# GPU Architectures

Prof. Onur Mutlu

ETH Zürich

Spring 2023

11 May 2023

## GPUs are SIMD Engines Underneath

- The instruction pipeline operates like an SIMD pipeline (e.g., an array processor)

- However, the programming is done using threads, NOT SIMD instructions

- Let us 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), …

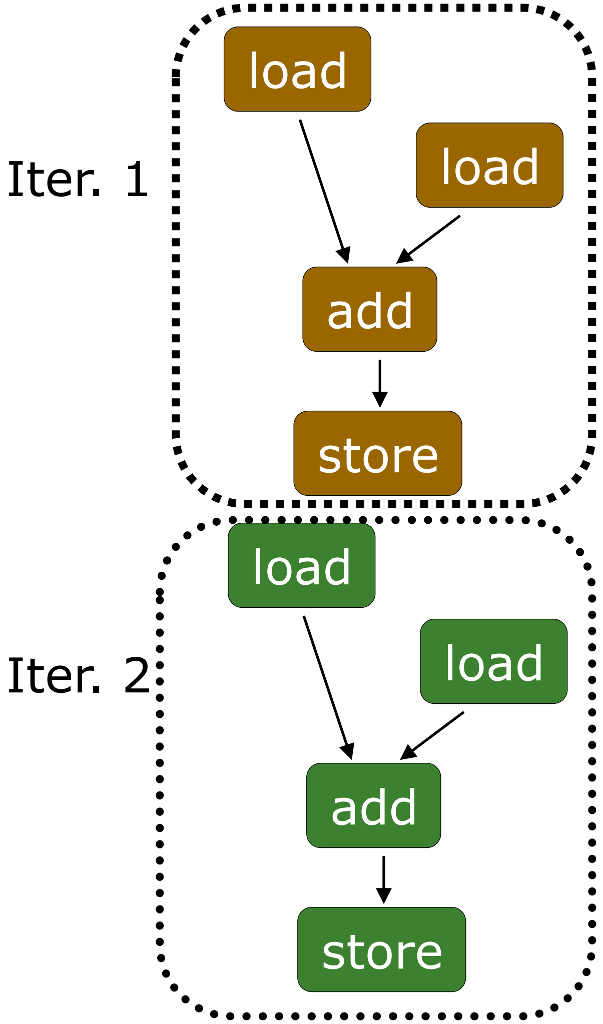
- Execution Model refers to how the hardware executes the code underneath
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, …

- 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?

*Scalar Sequential Code*

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



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

1. Sequential (SISD)

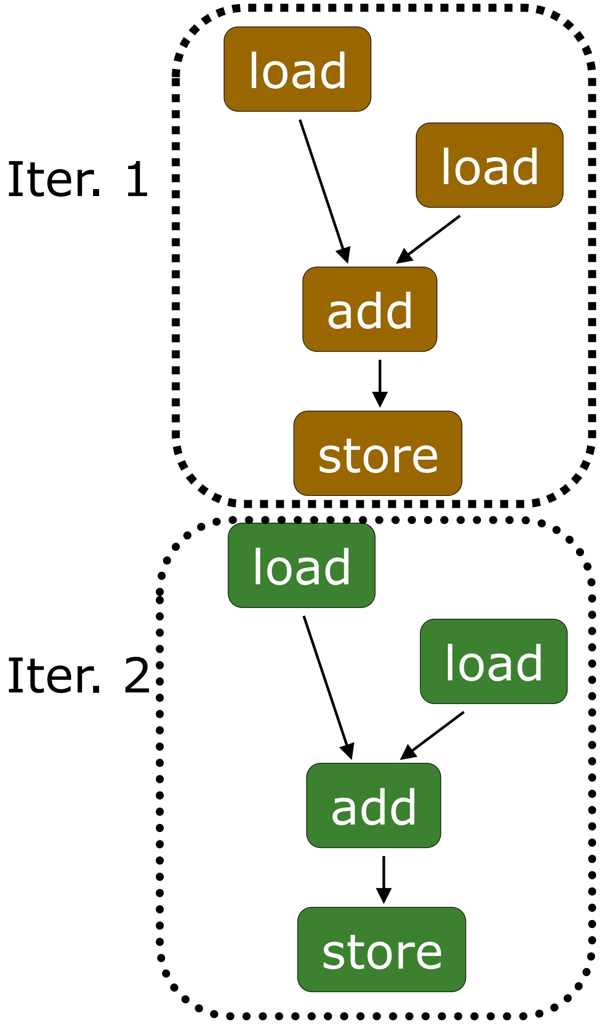
2. Data-Parallel (SIMD)

3. Multithreaded (MIMD/SPMD)

# Prog. Model 1: Sequential (SISD)

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

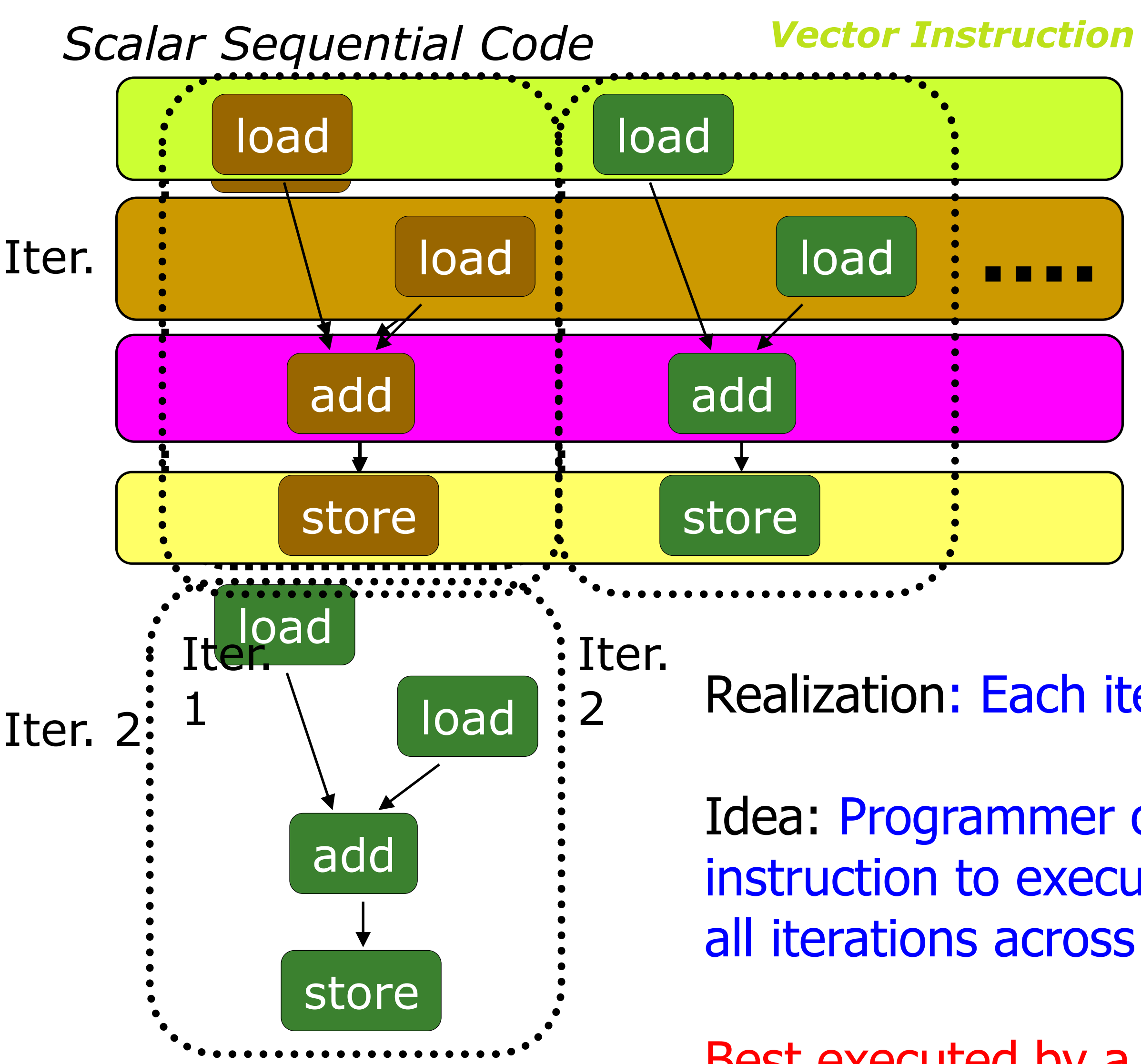
*Scalar Sequential Code*



- 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];
```



*Vectorized Code*

| | |
|---|---|
| VLD | A → V1 |
| VLD | B → V2 |
| VADD | V1 + V2 → V3 |
| VST | V3 → C |

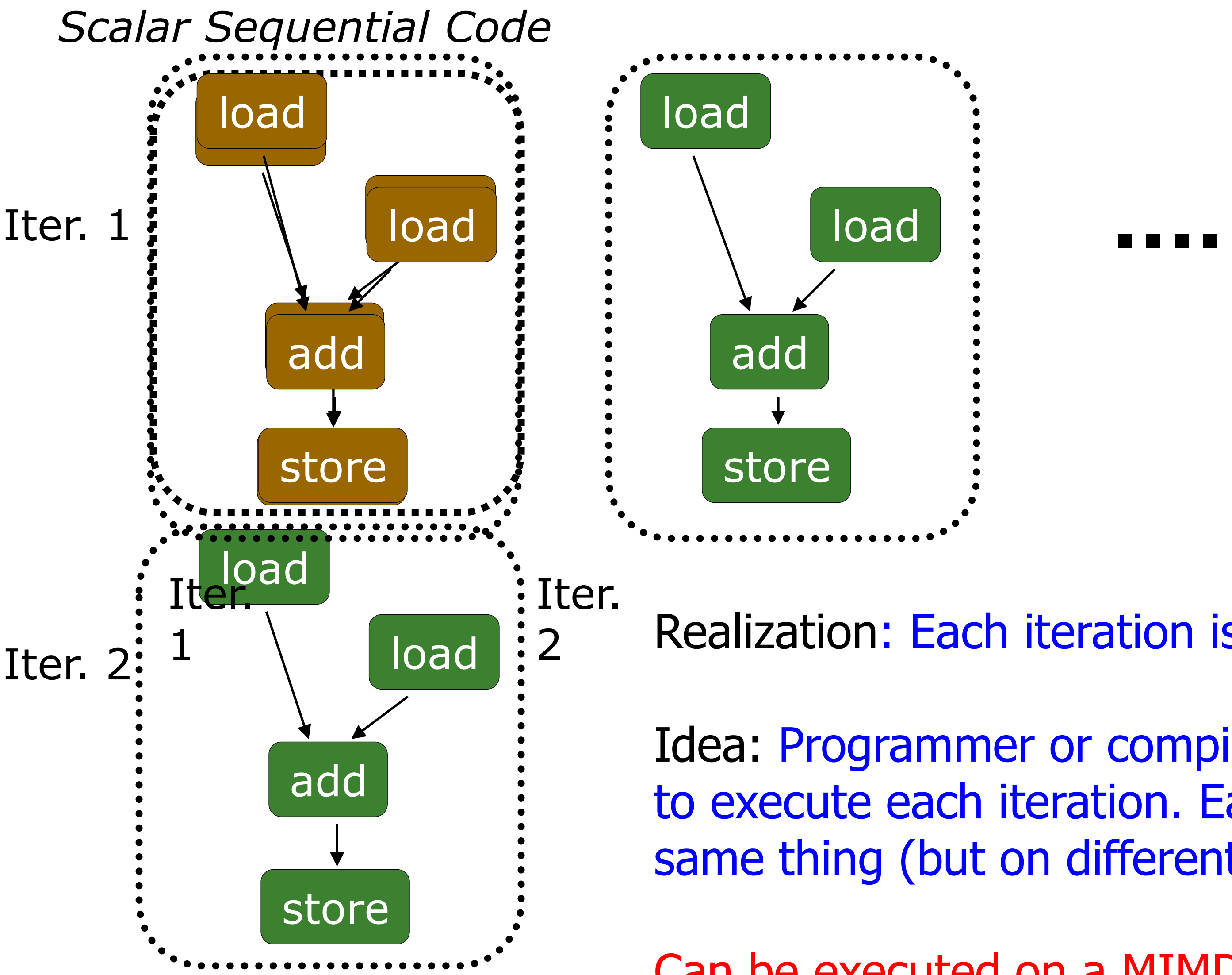
Realization: Each iteration is independent

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

Best executed by a SIMD processor (vector, array)

# Prog. Model 3: Multithreaded

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



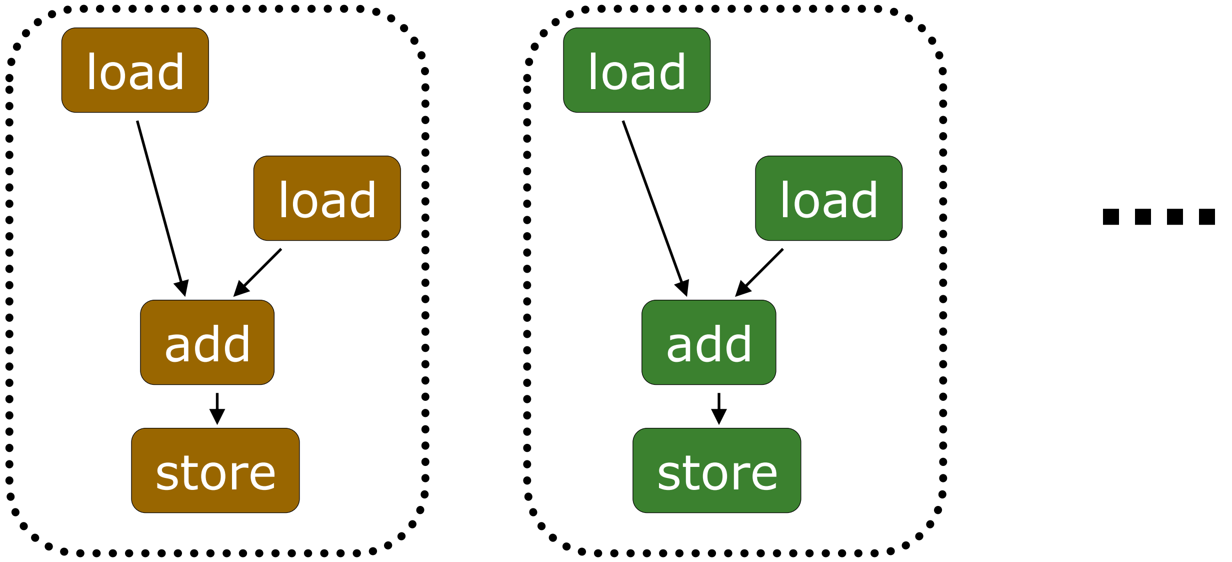
Realization: Each iteration is independent

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

Can be executed on a MIMD machine

# Prog. Model 3: Multithreaded

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



Iter. 1

Iter. 2

Realization: Each iteration is independent

This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread

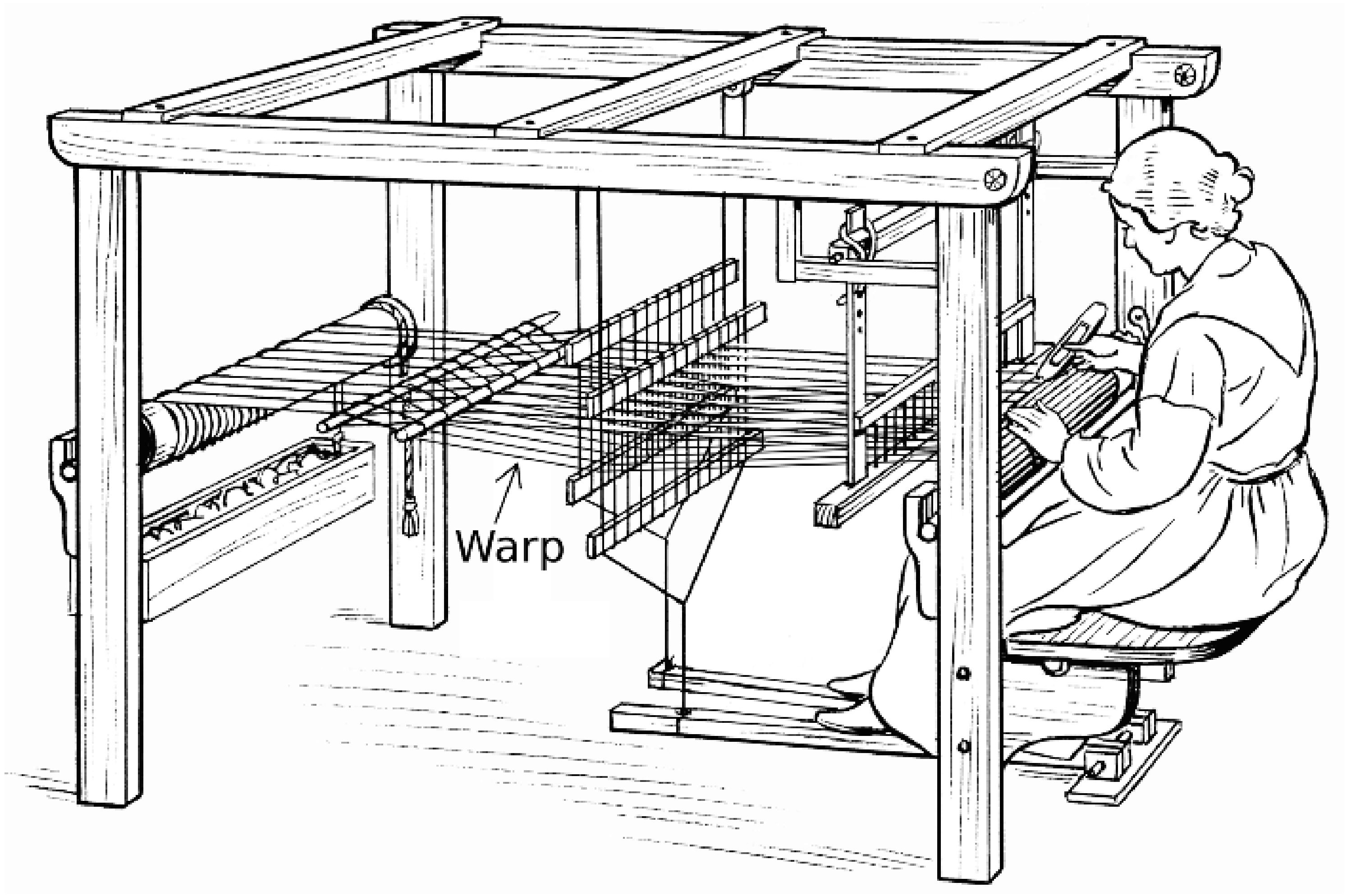
## A GPU is a SIMD (SIMT) Machine

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

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

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

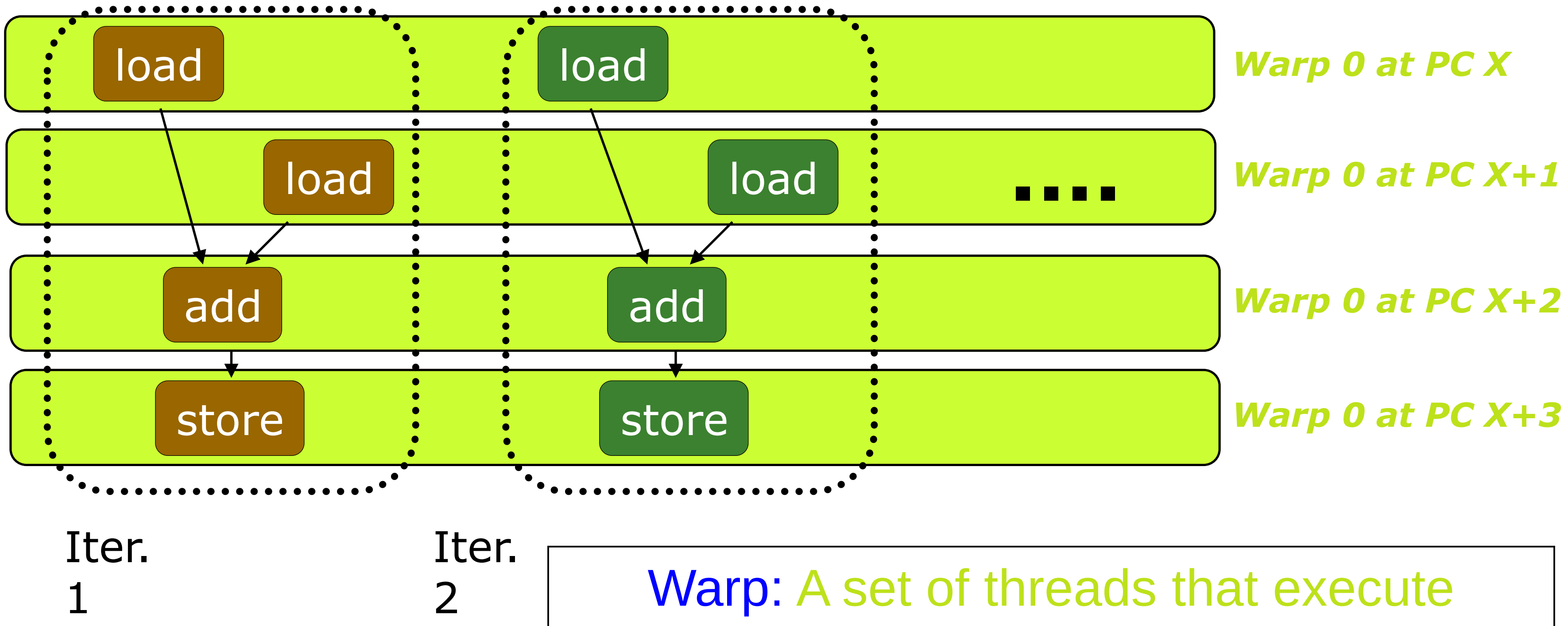
## Warp Terminology



Source: Wikipedia

# 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

## 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

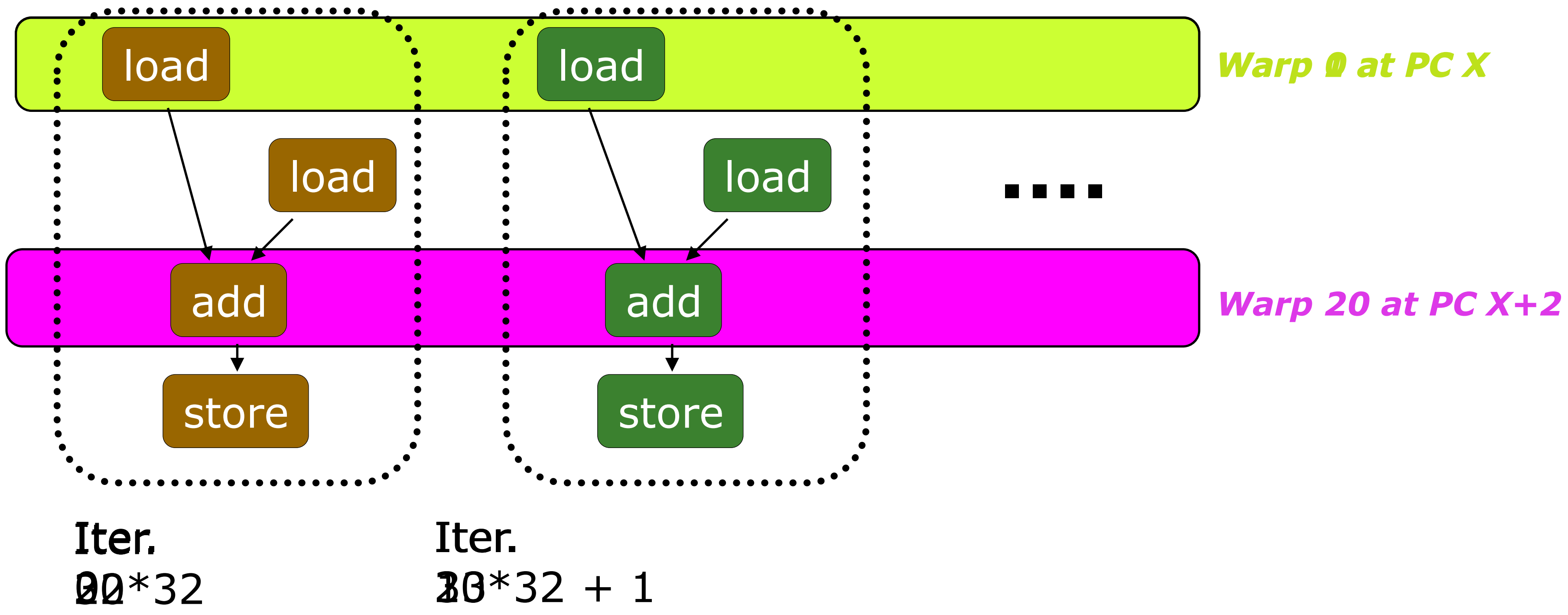
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- 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];
```

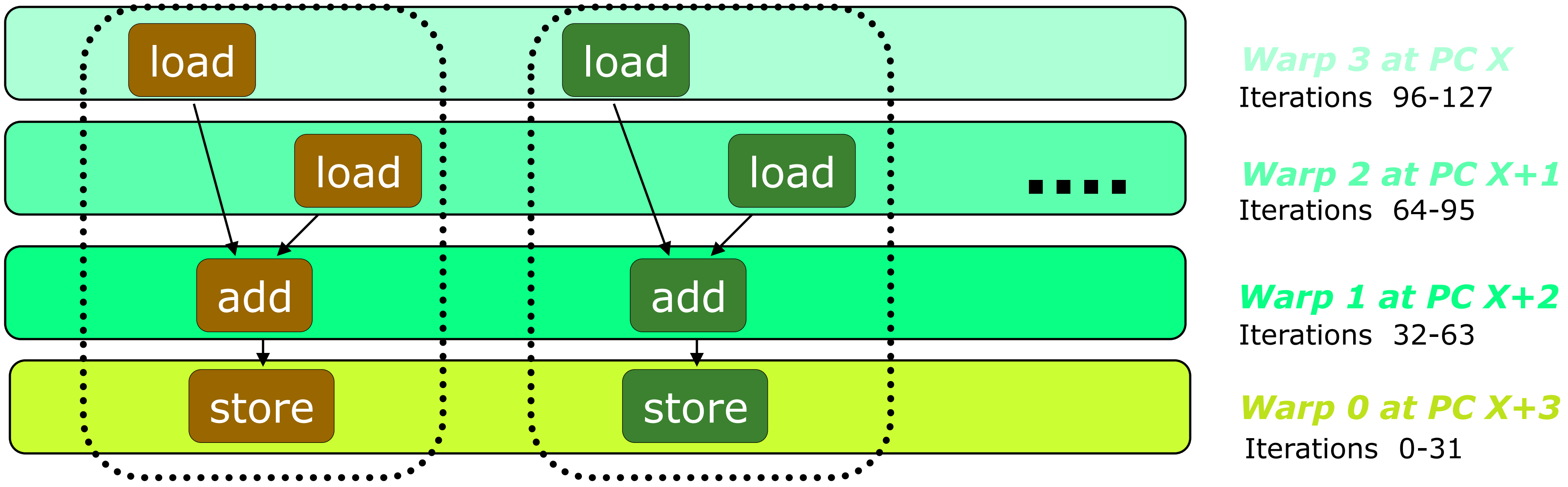
- Assume a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading of warps



# Fine-Grained Multithreading (FGMT) of Warps

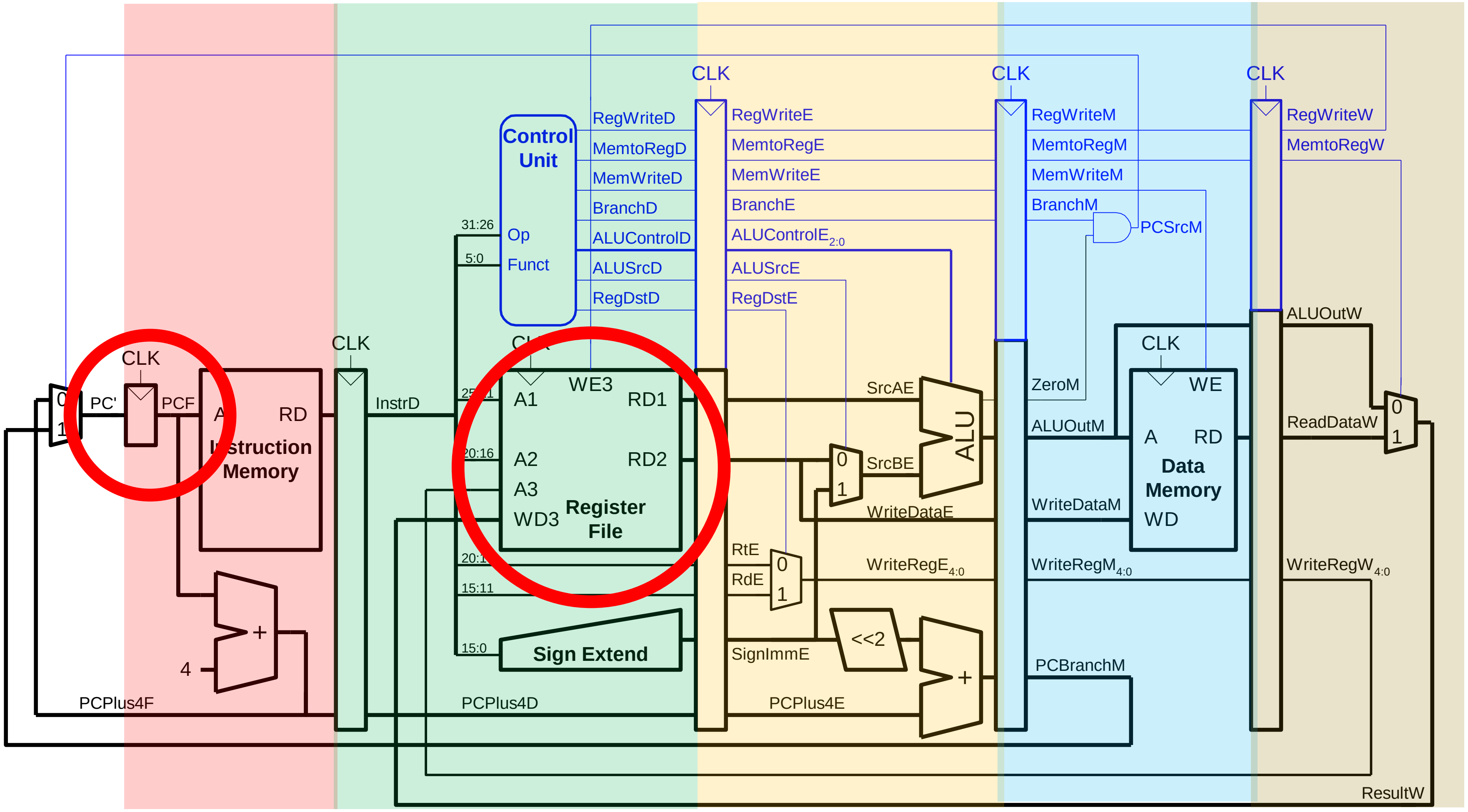
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

- 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

# Fine-Grained Multithreading: Basic Idea

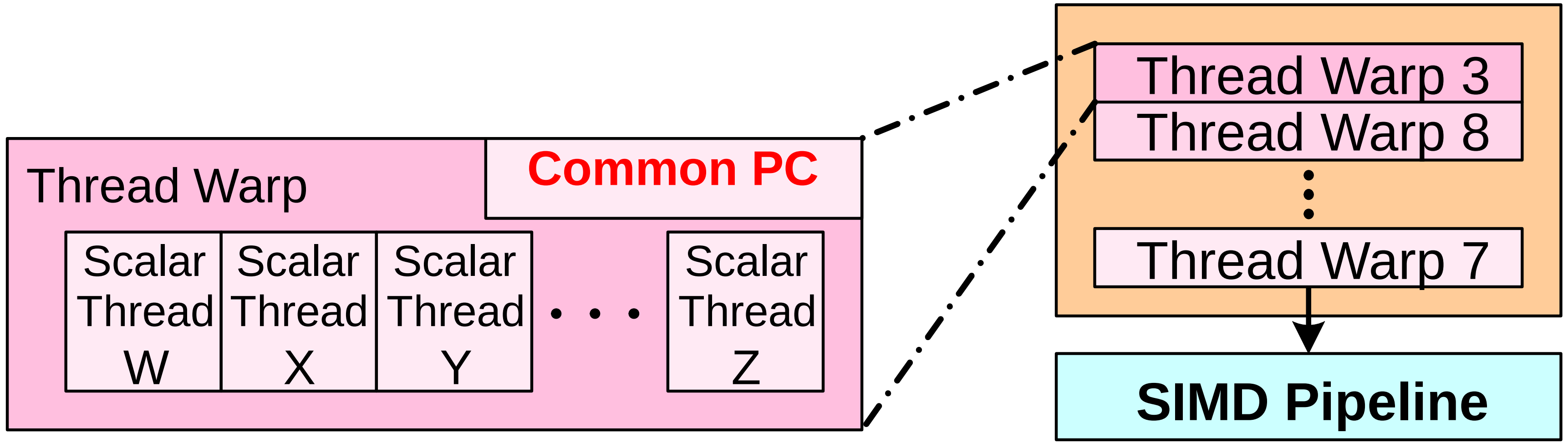


Each pipeline stage has an instruction from a different, completely-independent thread

We need a PC and a register file for each thread + muxes and control

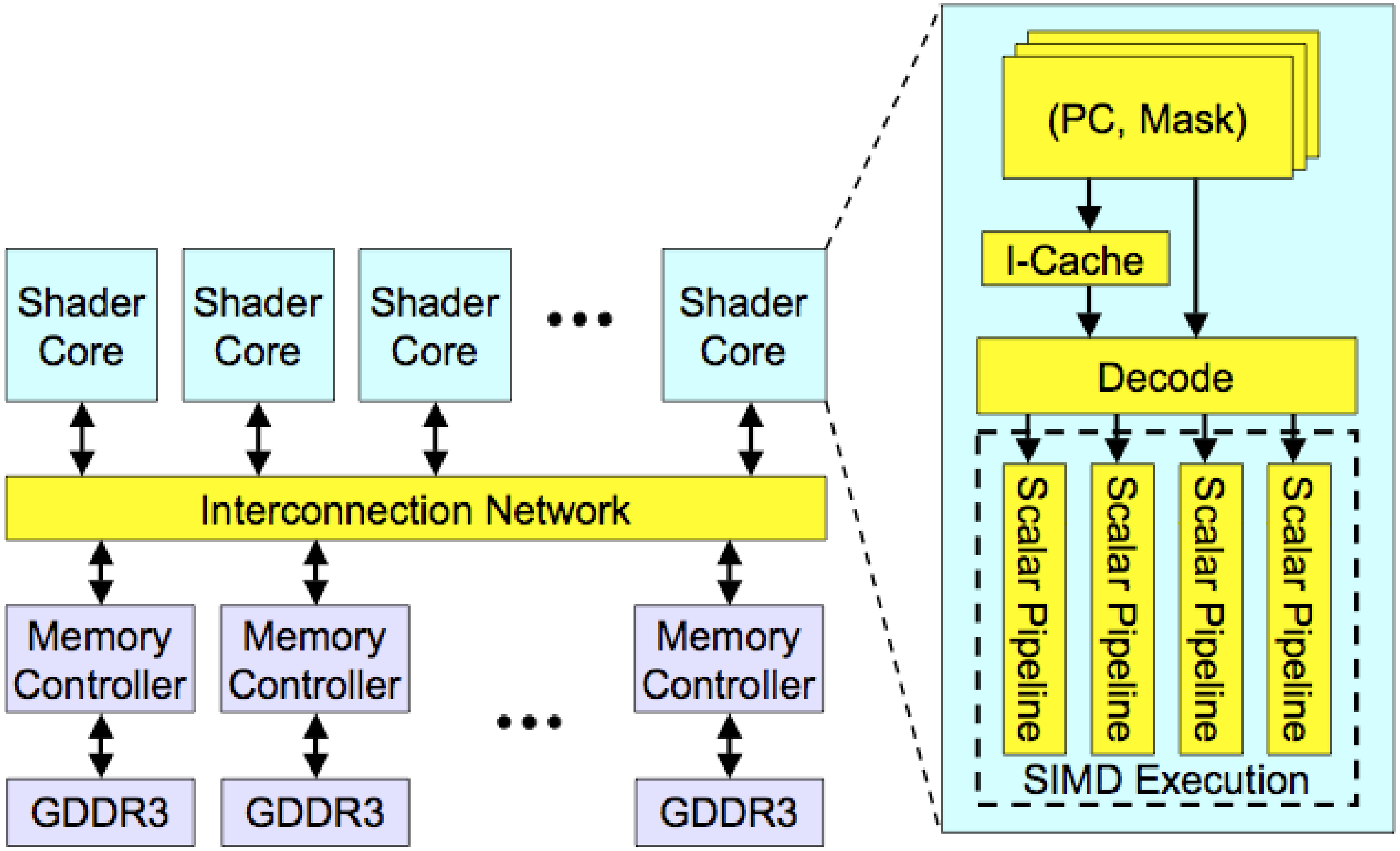
# Warps and Warp-Level FGMT

- 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.

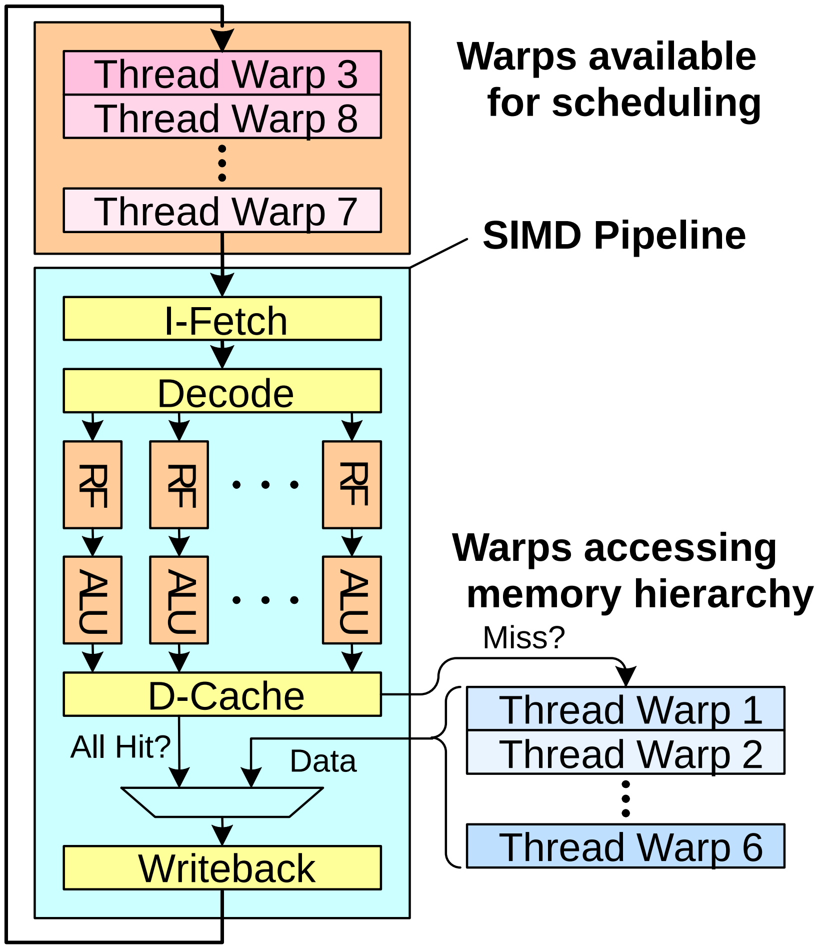
# High-Level View of a GPU



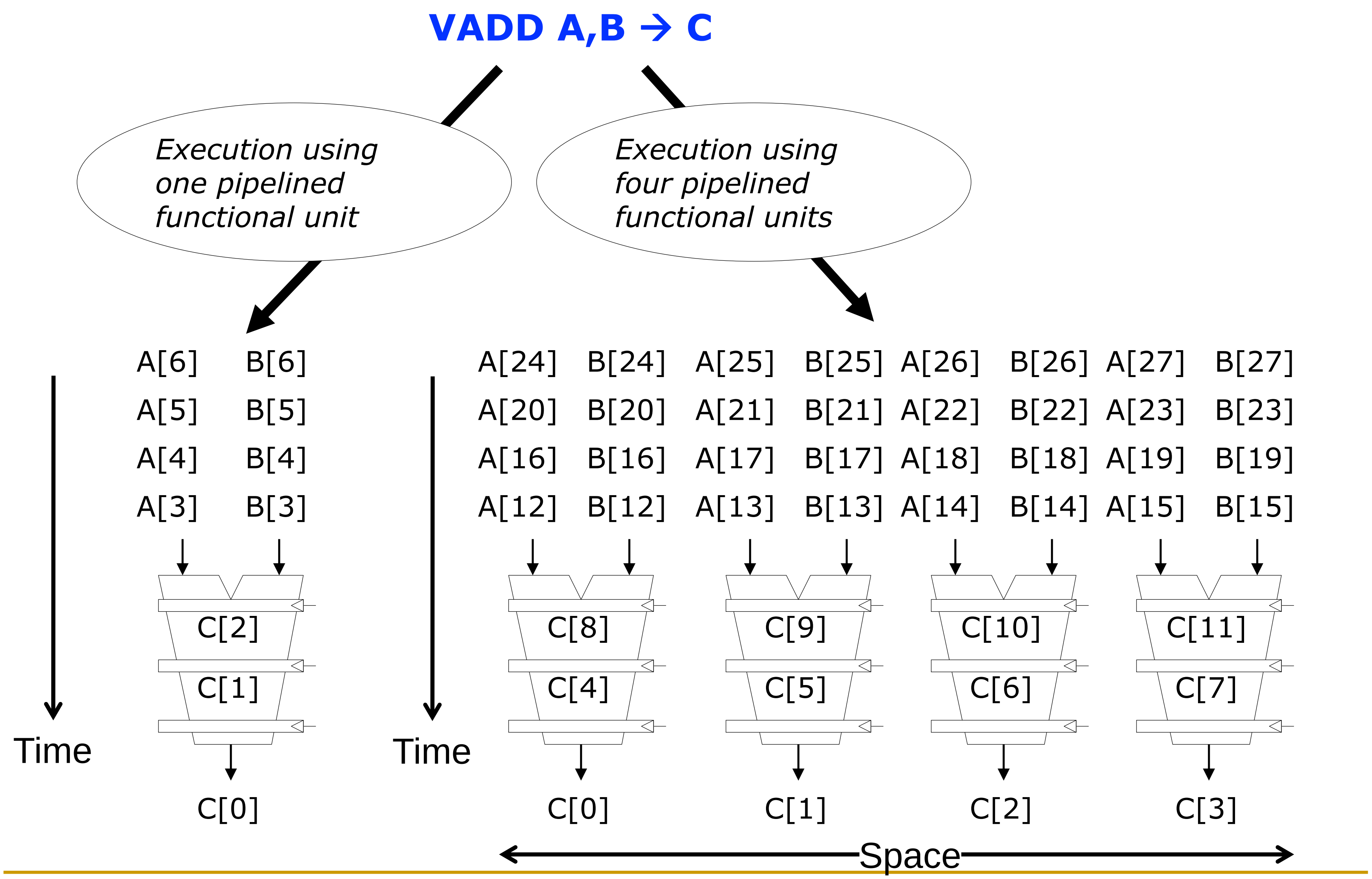
Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

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



# Recall: Vector Instruction Execution

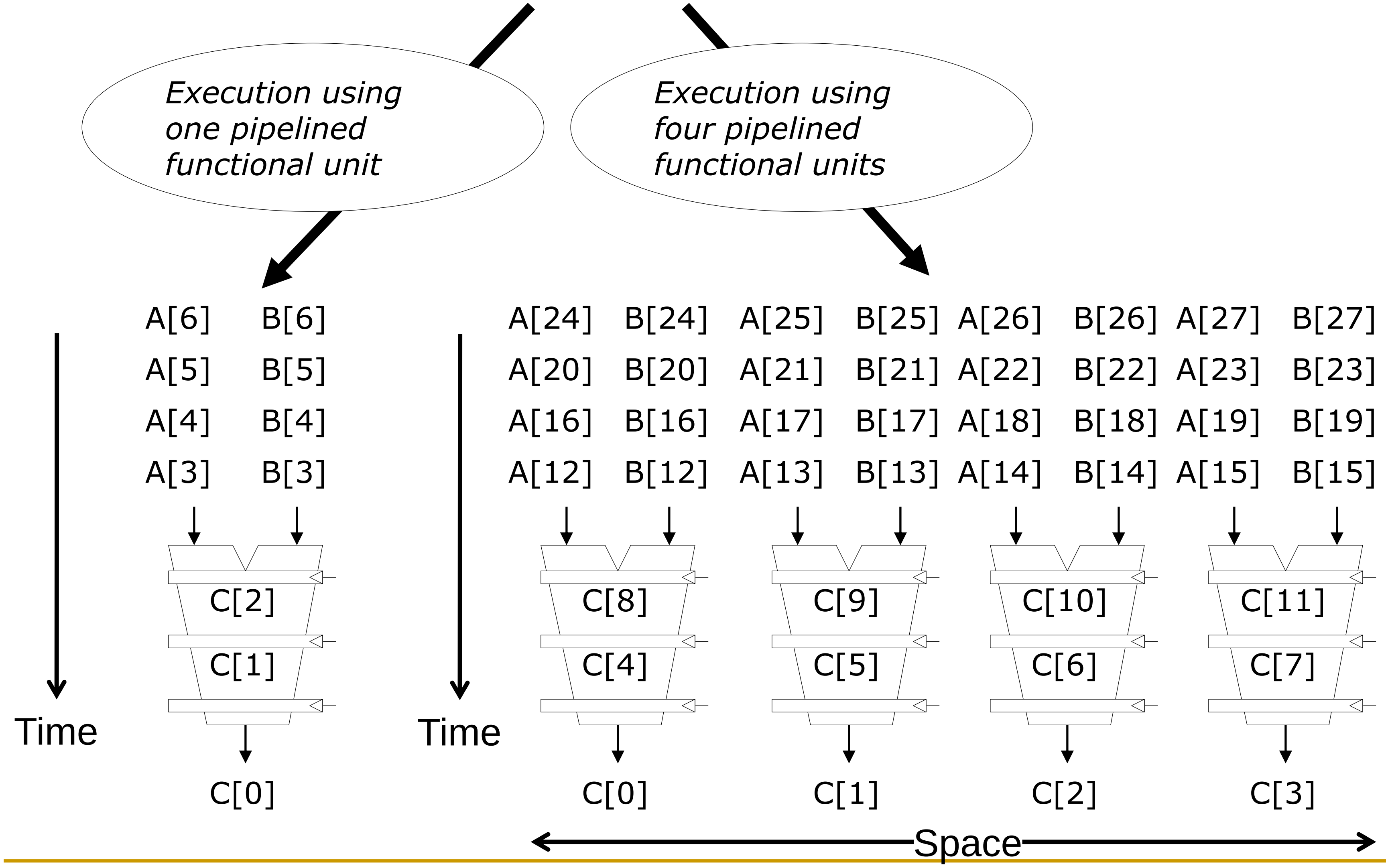


VADD A,B → C
Execution using one pipelined functional unit
Execution using four pipelined functional units
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
Time
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]
Time
Space

Slide credit: Krste Asanovic

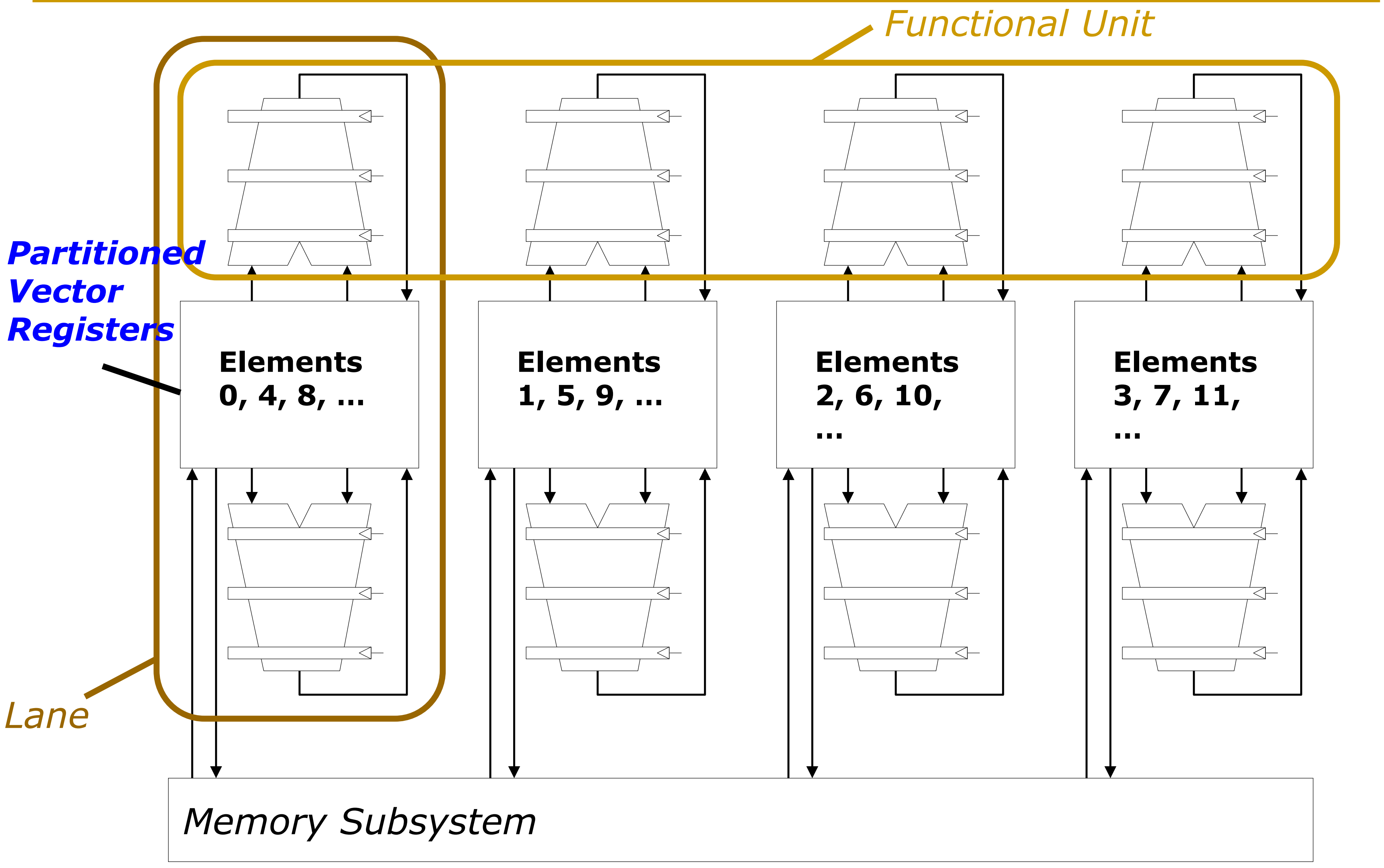
# Warp Execution (Recall the Previous Slide)

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

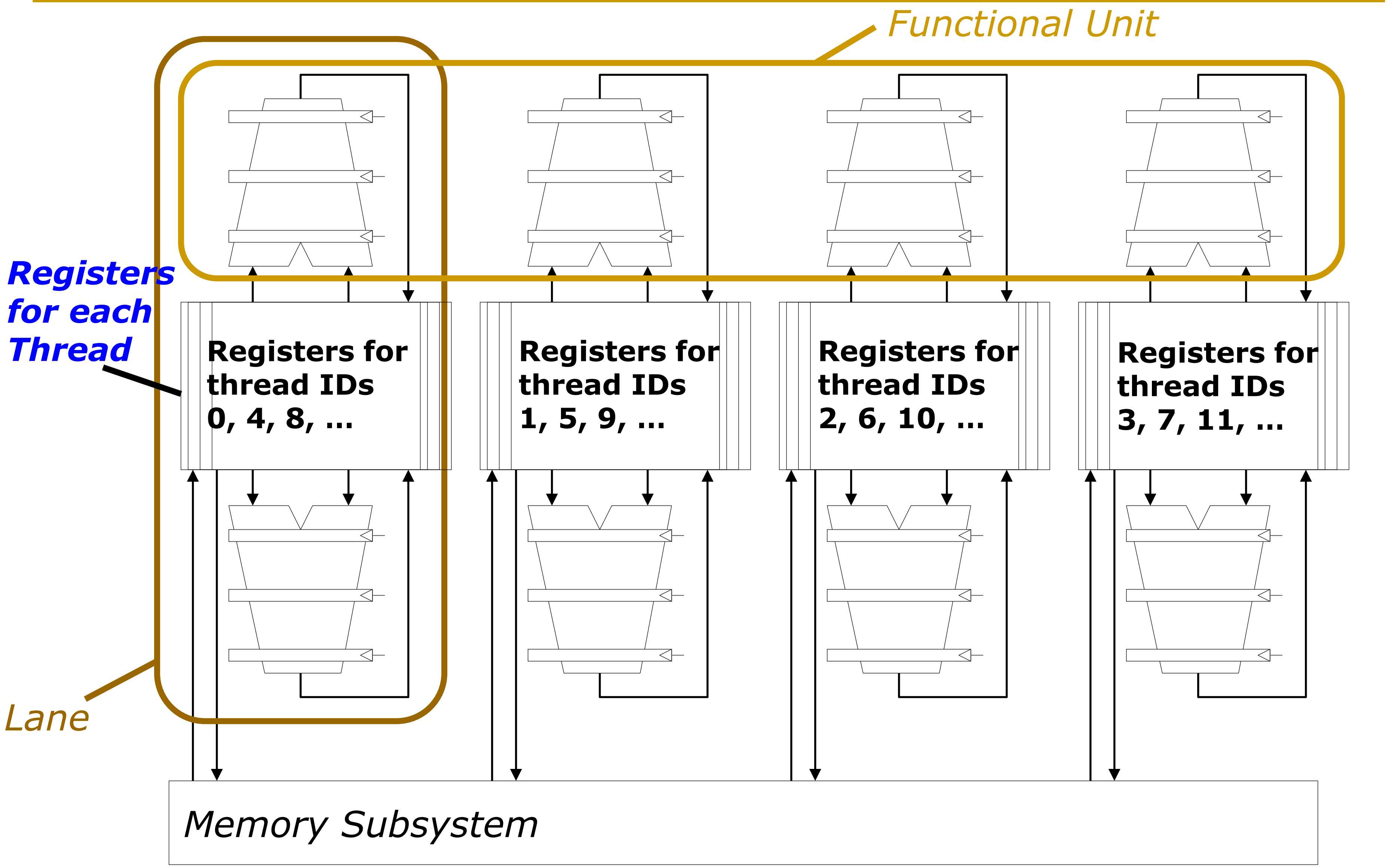


Execution using one pipelined functional unit
Execution using four pipelined functional units
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
Time
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]
Time
Space

# Recall: Vector Unit Structure



# GPU SIMD Execution Unit Structure

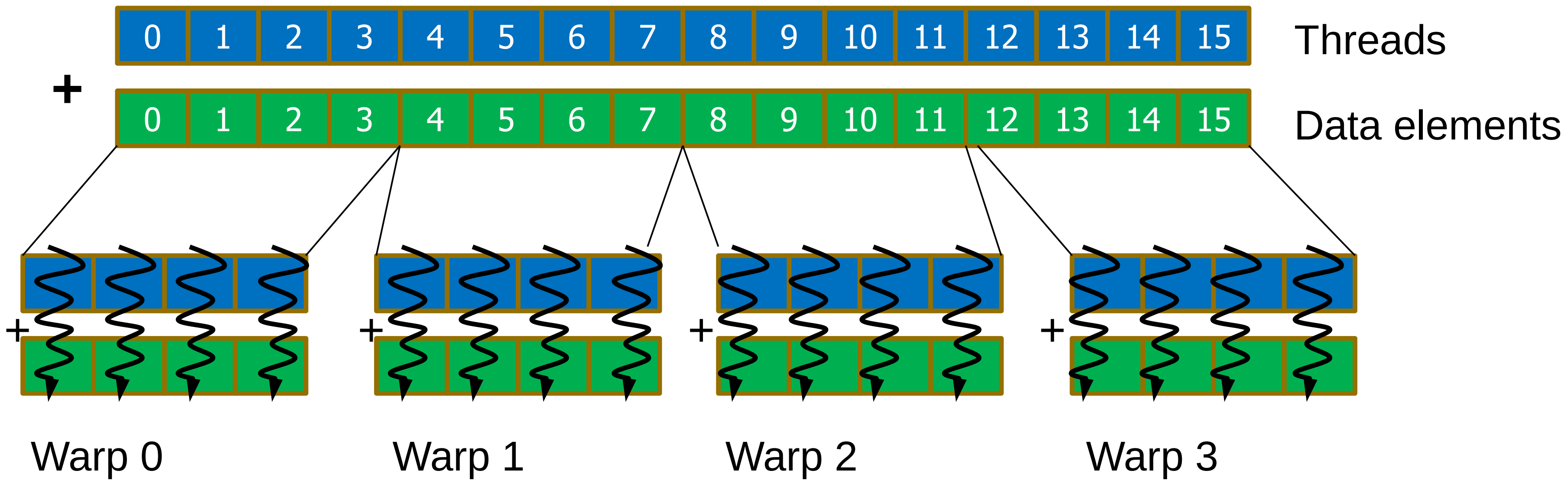

Functional Unit
Registers for each Thread
Registers for thread IDs 0, 4, 8, …
Registers for thread IDs 1, 5, 9, …
Registers for thread IDs 2, 6, 10, …
Registers for thread IDs 3, 7, 11, …
Lane
Memory Subsystem

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

Let's assume N=16, 4 threads per warp → 4 warps



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

Slide credit: Hyesoon Kim

# 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)**

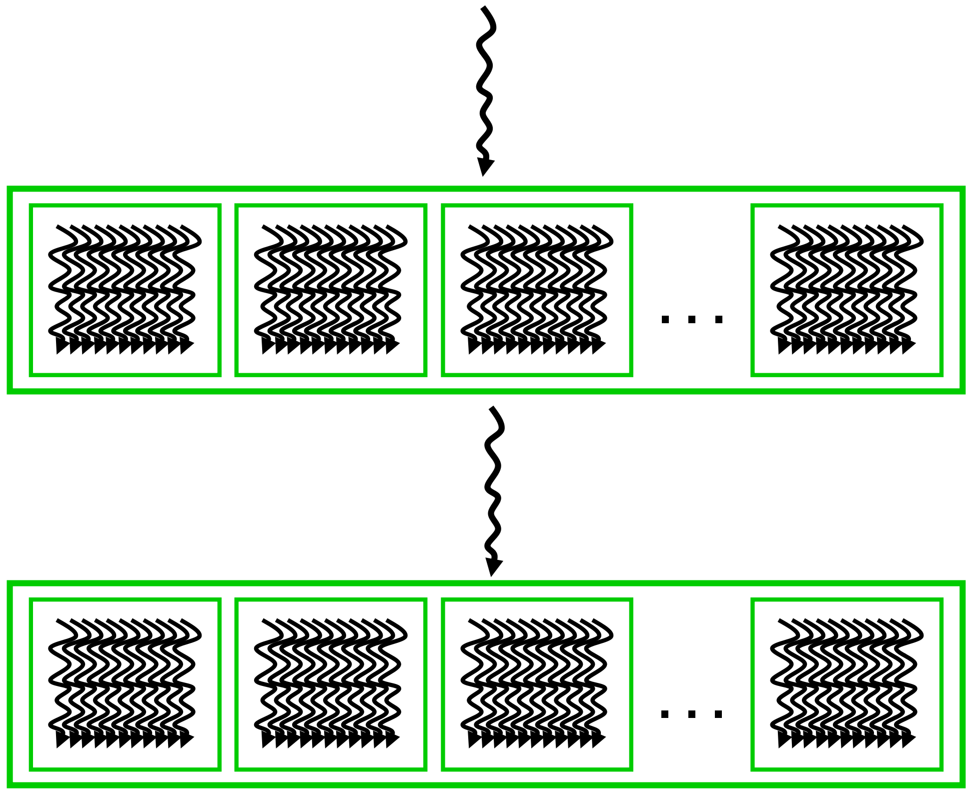
**Parallel Kernel (device)**

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

**Serial Code (host)**

**Parallel Kernel (device)**

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



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**

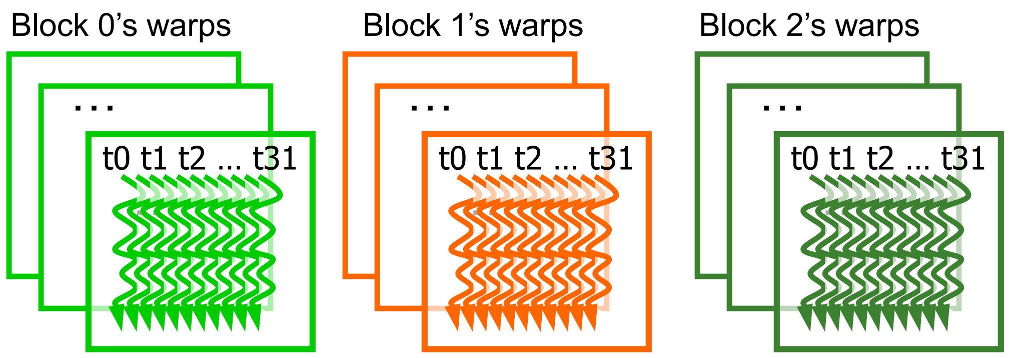
```
__global__  add_matrix
  ( float *a, float *b, float *c, int N) {
int i = blockIdx.x *  blockDim.x + threadIdx.x;
Int j = blockIdx.y * blockDim.y  + threadIdx.y;
int index = i + j*N;
 if (i < N && j < N)
   c[index] = a[index]+b[index];
}

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

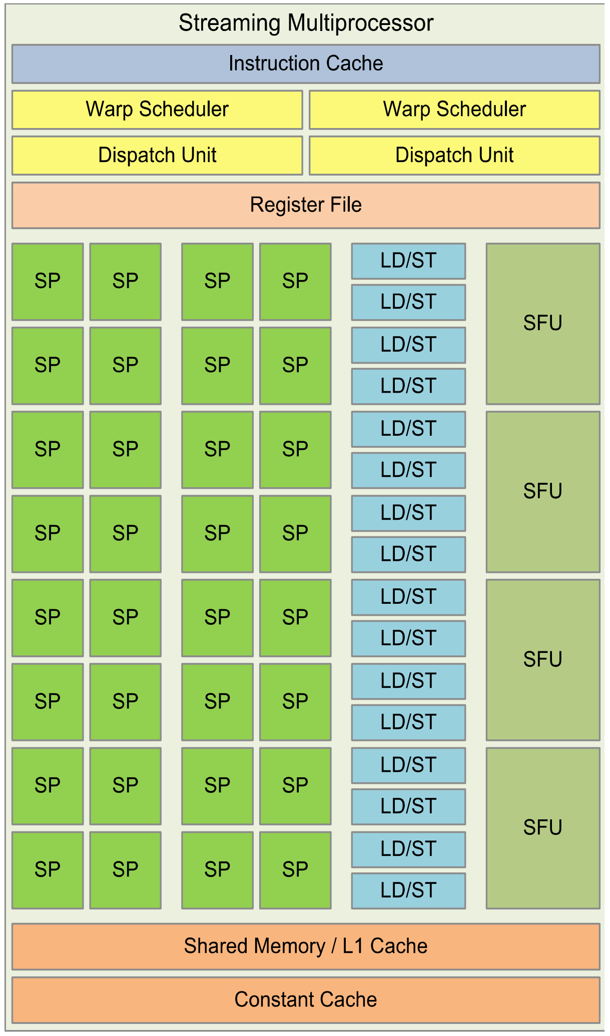
Slide credit: Hyesoon Kim

# From Blocks to Warps

- GPU 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

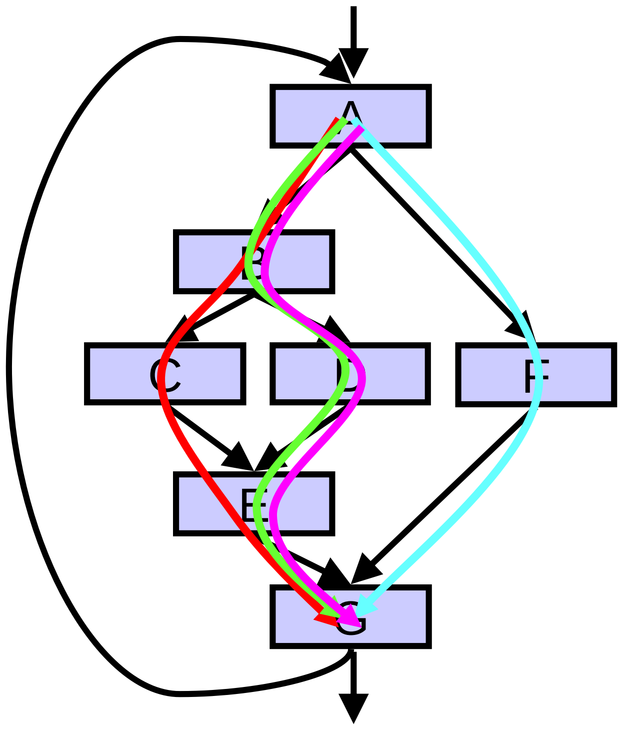
- Essentially, multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware

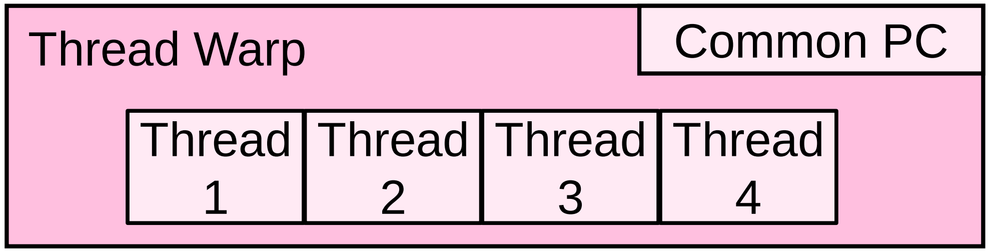
# SIMD vs. SIMT Execution Model

- 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



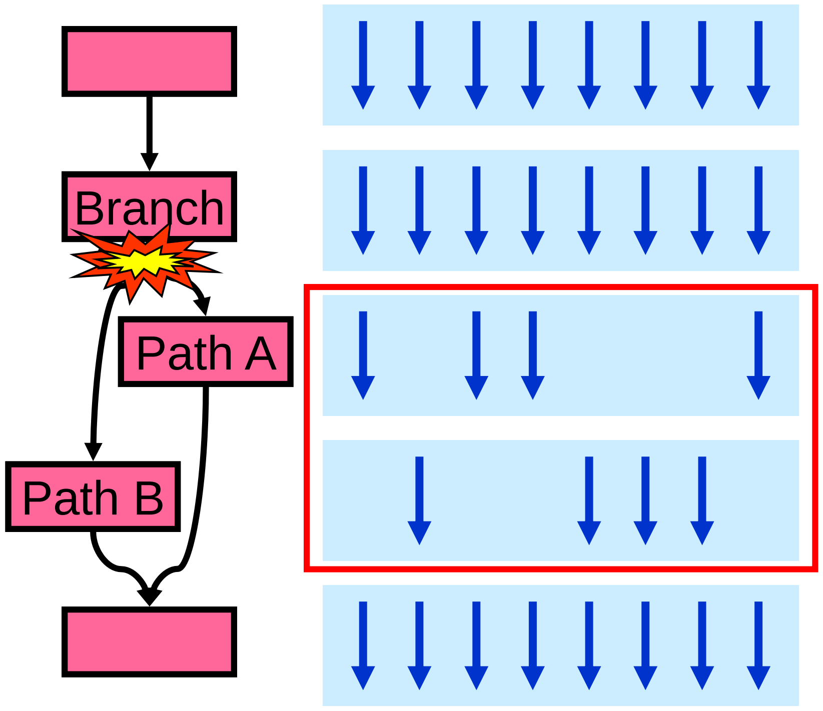


Slide credit: Tor Aamodt

# 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**

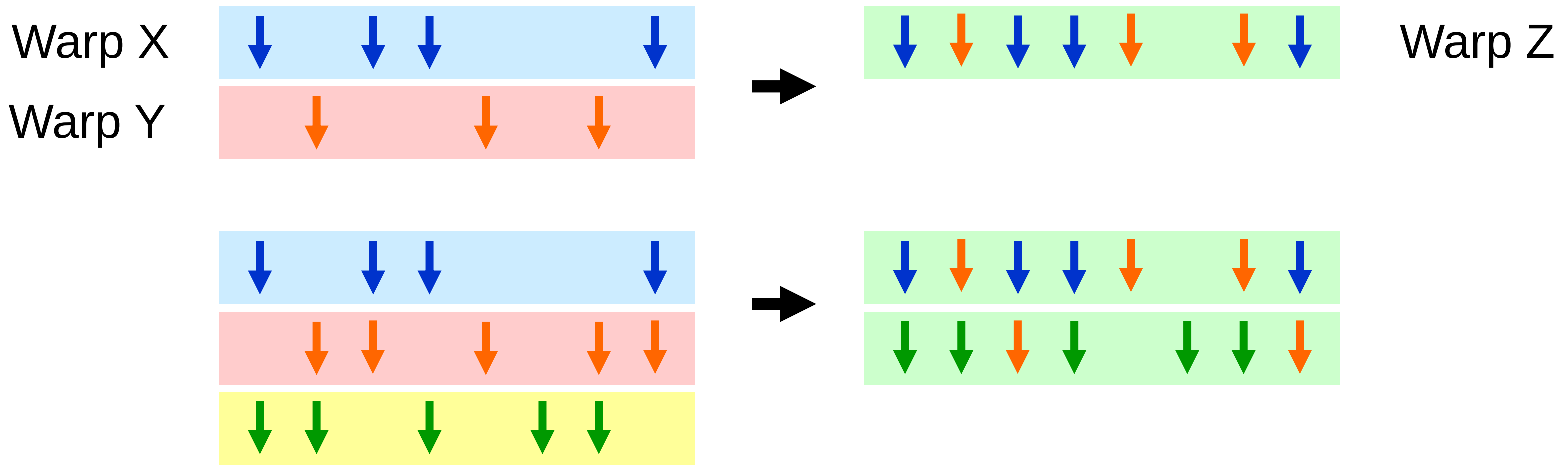
## Remember: Each Thread Is Independent

- 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)

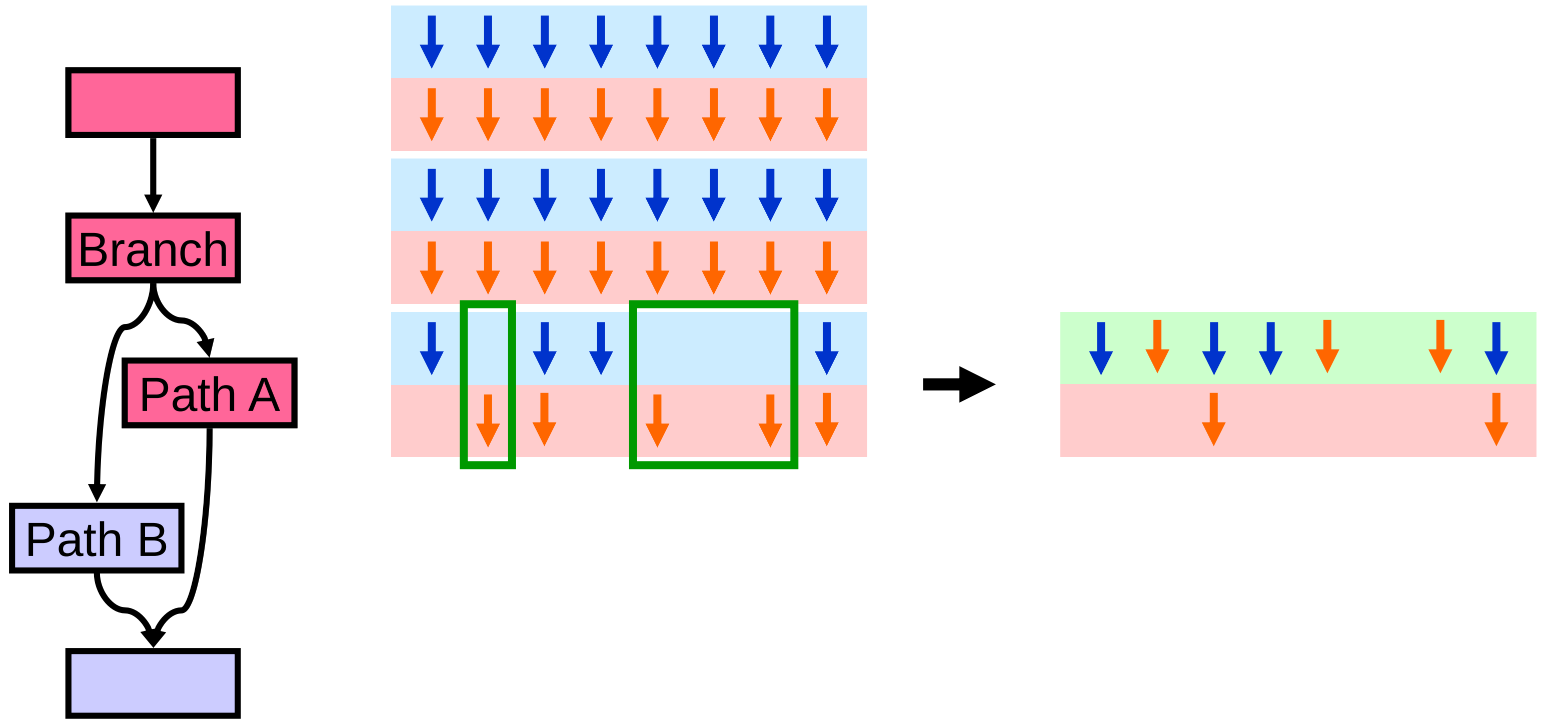
## Dynamic Warp Formation/Merging

- 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



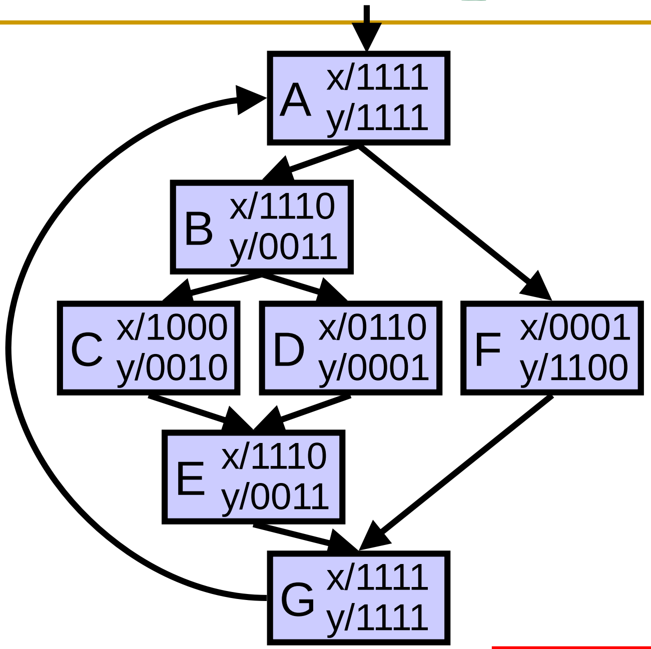
# Dynamic Warp Formation/Merging

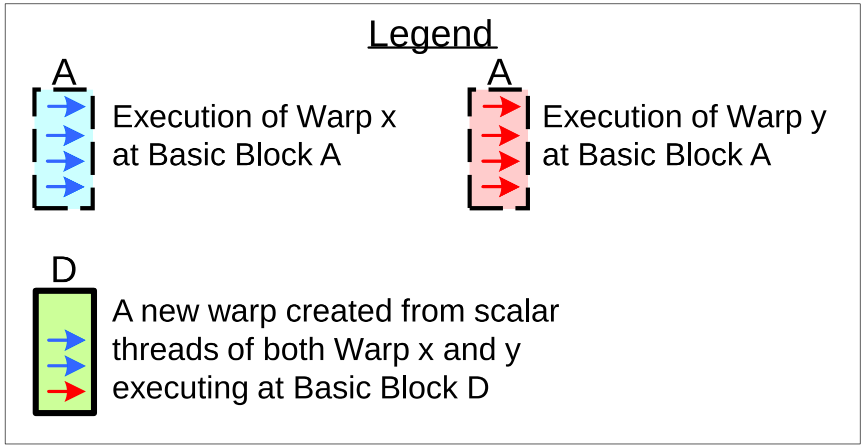
- Idea: Dynamically merge threads executing the same instruction, i.e., at the same PC (after branch divergence)

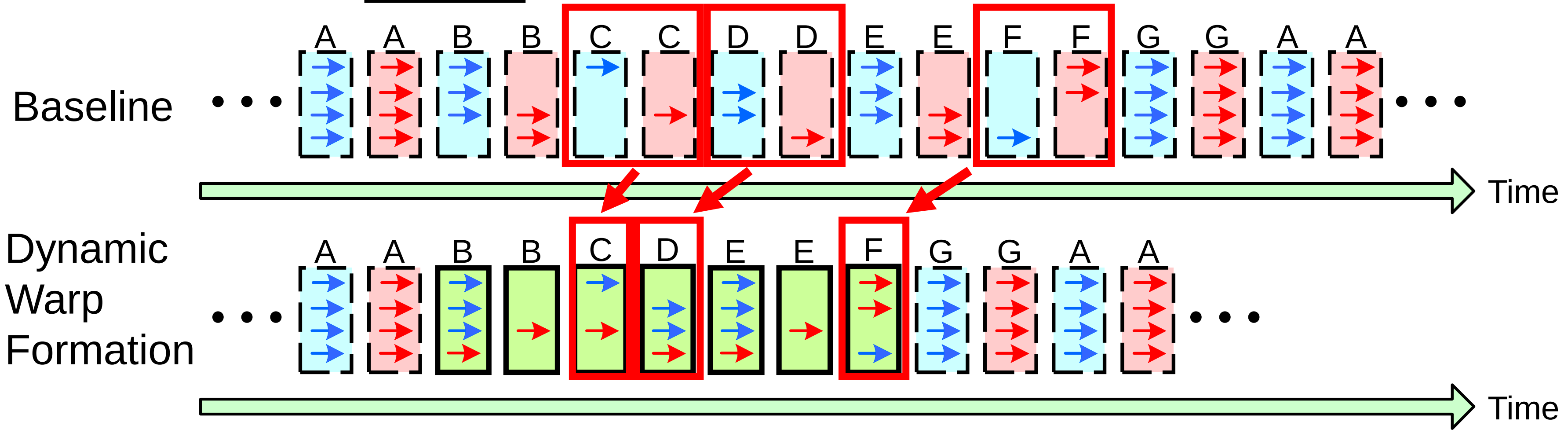


- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

# Dynamic Warp Formation Example

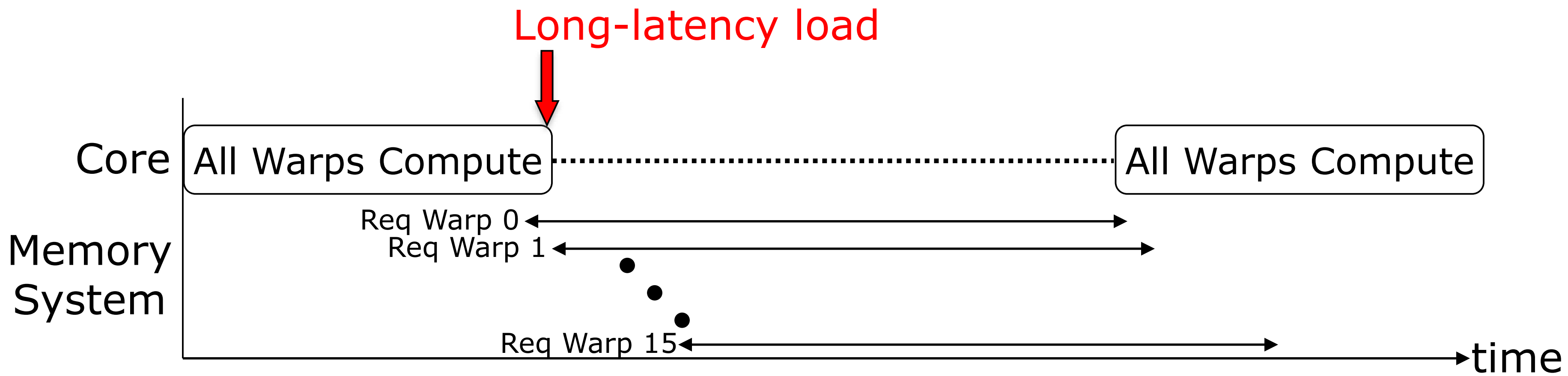



Legend
A
Execution of Warp x at Basic Block A
A
Execution of Warp y at Basic Block A
D
A new warp created from scalar threads of both Warp x and y executing at Basic Block D



# Large Warps and Two-Level Warp Scheduling

- Two main reasons for GPU resources be underutilized
  - Branch divergence
  - **Long latency operations**

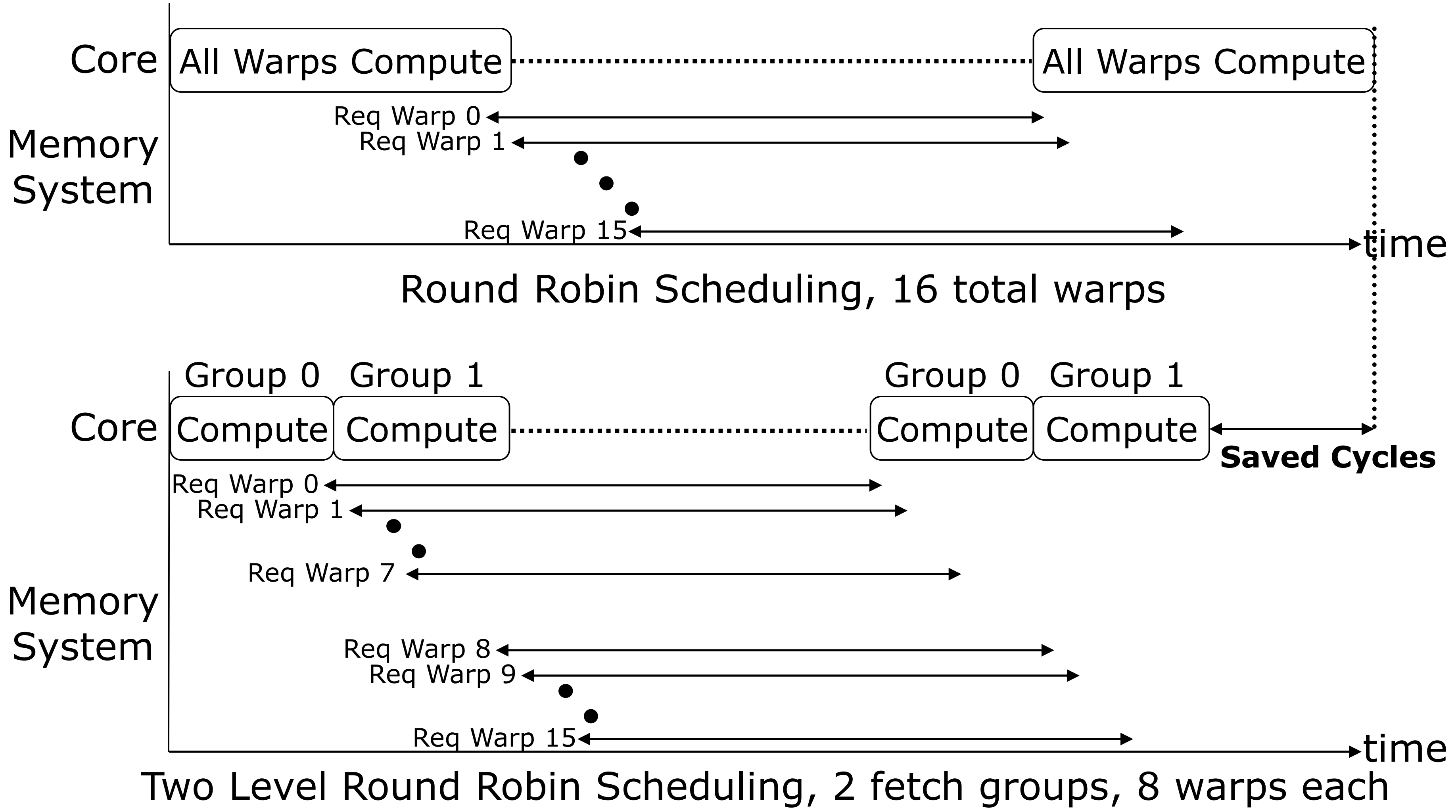


Round Robin Scheduling, 16 total warps

Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.

# Two-Level Scheduling of Warps

- Scheduling smaller warp groups reduces stalls due to long latency operations



Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.