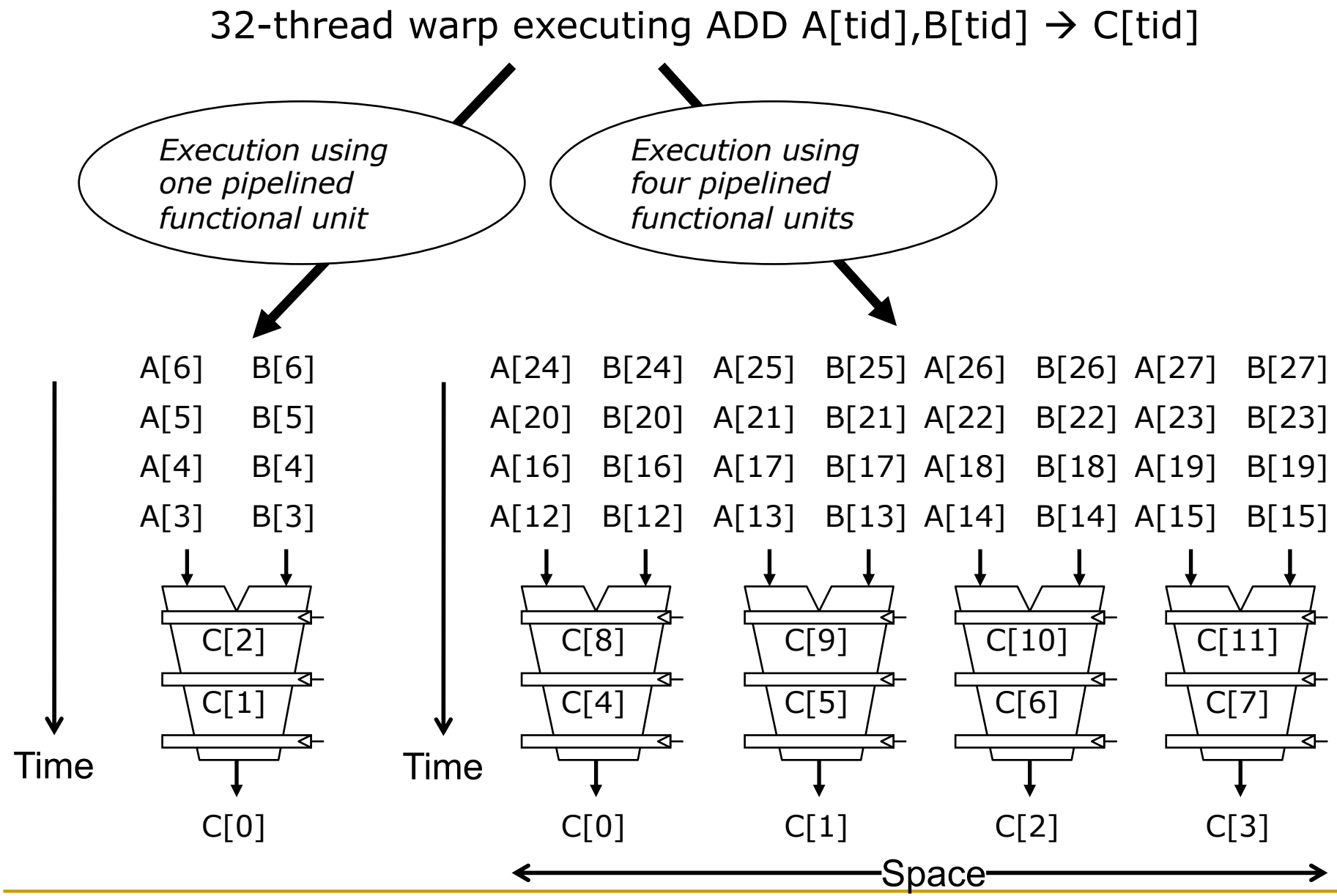
Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

Latency Hiding via Warp-Level FGMT

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables long latency tolerance
  - Millions of pixels

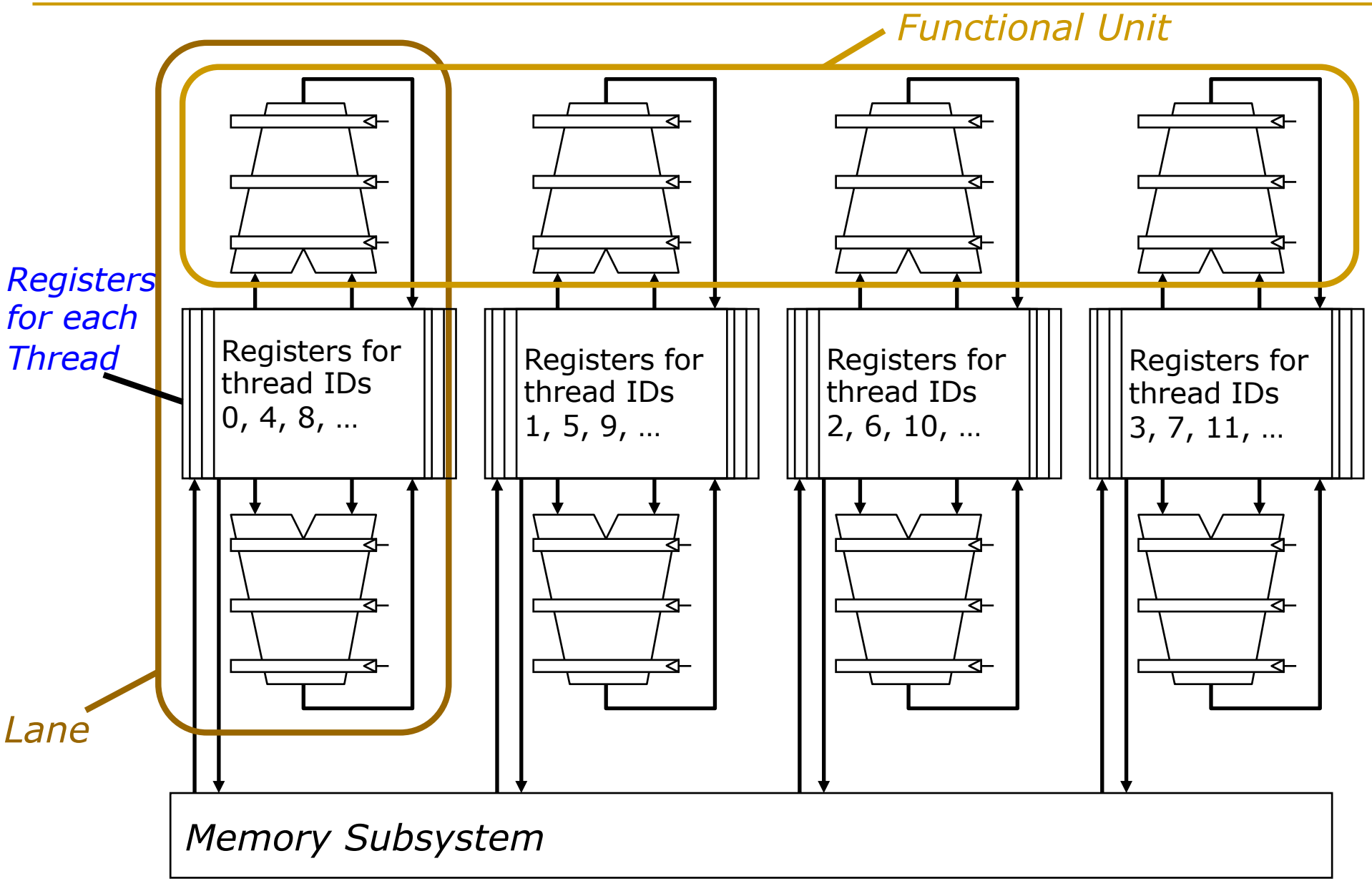


Warp Execution (Recall the Slide)



Slide credit: Krste Asanovic

SIMD Execution Unit Structure

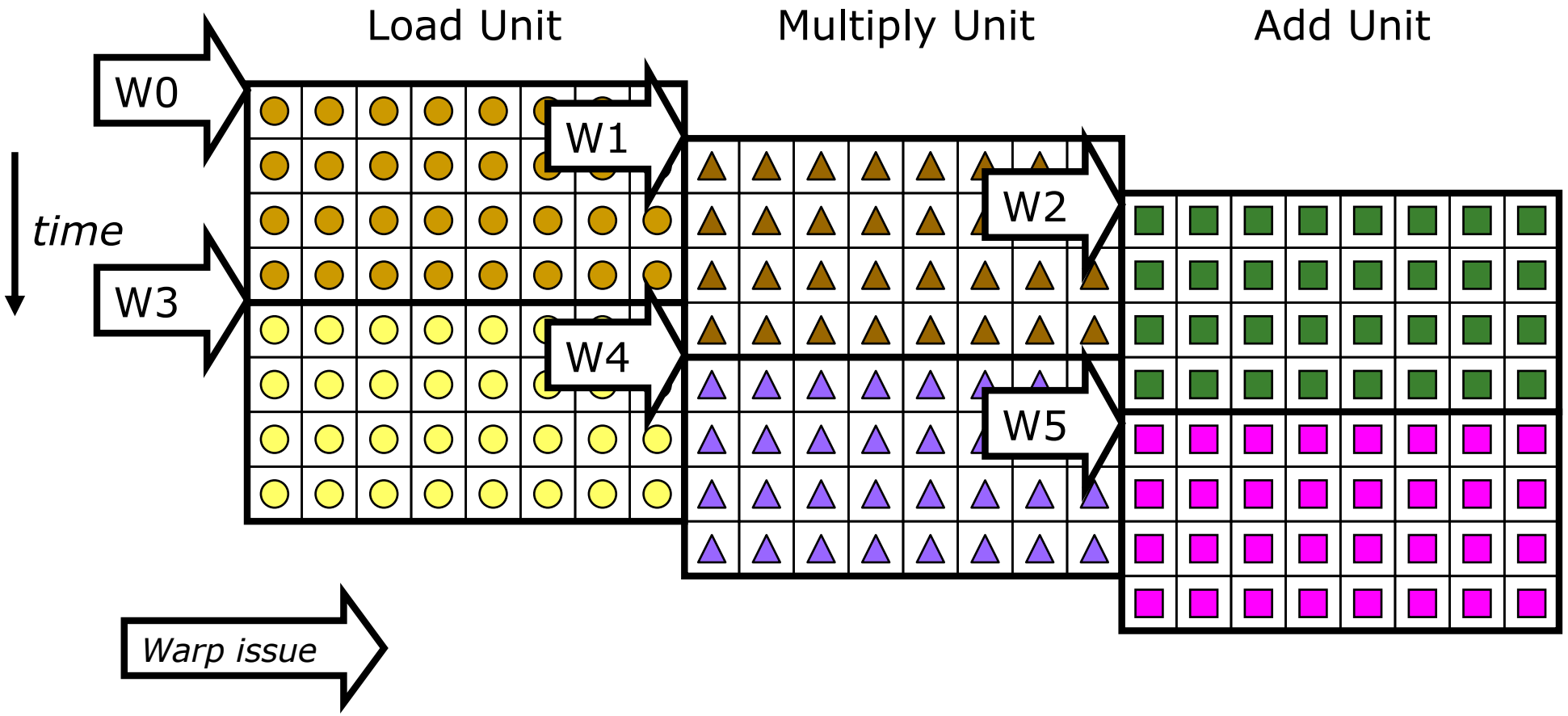


Slide credit: Krste Asanovic

Warp Instruction Level Parallelism

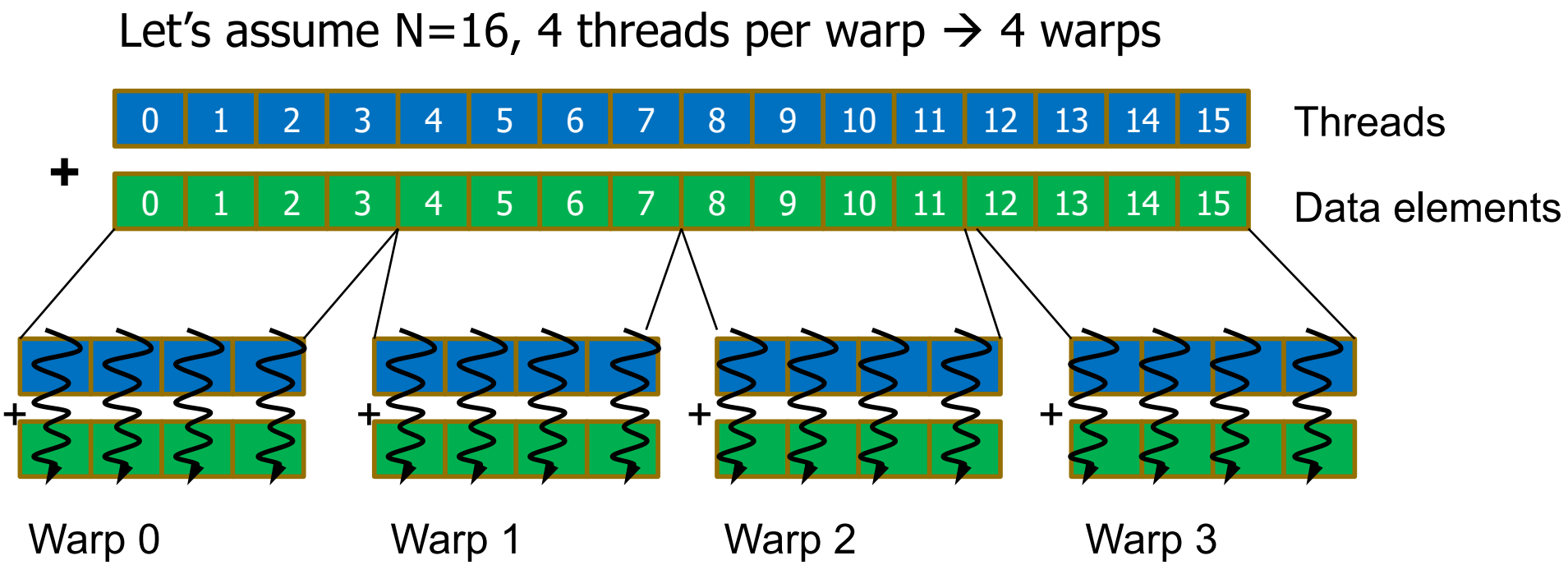
Can overlap execution of multiple instructions

- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle



SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements



Sample GPU SIMT Code (Simplified)

CPU code

```
for (ii = 0; ii < 100000; ++ii) {
C[ii] = A[ii] + B[ii];
}
```

CUDA code

```
// there are 100000 threads
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varA = aa[tid];
  int varB = bb[tid];
  C[tid] = varA + varB;
}
```

Sample GPU Program (Less Simplified)

CPU Program

```
void add matrix
 ( float *a, float* b, float *c, int N) {
   int index;
   for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j) {
         index = i + j*N;
         c[index] = a[index] + b[index];
        }
 }

int main () {

  add matrix (a, b, c, N);
}
```

GPU Program

```
__global__  add_matrix
  ( float *a, float *b, float *c, int N) {
int i = blockIdx.x *  blockDim.x + threadIdx.x;
Int j = blockIdx.y * blockDim.y  + threadIdx.y;
int index = i + j*N;
 if (i < N && j < N)
   c[index] = a[index]+b[index];
}

Int main() {
  dim3 dimBlock( blocksize, blocksize) ;
  dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```

Slide credit: Hyesoon Kim

Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Sequential instruction execution; lock-step operations in a SIMD instruction
  - Programming model is SIMD (no extra threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables multithreading and flexible dynamic grouping of threads
  - ISA is scalar → SIMD operations can be formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

SPMD

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization
- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers
- Essentially, multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware

SIMD vs. SIMT Execution Model

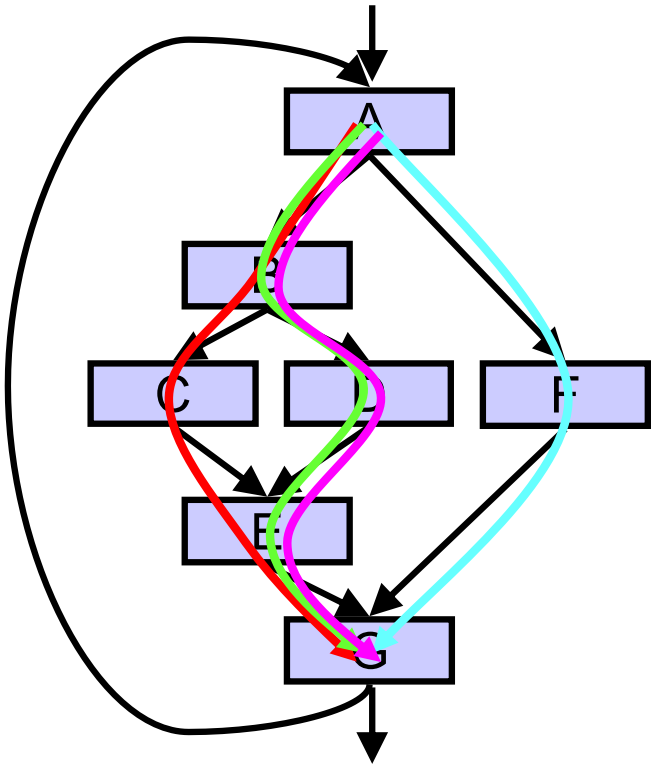
- SIMD: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

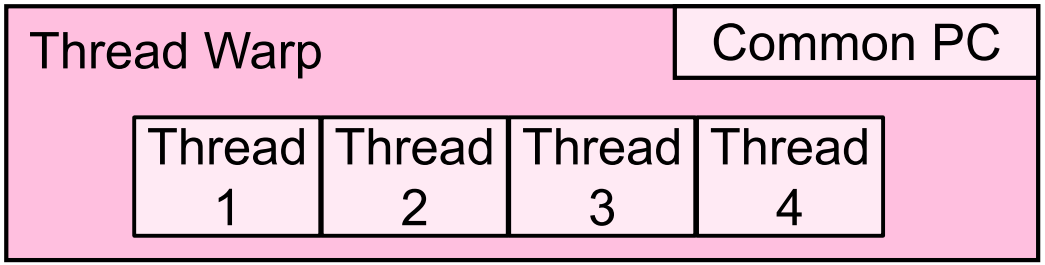
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths



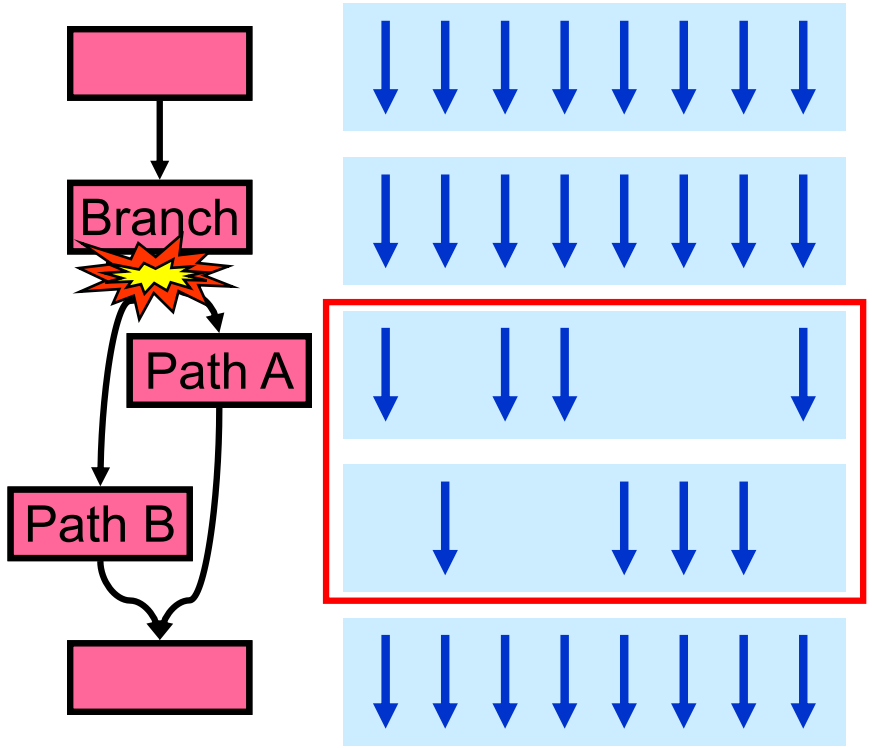


Slide credit: Tor Aamodt

Control Flow Problem in GPUs/SIMT

- A GPU uses a SIMD pipeline to save area on control logic
  - Groups scalar threads into warps

- Branch divergence occurs when threads inside warps branch to different execution paths



This is the same as conditional/predicated/masked execution. Recall the Vector Mask and Masked Vector Operations?

Slide credit: Tor Aamodt

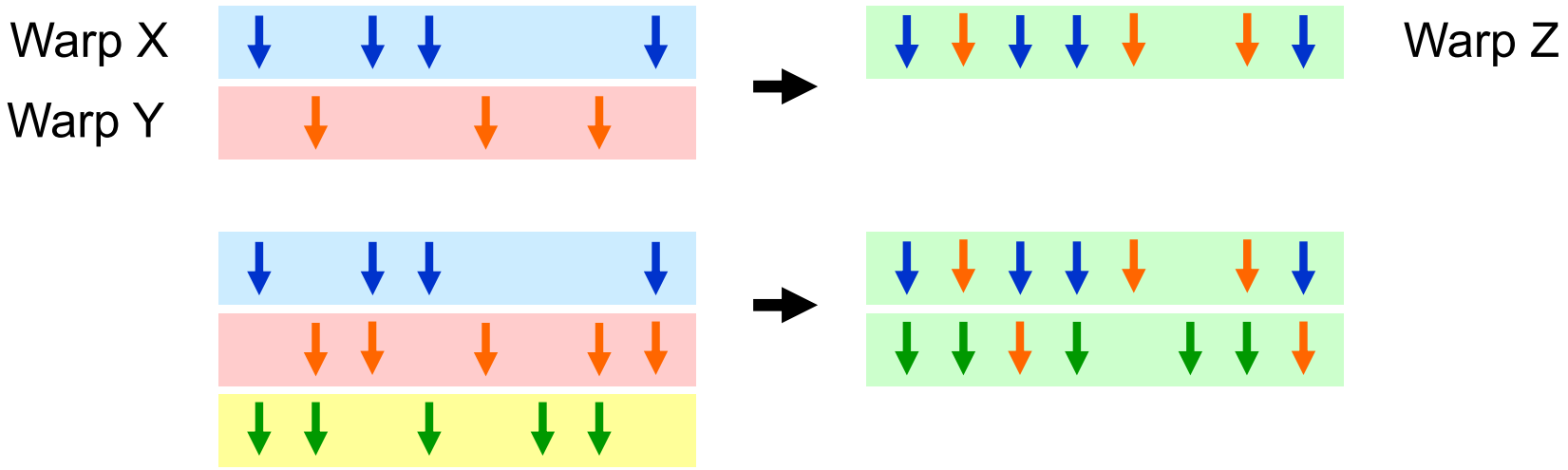
Remember: Each Thread Is Independent

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

- If we have many threads
- We can find individual threads that are at the same PC
- And, group them together into a single warp dynamically
- This reduces "divergence" → improves SIMD utilization
  - SIMD utilization: fraction of SIMD lanes executing a useful operation (i.e., executing an active thread)

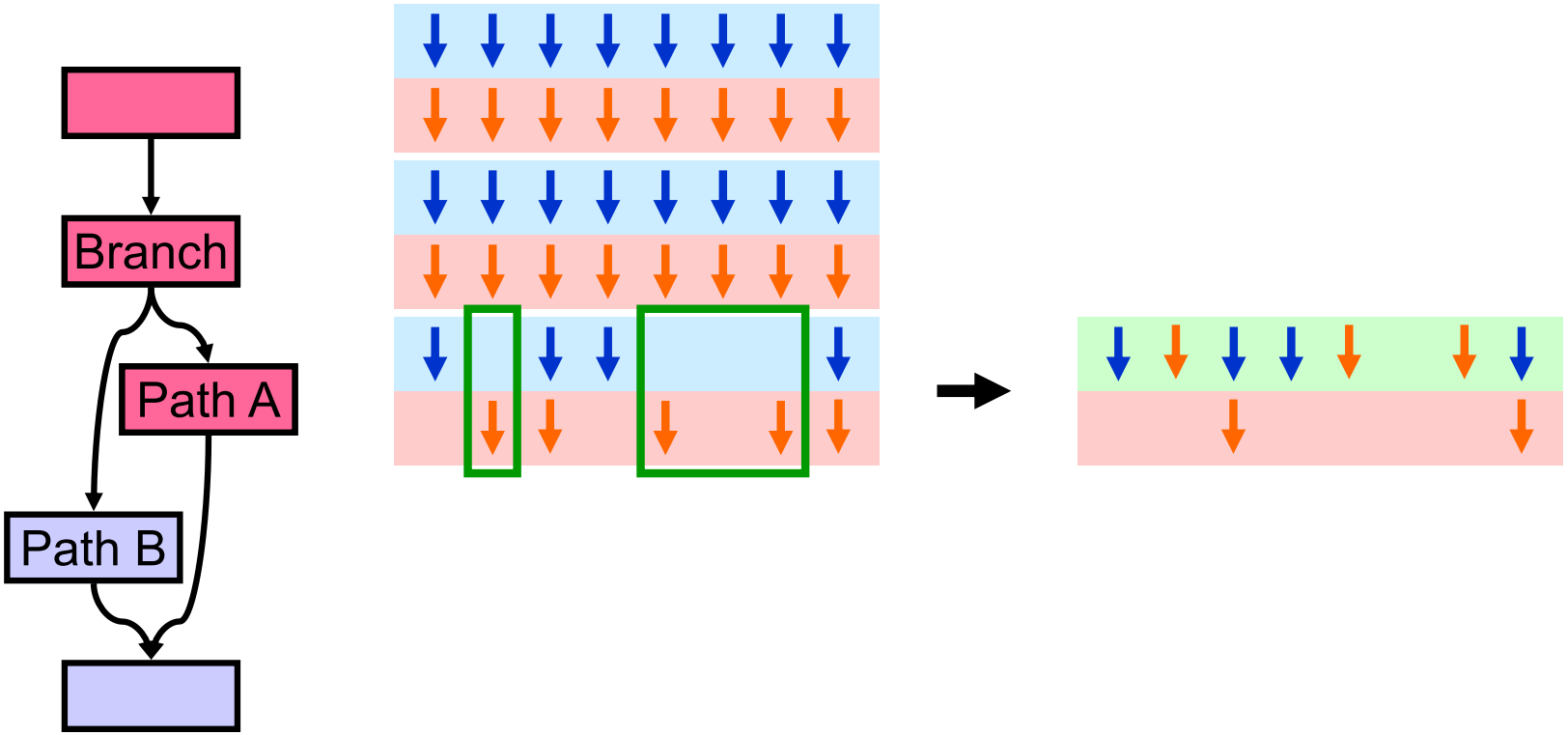
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warps from warps that are waiting
  - Enough threads branching to each path enables the creation of full new warps



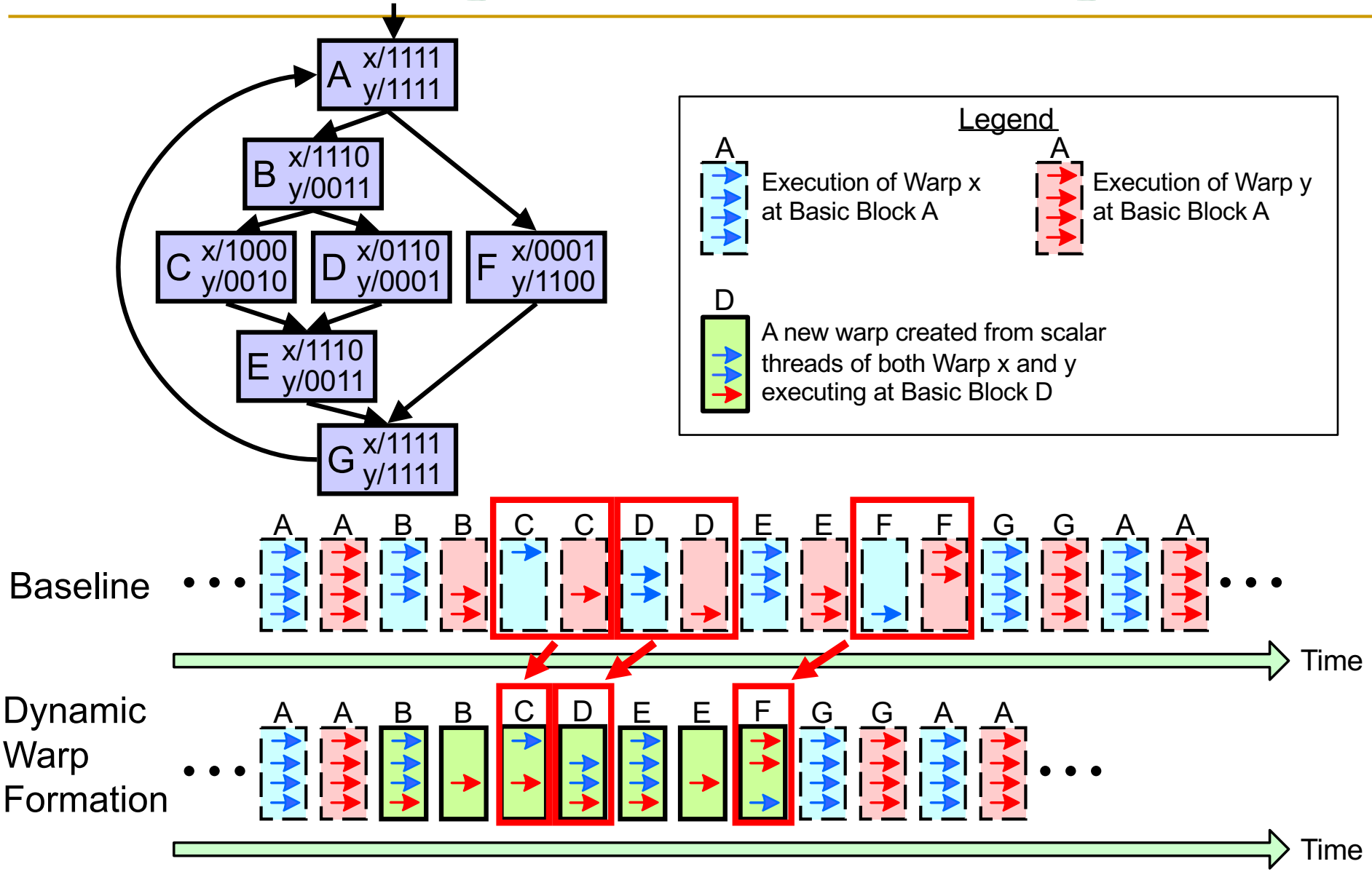
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)

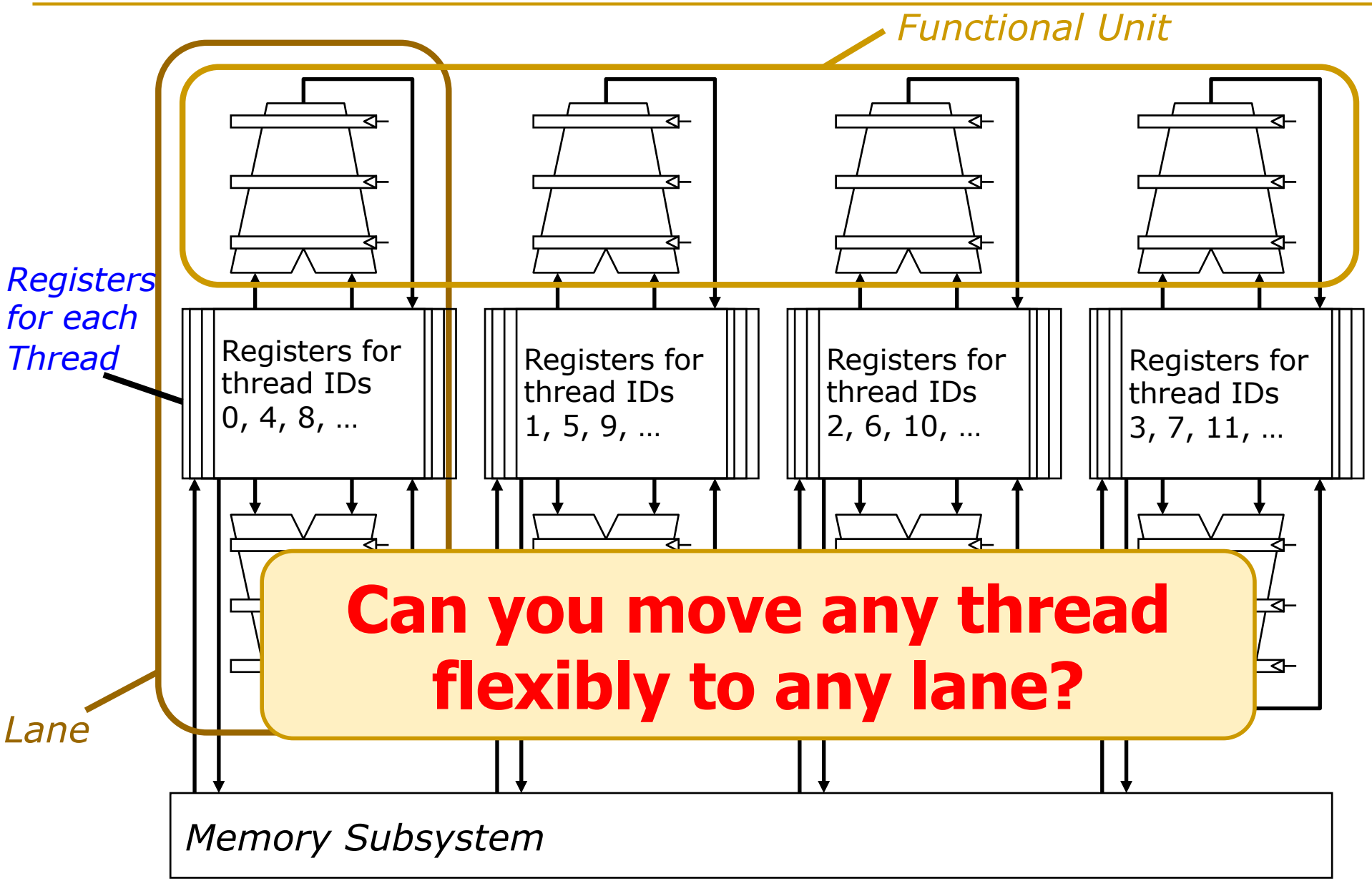


- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

Dynamic Warp Formation Example



Hardware Constraints Limit Flexibility of Warp Grouping



Slide credit: Krste Asanovic

An Example GPU

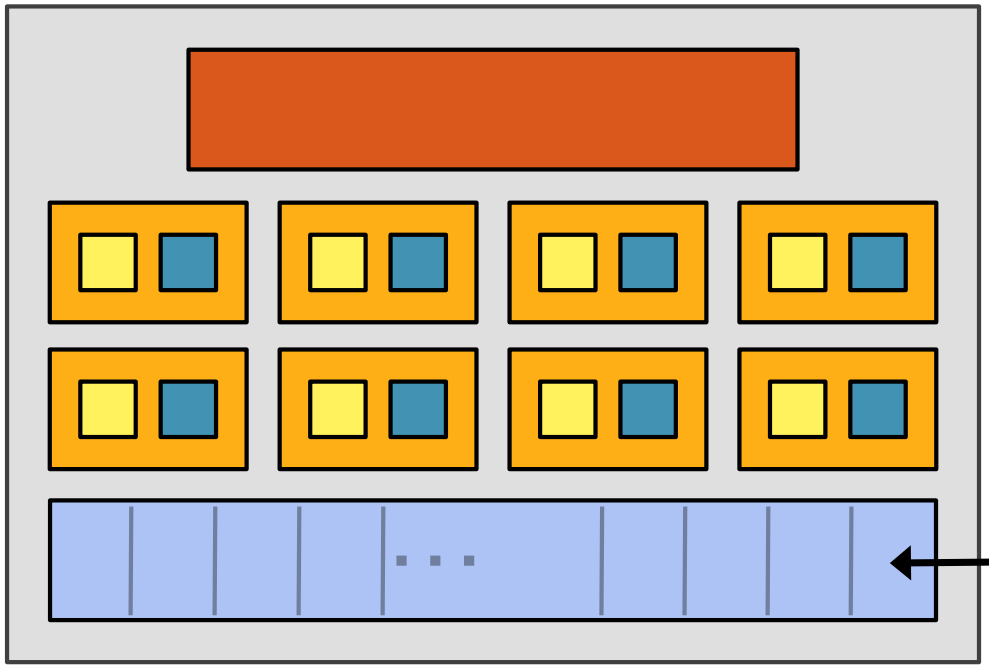
NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - “SIMT execution”

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285 "core"



64 KB of storage
for thread contexts
(registers)

= SIMD functional unit, control shared across 8 units

= multiply-add

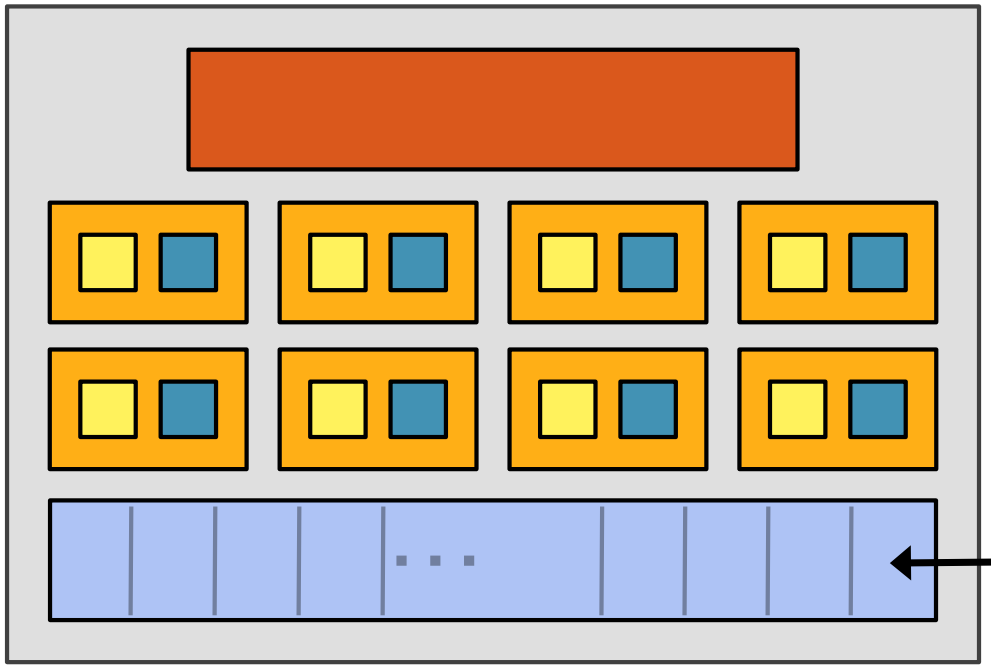
= multiply

= instruction stream decode

= execution context storage

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285 "core"

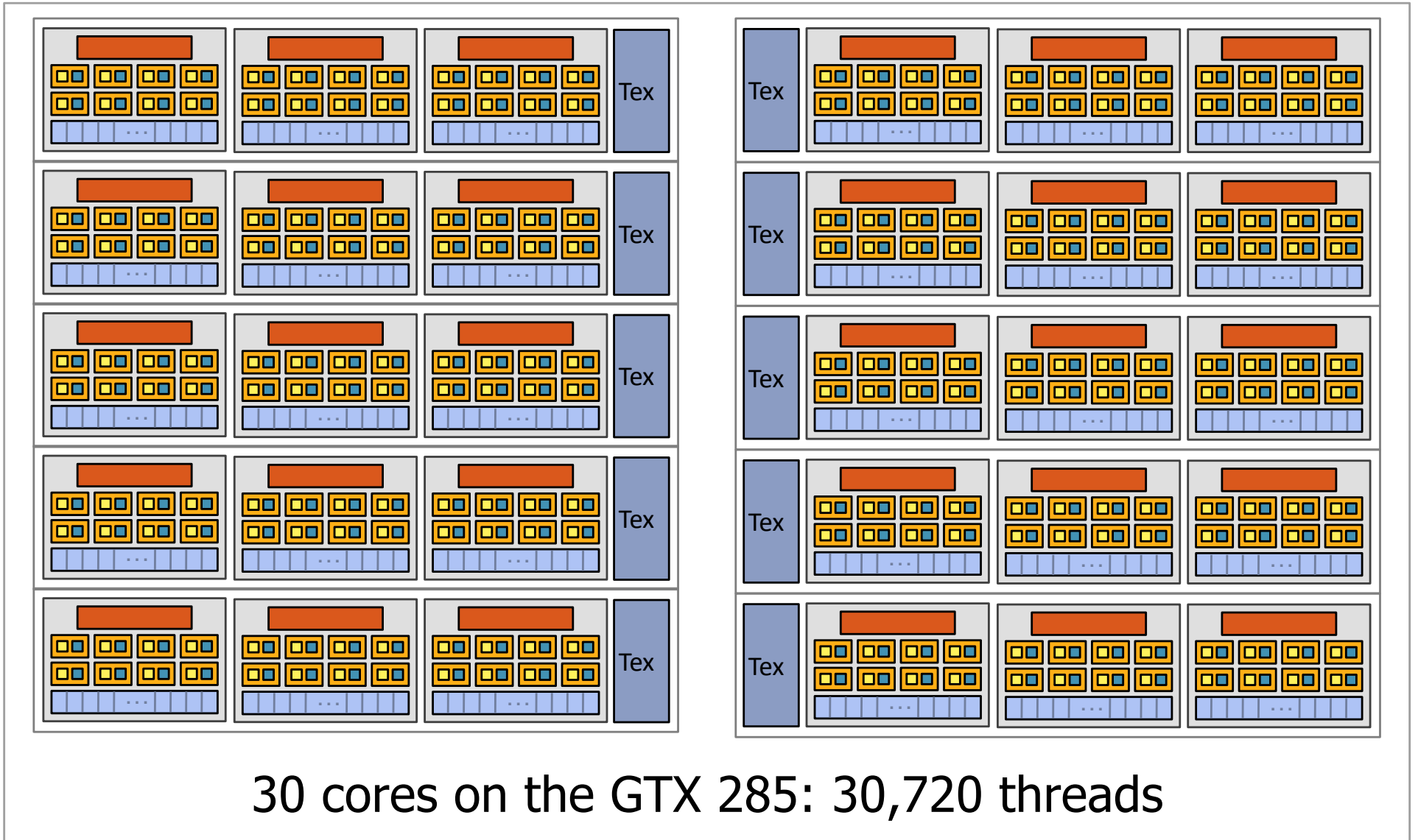


64 KB of storage
for thread contexts
(registers)

- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian

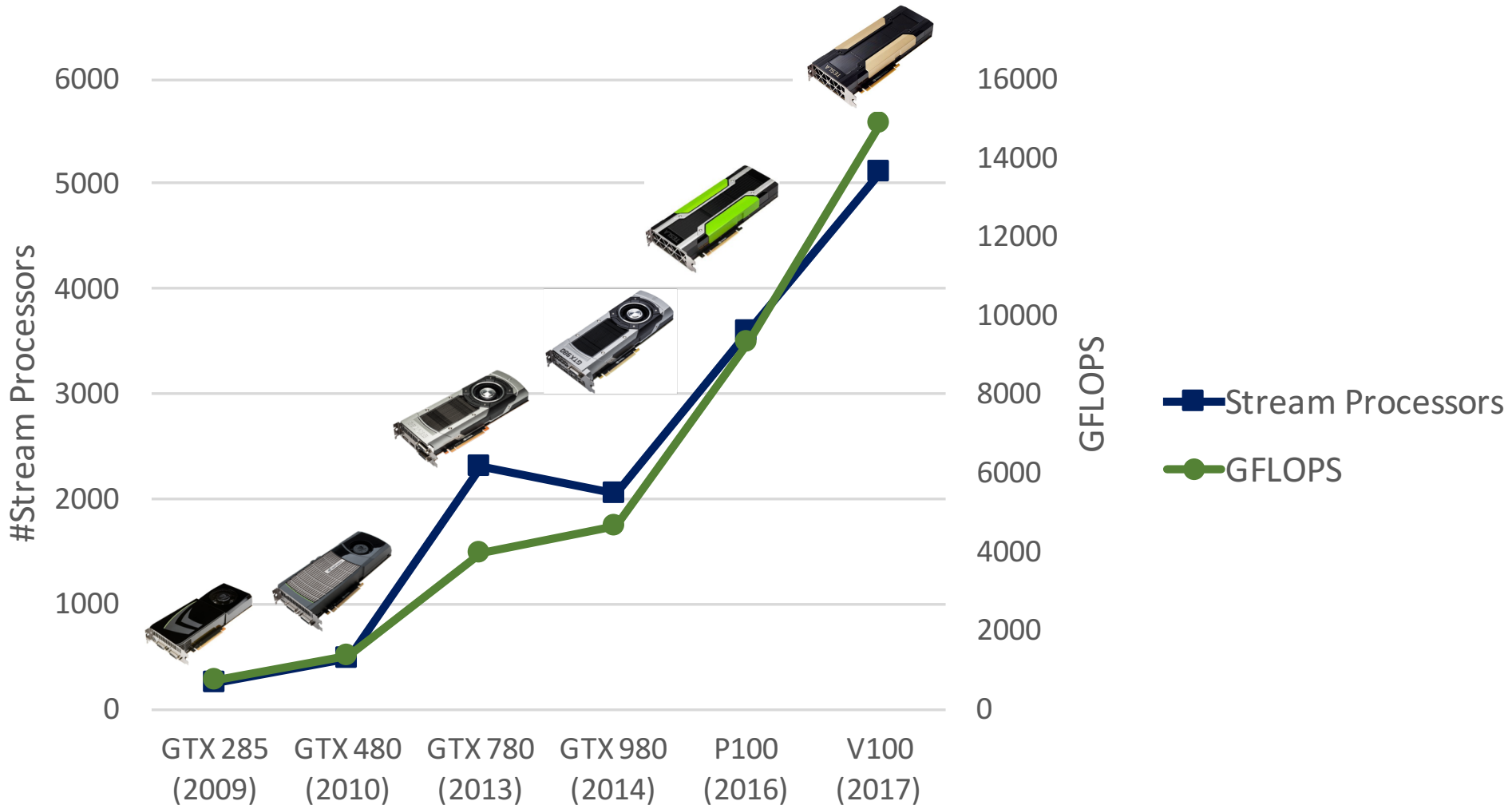
NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian

Evolution of NVIDIA GPUs



NVIDIA V100

- NVIDIA-speak:
  - 5120 stream processors
  - “SIMT execution”

- Generic speak:
  - 80 cores
  - 64 SIMD functional units per core
  - Tensor cores for Machine Learning

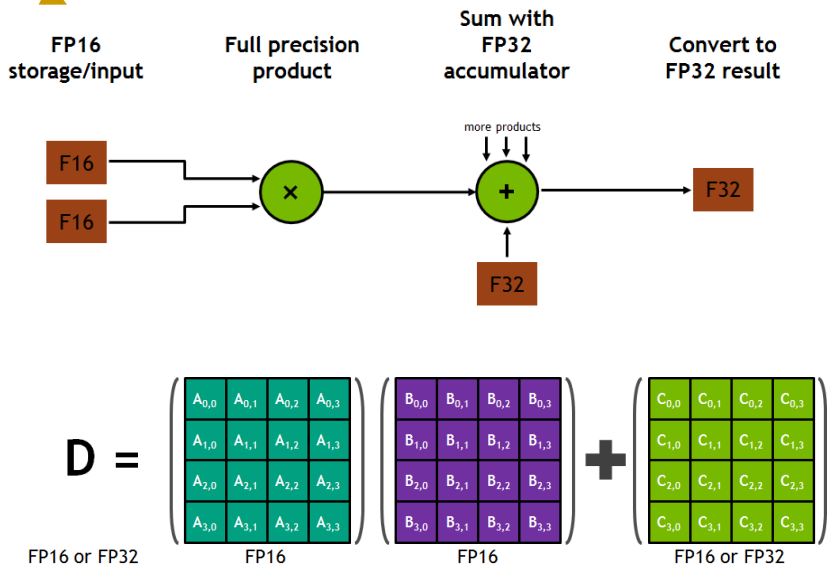
NVIDIA V100 Block Diagram

https://devblogs.nvidia.com/inside-volta/

80 cores on the V100

NVIDIA V100 Core

15.7 TFLOPS Single Precision
7.8 TFLOPS Double Precision
125 TFLOPS for Deep Learning (Tensor cores)



https://devblogs.nvidia.com/inside-volta/

Computer Architecture

Lecture 7: SIMD Processors and GPUs

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2018

10 October 2018