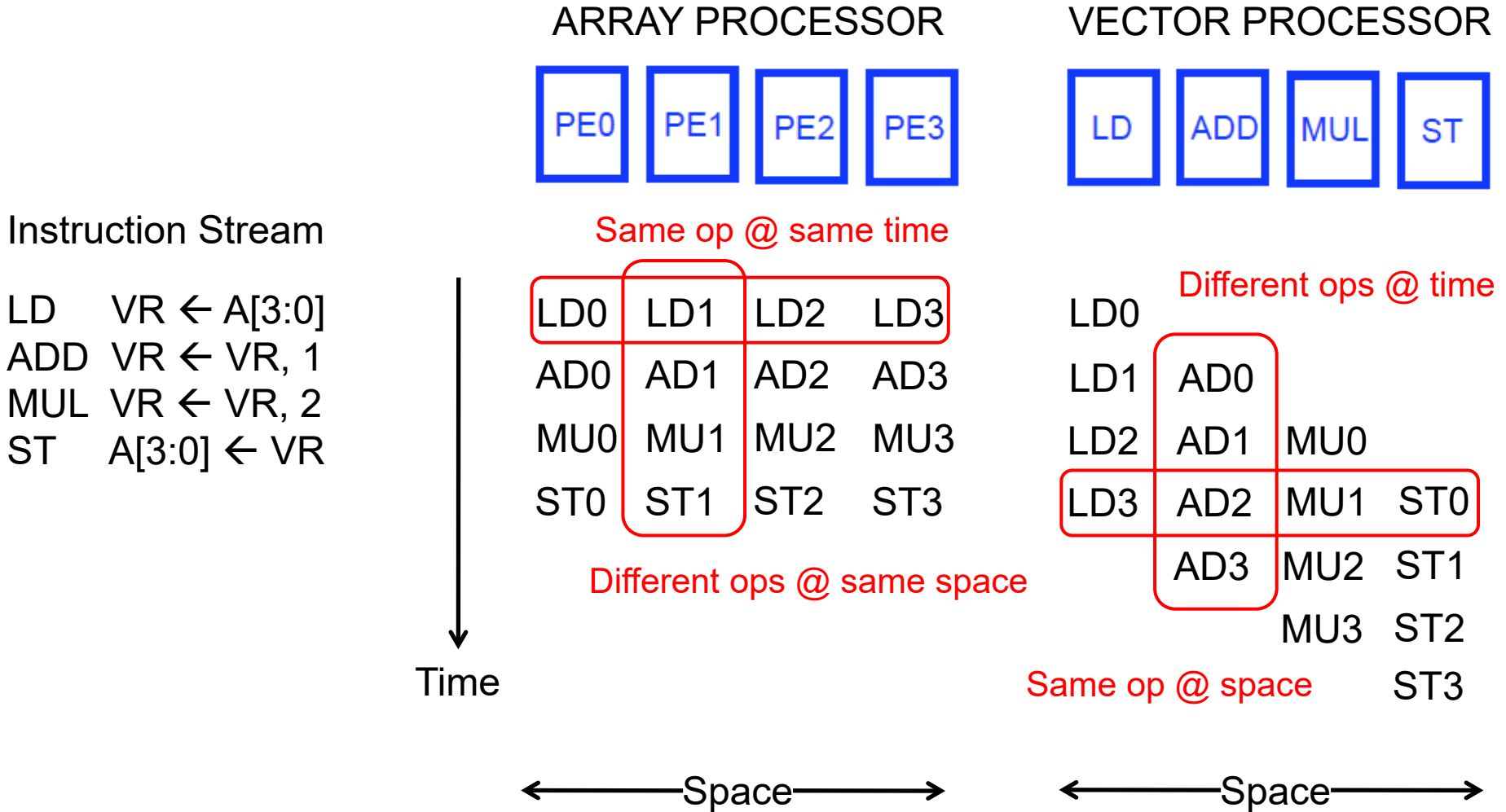


# Recall: Flynn's Taxonomy of Computers

---

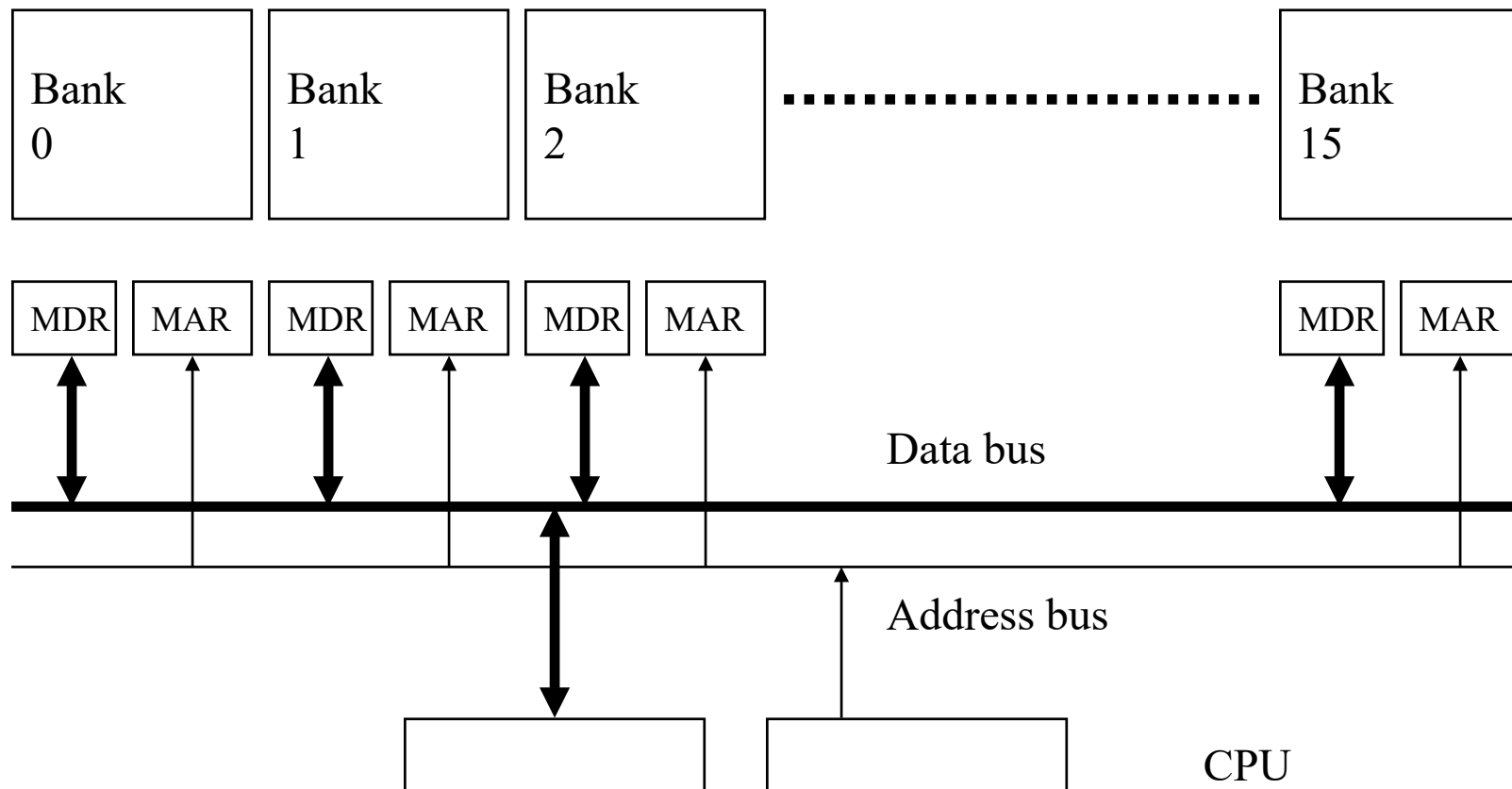
- Mike Flynn, “**Very High-Speed Computing Systems,**” Proc. of IEEE, 1966
- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

# Recall: Array vs. Vector Processors



# Recall: 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



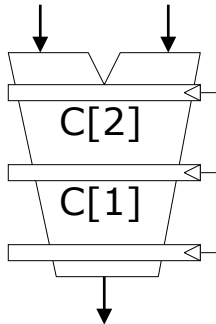
# Recall: Vector Instruction Execution

VADD A,B → C

*Execution using  
one pipelined  
functional unit*

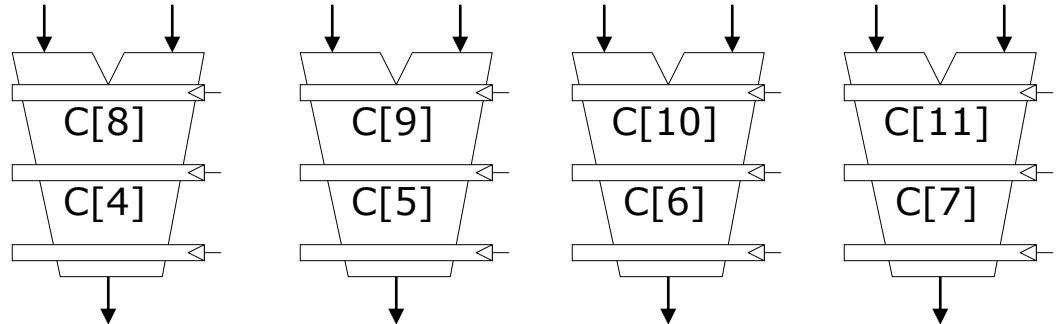
*Execution using  
four pipelined  
functional units*

A[6] B[6]  
A[5] B[5]  
A[4] B[4]  
A[3] B[3]



Time

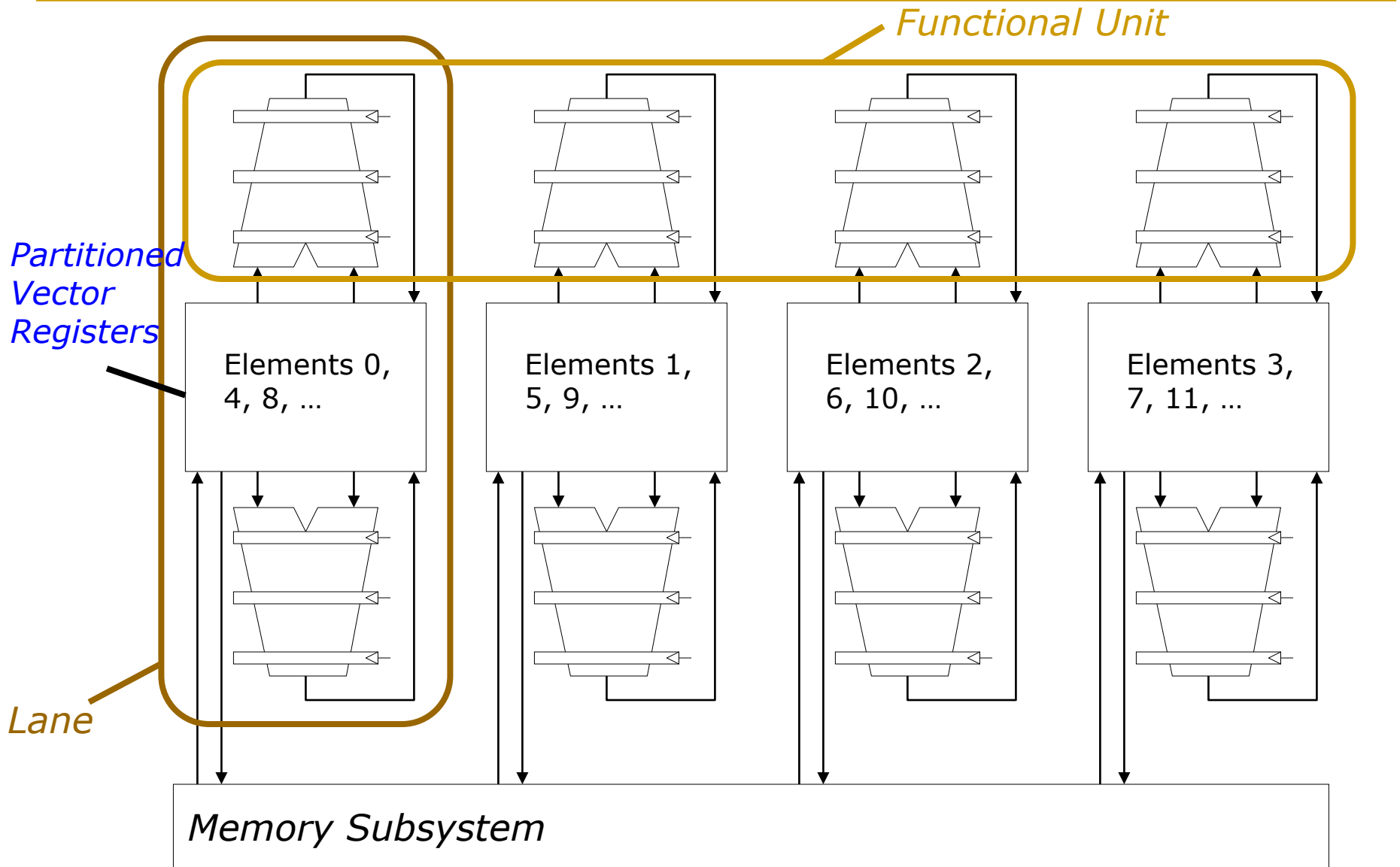
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]  
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]  
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]  
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]



Time

Space

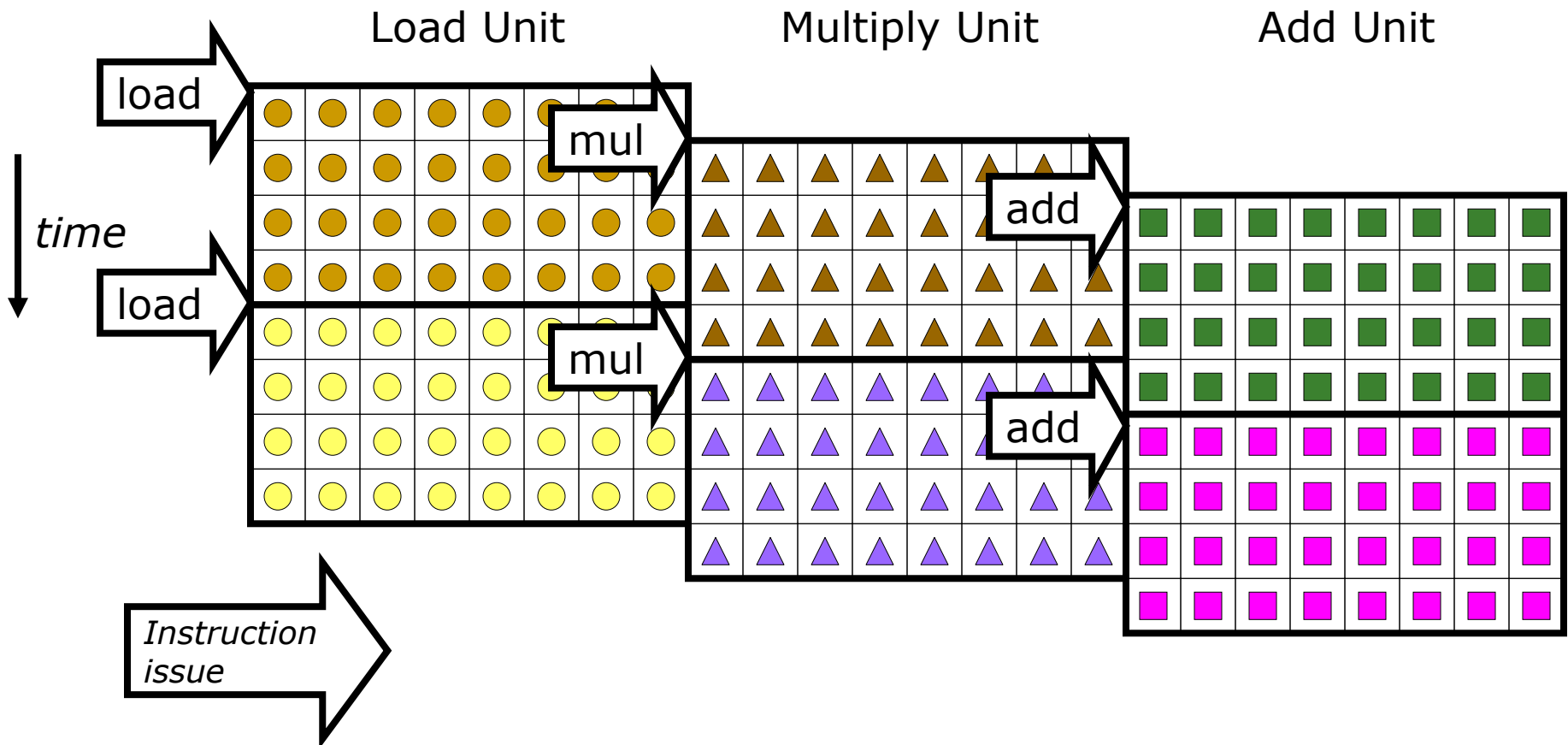
# Recall: Vector Unit Structure



# Recall: Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

- Example machine has 32 elements per vector register and 8 lanes
- Example with 24 operations/cycle (steady state) while issuing 1 vector instruction/cycle



# Recall: Vector Processor Disadvantages

---

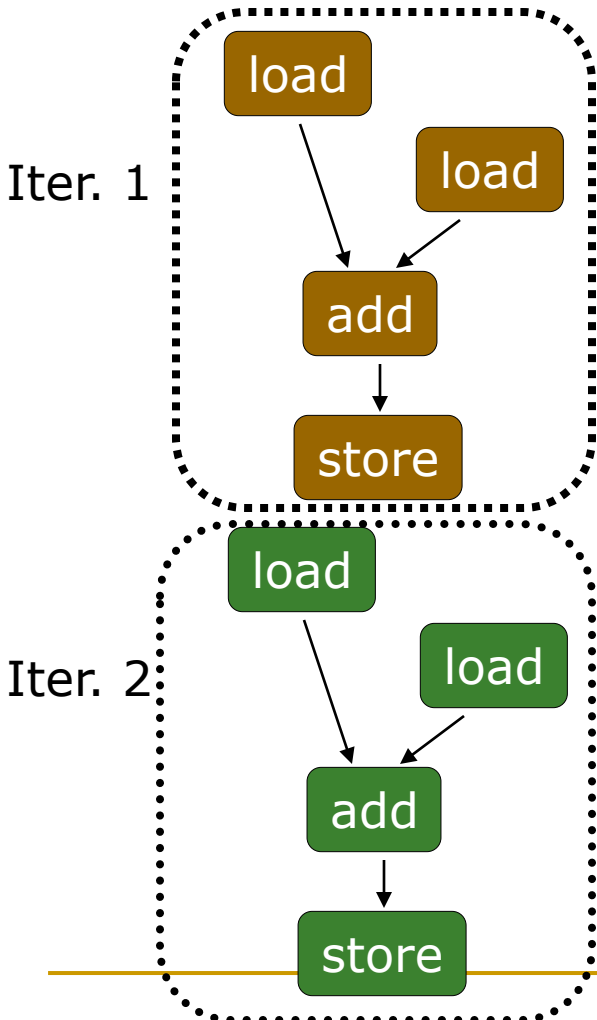
- Works (only) if parallelism is regular (data/SIMD parallelism)
  - ++ Vector operations
  - Very inefficient if parallelism is irregular
    - How about searching for a key in a linked list?

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

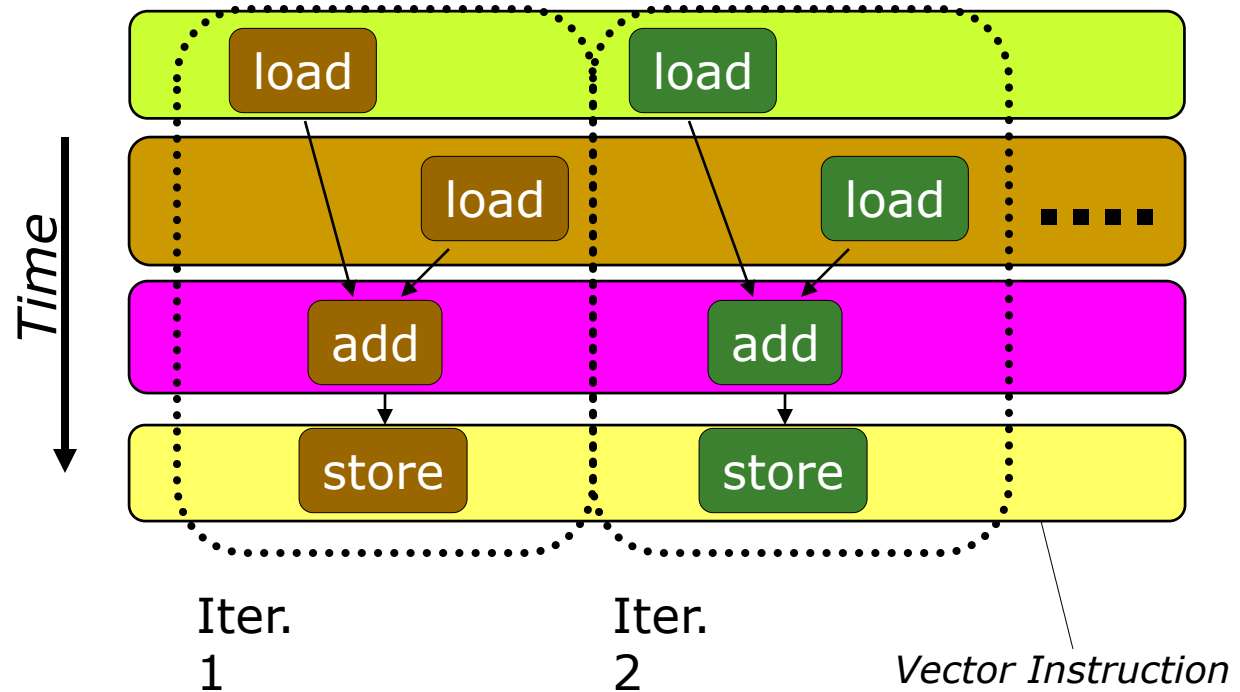
# Recall: Automatic Code Vectorization

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

*Scalar Sequential Code*



*Vectorized Code*



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



# Recall: Vector/SIMD Processing Summary

---

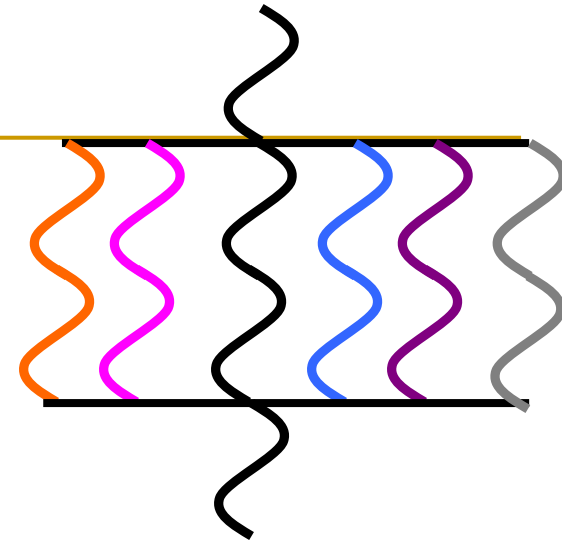
- Vector/SIMD machines are good at exploiting **regular data-level parallelism**
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)
- **Performance improvement limited by vectorizability** of code
  - Scalar operations limit vector machine performance
  - Remember **Amdahl's Law**
  - CRAY-1 was the fastest SCALAR machine at its time!
- **Many existing ISAs include SIMD operations**
  - Intel MMX/SSEn/AVX/AMX, PowerPC AltiVec, ARM Advanced SIMD, MIPS SIMD, ...

# Recall: Amdahl's Law

- Amdahl's Law

- f: Parallelizable fraction of a program
- N: Number of processors

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



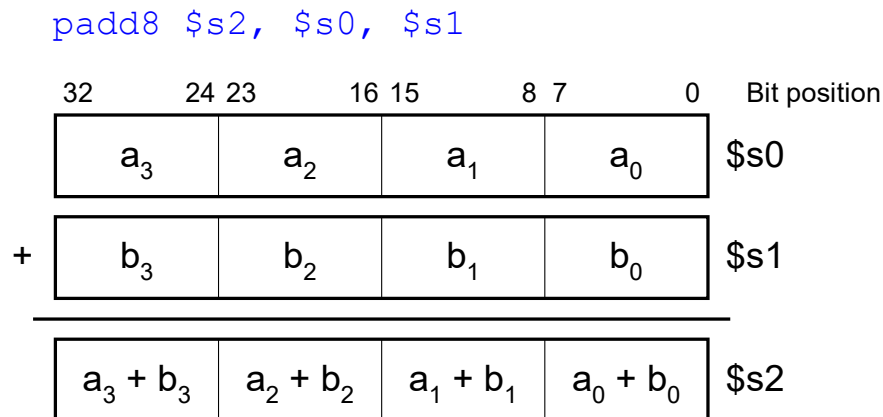
- Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” AFIPS 1967.

- **Maximum speedup limited by serial portion: Serial bottleneck**
- **All parallel machines “suffer from” the serial bottleneck**

# SIMD Operations in Modern ISAs

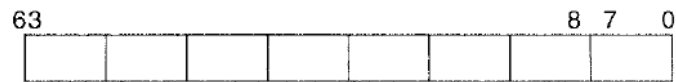
# SIMD ISA Extensions

- Single Instruction Multiple Data (SIMD) extension instructions
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics, multimedia, image processing
  - 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



# Intel Pentium MMX Operations

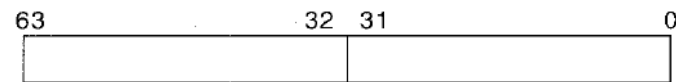
- Idea: One instruction operates on multiple data elements **simultaneously**
  - *À la* array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind



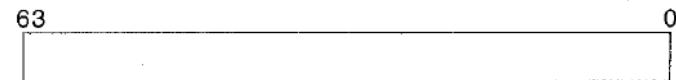
(a)



(b)



(c)



(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.

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

# Intel Pentium MMX Operations (II)

---

51	3	5	23
>	>	>	>
73	2	5	6
000 ... 0	111 ... 1	000 ... 0	111 ... 1

Figure 3. Packed compare greater-than word.

PCMPEQ(b,w,d), PCMPGT(b,w,d)	Equal or greater than	1	Compares packed 8 bytes, four 16-bit words, or two 32-bit elements in parallel. Result is mask of 1s if true or 0s if false.
---------------------------------	-----------------------	---	--

# Intel Pentium MMX Operations (II)

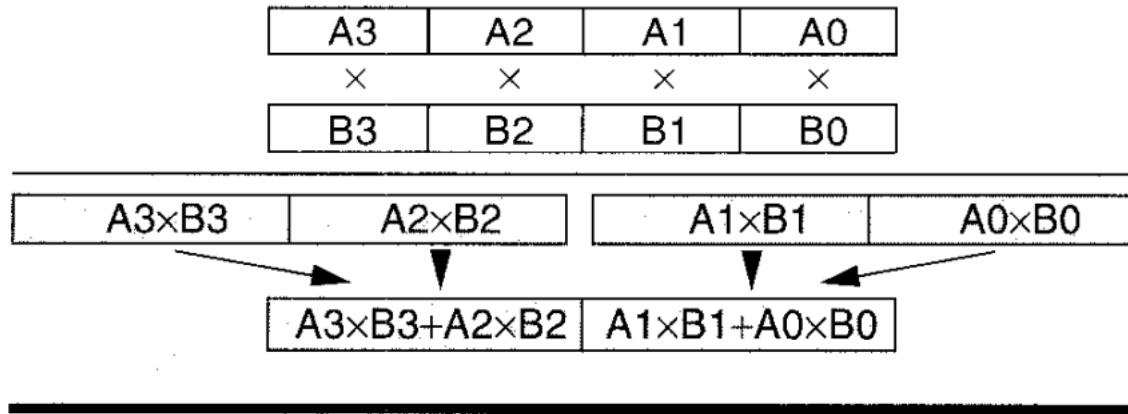


Figure 2. Packed multiply-add word to doubleword.

PMADDWD      Word to doubleword conversion      Latency: 3; throughput: 1      Multiplies four packed, signed 16-bit words and adds together adjacent pairs of 32-bit results in parallel. Result is a doubleword.

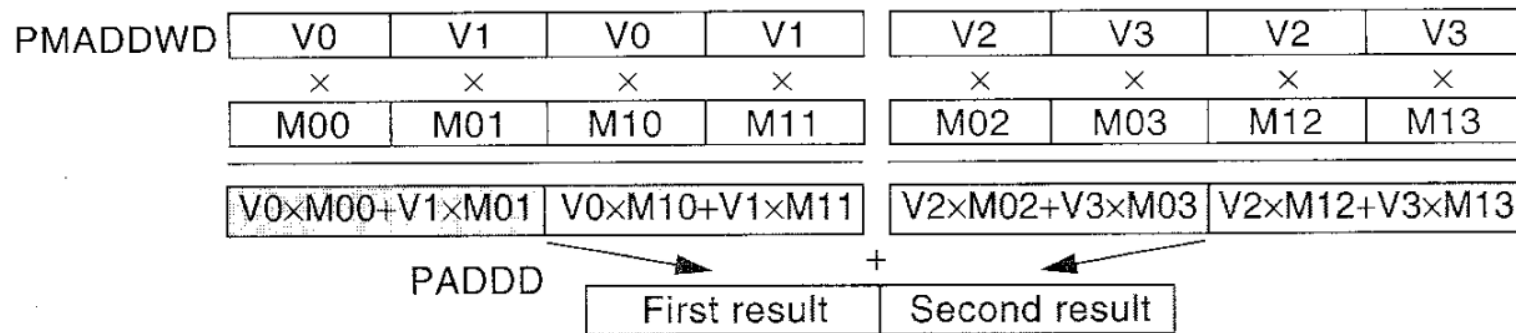


Figure 7. Flow diagram of matrix-vector multiply.

# MMX Example: Image Overlaying (I)

- Goal: Overlay the human in image  $x$  on top of the background in image  $y$

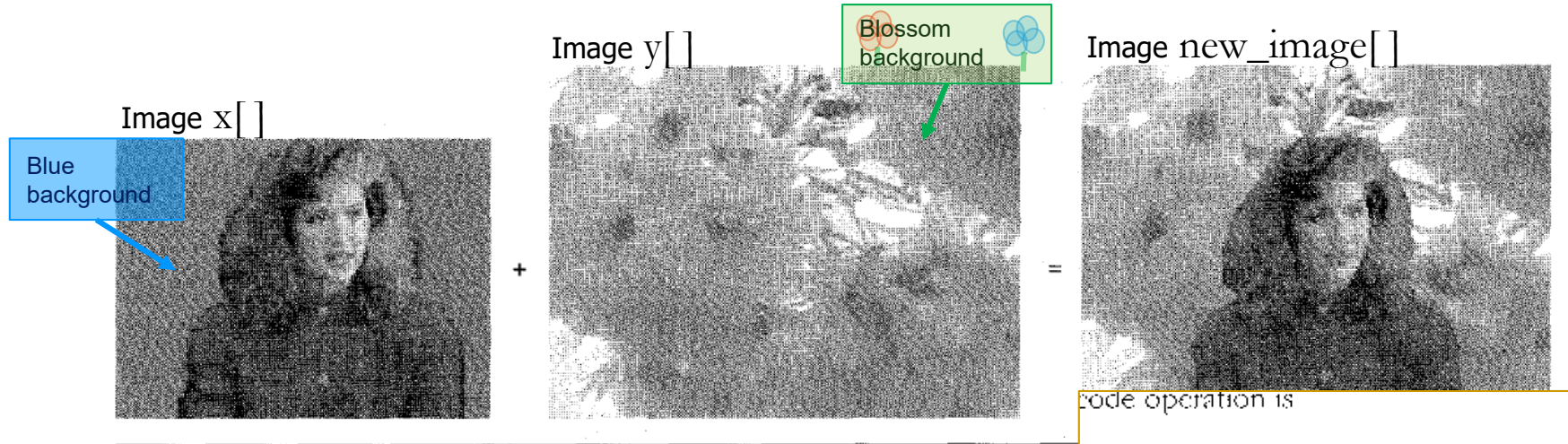


Image  $x[]$

Blue background

Image  $y[]$

Blossom background

Image  $new\_image[]$

code operation is

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

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



# MMX Example: Image Overlaying (II)

- Goal: Overlay the human in image  $x$  on top of the background in image  $y$

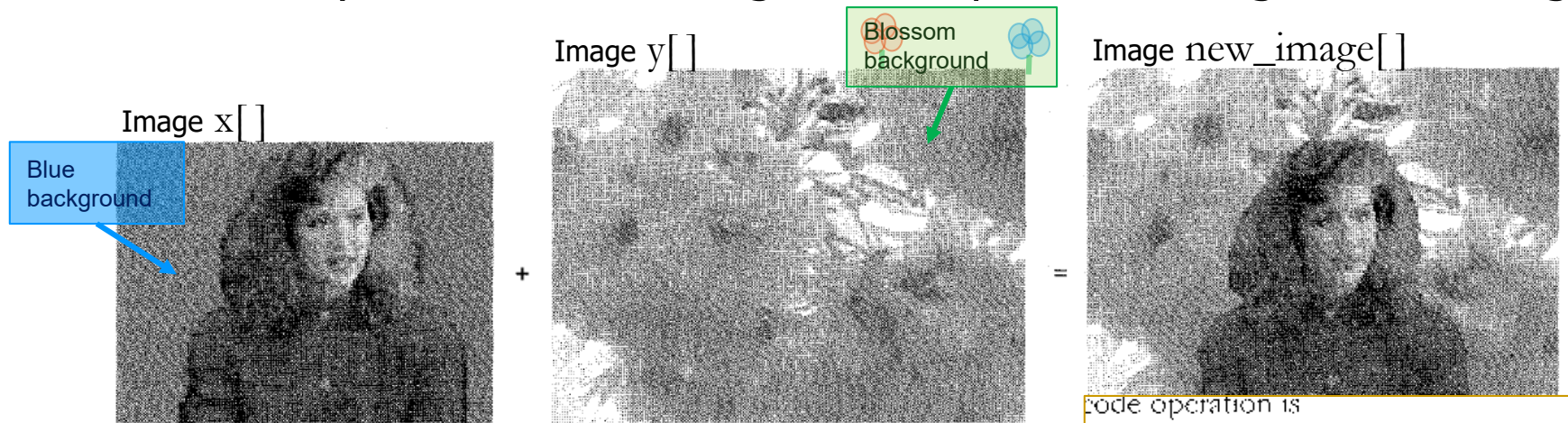
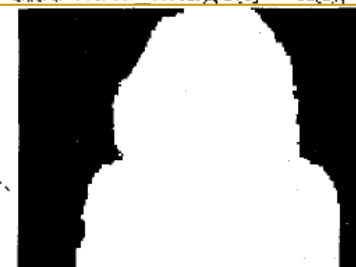


Figure 8. Chroma keying: image overlay using a background color.

PCMPEQB MM1, MM3

MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
Image $x$ [ ] MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
Bit mask MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF



Bitmask

Figure 9. Generating the selection bit mask.

# MMX Example: Image Overlaying (III)

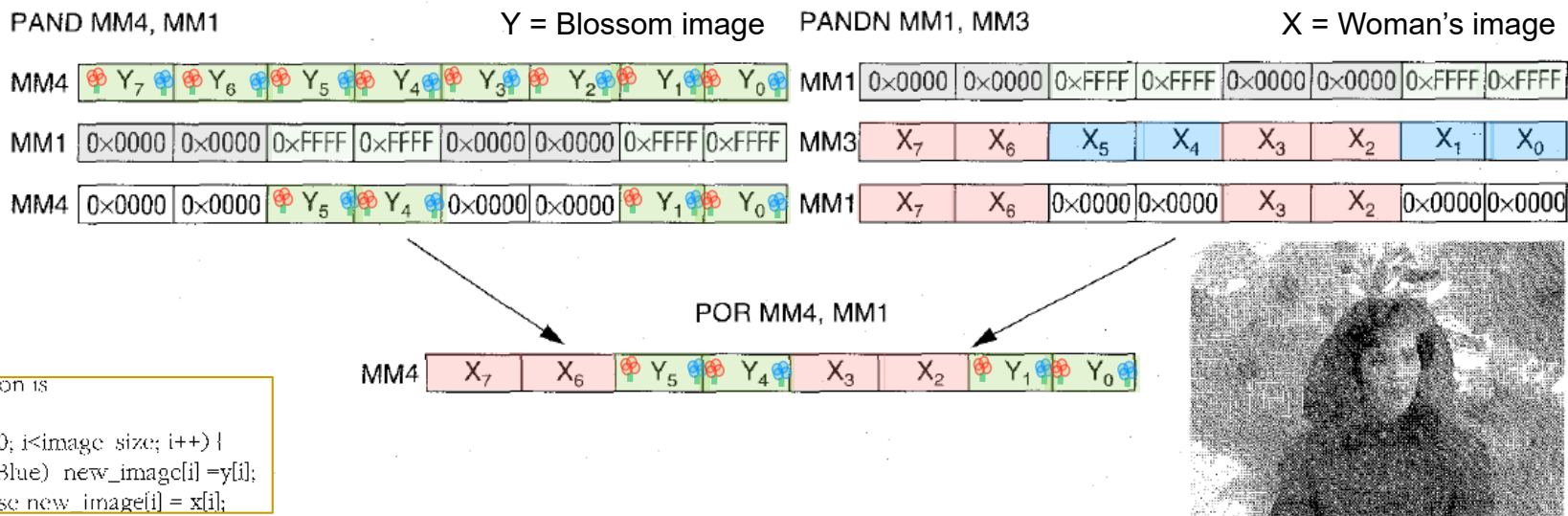


Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```

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

```


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

# Intel Pentium MMX Operations

---

MMX technology enhances applications that benefit from SIMD architecture and parallelism. MMX speeds up computationally intensive inner loops or subroutines on average between three to five times. When these are applied to the full application, that application typically runs on the same processor 1.5 to 2 times faster than the same application without MMX technology.

For example, a certain MPEG-1 video decoding application on a Pentium class processor with MMX technology executes 1.5 times faster than the same application on the same processor not using MMX technology. An assortment of image filters in an image-processing application execute just over four times faster.

INTEL PLANS TO IMPLEMENT MMX technology on future Pentium and Intel architecture processors. It will make MMX technology a base capability on all company CPUs to allow existing and new applications to run faster. We believe the performance gains from this technology will scale well with the CPU operating frequency and future Intel microarchitecture generations. 

# From MMX to AMX in x86 ISA

## ■ MMX

- ❑ 64-bit MMX registers for integers

## ■ SSE (Streaming SIMD Extensions)

- ❑ SSE-1: 128-bit XMM registers for integers and single-precision floating point
- ❑ SSE-2: Double-precision floating point
- ❑ SSE-3, SSSE-3 (supplemental): New instructions
- ❑ SSE-4: New instructions (not multimedia specific), shuffle operations

## ■ AVX (Advanced Vector Extensions)

- ❑ AVX: 256-bit floating point
- ❑ AVX2: 256-bit floating point with FMA (Fused Multiply Add)
- ❑ AVX-512: 512-bit

## ■ AMX (Advanced Matrix Extensions)

- ❑ Designed for AI/ML workloads
- ❑ 2-dimensional registers
- ❑ Tiled matrix multiply unit (TMUL)

### Tile matrix multiply unit [\[ edit \]](#)

TMUL unit supports BF16 and INT8 input types.<sup>[6]</sup> AMX-FP16 also adds support for FP16 numbers and AMX-COMPLEX - for FP16 complex numbers, where a pair of adjacent FP16 numbers represent real and imaginary parts of the complex number. The register file consists of 8 tiles, each with 16 rows of size of 64 bytes (32 BF16/FP16 or 64 INT8 elements). The only supported operation as for now is matrix multiplication  $C_{nm} + = \sum_{k=1}^K A_{nk} B_{km}$ .<sup>[7]</sup>

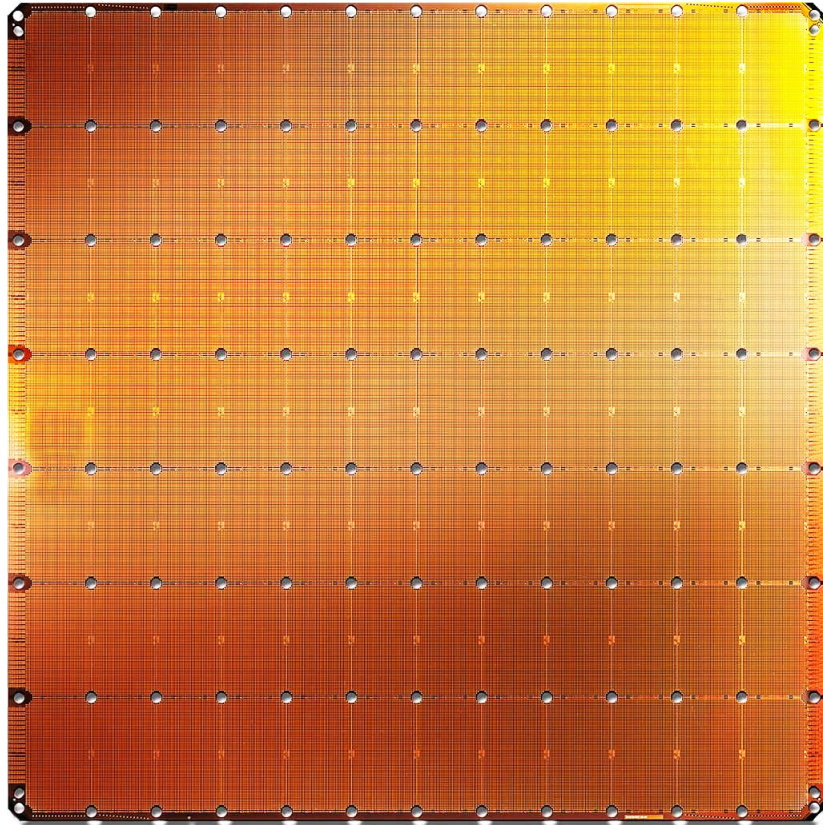
Ops/cycle per core:<sup>[8]</sup>

- Intel AMX-INT8: 2048 (=16 \* 64 \* 2)
- Intel AMX-BF16: 1024 (=16 \* 32 \* 2)

[https://en.wikipedia.org/wiki/Advanced\\_Matrix\\_Extensions](https://en.wikipedia.org/wiki/Advanced_Matrix_Extensions)

# SIMD Operations in Modern (Machine Learning) Accelerators

# Cerebras's Wafer Scale Engine (2019)



## Cerebras WSE

1.2 Trillion transistors  
46,225 mm<sup>2</sup>

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



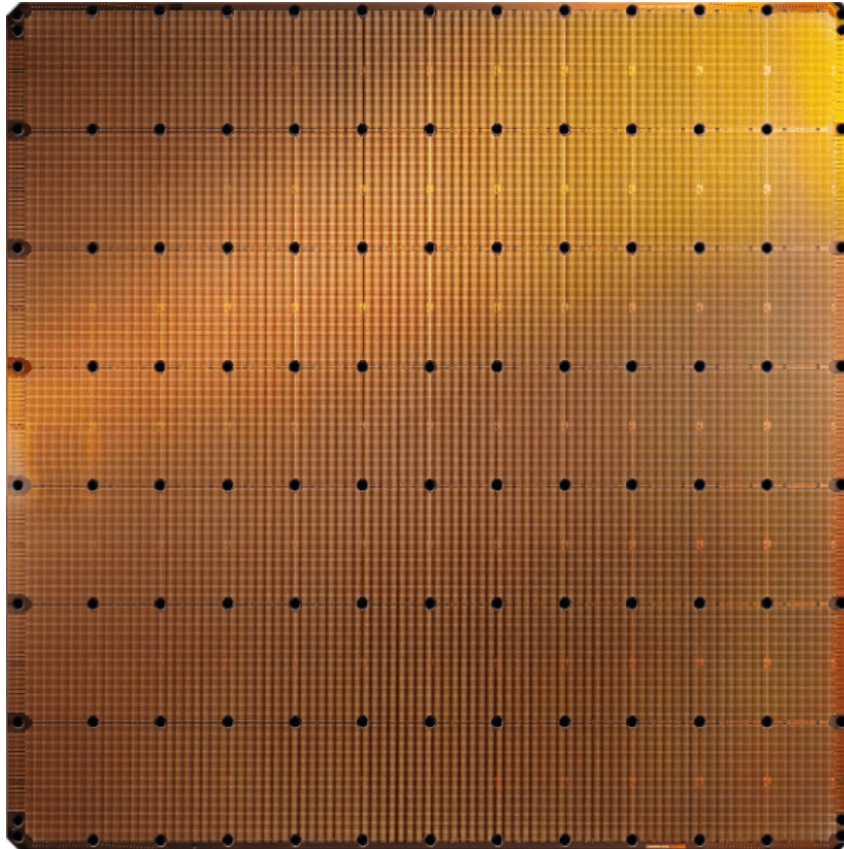
## Largest GPU

21.1 Billion transistors  
815 mm<sup>2</sup>  
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)



**Cerebras WSE-2**  
2.6 Trillion transistors  
46,225 mm<sup>2</sup>

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



**Largest GPU**  
54.2 Billion transistors  
826 mm<sup>2</sup>  
NVIDIA Ampere GA100

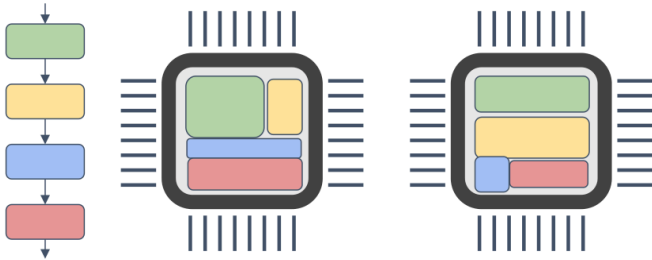
<https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning>

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

# Size, Place, and Route in Cerebras's WSE

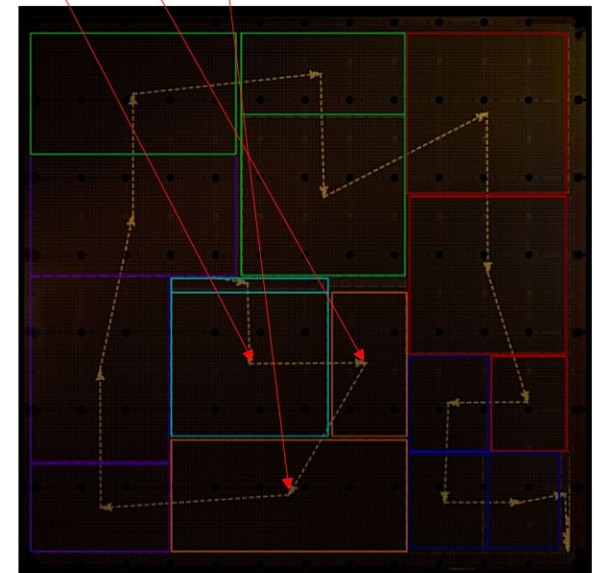
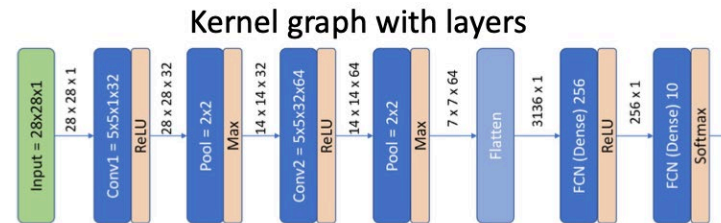
- Neural network mapping onto the whole wafer is a challenge

Multiple possible mappings



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

An example mapping



Layers mapped on Wafer Scale Engine



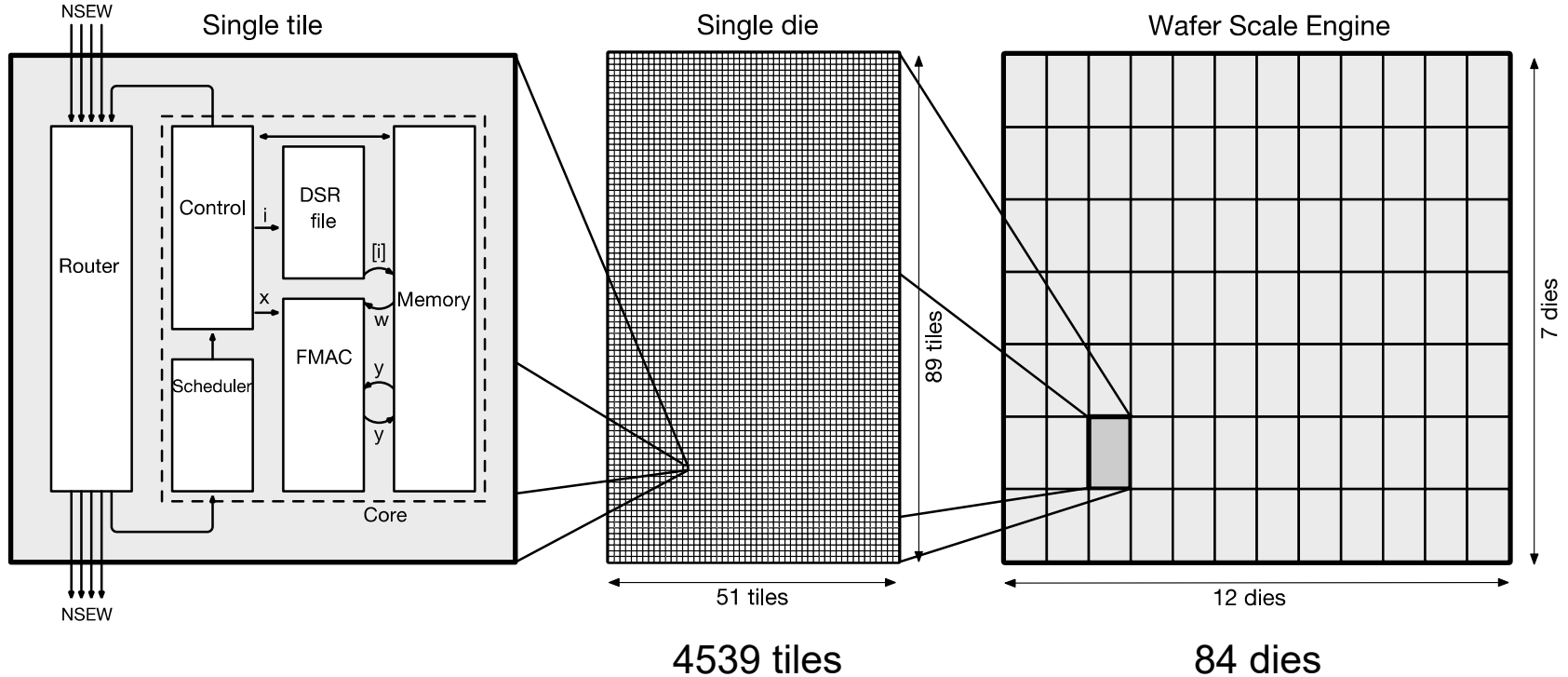
# Recall: Flynn's Taxonomy of Computers

---

- 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

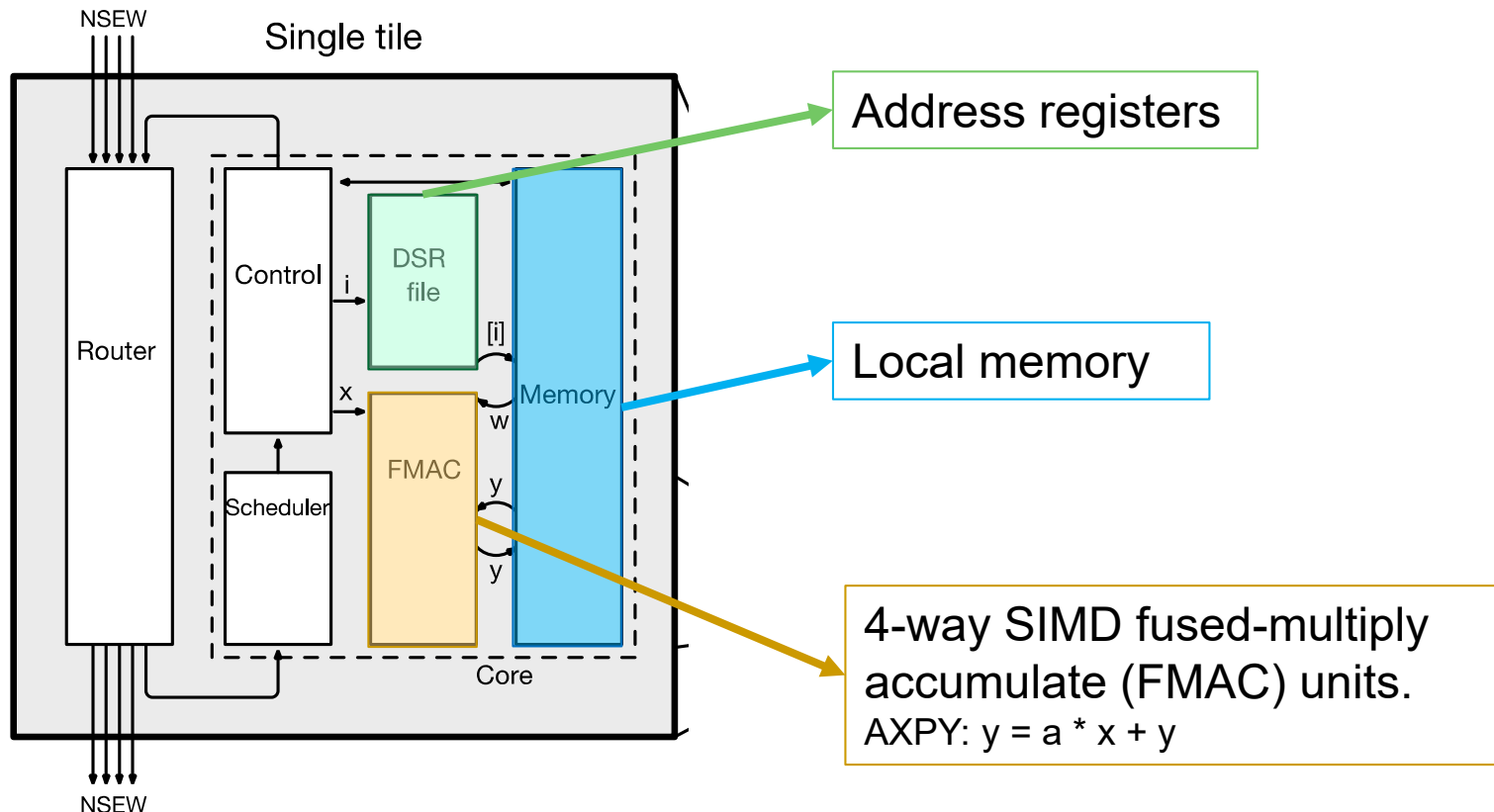
# A MIMD Machine with SIMD Processors (I)

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



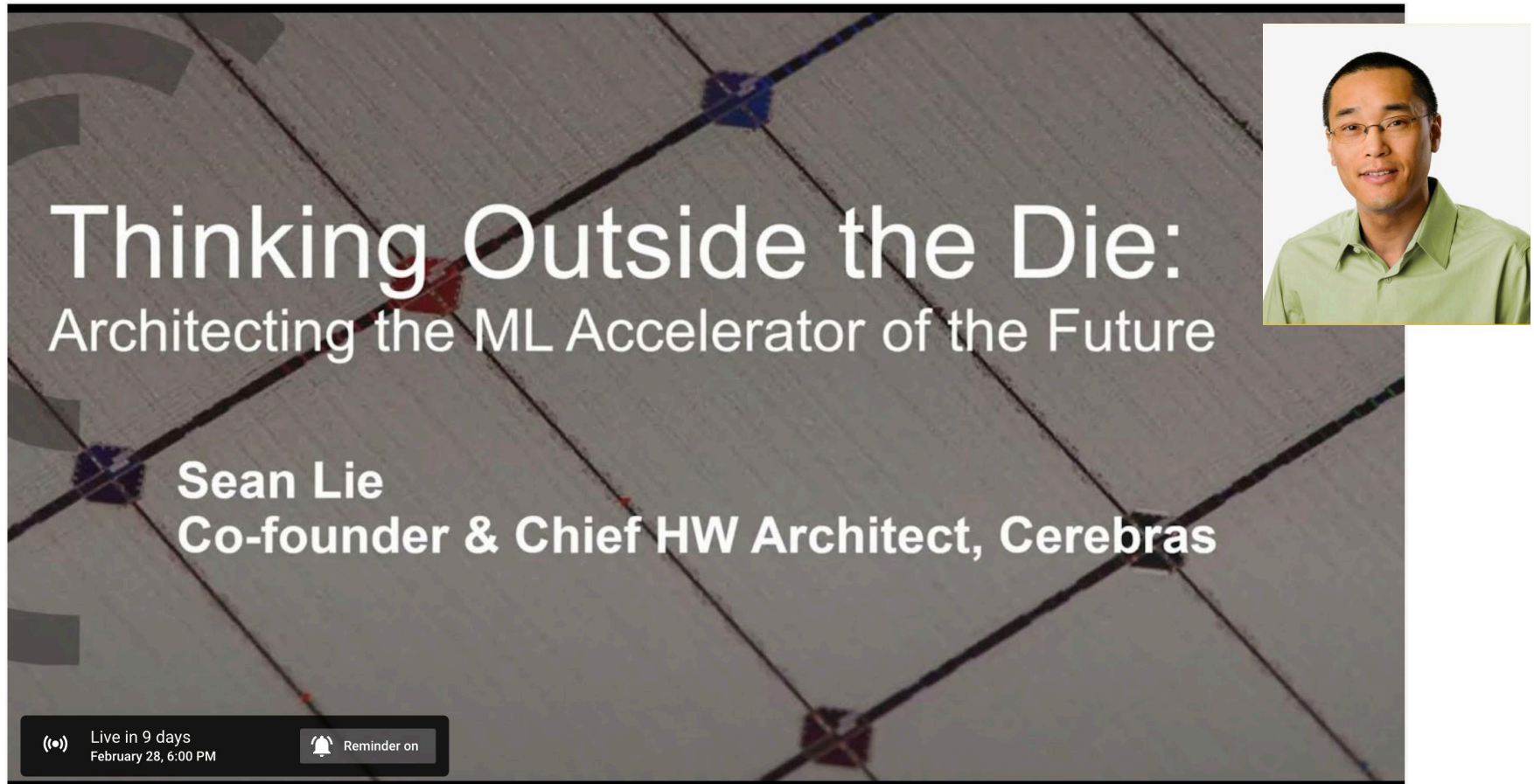
# A MIMD Machine with SIMD Processors (II)

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



# More on the Cerebras WSE

<https://www.youtube.com/watch?v=x2-qB0J7KHw>



Thinking Outside the Die:  
Architecting the ML Accelerator of the Future

Sean Lie  
Co-founder & Chief HW Architect, Cerebras

Live in 9 days  
February 28, 6:00 PM

Reminder on

SAFARI Live Seminar - Thinking Outside the Die: Architecting the ML Accelerator of the Future

1 waiting · Scheduled for Feb 28, 2022

👍 7 🗨 DISLIKE ➦ SHARE ⌵+ SAVE ...



Onur Mutlu Lectures  
22.6K subscribers

ANALYTICS

EDIT VIDEO

# 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

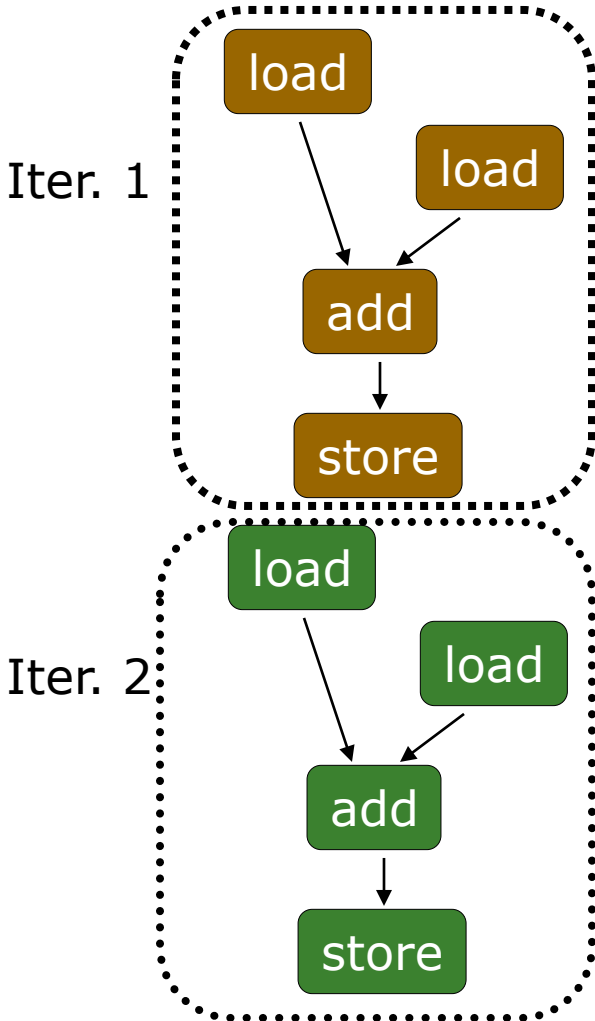
---

- 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:

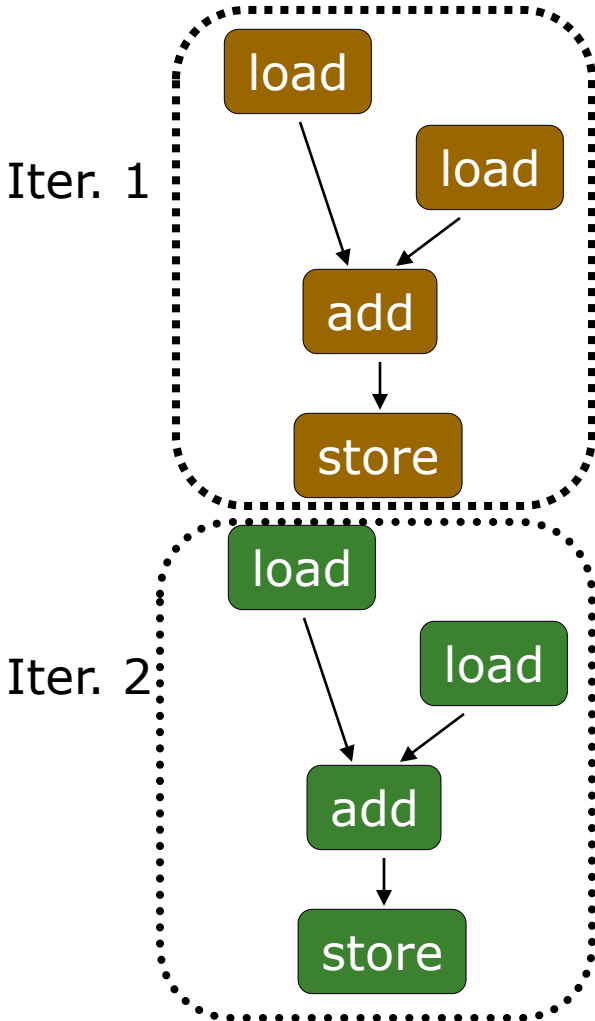
1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)



# Prog. Model 1: Sequential (SISD)

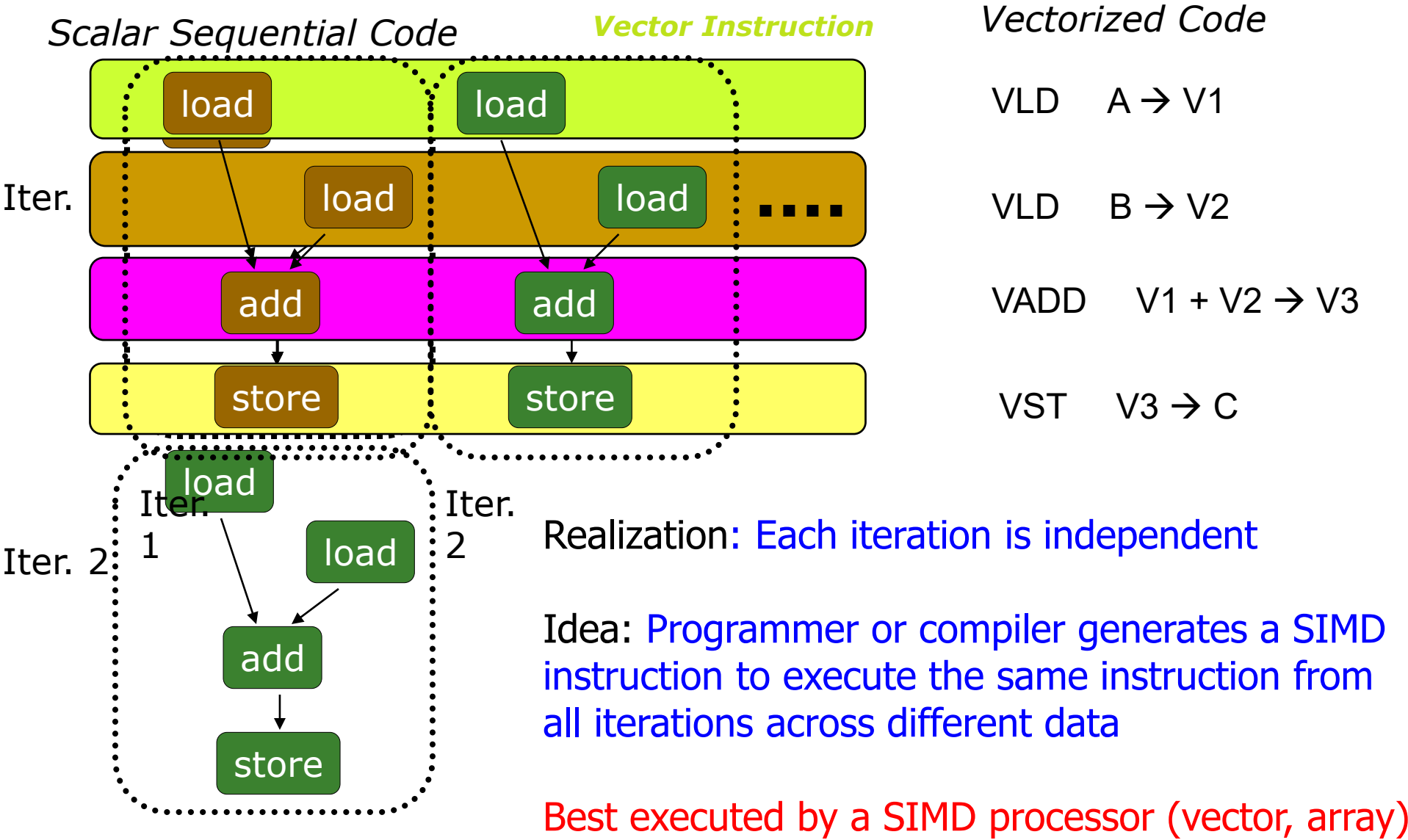
```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```

## Scalar Sequential Code



- 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

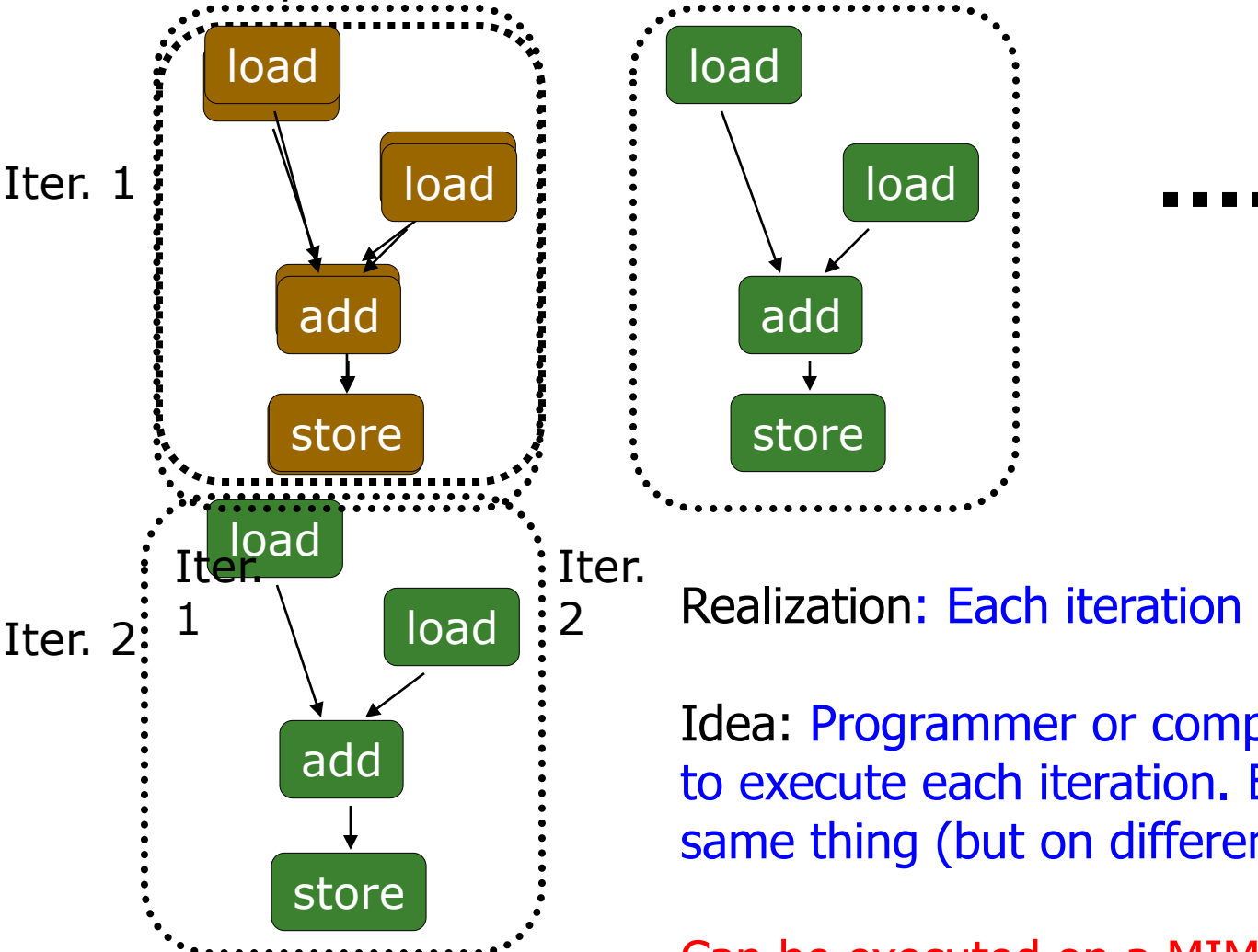
# Prog. Model 2: Data Parallel (SIMD) `for (i=0; i < N; i++) C[i] = A[i] + B[i];`



# Prog. Model 3: Multithreaded

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

## Scalar Sequential Code



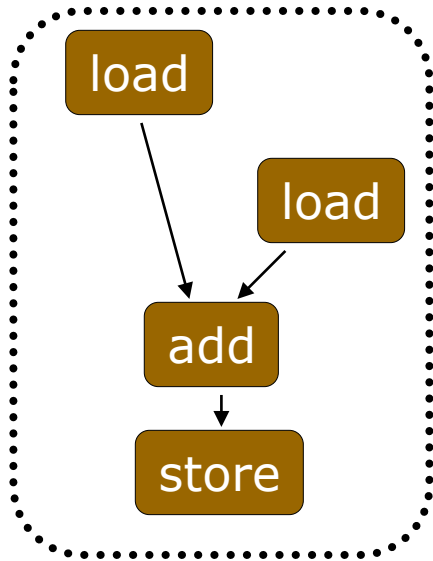
Realization: Each iteration is independent

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

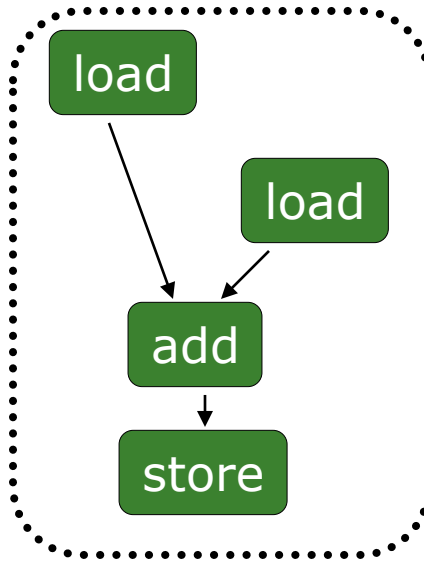
Can be executed on a MIMD machine

# 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**

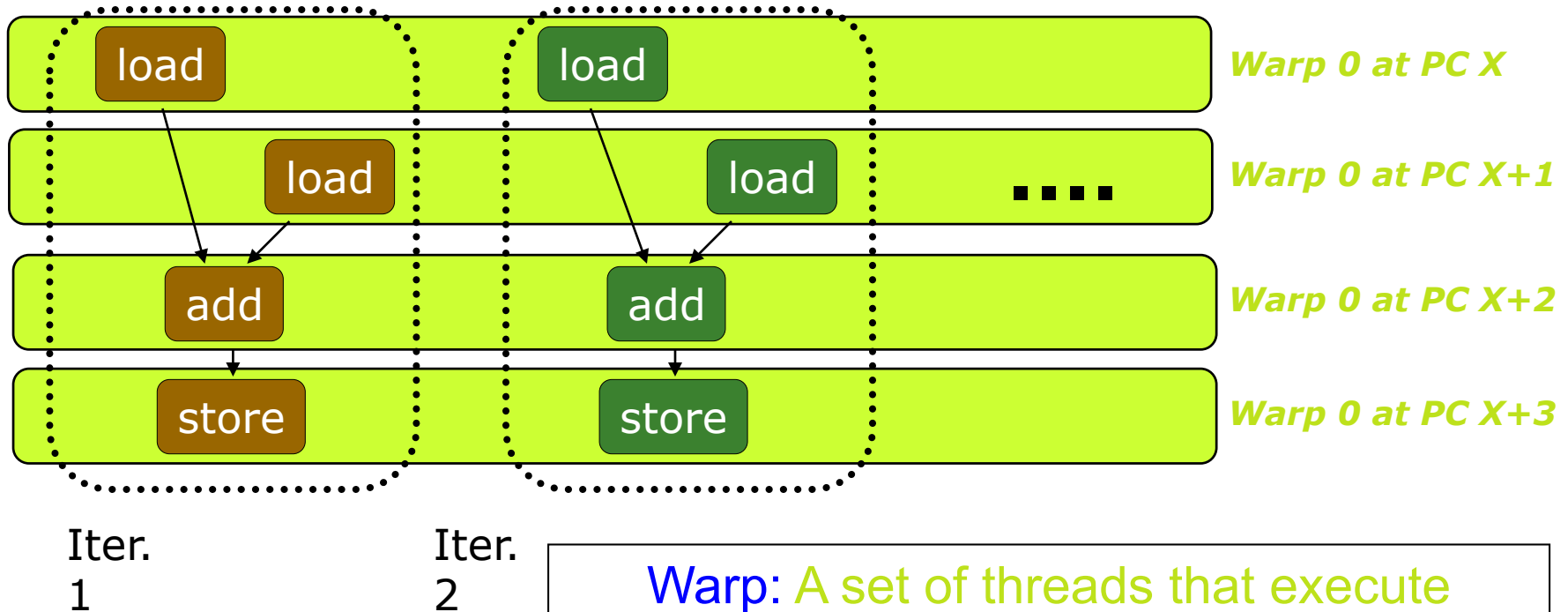
# A GPU is a SIMD (SIMT) Machine

---

- 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

---

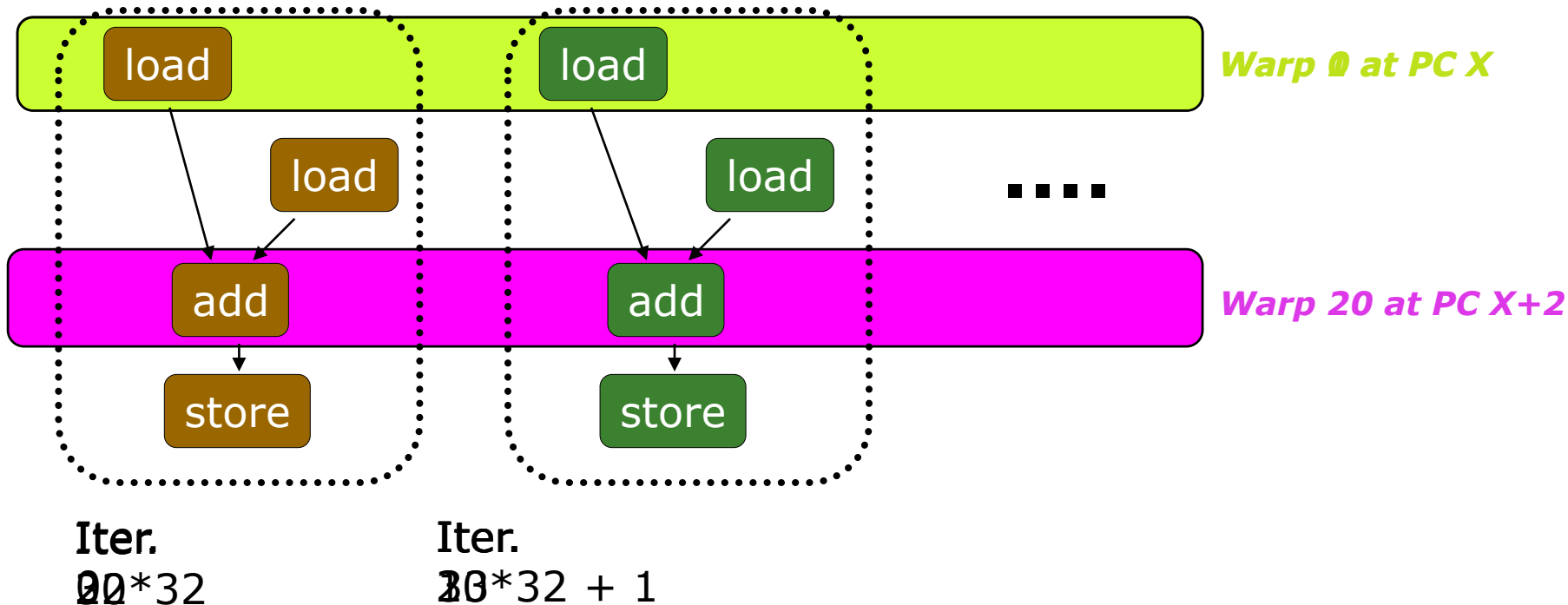
- SIMD: A single **sequential instruction stream** of **SIMD instructions** → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN
- SIMT: **Multiple instruction streams** of **scalar instructions** → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads
- 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];
```

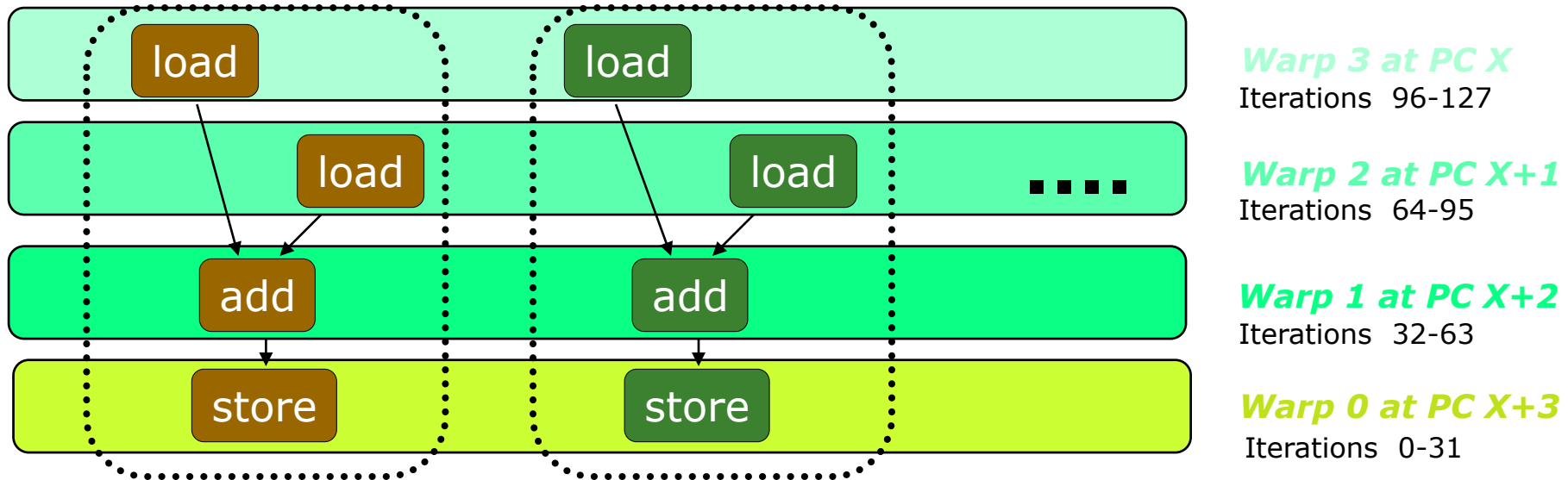
- Assume a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread  $\rightarrow$  1K warps
- Warps can be interleaved on the same pipeline  $\rightarrow$  Fine grained multithreading of warps



# 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



All threads in a warp are independent of each other  
→ They be executed seamlessly in a fine-grained multithreaded pipeline

# Lecture on Fine-Grained Multithreading

## 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

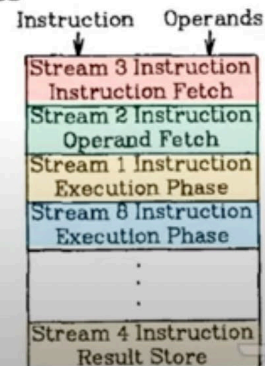
+ No logic needed for handling control and data dependences within a thread

+ High thread-level throughput

-- Single thread performance suffers

-- 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

Digital Design & Computer Architecture - Lecture 14: Pipelined Processor Design (Spring 2022)

1,066 views · Streamed live on Apr 8, 2022

👍 51 🗑 DISLIKE ➦ SHARE ✂ CLIP ⚙ SAVE ...



**Onur Mutlu Lectures**  
24.5K subscribers

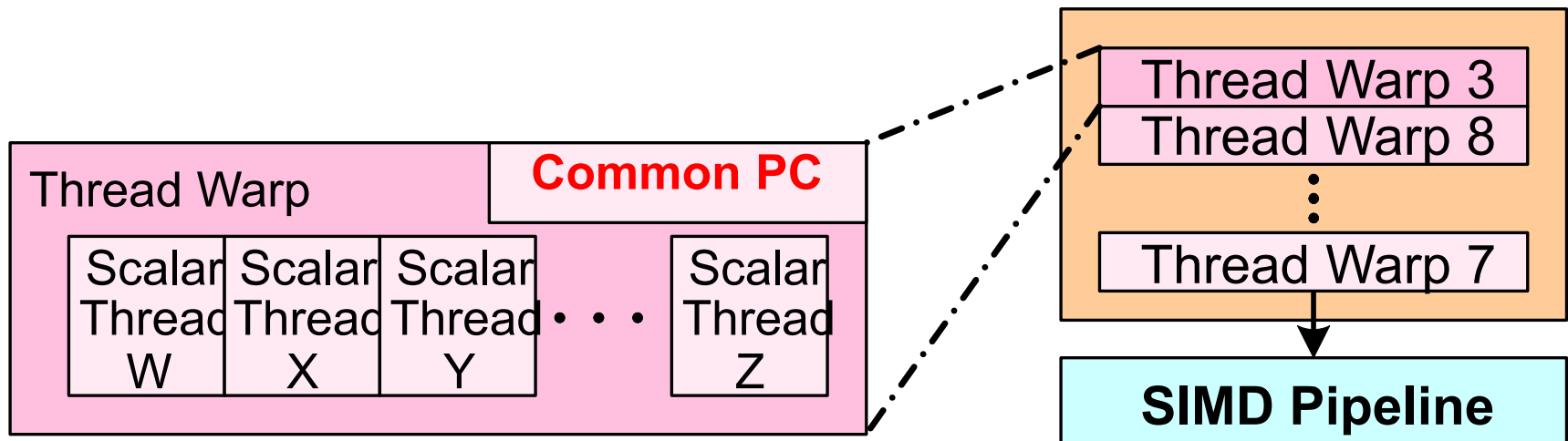
SUBSCRIBED 🔔

Digital Design and Computer Architecture, ETH Zürich, Spring 2022 (  
<https://safari.ethz.ch/digitaltechnik...>)

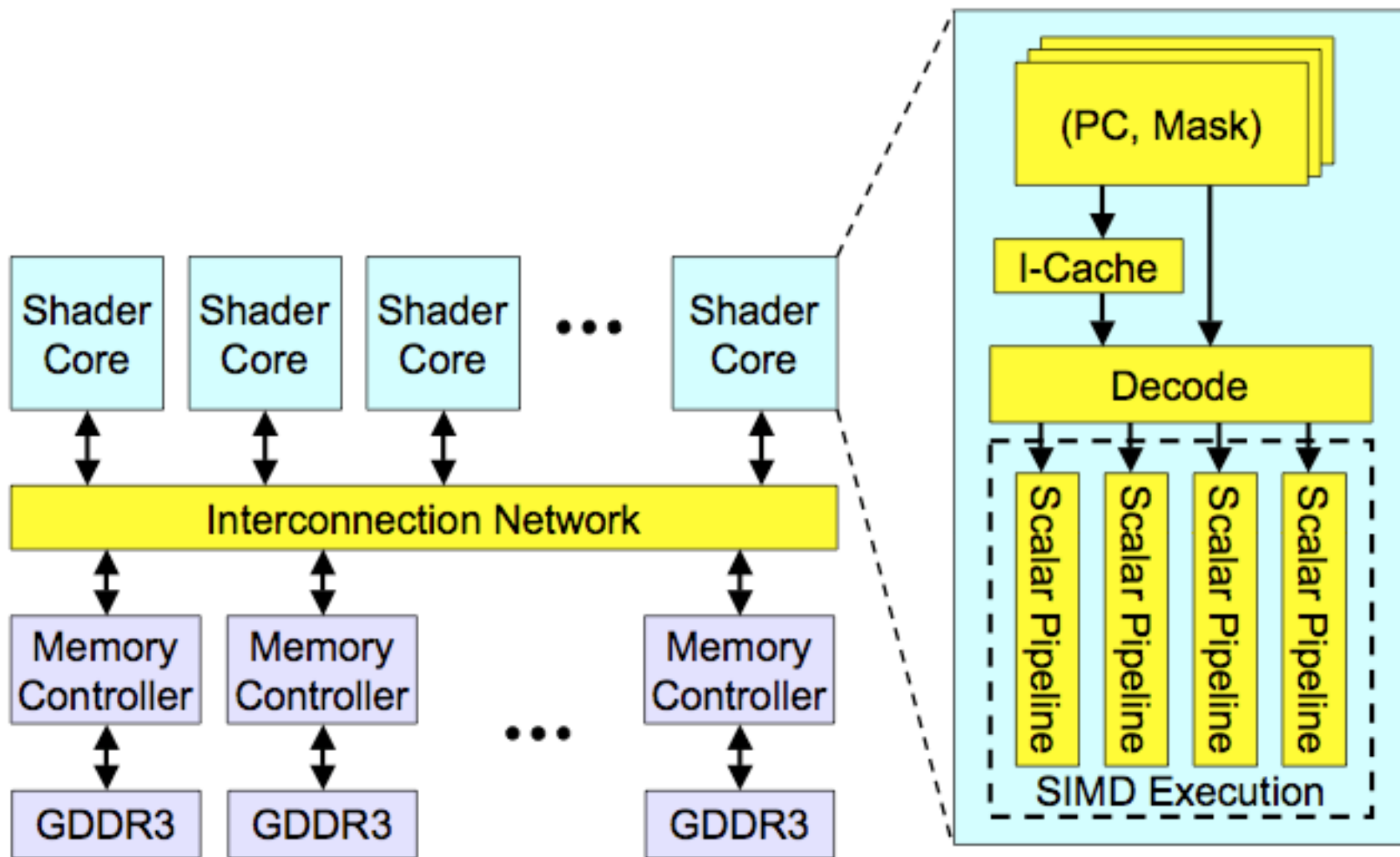
Lecture 14: Pipelined Processor Design  
Lecturer: Professor Onur Mutlu (<https://people.inf.ethz.ch/omutlu/>)  
Date: April 8, 2022

# 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 ...

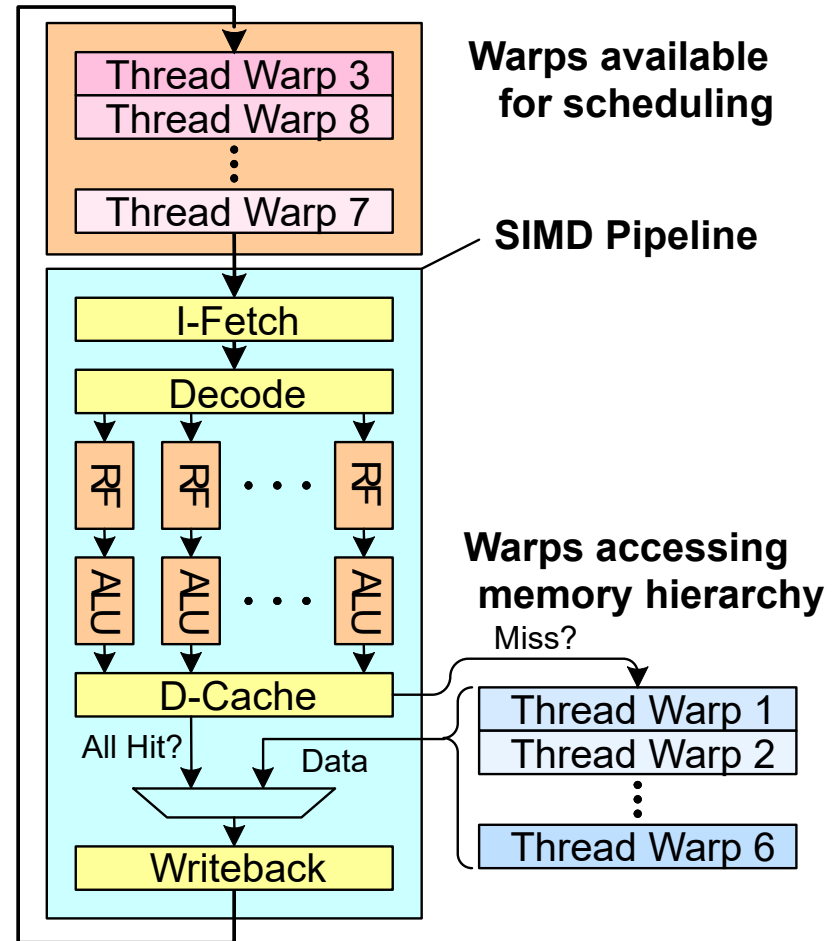


# High-Level View of a GPU



# 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 simple pipeline & long latency tolerance
  - Millions of threads operating on the same large image/video



# Recall: Vector Instruction Execution

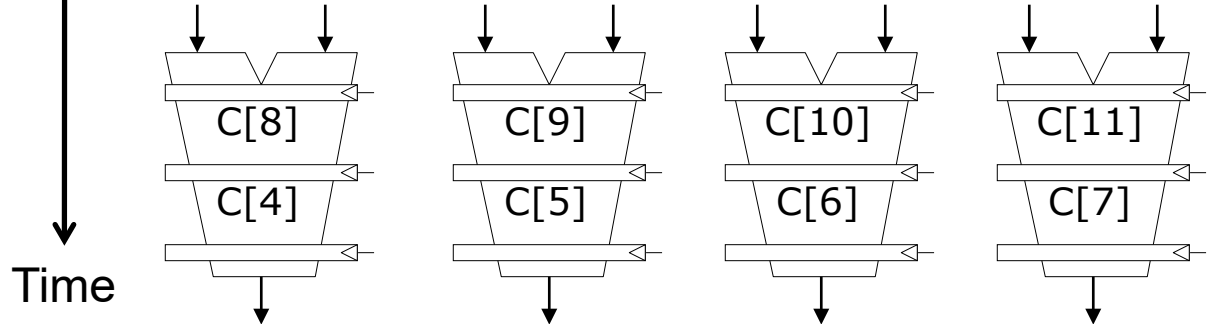
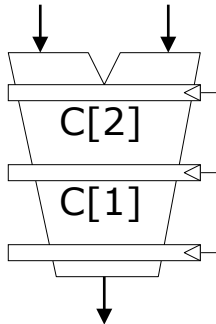
**VADD A,B → C**

*Execution using  
one pipelined  
functional unit*

*Execution using  
four pipelined  
functional units*

A[6] B[6]  
A[5] B[5]  
A[4] B[4]  
A[3] B[3]

A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]  
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]  
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]  
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]



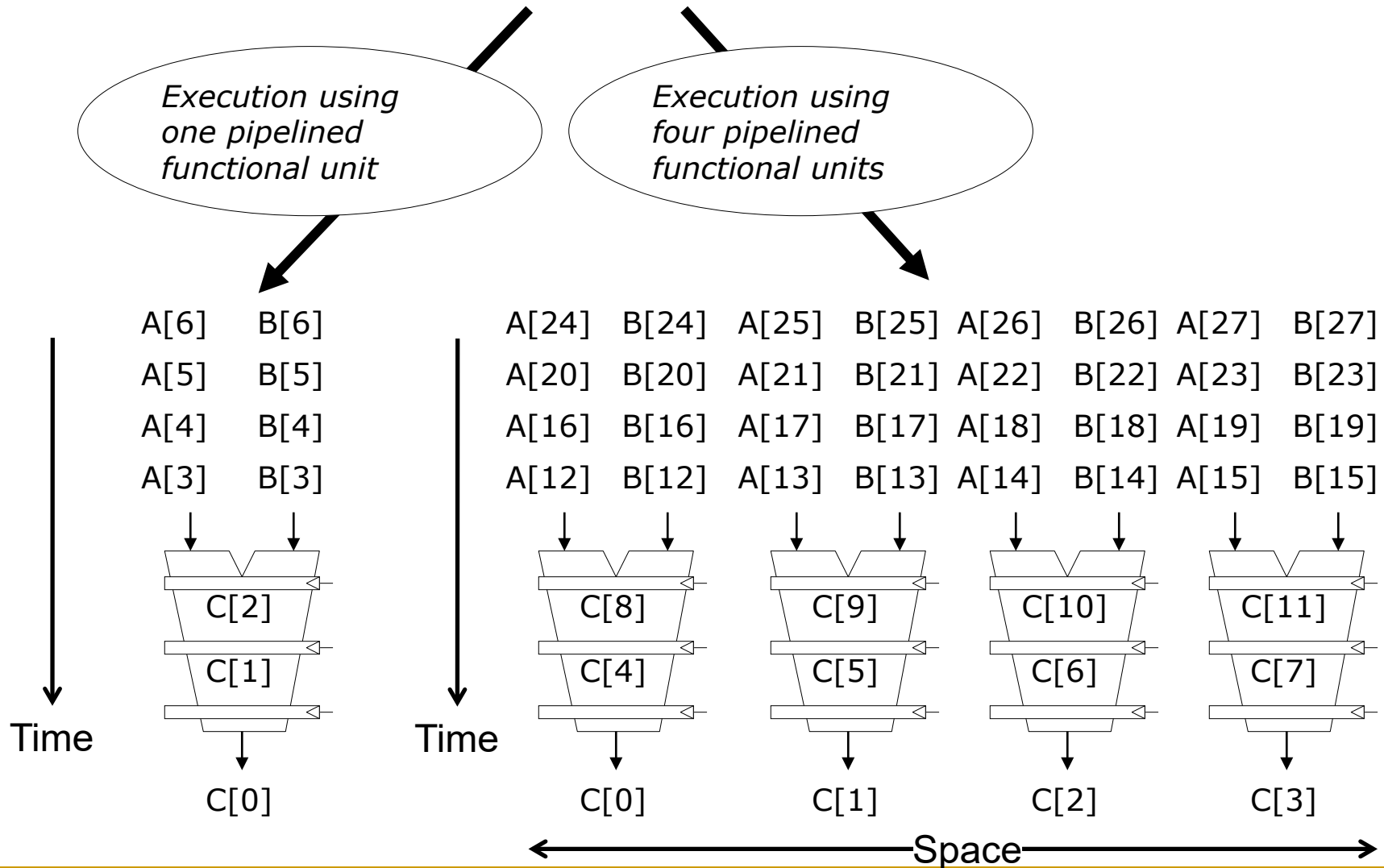
Time

Time

Space

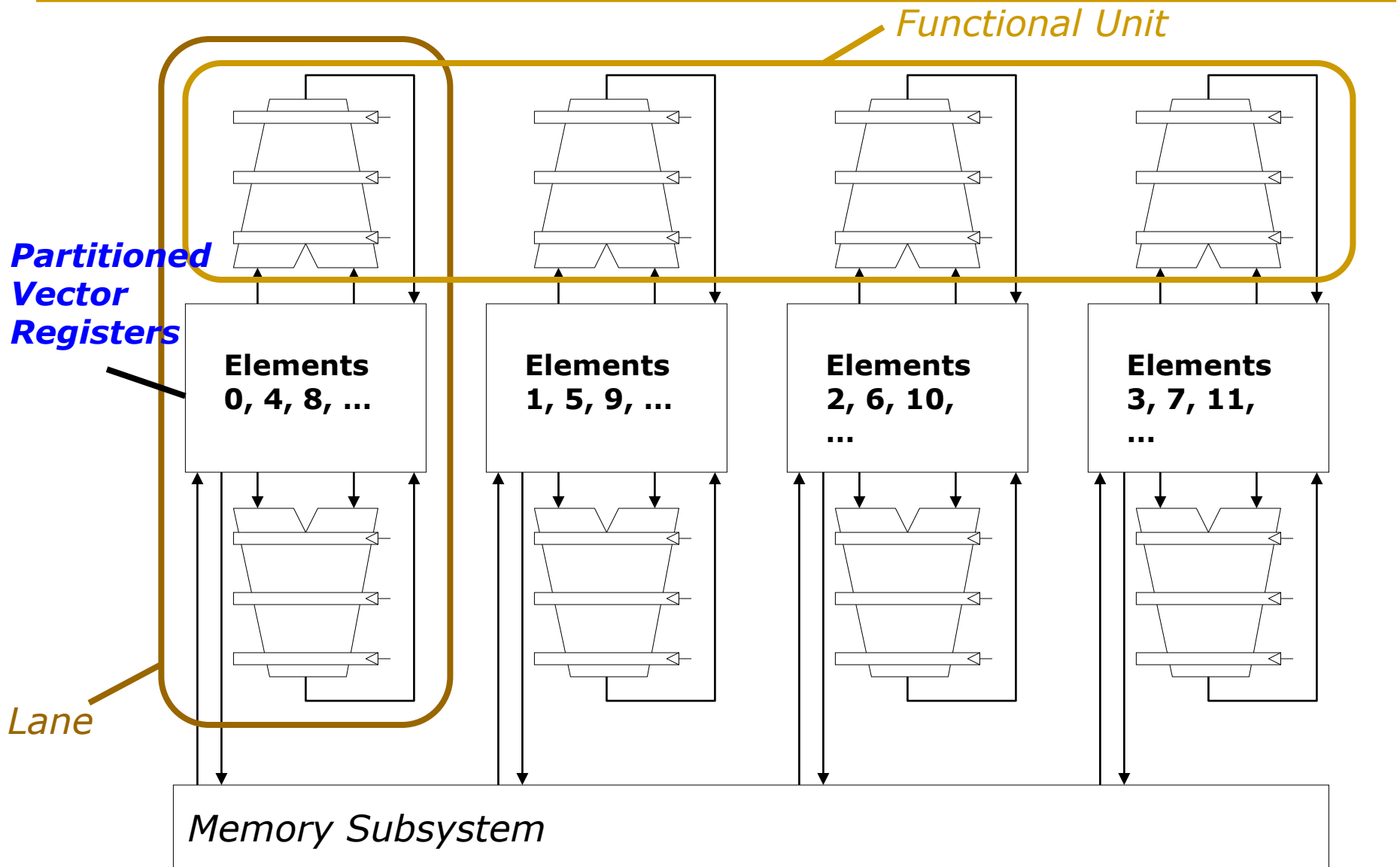
# Warp Execution (Recall the Previous Slide)

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

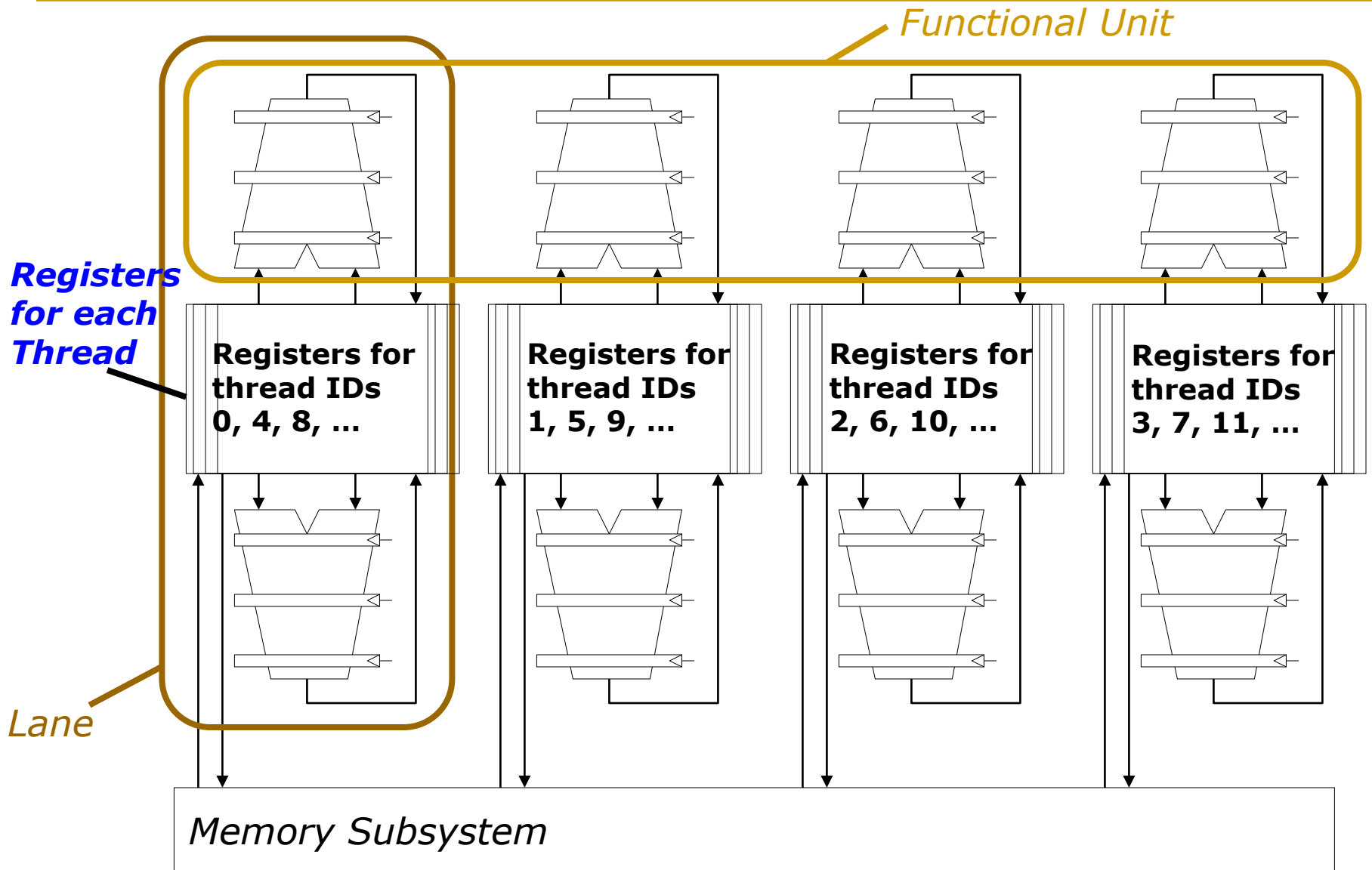




# Recall: Vector Unit Structure



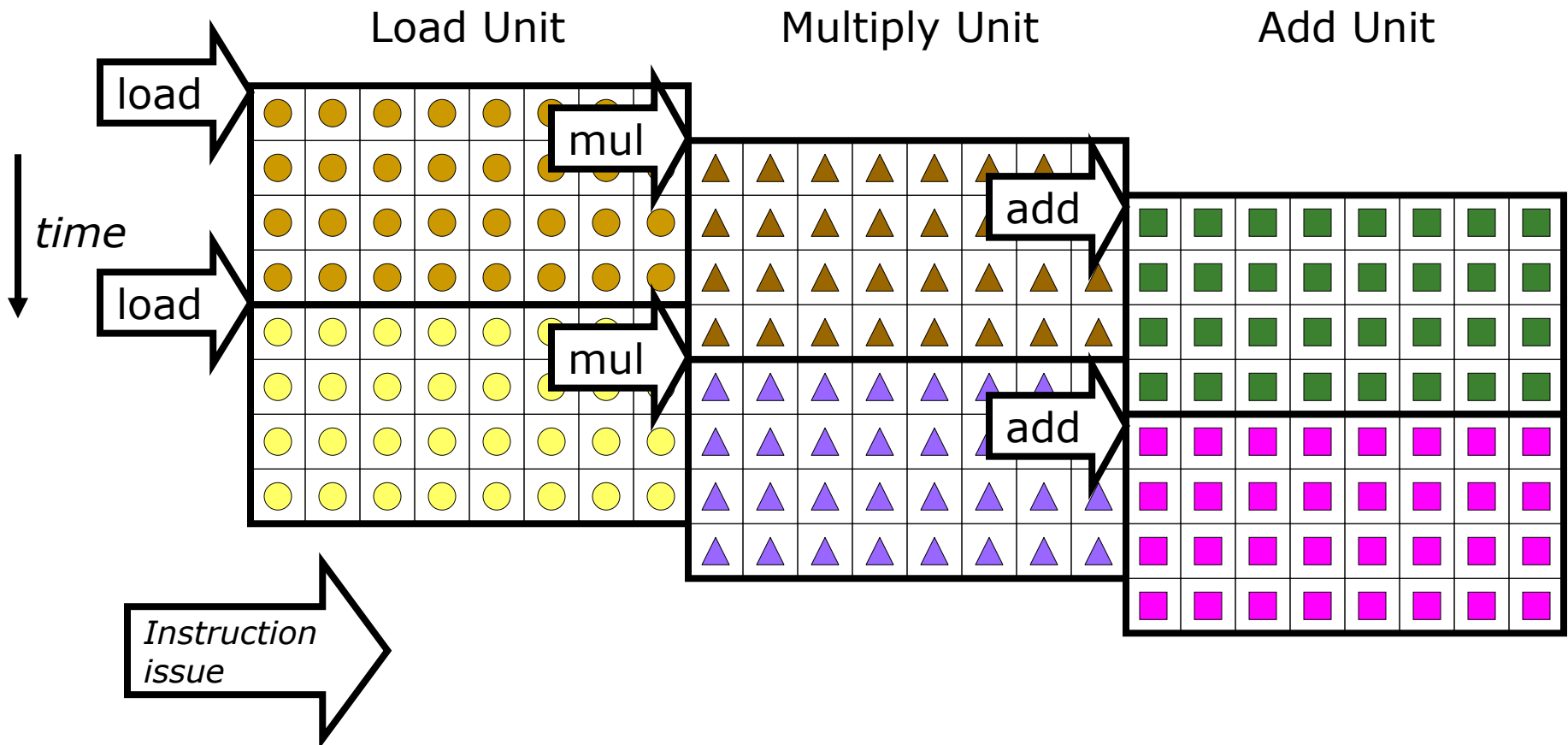
# GPU SIMD Execution Unit Structure



# Recall: Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

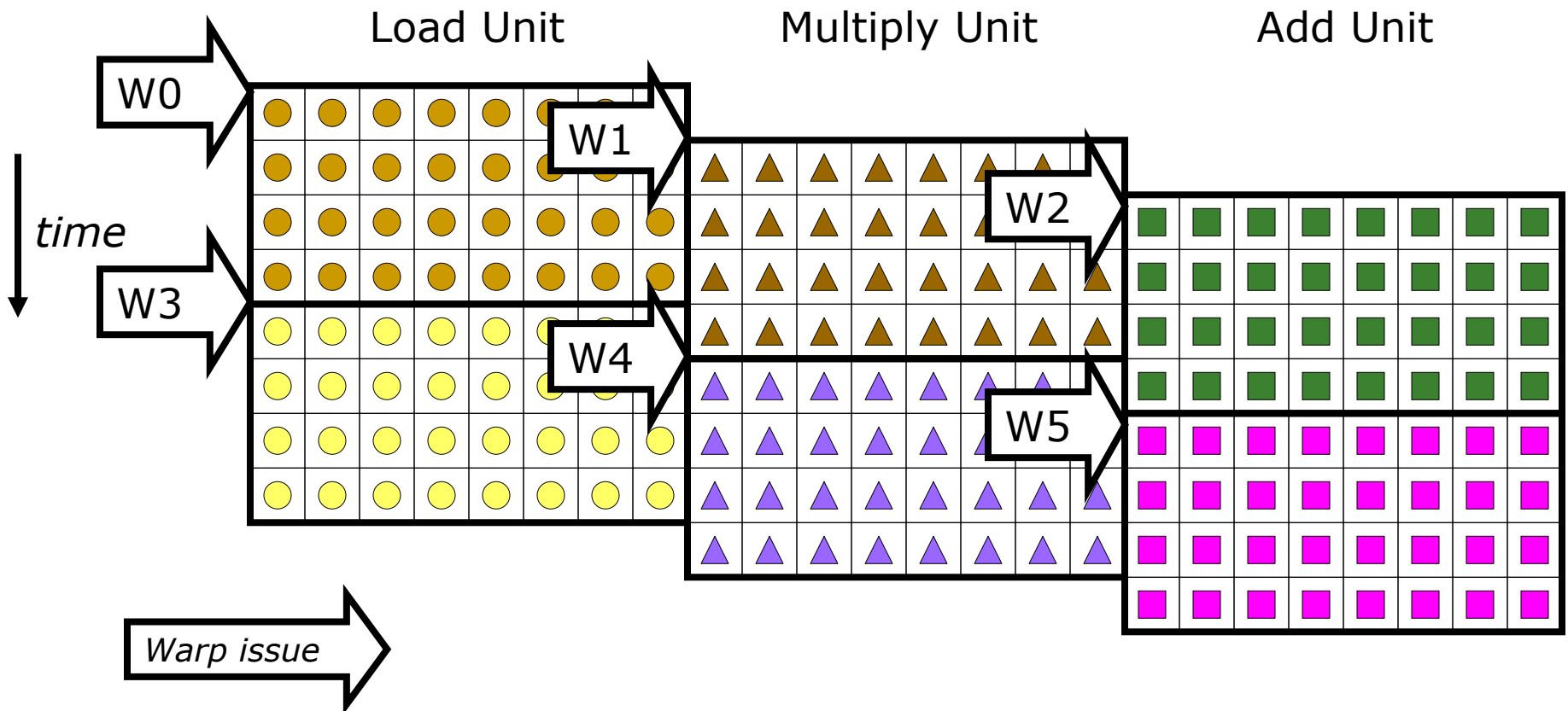
- Example machine has 32 elements per vector register and 8 lanes
- Example with 24 operations/cycle (steady state) while issuing 1 vector instruction/cycle



# Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

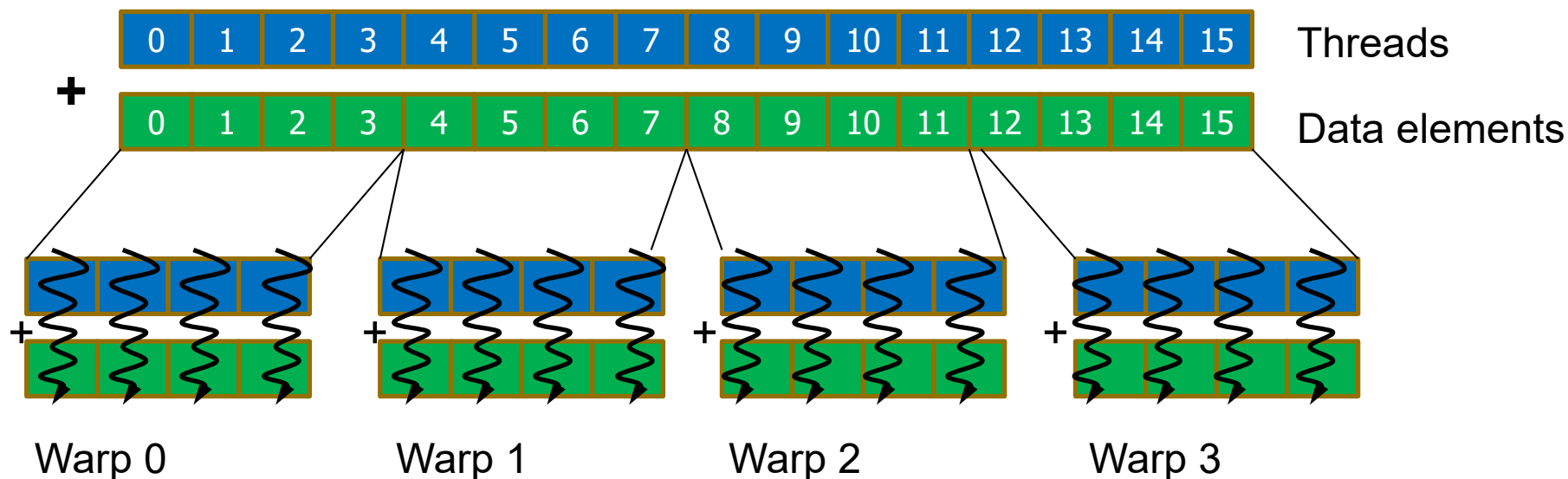
- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle (steady state) while issuing 1 warp/cycle



# SIMT Memory Access (Loads and Stores)

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

Let's assume  $N=16$ , 4 threads per warp  $\rightarrow$  4 warps



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

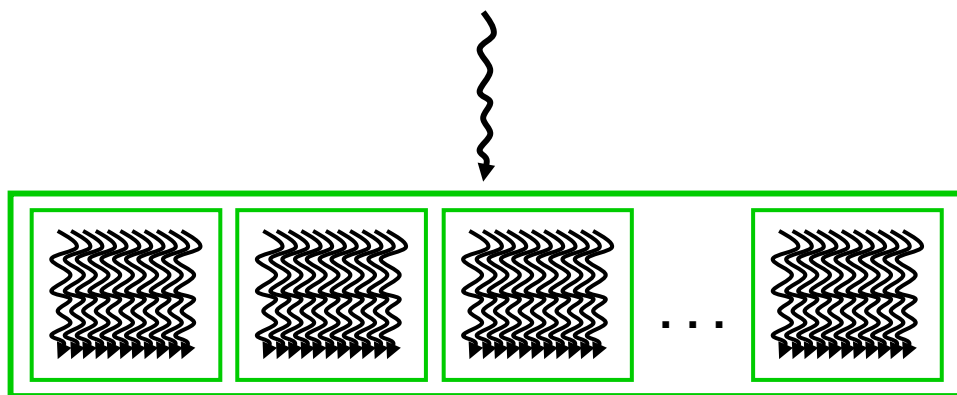
# Warps *not* Exposed to GPU Programmers

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

Serial Code (host)

Parallel Kernel (device)

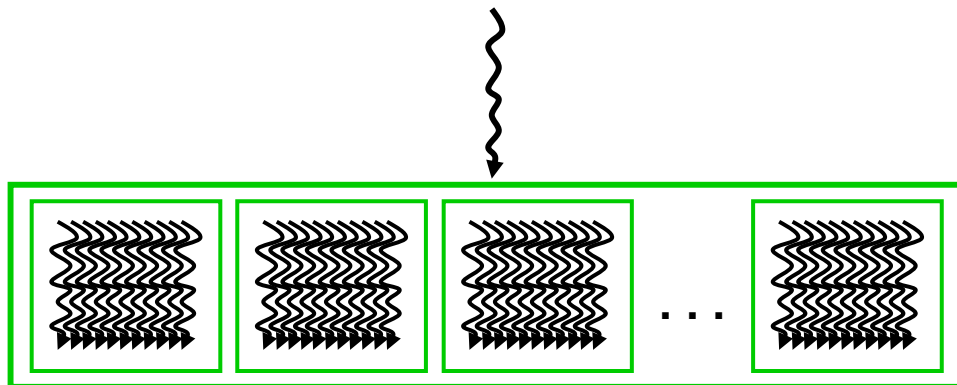
```
KernelA<<<nBlk, nThr>>>(args);
```



Serial Code (host)

Parallel Kernel (device)

```
KernelB<<<nBlk, nThr>>>(args);
```



# Sample GPU SIMT Code (Simplified)

---

CPU code

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



CUDA code

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

# Sample GPU Program (Less Simplified)

## CPU Program

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

int main () {

    add_matrix (a, b, c, N);
}
```

## GPU Program

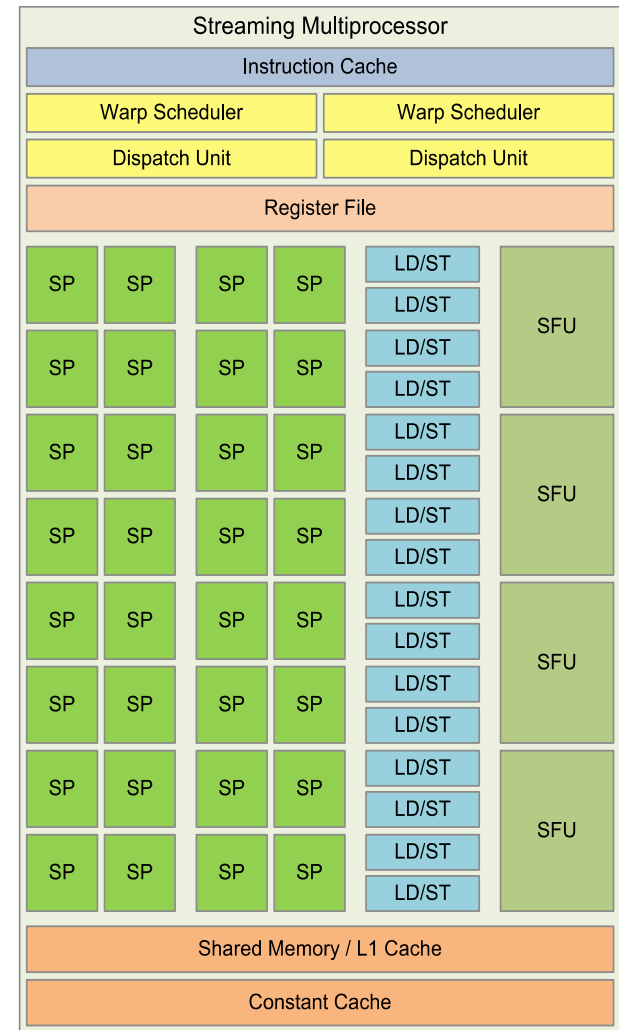
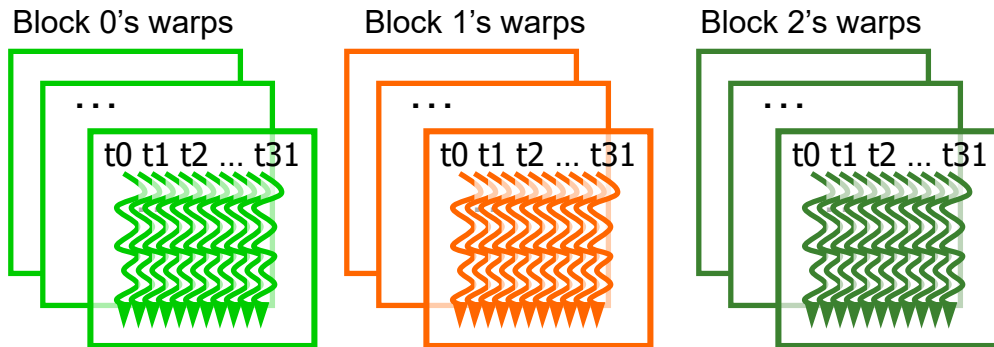
```
__global__ add_matrix
( float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index]+b[index];
}

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



# From Blocks to Warps

- GPU core: A SIMD pipeline
  - Streaming Processor (SP)
  - Many such SIMD Processors
    - Streaming Multiprocessor (SM)
- Blocks are divided into **warps**
  - SIMD/SIMT unit (32 threads)



NVIDIA Fermi architecture

# Warp-based SIMD vs. Traditional SIMD

---

- **Traditional SIMD** contains a **single thread**
  - **Sequential instruction execution**; lock-step operations in a SIMD instruction
  - **Programming model is SIMD** (no extra threads) → SW needs to know vector length
  - ISA contains **vector/SIMD instructions**
- **Warp-based SIMD** consists of **multiple scalar threads** executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - **Each thread can be treated individually** (i.e., placed in a different warp)  
→ **programming model not SIMD**
    - SW does **not need to know vector length**
    - Enables multithreading and flexible dynamic grouping of threads
  - **ISA is scalar** → SIMD operations can be formed dynamically
  - Essentially, it is **SPMD programming model implemented on SIMD hardware**

# SPMD

---

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

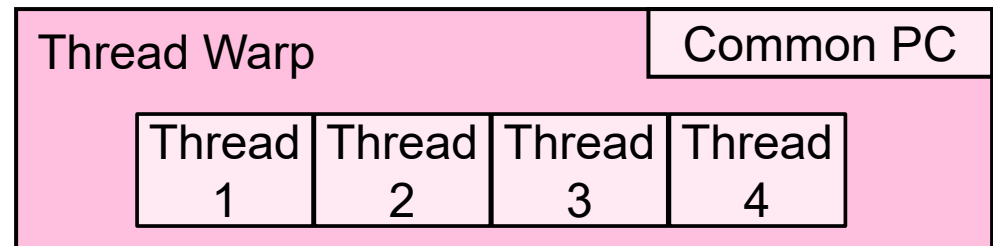
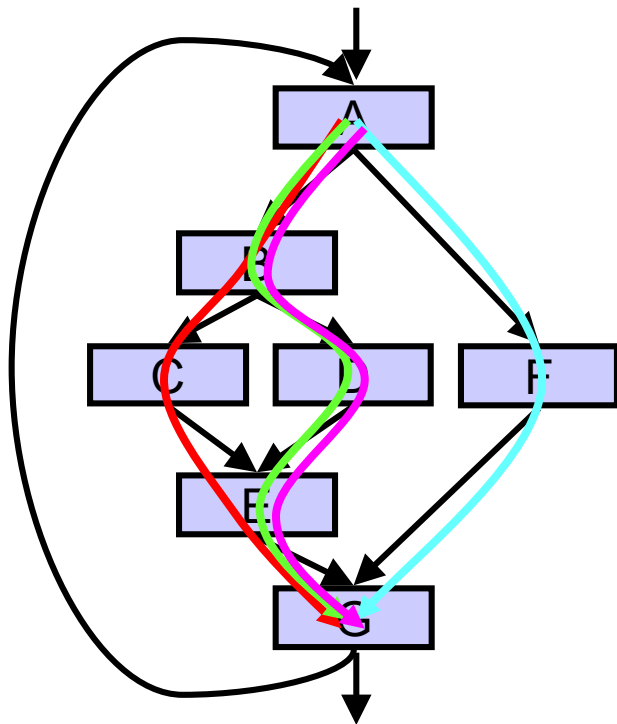
# SIMD vs. SIMT Execution Model

---

- SIMD: A single **sequential instruction stream** of **SIMD instructions** → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN
- SIMT: **Multiple instruction streams** of **scalar instructions** → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads
- Two Major SIMT Advantages:
  - **Can treat each thread separately** → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - **Can group threads into warps flexibly** → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

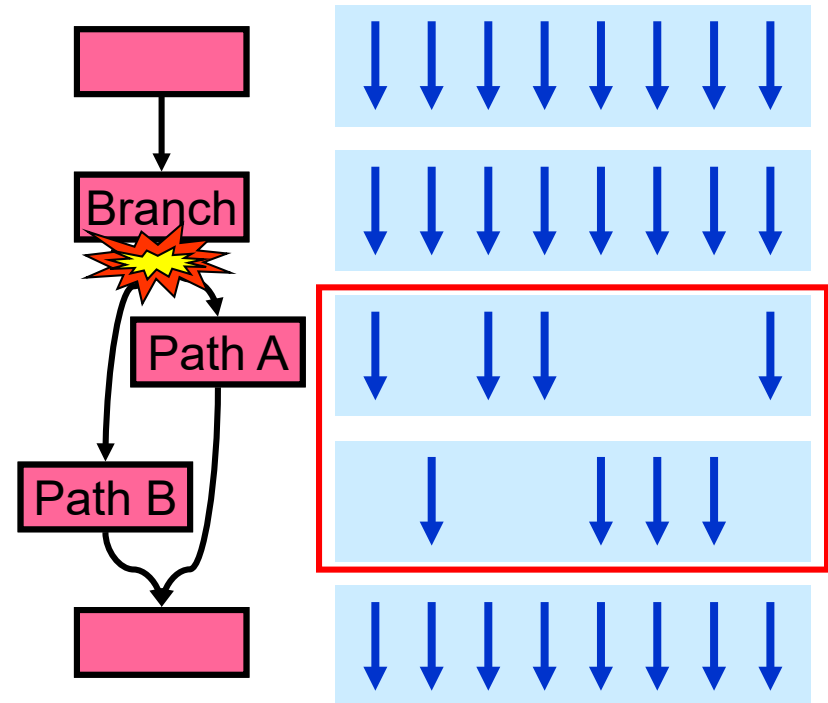
# Threads Can Take Different Paths in Warp-based SIMD

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



# 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**

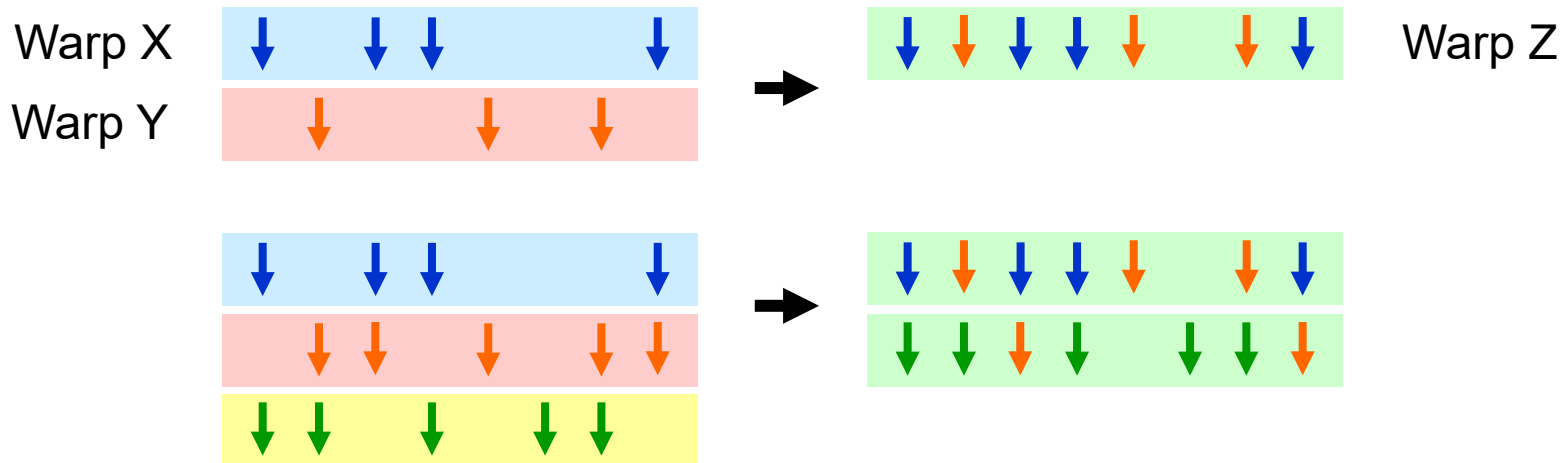
# Remember: Each Thread Is Independent

---

- 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

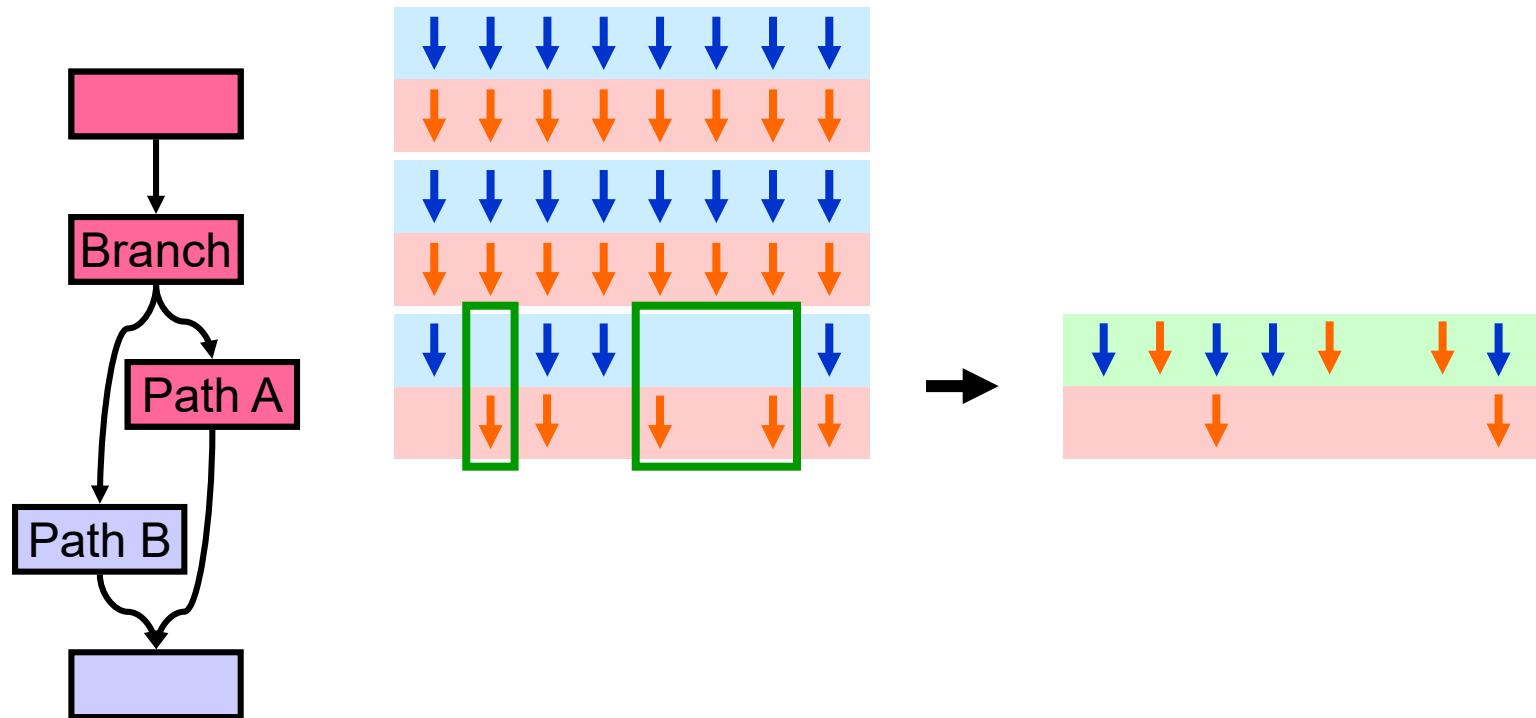
- Idea: Dynamically merge threads executing the same instruction, i.e., at the same PC (after branch divergence)
- Form new warps from warps that are waiting
  - Enough threads branching to each path enables the creation of full new warps





# Dynamic Warp Formation/Merging

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



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

# An Example GPU

# NVIDIA GeForce GTX 285

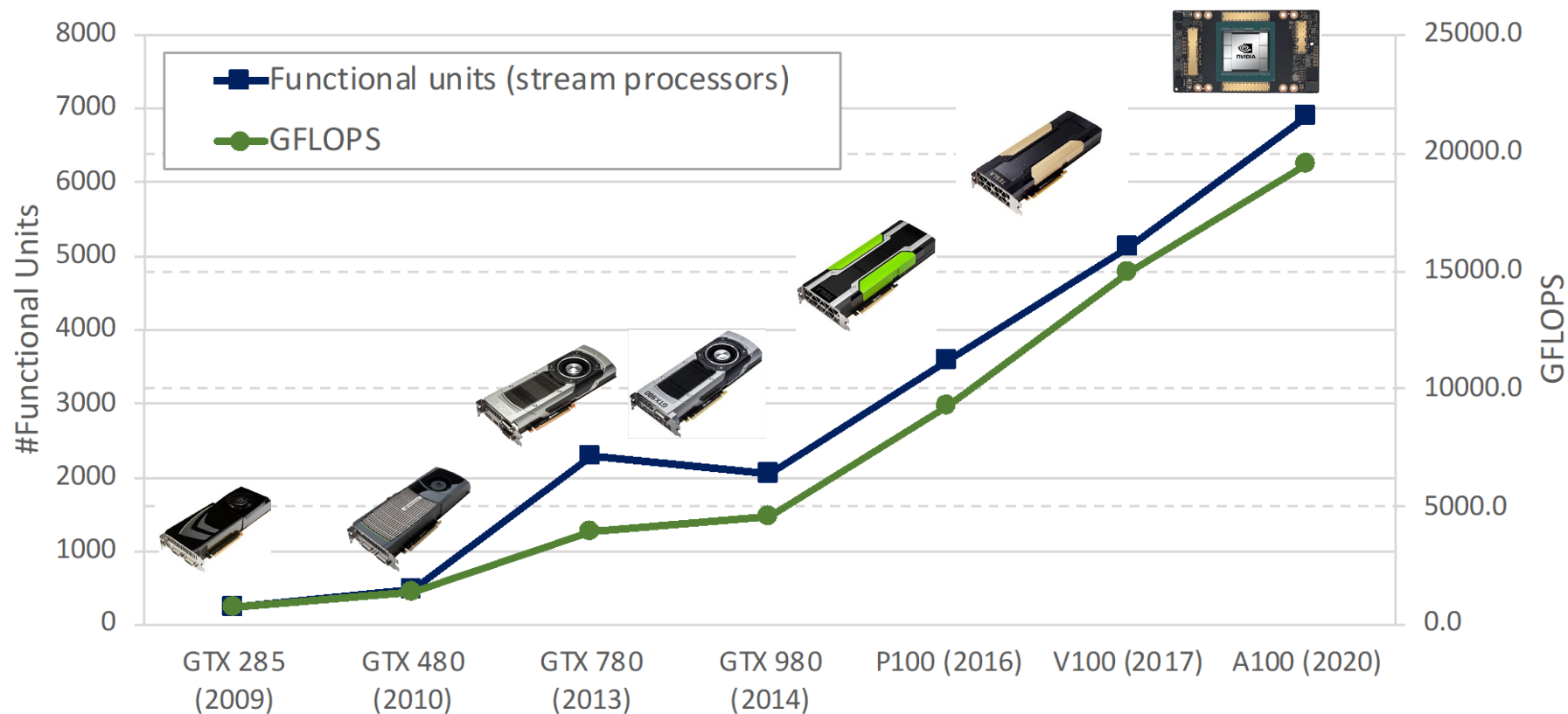
---

- 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.

# Evolution of NVIDIA GPUs



# NVIDIA V100

---

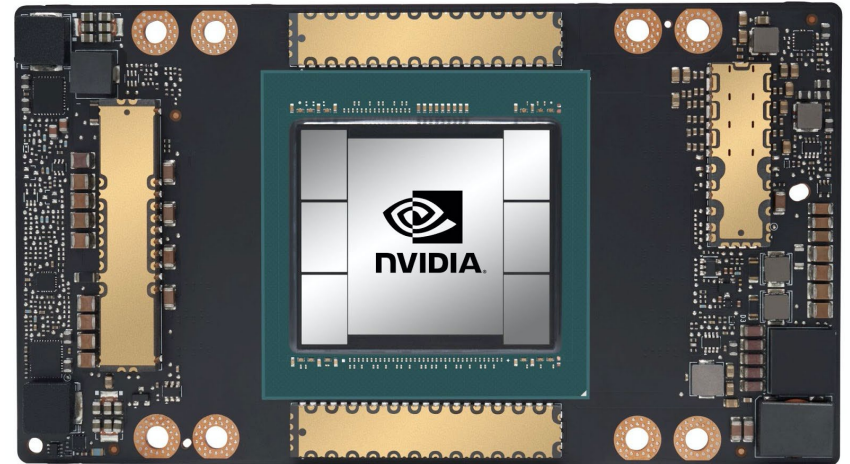
- NVIDIA-speak:
  - 5120 stream processors
  - “SIMT execution”
  
- Generic speak:
  - 80 cores
  - 64 SIMD functional units per core
  
  - Tensor cores for Machine Learning
  
- NVIDIA, [“NVIDIA Tesla V100 GPU Architecture. White Paper,”](#) 2017.



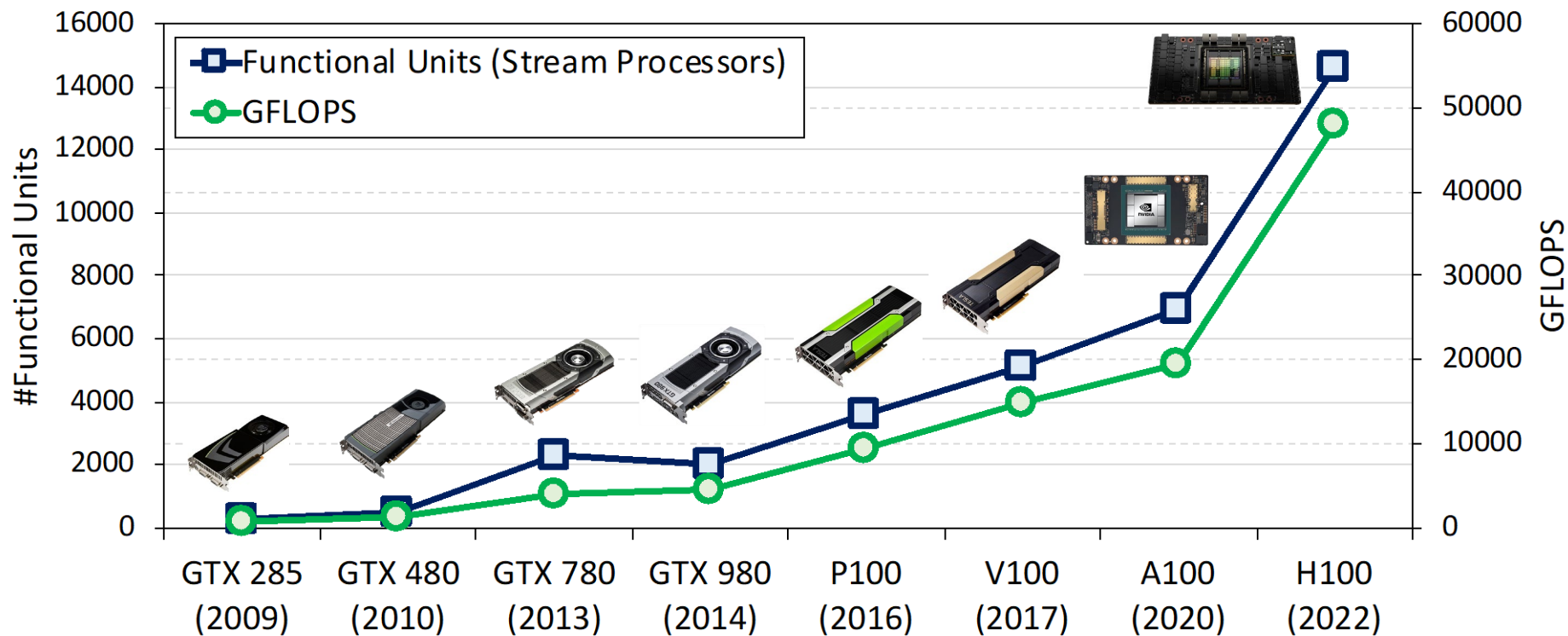
# NVIDIA A100

---

- 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)



# Evolution of NVIDIA GPUs (Updated)



# Clarification of Some GPU Terms

---

Generic Term	NVIDIA Term	AMD Term	Comments
Vector length	Warp size	Wavefront size	Number of threads that run in parallel (lock-step) on a SIMD functional unit
Pipelined functional unit / Scalar pipeline	Streaming processor / CUDA core	-	Functional unit that executes instructions for one GPU thread
SIMD functional unit / SIMD pipeline	Group of N streaming processors (e.g., N=8 in GTX 285, N=16 in Fermi)	Vector ALU	SIMD functional unit that executes instructions for an entire warp
GPU core	Streaming multiprocessor	Compute unit	It contains one or more warp schedulers and one or several SIMD pipelines