EE-508: Hardware Foundations for Machine Learning GPU Architecture

University of Southern California

Ming Hsieh Department of Electrical and Computer Engineering

Instructors:
Arash Saifhashemi

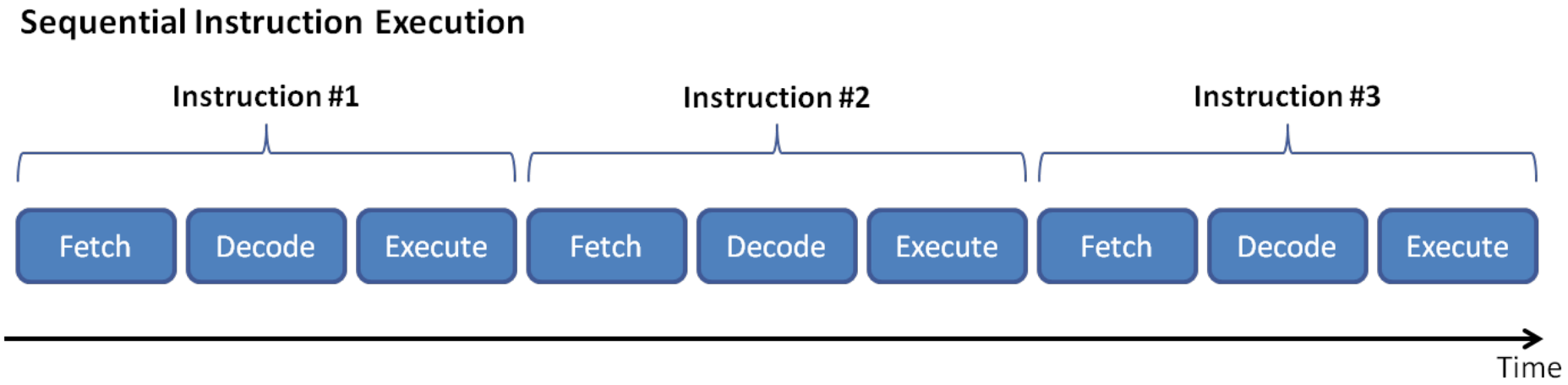
Acknowledgements

- Slide credits:
  - Onur Mutlu

Reminder: CPU Pipelining

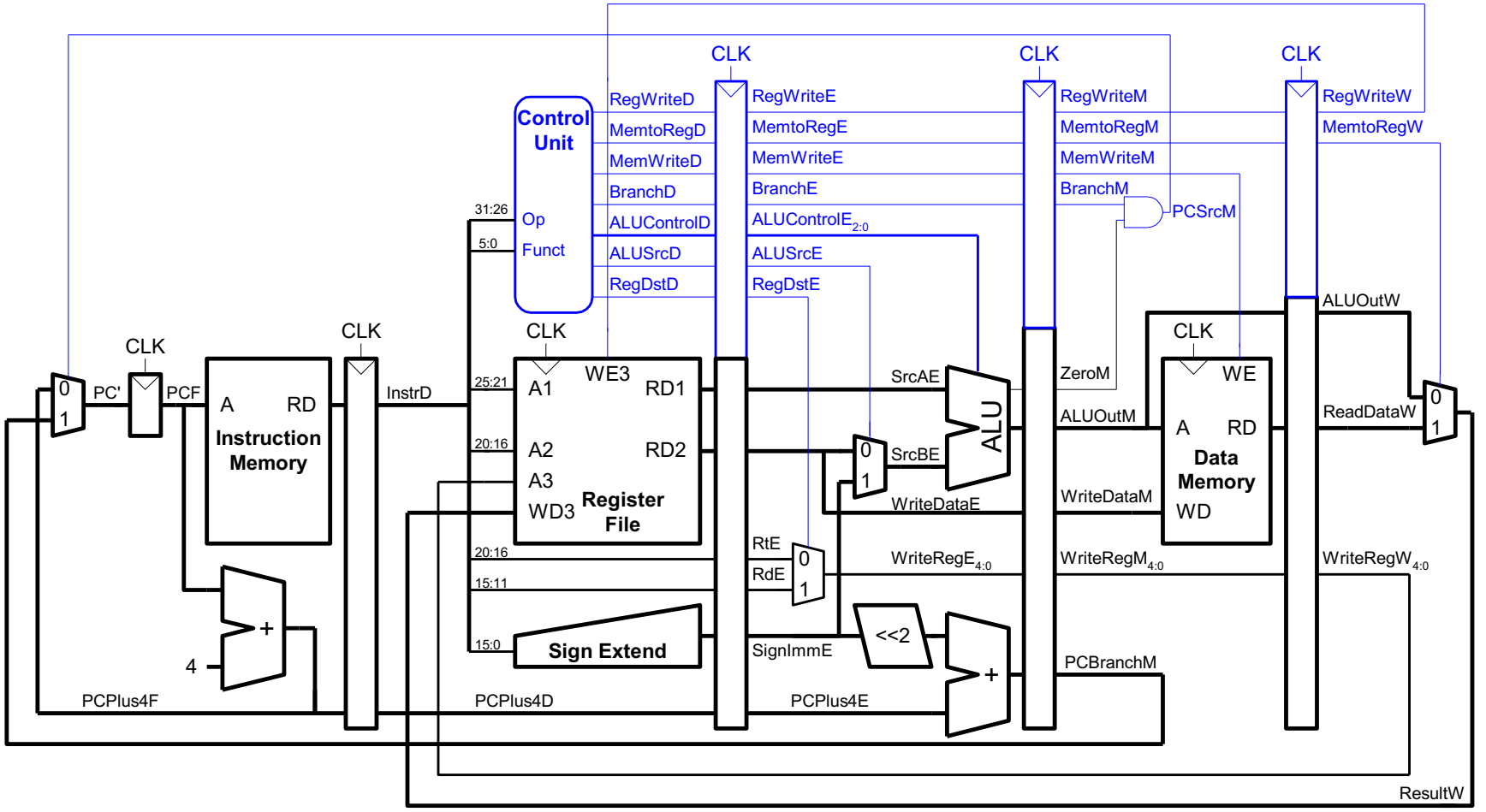
Sequential v.s Pipeline

- Suppose Each Instruction has Fetch, Decode, and Execute steps



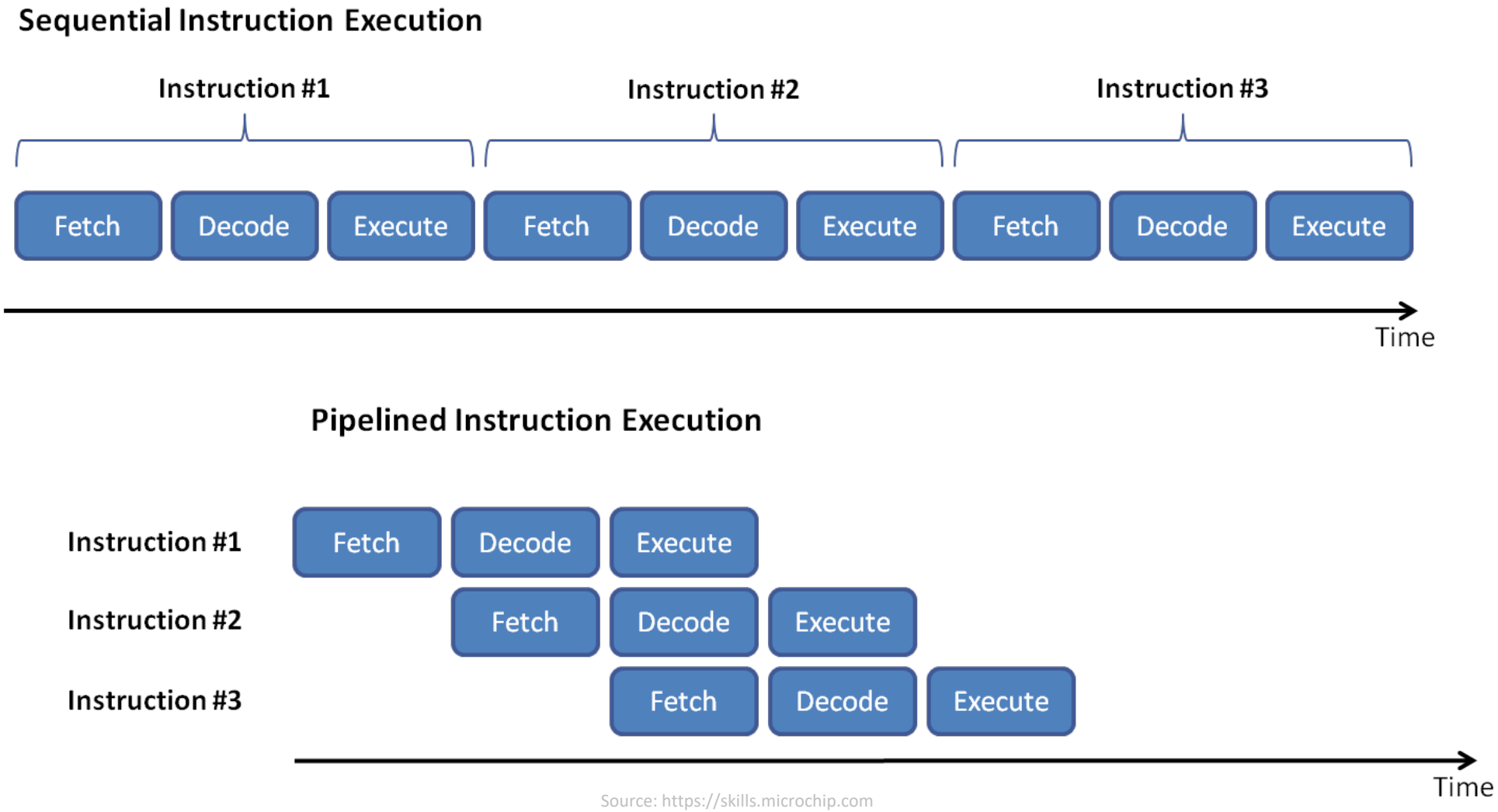
Source: https://skills.microchip.com

MIPS Datapath



Sequential v.s Pipeline

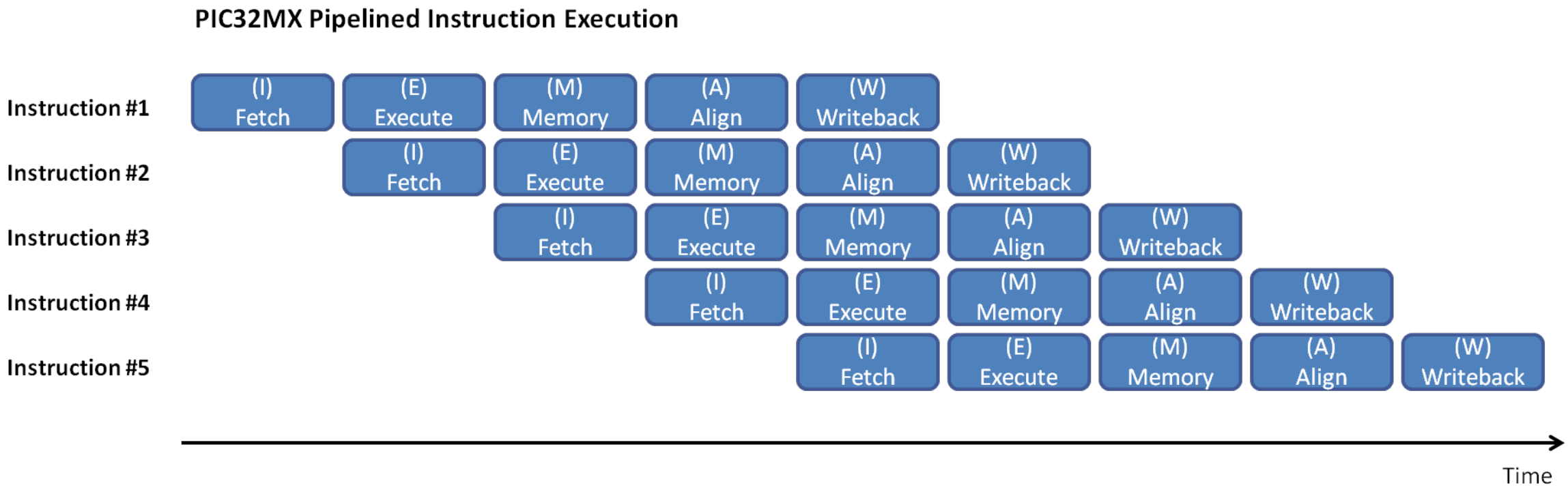
- Suppose Each Instruction has Fetch, Decode, and Execute stejps



Source: https://skills.microchip.com

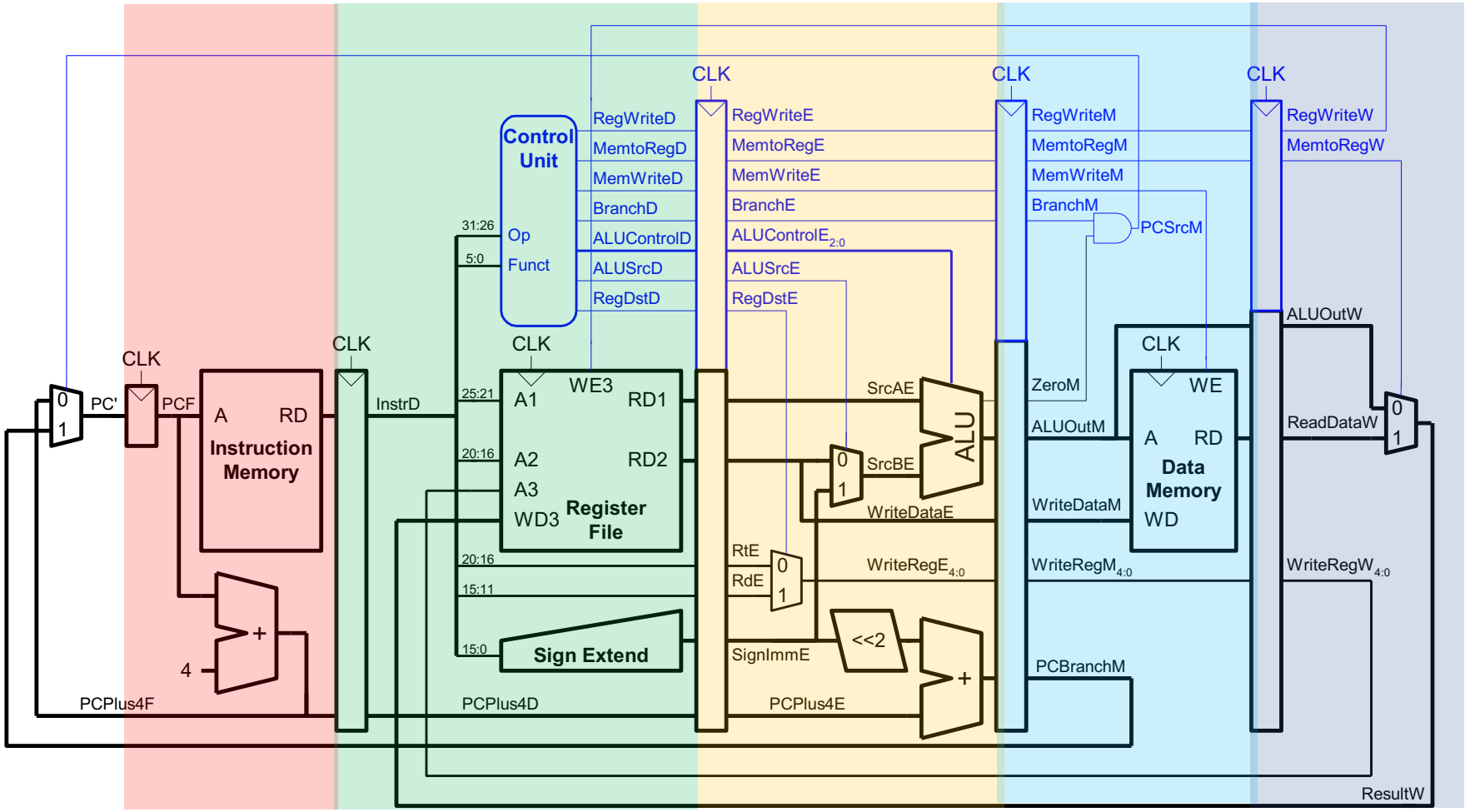
Sequential v.s Pipeline

- More advanced pipeline



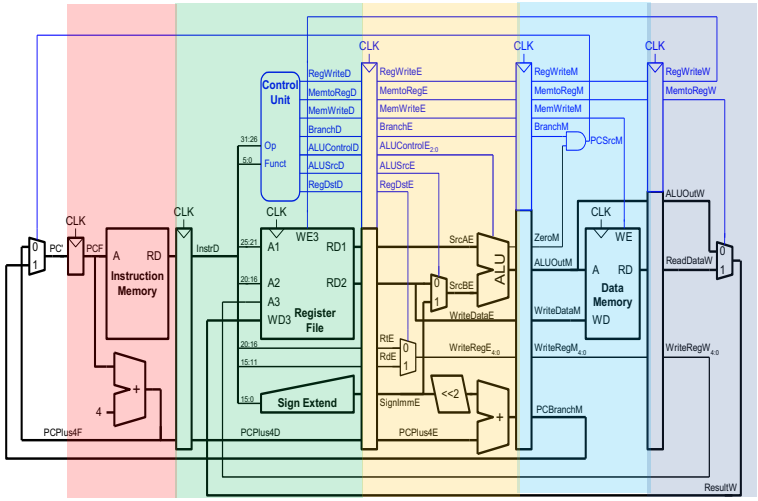
Source: https://skills.microchip.com

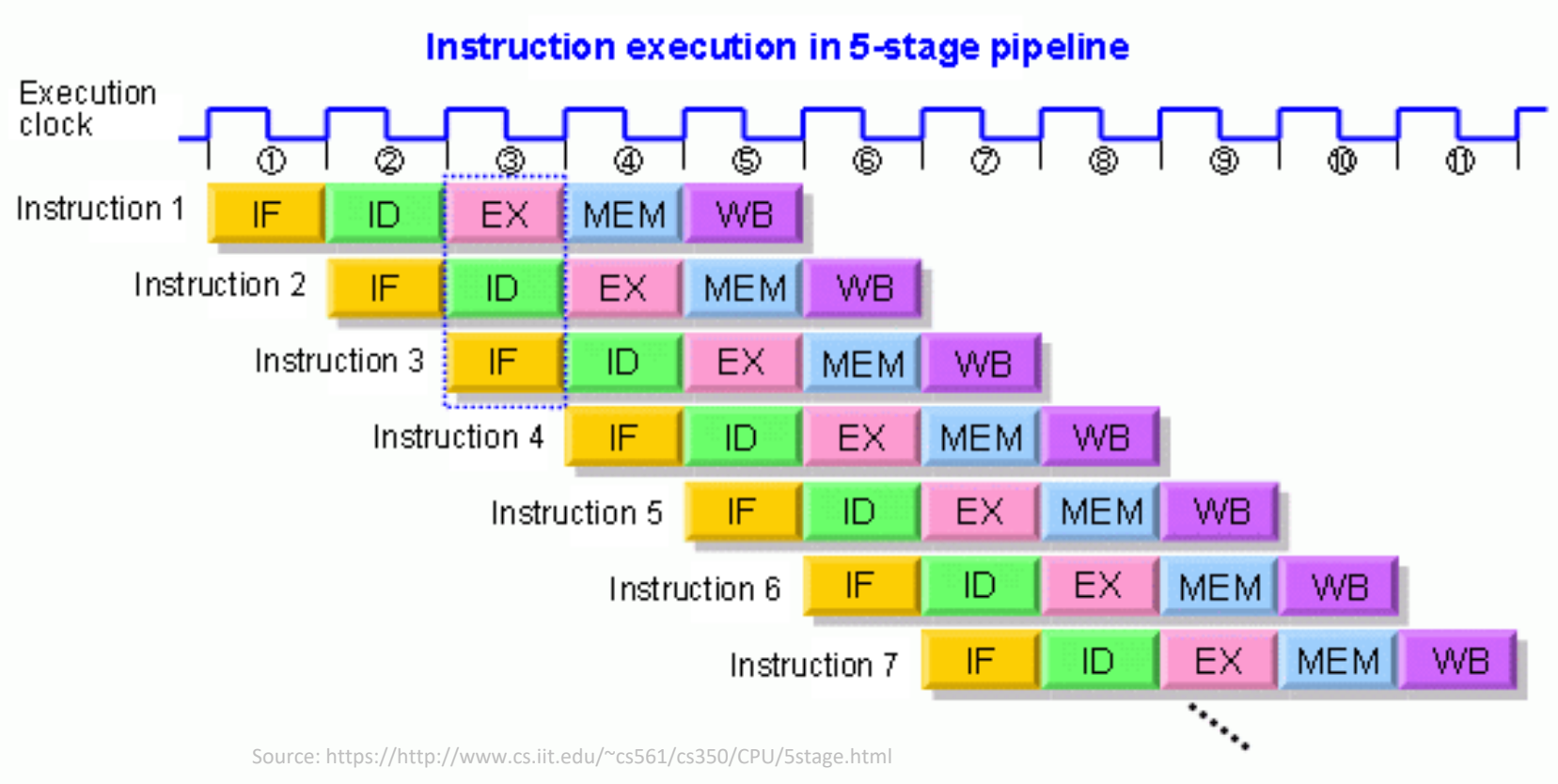
MIPS Pipeline



Fetch Decode Execution Memory Writeback

MIPS Pipeline





Source: https://http://www.cs.iit.edu/~cs561/cs350/CPU/5stage.html

Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—**Very high-speed computers may be classified as follows:**

1) **Single Instruction Stream–Single Data Stream (SISD)**
2) **Single Instruction Stream–Multiple Data Stream (SIMD)**
3) **Multiple Instruction Stream–Single Data Stream (MISD)**
4) **Multiple Instruction Stream–Multiple Data Stream (MIMD).**

"Stream," as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

Representative organizations are selected from each class and the arrangement of the constituents is shown.

Manuscript received June 30, 1966; revised August 16, 1966. This work was performed under the auspices of the U. S. Atomic Energy Commission.

The author is with Northwestern University, Evanston, Ill., and Argonne National Laboratory, Argonne, Ill.

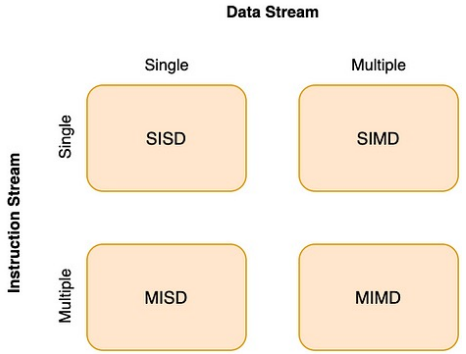
INTRODUCTION

MANY SIGNIFICANT scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially de facto situation

SIMD Architecture

Flynn's Taxonomy

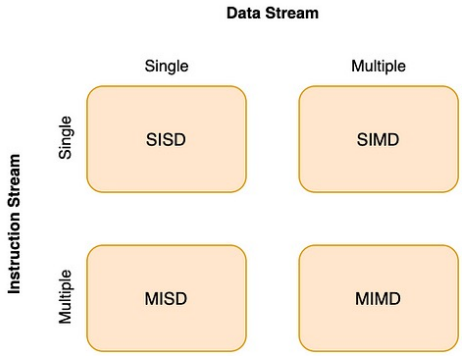
- SISD: Single instruction operates on single data element
  - Example: Most conventional laptops



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

Flynn's Taxonomy

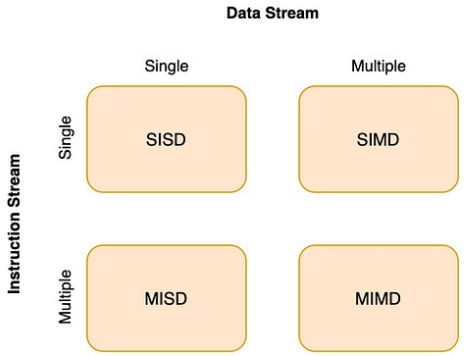
- SISD: Single instruction operates on single data element
  - Example: Most conventional laptops
- SIMD: Single instruction operates on multiple data elements
  - Example: GPUs and Vector Processors



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

Flynn's Taxonomy

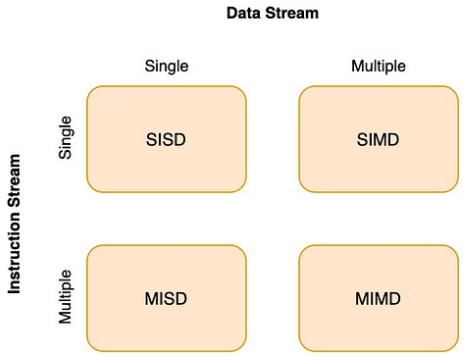
- SISD: Single instruction operates on single data element
  - Example: Most conventional laptops
- SIMD: Single instruction operates on multiple data elements
  - Example: GPUs and Vector Processors
- MISD: Multiple instructions operate on single data element
  - Example: Systolic array processor, streaming processor



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

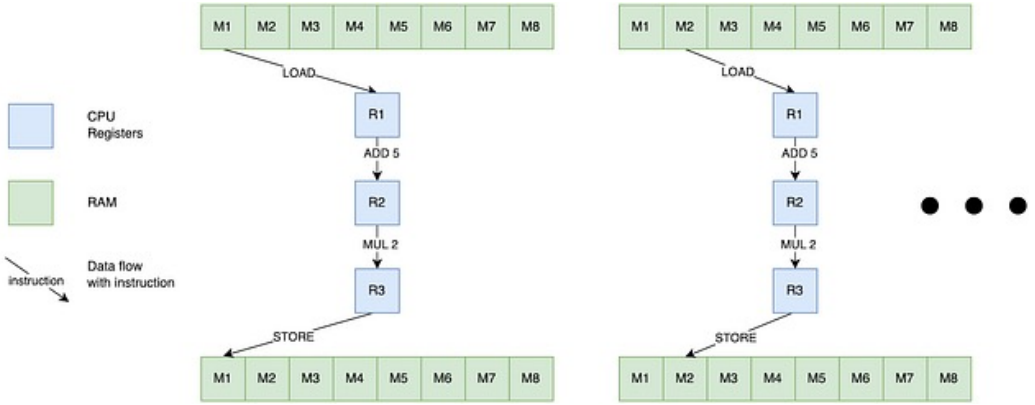
Flynn's Taxonomy

- SISD: Single instruction operates on single data element
  - Example: Most conventional laptops
- SIMD: Single instruction operates on multiple data elements
  - Example: GPUs and Vector Processors
- MISD: Multiple instructions operate on single data element
  - Example: Systolic array processor, streaming processor
- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
  - Supercomputers

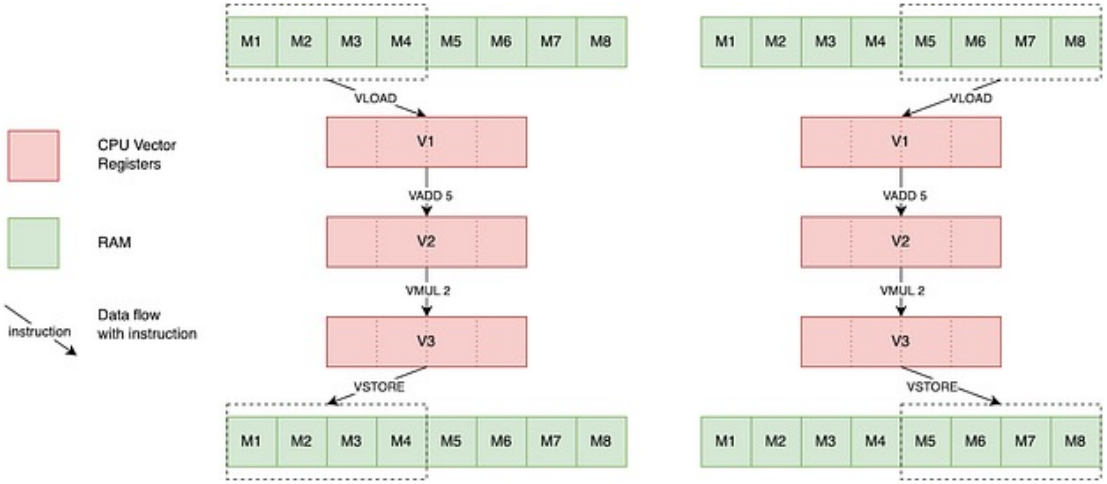


Source: Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

Execution in Scalar vs SIMD Processor



SCALAR



SIMD

Source: https://medium.com/e4r/a-primer-to-simd-architecture-from-concept-to-code-d3cc470d6709

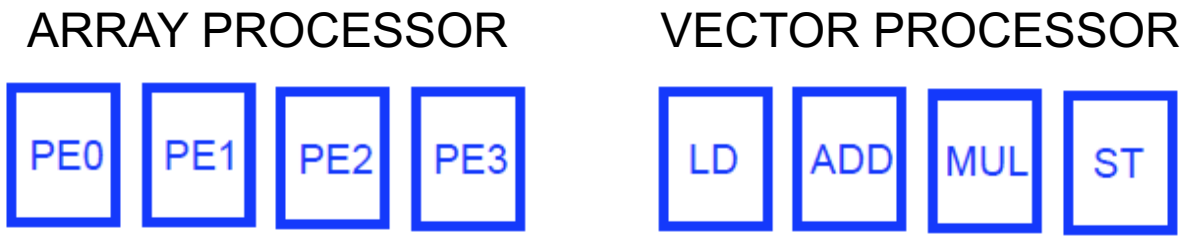
Source of Parallelism

- SIMD:
  - Perform same operation on different data
- Dataflow:
  - Perform multiple operations in parallel (data – driven)
- Thread:
  - Executing multiple threads in parallel

SIMD Parallelism in Time and Space

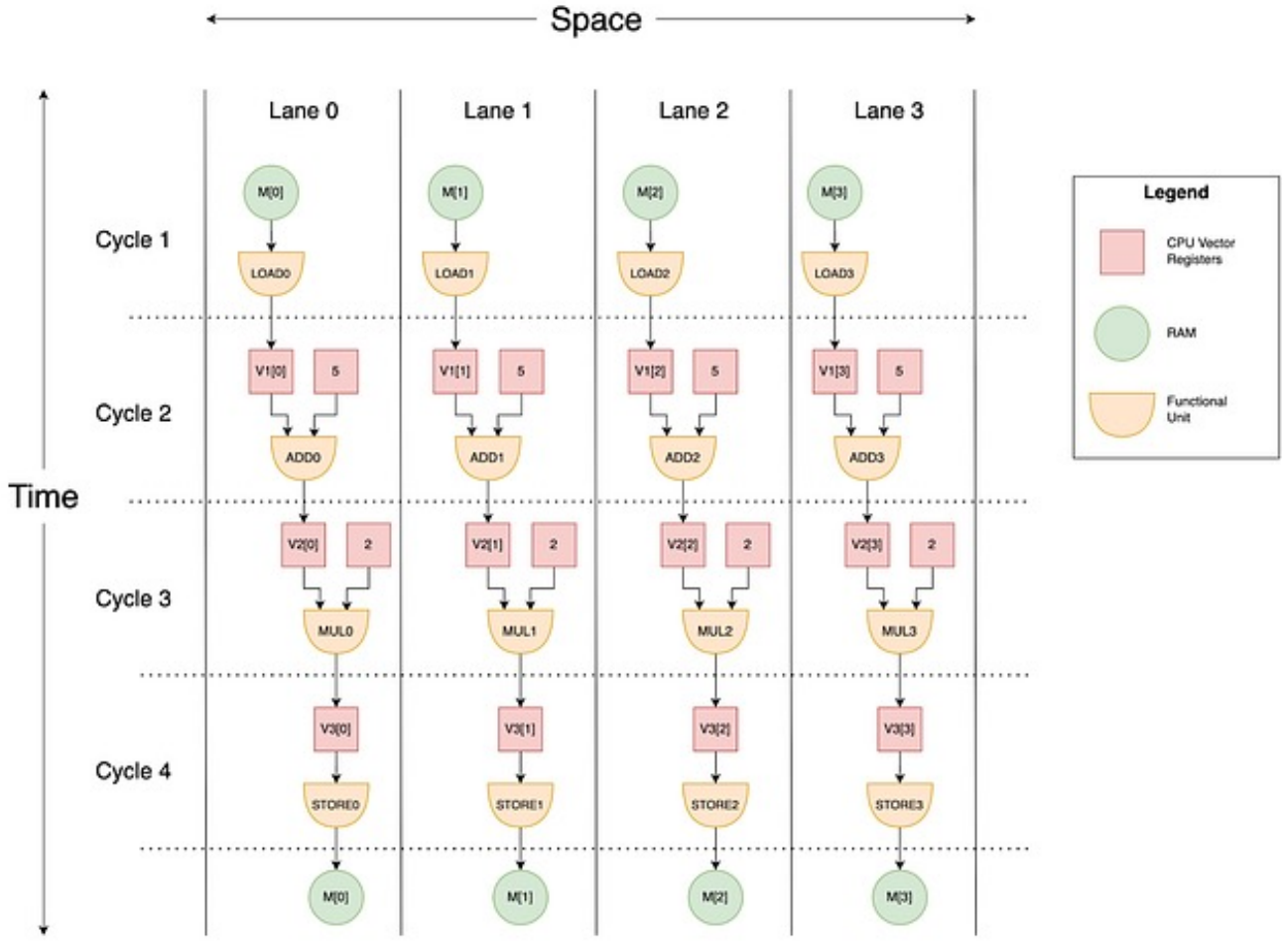
- SIMD: Single instruction operates on multiple data elements
  - In time (Same processing element reused for next data)
  - In space (Multiple processing elements operate on the same data)

- Space (Array processor): Instruction operates on multiple data elements at the same time using different (PEs)
- Time (Vector processor): Instruction operates on multiple data elements in consecutive time steps using the same space (PE)



Array Processor

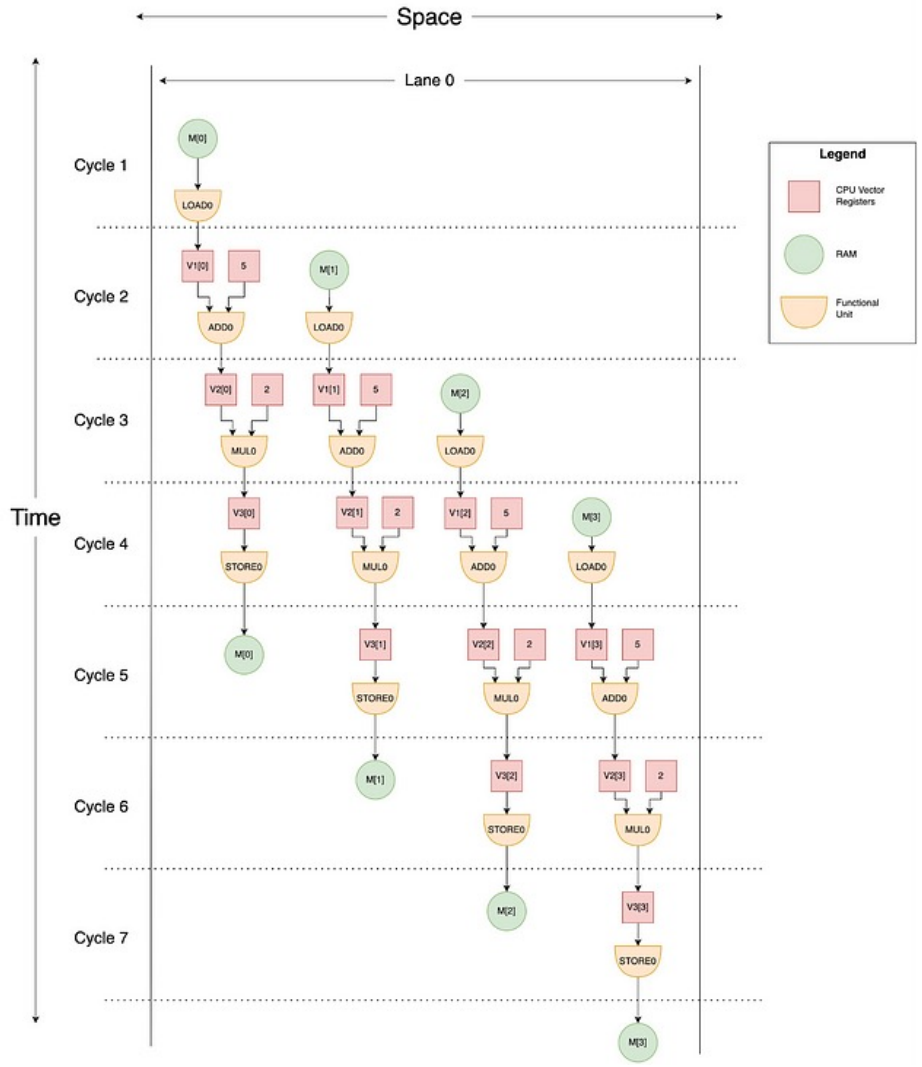
- **Multiple Functional Units**: Contains multiple sets of **ALUs, Floating Point Units, Load/Store Units**, etc.
- **Distributed Across Lanes**:
  - Each **lane** has its own set of functional units.
  - Lanes operate **independently** on different data elements (SIMD-style).
- **Distributed Register File**:
  - Registers are **split among lanes** instead of a single shared register file.
  - Each lane accesses only **its own portion** of the register file.



Source: https://medium.com/e4r/a-primer-to-simd-architecture-from-concept-to-code-d3cc470d6709

Vector Processor

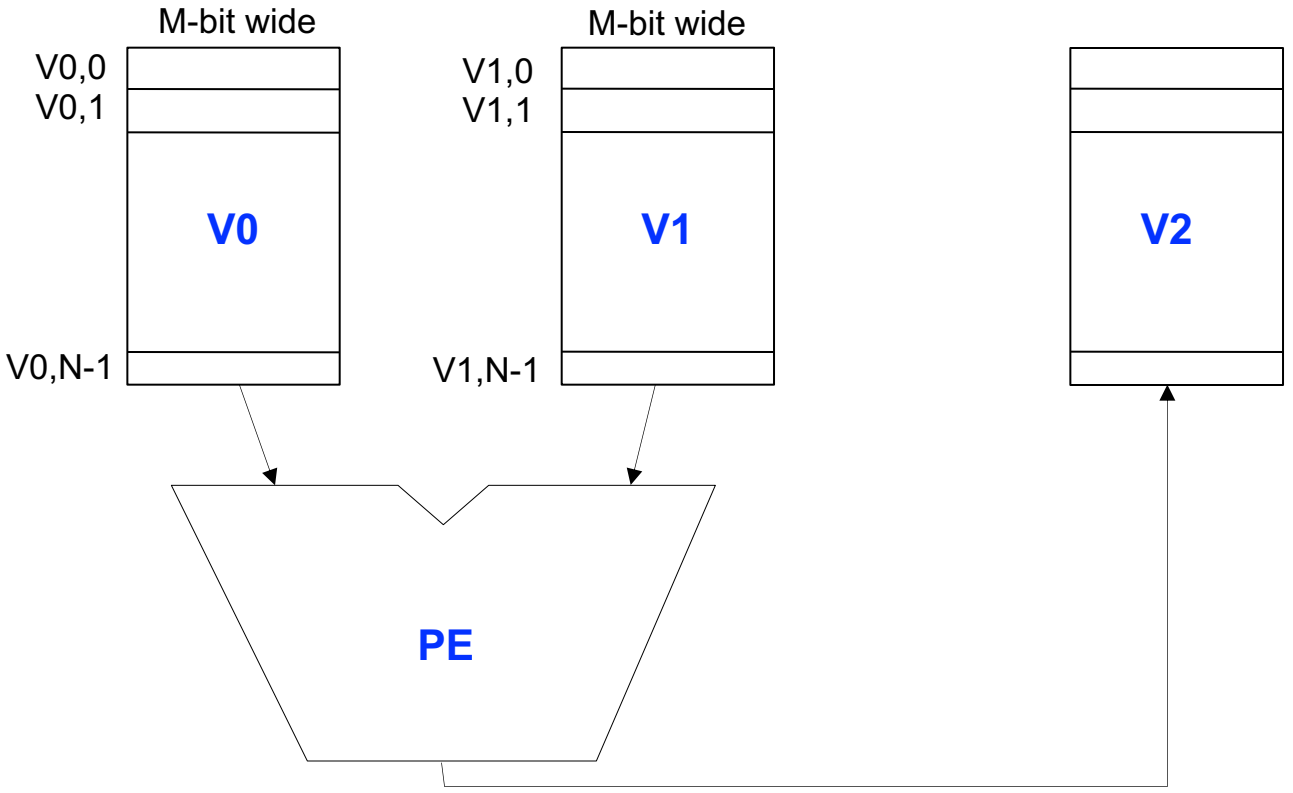
- A vector processor benefits from pipelining, where a single vector instruction processes multiple data elements in stages.
- Unlike an array processor, a vector processor typically has fewer lanes.
- A vector processor uses deep pipelining within functional units to achieve high throughput.



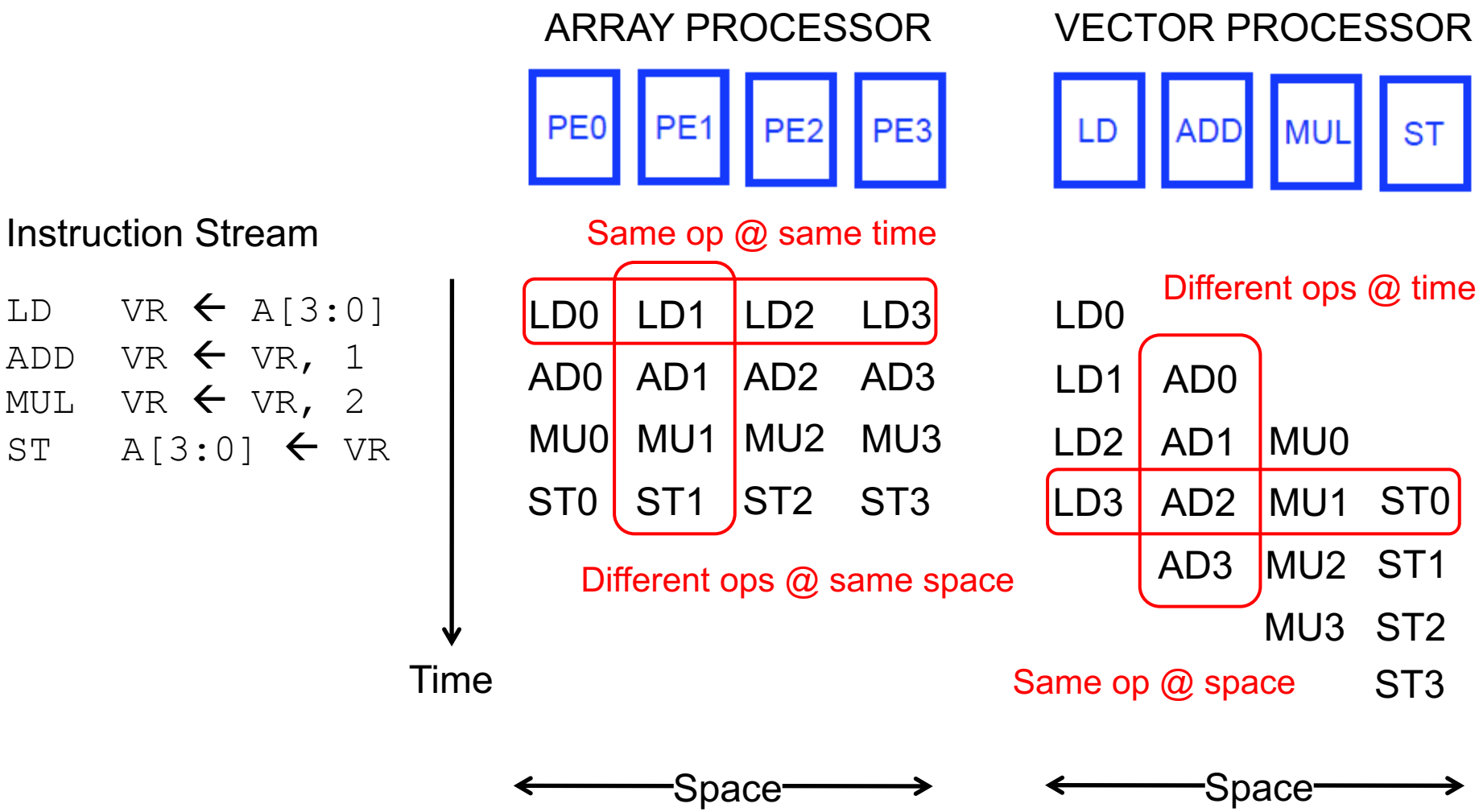
Source: https://medium.com/e4r/a-primer-to-simd-architecture-from-concept-to-code-d3cc470d6709

SIMD Vector Registers

- Each vector data register holds N x M-bit values
  - Each register stores a vector

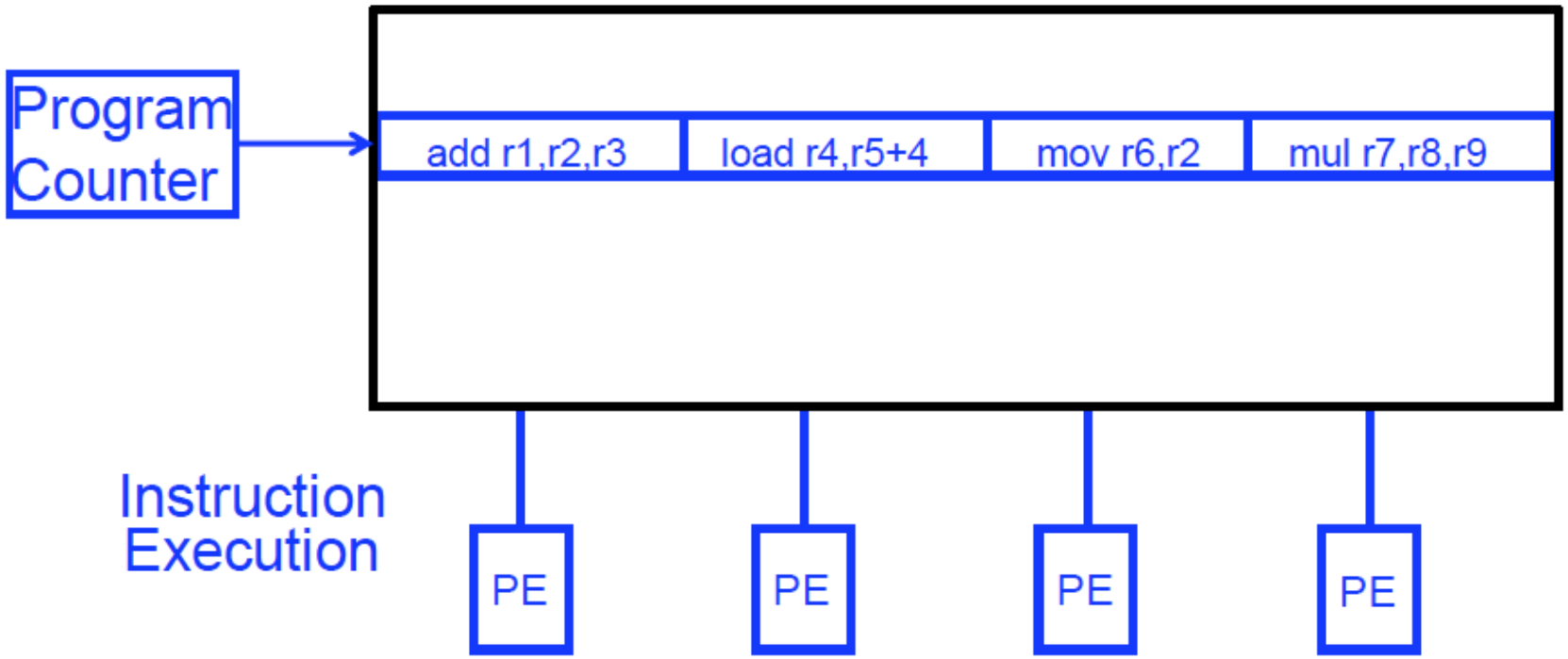


Array vs. Vector Processor



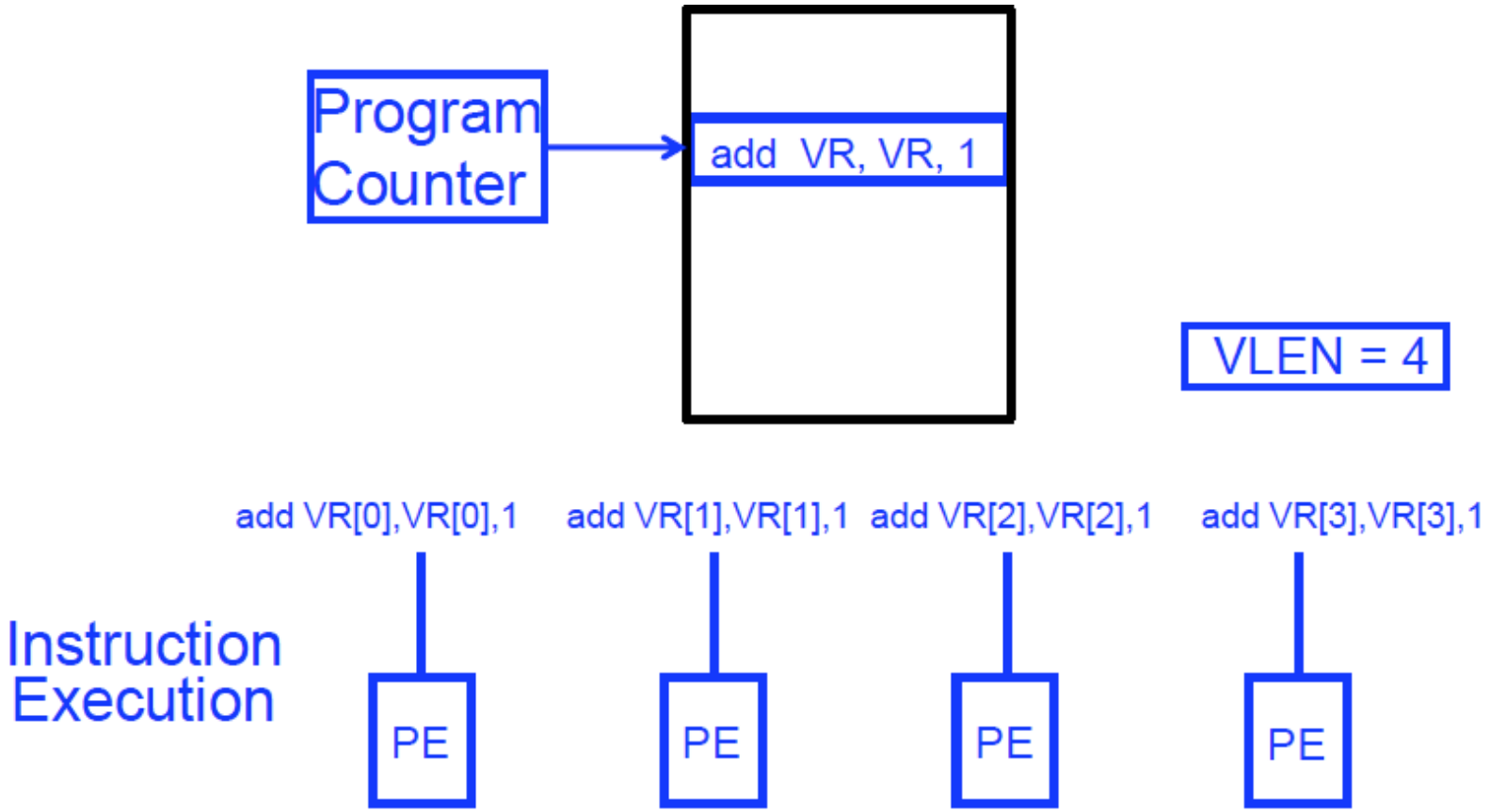
SIMD Array Processing vs. VLIW

- VLIW



SIMD Array Processing vs. VLIW

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



Vector Processors

- A vector processor is one whose instructions operate on **vectors** rather than **scalar** values.
  - Performs an operation on each element in consecutive cycles
    - Vector functional units are pipelined
    - Each pipeline stage operates on a different data element

```
for (let i = 0 to 49) {
  C[i] = (A[i] + B[i]) / 2
}
```

Vector Processors

- A vector processor is one whose instructions operate on **vectors** rather than **scalar** values.
  - 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 easy loading (or even early loading, i.e., prefetching) of vectors into registers/cache/memory

```
for (let i = 0 to 49) {
  C[i] = (A[i] + B[i]) / 2
}
```

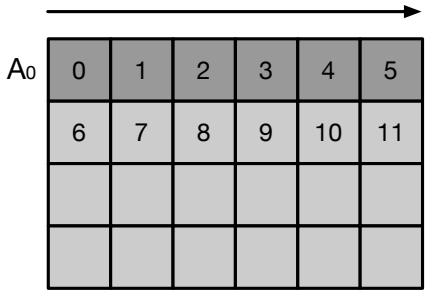
Vector Processors

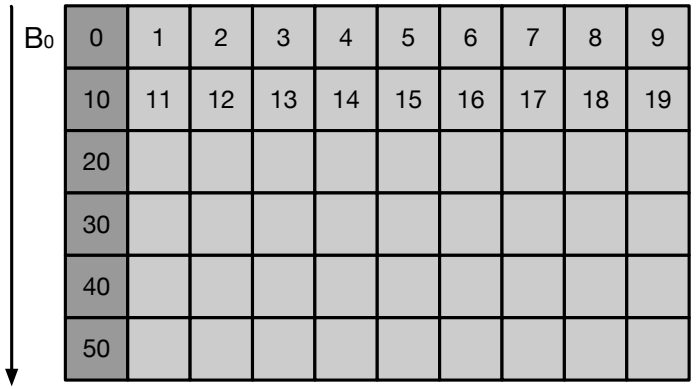
- Requirements
  - Need to **load/store** vectors.
    - Need **vector registers.**
  - Need to operate on **vectors** of different lengths.
    - Need **vector length** register (**VLEN**).
  - Elements of a vector might be stored apart from each other in memory
    - Need **vector stride** register (**VSTR**)
    - Stride: distance in memory between two elements of a vector

```
for (let i = 0 to 49) {
  C[i] = (A[i] + B[i]) / 2
}
```

Vector Stride Example, Matrix Multiply

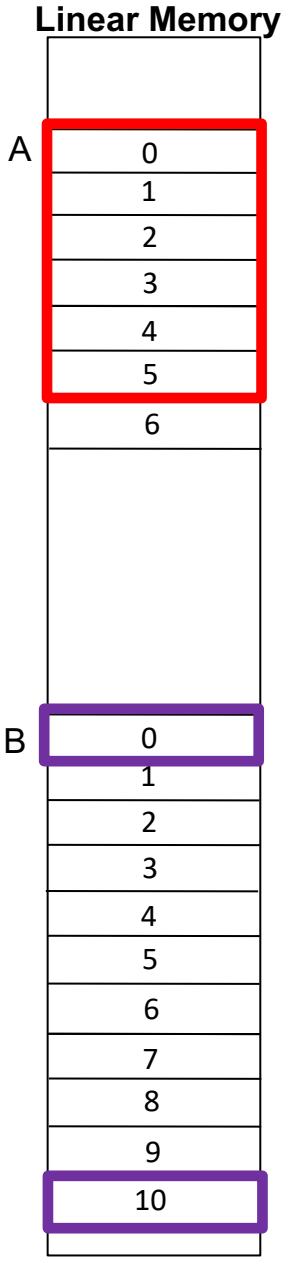
- A and B matrices, both stored in memory in row-major order
- Load A's row 0 (A0 through A5) into vector register V_1
  - Each time, increment address by 1 to access the next column
  - Accesses have a stride of 1
- Load B's column 0 (B0 through B50) into vector register V_2
  - Each time, increment address by 10 to access the next row
  - Accesses have a stride of 10





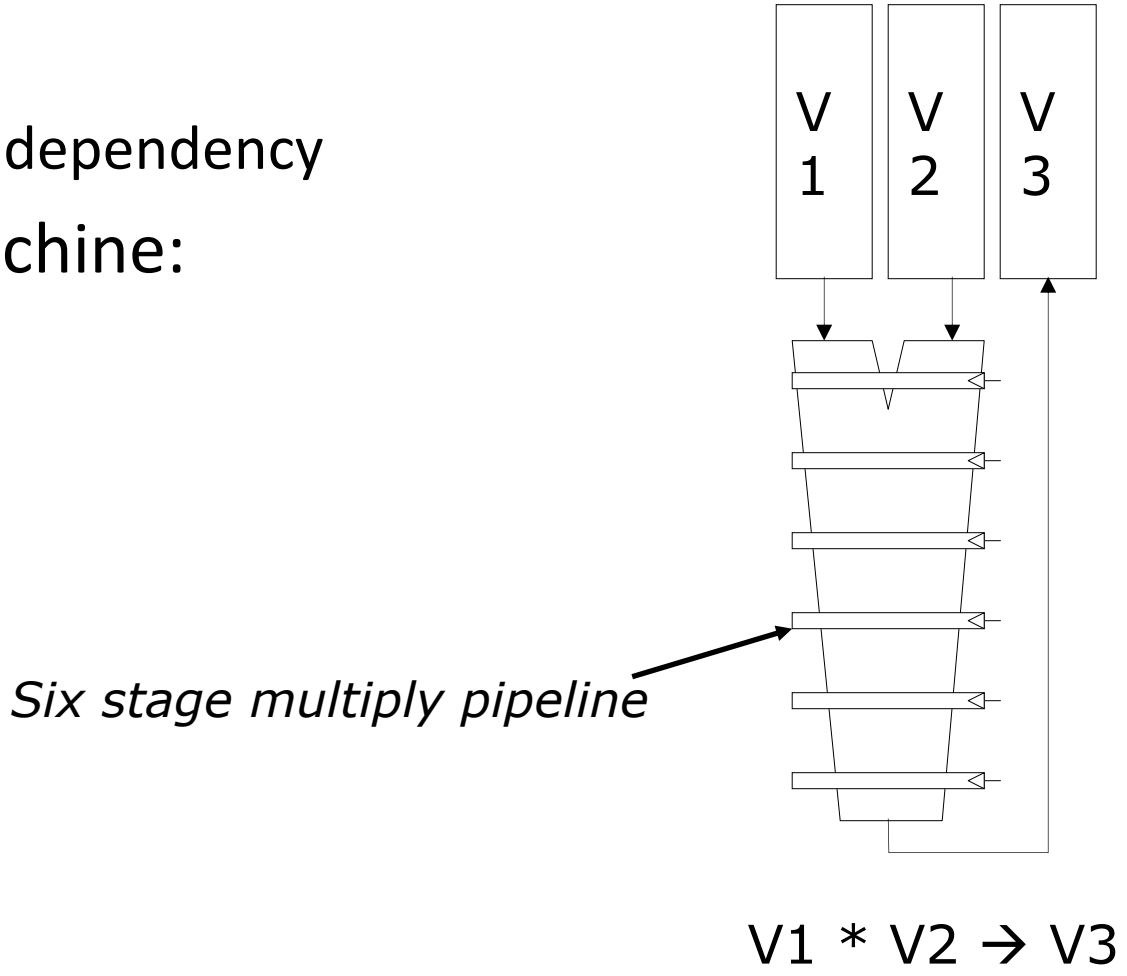
Dot product of each row vector of A with each column vector of B

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



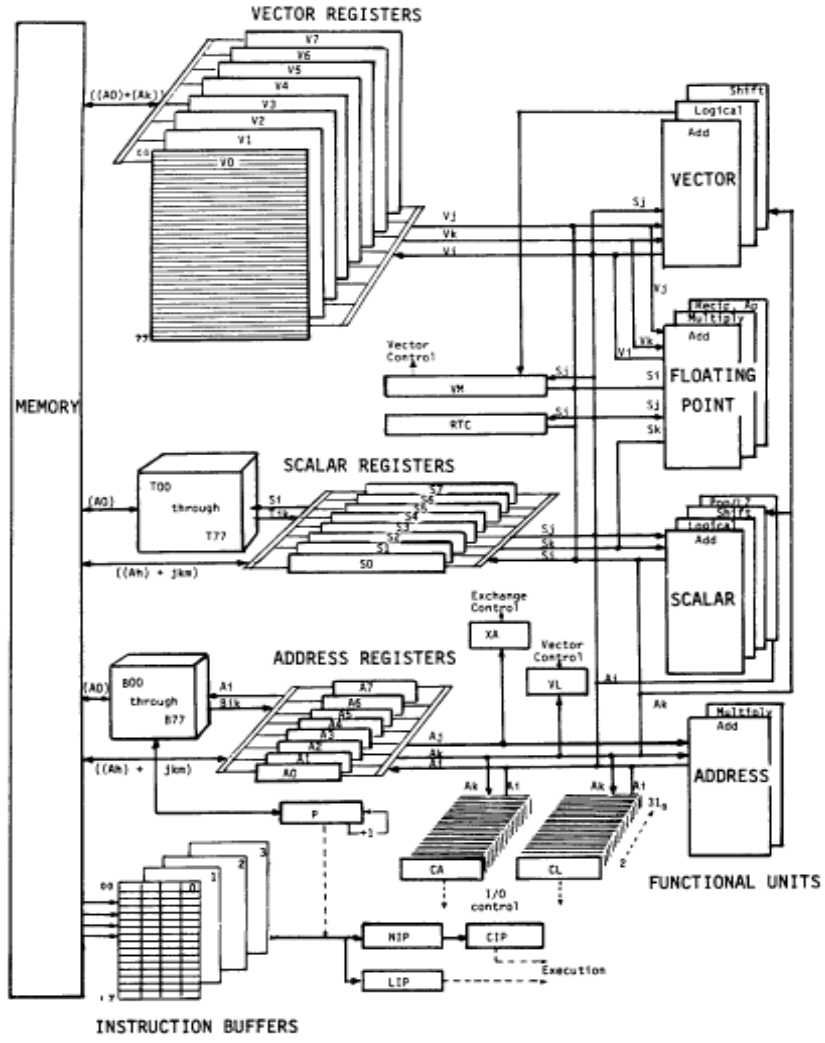
Vector Processor Functional Unit

- A deep pipeline
  - Fast clock cycle
  - Simple control due to lack of data dependency
- Example of Vector Processor Machine:
  - Cray-1, 1978



Vector Machine Organization (CRAY-1)

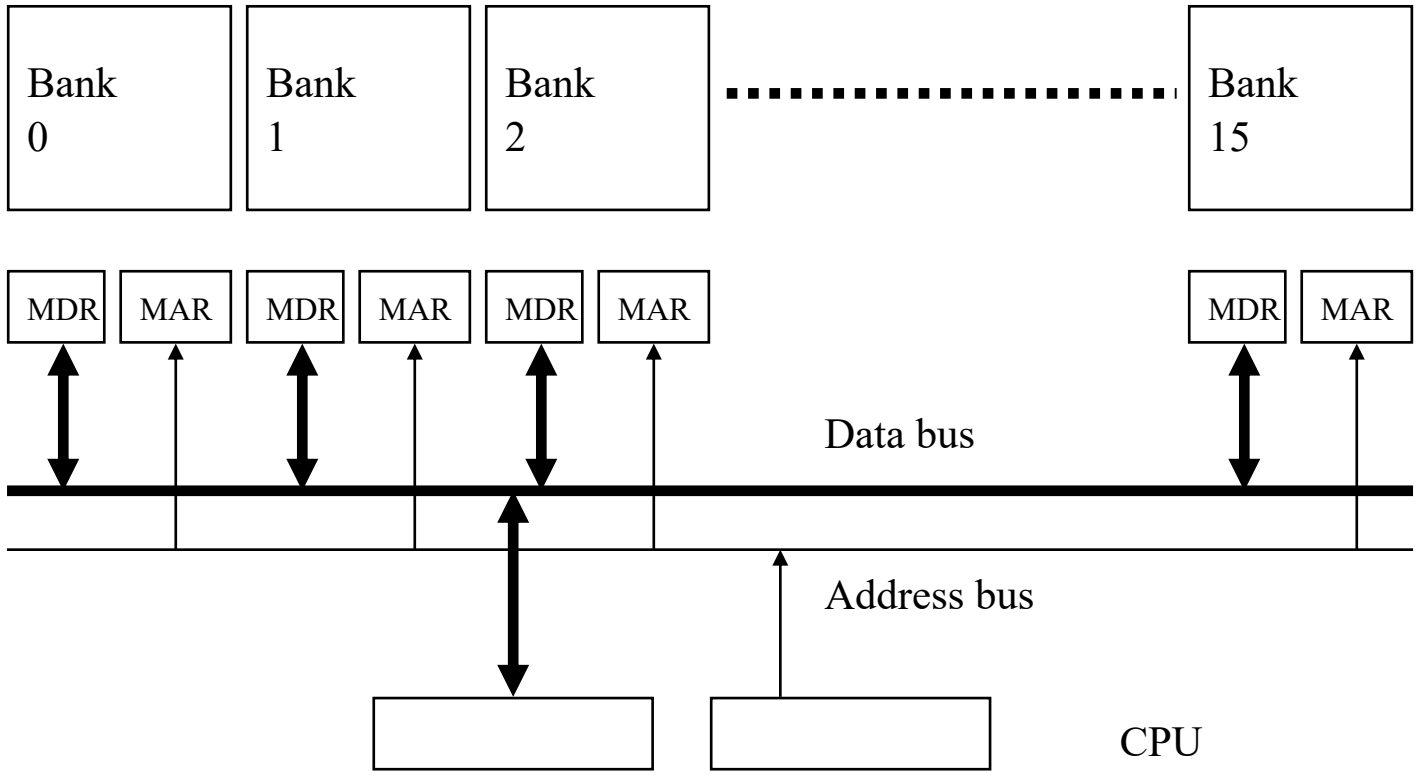
- CRAY-1
  - Scalar and vector modes
  - 8 64-element vector registers
  - 64 bits per element
  - 16 memory banks
  - 8 24-bit address registers



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

Memory Banking

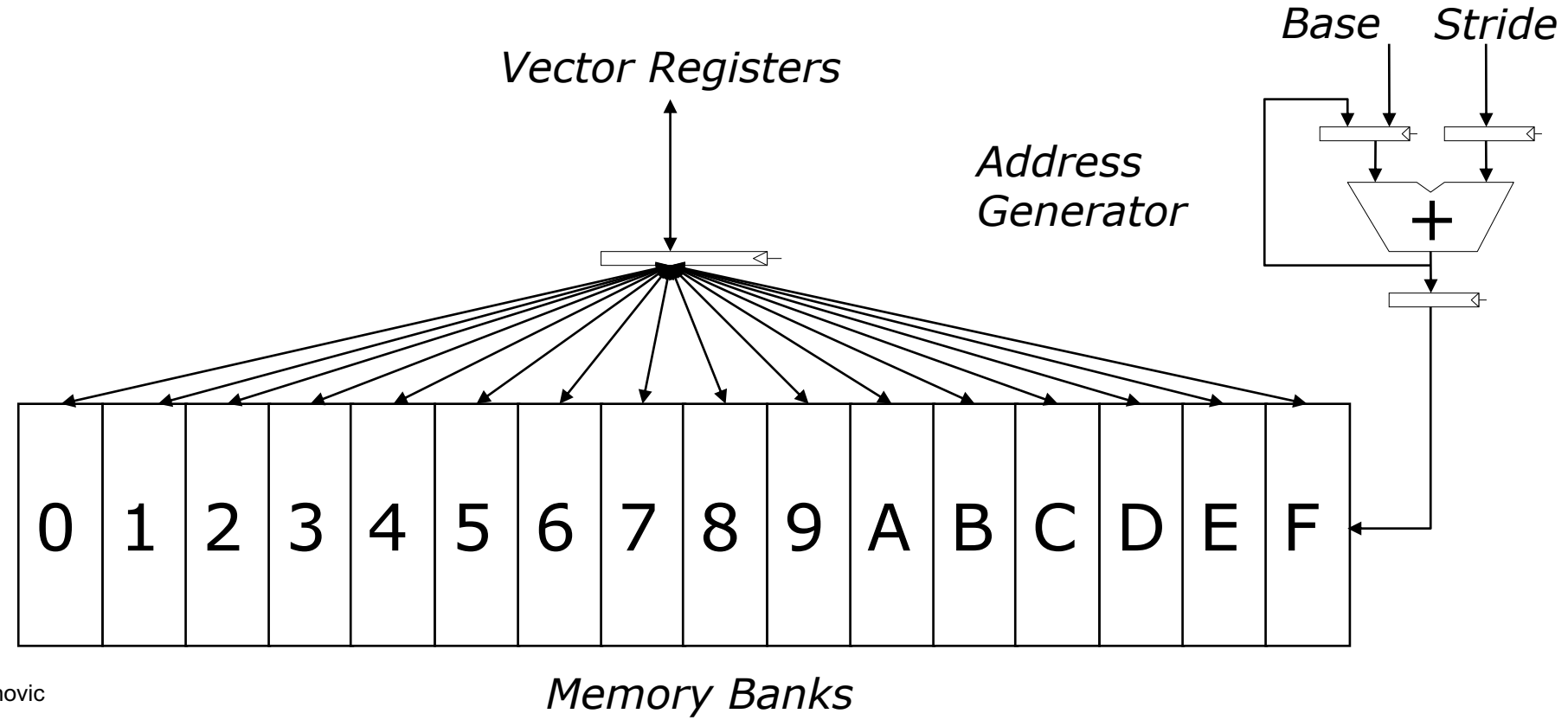
- Memory is divided into banks that can be accessed independently; banks share address and data buses (to reduce memory chip pins)
- Can start and complete one bank access per cycle
- Can sustain N concurrent accesses if all N go to different banks



Picture credit: Krste Asanovic

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

- Execution time
  - in-order with one memory bank:
    - 4 + 50 * 40 = 2004 cycles

```
for (let i = 0 to 49) {
  C[i] = (A[i] + B[i]) / 2
}
```

304 dynamic instructions

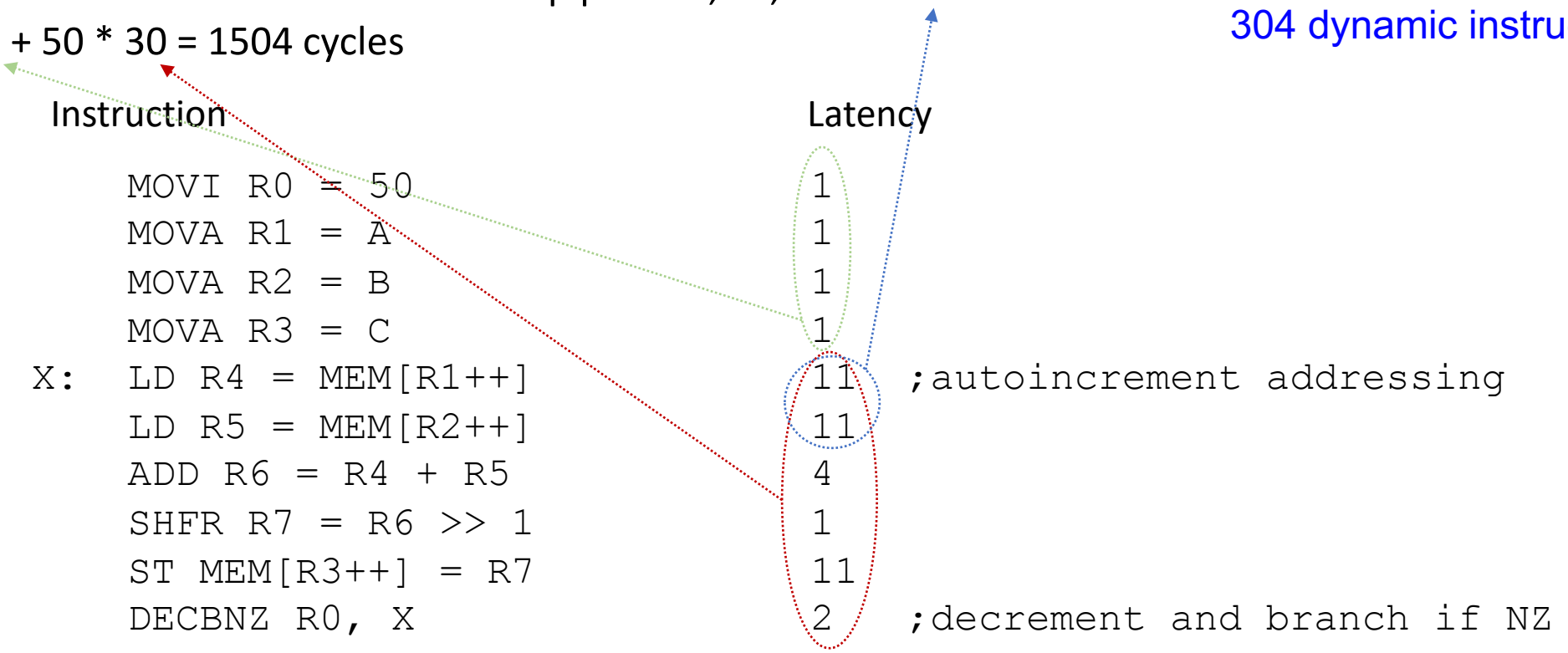
Instruction		Latency	
	MOVI R0 = 50	1	
	MOVA R1 = A	1	
	MOVA R2 = B	1	
	MOVA R3 = C	1	
X:	LD R4 = MEM[R1++]	11	;autoincrement addressing
	LD R5 = MEM[R2++]	11	
	ADD R6 = R4 + R5	4	
	SHFR R7 = R6 >> 1	1	
	ST MEM[R3++] = R7	11	
	DECBNZ R0, X	2	;decrement and branch if NZ

Scalar Code Example

- Execution time
  - in-order with one memory bank:
    - 4 + 50 * 40 = 2004 cycles
  - with two memory banks:
    - The two load instructions can be pipelined, i.e, 1+11 = 12
    - 4 + 50 * 30 = 1504 cycles

```
for (let i = 0 to 49) {
  C[i] = (A[i] + B[i]) / 2
}
```

304 dynamic instructions



Vector Processor

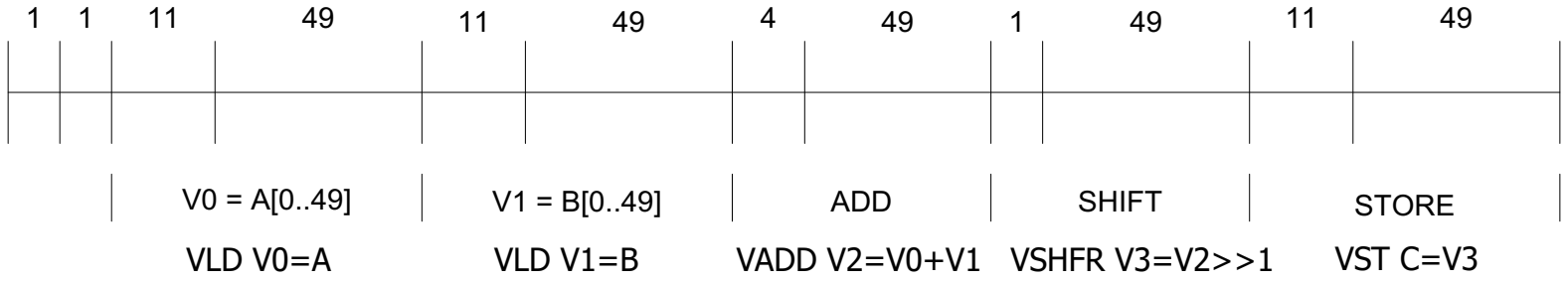
- A loop is vectorizable if each iteration is independent of any other
- Assuming no data forwarding, 16 banks
  - 16 (>11) banks ensures there are enough banks to overlap enough memory operations
    - **Total: 285 cycles**

```
for (let i = 0 to 49) {
  C[i] = (A[i] + B[i]) / 2
}
```

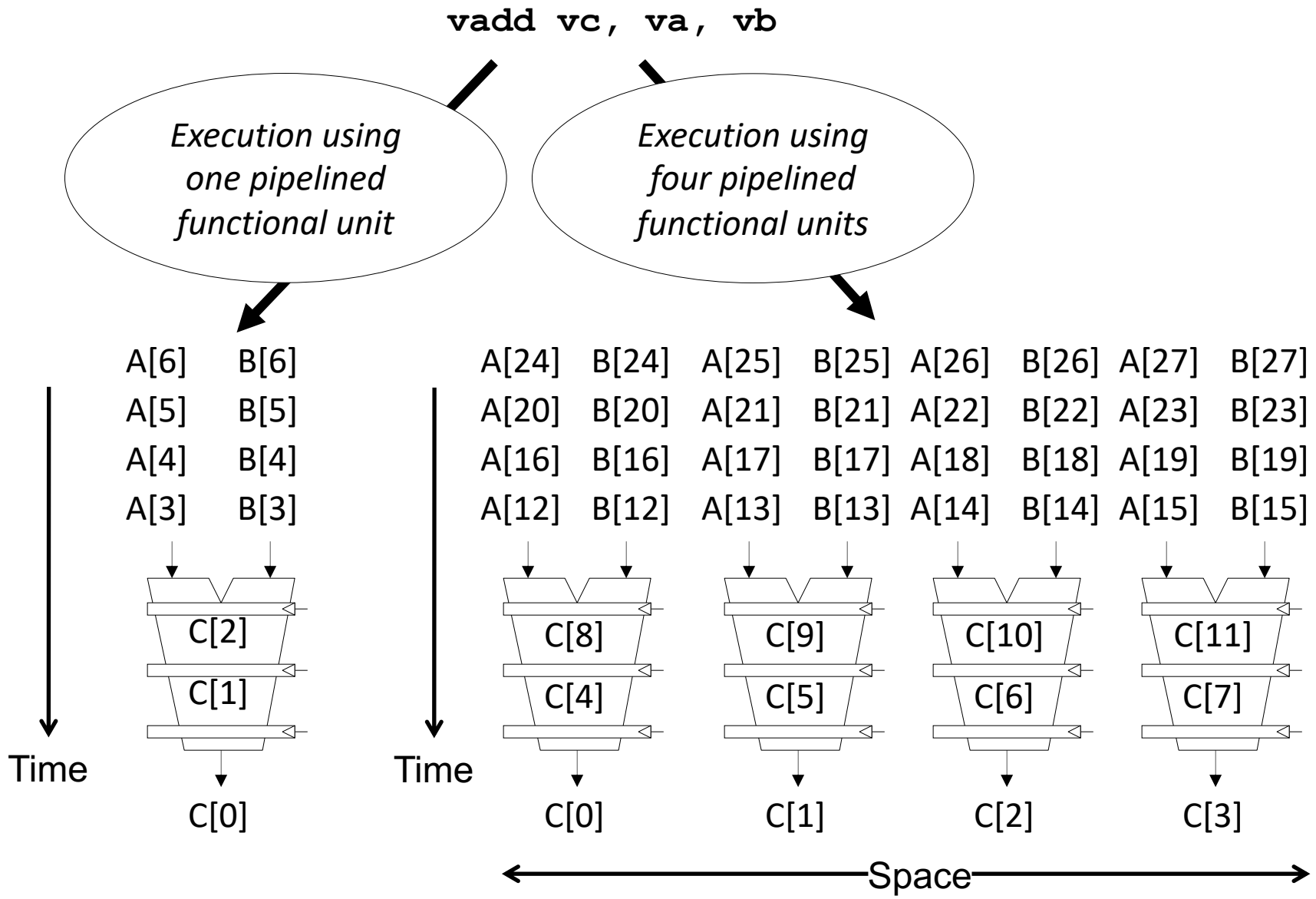
7 dynamic instructions

Instruction	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

11 cycles initial delay + one element per cycle

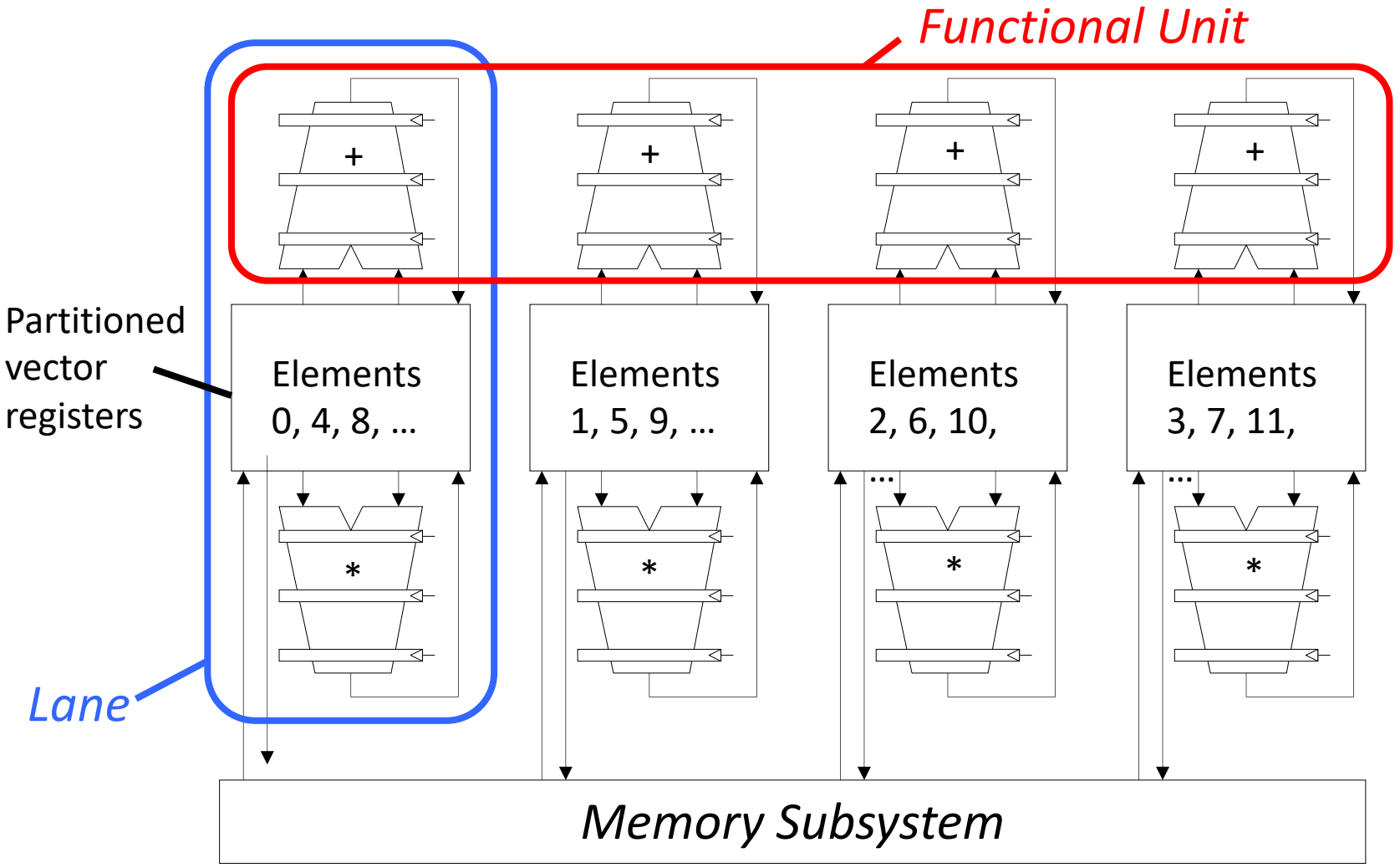


Vector Instruction Execution



Slide credit: Krste Asanovic

Vector Unit Structure

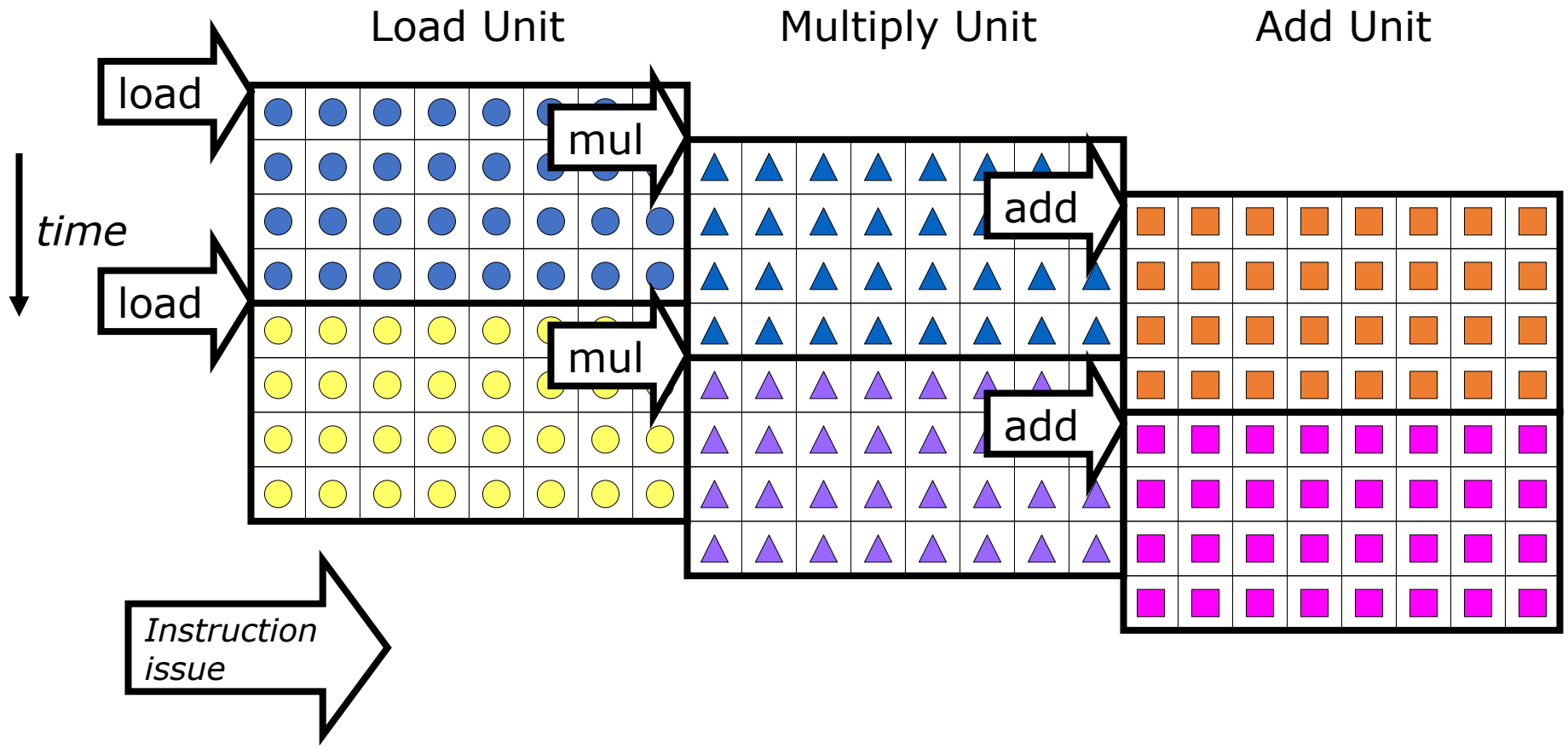


Slide credit: Krste Asanovic

Vector Instruction Level Parallelism

```
vld  v1
vmul v3,v1,v2
vadd v5,v3,v4
```

- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes



Complete 24 operations/cycle while issuing 1 short instruction/cycle

Slide credit: Krste Asanovic

Vector Processor

- Advantages:
  - No dependencies within a vector
    - Easy to pipeline and parallelize

Vector Processor

- Advantages:
  - No dependencies within a vector
    - Easy to pipeline and parallelize
  - Each instruction generates a lot of work (i.e., operations)
    - Amortizes instruction fetch and control overhead over many data

Vector Processor

- Advantages:
  - No dependencies within a vector
    - Easy to pipeline and parallelize
  - Each instruction generates a lot of work (i.e., operations)
    - Amortizes instruction fetch and control overhead over many data
  - No need to explicitly code loops
    - Fewer branches in the instruction sequence

Vector Processor

- Advantages:
  - No dependencies within a vector
    - Easy to pipeline and parallelize
  - Each instruction generates a lot of work (i.e., operations)
    - Amortizes instruction fetch and control overhead over many data
  - No need to explicitly code loops
    - Fewer branches in the instruction sequence
  - Highly regular memory access pattern

Vector Processor

- Advantages:
  - No dependencies within a vector
    - Easy to pipeline and parallelize
  - Each instruction generates a lot of work (i.e., operations)
    - Amortizes instruction fetch and control overhead over many data
  - No need to explicitly code loops
    - Fewer branches in the instruction sequence
  - Highly regular memory access pattern
- Disadvantages
  - Works (only) if parallelism is regular (data/SIMD parallelism)
  - Memory (bandwidth) can easily become a bottleneck
    - A lot of data is accessed

Vector Processor

- Advantages:
  - No dependencies within a vector
    - Easy to pipeline and parallelize
  - Each instruction generates a lot of work (i.e., operations)
    - Amortizes instruction fetch and control overhead over many data
  - No need to explicitly code loops
    - Fewer branches in the instruction sequence
  - Highly regular memory access pattern
- Disadvantages
  - Works (only) if parallelism is regular (data/SIMD parallelism)
  - Memory (bandwidth) can easily become a bottleneck
    - A lot of data is accessed

Many existing ISAs include SIMD operations

Intel MMX/SSEn/AVX/AMX, PowerPC AltiVec, ARM Advanced SIMD, MIPS SIMD, …

GPU (Graphics Processing Units)

GPUs are SIMD Engines

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)
- However, the programming is done using threads, NOT SIMD instructions

GIGABYTE GeForce RTX 4070 WINDFORCE OC 12G Graphics Card, 3X WINDFORCE Fans, 12GB 192-bit GDDR6X, GV-N4070WF3OC-12GD Video Card

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), …

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, …

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)

Exploiting Parallelism

- Let's examine three programming options:
  - Sequential (SISD)
  - Data-Parallel (SIMD)
  - Multithreaded (MIMD/SPMD)

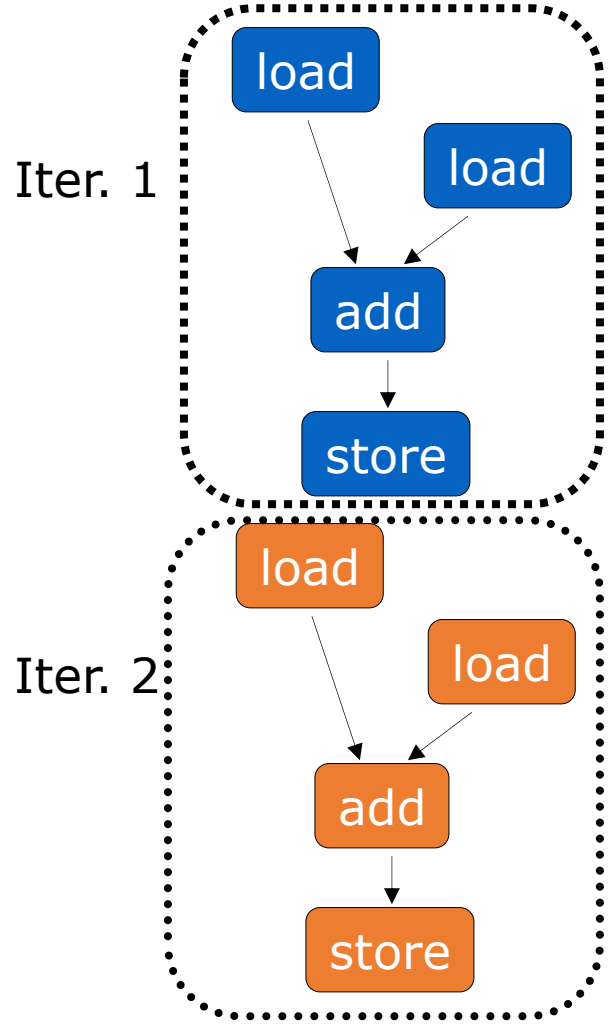
```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Model 1: SISD

- Can be executed on:
  - Pipelined processor

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Scalar Sequential Code

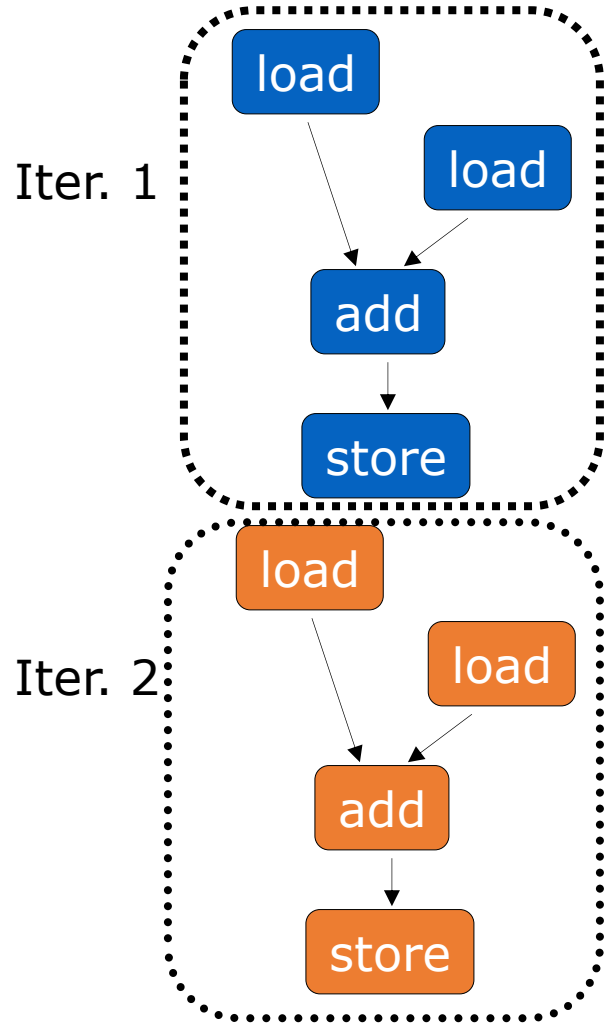


Model 1: SISD

- Can be executed on:
  - 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

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Scalar Sequential Code

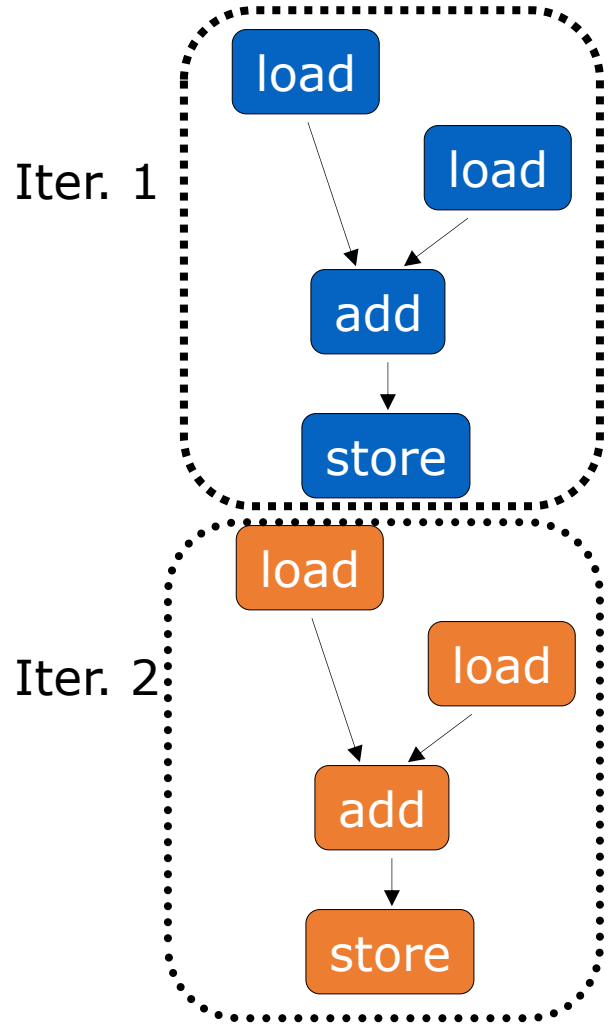


Model 1: SISD

- Can be executed on:
  - 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

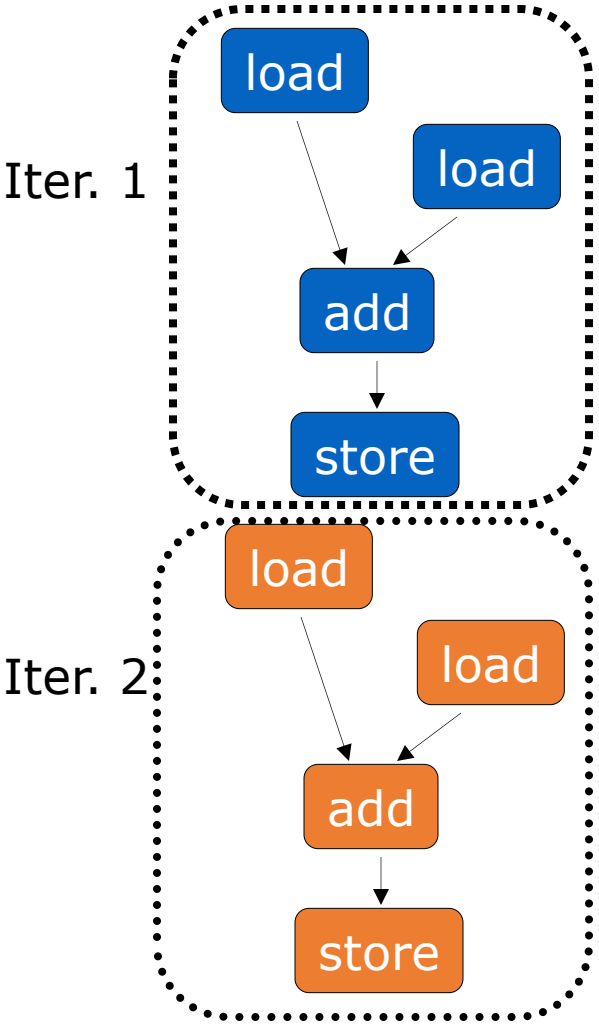
```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Scalar Sequential Code



Model 2: SIMD

Scalar Sequential Code

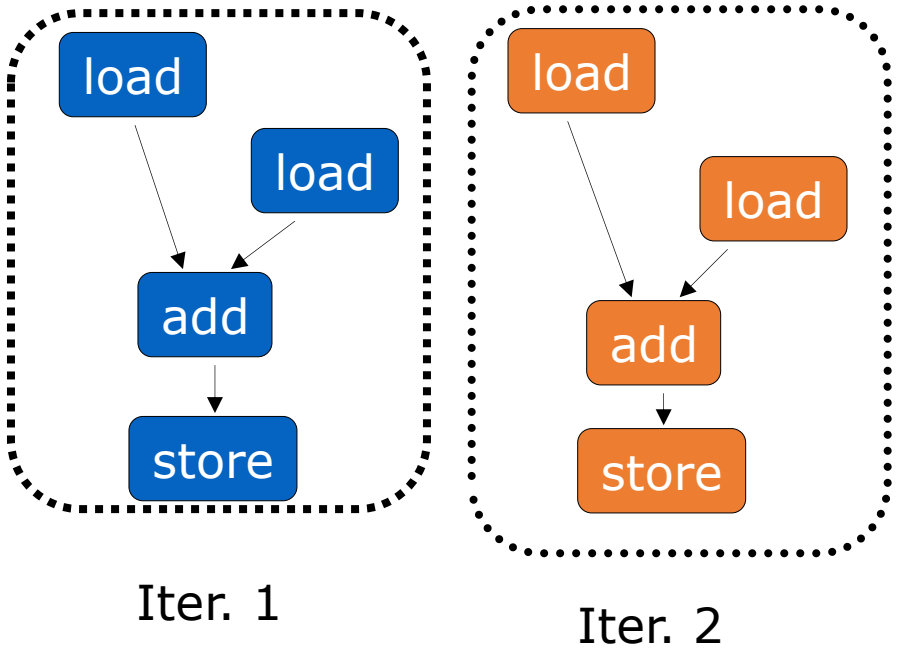



```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Model 2: SIMD

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

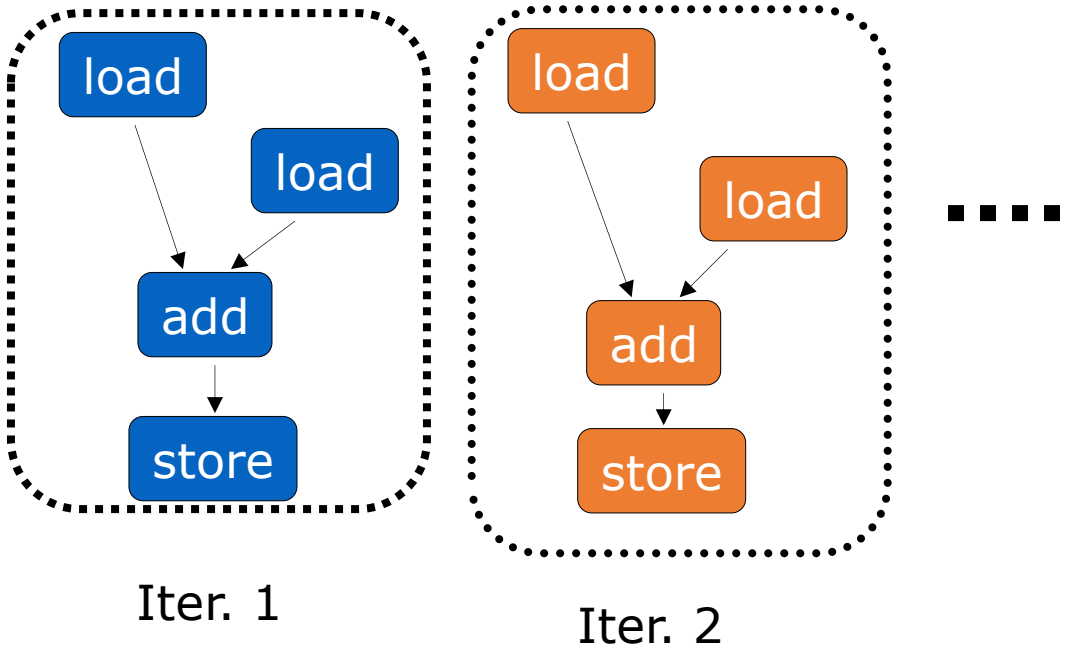
Scalar Sequential Code



Model 2: SIMD

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

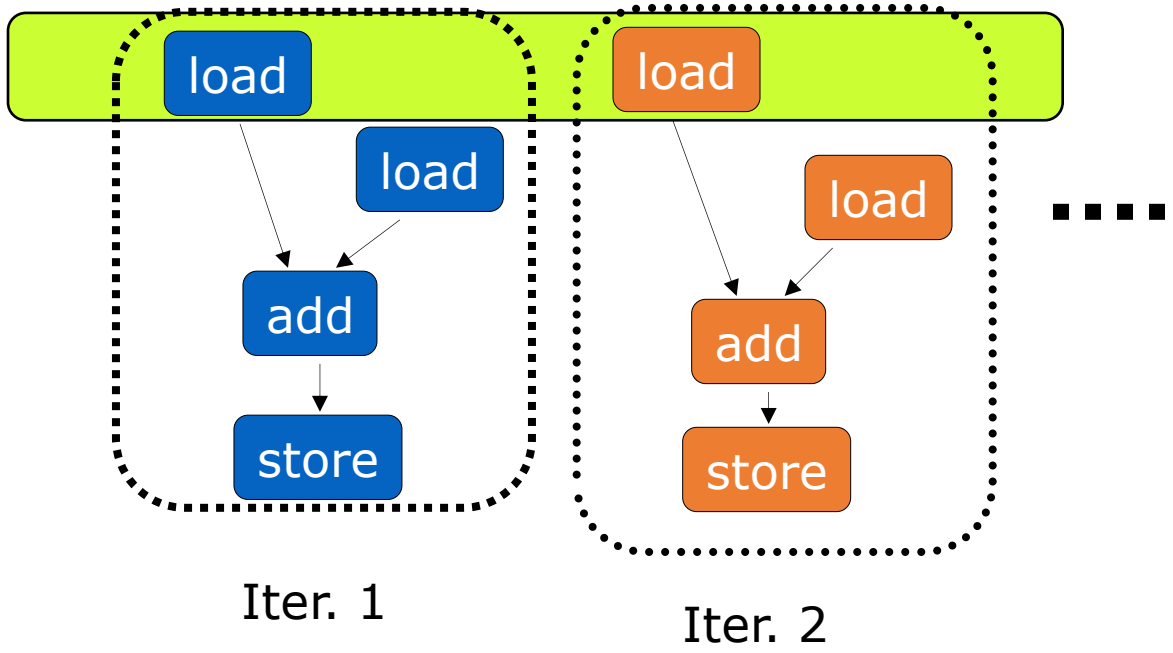
Scalar Sequential Code



Model 2: SIMD

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Scalar Sequential Code



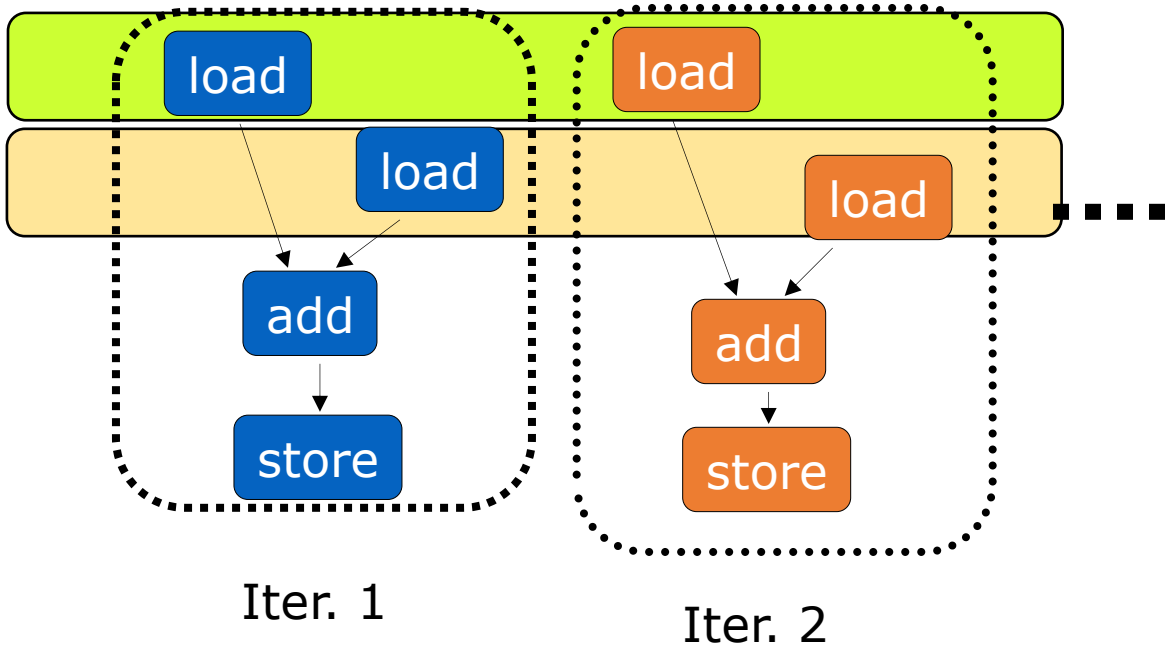
Vectorized Code

```
VLD      A → V1
```

Model 2: SIMD

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Scalar Sequential Code



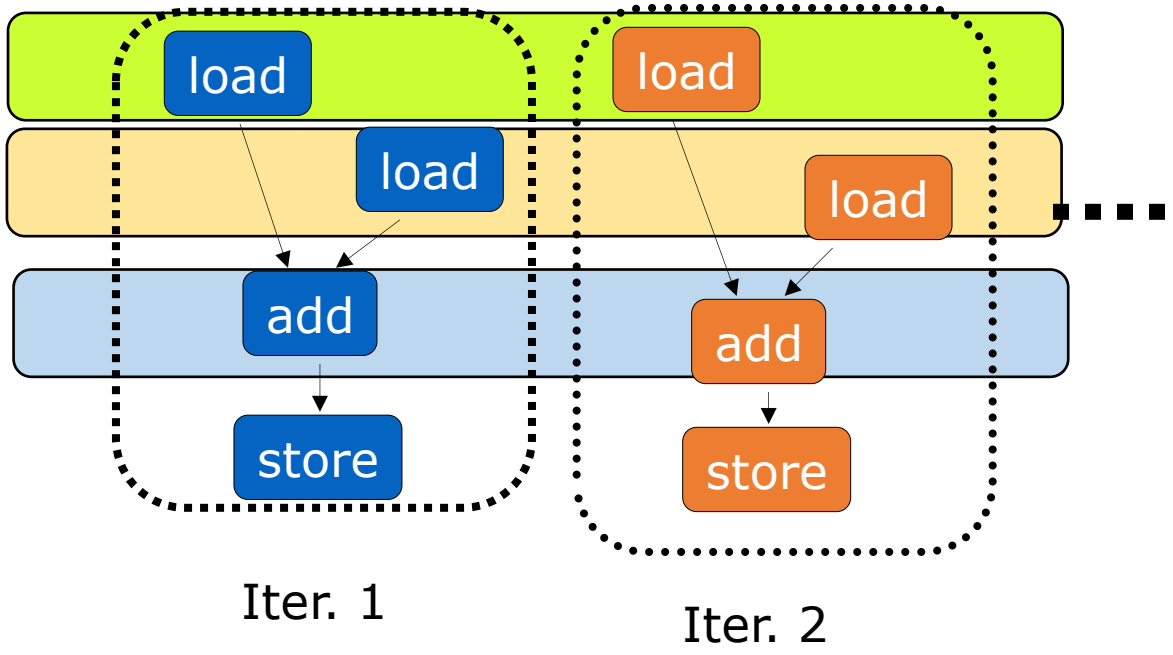
Vectorized Code

```
VLD     A → V1
VLD     B → V2
```

Model 2: SIMD

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Scalar Sequential Code



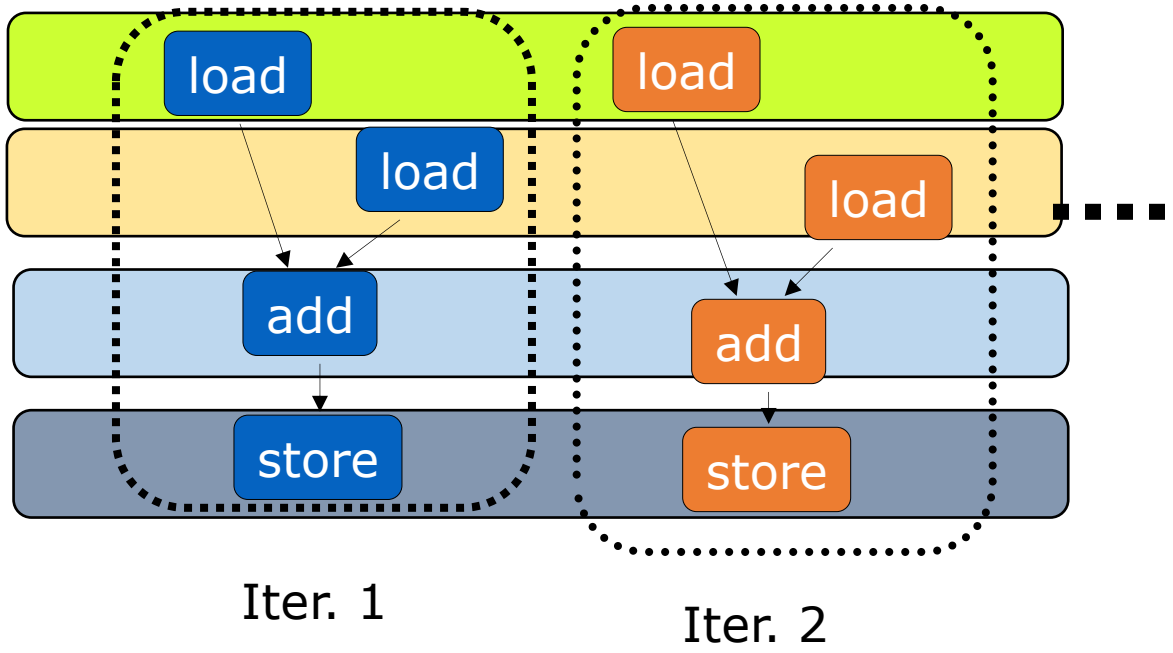
Vectorized Code

```
VLD     A → V1
VLD     B → V2
VADD    V1 + V2 → V3
```

Model 2: SIMD

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Scalar Sequential Code



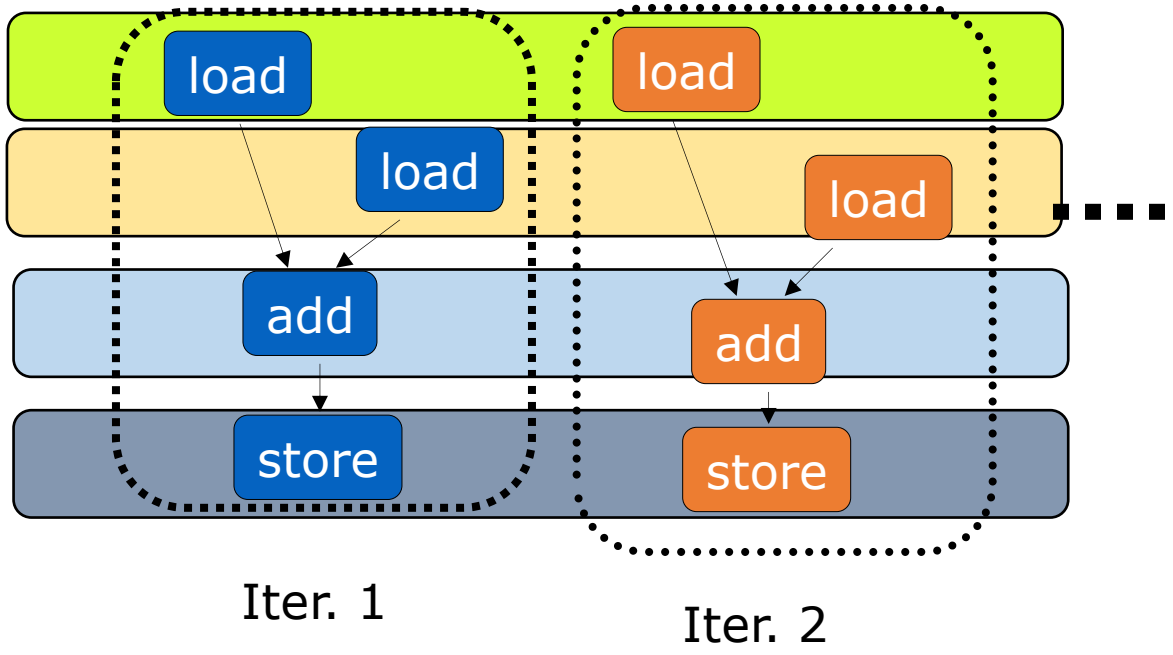
Vectorized Code

```
VLD     A → V1
VLD     B → V2
VADD    V1 + V2 → V3
VST     V3 → C
```

Model 2: SIMD

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Scalar Sequential Code



Vectorized Code

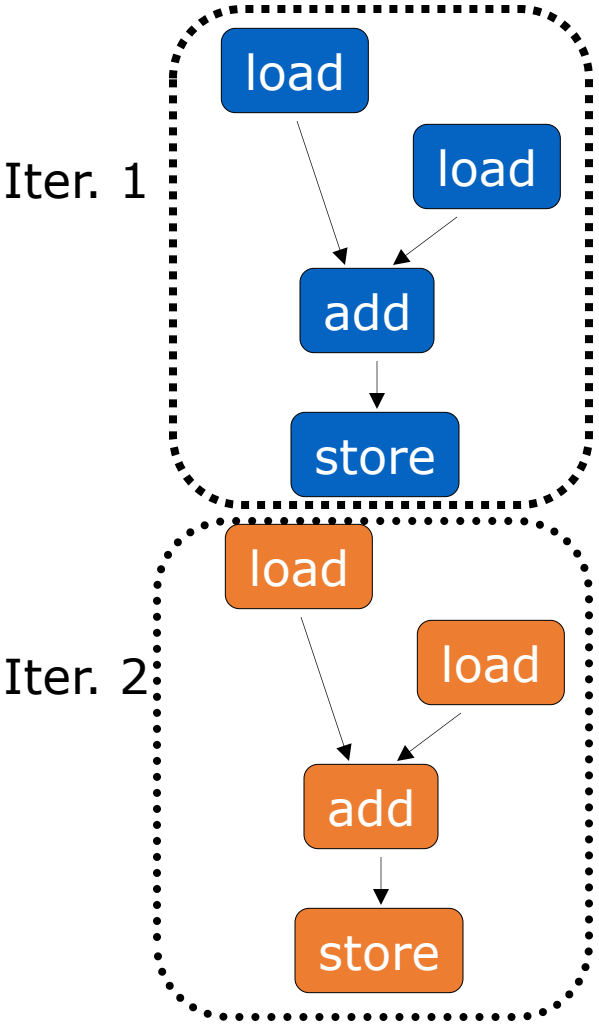
```
VLD     A → V1
VLD     B → V2
VADD    V1 + V2 → V3
VST     V3 → C
```

- Each iteration is independent
  - Idea: Generate a SIMD instruction to execute the same instruction from all iterations across different data
- Best executed by a SIMD processor (vector, array)

Model 3: Multithreaded

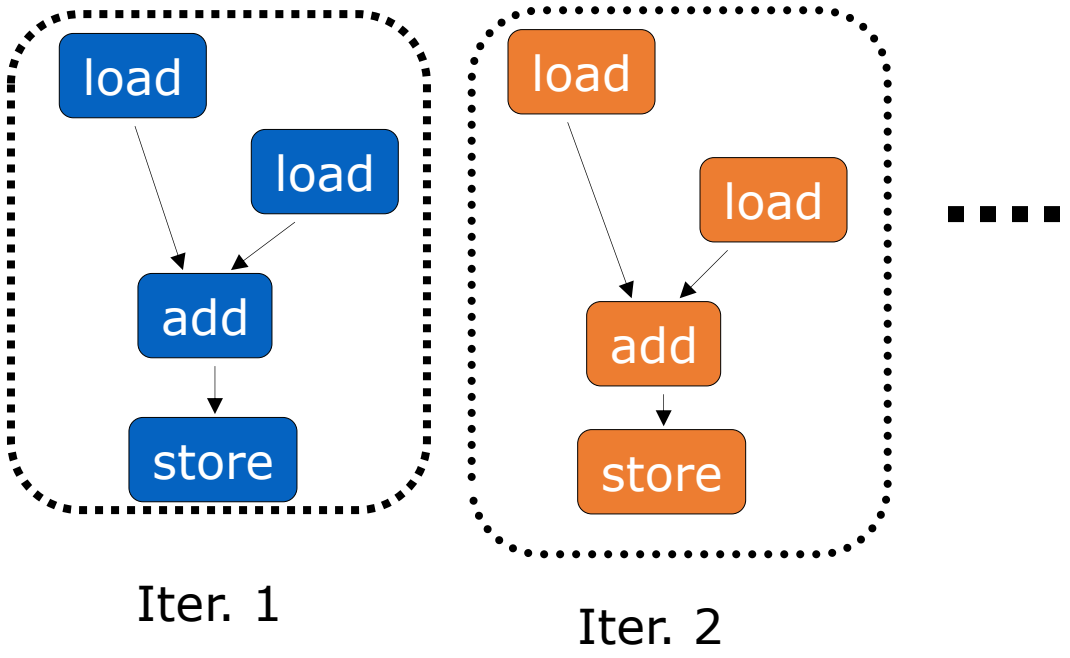
```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```

Scalar Sequential Code



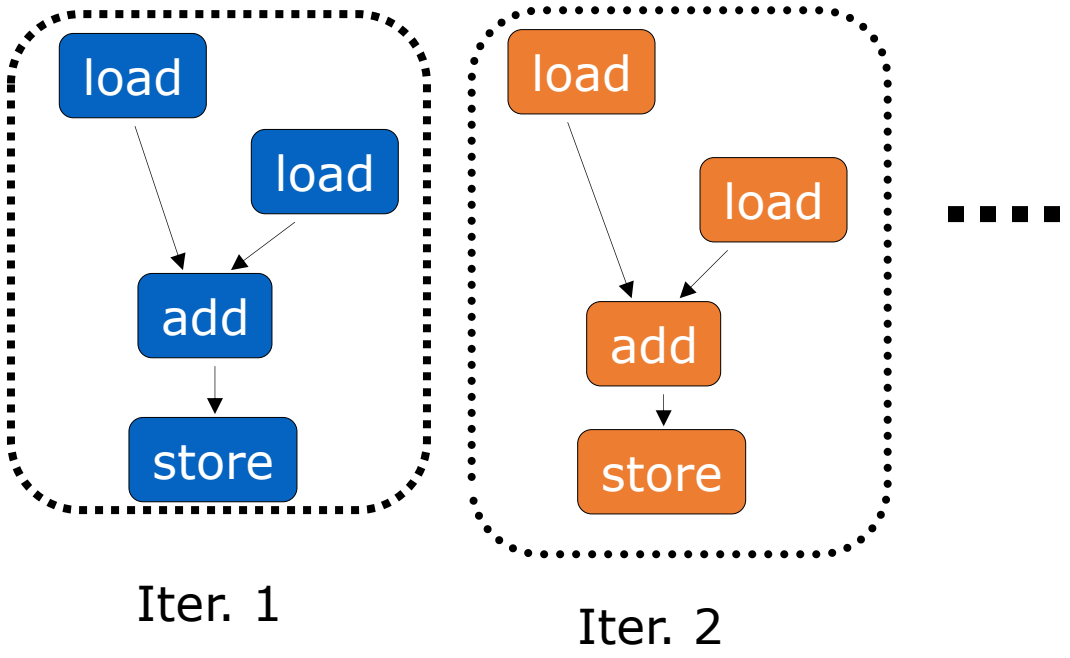
Model 3: Multithreaded

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



Model 3: Multithreaded

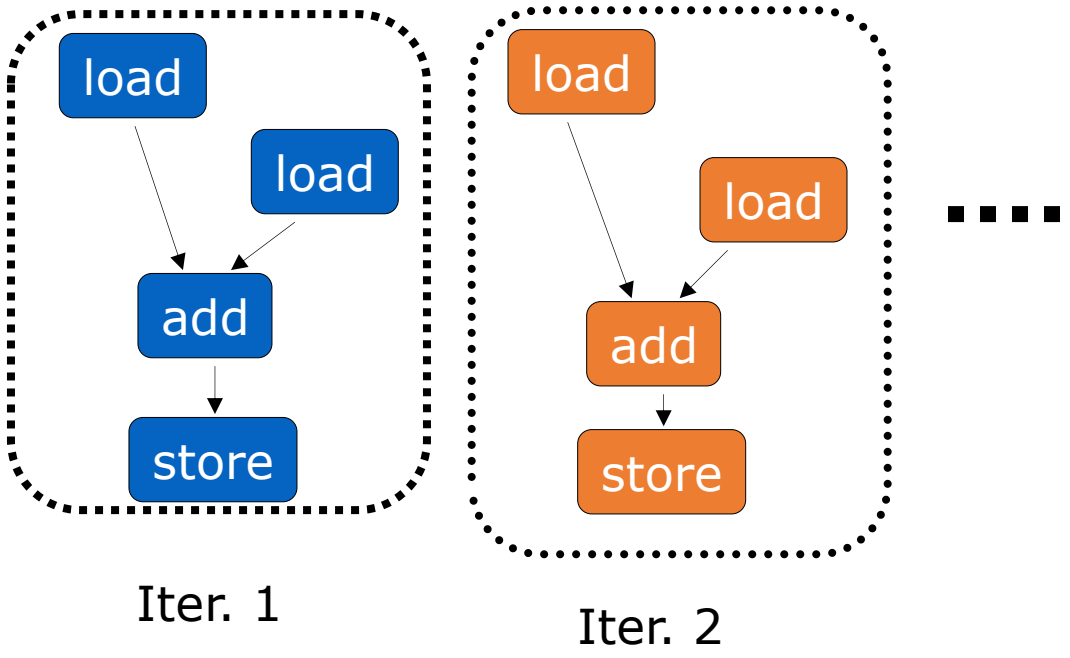
```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



- Each iteration is independent
  - Idea: Generate a thread to execute each iteration. Each thread runs the same operations on different data

Model 3: Multithreaded

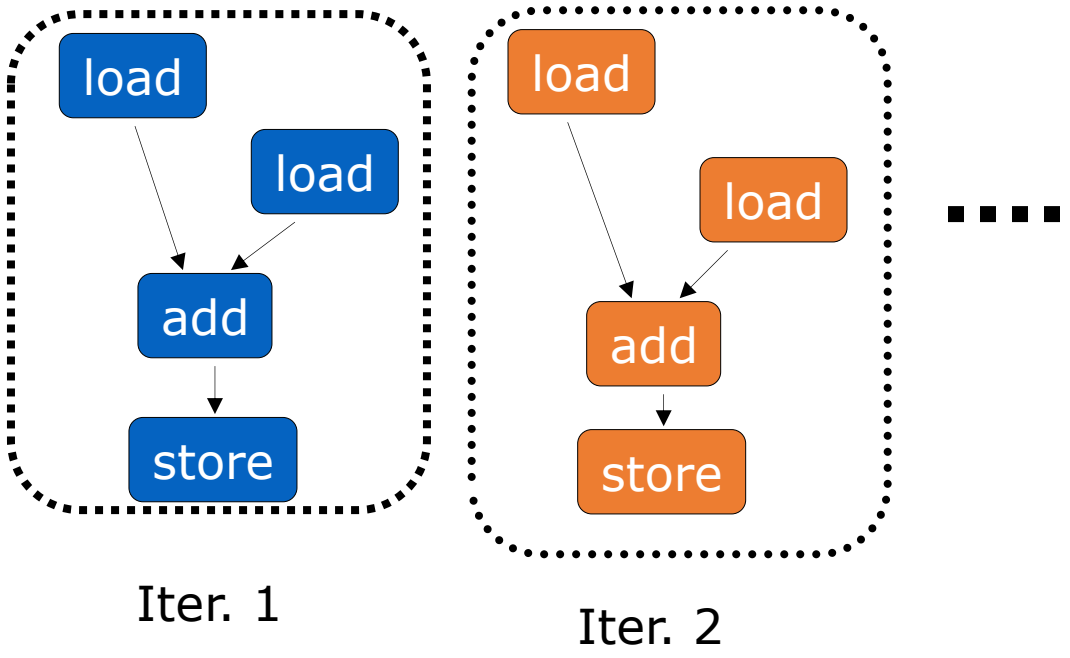
```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



- Each iteration is independent
  - Idea: Generate a thread to execute each iteration. Each thread runs the same operations on different data
- AKA SPMD(Single program, multiple data) or SIMT(Single instruction, multiple thread)

Model 3: Multithreaded

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



- Each iteration is independent
  - Idea: Generate a thread to execute each iteration. Each thread runs the same operations on different data
- Can be executed by a MIMD machine
- AKA SPMD(Single program, multiple data) or SIMT(Single instruction, multiple thread)

A GPU is a SIMD (SIMT) Machine

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

A GPU is a SIMD (SIMT) Machine

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

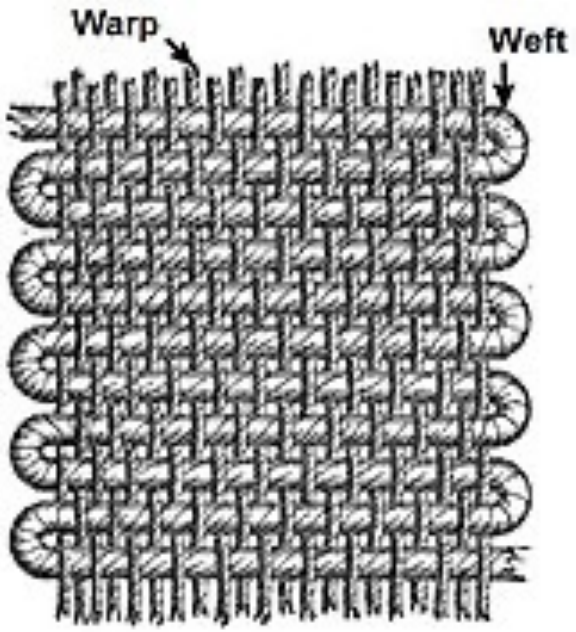
- It is programmed using threads (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

A GPU is a SIMD (SIMT) Machine

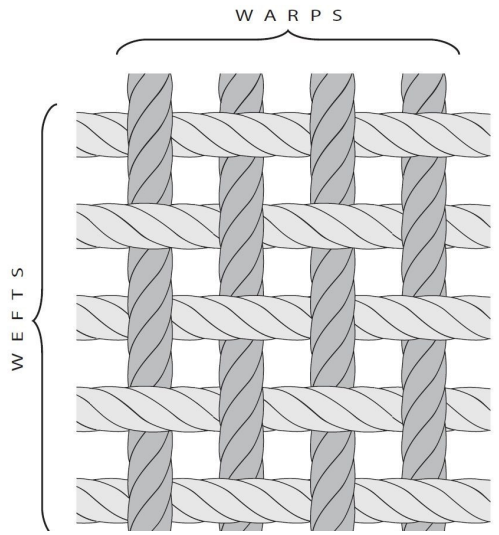
- Except it is **not** programmed using SIMD instructions

- 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!



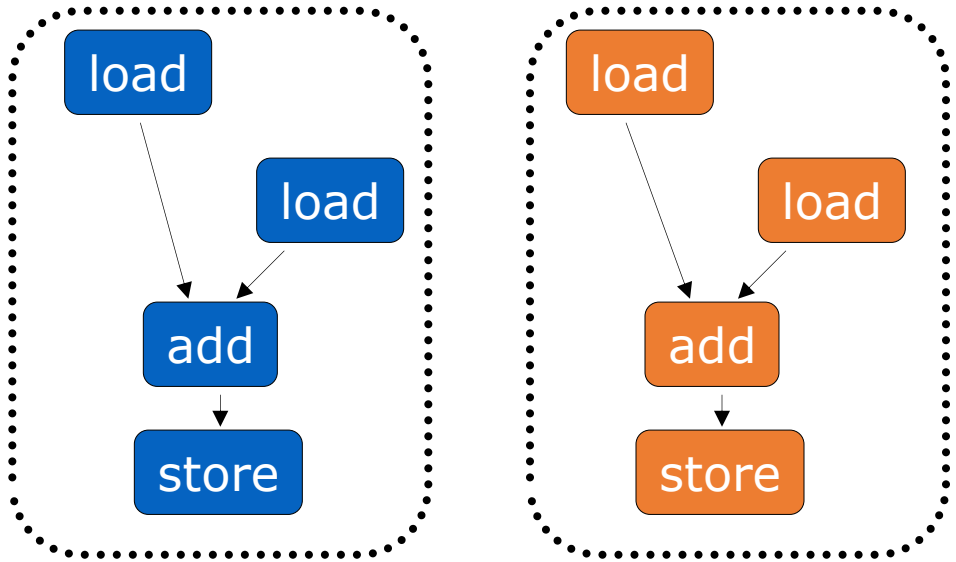
Source: Wikipedia



Source: peggyosterkamp.com

SPMD on SIMT Machine

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



....

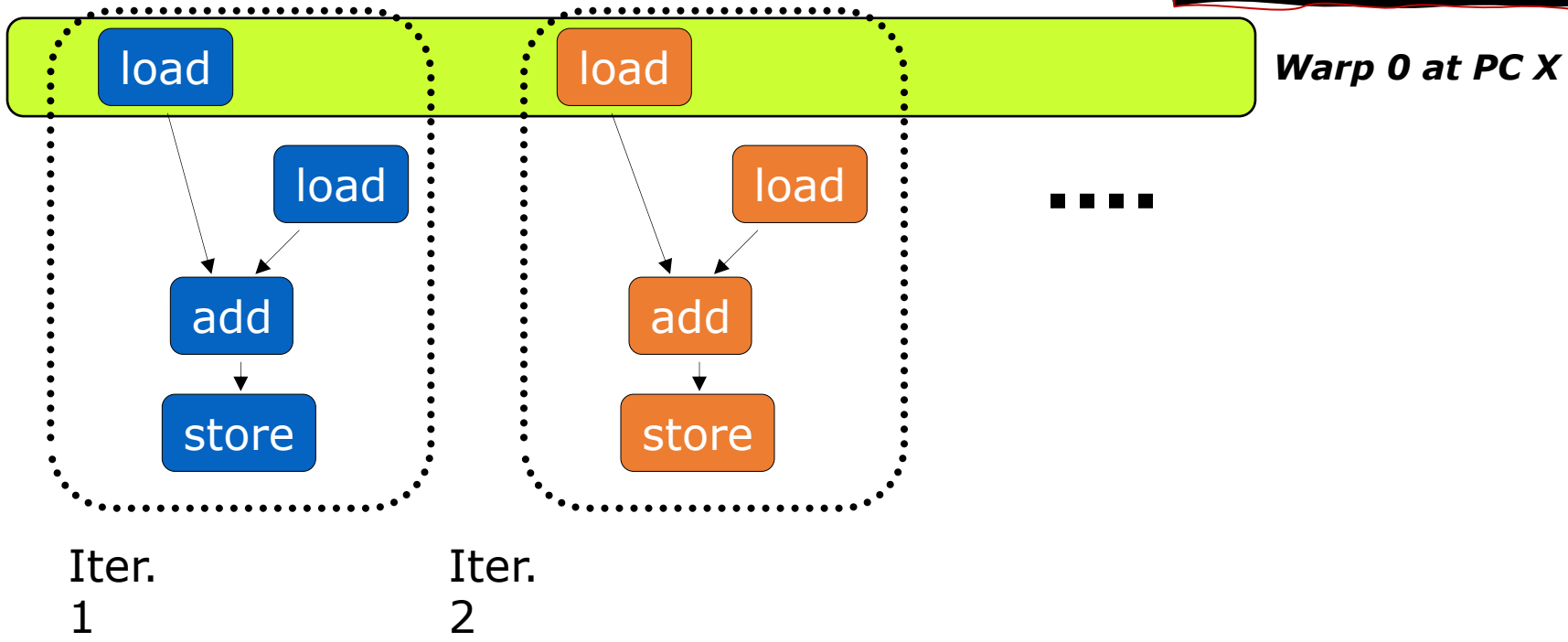
Iter. 1

Iter. 2

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

SPMD on SIMT Machine

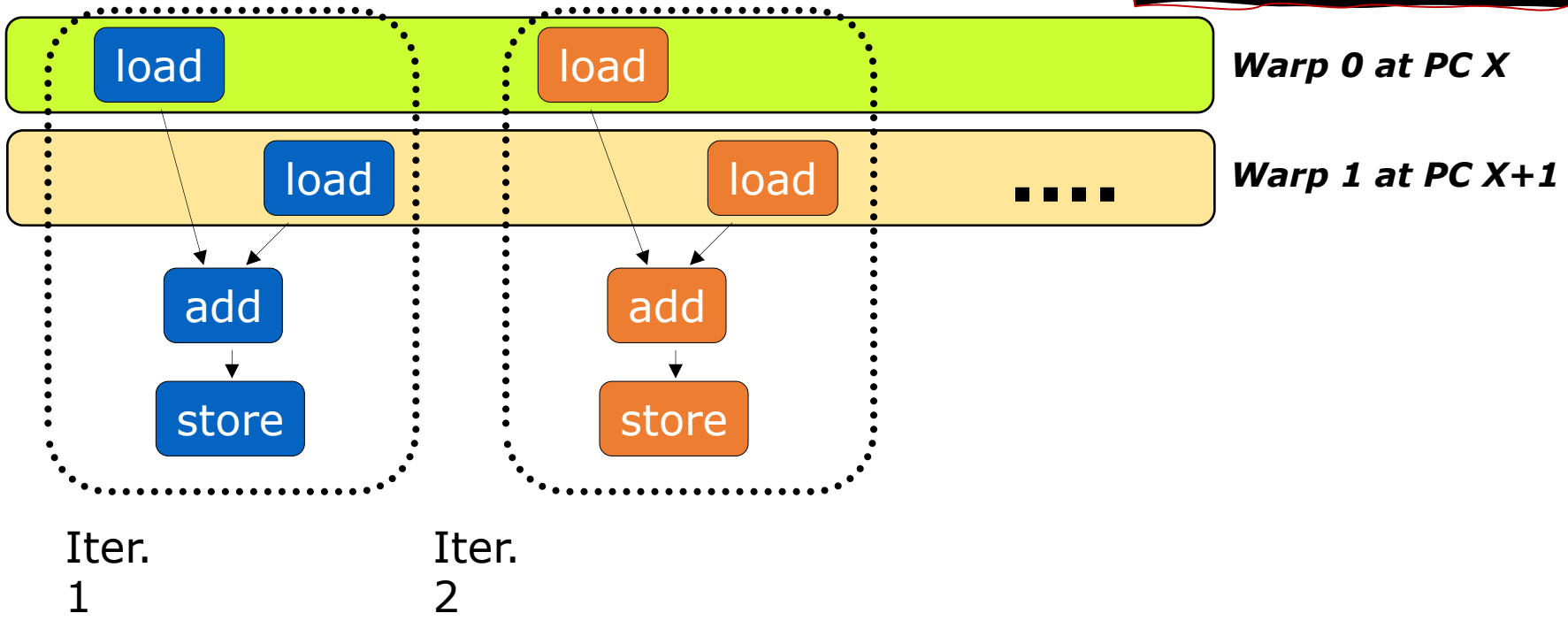
```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



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

SPMD on SIMT Machine

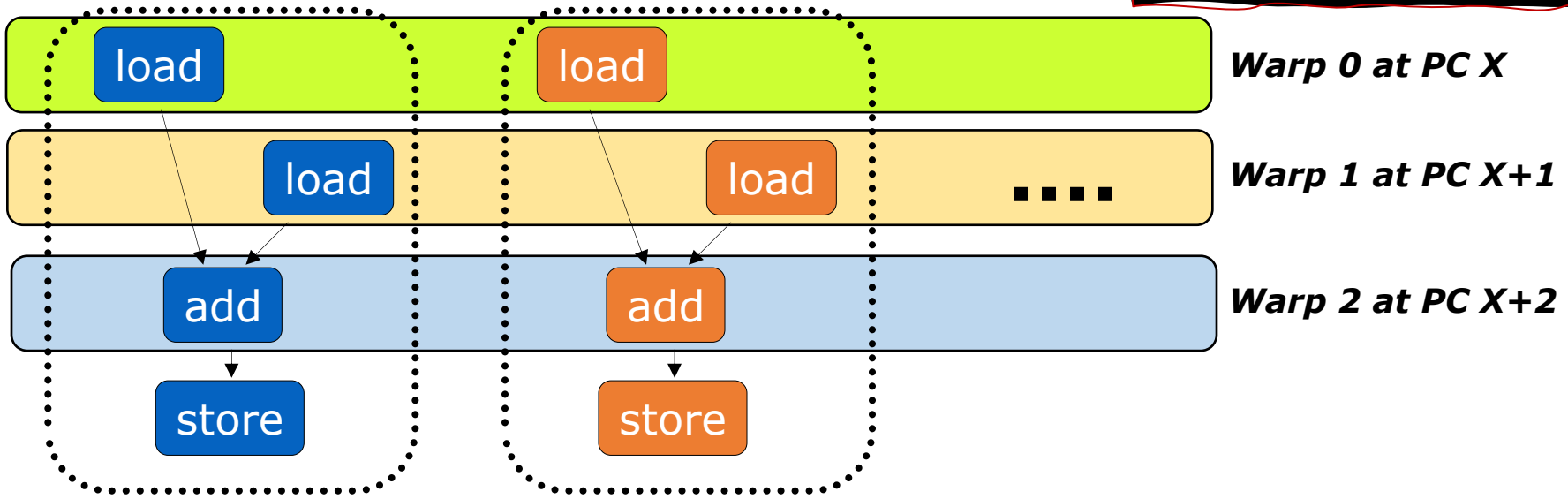
```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



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

SPMD on SIMT Machine

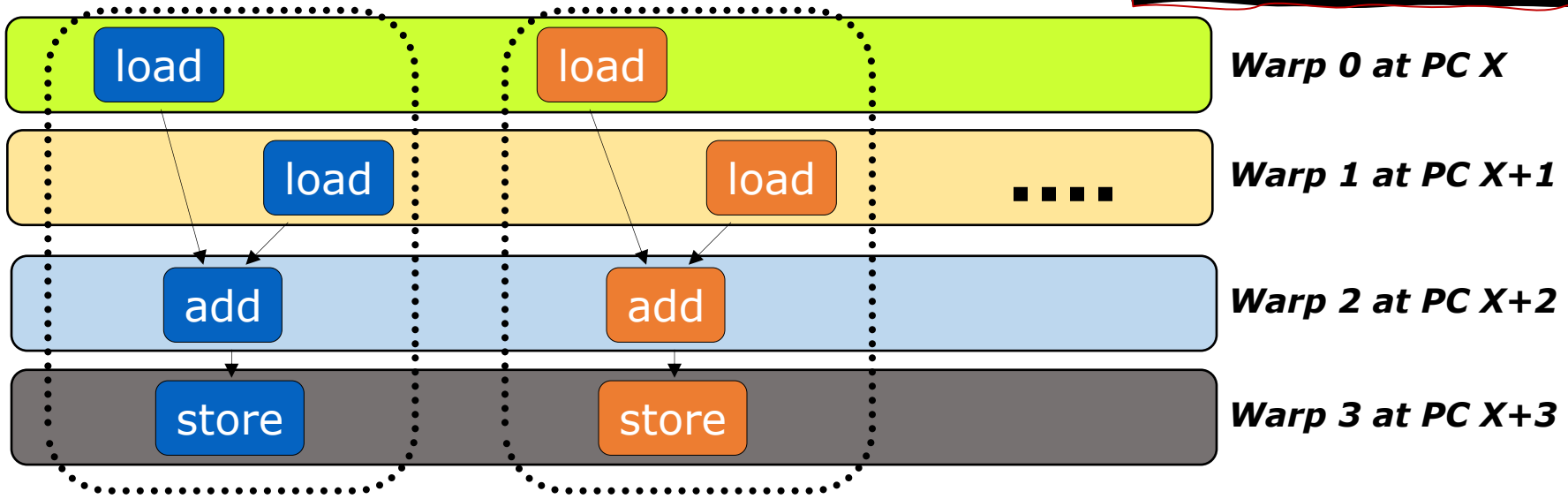
```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



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

SPMD on SIMT Machine

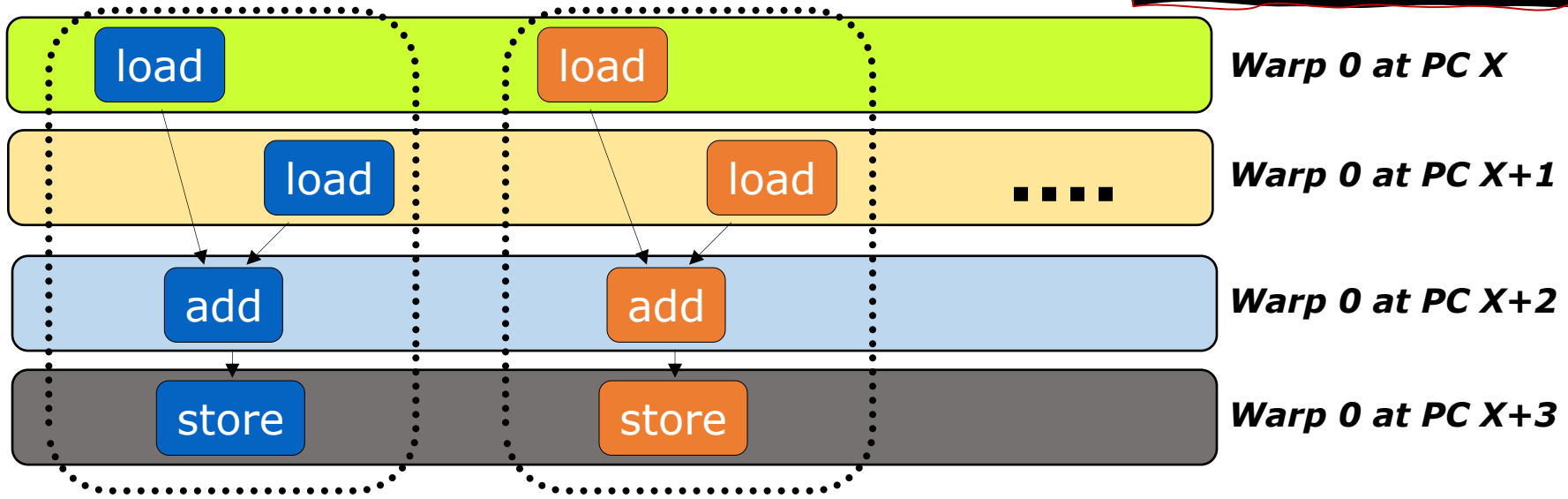
```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



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

SPMD on SIMT Machine

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



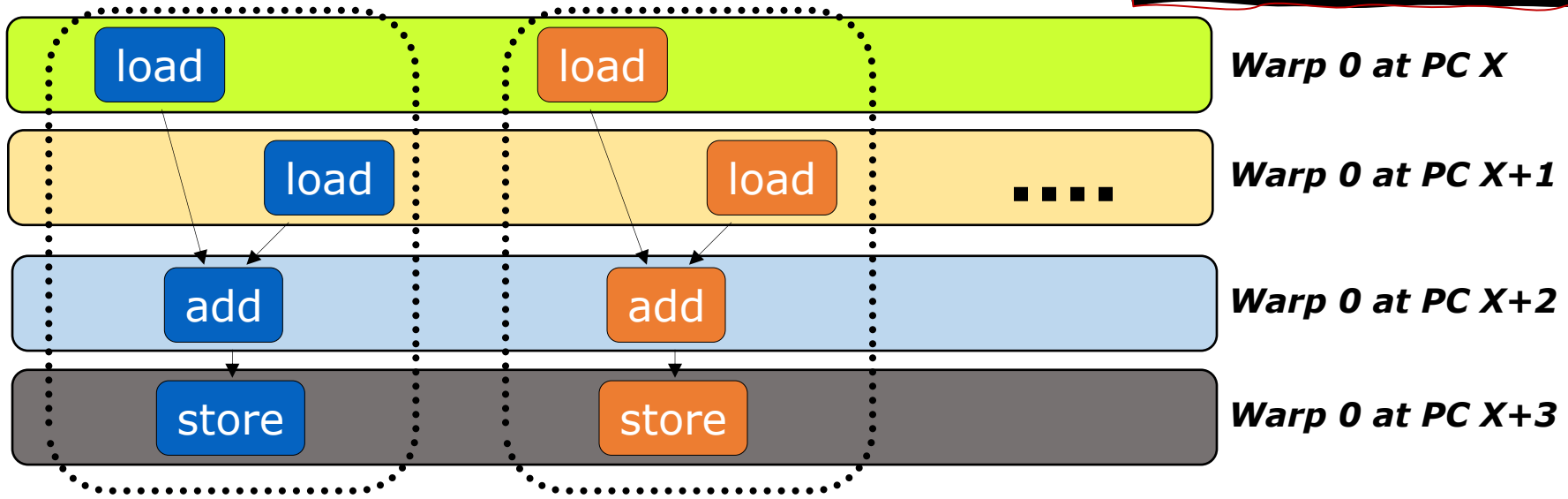
Warp: A set of threads that execute the same instruction (i.e., at the same PC)

This model is also called:

SPMD: Single Program Multiple Data

SPMD on SIMT Machine

```
for (let i = 0 to N) {
  C[i] = A[i] + B[i]
}
```



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

This model is also called:

SPMD: Single Program Multiple Data

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

GPU: SIMT programming model, SIMD underneath

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

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

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

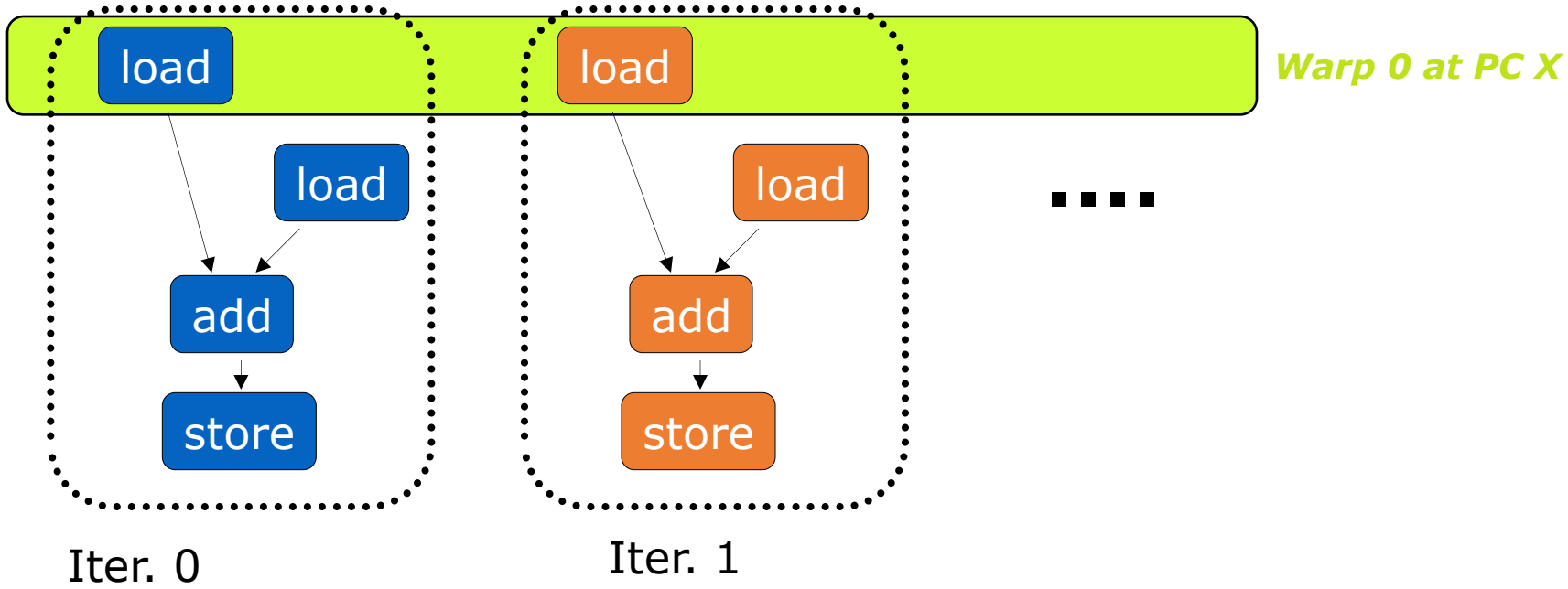
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- 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

Fine-Grained Multithreading of Warps

```
for (let i = 0 to 32000) {
  C[i] = A[i] + B[i]
}
```

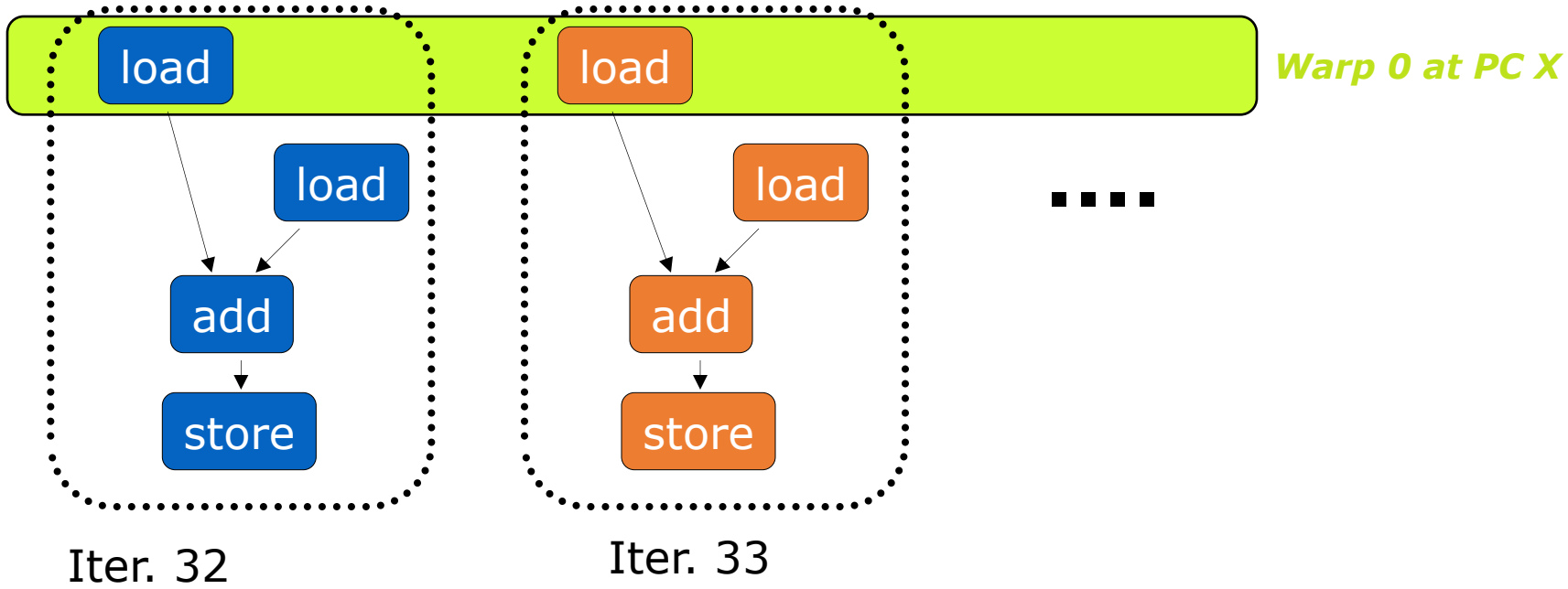
- Suppose a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading of warps



Fine-Grained Multithreading of Warps

```
for (let i = 0 to 32000) {
  C[i] = A[i] + B[i]
}
```

- Suppose 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

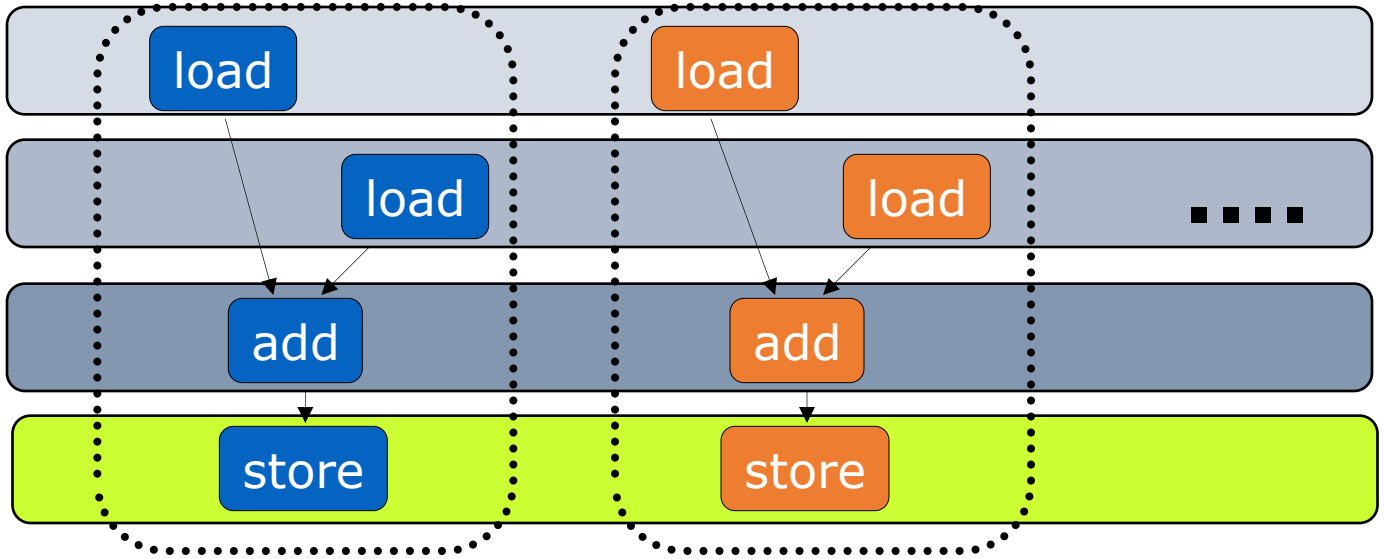


Fine-Grained Multithreading of Warps

```
for (let i = 0 to 32000) {
  C[i] = A[i] + B[i]
}
```

- Suppose we have 4 functional units
  - load 1
  - load 2
  - add
  - store

A moment in time where Warp 3 reached iterations 0-31



Warp 0 at PC X
Iterations 96-127

Warp 1 at PC X+1
Iterations 64-95

Warp 2 at PC X+2
Iterations 32-63

Warp 3 at PC X+3
Iterations 0-31

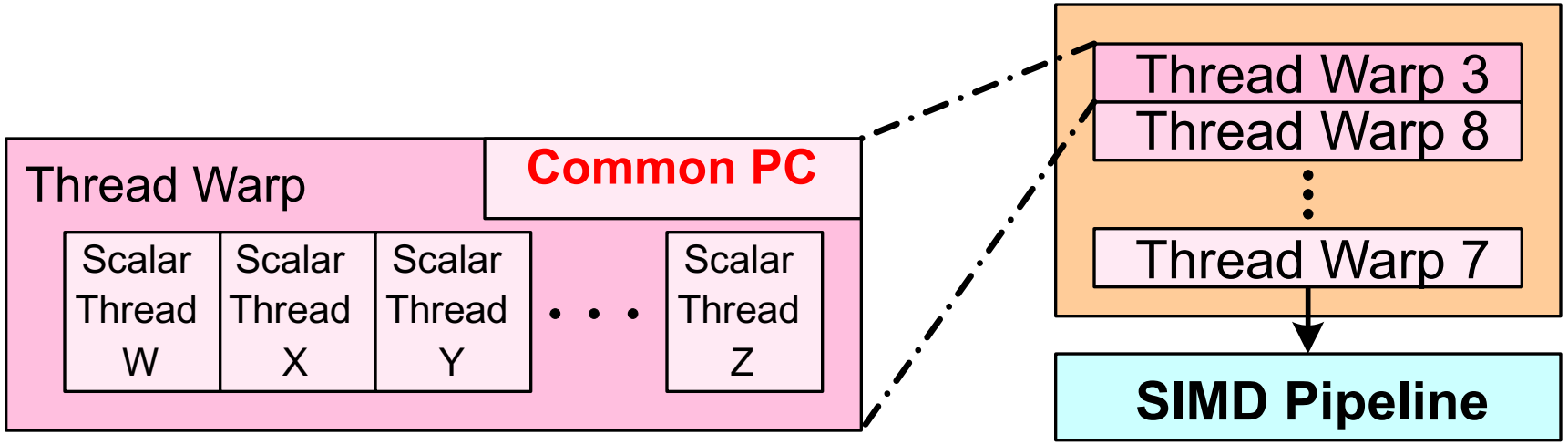
All threads in a warp are independent of each other

→ They be executed seamlessly in a fine-grained multithreaded pipeline

Warps and Warp-Level FGMT

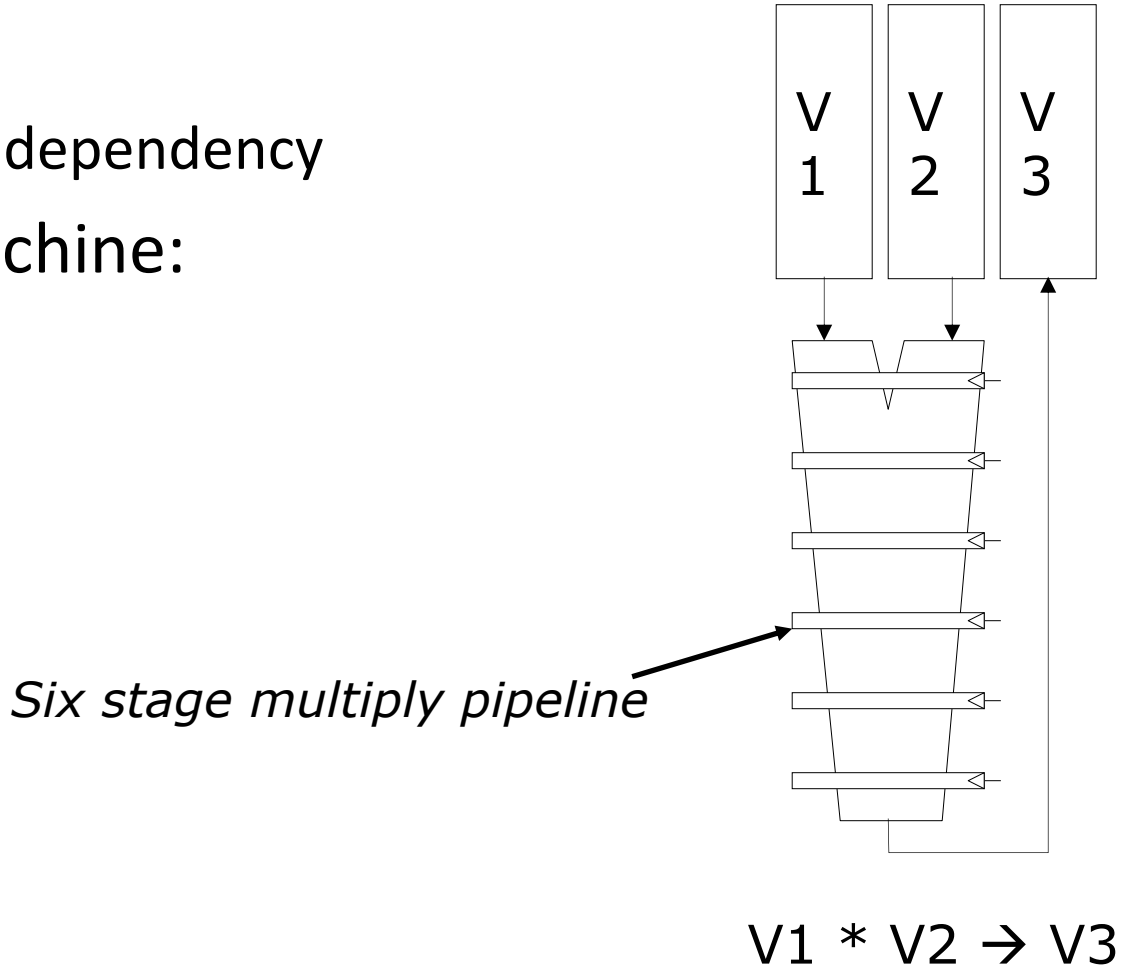
```
for (let i = 0 to 32000) {
  C[i] = A[i] + B[i]
}
```

- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same code

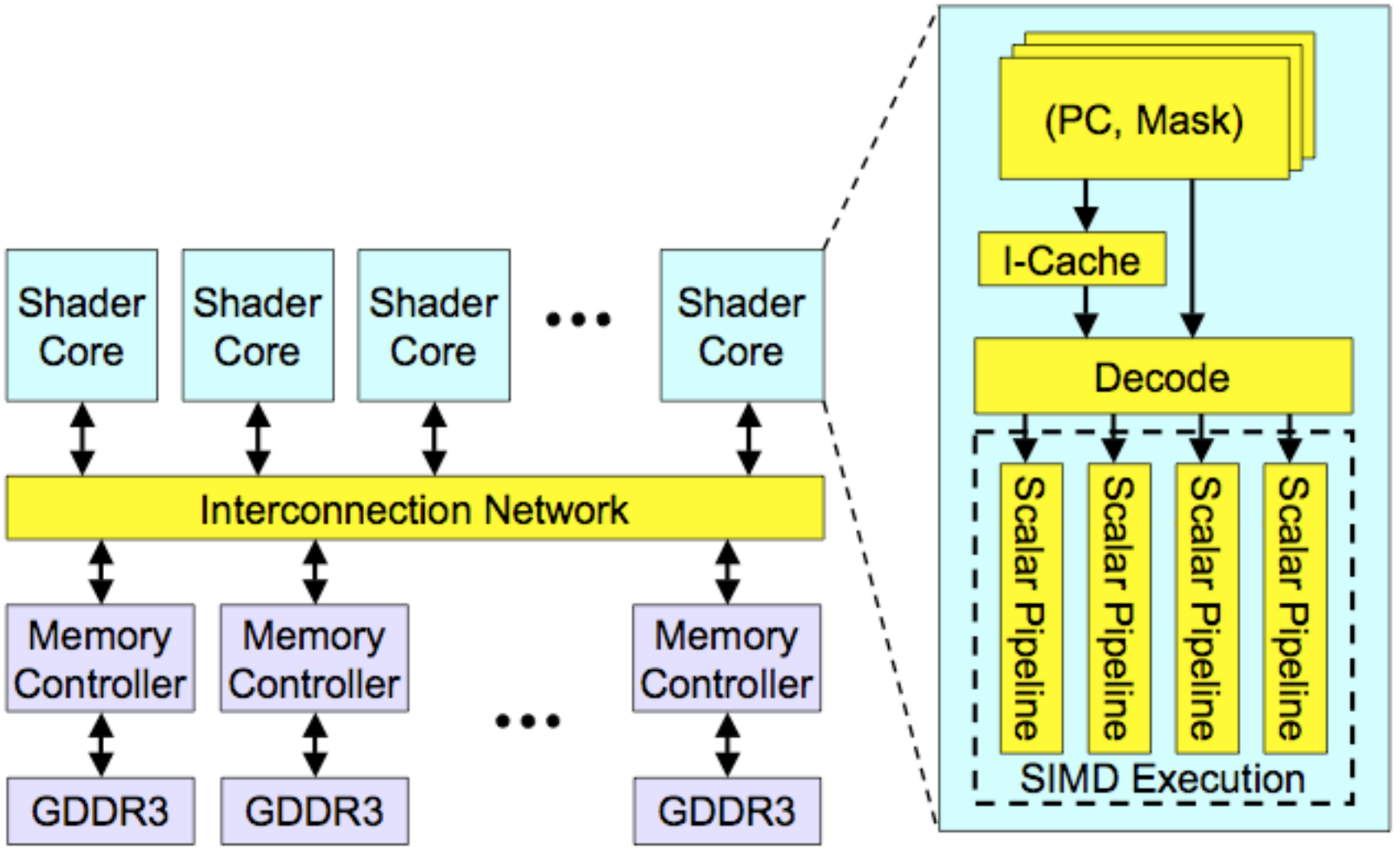


Remember: Vector Processor Functional Unit

- A deep pipeline
  - Fast clock cycle
  - Simple control due to lack of data dependency
- Example of Vector Processor Machine:
  - Cray-1, 1978

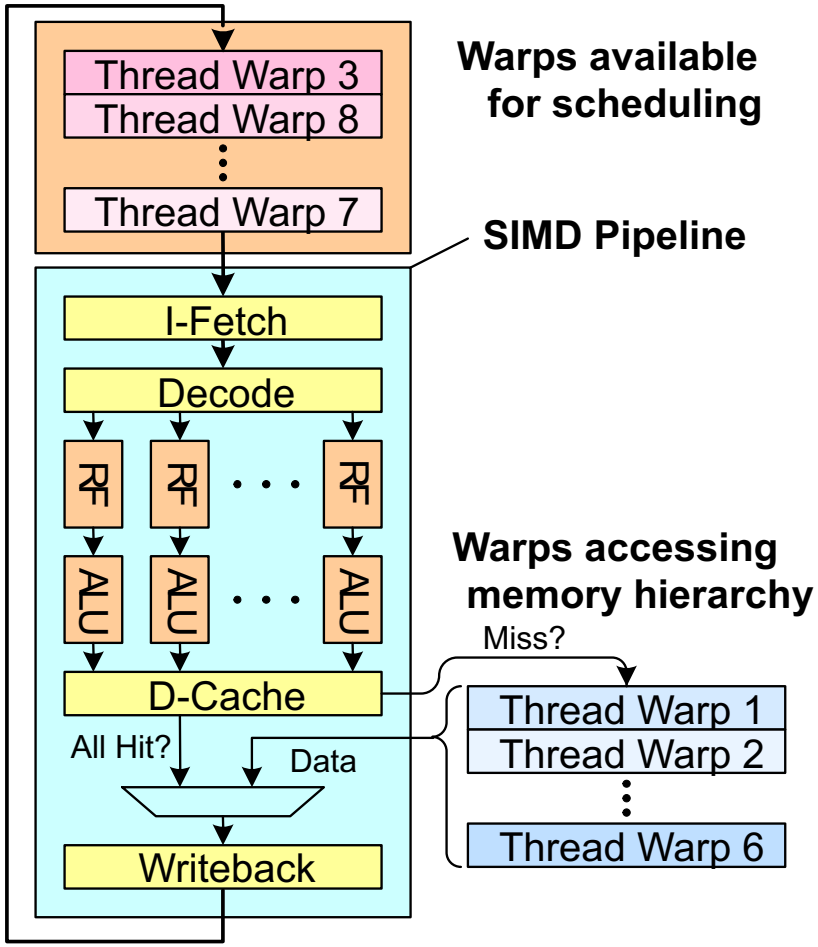


High-Level View of a GPU (Old GPUs)



Latency Hiding via Warp-Level FGMT

- Fine-grained multithreading
  - Only **One instruction per thread** in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies



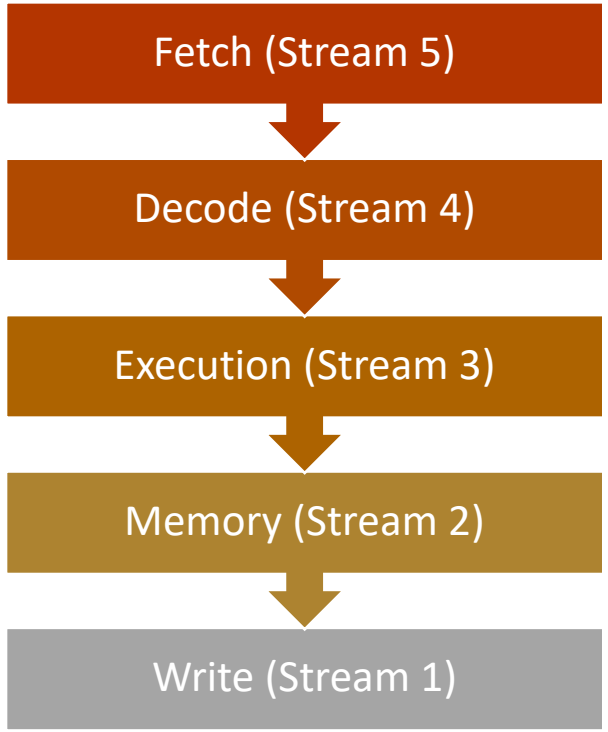
Slide credit: Tor Aamodt

Fine-Grained Multithreading

- Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
  - Hardware has multiple thread contexts (PC+registers per thread)
  - Threads are completely independent
  - No instruction is fetched from the same thread until the prior branch/instruction from the thread completes

Fine-Grained Multithreading

- Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
  - Hardware has multiple thread contexts (PC+registers per thread)
  - Threads are completely independent
  - No instruction is fetched from the same thread until the prior branch/instruction from the thread completes



Fine-Grained Multithreading

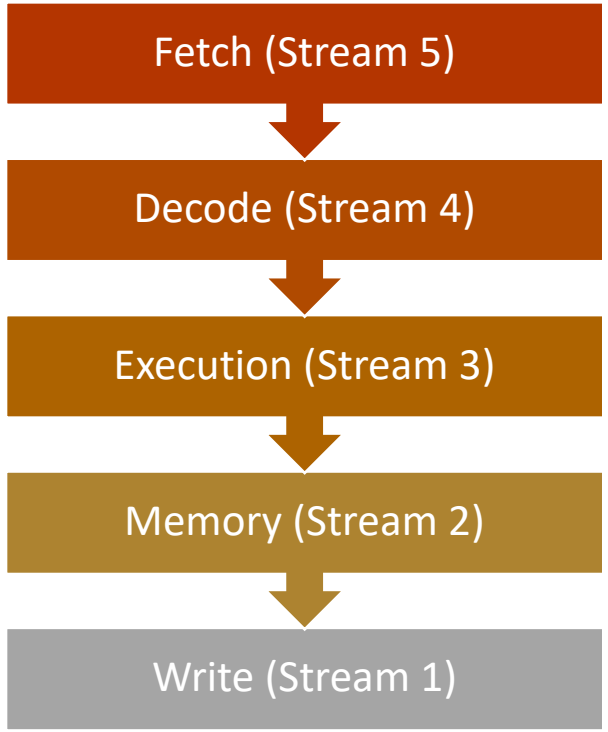
- Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
  - Hardware has multiple thread contexts (PC+registers per thread)
  - Threads are completely independent
  - No instruction is fetched from the same thread until the prior branch/instruction from the thread completes

+ No logic needed for handling control and data dependences within a thread

+ High thread-level throughput

-- Extra logic for keeping thread contexts

-- Throughput loss when there are not enough threads to keep the pipeline full



Fine-Grained Multithreading

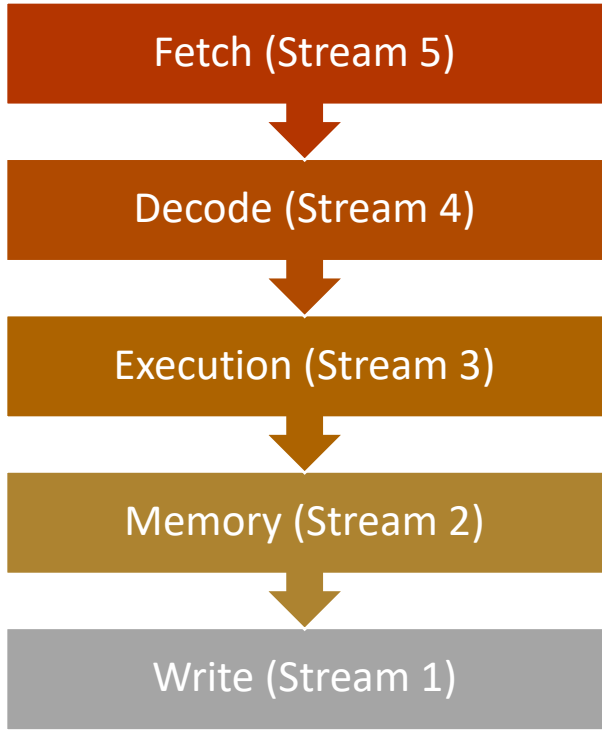
- Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
  - Hardware has multiple thread contexts (PC+registers per thread)
  - Threads are completely independent
  - No instruction is fetched from the same thread until the prior branch/instruction from the thread completes

+ No logic needed for handling control and data dependences within a thread

+ High thread-level throughput

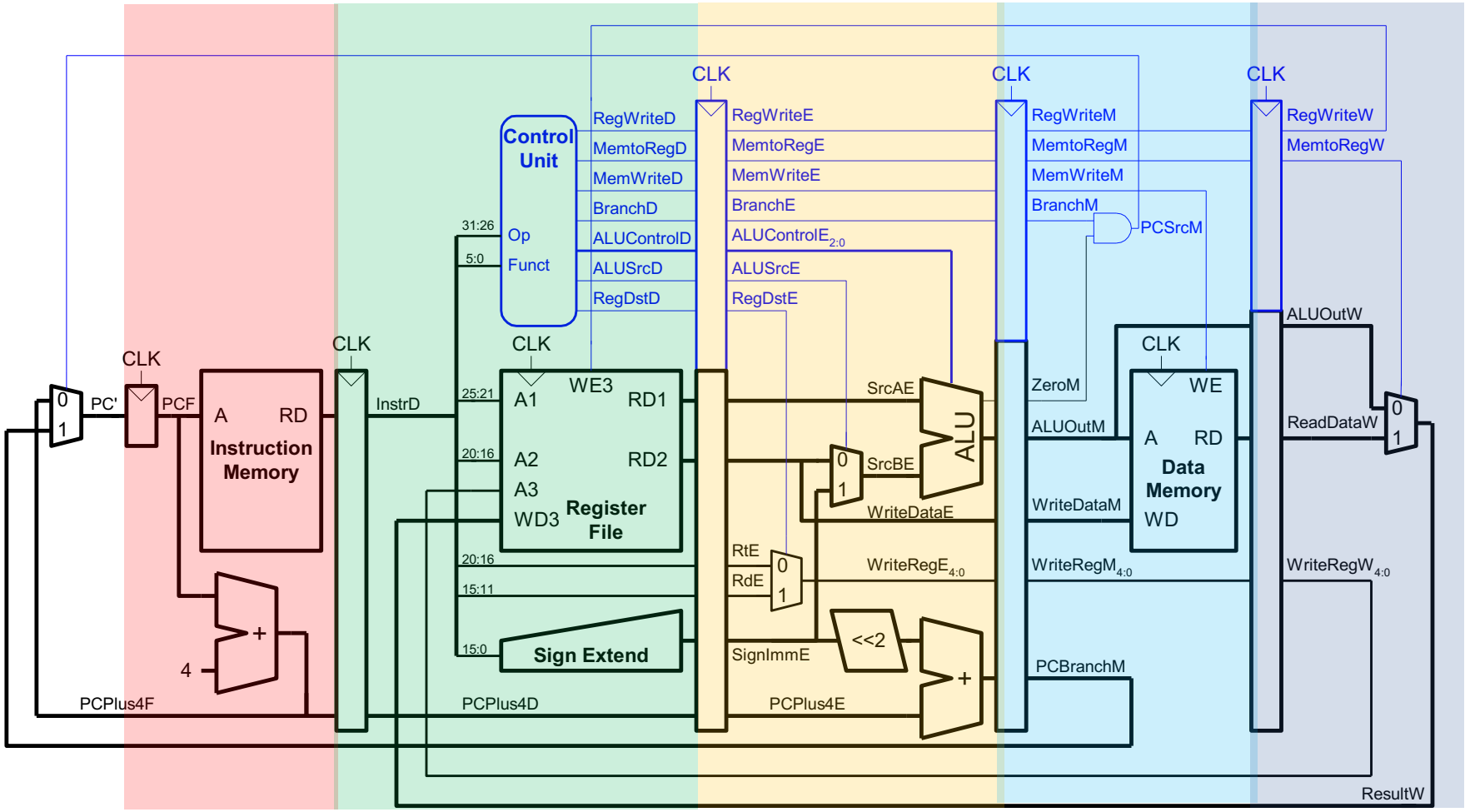
-- Extra logic for keeping thread contexts

-- Throughput loss when there are not enough threads to keep the pipeline full



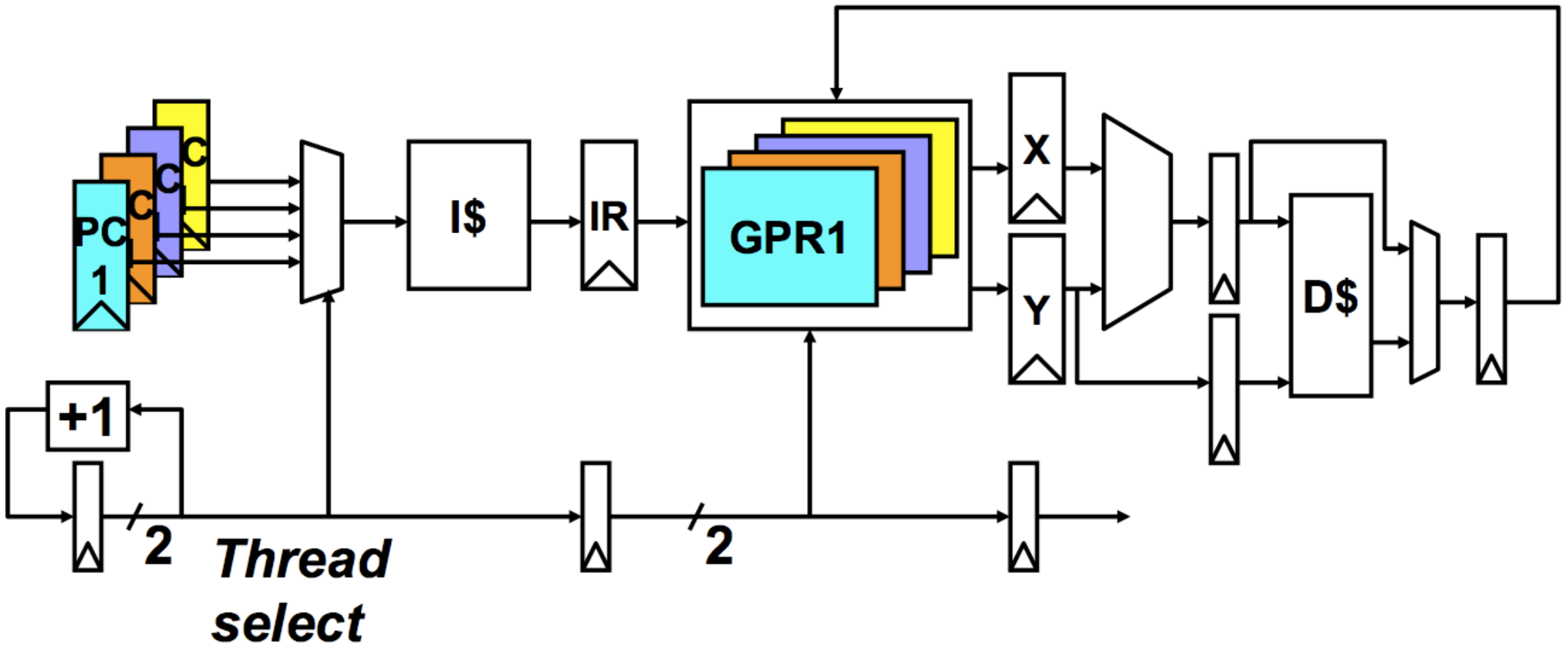
Each pipeline stage has an instruction from a different, completely-independent thread

Scalar Pipeline



Each pipeline stage has an instruction from a different, completely-independent thread

Multithreaded Pipeline Example



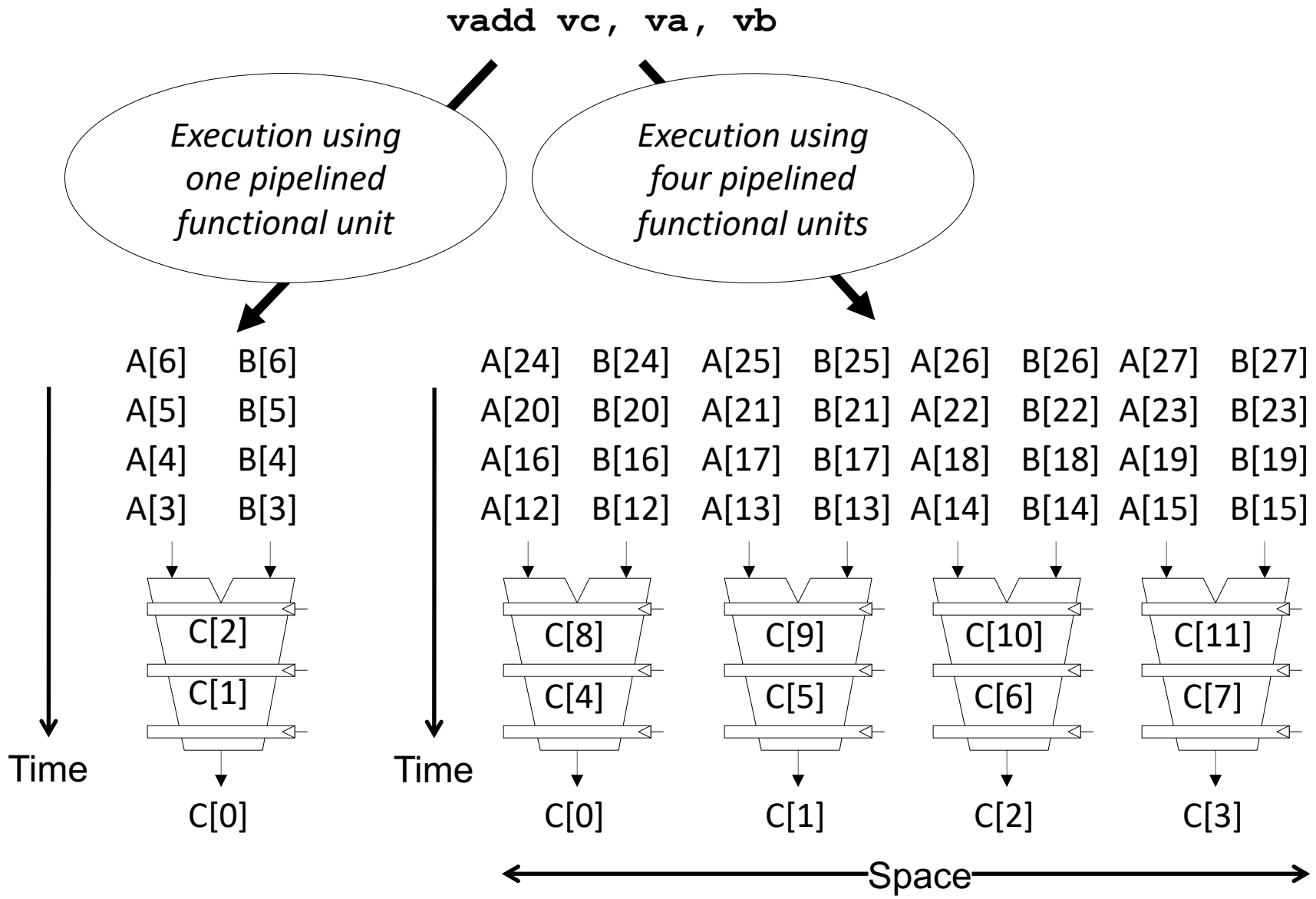
Carrying "thread select" value down the pipeline

Slide credit: Joel Emer

Multithreaded Pipeline Example

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
T1: LW r1, 0(r2)	F	D	X	M	W					
T2: ADD r7, r1, r4		F	D	X	M	W				
T3: XORI r5, r4, #12			F	D	X	M	W			
T4: SW 0(r7), r5				F	D	X	M	W		
T1: LW r5, 12(r1)					F	D	X	M	W	

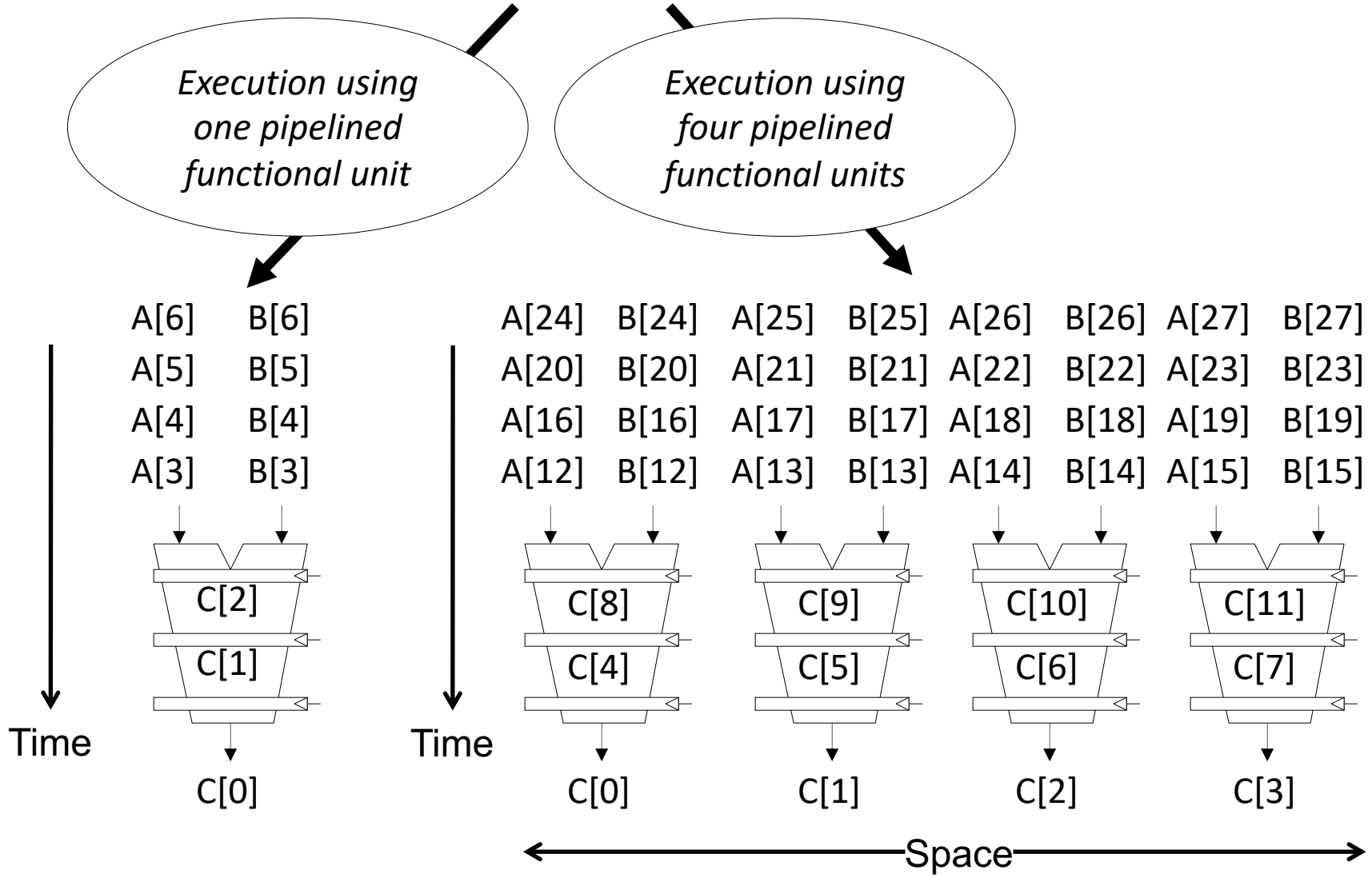
Vector Instruction Execution



Slide credit: Krste Asanovic

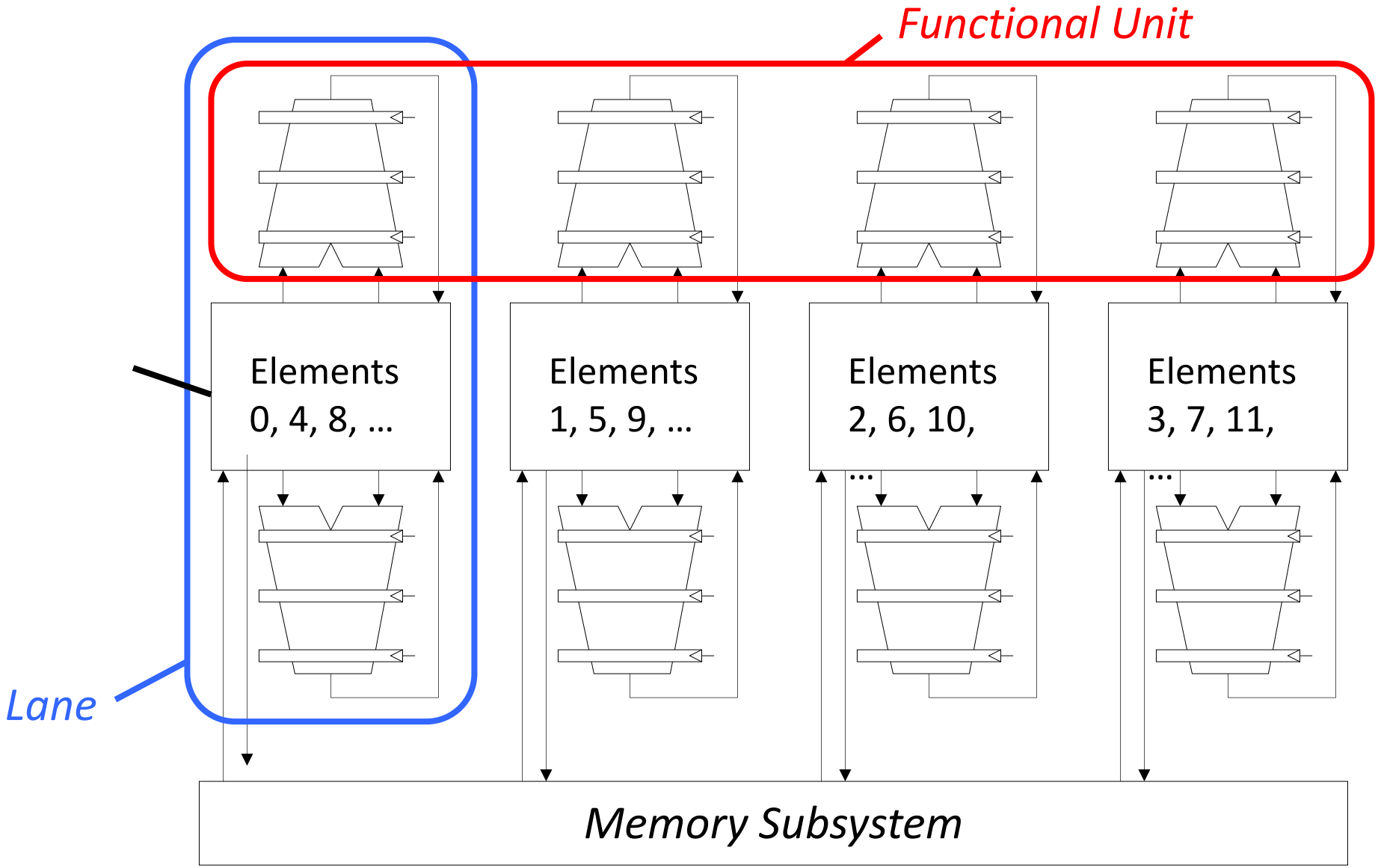
Vector Instruction Execution

32-thread warp executing **ADD A[tid],B[tid] → C[tid]**



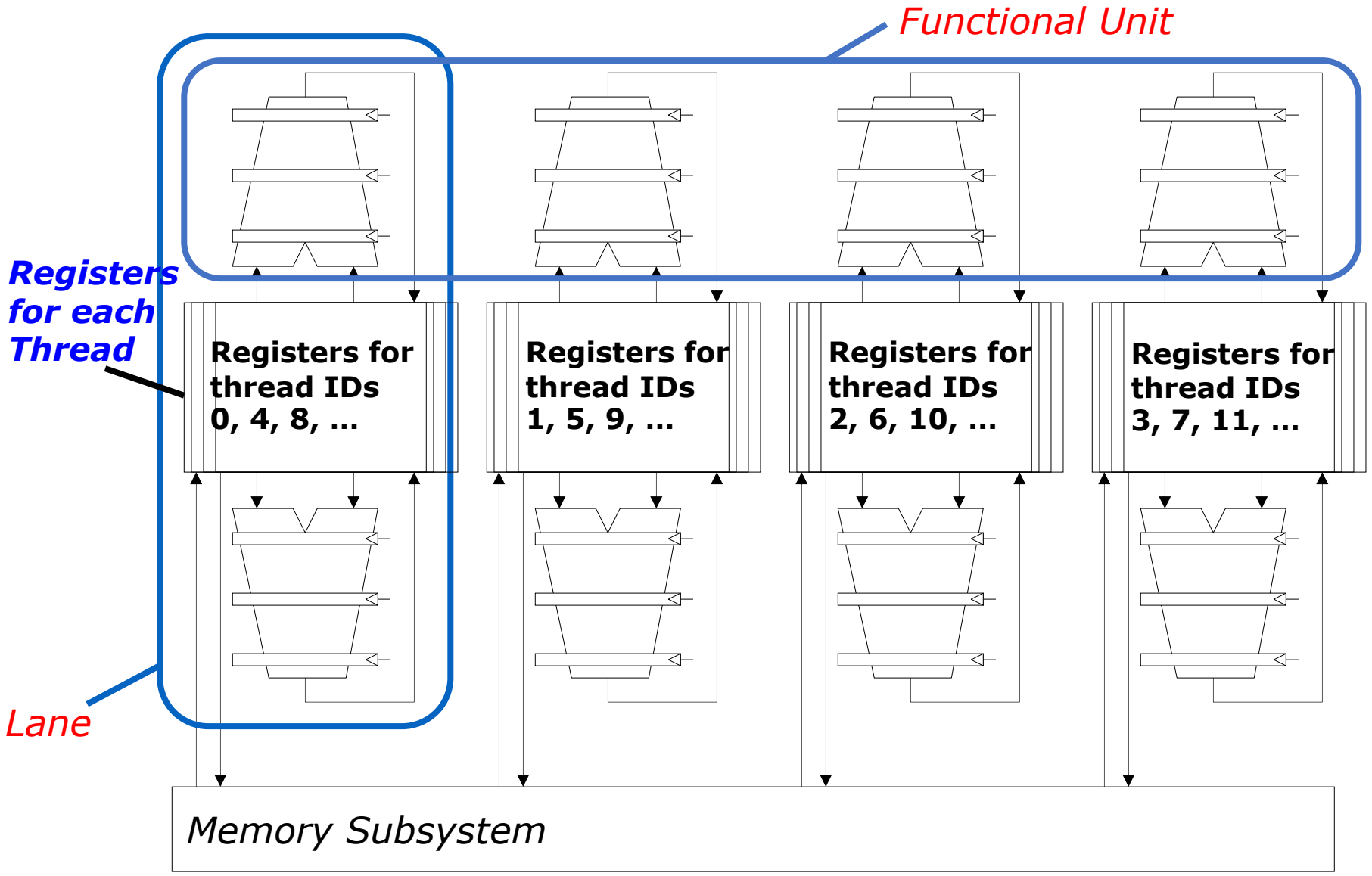
Slide credit: Krste Asanovic

Vector Unit Structure



Slide credit: Krste Asanovic

Vector Unit Structure

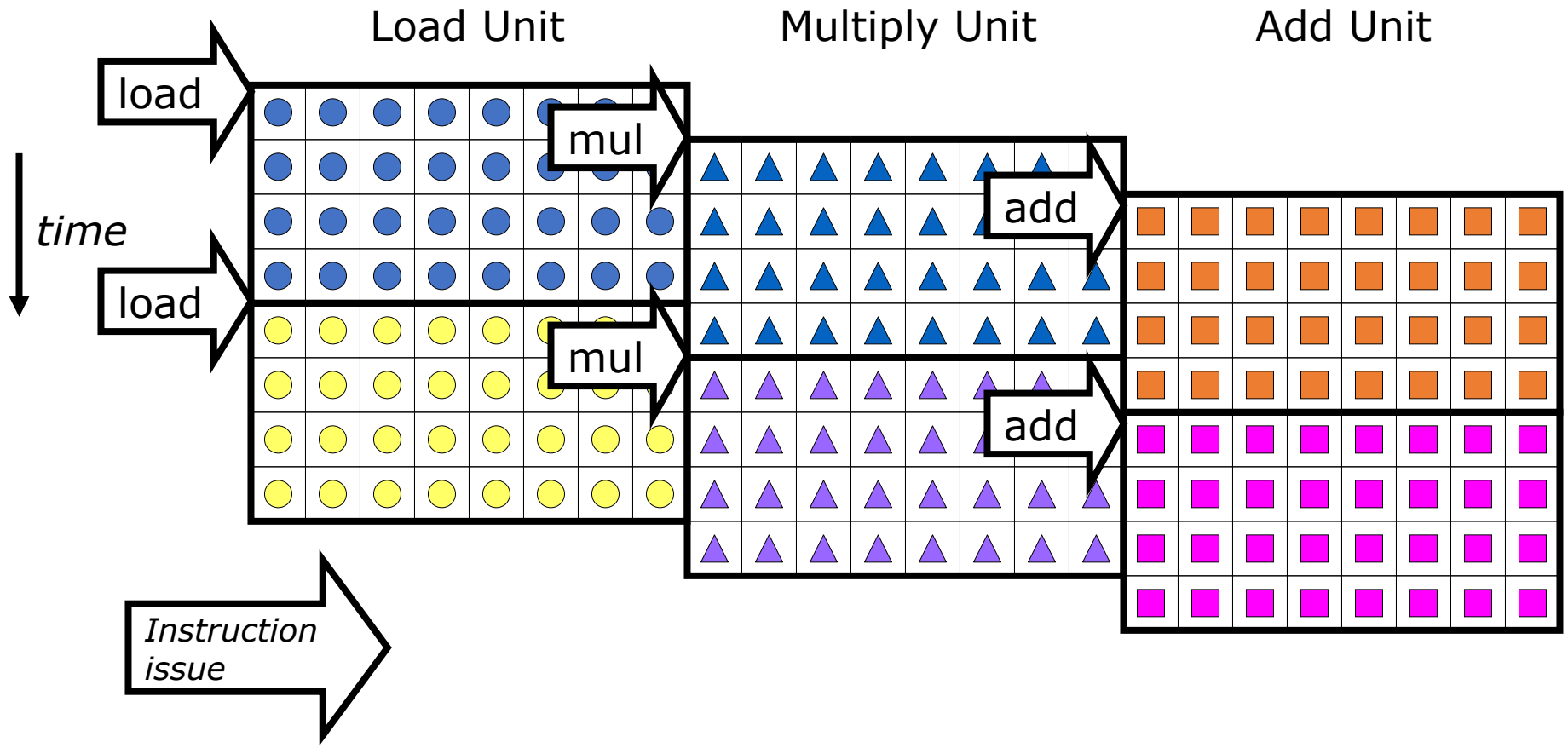


Slide credit: Krste Asanovic

Vector Instruction Level Parallelism

```
vld  v1
vmul v3,v1,v2
vadd v5,v3,v4
```

- Can overlap execution of multiple vector instructions
  - example machine has **32 elements per vector register** and 8 lanes



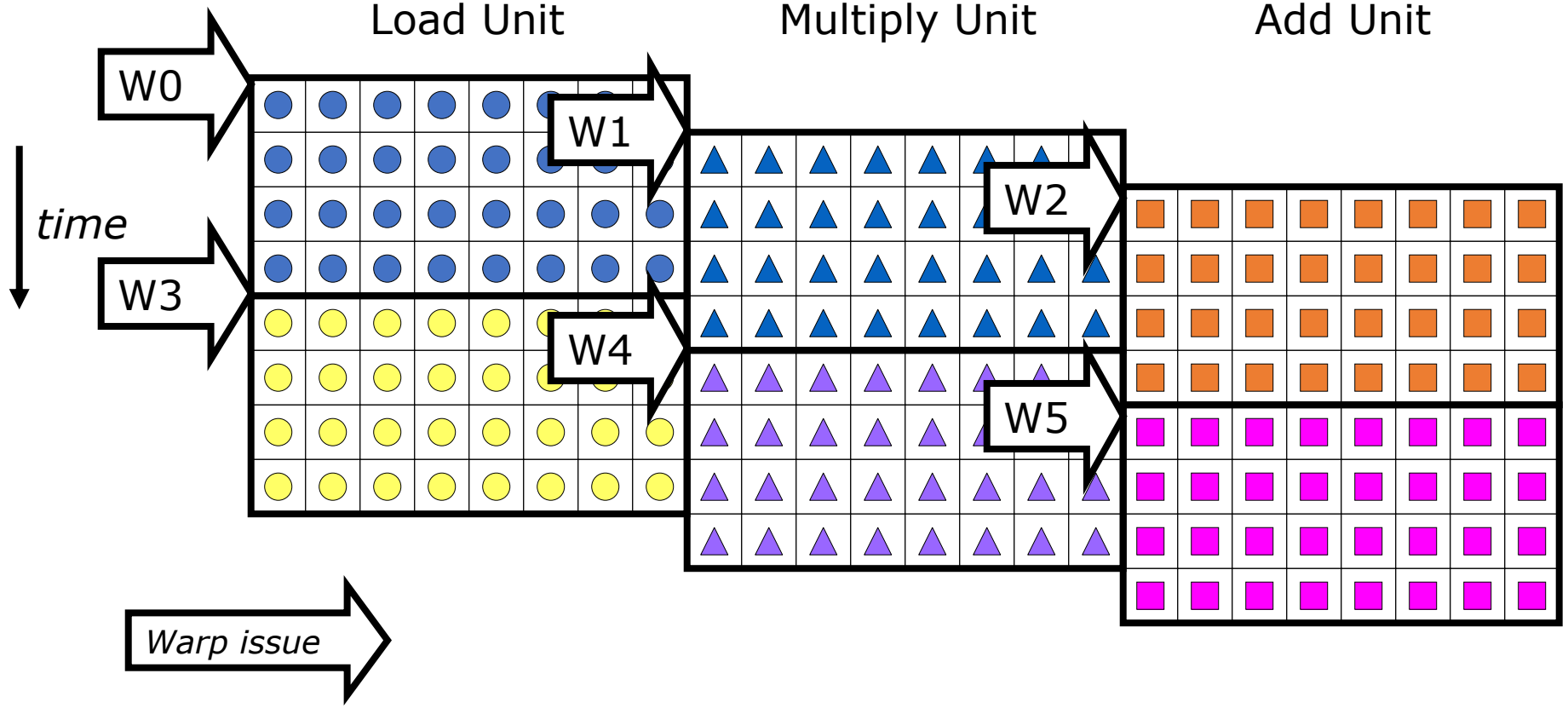
Complete 24 operations/cycle while issuing 1 short instruction/cycle

Slide credit: Krste Asanovic

Vector Instruction Level Parallelism

- Can overlap execution of multiple vector instructions
  - Example machine has **32 threads per warp and 8 lanes**

```
vld  v1
vmul v3,v1,v2
vadd v5,v3,v4
```



Complete 24 operations/cycle while issuing 1 short instruction/cycle

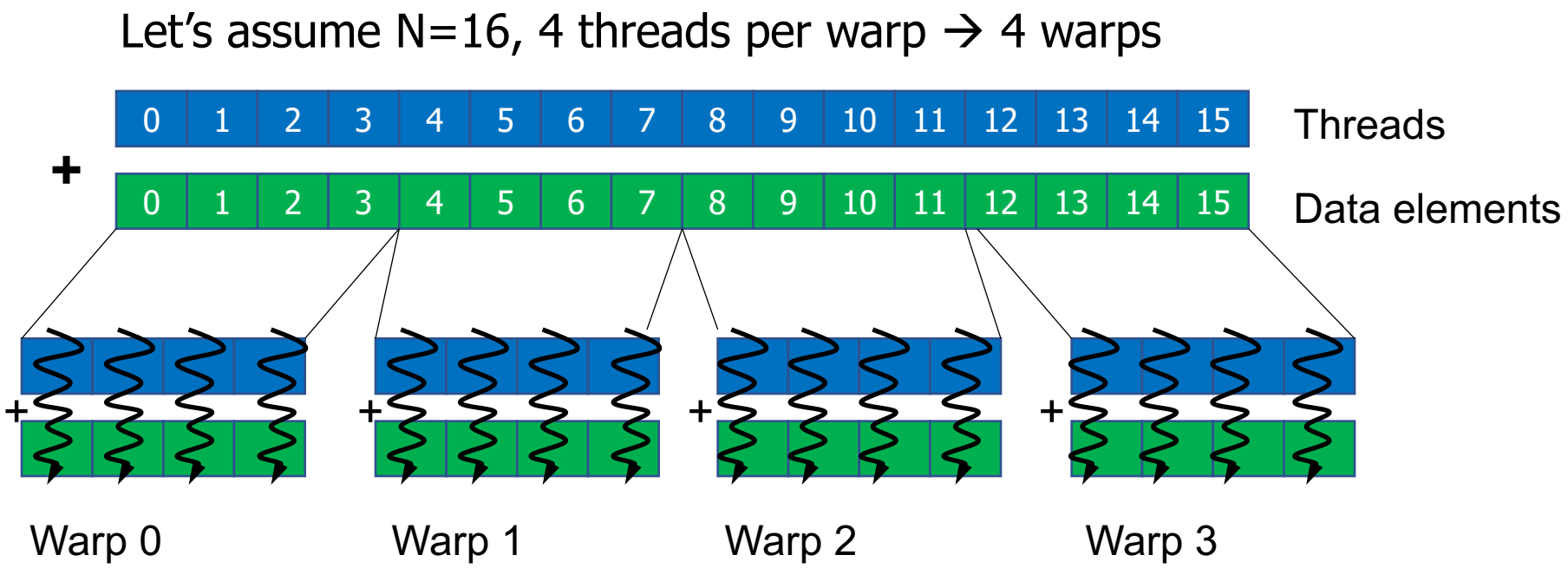
Slide credit: Krste Asanovic

SIMT Memory Access (Loads and Stores)

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

SIMT Memory Access (Loads and Stores)

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



For maximum performance, memory should provide enough bandwidth
(i.e., elements per cycle throughput to match computation unit throughput)

Slide credit: Hyesoon Kim

Kernel, Block, and Grid

- **kernel**: a function that is executed on the GPU.

Simple Kernel

```
__global__ void simpleKernel(int *data) {
  int tid = threadIdx.x; // Get thread index
  data[tid] = tid * 2; // Each thread computes a value
}
```

Kernel, Block, and Grid

- **kernel**: a function that is executed on the GPU.
- **Block**: A group of **threads** that **execute the same kernel** and can **share data through shared memory**.

Kernel, Block, and Grid

- **kernel**: a function that is executed on the GPU.
- **Block**: A group of **threads** that **execute the same kernel** and can **share data through shared memory**.
  - Threads within the same block can synchronize their execution.

Thread synchronizatioin

```
__global__ void syncExample(int *data) {
  int tid = threadIdx.x;
  data[tid] += 1; // First computation
  __syncthreads();// Synchronize all threads in the block
  data[tid] *= 2; // Second computation
}
```

Kernel, Block, and Grid

- **kernel**: a function that is executed on the GPU.
- **Block**: A group of **threads** that **execute the same kernel** and can **share data through shared memory**.
  - Threads within the same block can synchronize their execution.
  - Blocks are organized into a one-, two-, or three-dimensional structure, and each block can contain a maximum number of threads, depending on the GPU's capabilities and the requirements of the kernel.

Kernel, Block, and Grid

- **kernel**: a function that is executed on the GPU.
- **Block**: A group of **threads** that **execute the same kernel** and can **share data through shared memory**.
  - Threads within the same block can synchronize their execution.
  - Blocks are organized into a one-, two-, or three-dimensional structure, and each block can contain a maximum number of threads, depending on the GPU's capabilities and the requirements of the kernel.
- **Grid**: **A collection of blocks that executes a kernel** across the GPU.

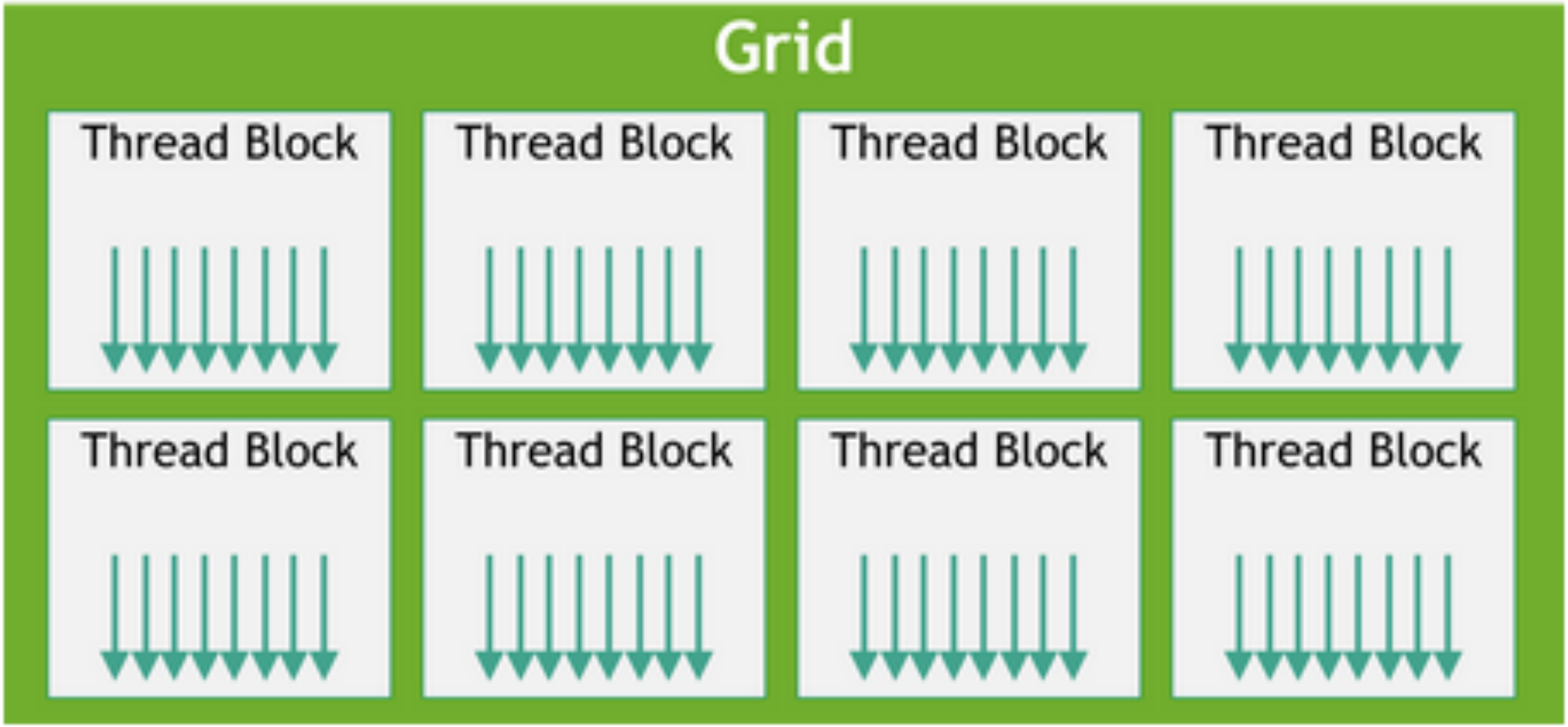
Kernel, Block, and Grid

- **kernel**: a function that is executed on the GPU.
- **Block**: A group of **threads** that **execute the same kernel** and can **share data through shared memory**.
  - Threads within the same block can synchronize their execution.
  - Blocks are organized into a one-, two-, or three-dimensional structure, and each block can contain a maximum number of threads, depending on the GPU's capabilities and the requirements of the kernel.
- **Grid**: **A collection of blocks that executes a kernel** across the GPU.
  - Like blocks, grids can be organized into one-, two-, or three-dimensional structures, depending on the computational problem being solved.

Kernel, Block, and Grid

- **kernel**: a function that is executed on the GPU.
- **Block**: A group of **threads** that **execute the same kernel** and can **share data through shared memory**.
  - Threads within the same block can synchronize their execution.
  - Blocks are organized into a one-, two-, or three-dimensional structure, and each block can contain a maximum number of threads, depending on the GPU's capabilities and the requirements of the kernel.
- **Grid**: **A collection of blocks that executes a kernel** across the GPU.
  - Like blocks, grids can be organized into one-, two-, or three-dimensional structures, depending on the computational problem being solved.
  - The grid encompasses the total number of blocks needed to complete a task, allowing the kernel to be executed in parallel across a very large number of threads.

Kernel, Block, and Grid



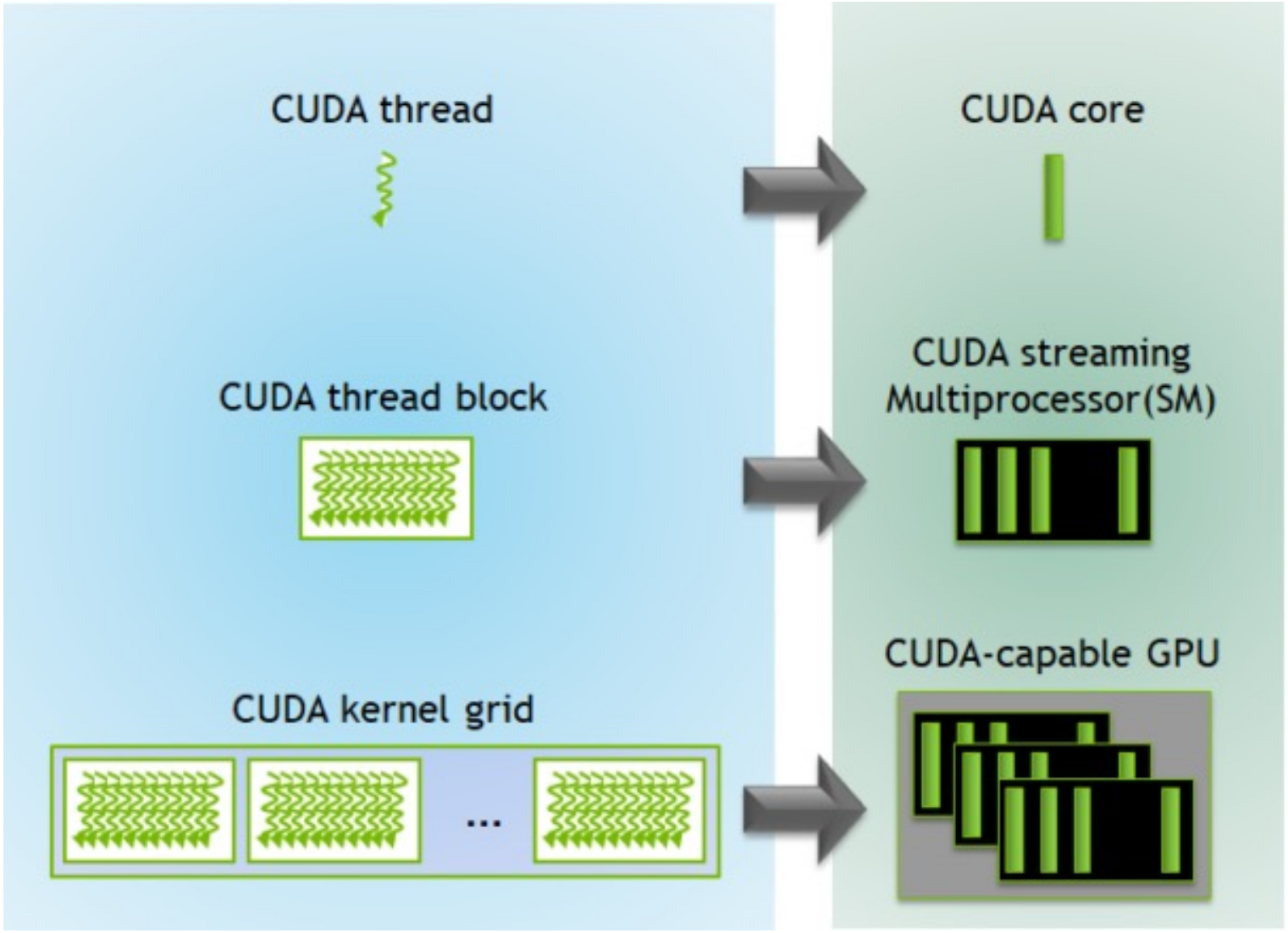
Block Dimensions

```
__global__ void kernelExample() {
  // Get thread index within a block
  int threadX = threadIdx.x;
  int threadY = threadIdx.y;

  // Get block index within the grid
  int blockX = blockIdx.x;
  int blockY = blockIdx.y;

  // Get global thread index across the entire grid
  int globalX = blockX * blockDim.x + threadX;
  int globalY = blockY * blockDim.y + threadY;

  // Print thread and block information
  printf("Block (%d, %d), Thread (%d, %d) => Global (%d, %d)\n",
  blockX, blockY, threadX, threadY, globalX, globalY);
}
```

Block Dimensions

```
int main() {
  // Define a 2D grid of (2,2) blocks
  dim3 gridDim(2, 2);

  // Define a 2D block of (3,3) threads
  dim3 blockDim(3, 3);

  // Launch kernel
  kernelExample<<<gridDim, blockDim>>>();

  // Synchronize and wait for kernel completion
  cudaDeviceSynchronize();

  return 0;
}
```

Kernel execution on GPU



Source: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Warps **not** Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections run on CPU

Warps **not** Exposed to GPU Programmers

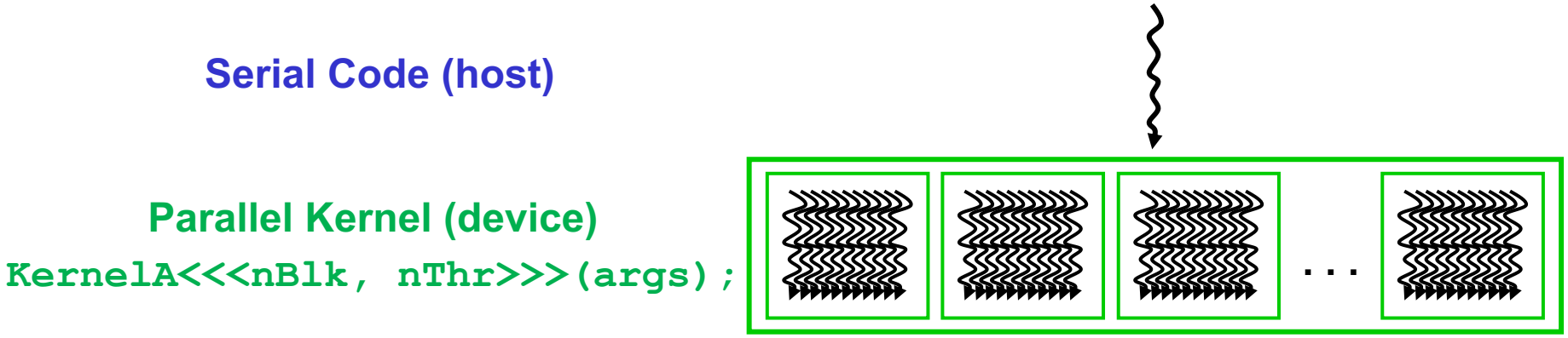
- CPU threads and GPU kernels
  - Sequential or modestly parallel sections run on CPU
  - Massively parallel sections on GPU: Blocks of threads

Warps **not** Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections run on CPU
  - Massively parallel sections on GPU: Blocks of threads

Warps **not** Exposed to GPU Programmers

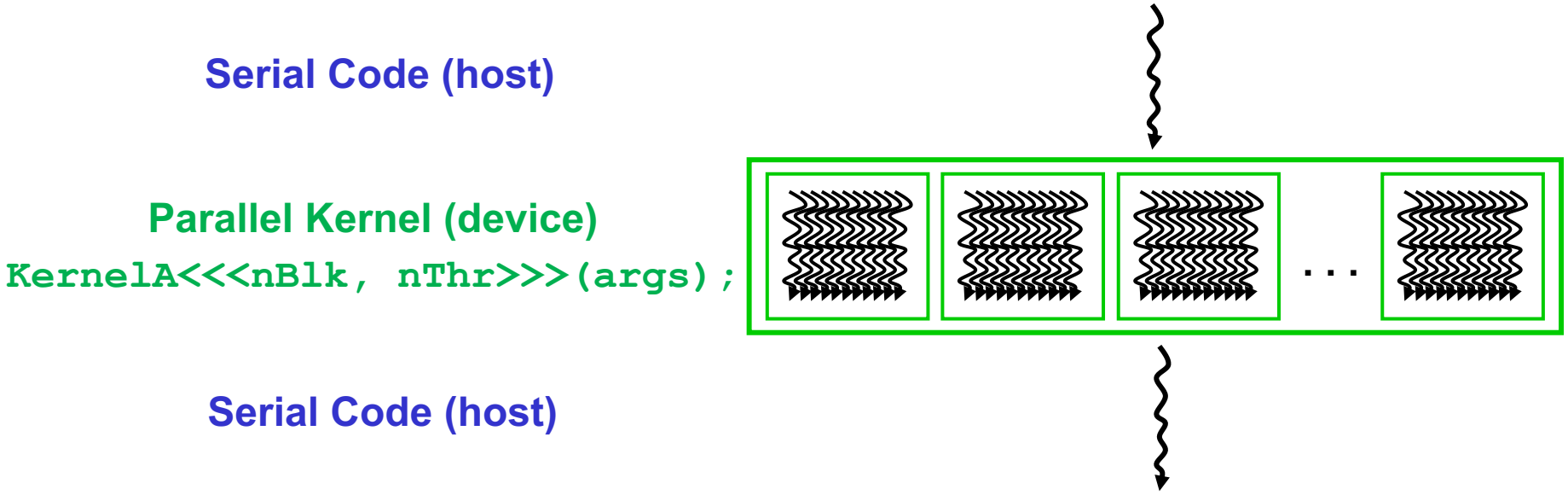
- CPU threads and GPU kernels
  - Sequential or modestly parallel sections run on CPU
  - Massively parallel sections on GPU: Blocks of threads



Slide credit: Hwu & Kirk

Warps **not** Exposed to GPU Programmers

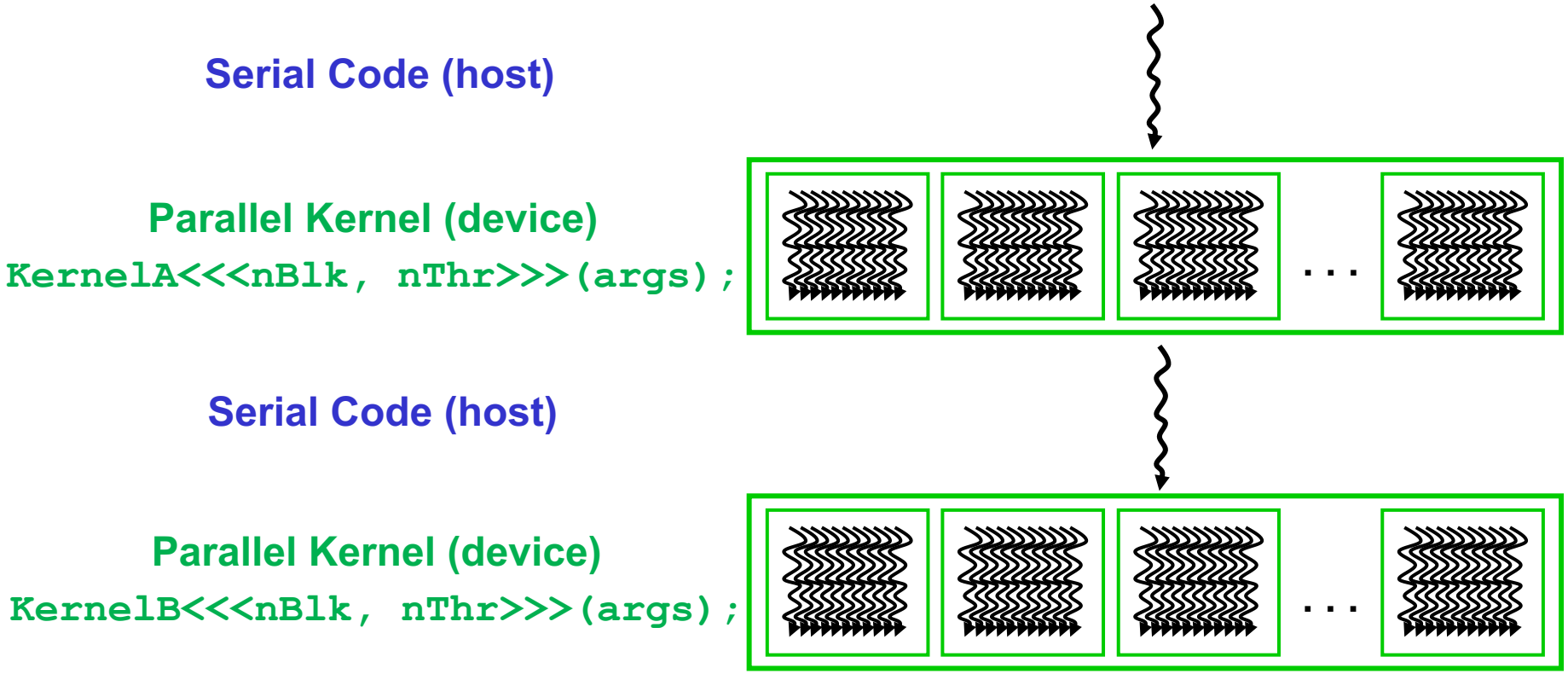
- CPU threads and GPU kernels
  - Sequential or modestly parallel sections run on CPU
  - Massively parallel sections on GPU: Blocks of threads



Slide credit: Hwu & Kirk

Warps **not** Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections run on CPU
  - Massively parallel sections on GPU: Blocks of threads

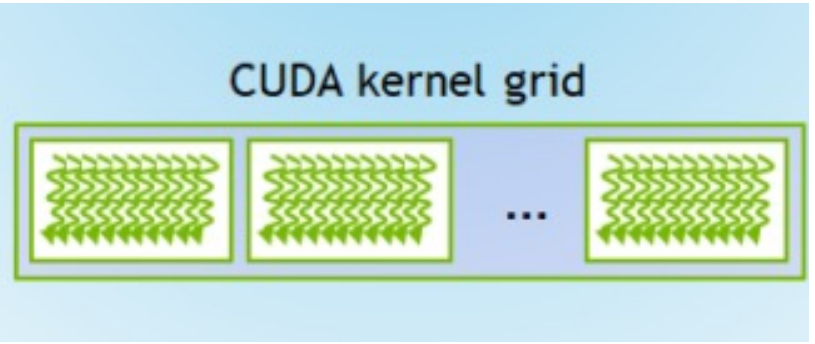


Slide credit: Hwu & Kirk

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;
}
```

Slide credit: Hyesoon Kim

Sample GPU Program (Less Simplified)

```
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);
}
```

Sample CPU code to be converted to CUDA (see next slide)

Sample GPU Program (Less Simplified)

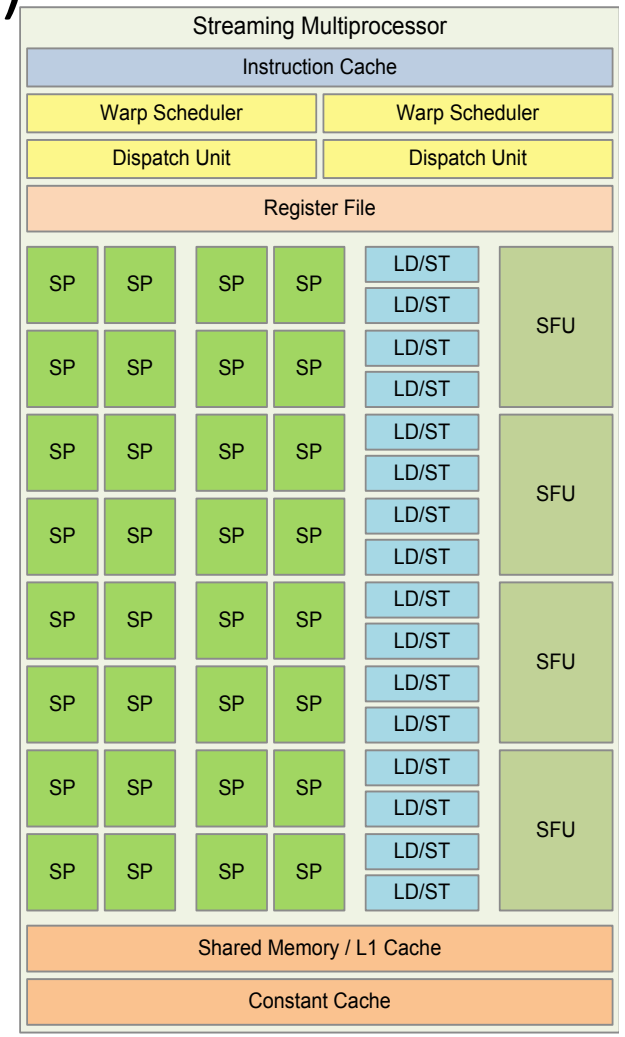
```
__global__ void add_matrix(float *a, float *b, float *c, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if (i < N && j < N) {
    c[index] = a[index] + b[index];
  }
}

int main() {
  //(setup code would be here, including memory allocation and initialization)

  dim3 dimBlock(blocksize, blocksize);
  dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
  add_matrix<<<dimGrid, dimBlock>>>(a, b, c, N);

  // ... (cleanup code would be here, including memory deallocation)
}
```

Equivalent CUDA code

Example credit: Hyesoon Kim

NVIDIA Fermi GPU Architecture (2010)

- GPU core:
  - Streaming Processor (SP)
    - AKA CUDA Core
  - Streaming Multiprocessor (SM): Many such SPs
    - A SIMD pipeline

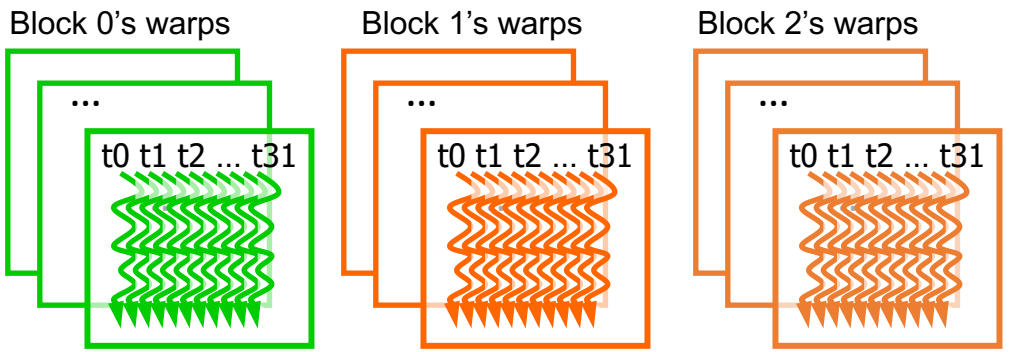


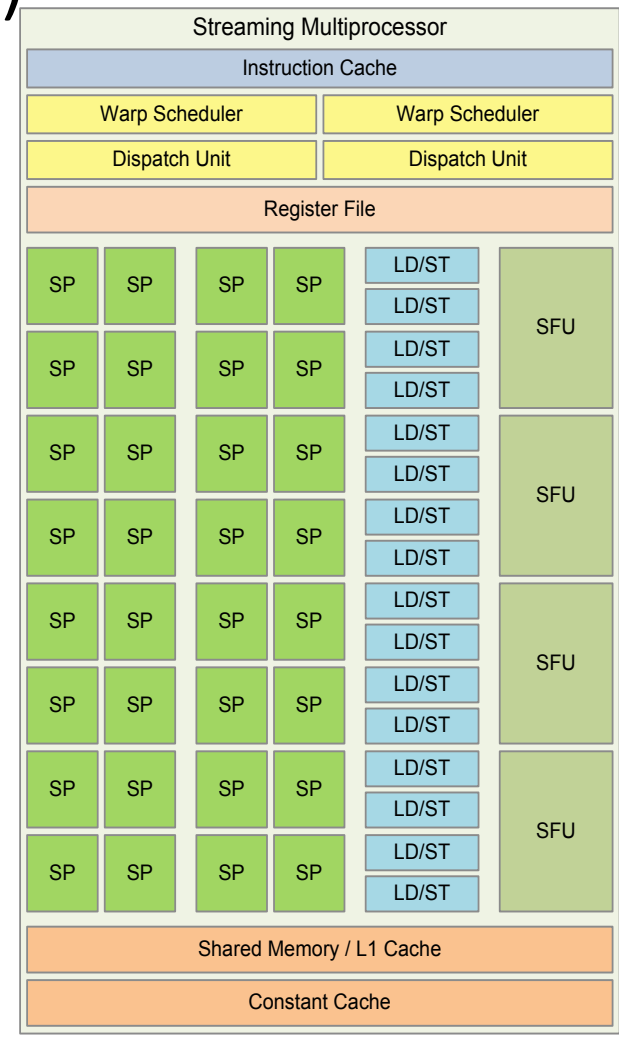
NVIDIA Fermi architecture

NVIDIA Fermi GPU Architecture (2010)

- GPU core:
  - Streaming Processor (SP)
    - AKA CUDA Core
  - Streaming Multiprocessor (SM): Many such SPs
    - A SIMD pipeline

- Blocks are divided into warps
  - SIMD/SIMT unit (32 threads)





NVIDIA Fermi architecture

Block: A group of **threads** that **execute the same kernel** and can **share data through shared memory**.

Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Sequential instruction execution; lock-step operations in a SIMD instruction

lock-step

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

lock-step

All processing elements execute the same instruction at the same time but operate on different data

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

lock-step

All processing elements execute the same instruction at the same time but operate on different data

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)

lock-step

All processing elements execute the same instruction at the same time but operate on different data

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

lock-step

All processing elements execute the same instruction at the same time but operate on different data

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

lock-step

All processing elements execute the same instruction at the same time but operate on different data

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

lock-step

All processing elements execute the same instruction at the same time but operate on different data

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

lock-step

All processing elements execute the same instruction at the same time but operate on different data

Essentially, it is SPMD programming model implemented on SIMD hardware

Recal: 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

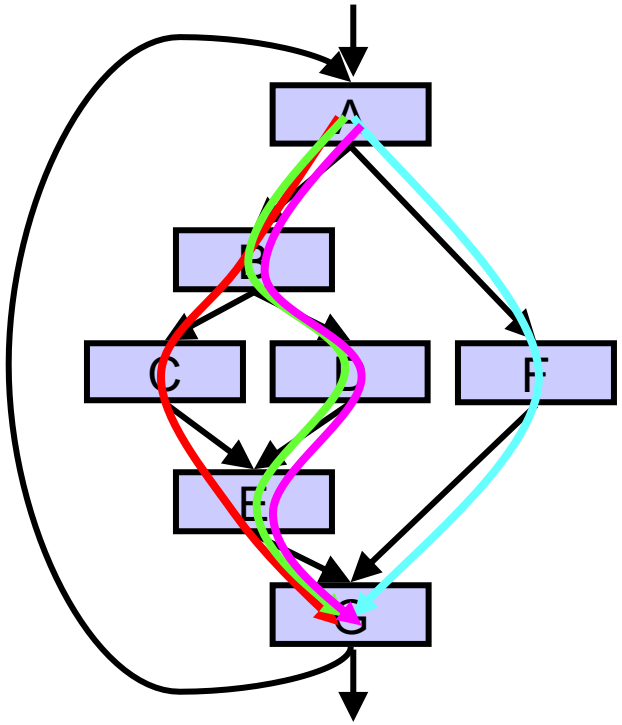
- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently (on any type of scalar pipeline) → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically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

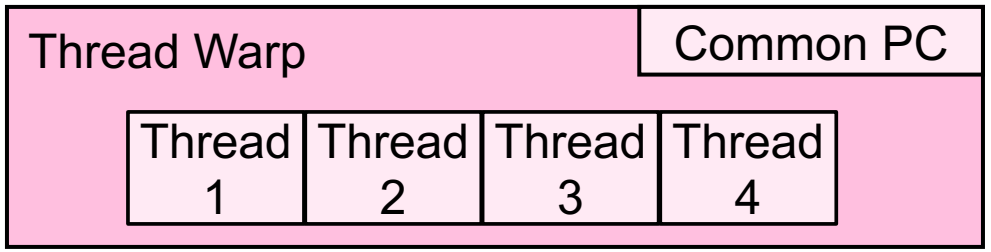
Branching in Threads

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

Branching in Threads

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

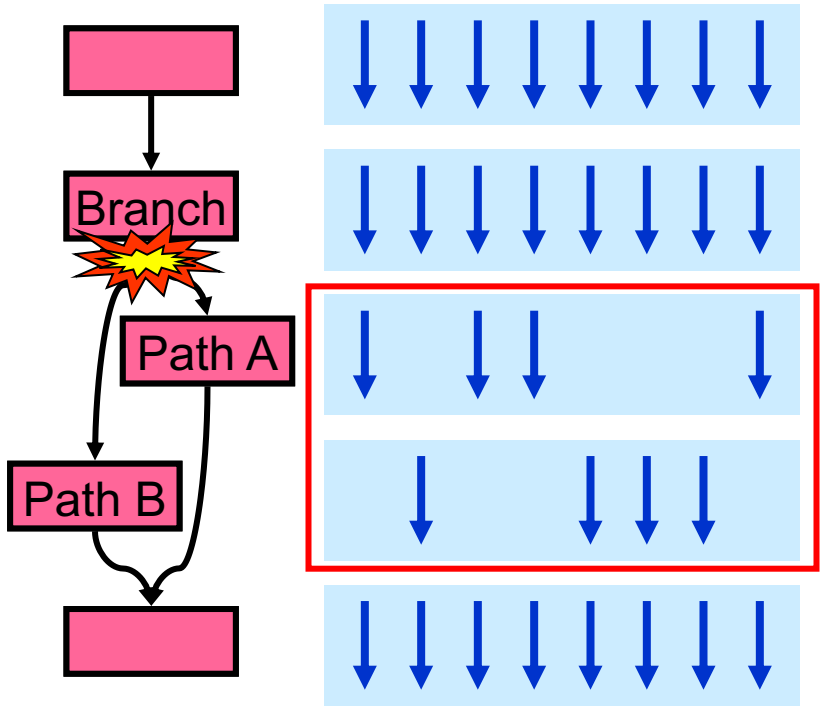




Slide credit: Tor Aamodt

Branching in Threads

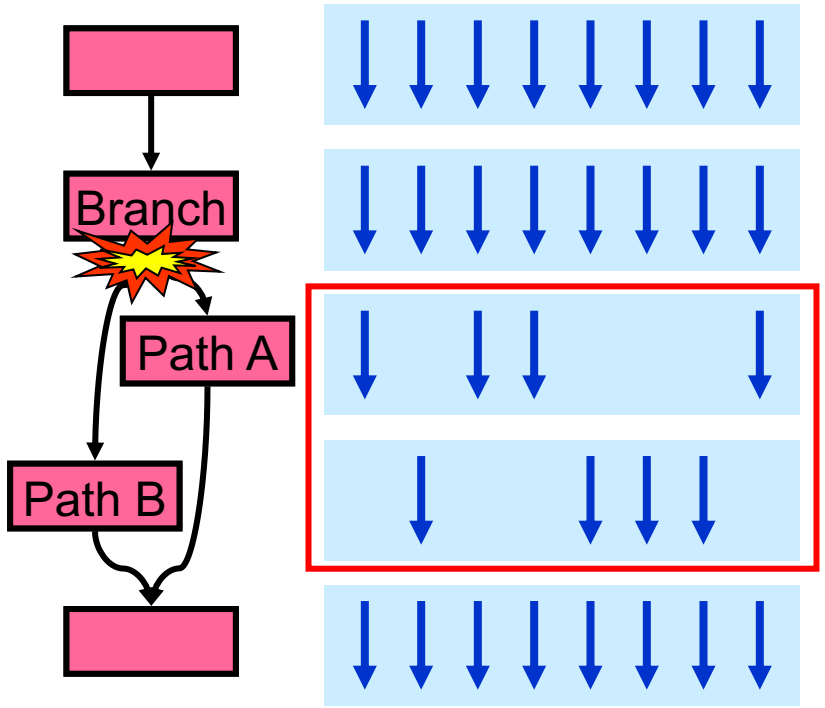
- Threads in a warp start together at the same program address, but they are free to branch and execute independently.



Slide credit: Tor Aamodt

Branching in Threads

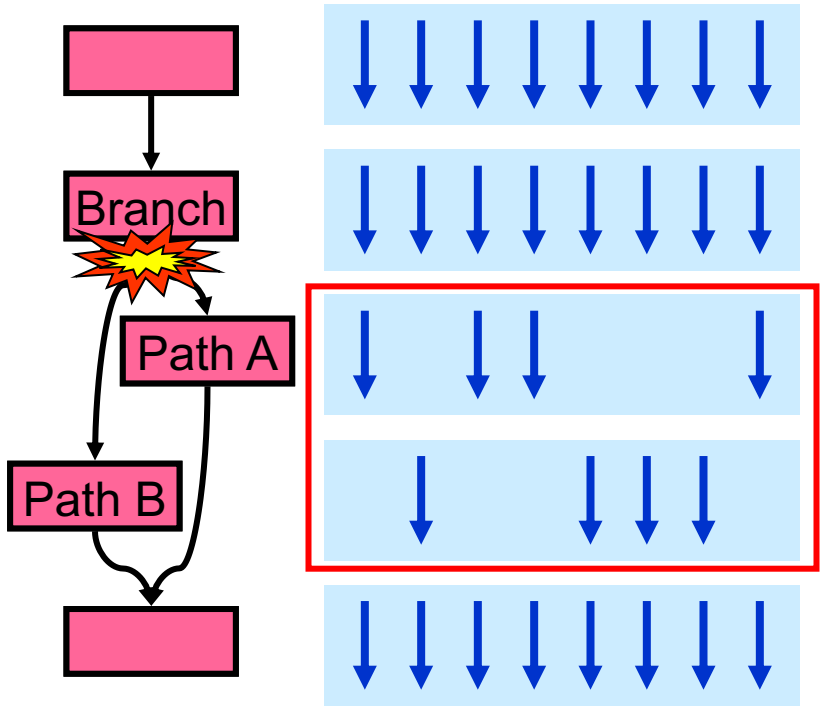
- Threads in a warp start together at the same program address, but they are free to branch and execute independently.
- Branch divergence occurs when threads inside warps branch to different execution paths



Slide credit: Tor Aamodt

Branching in Threads

- Threads in a warp start together at the same program address, but they are free to branch and execute independently.
- Branch divergence occurs when threads inside warps branch to different execution paths



If threads of a warp **diverge** via a data dependent conditional branch, the warp **serially executes** each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads **reconverge** to the original execution path.

Slide credit: Tor Aamodt

Dynamic Warp Formation/Merging

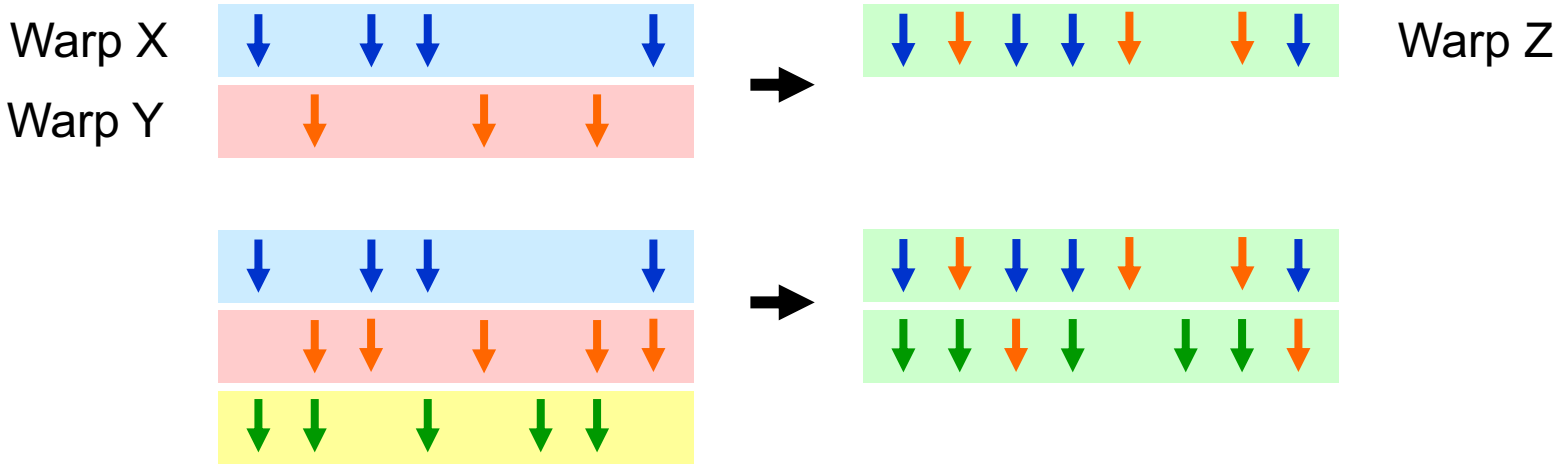
- Idea: Dynamically merge threads executing the same instruction, i.e., at the same PC (after branch divergence)

Dynamic Warp Formation/Merging

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

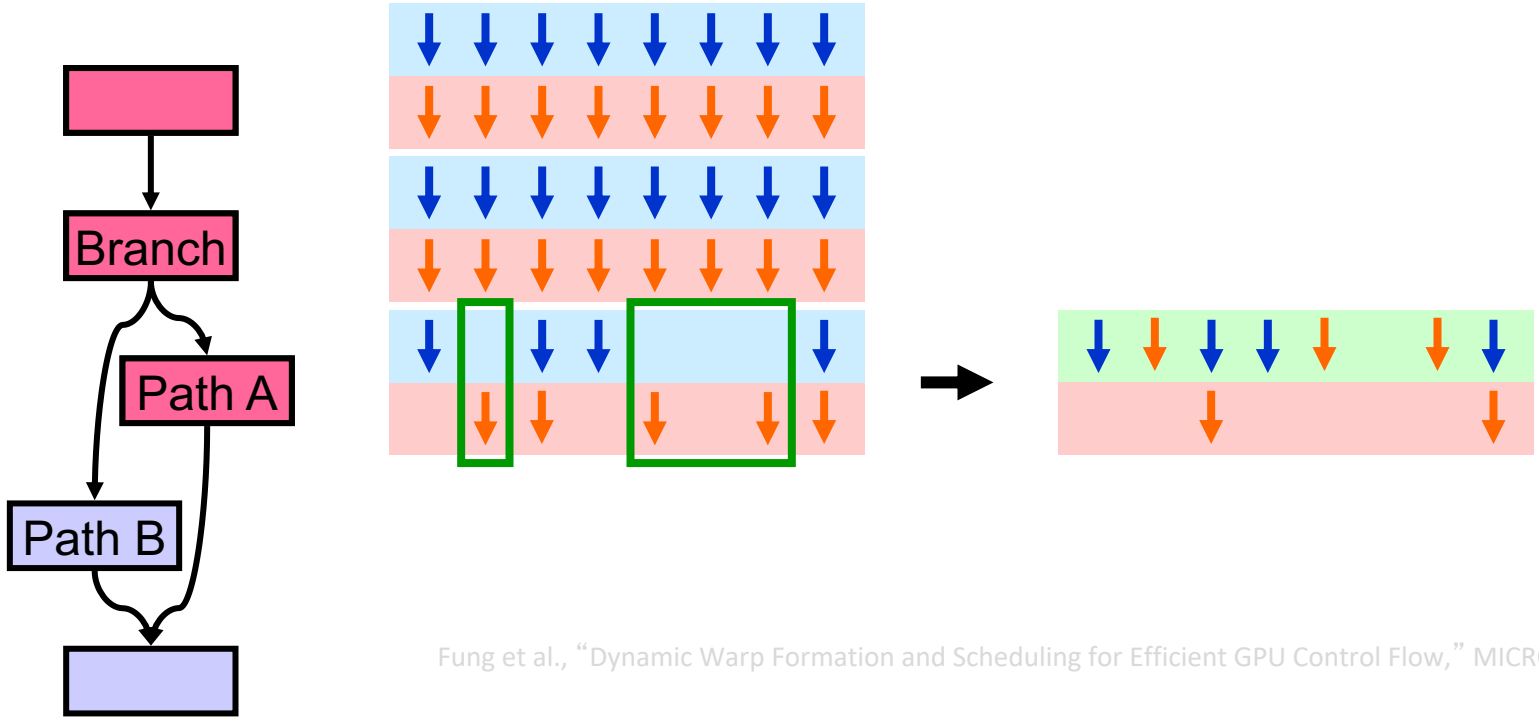
Dynamic Warp Formation/Merging

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



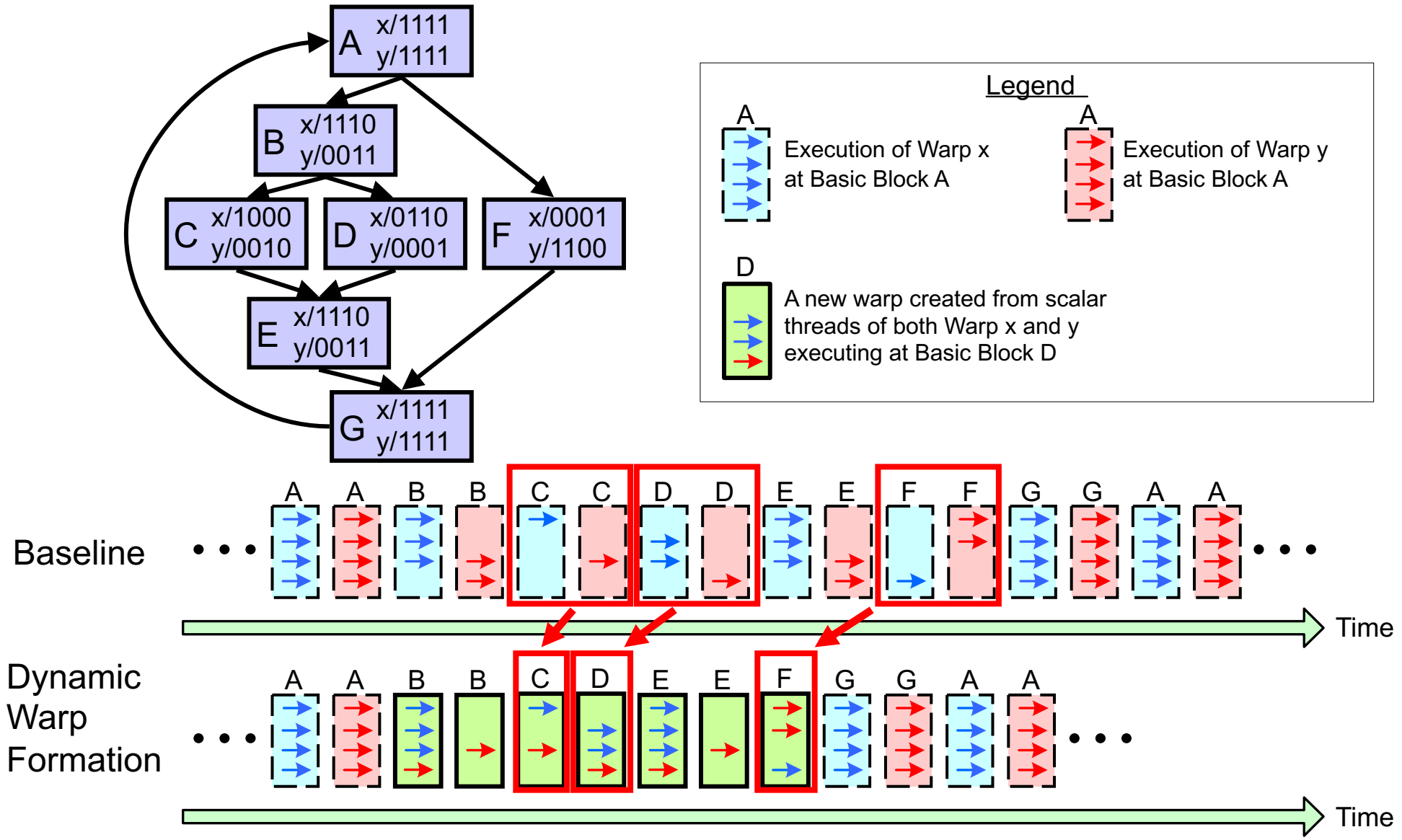
Dynamic Warp Formation/Merging

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



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

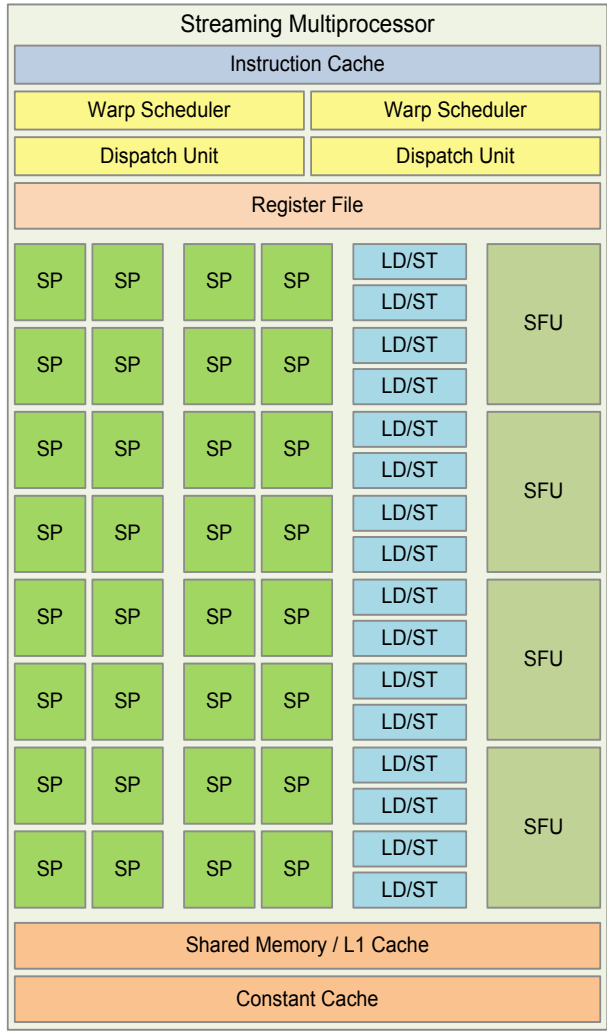
Dynamic Warp Formation Example



Example GPUs

NVIDIA GPUs

- Streaming Multiprocessors (SM) or Compute Units (CU)
  - SIMD pipelines
- Streaming Processors (SP) or CUDA ”cores”
  - Similar to Vector lanes (Each lane executes the same instruction on different data)
- Each CUDA core performs simple arithmetic operations but in massive parallelism.
- Number of SMs x SPs across generations
  - Tesla (2007): 30 x 8
  - Fermi (2010): 16 x 32
  - Kepler (2012): 15 x 192
  - Maxwell (2014): 24 x 128
  - Pascal (2016): 56 x 64
  - Volta (2017): 80 x 64



NVIDIA Fermi architecture

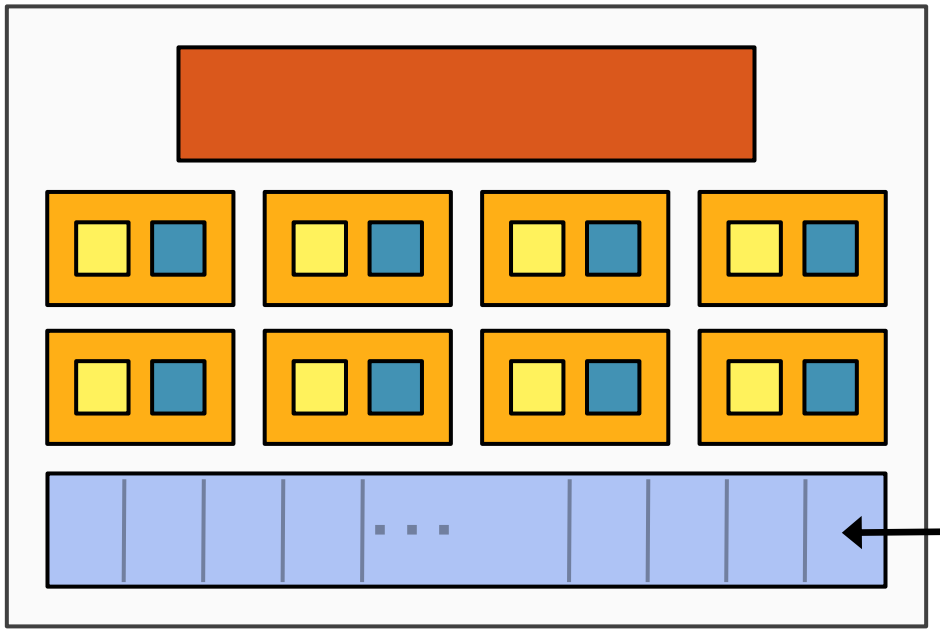
NVIDIA GeForce GTX 285

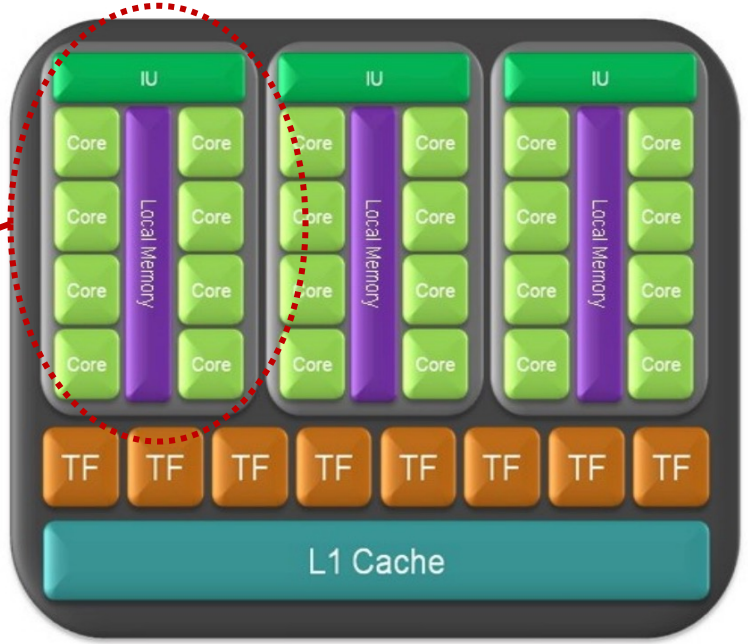
- 30 Stream multiprocessors (SM)
- 8 Stream processor (SP) per SM
- “SIMT execution”

NVIDIA, “NVIDIA GeForce GTX 200 GPU. Architectural Overview. White Paper,” 2008.

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285 (One SM)





64 KB of storage
for thread contexts
(registers)

= SIMD functional unit (SP), control shared across 8 units

= multiply-add

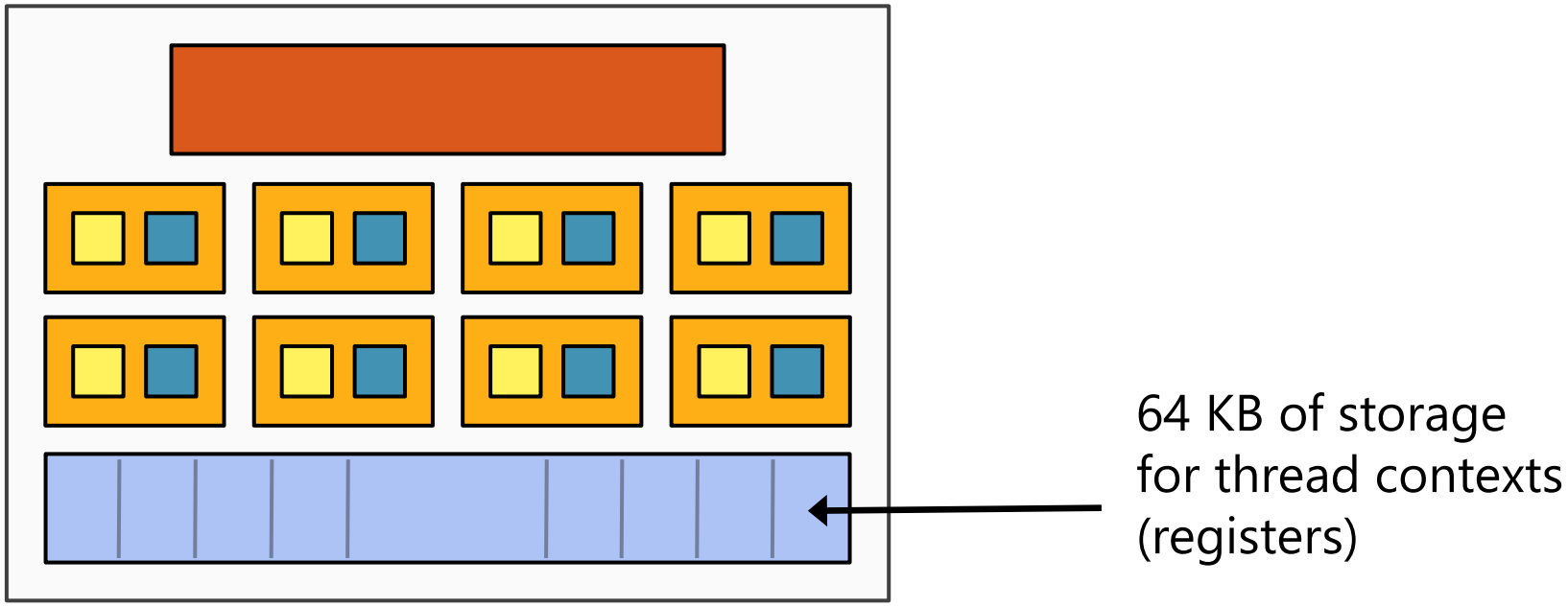
= multiply

= instruction stream decode

= execution context storage

Slide credit: Kayvon Fatahalian

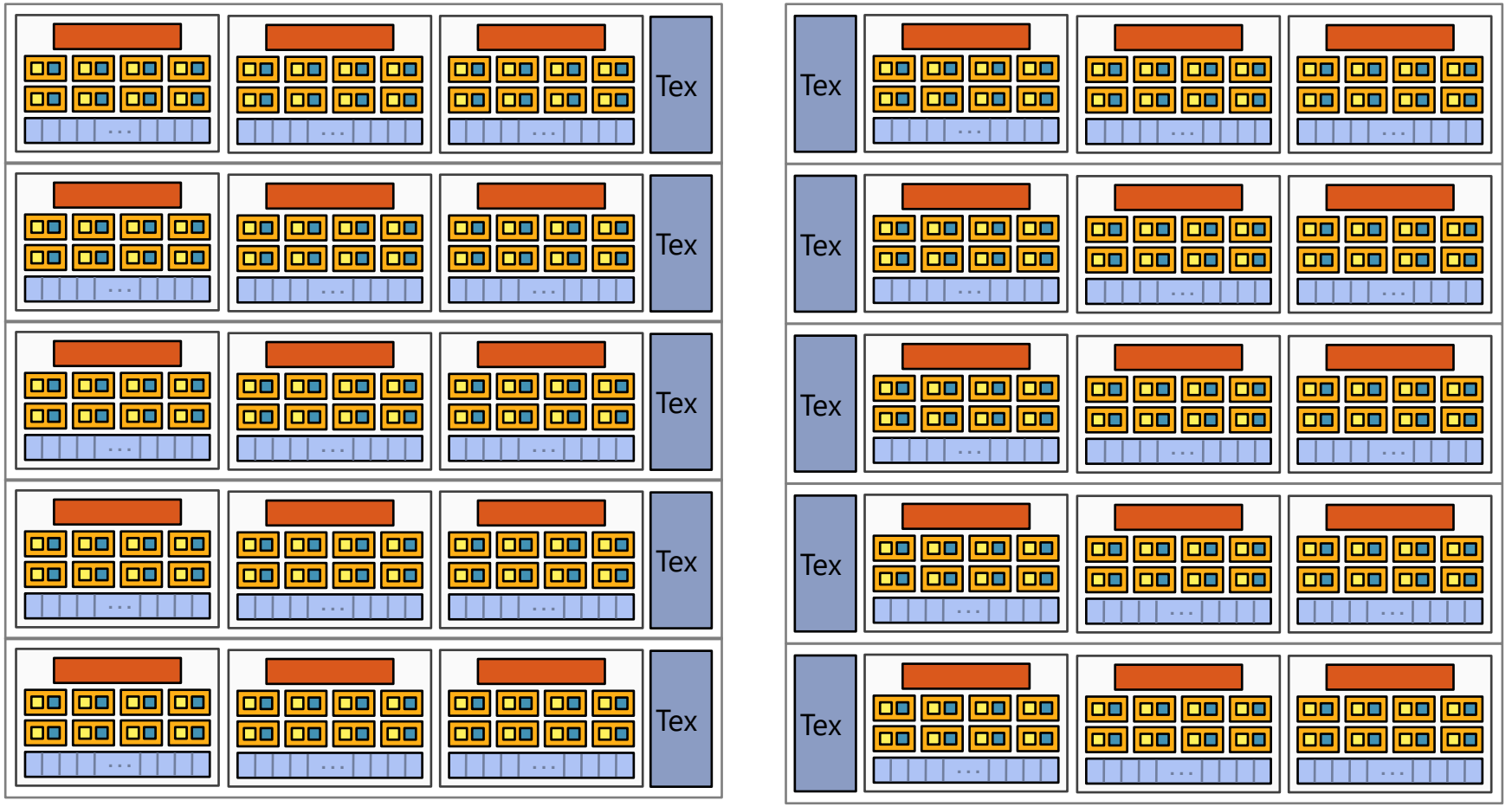
NVIDIA GeForce GTX 285



- Groups of 32 threads share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored per SM
  - A single warp (32 threads) is not enough to fully utilize an SM
  - If one warp is waiting on memory access (global memory latency ~400-600 cycles), the SM can schedule another warp instead of idling.

Slide credit: Kayvon Fatahalian

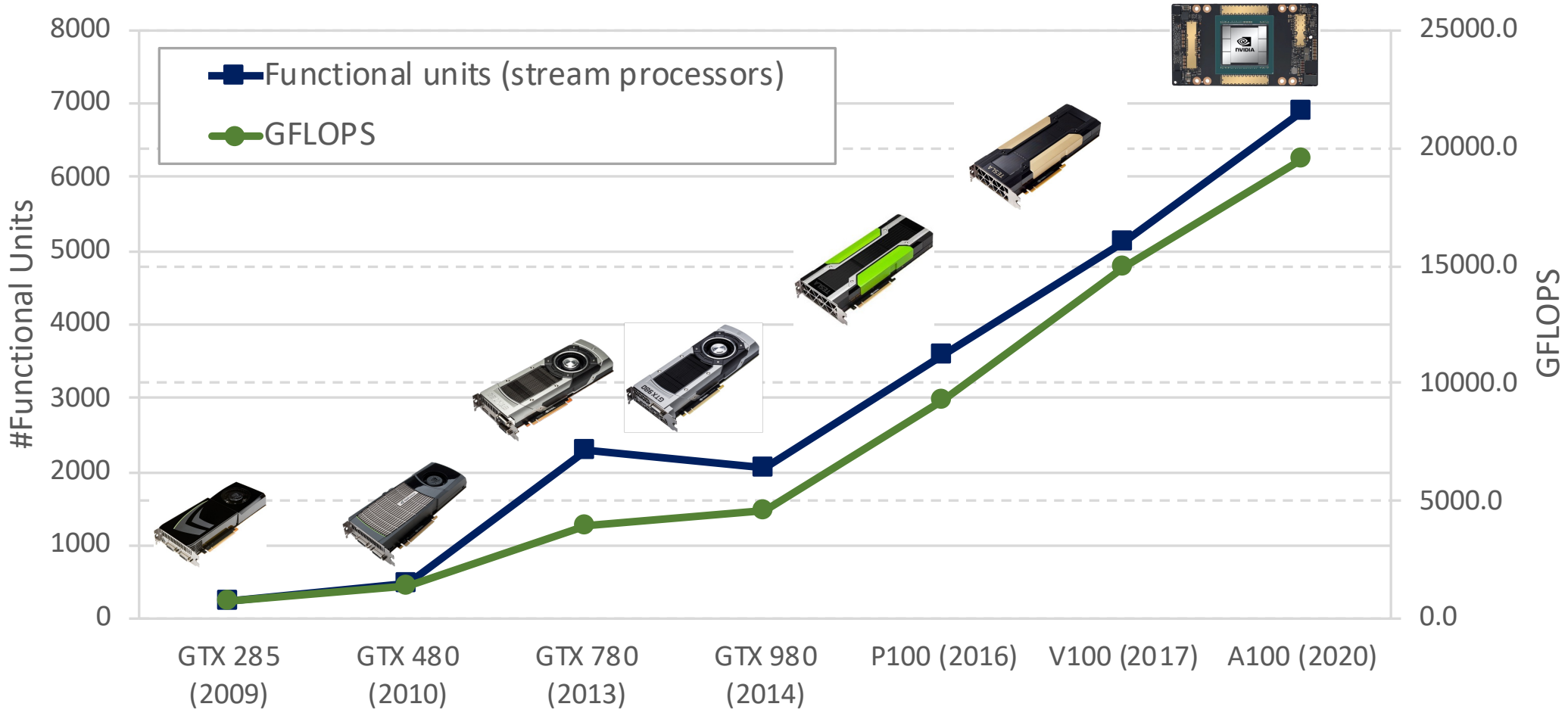
NVIDIA GeForce GTX 285



30 SMs on the GTX 285: 30,720 total store threads
(960 active)

Slide credit: Kayvon Fatahalian

NVIDIA GPUs


Functional units (stream processors)
GFLOPS
#Functional Units
GFLOPS
8000
7000
6000
5000
4000
3000
2000
1000
0
25000.0
20000.0
15000.0
10000.0
5000.0
0.0
GTX 285 (2009)
GTX 480 (2010)
GTX 780 (2013)
GTX 980 (2014)
P100 (2016)
V100 (2017)
A100 (2020)

NVIDIA V100

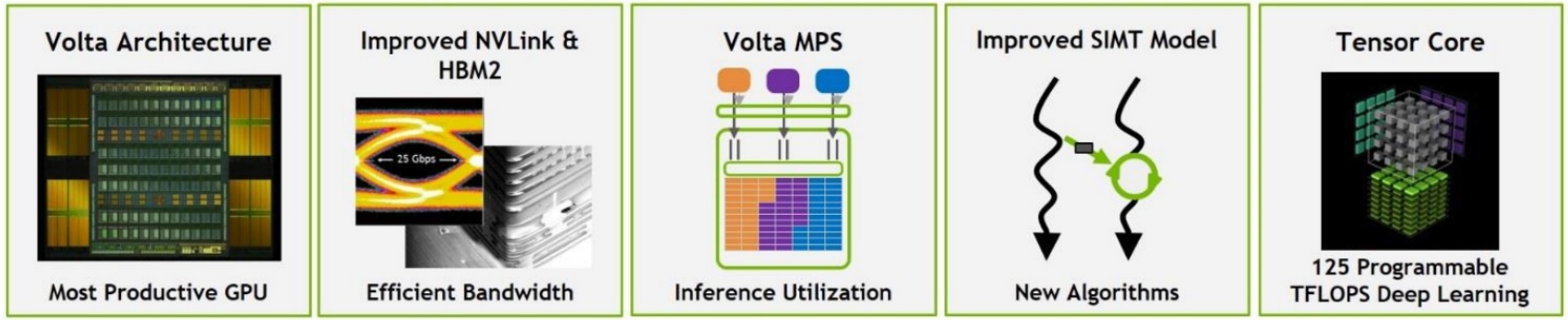
- **CUDA Cores (SPs)**:
  - 5,120 CUDA cores for the 16 GB HBM2 memory version.
  - 5,248 CUDA cores for the 32 GB HBM2 memory version.

NVIDIA V100

- **CUDA Cores (SPs)**:
  - 5,120 CUDA cores for the 16 GB HBM2 memory version.
  - 5,248 CUDA cores for the 32 GB HBM2 memory version.
- **Streaming Multiprocessors (SMs)**:
  - 80 SMs for the 16 GB version.
  - 82 SMs for the 32 GB version.

NVIDIA V100

- **CUDA Cores (SPs)**:
  - 5,120 CUDA cores for the 16 GB HBM2 memory version.
  - 5,248 CUDA cores for the 32 GB HBM2 memory version.
- **Streaming Multiprocessors (SMs)**:
  - 80 SMs for the 16 GB version.
  - 82 SMs for the 32 GB version.
- **Memory**:
  - High Bandwidth Memory 2 (HBM2)
- **Performance**:
  - Designed for deep learning and AI with Tensor Cores for mixed-precision computing.
  - Suitable for scientific computing and high-performance computing tasks.
- **Tensor Cores**: Contains 640 Tensor Cores (in the 16 GB version).



Source: NVIDIA, “NVIDIA Tesla V100 GPU Architecture. White Paper,” 2017.

NVIDIA V100

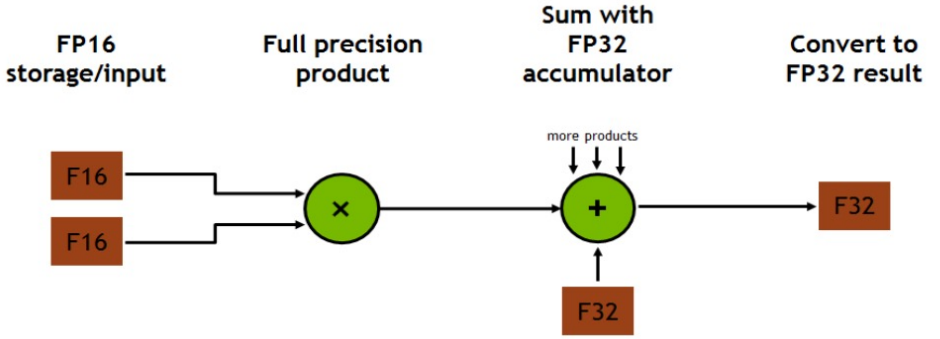
Figure 4. Volta GV100 Full GPU with 84 SM Units

Source: NVIDIA, “NVIDIA Tesla V100 GPU Architecture. White Paper,” 2017.

Tensor Cores

- Each Tensor Core operates on a 4x4 matrix and performs the following operation:

$$\boldsymbol{D} = \boldsymbol{A} \times \boldsymbol{B} + \boldsymbol{C}$$



- A, B, C, and D are 4x4 matrices.
  - A and B are FP16.
  - C and D may be FP16 or FP32 matrices.

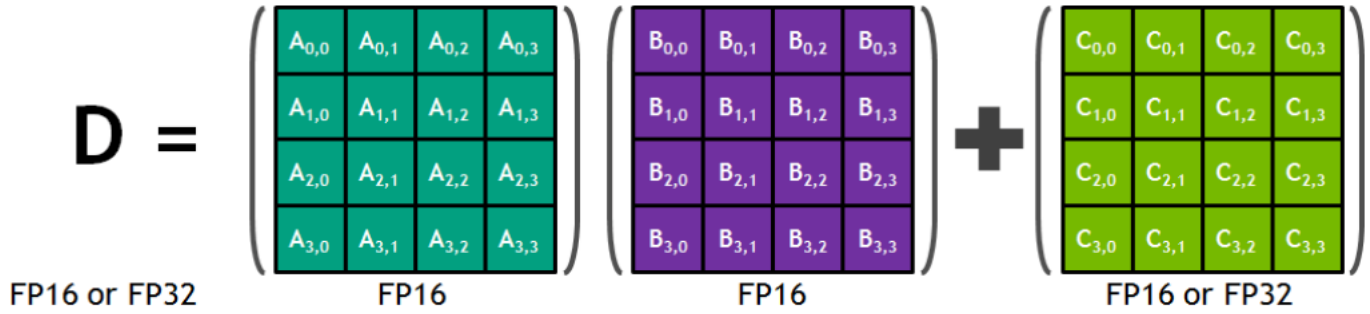
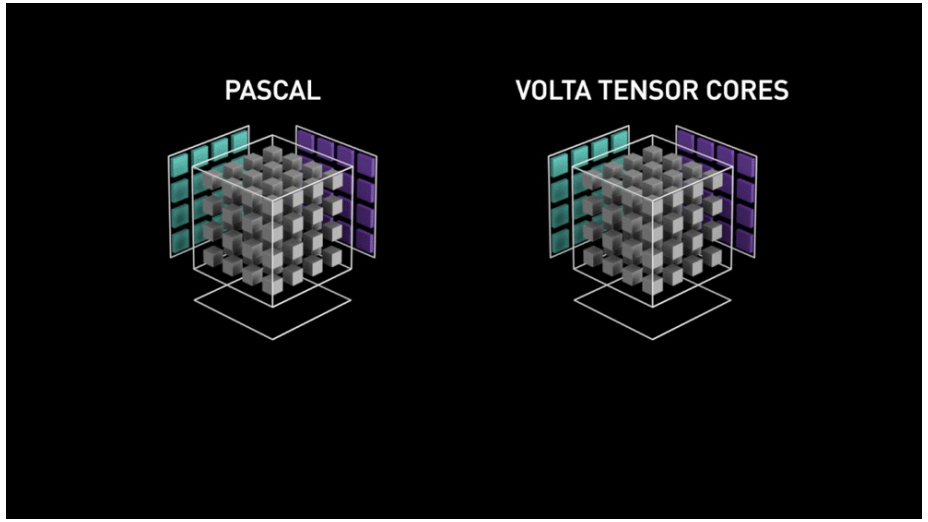
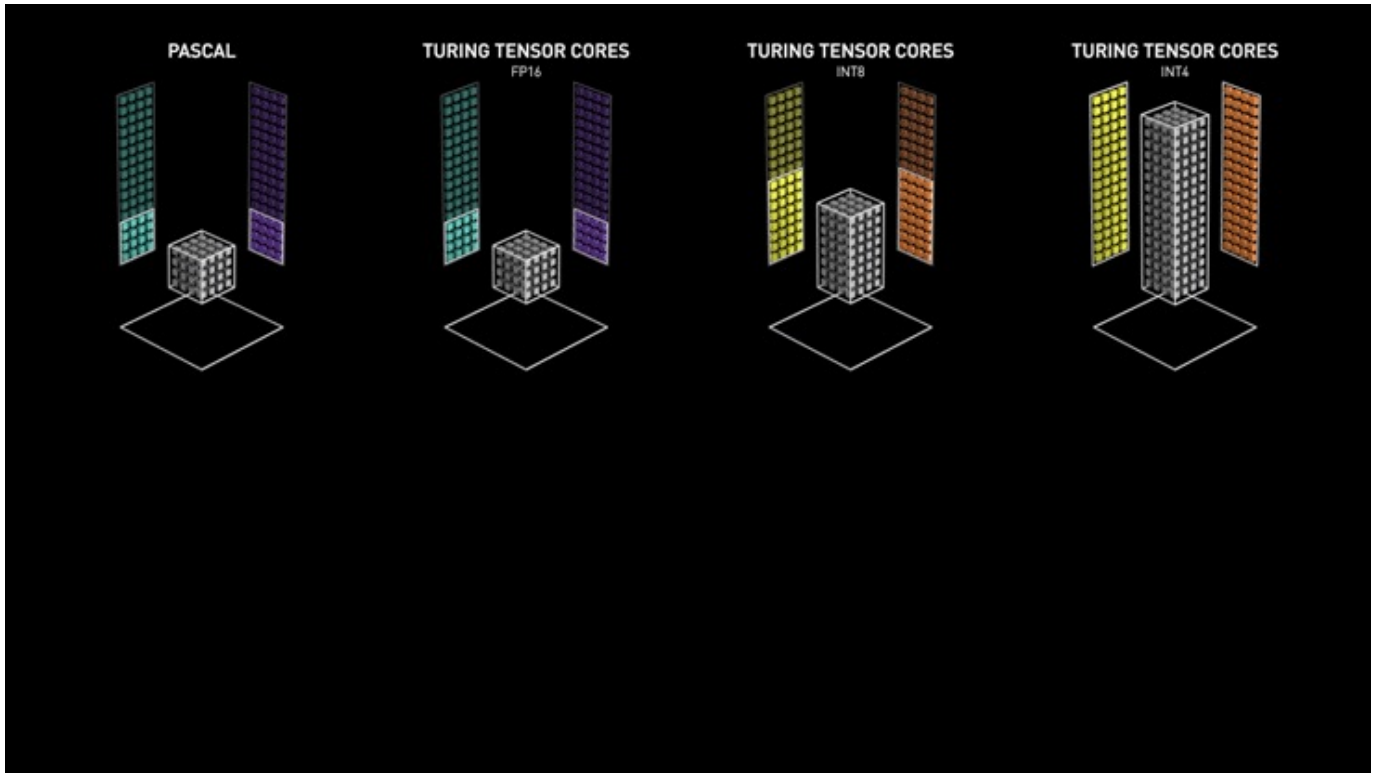


Figure 8. Tensor Core 4x4 Matrix Multiply and Accumulate

Matrix Multiplication with Tensor Cores

- Tensor cores are accessible and exposed as Warp-Level Matrix Operations in the CUDA 9 C++ API.





NVIDIA V100

Tensor core

Figure 5. Volta GV100 Streaming Multiprocessor (SM)

Source: NVIDIA, “NVIDIA Tesla V100 GPU Architecture. White Paper,” 2017.

NVIDIA A100

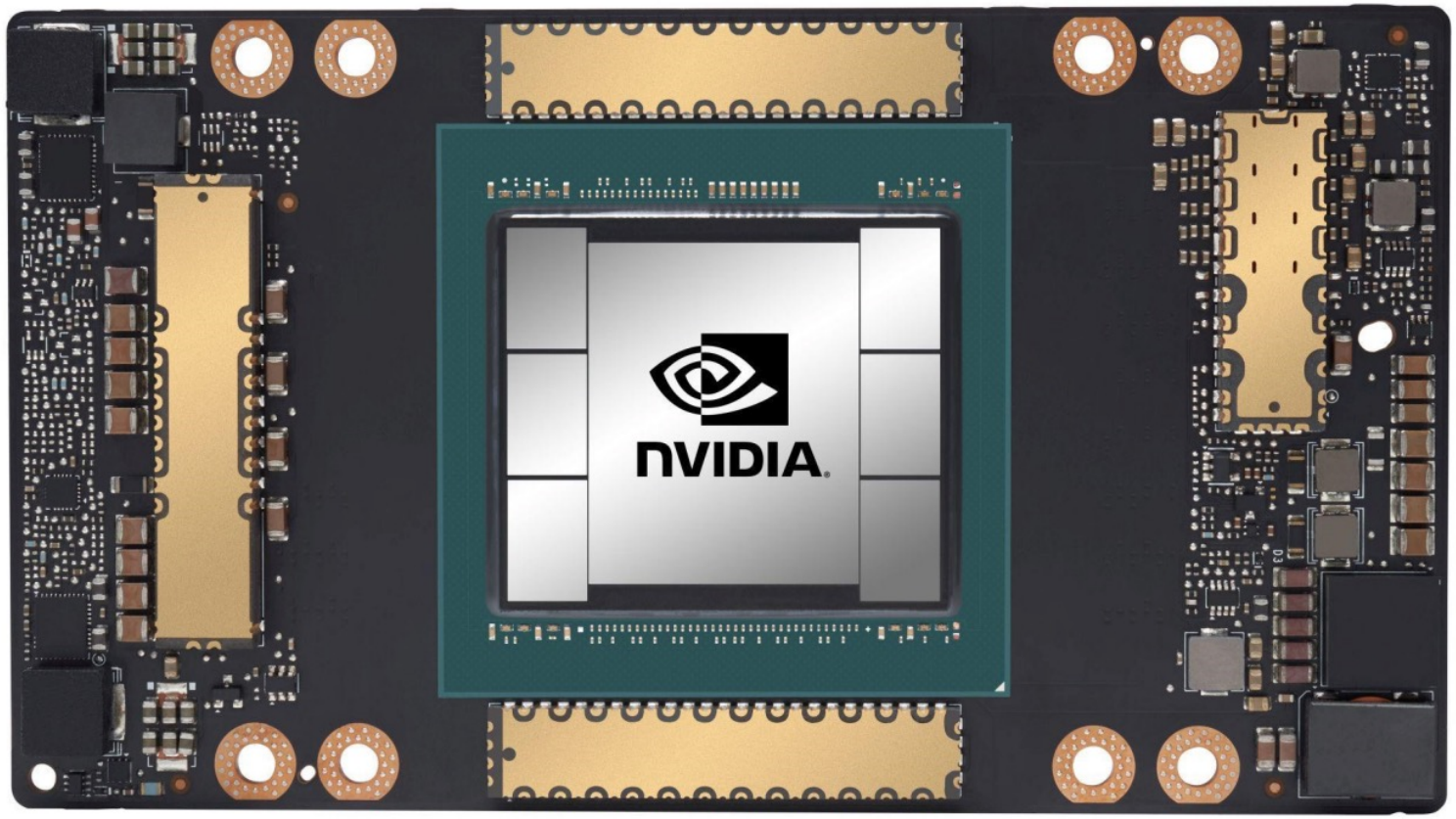
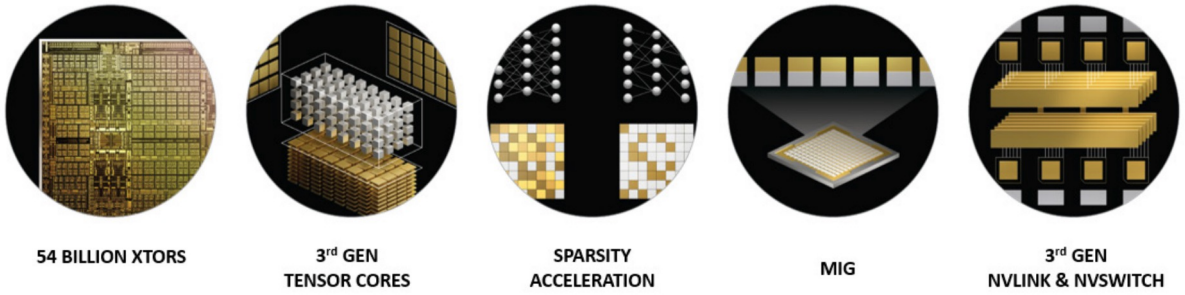


Figure 3. NVIDIA A100 GPU on new SXM4 Module

Source : NVIDIA A100 Tensor Core GPU Architecture

NVIDIA A100

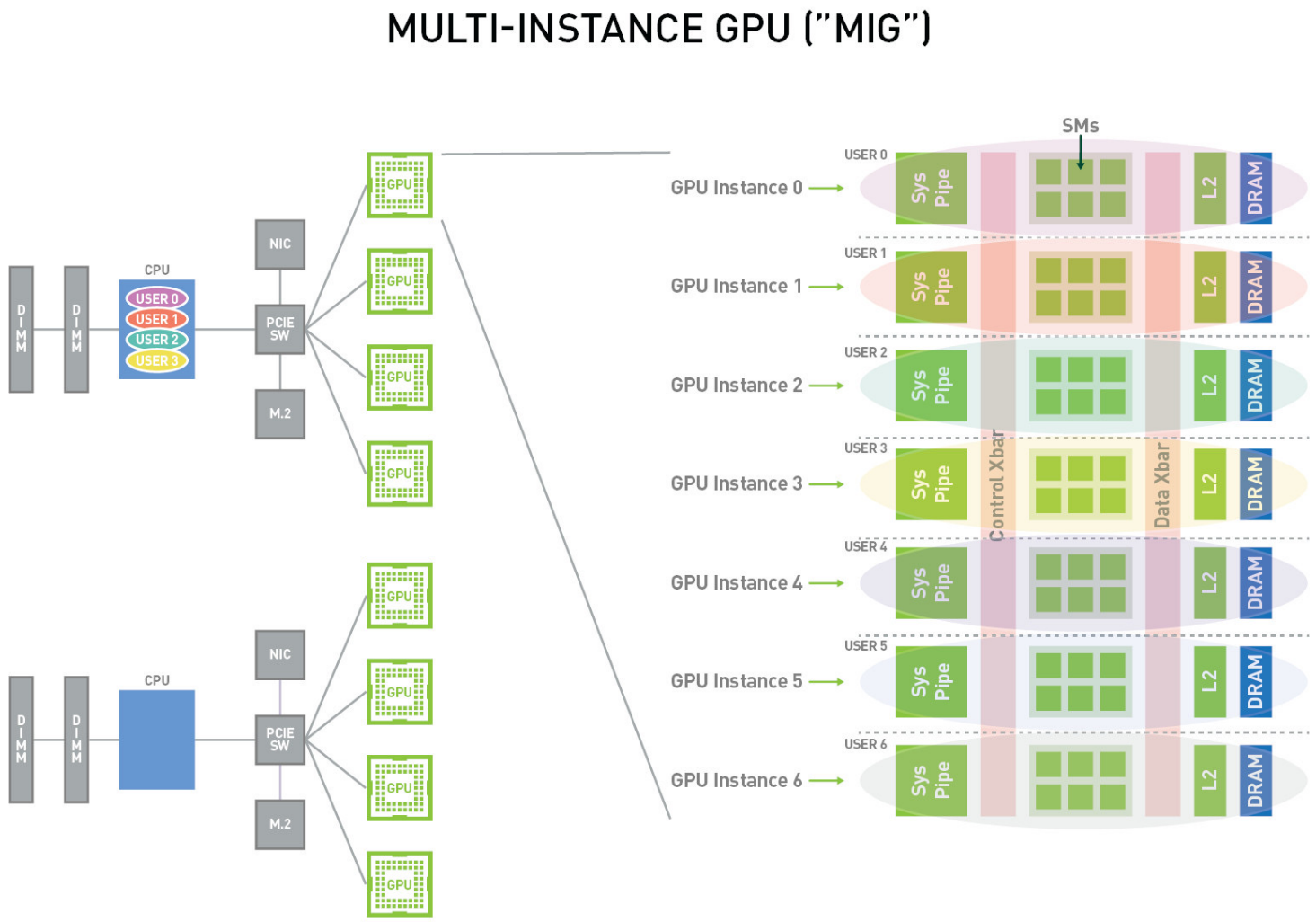
- **Architecture**: Built on the Ampere architecture.
- **CUDA Cores**:
  - 6,912 CUDA cores for parallel processing.
- **Tensor Cores**:
  - 432 Tensor Cores optimized for AI and deep learning workloads.
- **Memory**:
  - Up to 40 GB of HBM2e (High Bandwidth Memory) offering high memory bandwidth.



Source : NVIDIA A100 Tensor Core GPU Architecture

Multi Instance GPU (MIG)

- MIG allows a single NVIDIA GPU to be split into multiple independent GPUs.
- Each instance has its own compute cores, memory, and cache.
- Essential for cloud computing, AI inference, and HPC workloads.



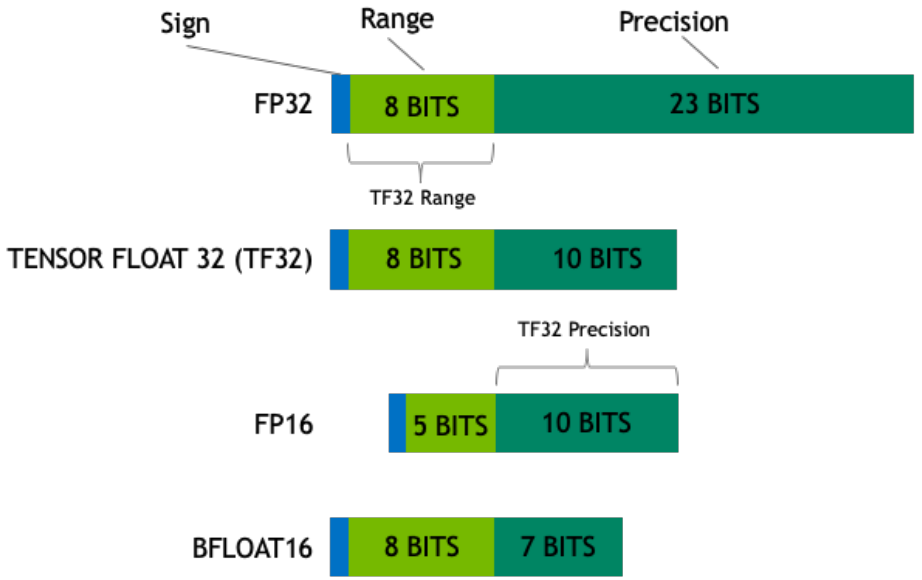
NVIDIA A100

Figure 6. GA100 Full GPU with 128 SMs (A100 Tensor Core GPU has 108 SMs)

Source : NVIDIA A100 Tensor Core GPU Architecture

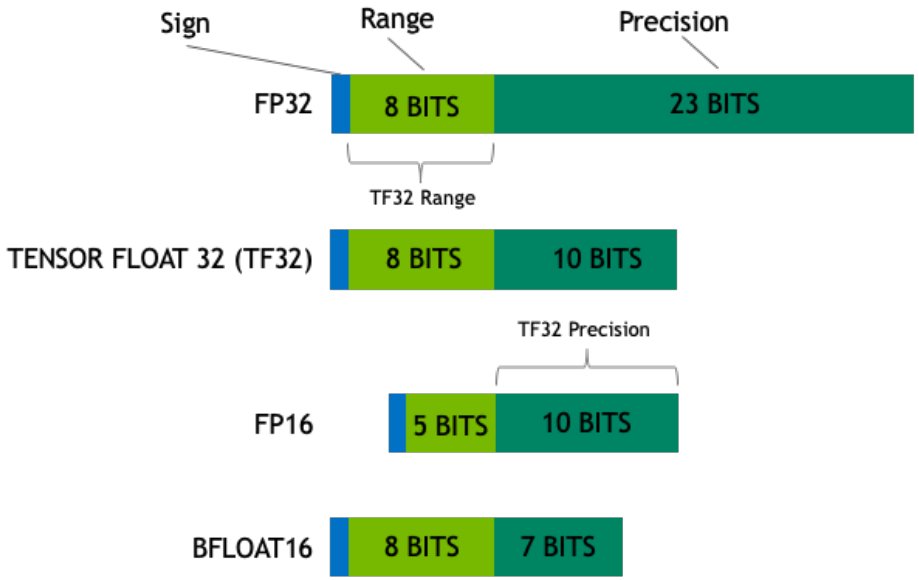
TF32 (TensorFloat-32)

- TF32:
  - 1 sign bit, 8 exponent bits, 10 mantissa bits



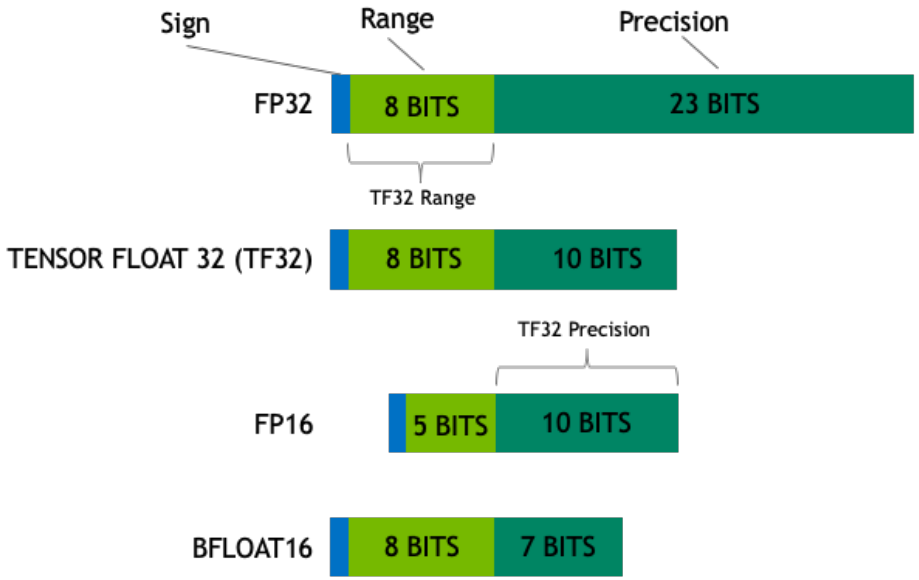
TF32

- TF32:
  - 1 sign bit, 8 exponent bits, 10 mantissa bits
- Features:
  - **Same range as FP32 (32-bit floating-point):** This ensures compatibility.



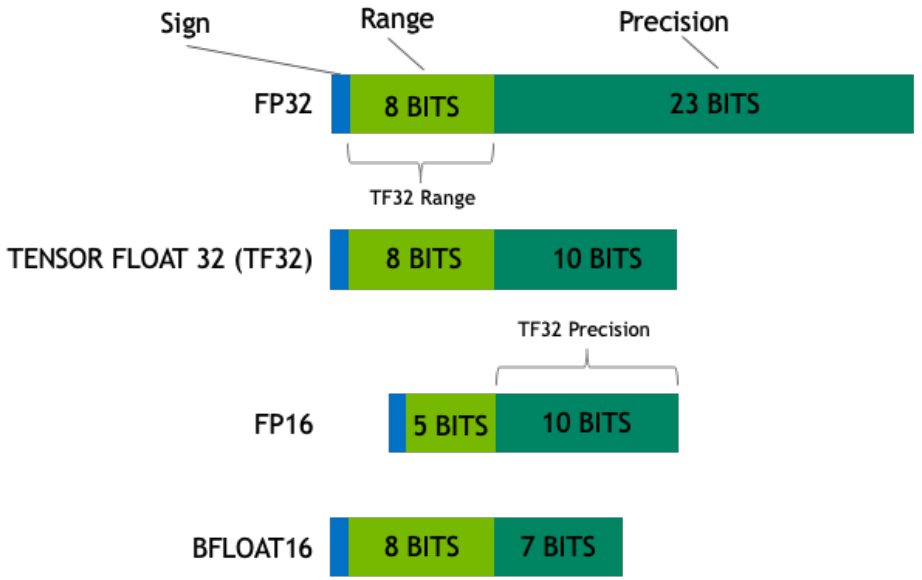
TF32

- TF32:
  - 1 sign bit, 8 exponent bits, 10 mantissa bits
- Features:
  - **Same range as FP32 (32-bit floating-point):** This ensures compatibility.
  - **Reduced precision compared to FP32:** The fewer mantissa bits in TF32 lead to slightly lower precision, but mostly sufficient for ML.

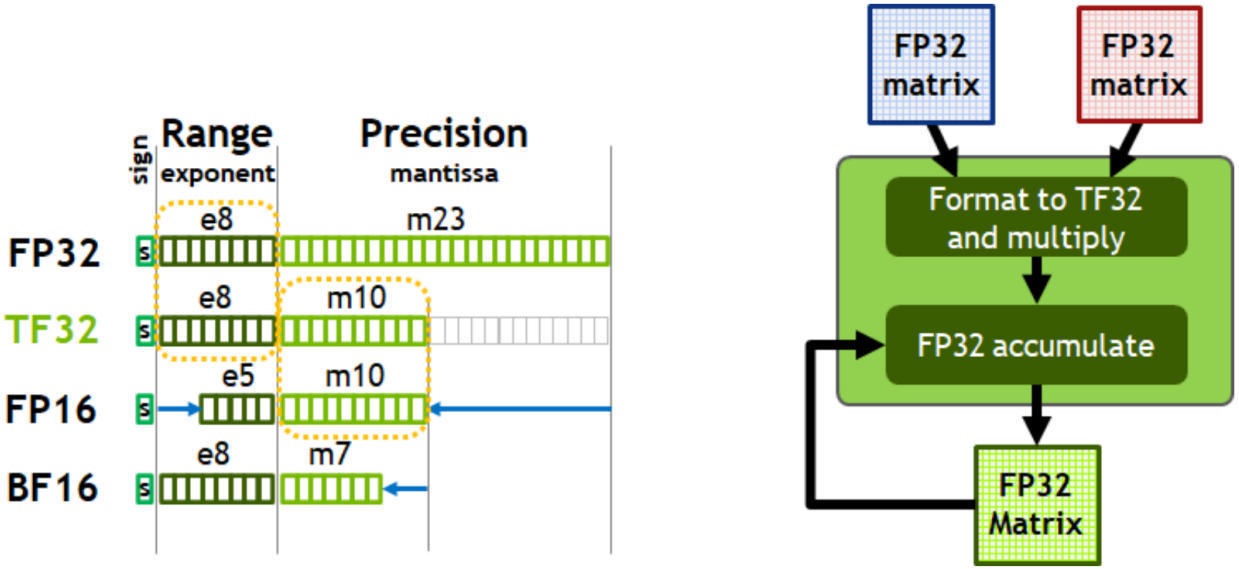


TF32

- TF32:
  - 1 sign bit, 8 exponent bits, 10 mantissa bits
- Features:
  - **Same range as FP32 (32-bit floating-point):** This ensures compatibility.
  - **Reduced precision compared to FP32:** The fewer mantissa bits in TF32 lead to slightly lower precision, but mostly sufficient for ML.
  - **Faster processing than FP32:** By using fewer bits, TF32 enables faster calculations.



NVIDIA A100



TensorFloat-32 (TF32) provides the range of FP32 with the precision of FP16, 8x precision vs. BF16 (left). A100 accelerates tensor math with TF32 while supporting FP32 input and output data (right), enabling easy integration into DL and HPC programs and automatic acceleration of DL frameworks.

Figure 9. TensorFloat-32 (TF32)

Figure 7. GA100 Streaming Multiprocessor (SM)

Source : NVIDIA A100 Tensor Core GPU Architecture