Graphics Processing Unit

Parallel Processing

--Credit: Onur Mutlu

Recall: 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

Recall: SIMD Processing

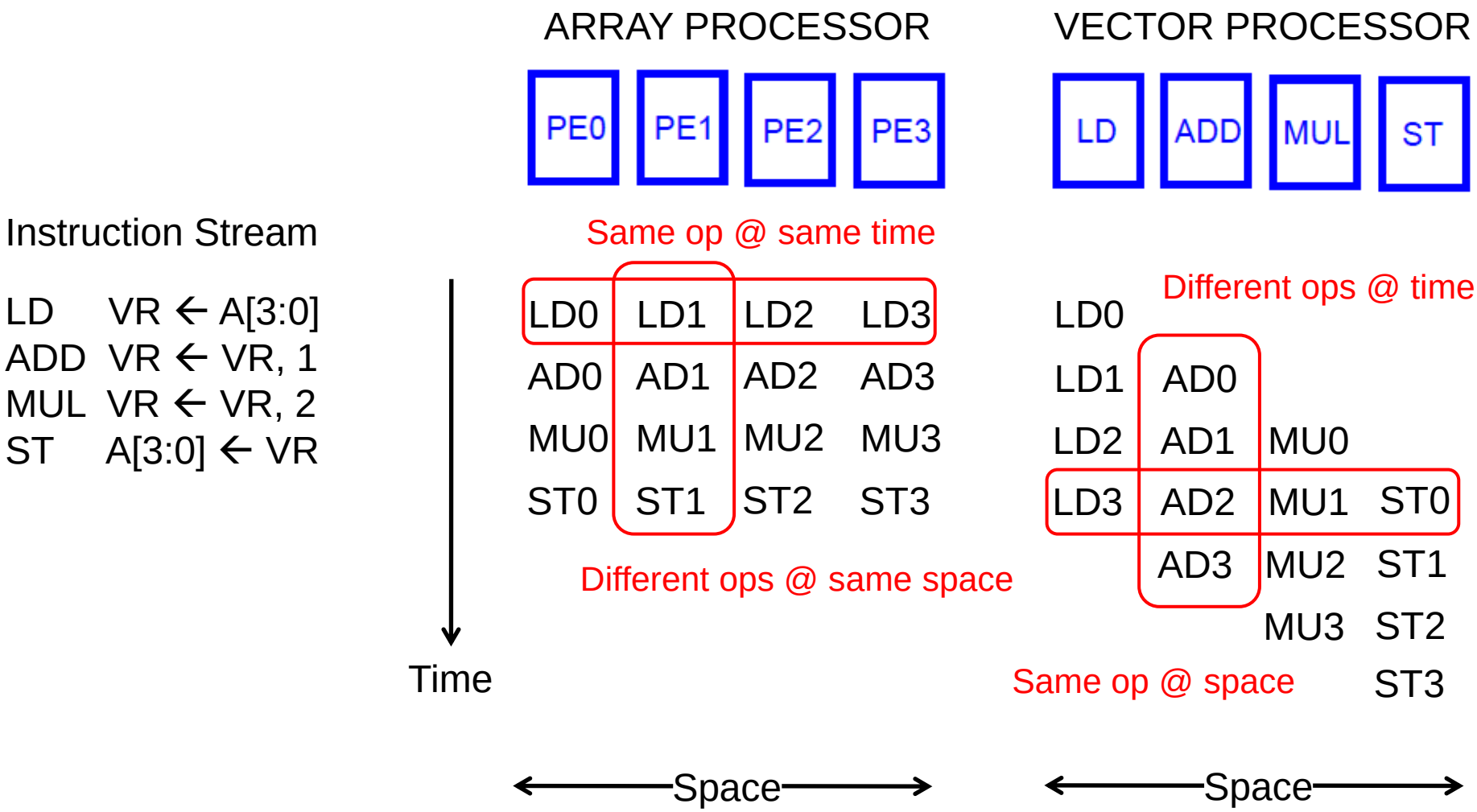
- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements (PEs), i.e., execution units

- Time-space duality

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

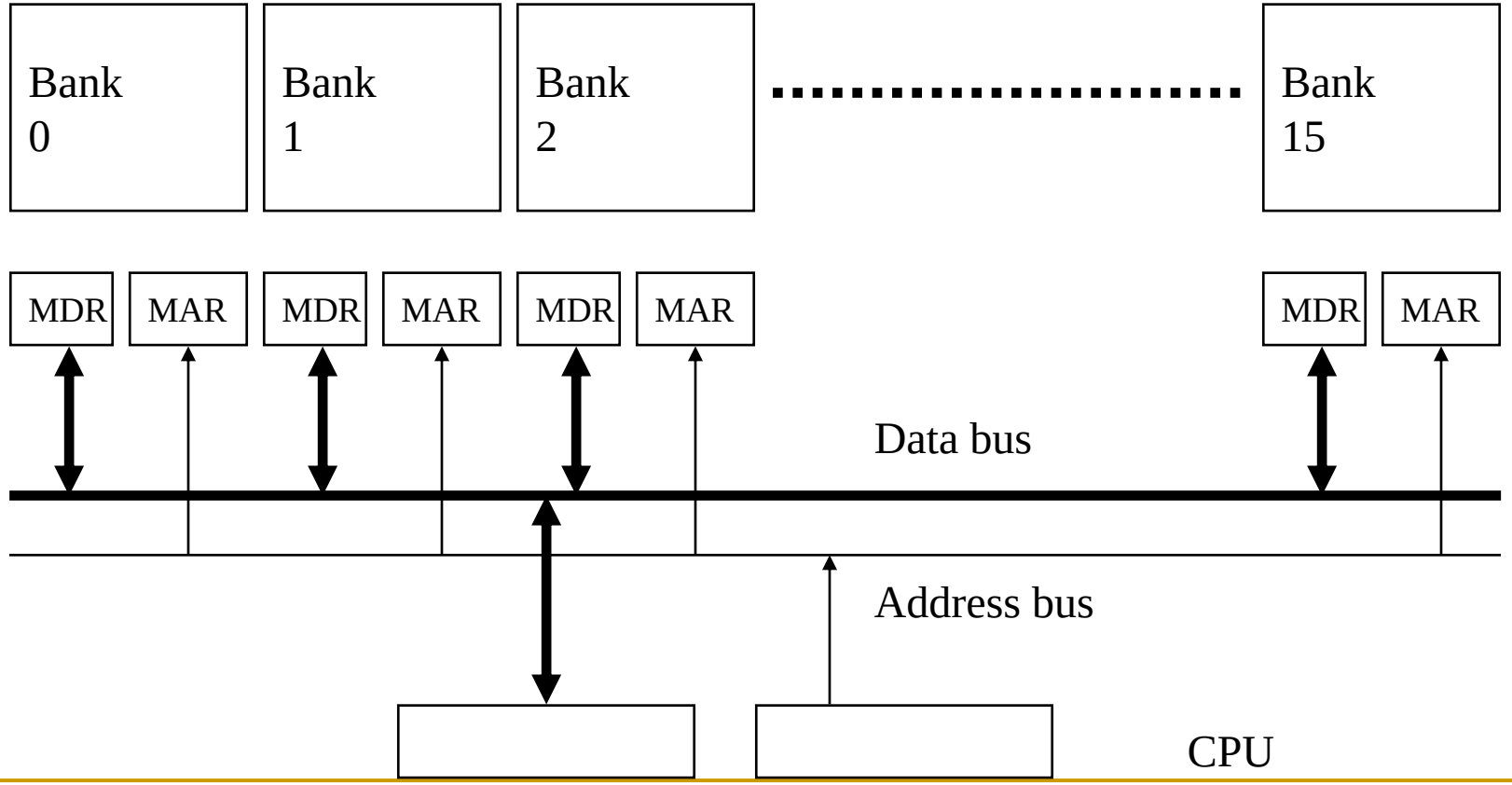
  - Vector processor: Instruction operates on multiple data elements in consecutive time steps using the same space (PE)

Recall: Array vs. Vector Processors



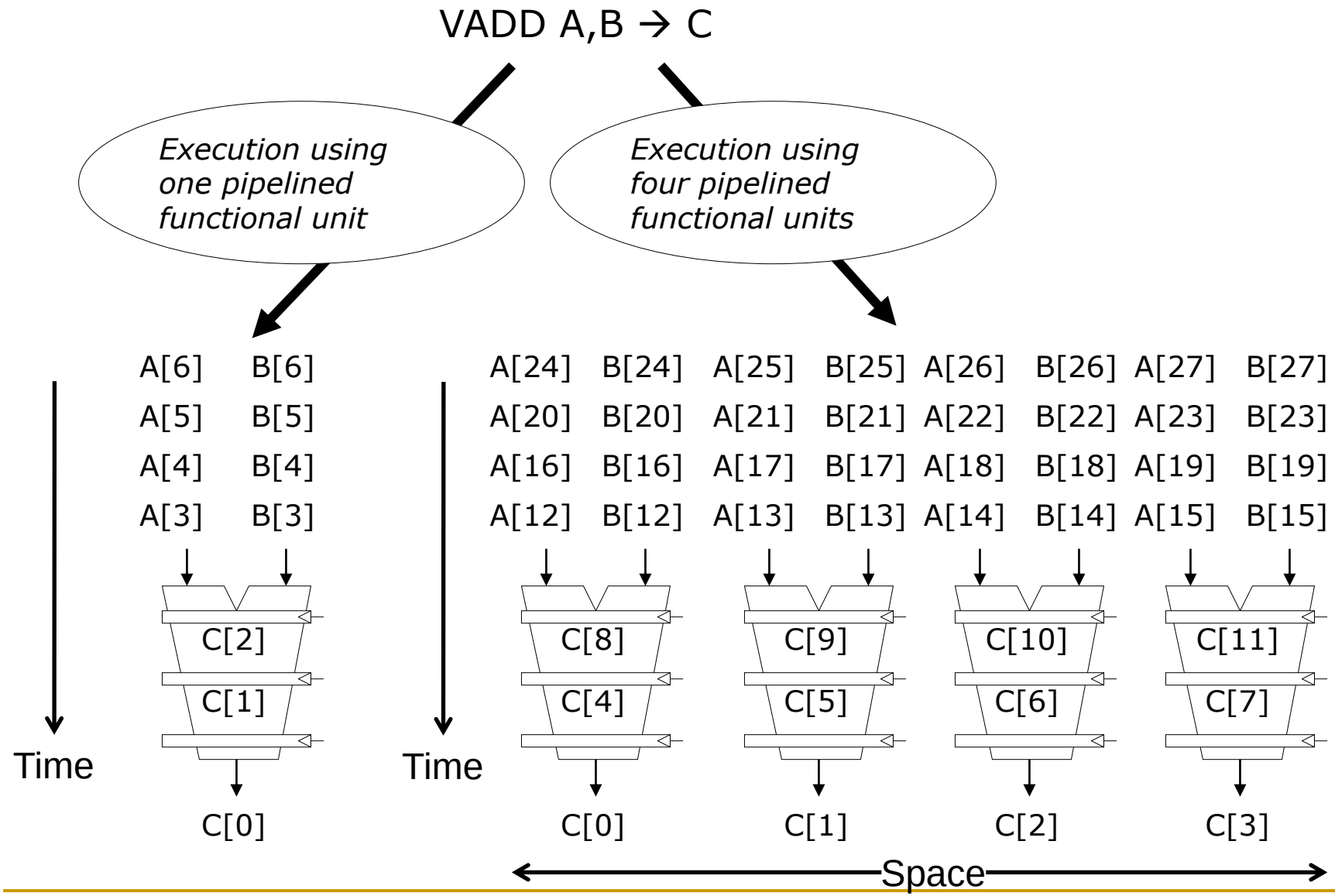
Recall: Memory Banking

- 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 concurrent accesses if all N go to different banks



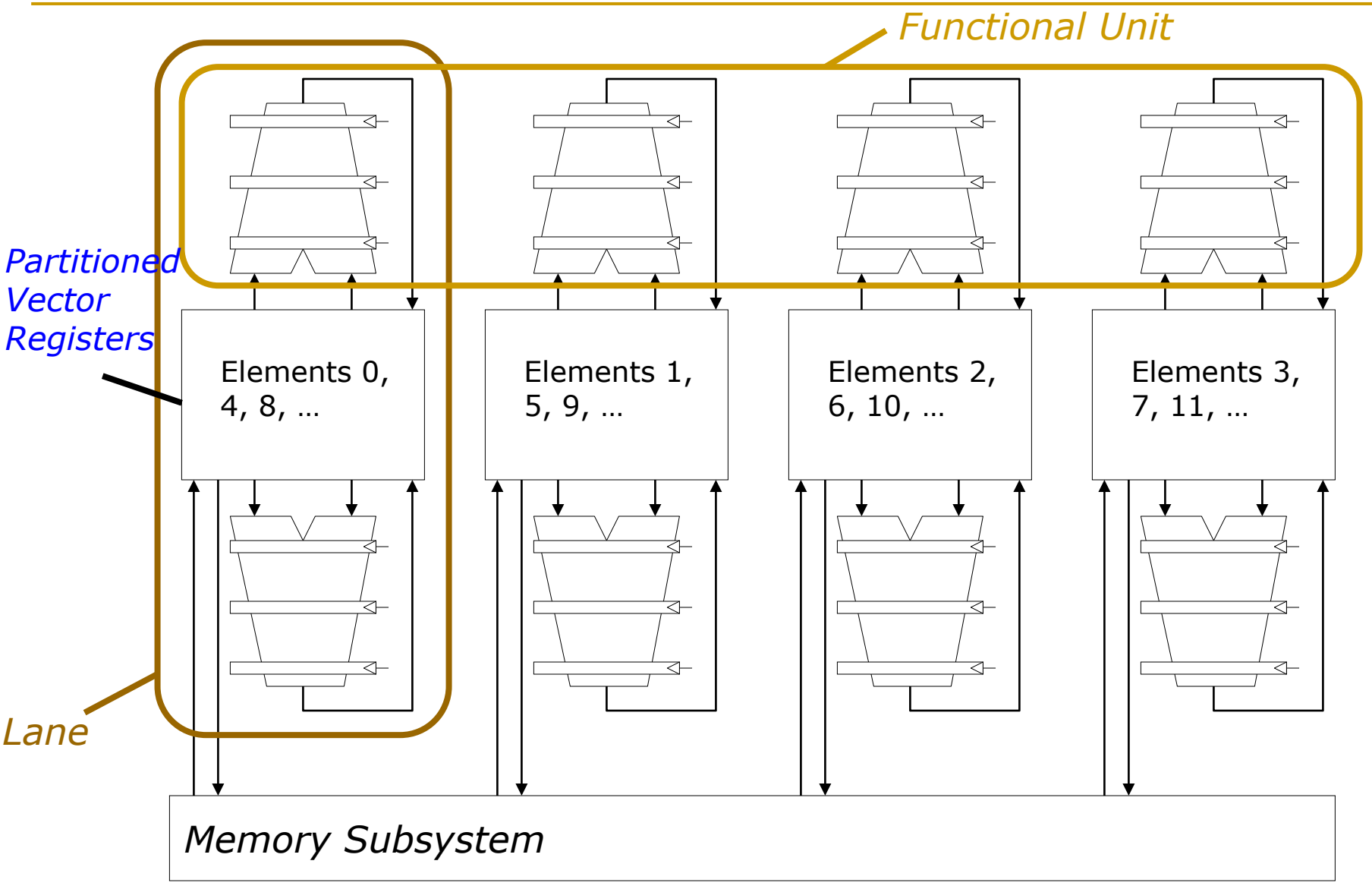
Picture credit: Derek Chiou

Recall: Vector Instruction Execution



Slide credit: Krste Asanovic

Recall: Vector Unit Structure

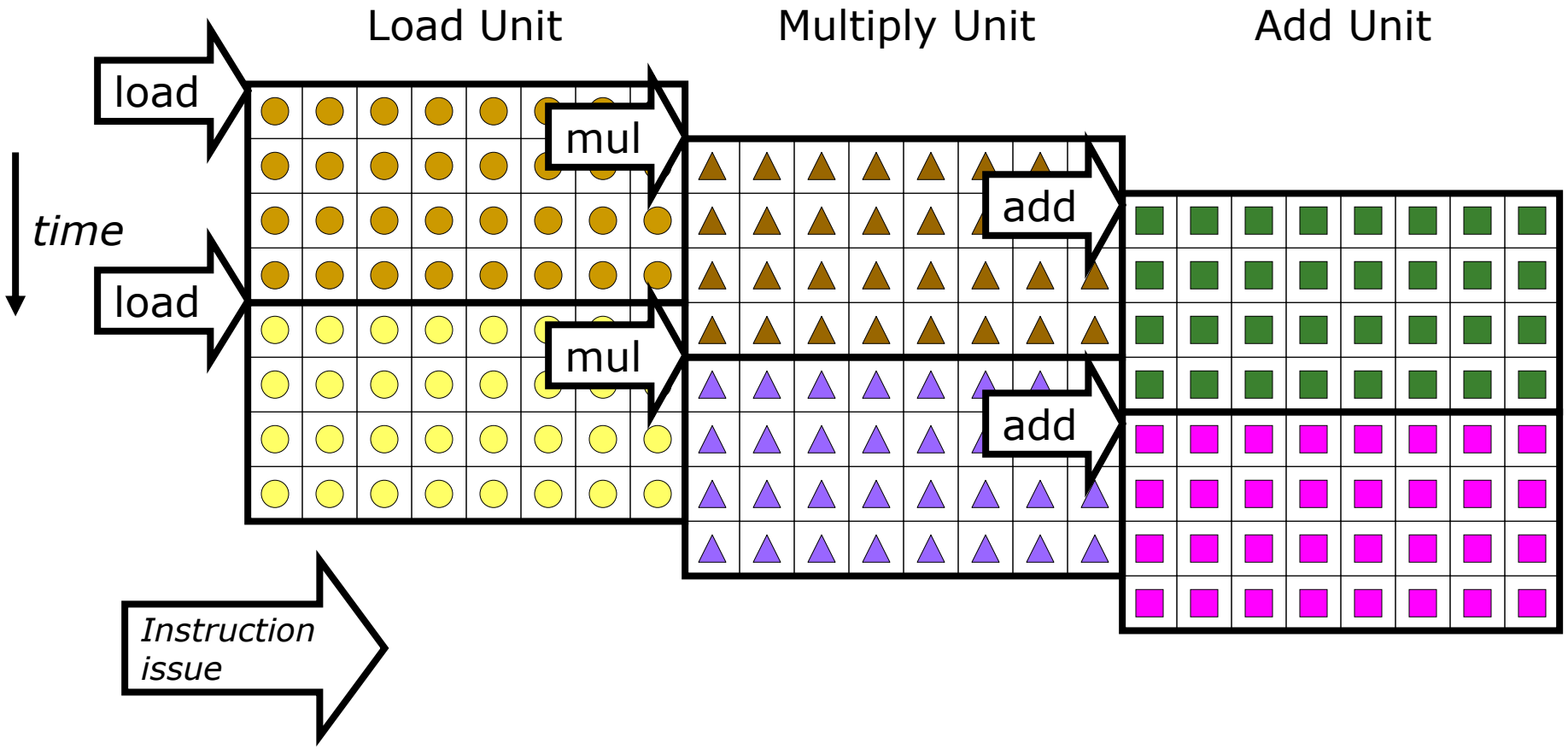


Slide credit: Krste Asanovic

Recall: 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



Recall: Vector Processor Disadvantages

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

++ Vector operations

-- Very inefficient if parallelism is irregular

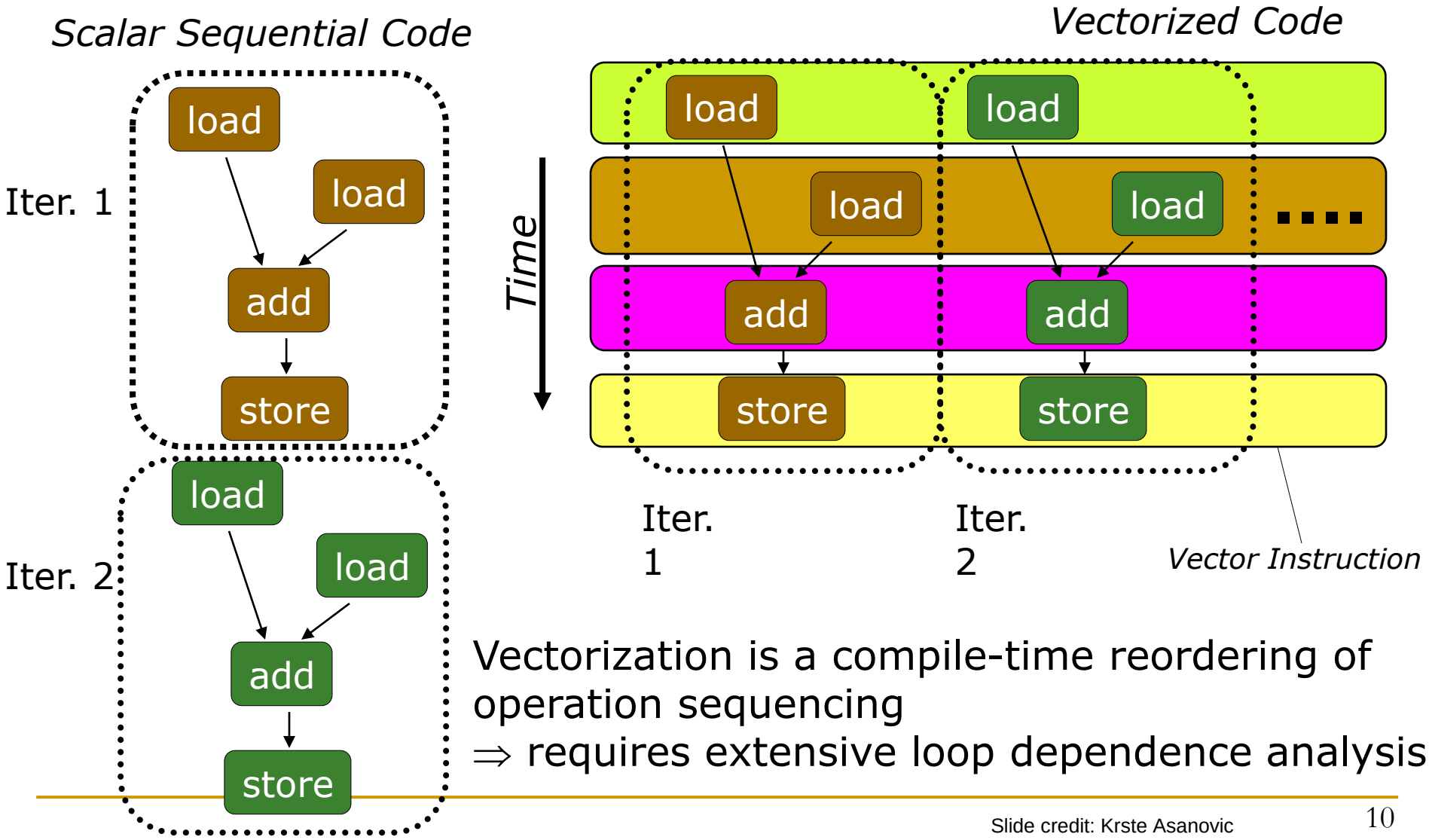
-- How about searching for a key in a linked list?

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.

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

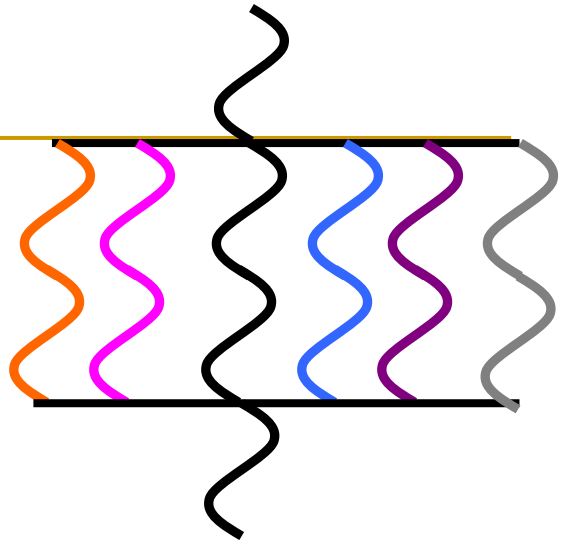
Vector/SIMD Processing Summary

- 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

Recall: Amdahl's Law



- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

- Maximum speedup limited by serial portion: Serial bottleneck

- All parallel machines "suffer from" the serial bottleneck

Extra Assignment 3: Amdahl's Law (I)

- **Paper review**
  - G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page review**
  - Upload PDF file to Moodle – Deadline: June 15

- I strongly recommend that you follow my guidelines for (paper) review (see next slide)

Extra Assignment 3: Amdahl's Law (II)

- **Guidelines on how to review papers critically**
  - Guideline slides: pdf ppt
  - Video: https://www.youtube.com/watch?v=tOL6FANAJ8c
  - Example reviews on "Main Memory Scaling: Challenges and Solution Directions" (link to the paper)
    - Review 1
    - Review 2
  - Example review on "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems" (link to the paper)
    - Review 1

Extra Assignment 3: Amdahl's Law (III)

Validity of the single processor approach to achieving large scale computing capabilities

by DR. GENE M. AMDAHL
International Business Machines Corporation
Sunnyvale, California

INTRODUCTION

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. Variously the proper direction has been pointed out as general purpose computers with a generalized interconnection of memories, or as specialized computers with geometrically related memory interconnections and controlled by one or more instruction streams.

Demonstration is made of the continued validity of the single processor approach and of the weaknesses of the multiple processor approach in terms of application to real problems and their attendant irregularities.

The arguments presented are based on statistical characteristics of computation on computers over the last decade and upon the operational requirements within problems of physical interest. An additional cessing rate, even if the housekeeping were done in a separate processor. The non-housekeeping part of the problem could exploit at most a processor of performance three to four times the performance of the housekeeping processor. A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.

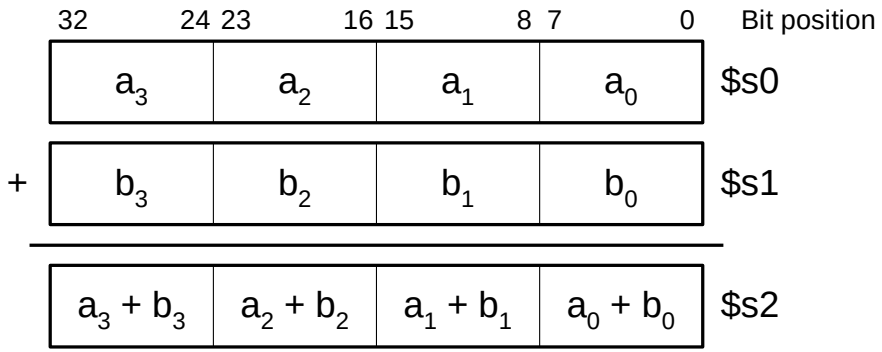
Data management housekeeping is not the only problem to plague oversimplified approaches to high speed computation. The physical problems which are of practical interest tend to have rather significant complications. Examples of these complications are as follows: boundaries are likely to be irregular; interiors are likely to be inhomogeneous; computations required may be dependent on the states of the variables at each point; propagation rates of different physical effects may be quite different; the

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

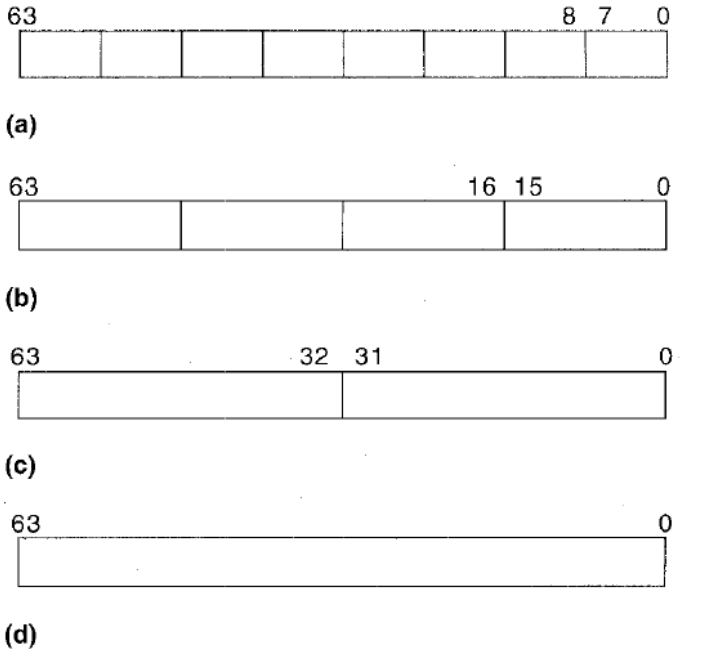


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

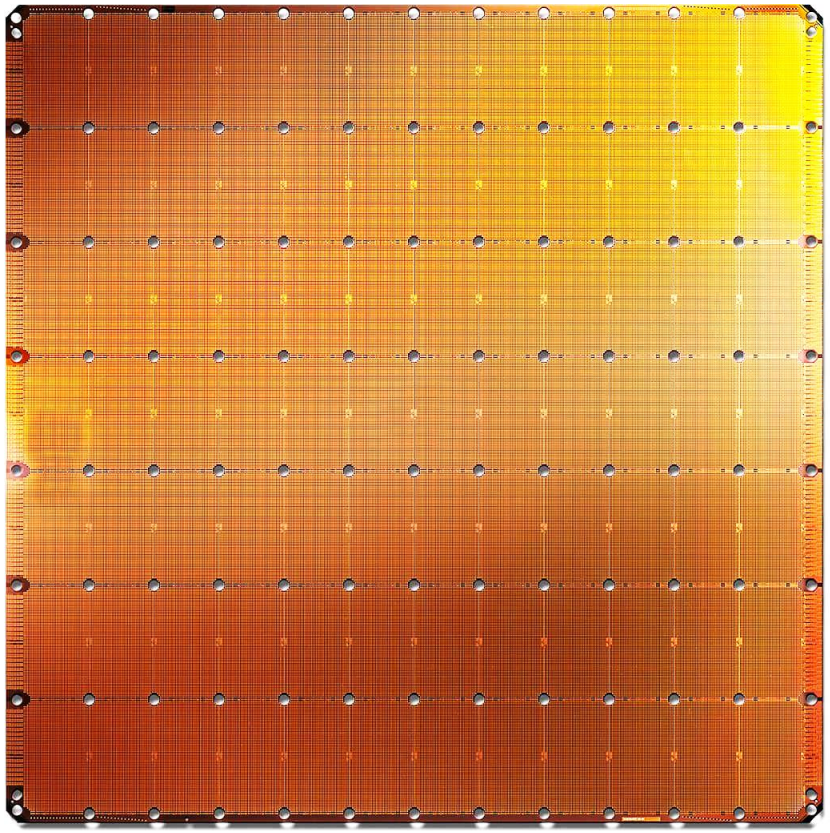
No VLEN register
Opcode determines data type:
8 8-bit bytes
4 16-bit words
2 32-bit doublewords
1 64-bit quadword

Stride is always equal to 1.

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

SIMD Operations in Modern (Machine Learning) Accelerators

Cerebras's Wafer Scale Engine (2019)



- The largest ML accelerator chip (2019)
- 400,000 cores

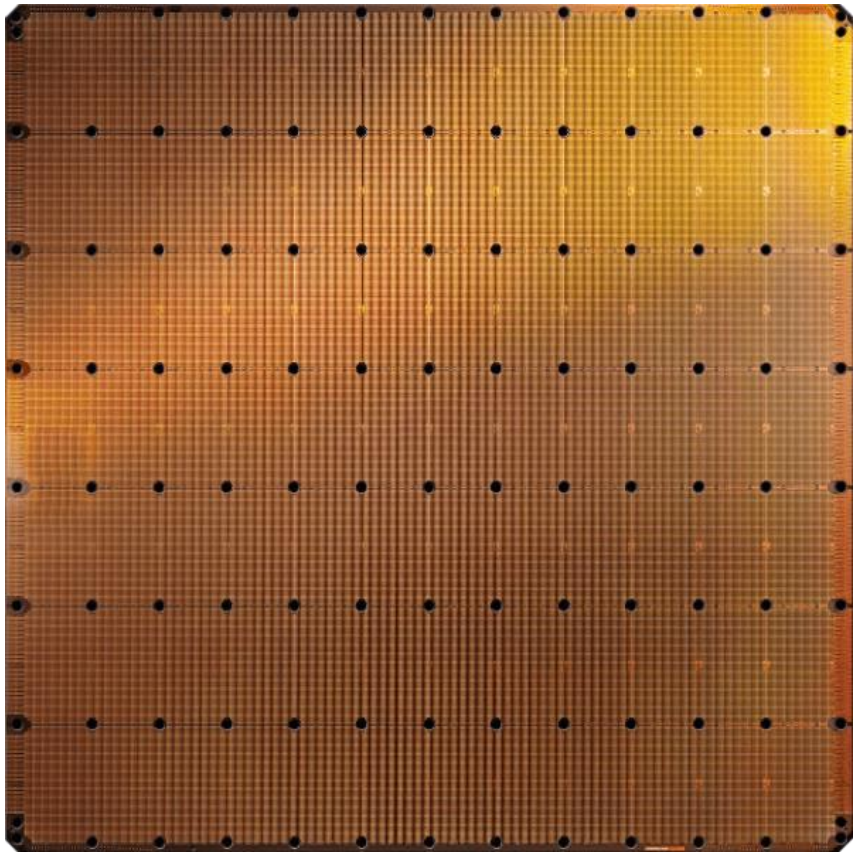
Cerebras WSE
1.2 Trillion transistors
46,225 mm^2

Largest GPU
21.1 Billion transistors
815 mm^2
NVIDIA TITAN V

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning

https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/

Cerebras's Wafer Scale Engine-2 (2021)



- The largest ML accelerator chip (2021)
- 850,000 cores

Cerebras WSE-2
2.6 Trillion transistors
46,225 mm^2

Largest GPU
54.2 Billion transistors
826 mm^2
NVIDIA Ampere GA100

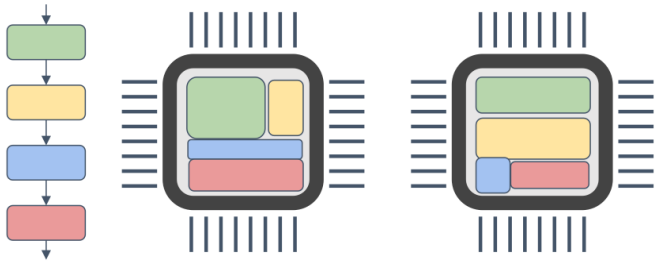
https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning

https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/

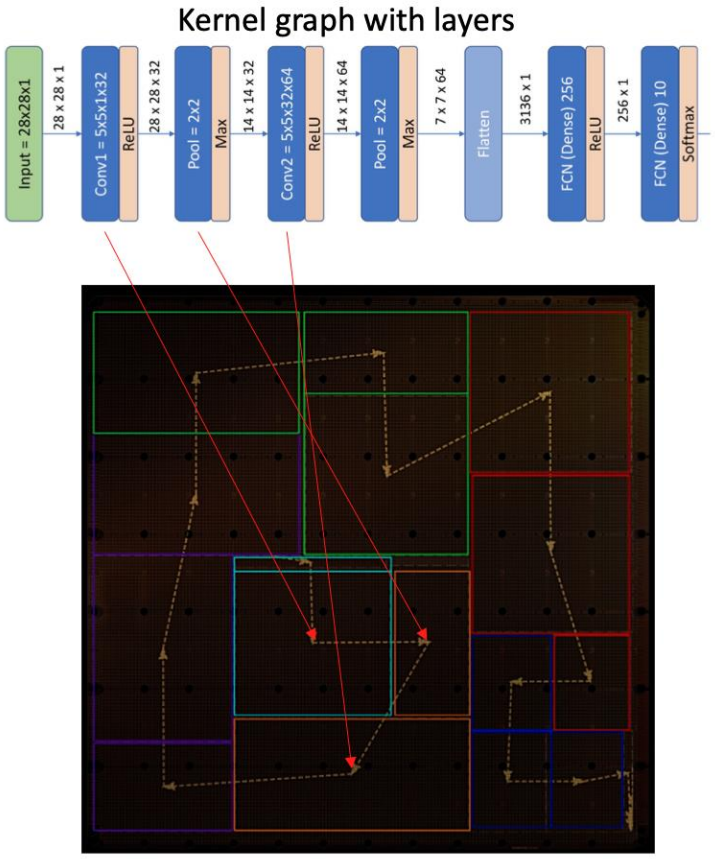
Size, Place, and Route in Cerebras's WSE

- Neural network mapping onto the whole wafer is a challenge

Multiple possible mappings



An example mapping



Layers mapped on Wafer Scale Engine

Different dies of the wafer work on different layers of the neural network: MIMD machine

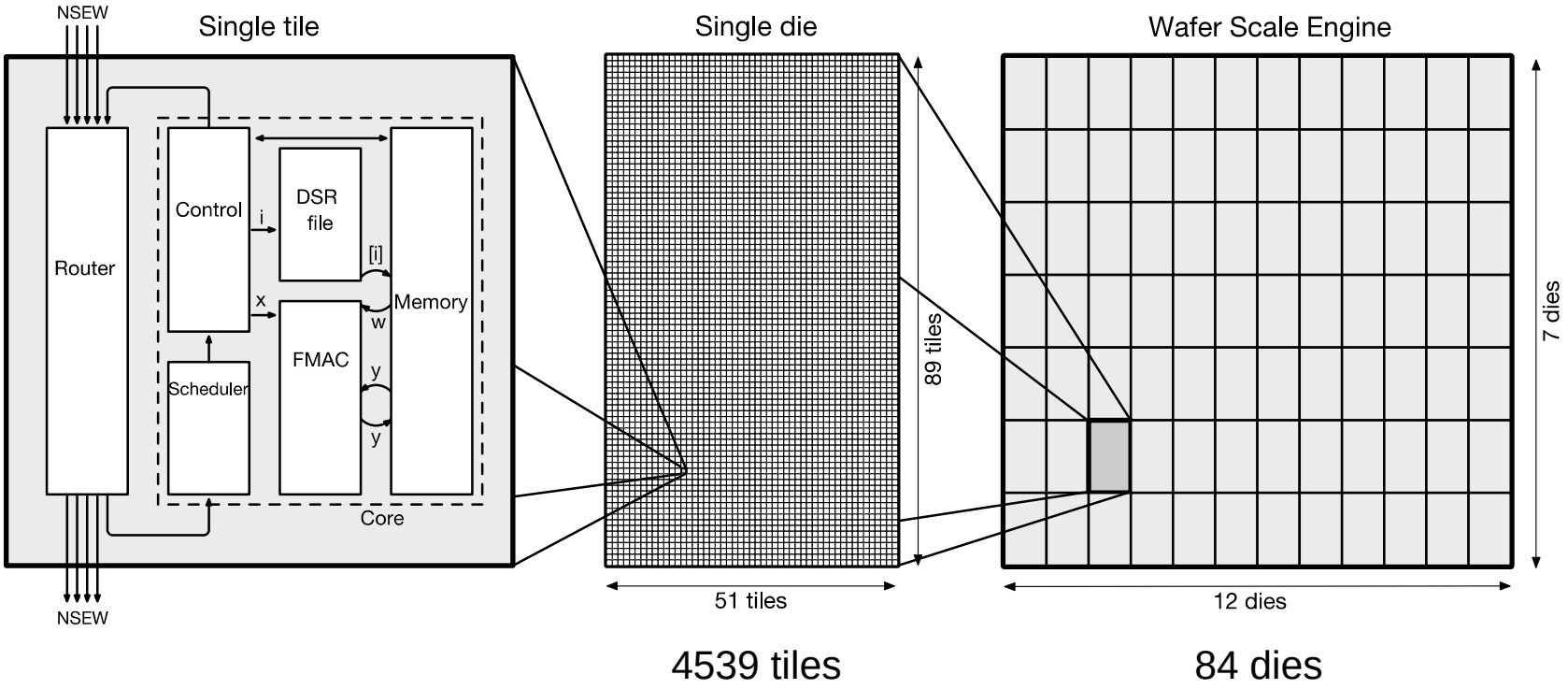
Recall: Flynn's Taxonomy of Computers

- 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

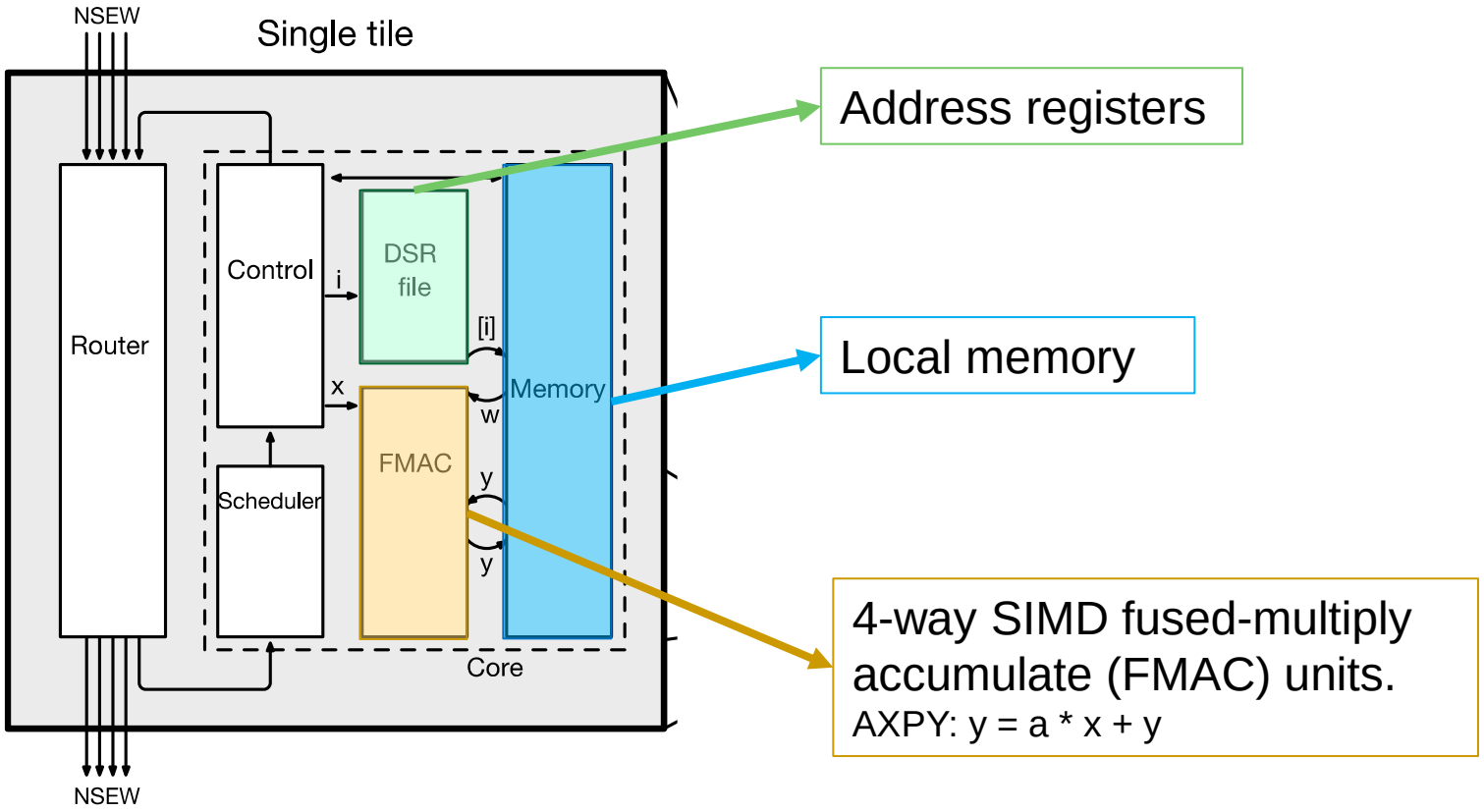
A MIMD Machine with SIMD Processors (I)

- MIMD machine
  - Distributed memory (no shared memory)
  - 2D-mesh interconnection fabric



A MIMD Machine with SIMD Processors (II)

- SIMD processors
  - 4-way SIMD for 16-bit floating point operands
  - 48 KB of local SRAM



Recall: Fine-Grained Multithreading

Recall: Fine-Grained Multithreading

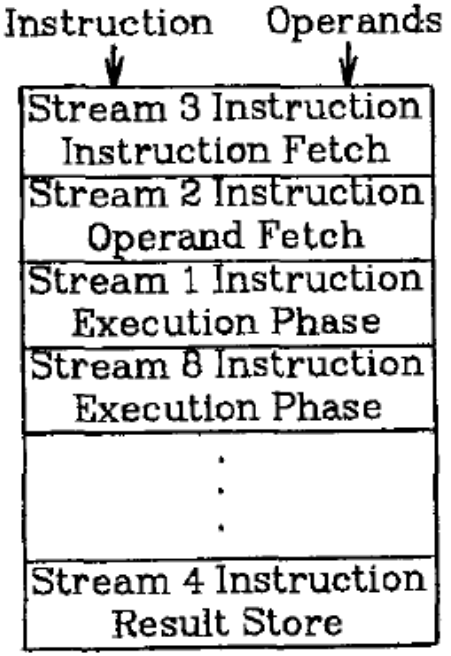
- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads' instructions

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

-- Single thread performance suffers

-- Extra logic for keeping thread contexts

-- Does not overlap latency if not enough threads to cover the whole pipeline



Recall: Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently

- Tolerates the control and data dependence latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

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)

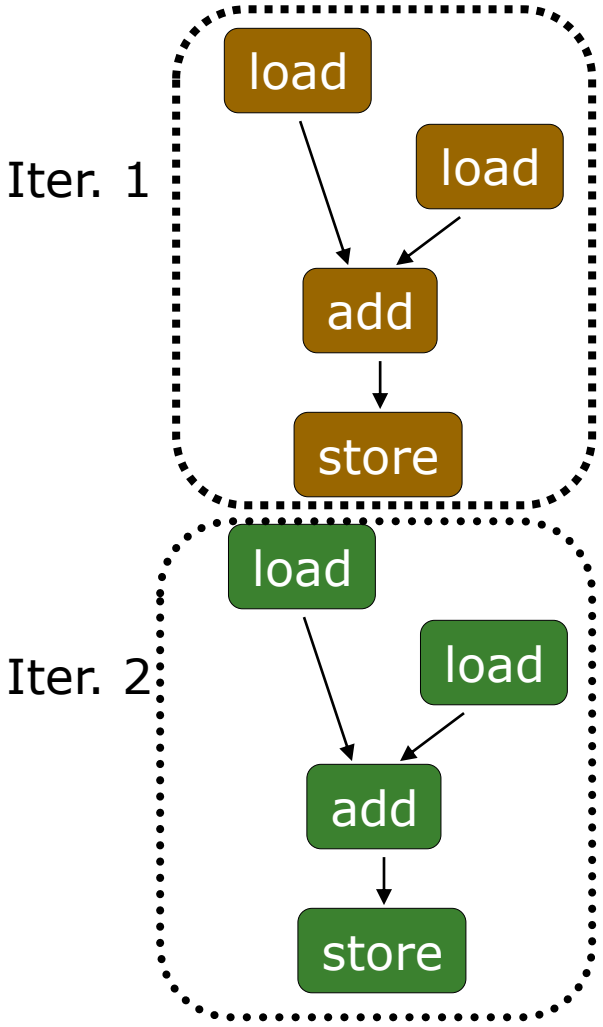
Programming Model vs. Hardware Execution Model

- 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:

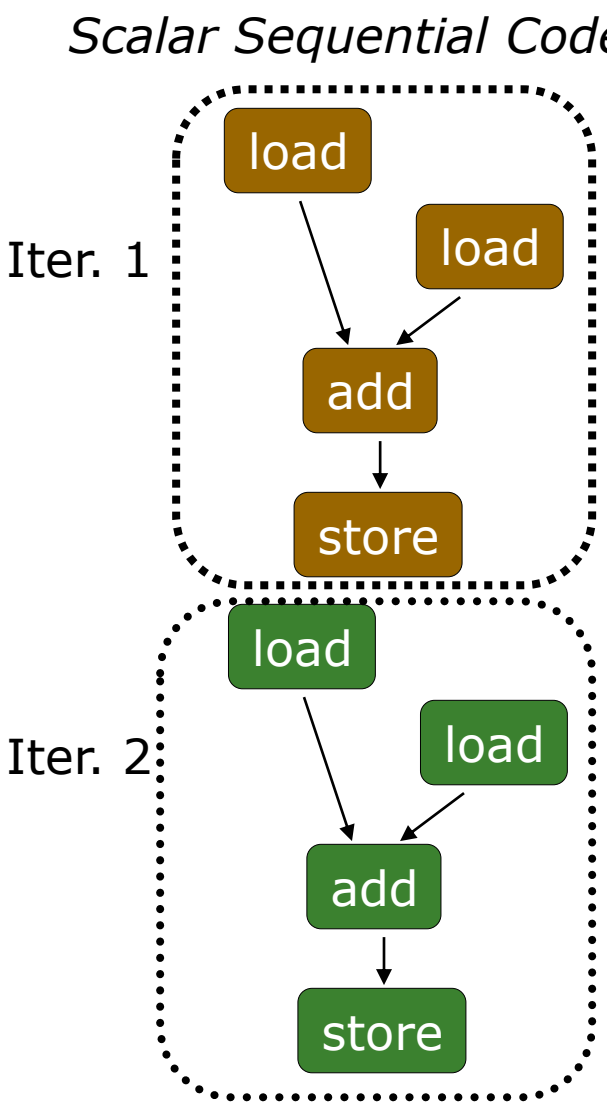
1. Sequential (SISD)

2. Data-Parallel (SIMD)

3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

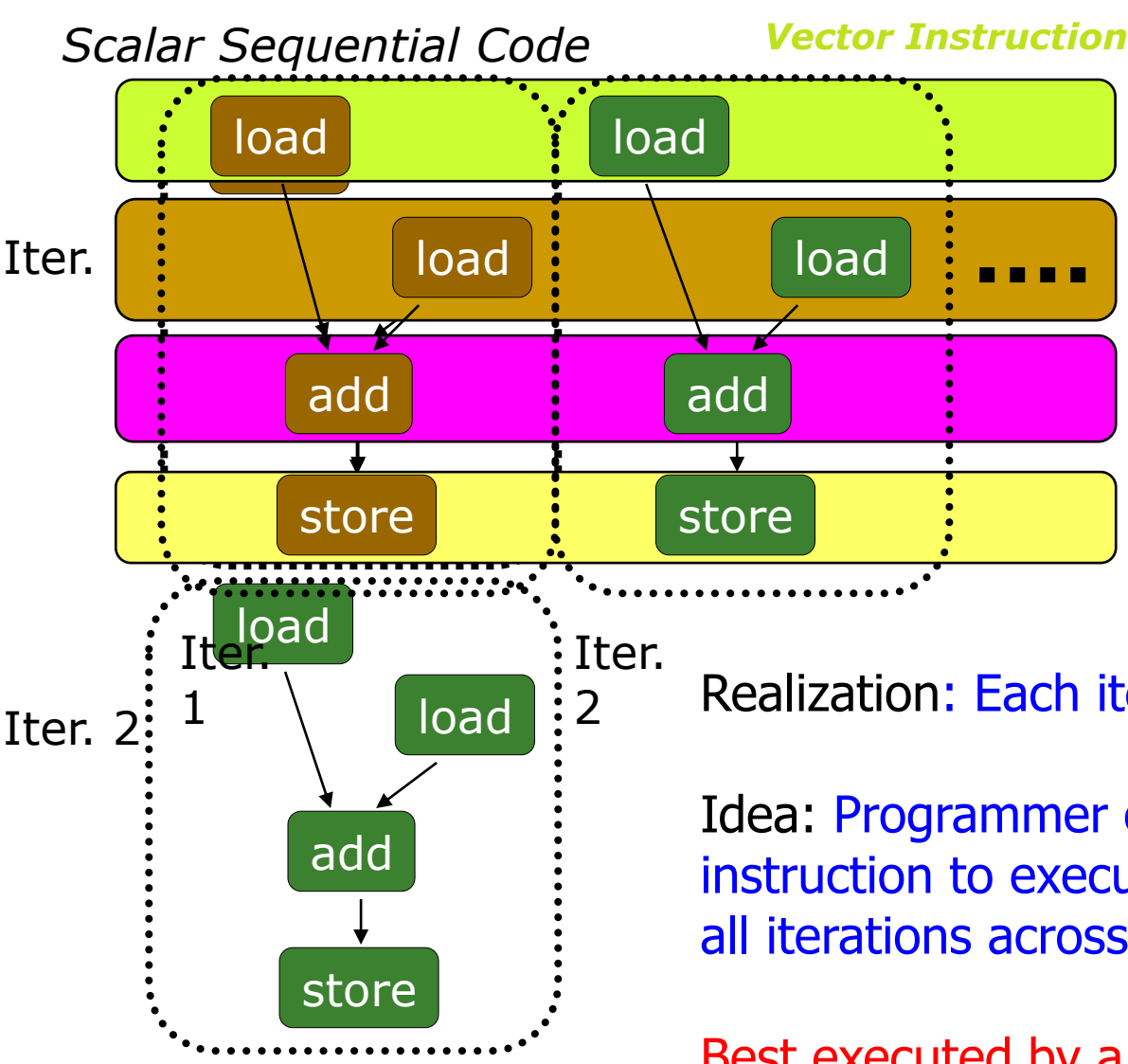
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



- 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

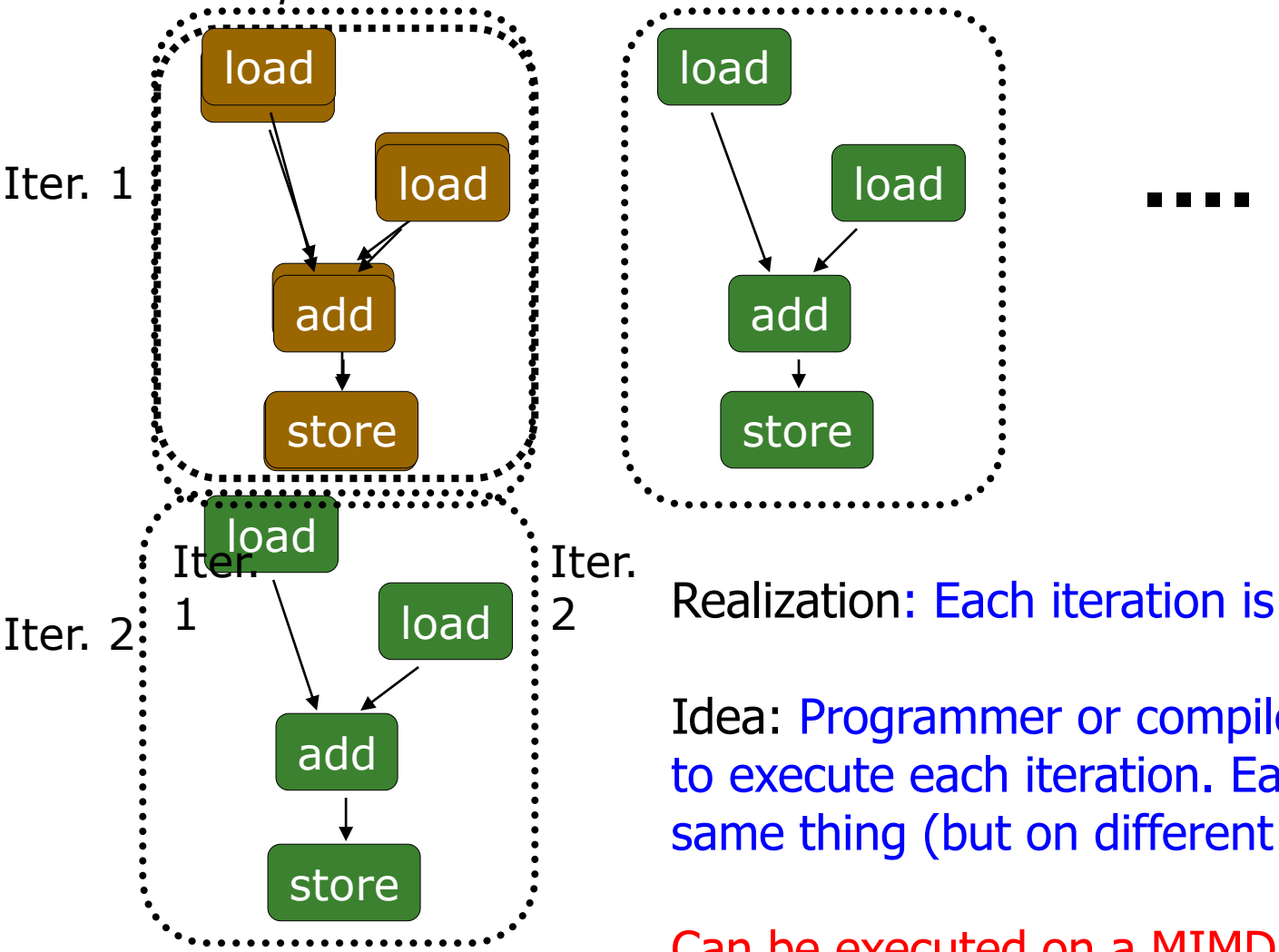
Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Best executed by a SIMD processor (vector, array)

Prog. Model 3: Multithreaded

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

Scalar Sequential Code



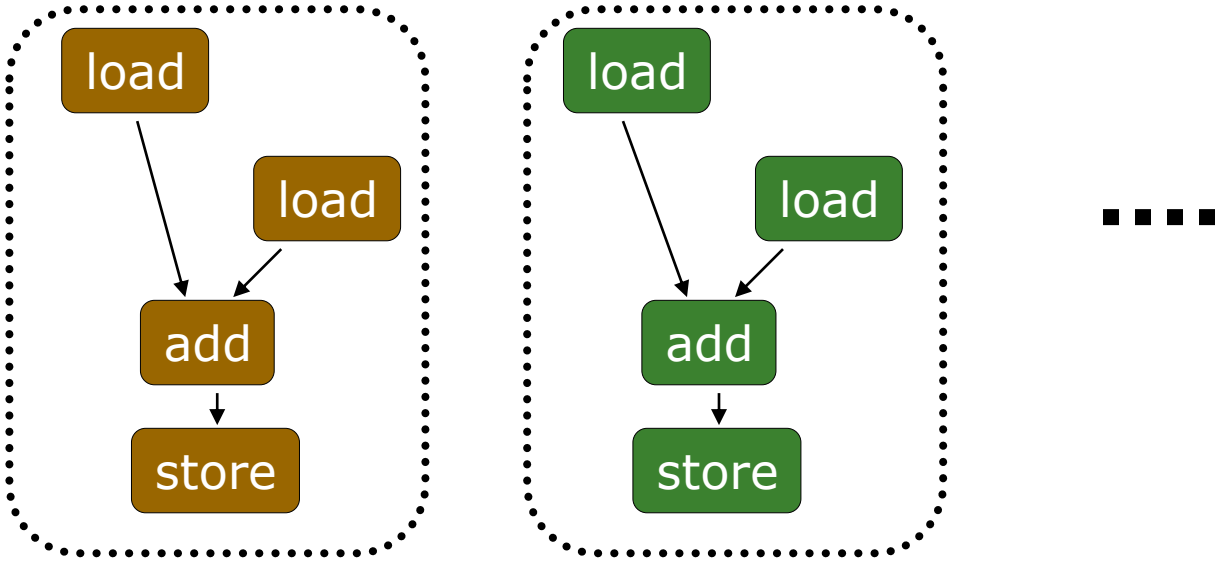
Realization: Each iteration is independent

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

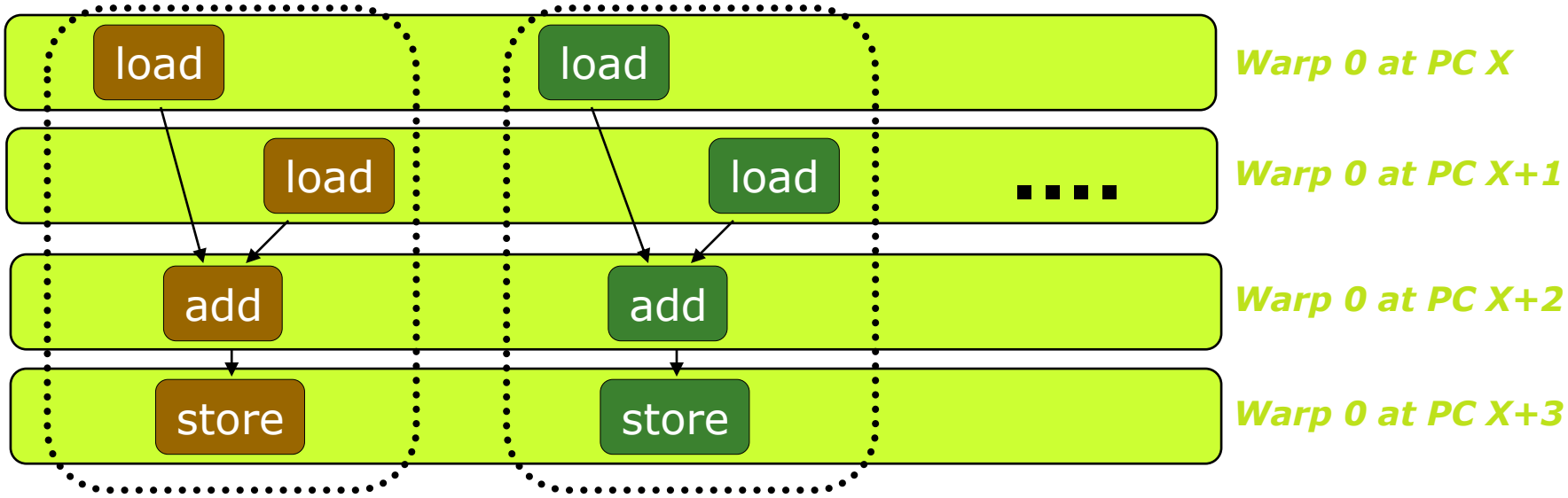
- Except it is **not** programmed using SIMD instructions

- 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

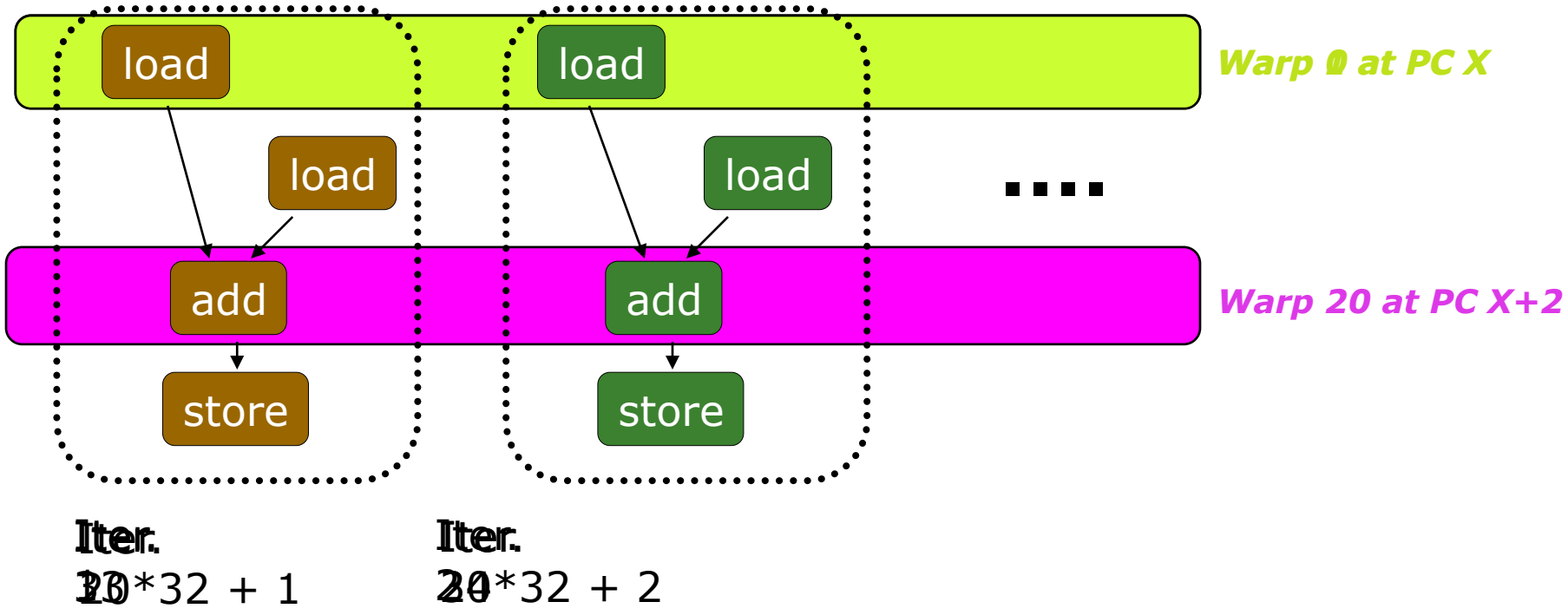
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- 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

Fine-Grained Multithreading of Warps

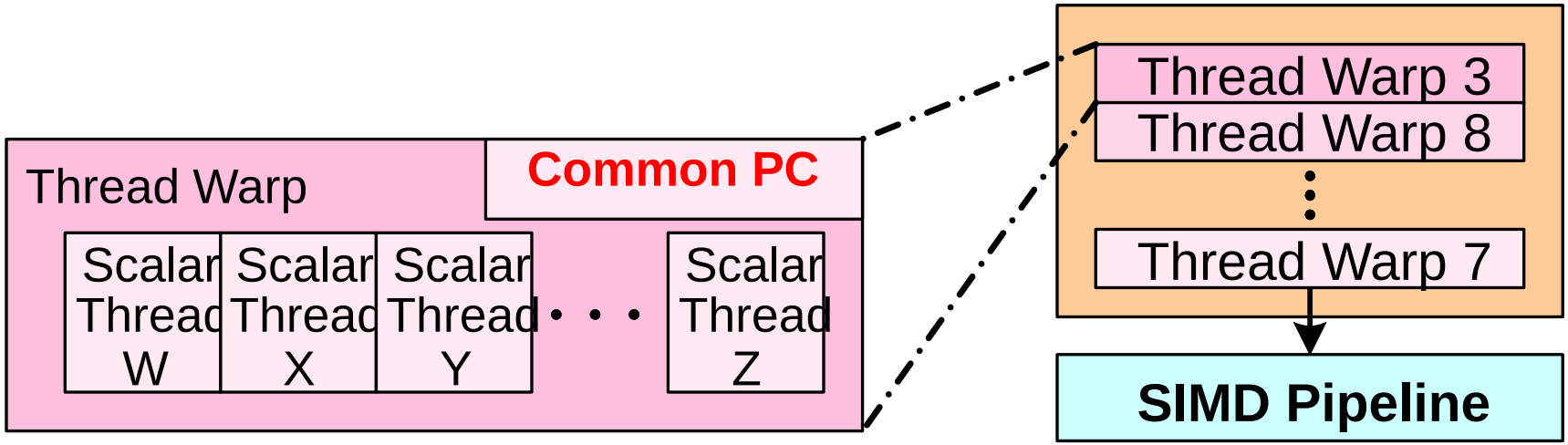
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

- 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



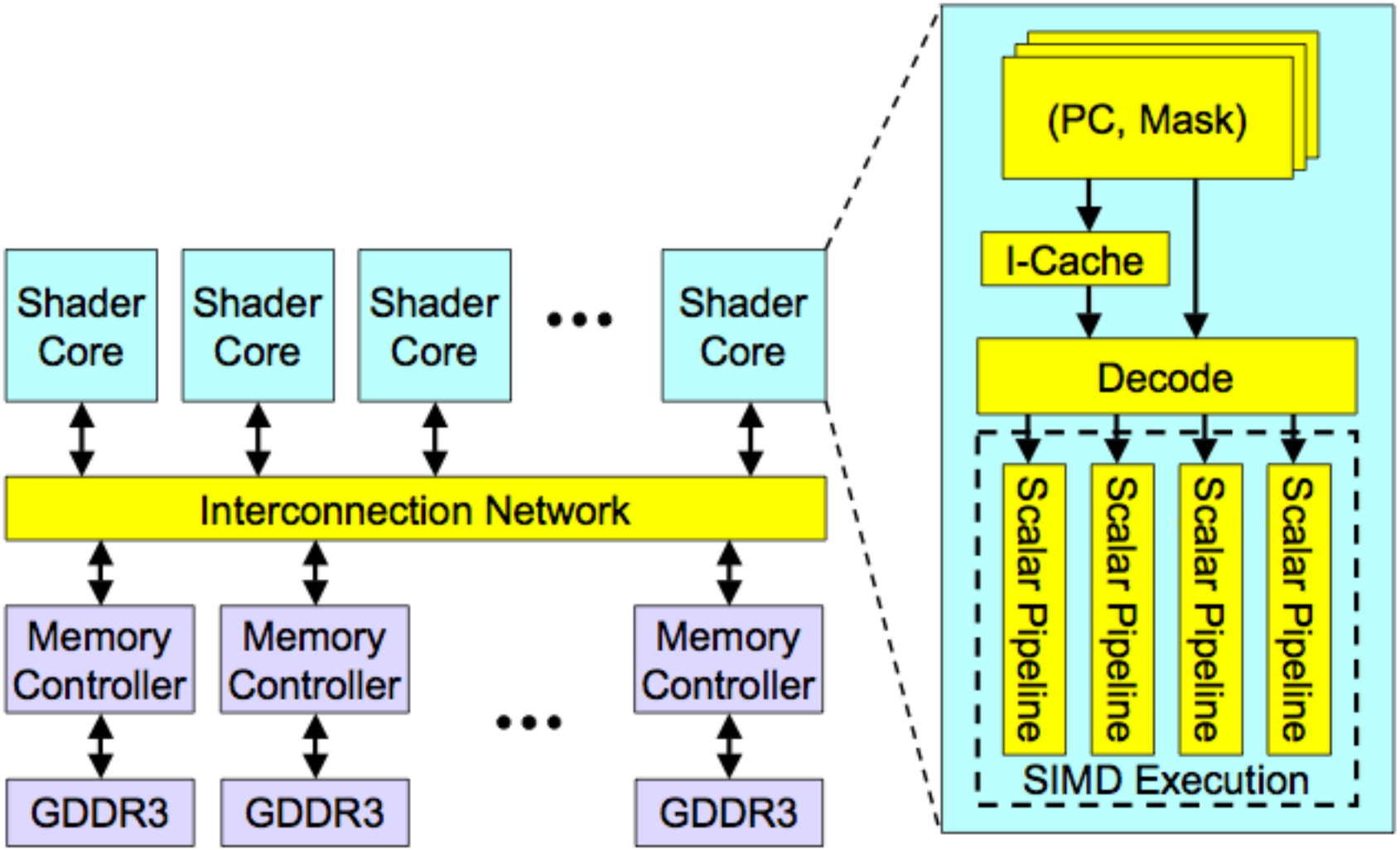
Warps and Warp-Level FGMT

- 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.

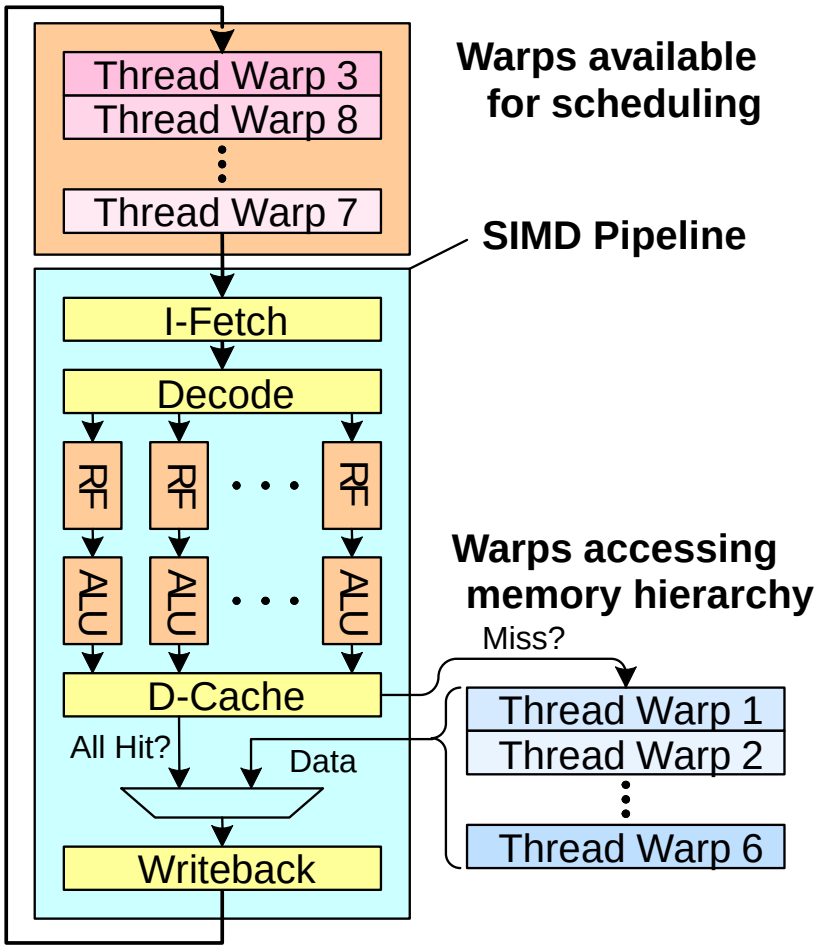
High-Level View of a GPU



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

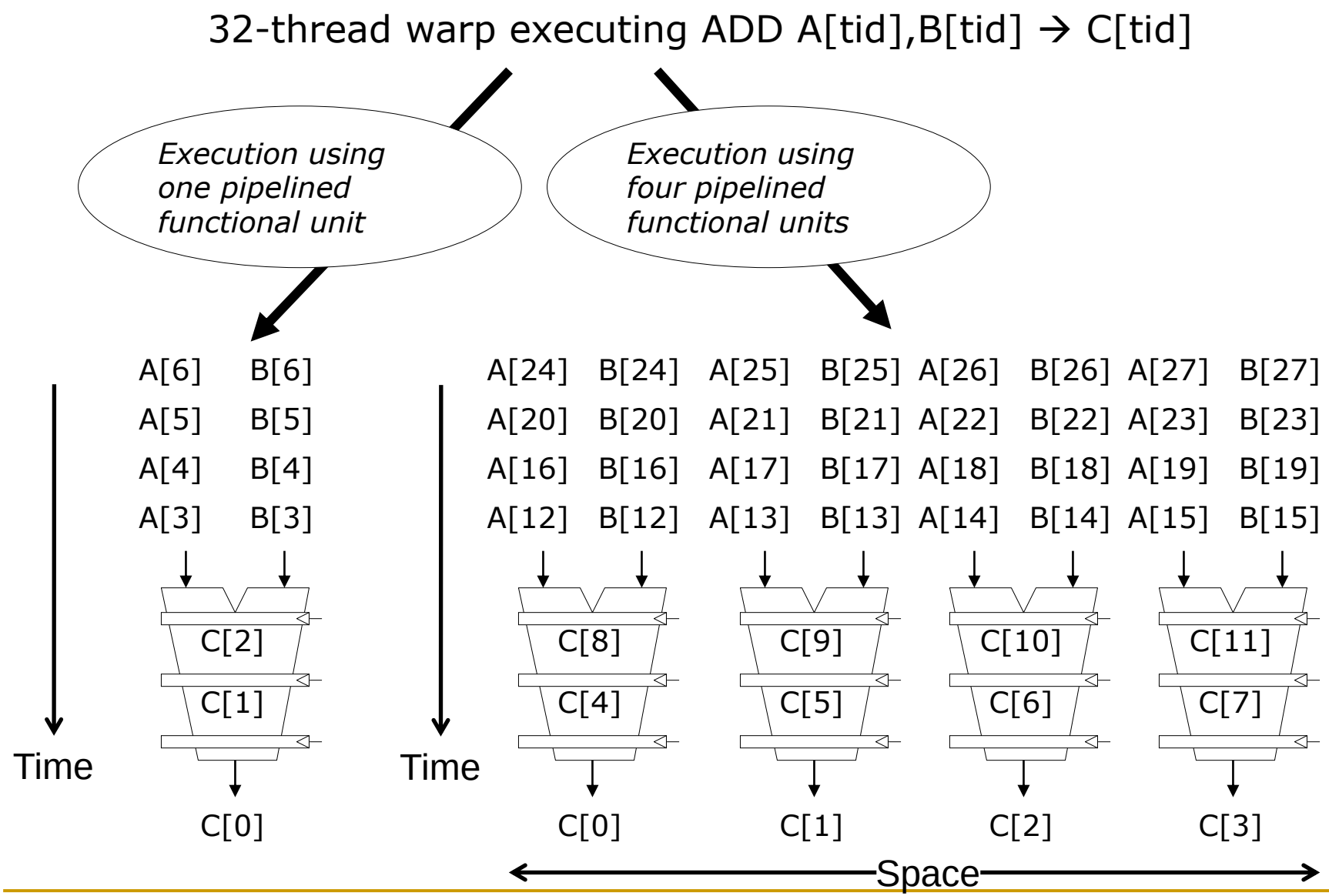
Latency Hiding via Warp-Level FGMT

- 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



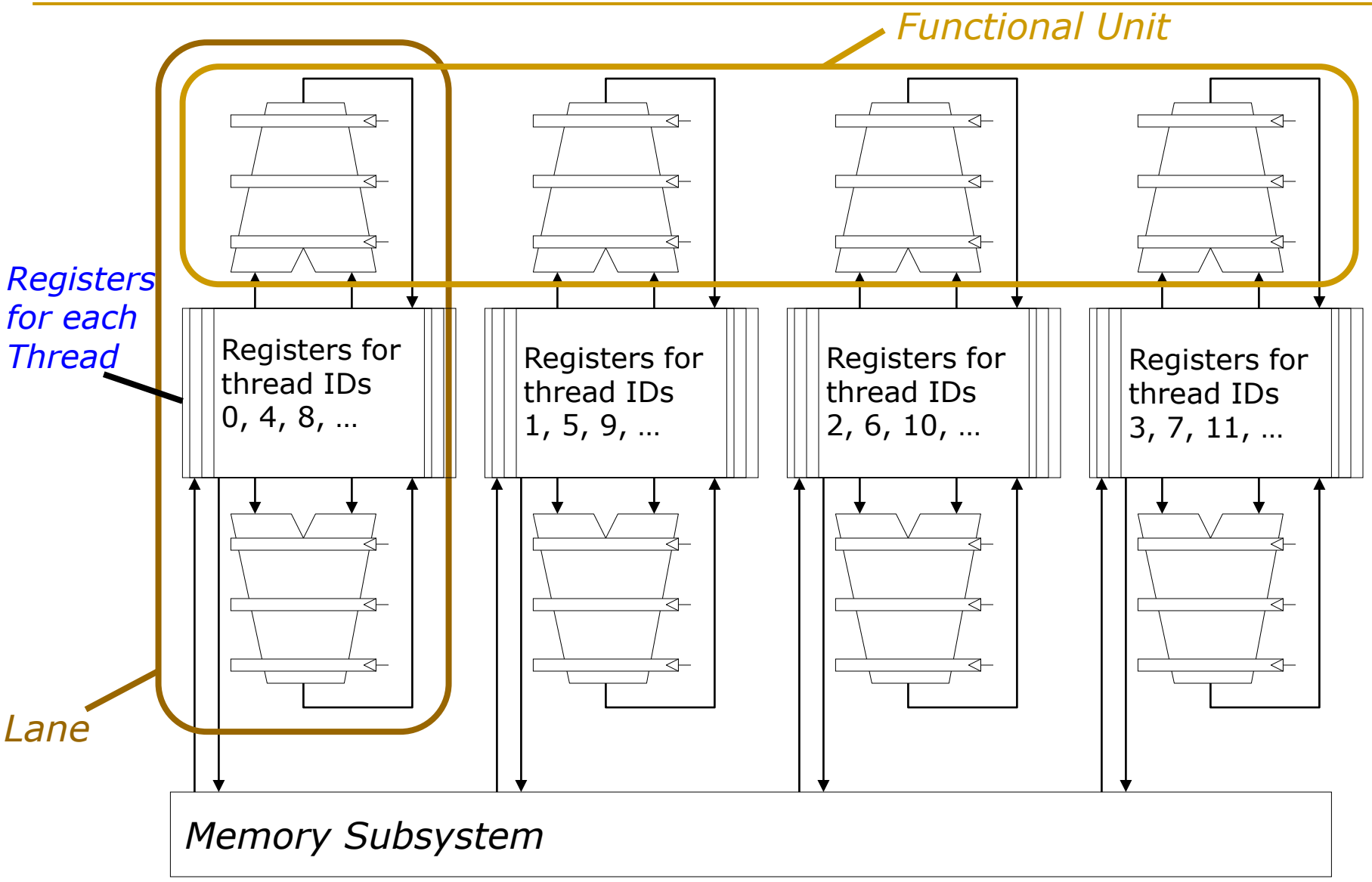
Slide credit: Tor Aamodt

Warp Execution (Recall the Slide)



Slide credit: Krste Asanovic

SIMD Execution Unit Structure

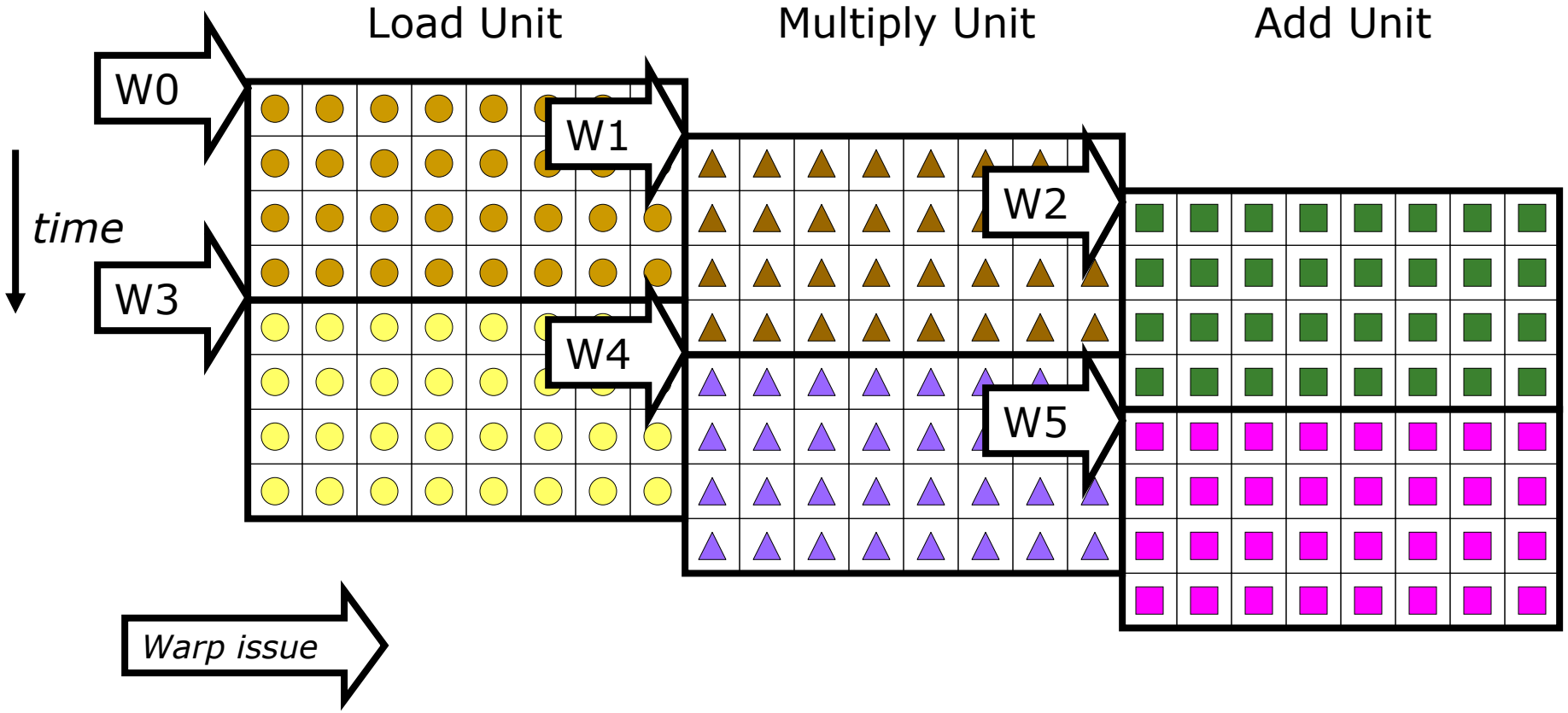


Slide credit: Krste Asanovic

Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

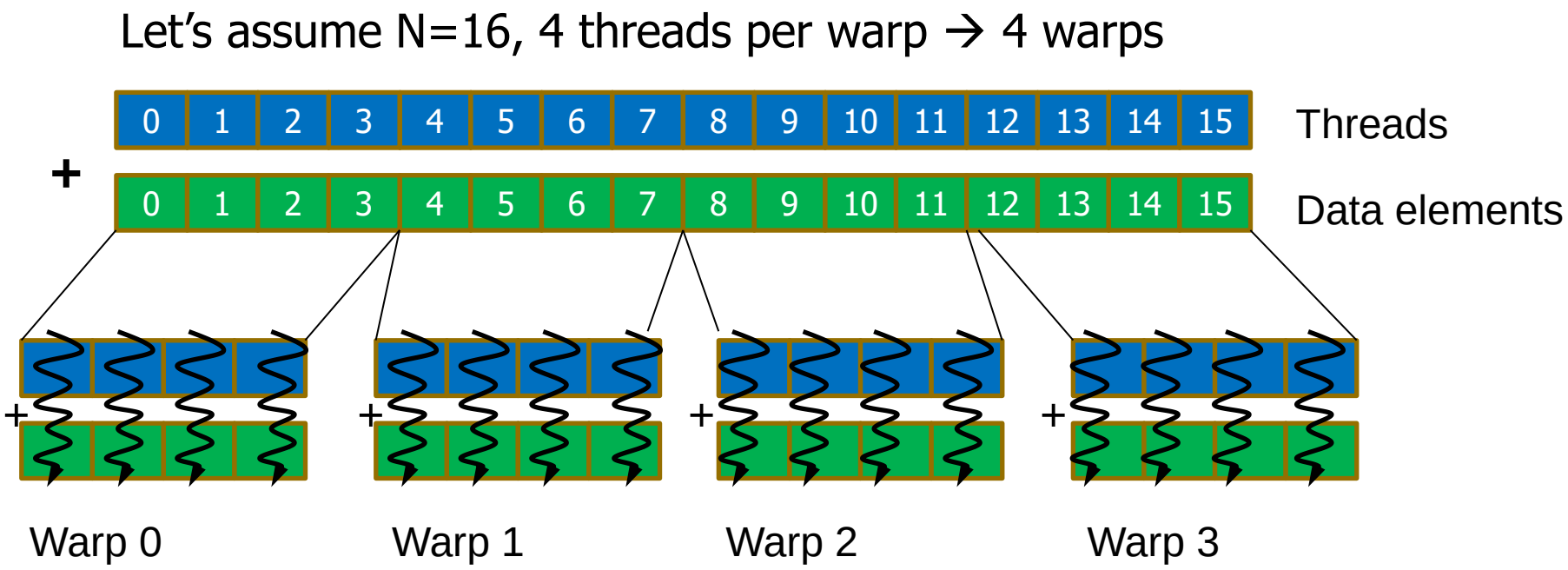
- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle



Slide credit: Krste Asanovic

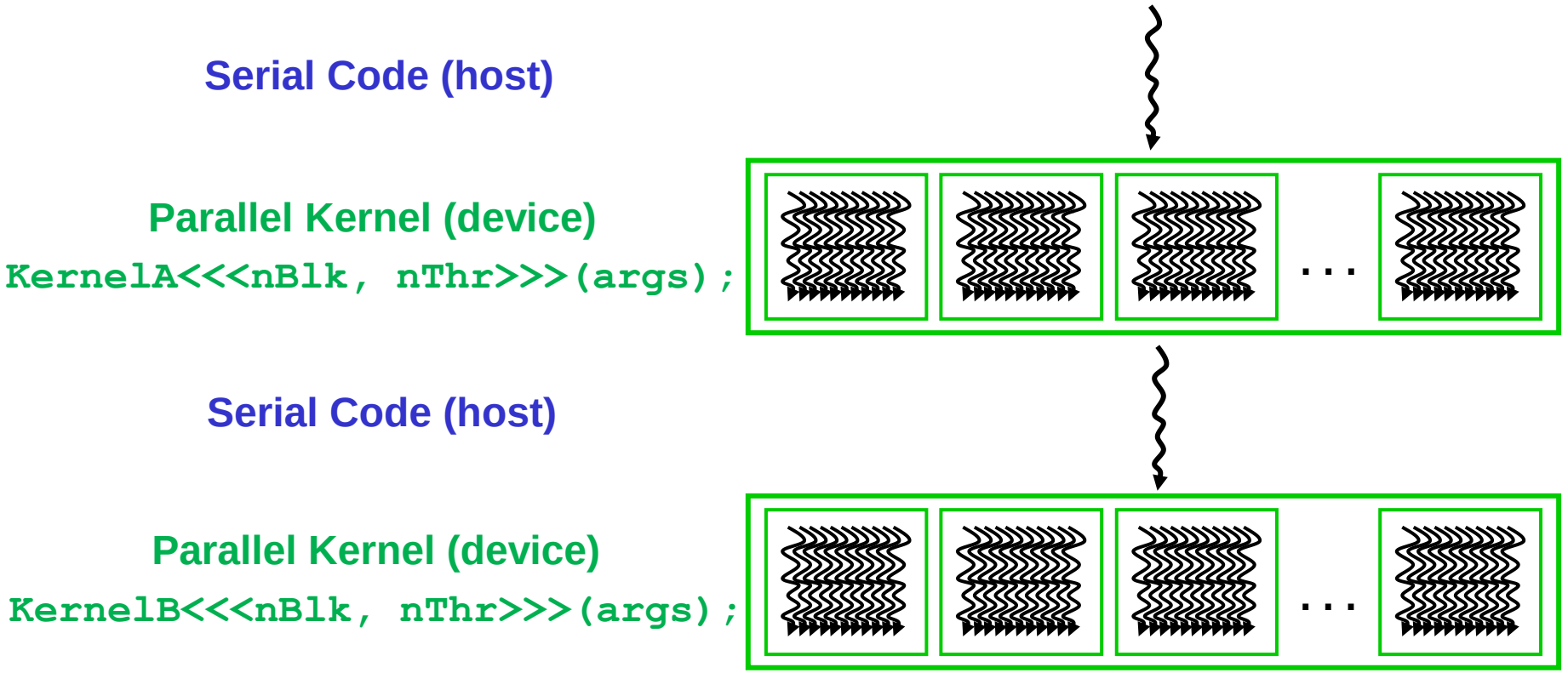
SIMT Memory Access

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



Warps *not* Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections 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)

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

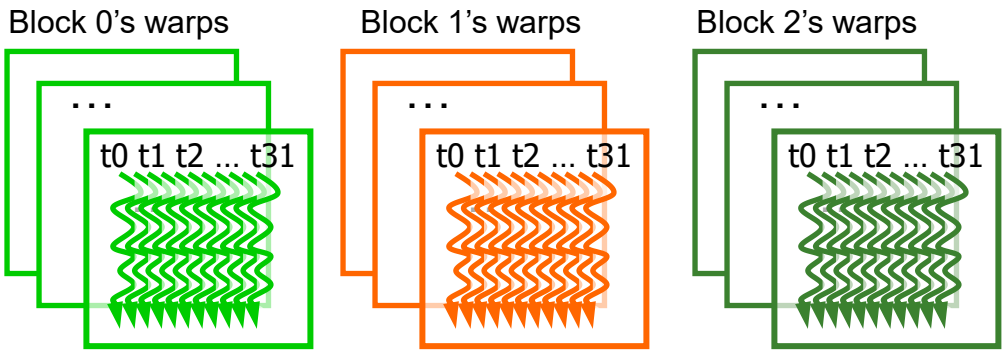
```
__global__  add_matrix
  ( float *a, float *b, float *c, int N) {
int i = blockIdx.x *  blockDim.x + threadIdx.x;
Int j = blockIdx.y * blockDim.y  + threadIdx.y;
int index = i + j*N;
 if (i < N && j < N)
   c[index] = a[index]+b[index];
}

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

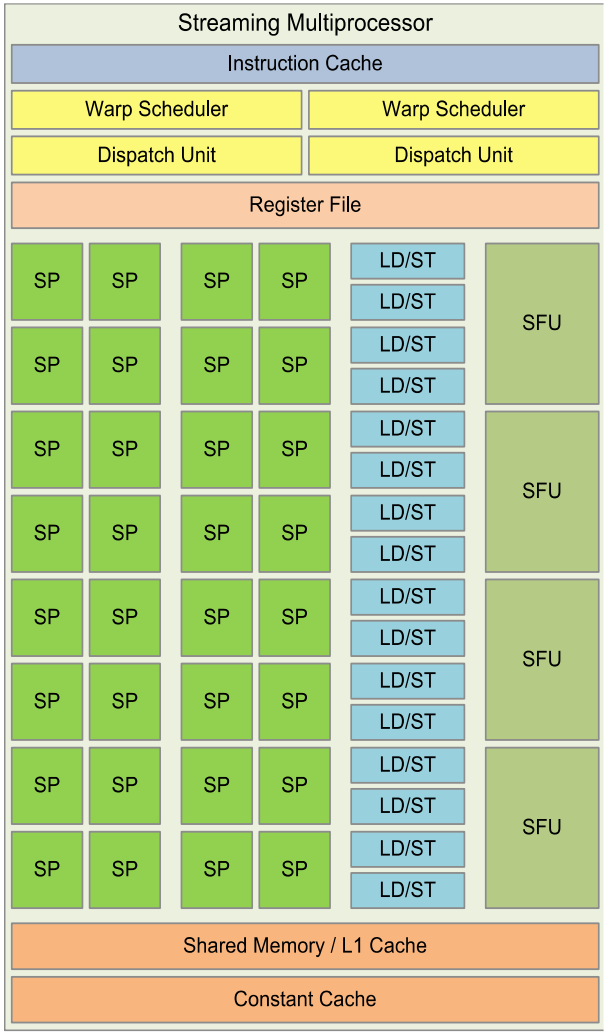
Slide credit: Hyesoon Kim

From Blocks to Warps

- GPU cores: SIMD pipelines
  - Streaming Multiprocessors (SM)
  - Streaming Processors (SP)

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





NVIDIA Fermi architecture

SIMD vs. SIMT Execution Model

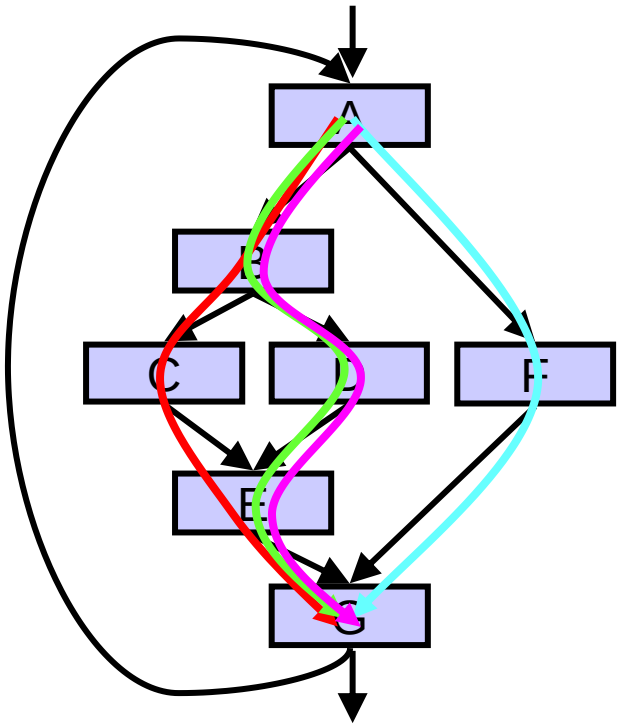
- SIMD: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

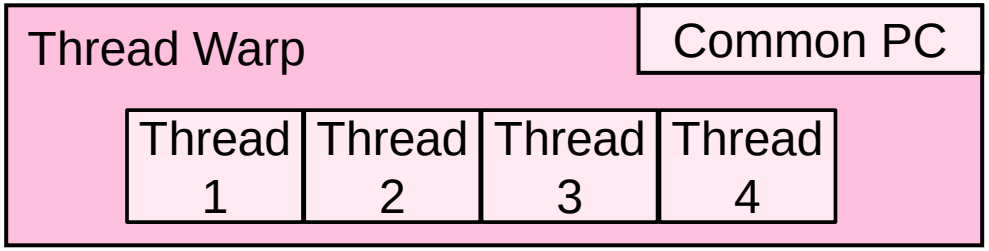
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- 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



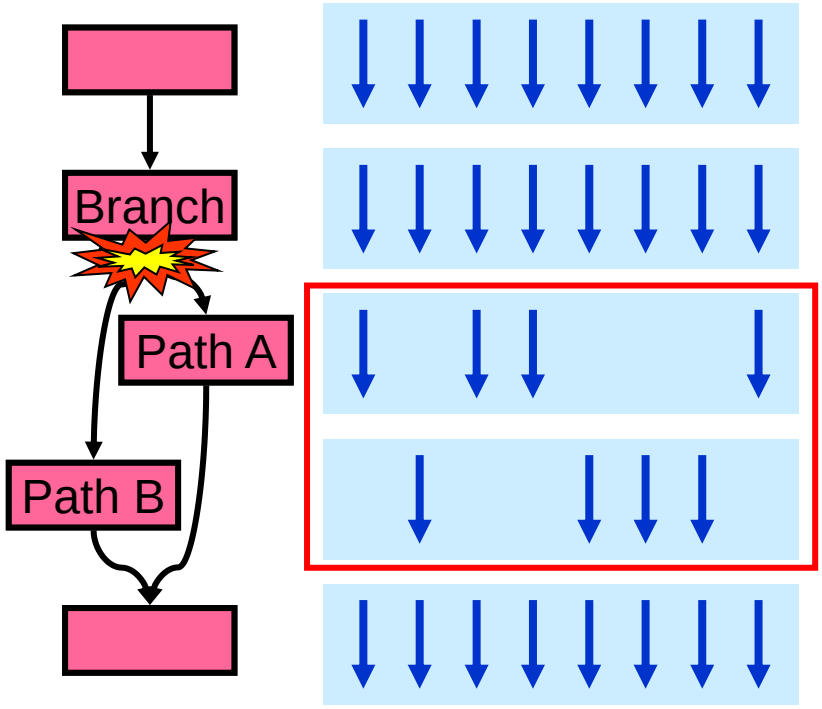


Slide credit: Tor Aamodt

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

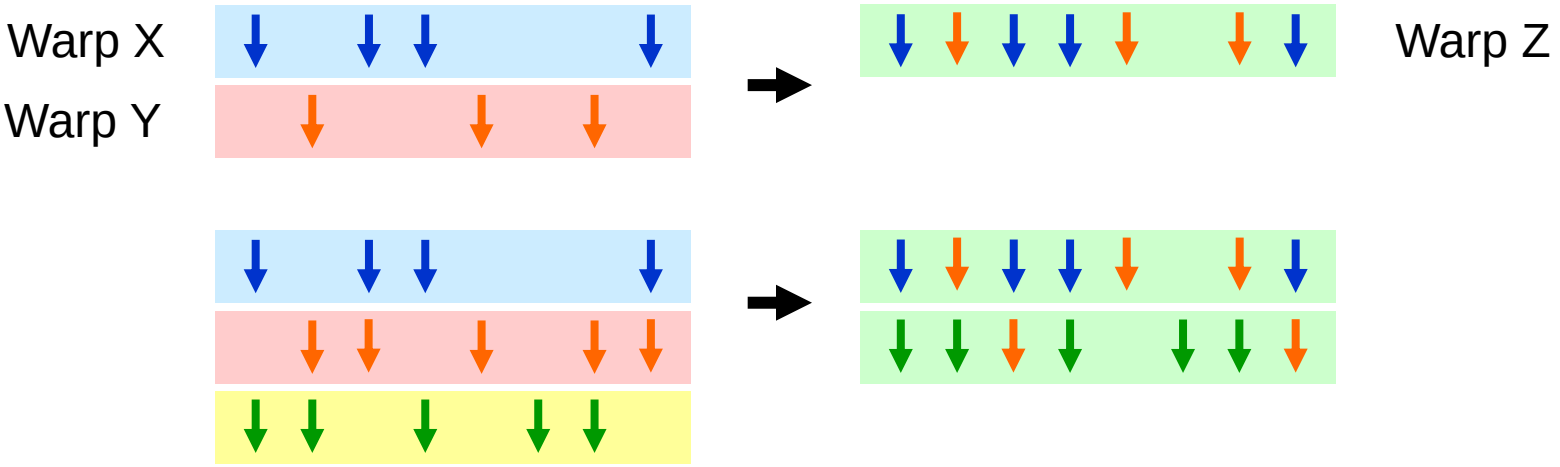
Remember: Each Thread Is Independent

- 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)

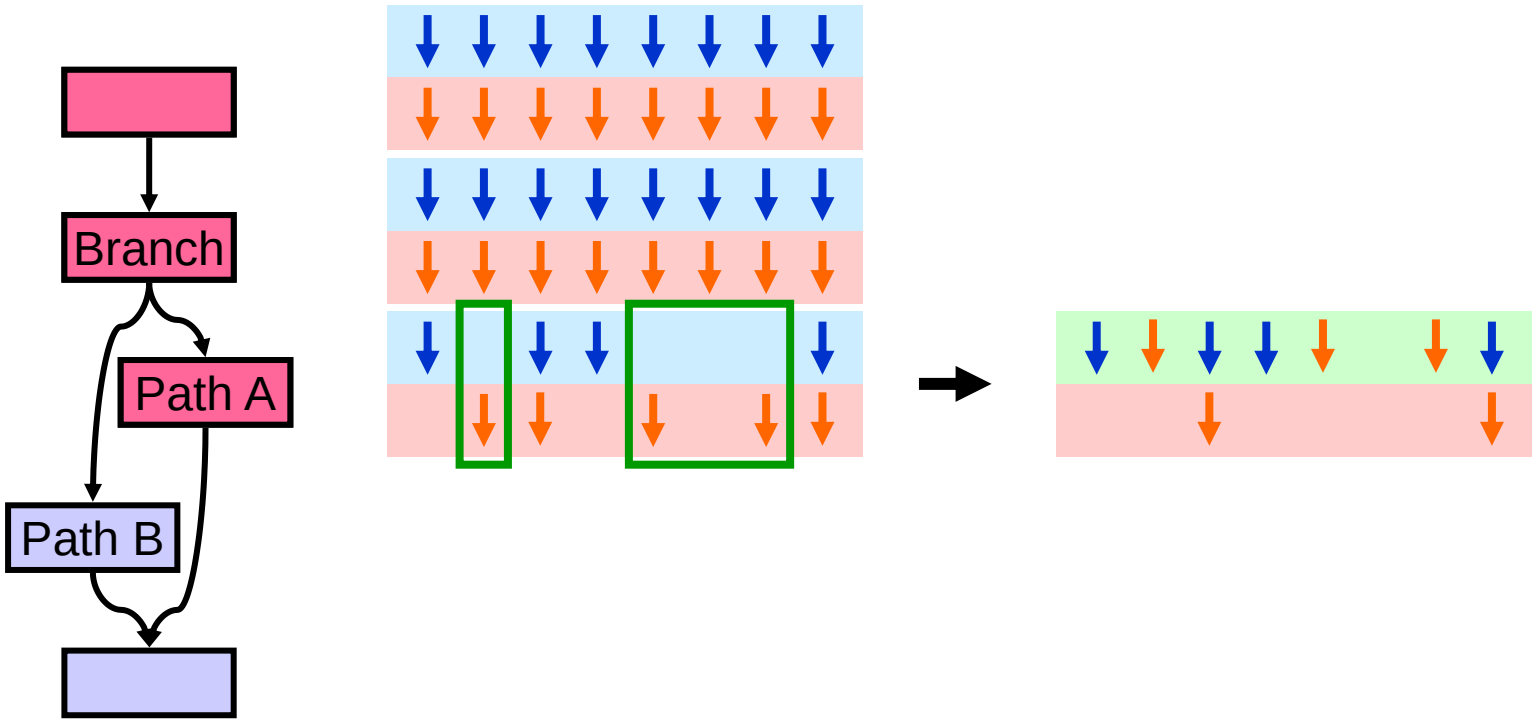
Dynamic Warp Formation/Merging

- 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.

An Example GPU

NVIDIA GeForce GTX 285

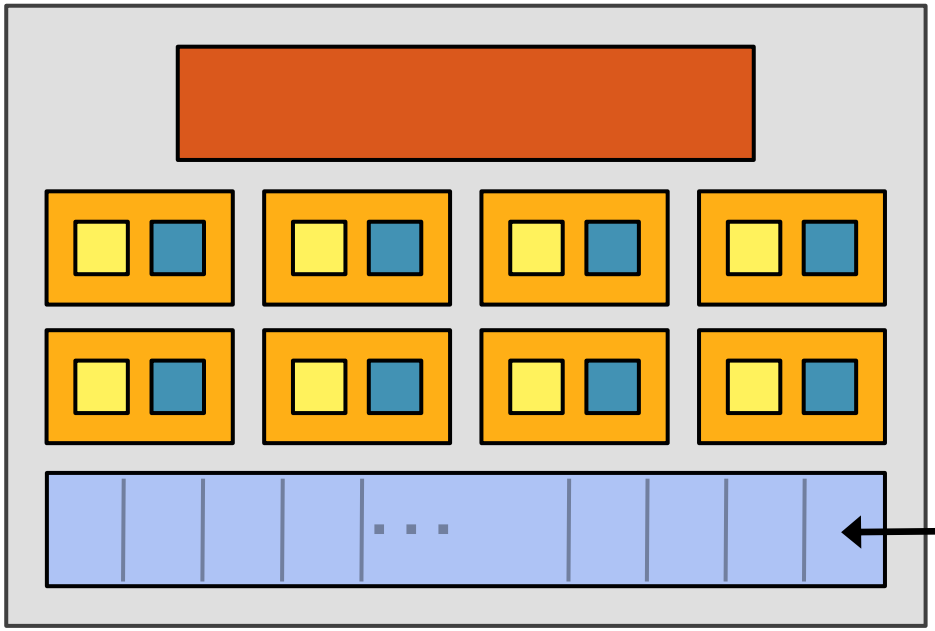
- NVIDIA-speak:
  - 240 stream processors
  - "SIMT execution"

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

- NVIDIA, "NVIDIA GeForce GTX 200 GPU. Architectural Overview. White Paper," 2008.

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

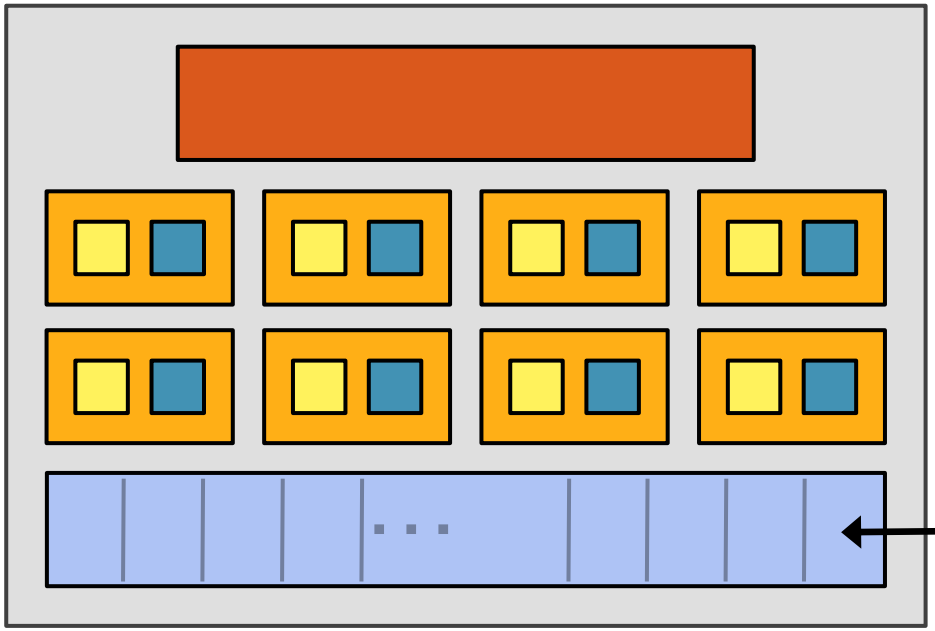
= multiply

= instruction stream decode

= execution context storage

Slide credit: Kayvon Fatahalian

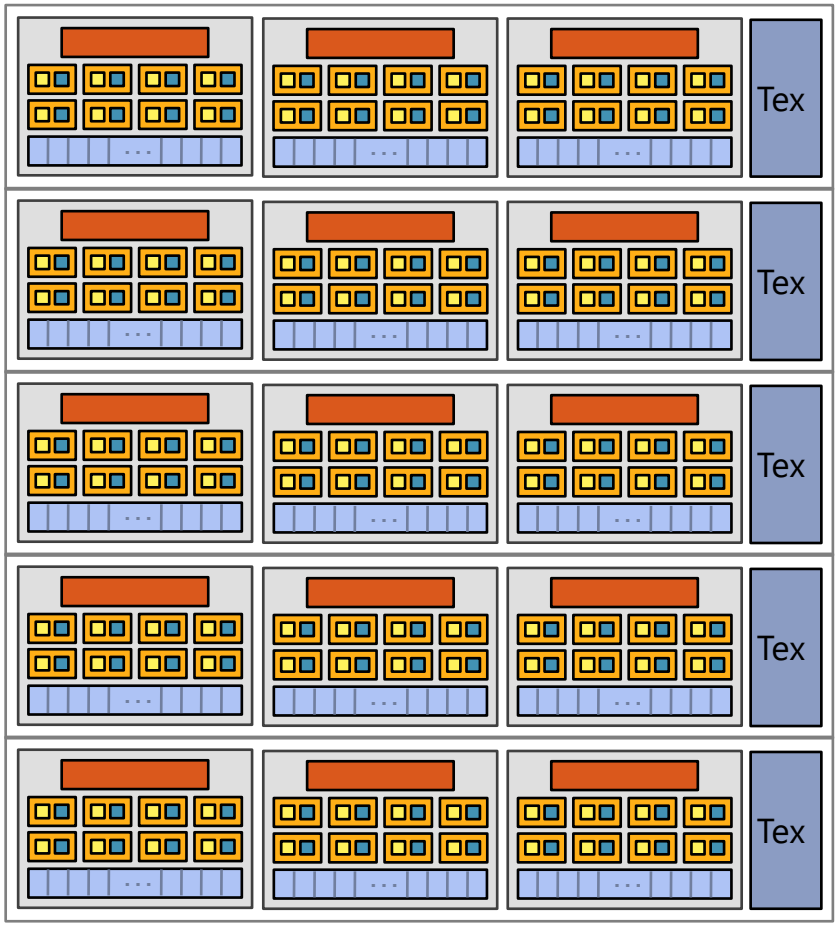
NVIDIA GeForce GTX 285 "core"

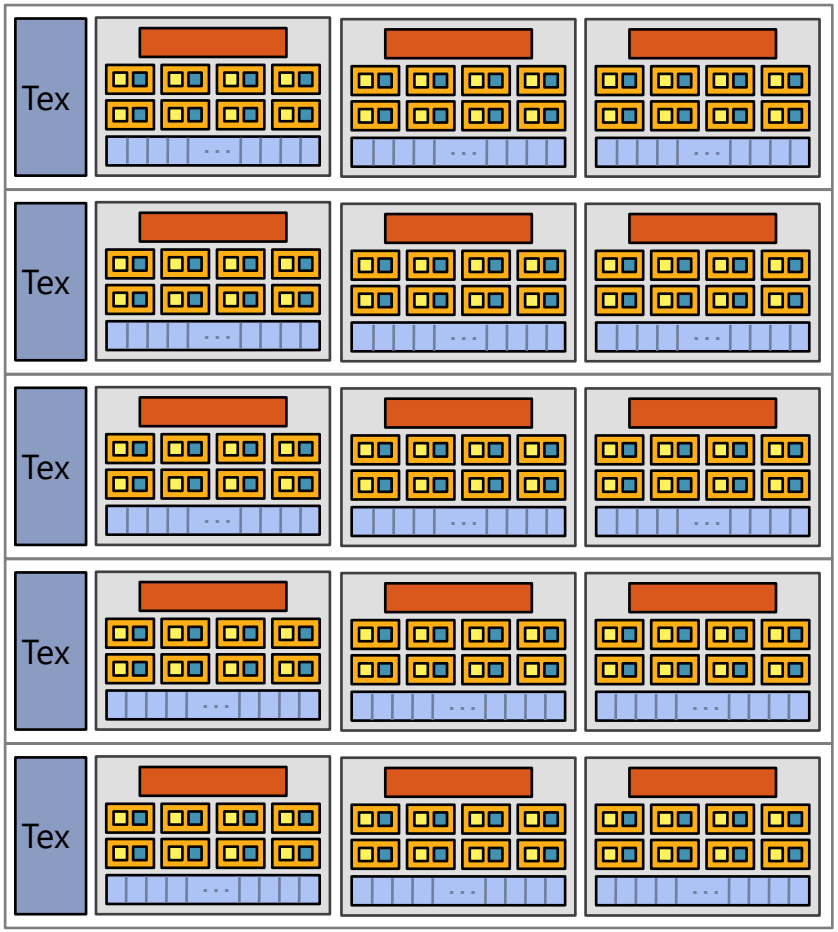


- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285

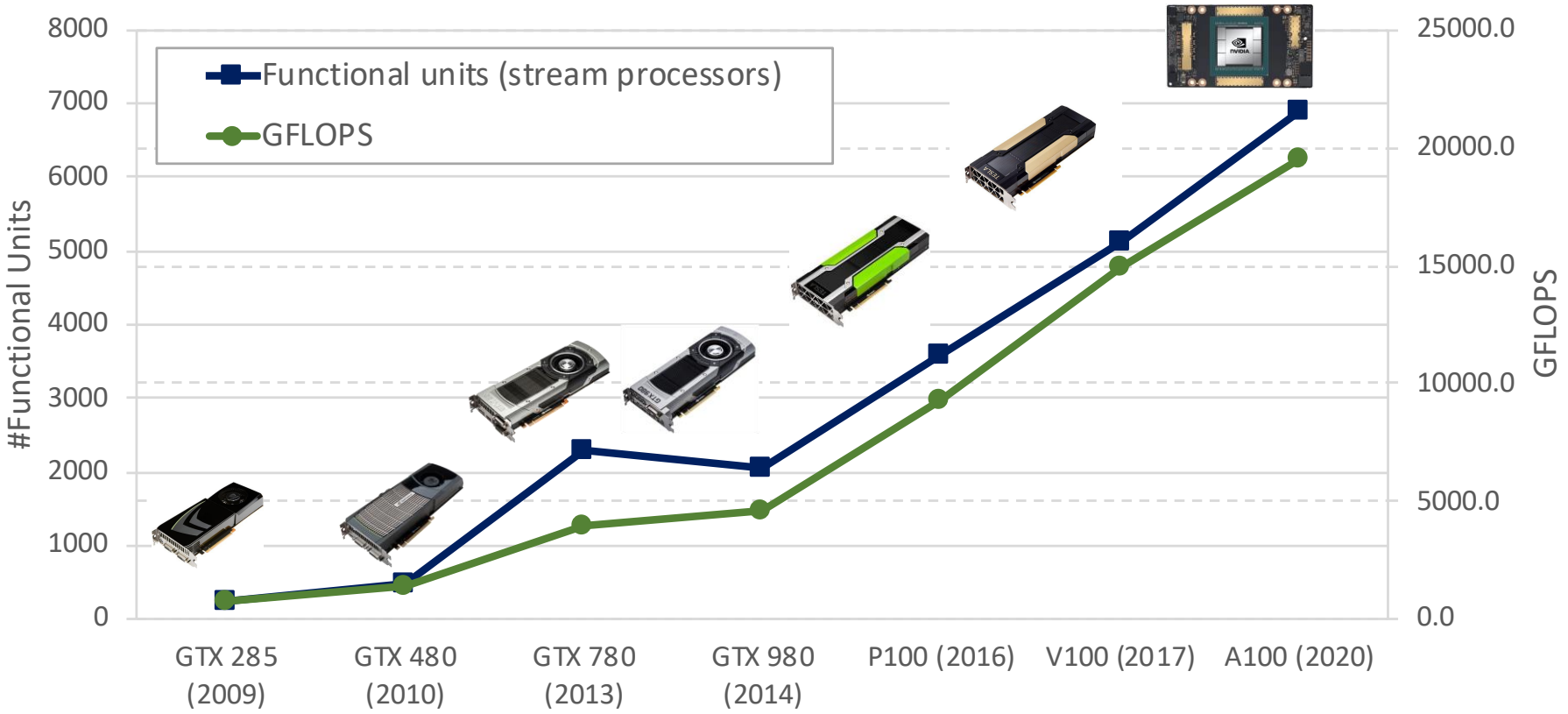




30 cores on the GTX 285: 30,720 threads

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

NVIDIA V100 Block Diagram

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

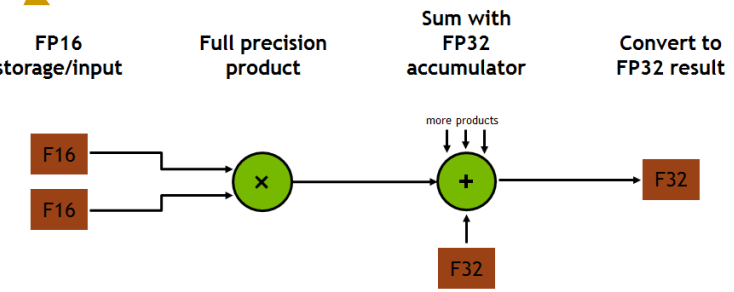
80 cores on the V100

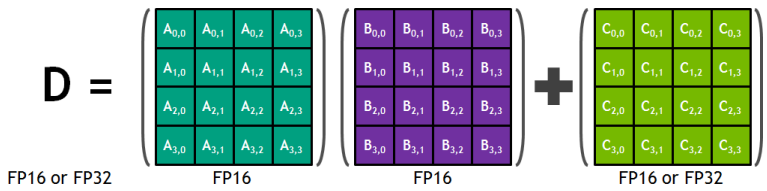
NVIDIA V100 Core

15.7 TFLOPS Single Precision

7.8 TFLOPS Double Precision

125 TFLOPS for Deep Learning (Tensor cores)

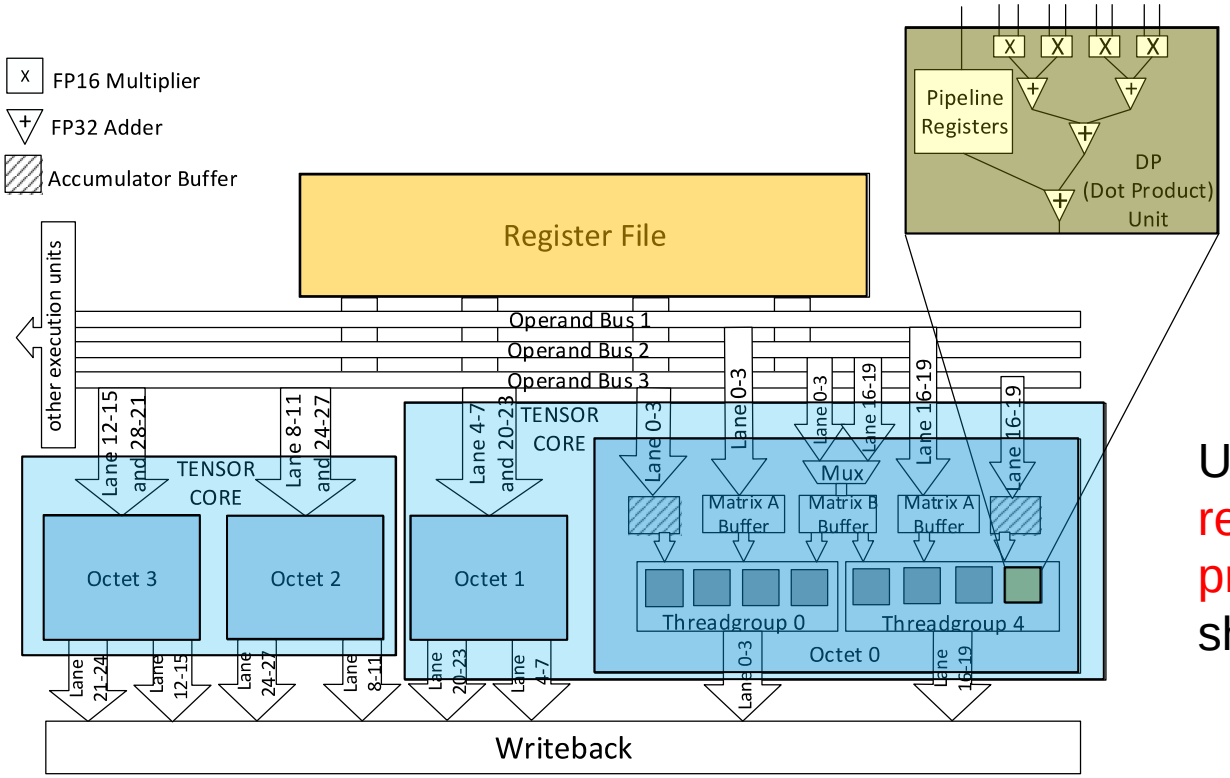




https://devblogs.nvidia.com/inside-volta/

Tensor Core Microarchitecture (Volta)

- Each warp utilizes two tensor cores
- Each tensor core contains two "octets"
  - 16 SIMD units per tensor core (8 per octet)
  - 4x4 matrix-multiply and accumulate each cycle per tensor core



SIMD unit

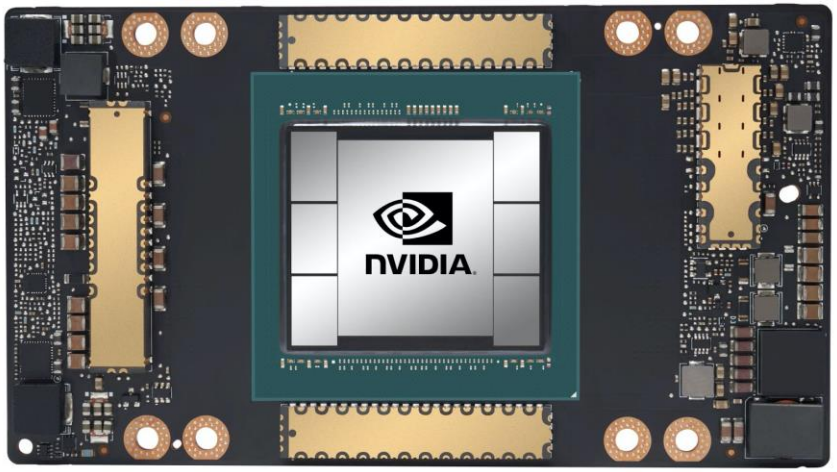
Unlike conventional SIMD, register contents are *not* private to each thread, but shared inside the warp

Proposed* tensor core microarchitecture

* M. A. Raihan, N. Goli and T. M. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," ISPASS 2019.

NVIDIA A100

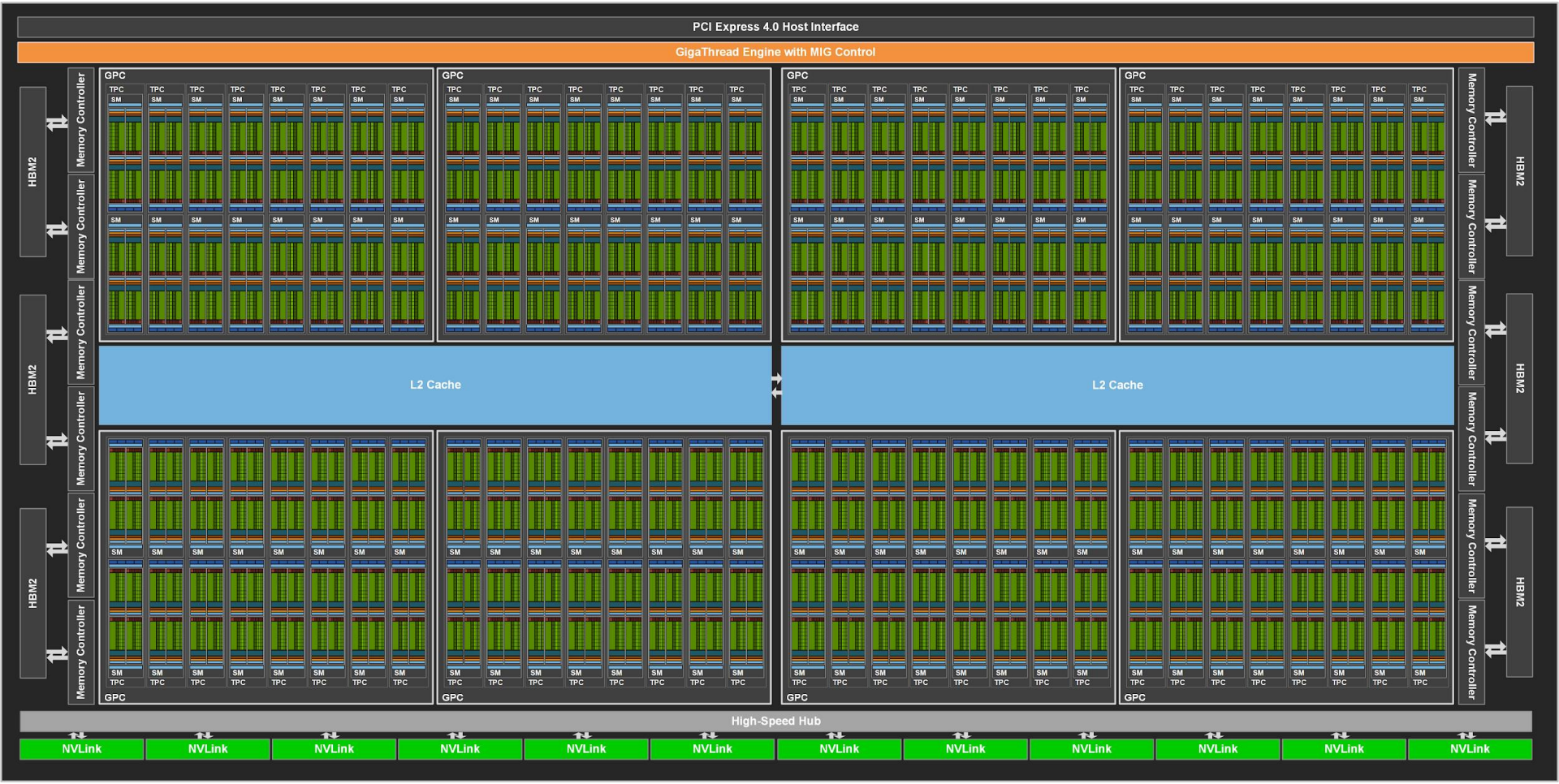
- NVIDIA-speak:
  - 6912 stream processors
  - “SIMT execution”



- Generic speak:
  - 108 cores
  - 64 SIMD functional units per core
  - Tensor cores for Machine Learning
    - Support for sparsity
    - New floating point data type (TF32)

- https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

NVIDIA A100 Block Diagram



https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

108 cores on the A100

(Up to 128 cores in the full-blown chip)

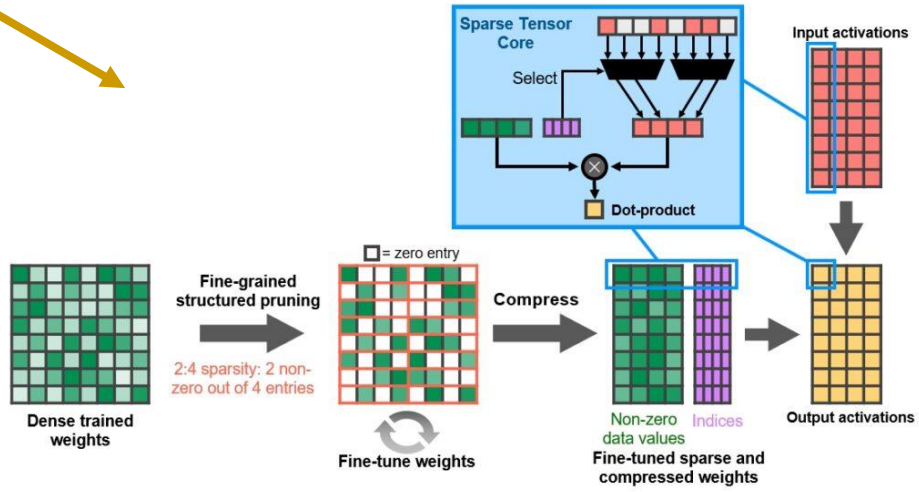
40MB L2 cache

NVIDIA A100 Core

19.5 TFLOPS Single Precision

9.7 TFLOPS Double Precision

312 TFLOPS for Deep Learning (Tensor cores)



https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/