CS425
Computer Systems Architecture

Fall 2023

Graphics Processing Units (GPU)

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
- First let's distinguish between
  - Programming Model (Software)
  - Execution Model (Hardware)

Programming Model vs. Hardware Execution Model

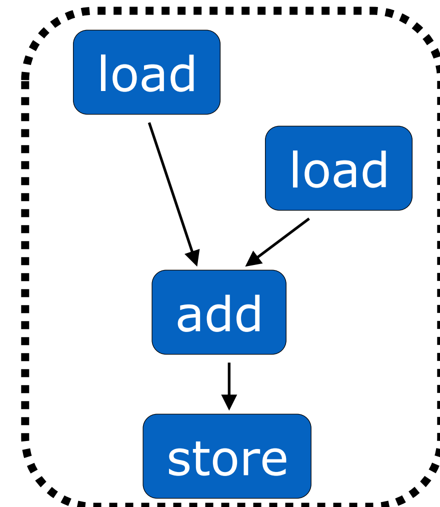
- Programming Model refers to how the programmer expresses the code
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), …
- Execution Model refers to how the hardware executes the code underneath
  - E.g., Out-of-order execution, Vector processor, Array processor, 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?

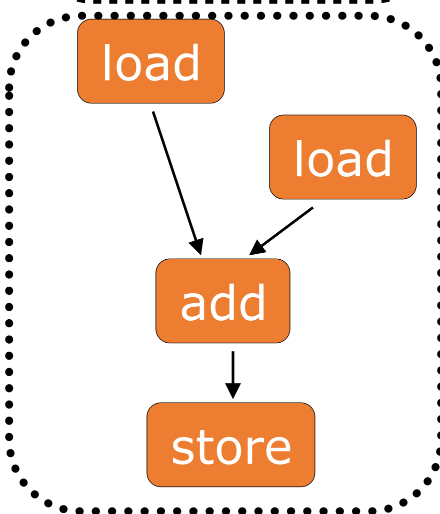
Scalar Sequential Code

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

Iter. 1



Iter. 2



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

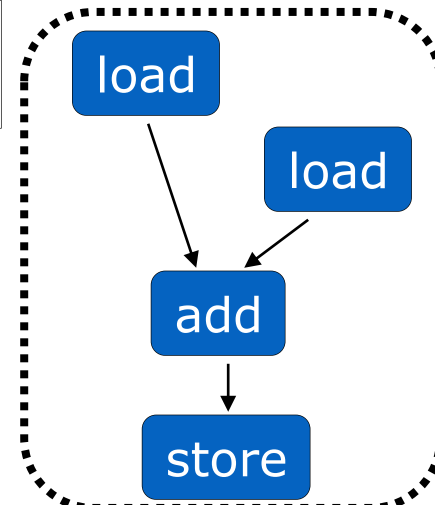
1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

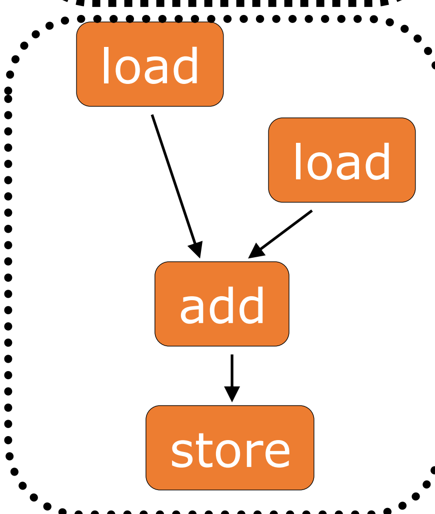
Scalar Sequential Code

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

Iter. 1



Iter. 2

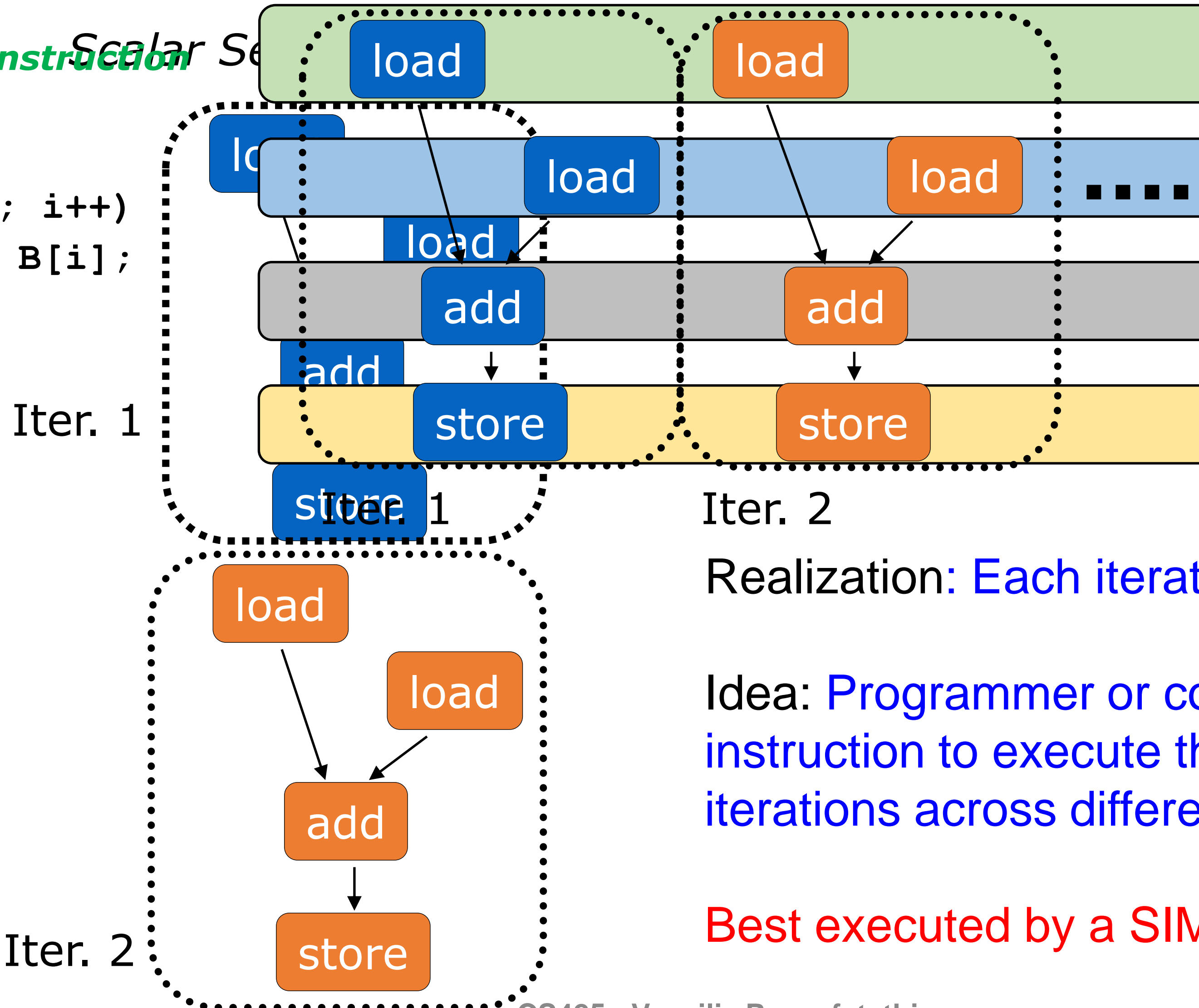


Can be executed on a:

- 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 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

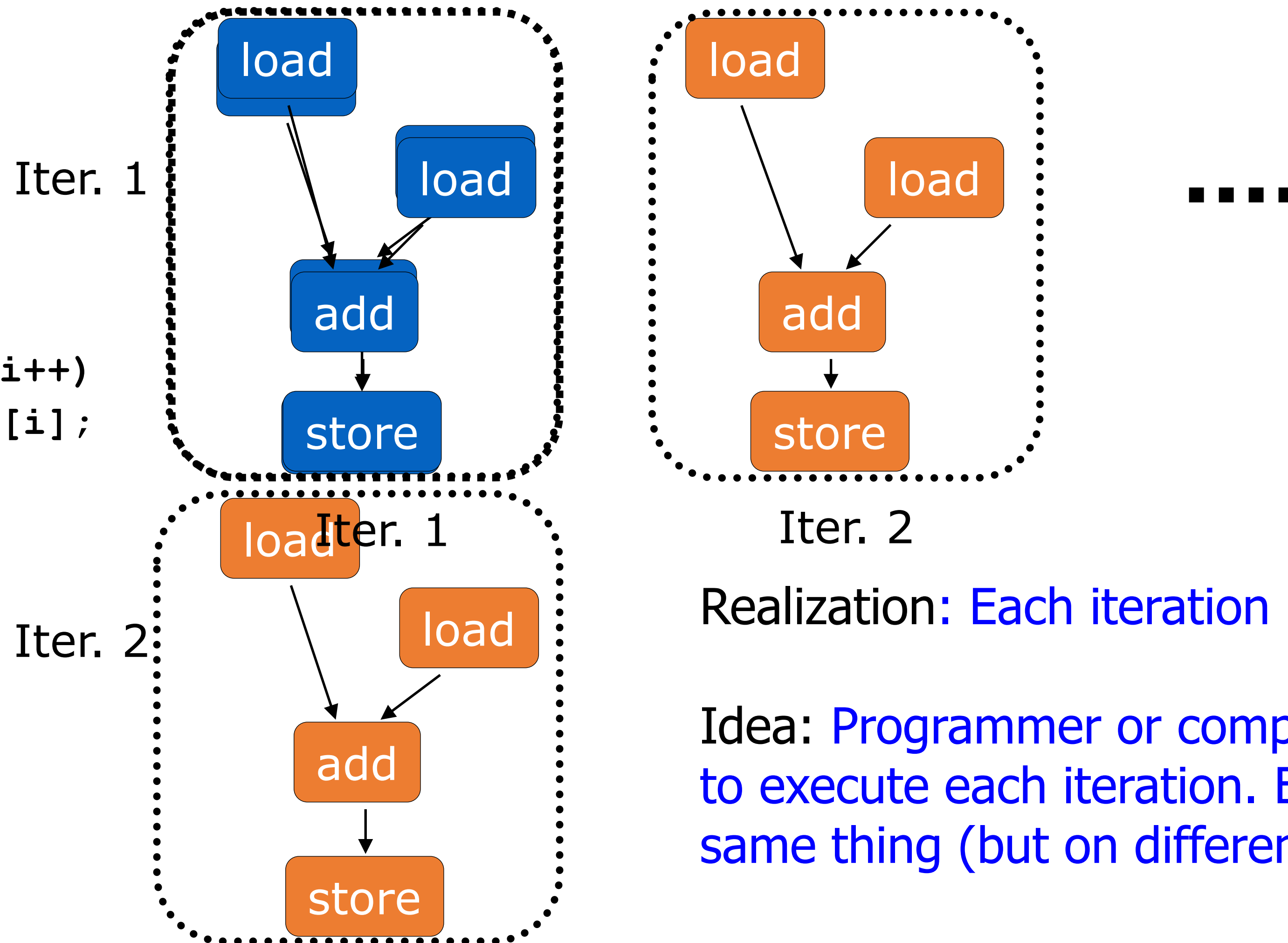
VST V3 → C

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



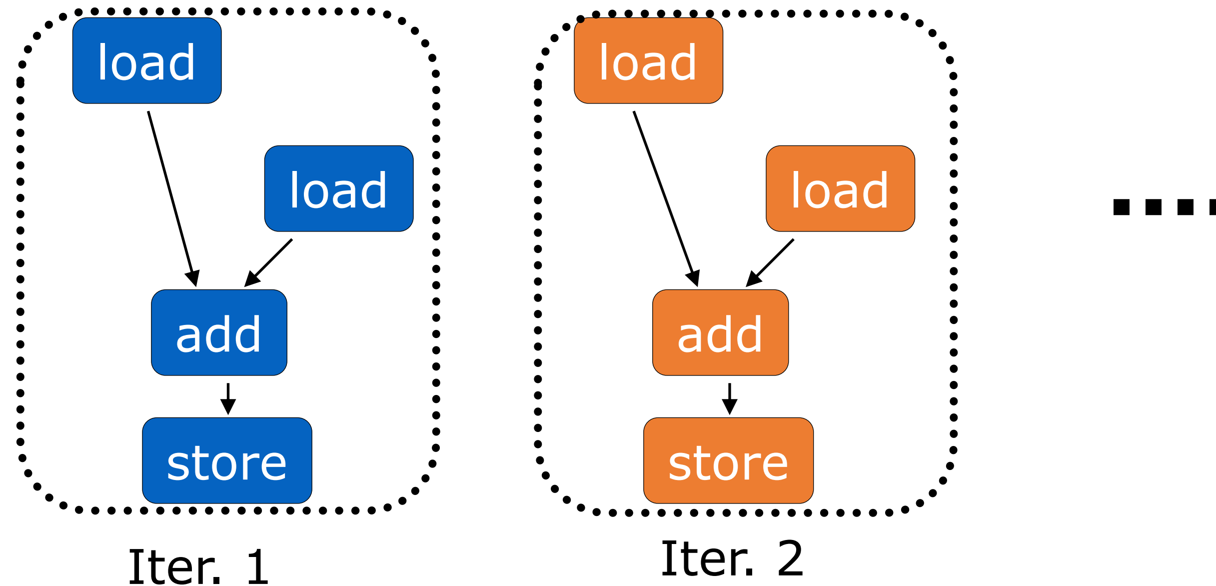

```
for (i=0; i < N; i++)
  C[i] = A[i] + B[i];
```

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



Realization: Each iteration is independent

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

This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread

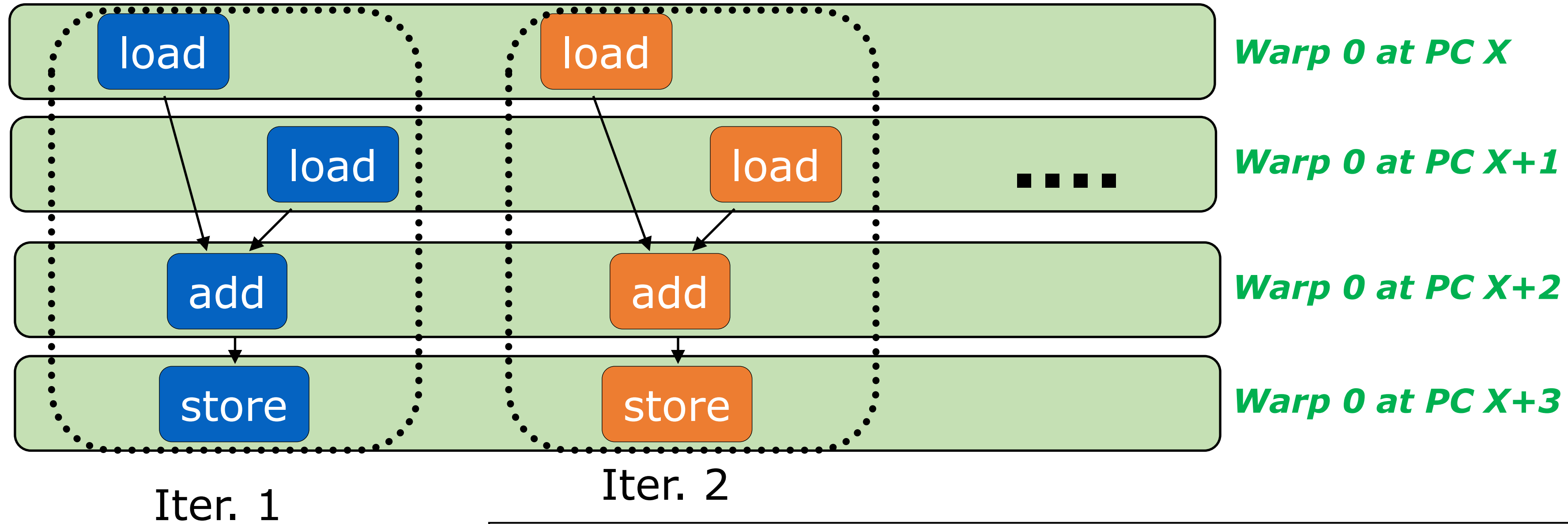
A GPU is a SIMD (SIMT) Machine

- 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

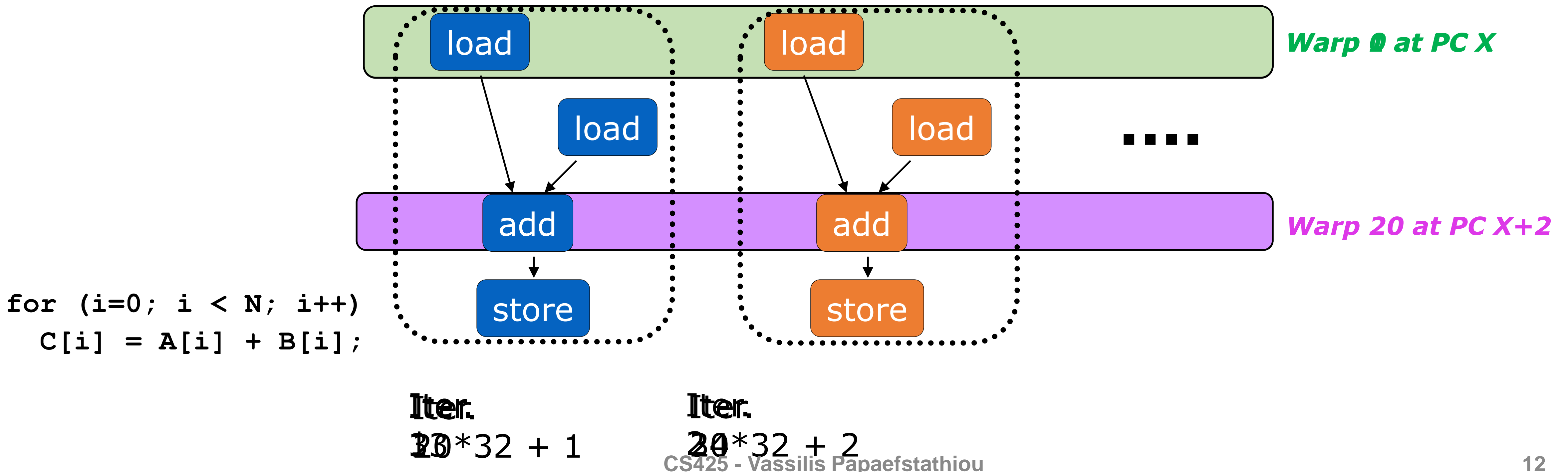
A GPU executes it using the SIMT model:
Single Instruction Multiple Thread

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

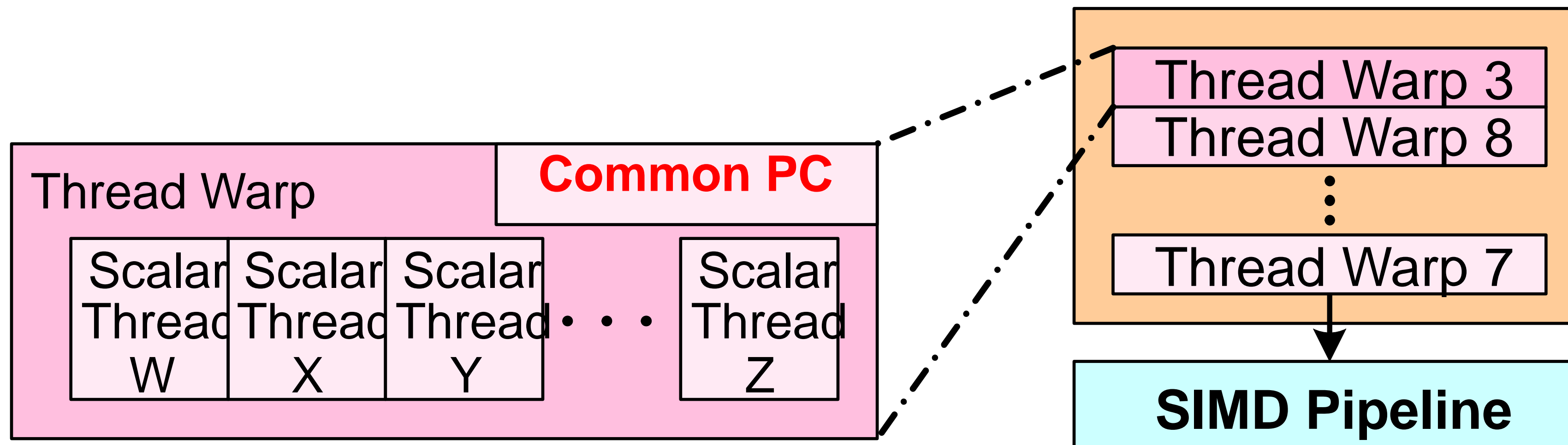
Fine-Grained Multithreading of Warps

- 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



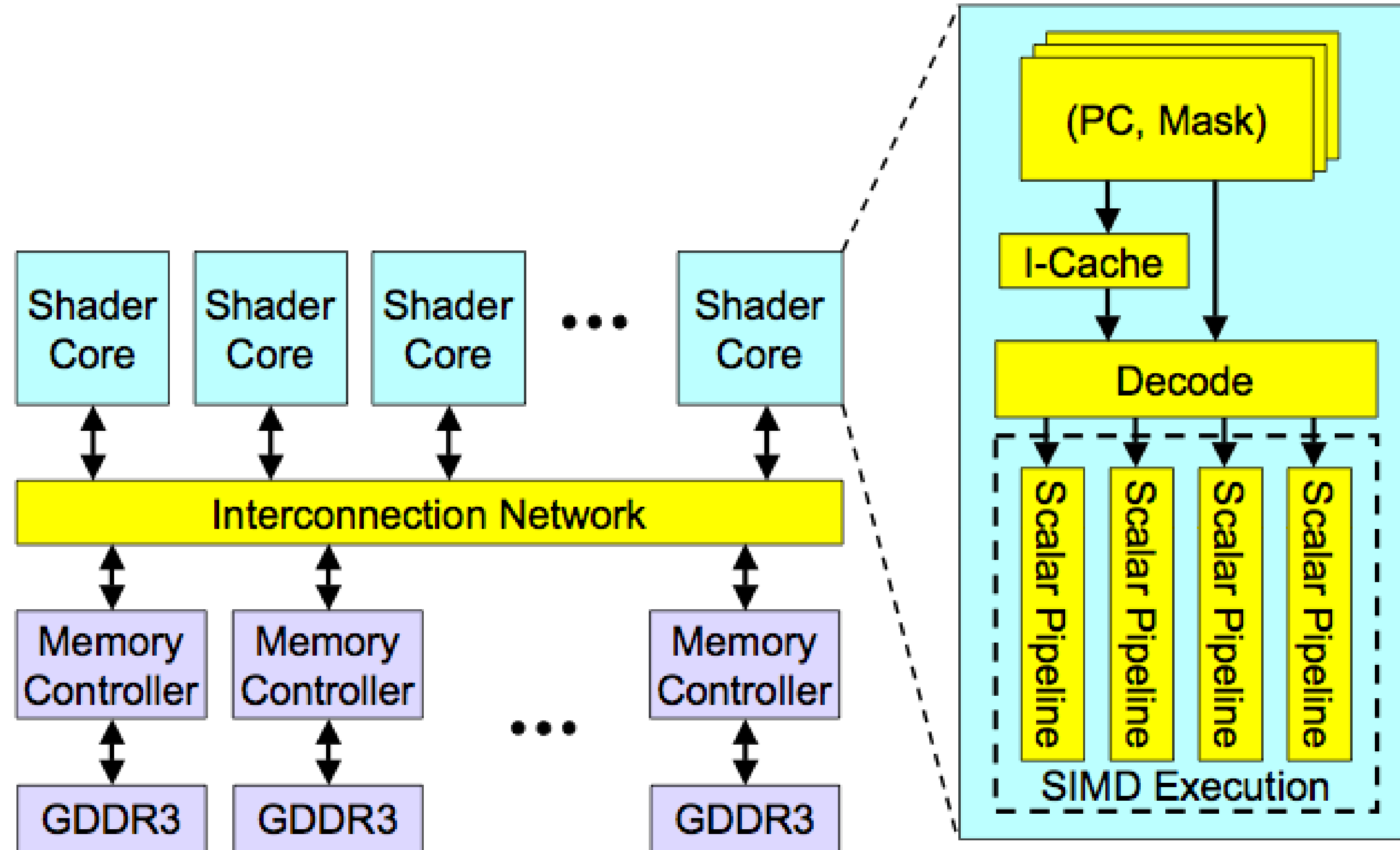
Warps and Warp-Level FGMT

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



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

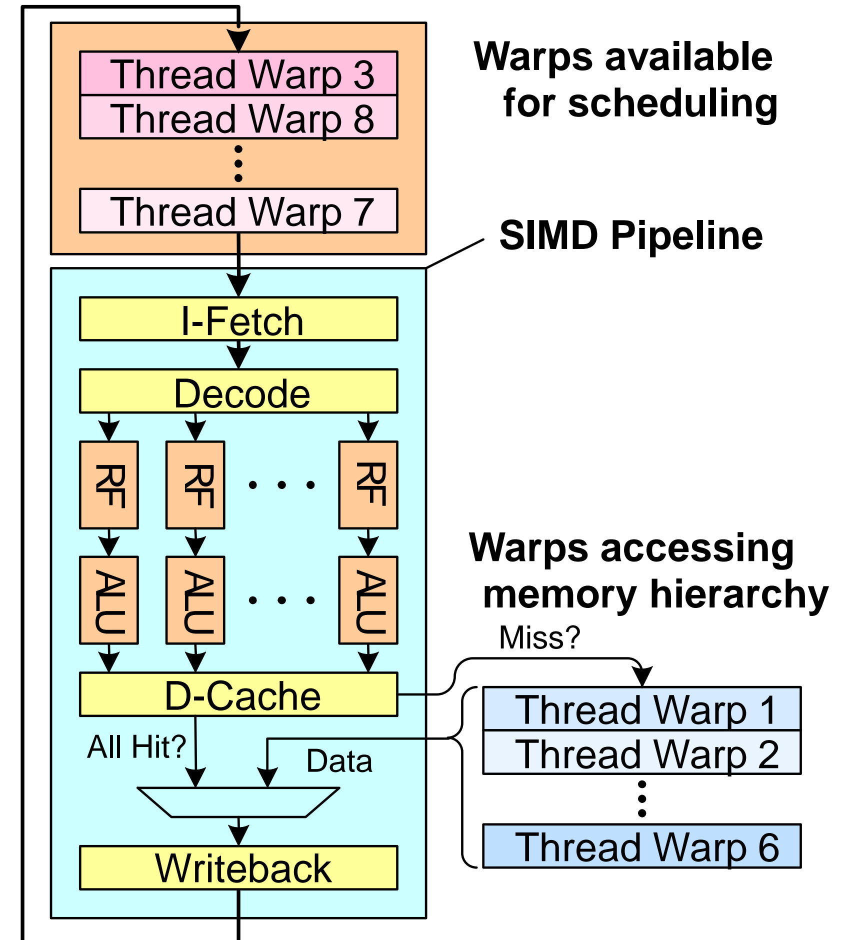
High-Level View of a GPU



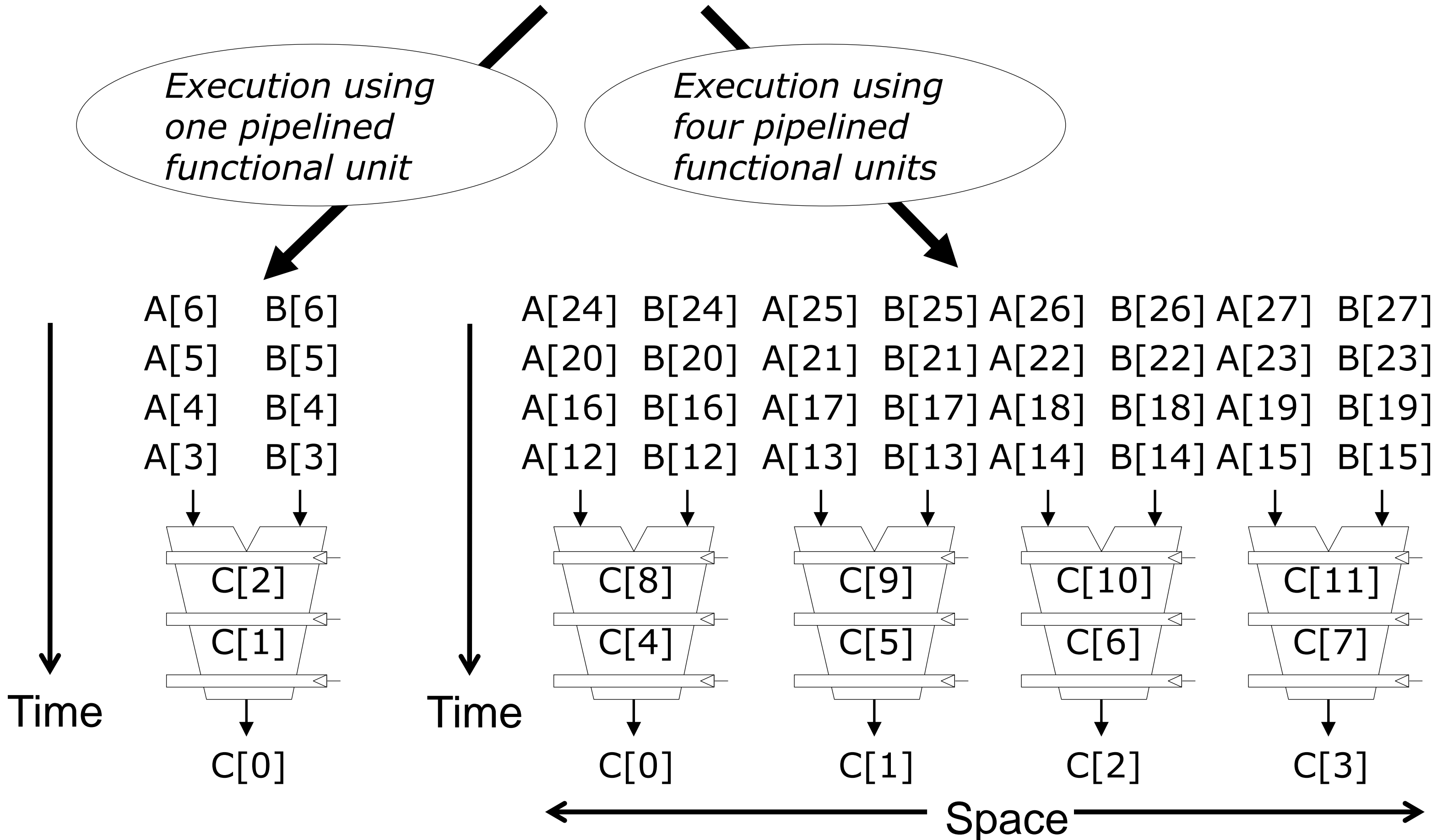
Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

Latency Hiding via Warp-Level FGMT

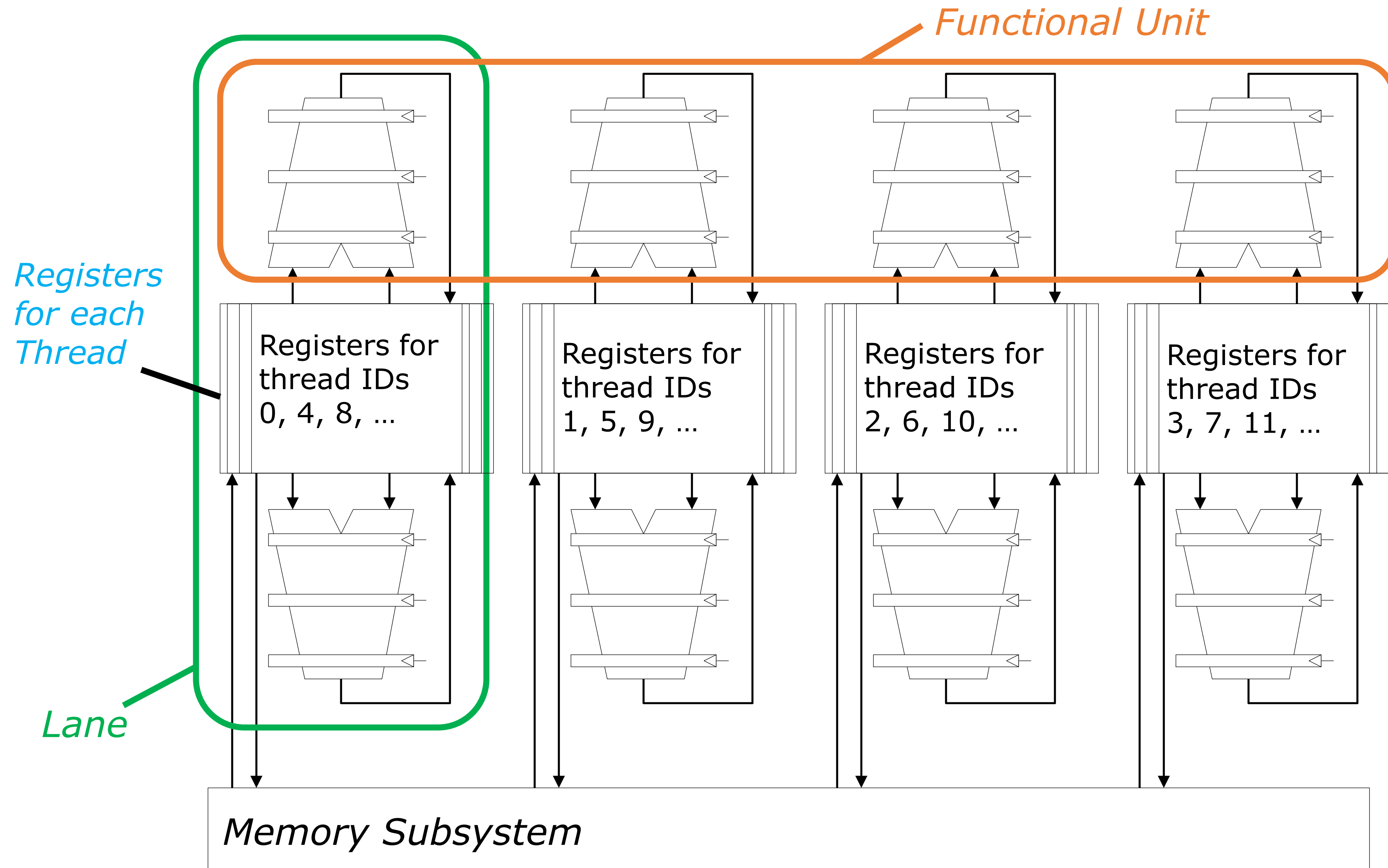
- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables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



Warp Execution



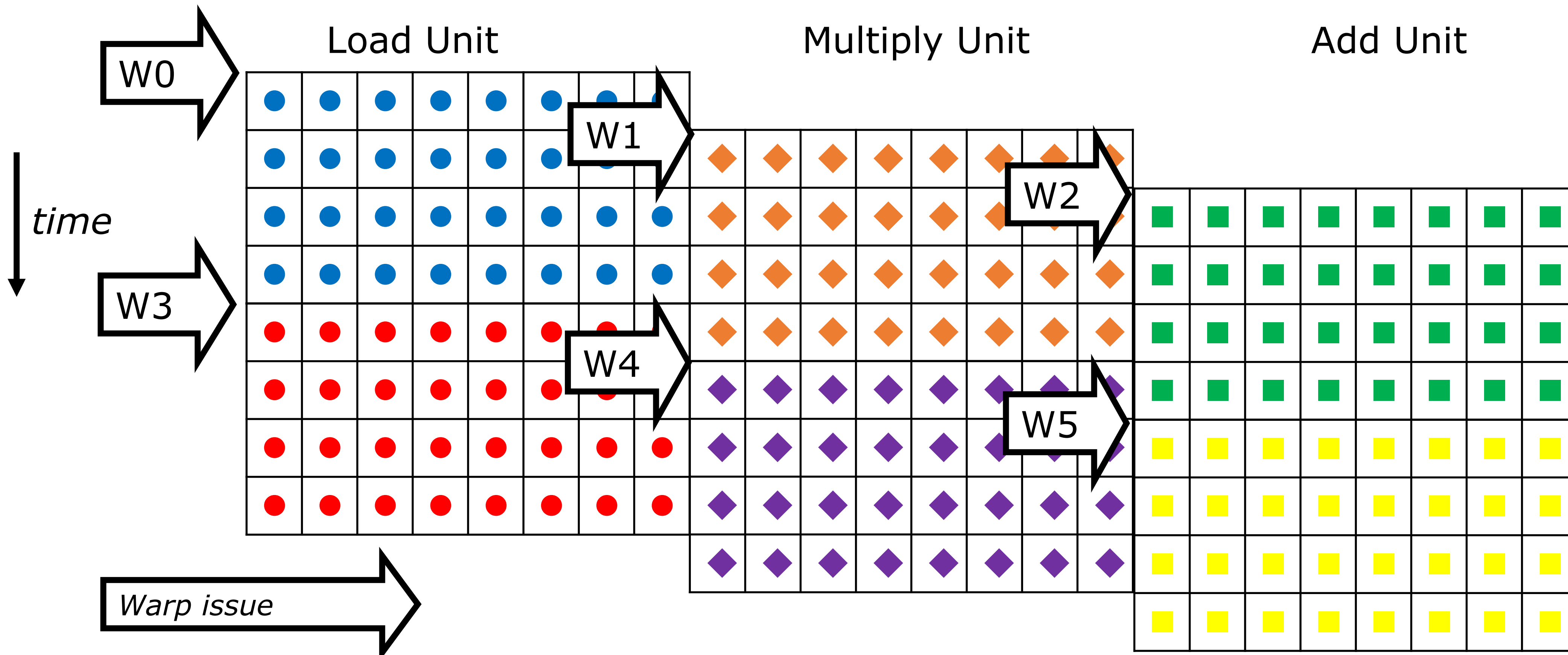
SIMD Execution Unit Structure



Warp Instruction Level Parallelism

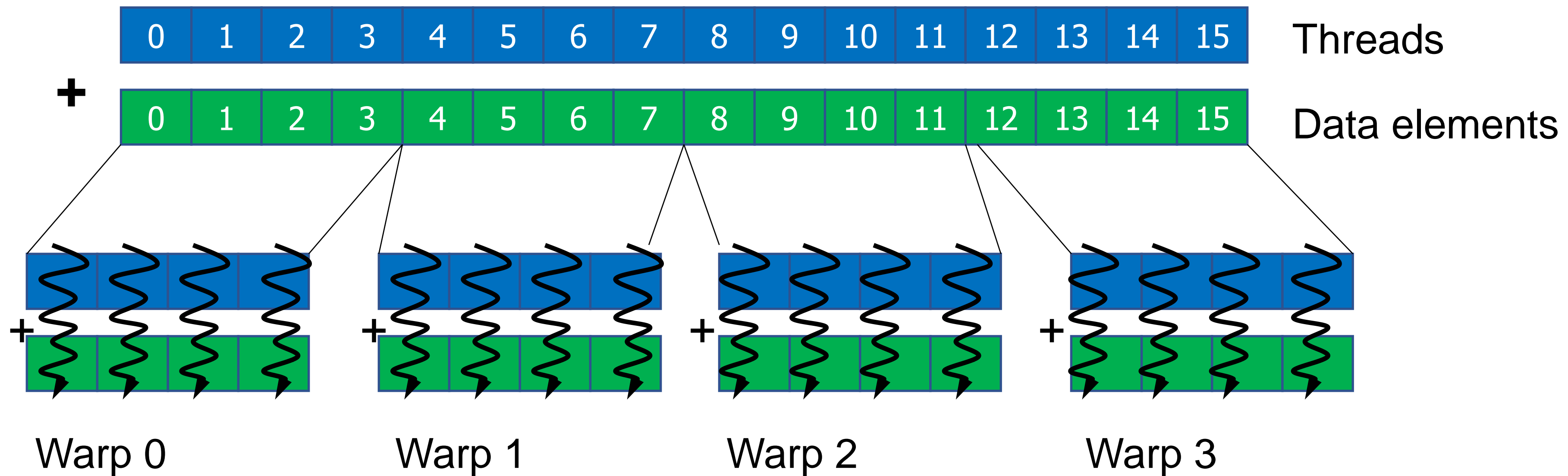
Can overlap execution of multiple instructions

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



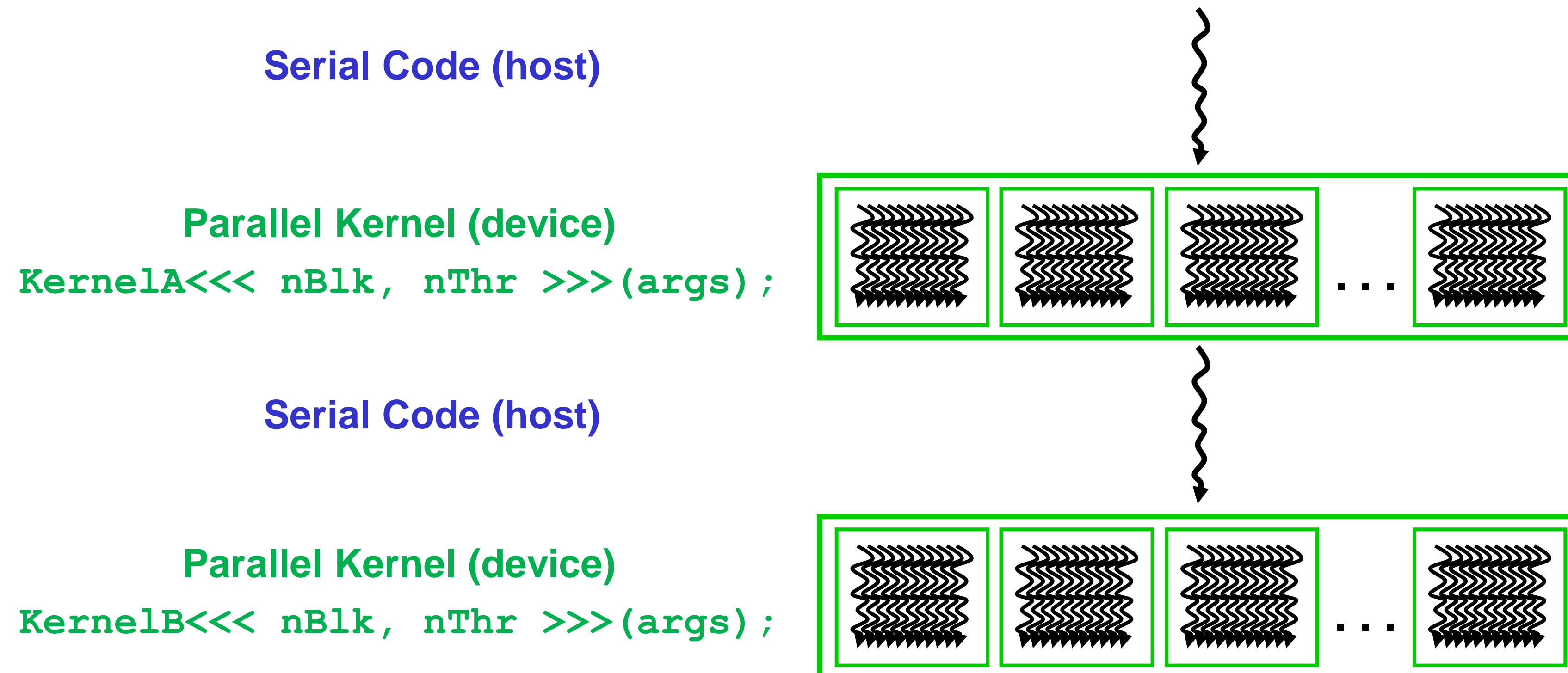
SIMT Memory Access

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



Warps *not* Exposed to GPU Programmers

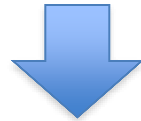
- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU: Blocks of threads



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;
}
```

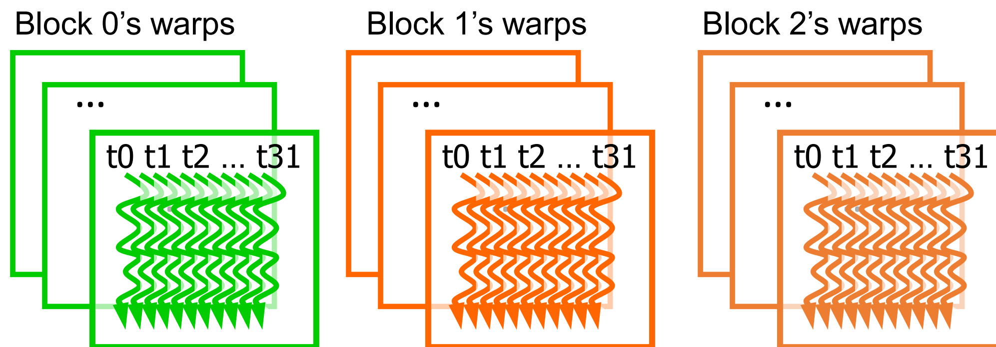
Sample GPU Program (Less Simplified)

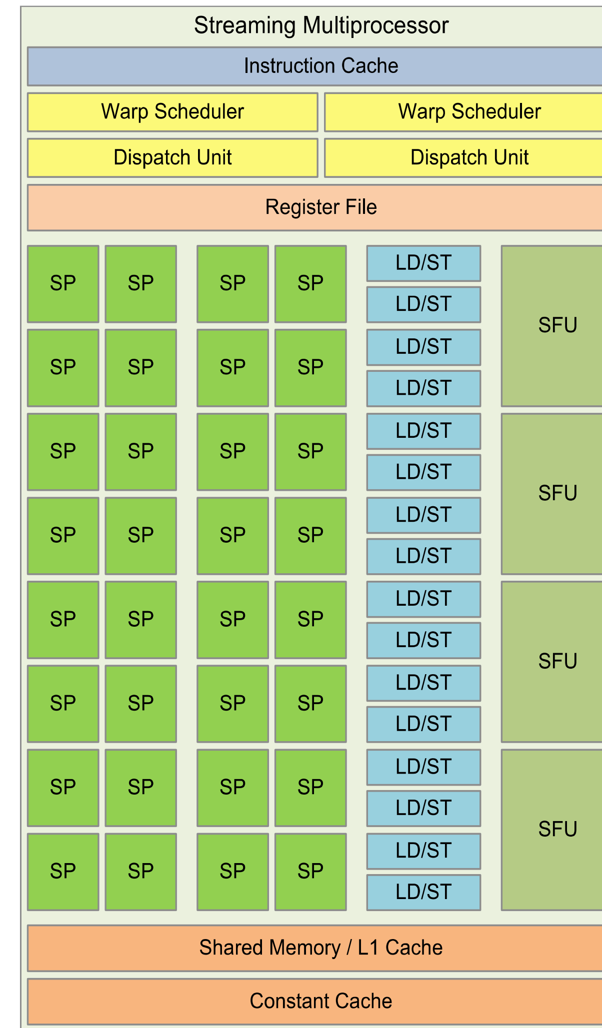
CPU Program	GPU Program

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);
}
	__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 cores: SIMD pipelines
  - Streaming Multiprocessors (SM)
  - Streaming Processors (SP)
- Blocks are divided into warps
  - SIMD 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

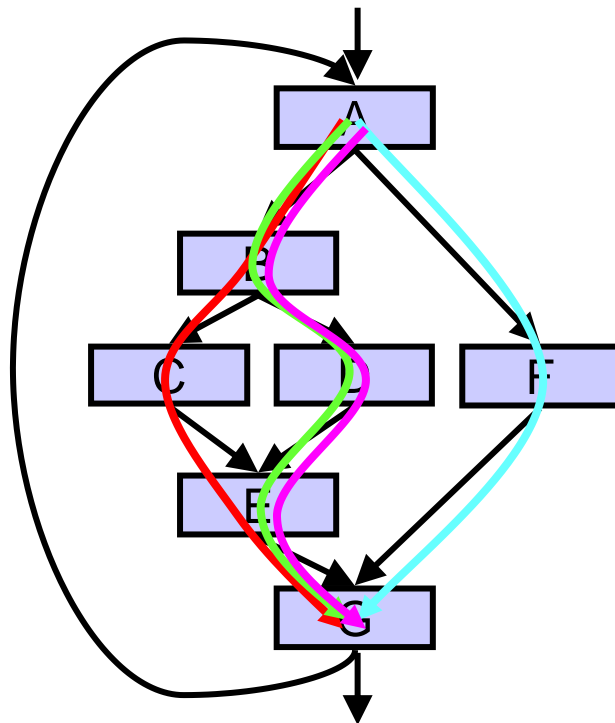
- SIMD: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

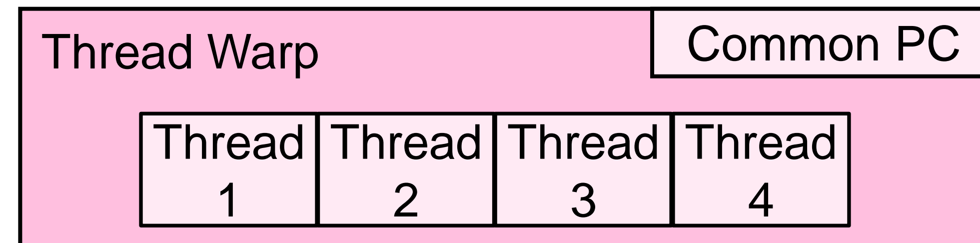
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- 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

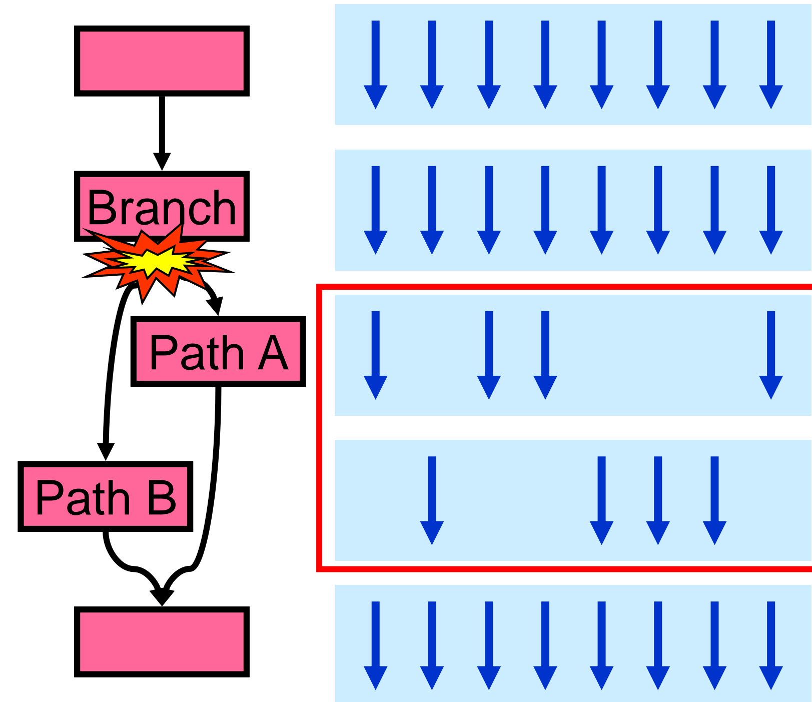




Control Flow Problem in GPUs/SIMT

- 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?

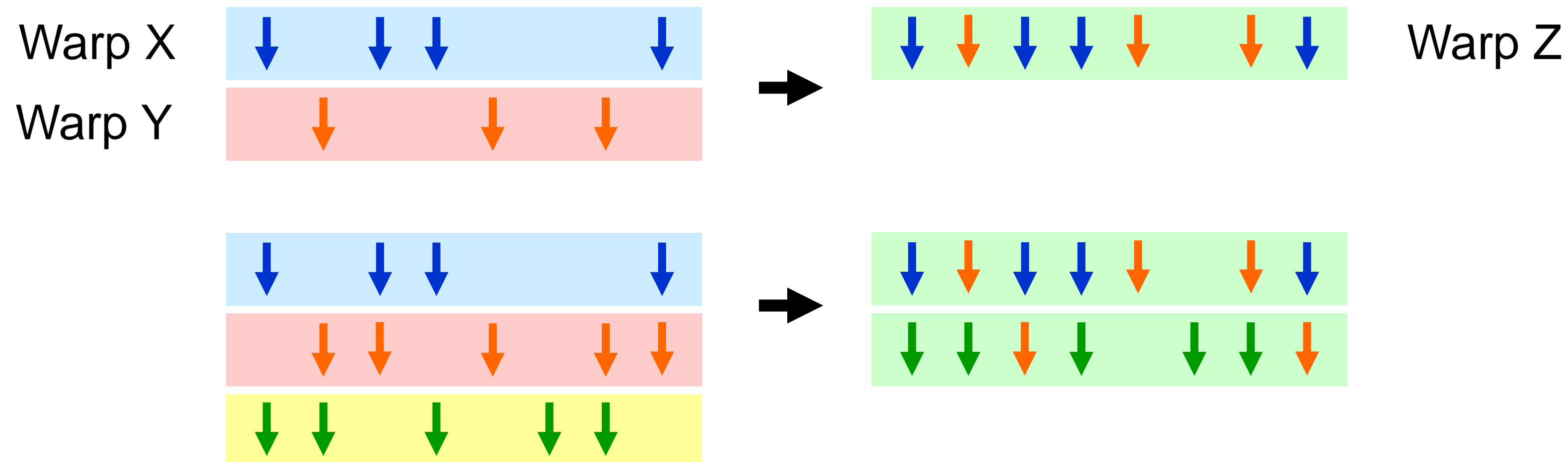
Remember: Each Thread Is Independent

- 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)

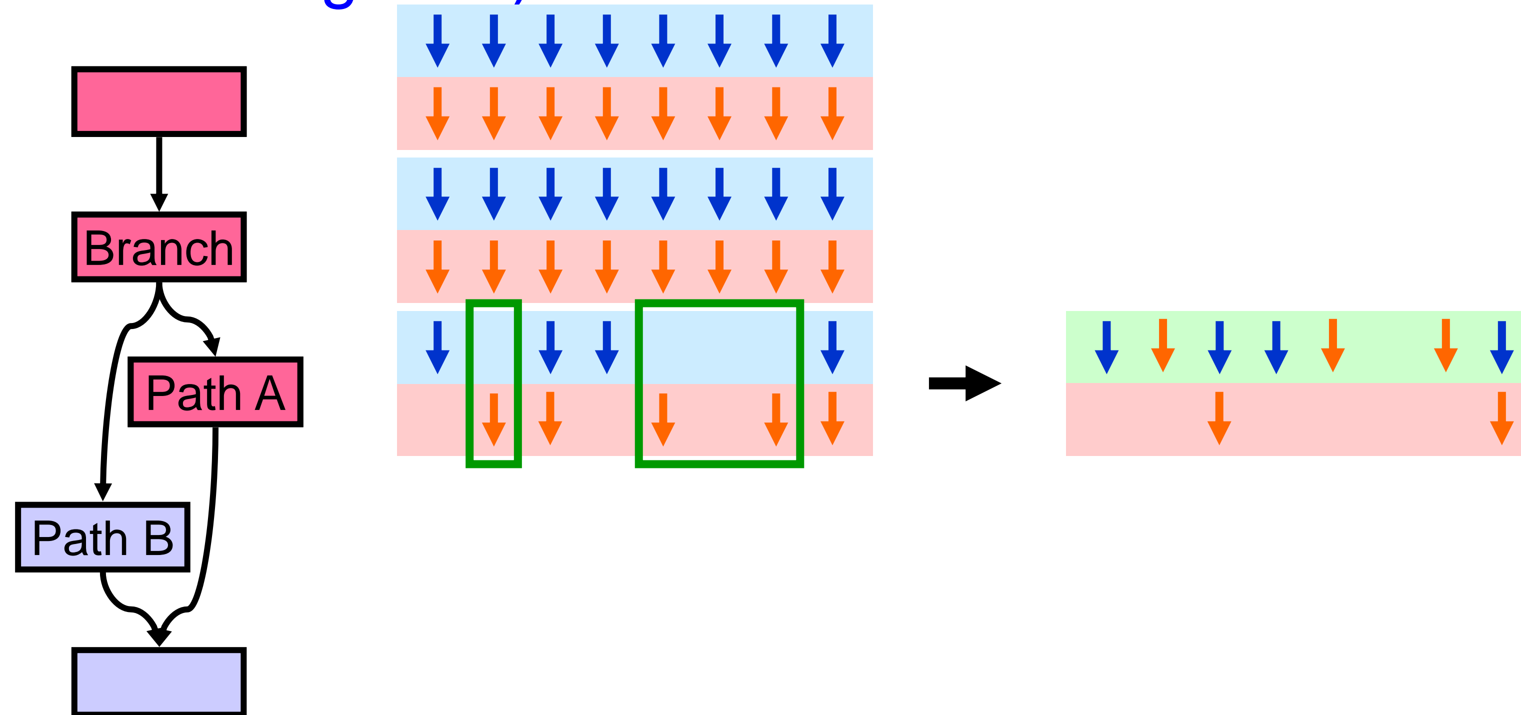
Dynamic Warp Formation/Merging

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



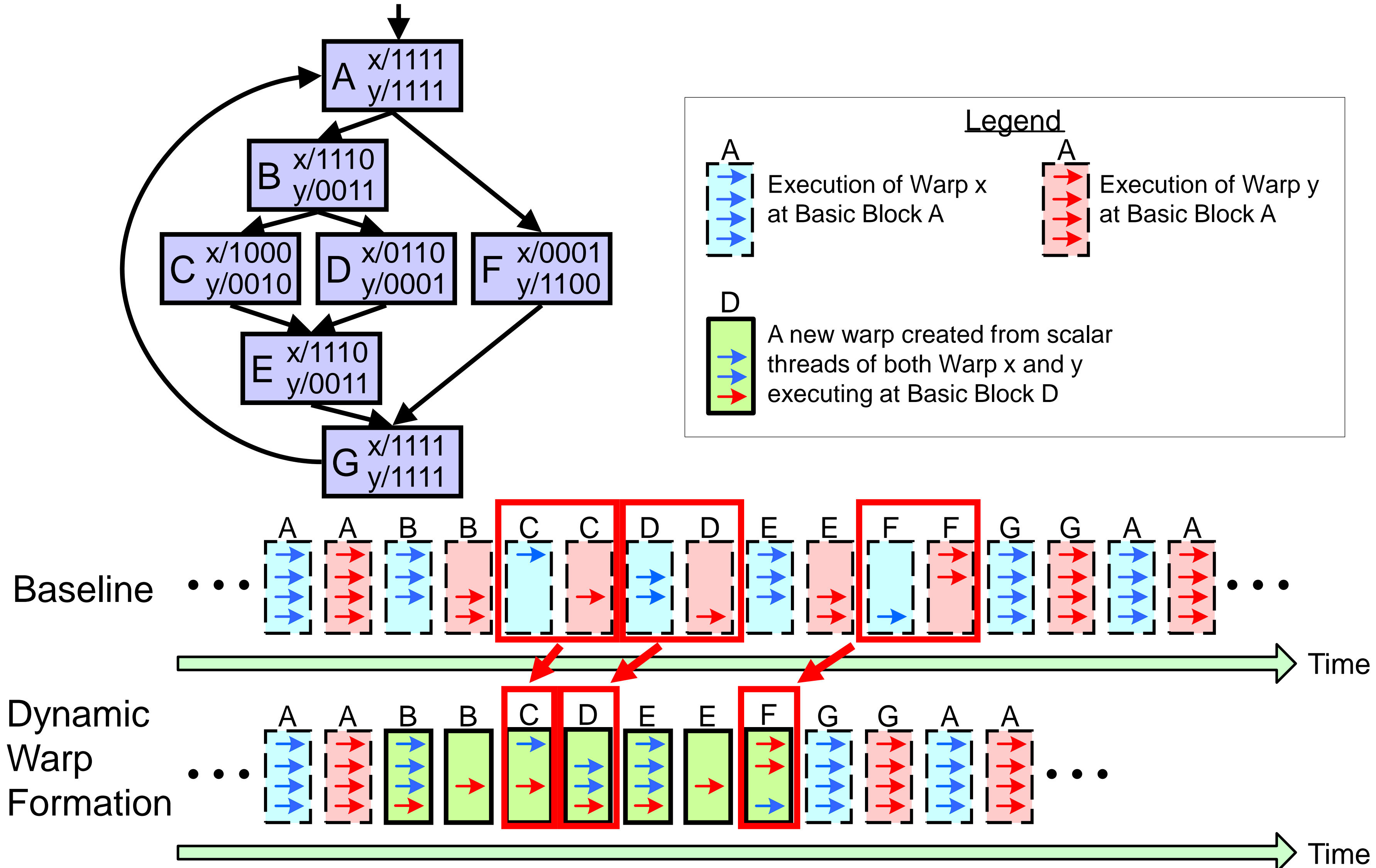
Dynamic Warp Formation/Merging

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

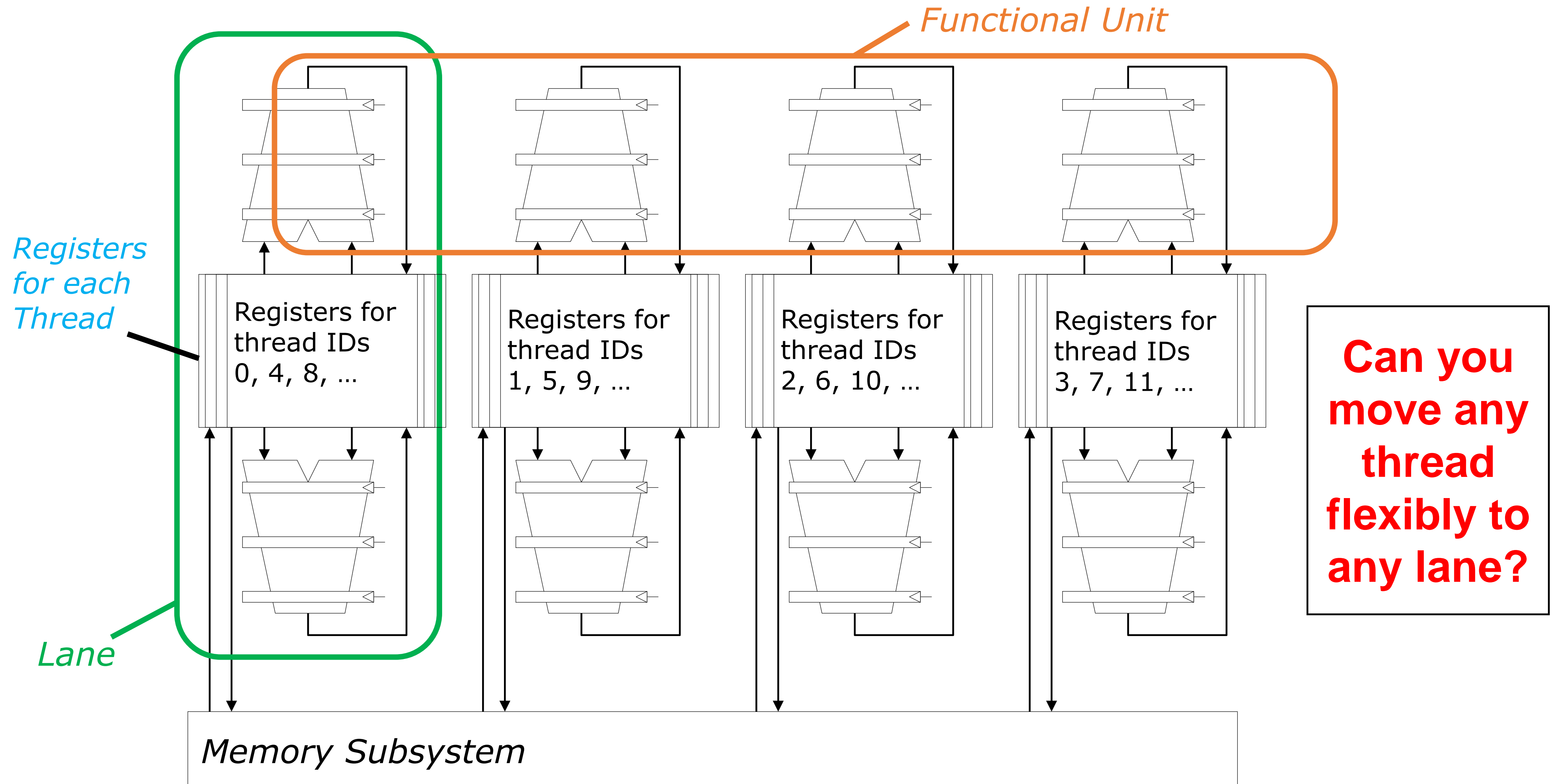


Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

Dynamic Warp Formation Example

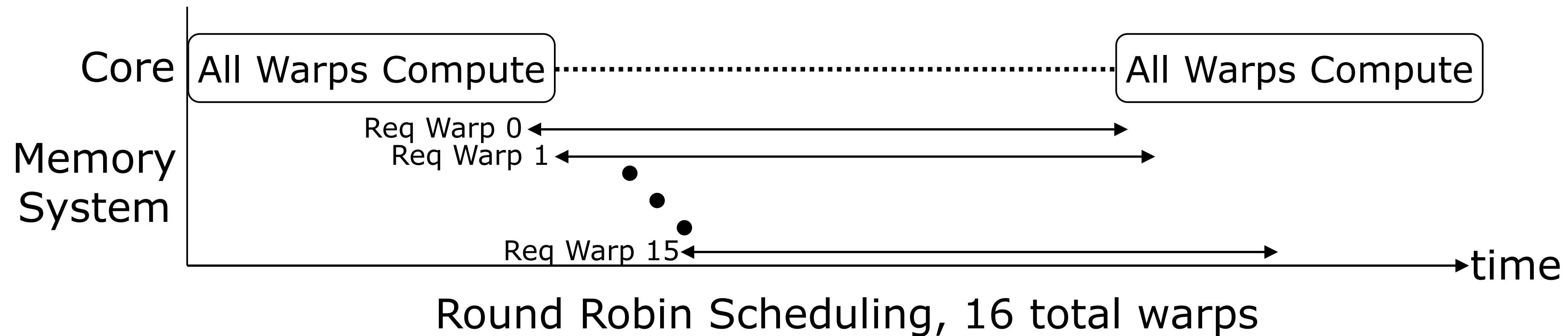


Hardware Constraints Limit Flexibility of Warp Grouping



Large Warps and Two-Level Warp Scheduling

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



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

Large Warp Microarchitecture Example

- Reduce branch divergence by having large warps
- Dynamically break down a large warp into sub-warps

Decode Stage

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Sub-warp 2 mask

1	1	1	1

Sub-warp 1 mask

1	1	1	1

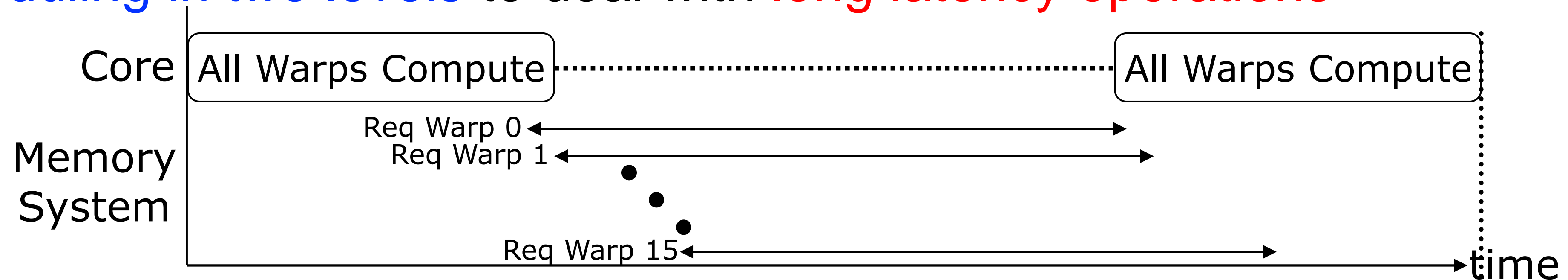
Sub-warp 0 mask

1	1	1	1

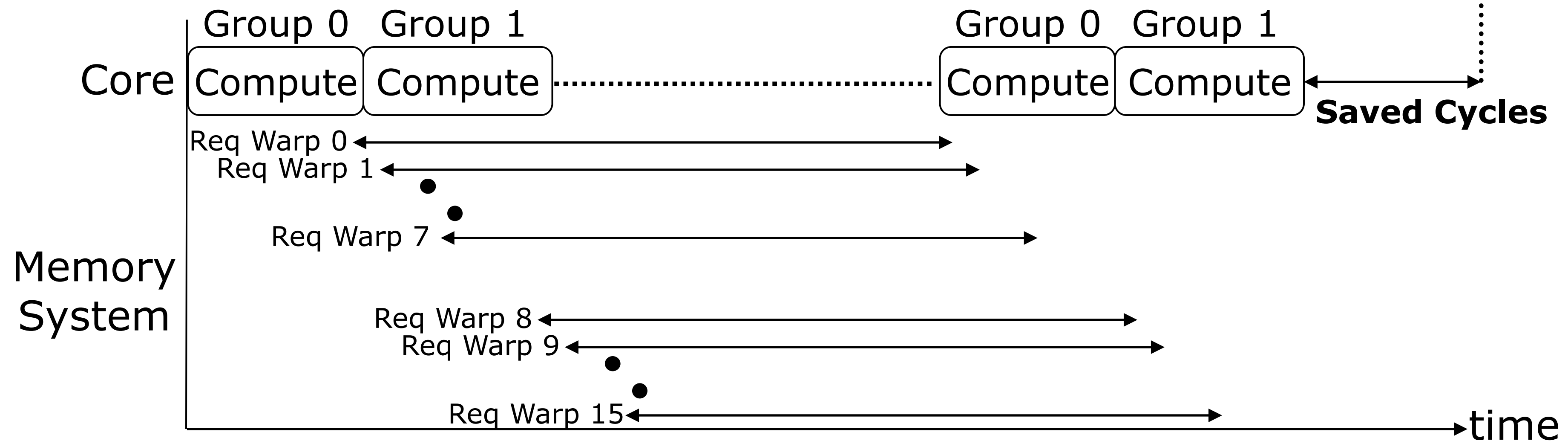
Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.

Two-Level Round Robin

Scheduling in two levels to deal with long latency operations



Round Robin Scheduling, 16 total warps

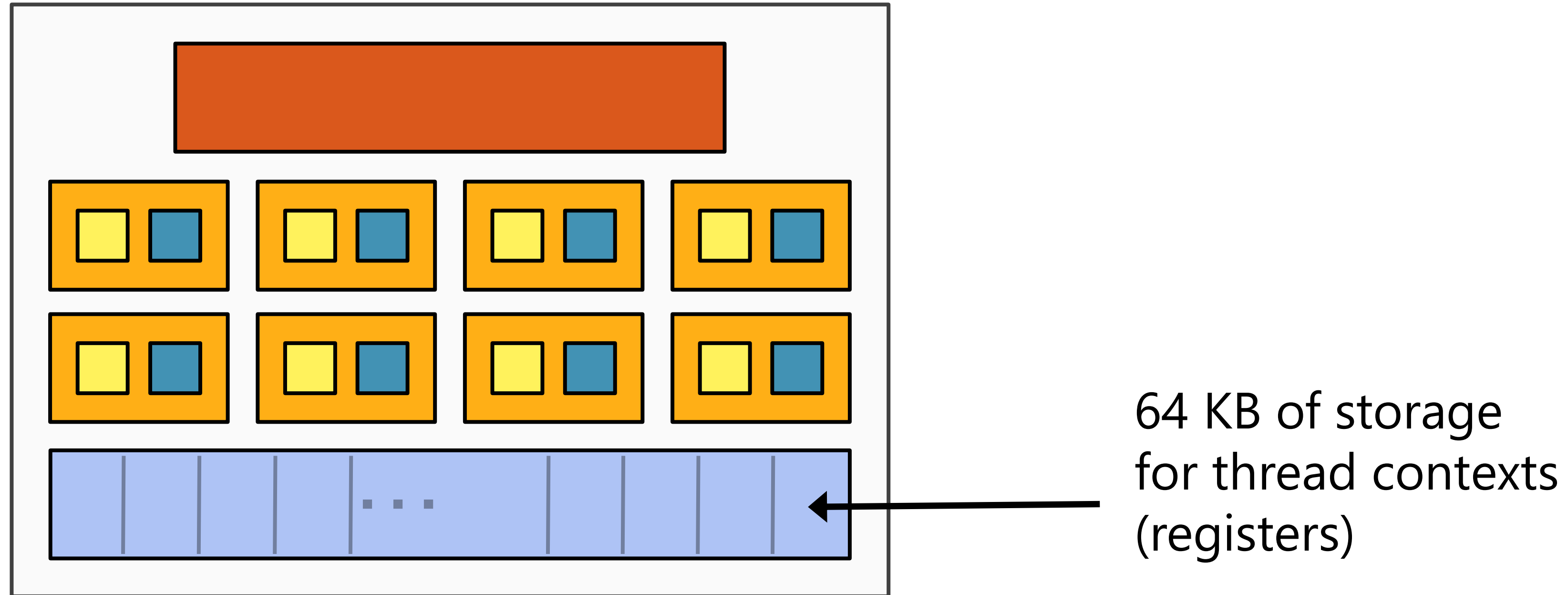


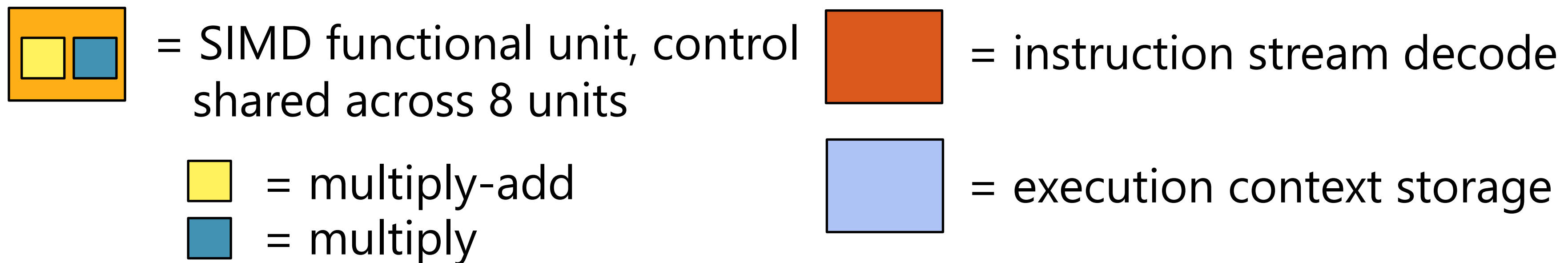
Two Level Round Robin Scheduling, 2 fetch groups, 8 warps each

NVIDIA GeForce GTX 285

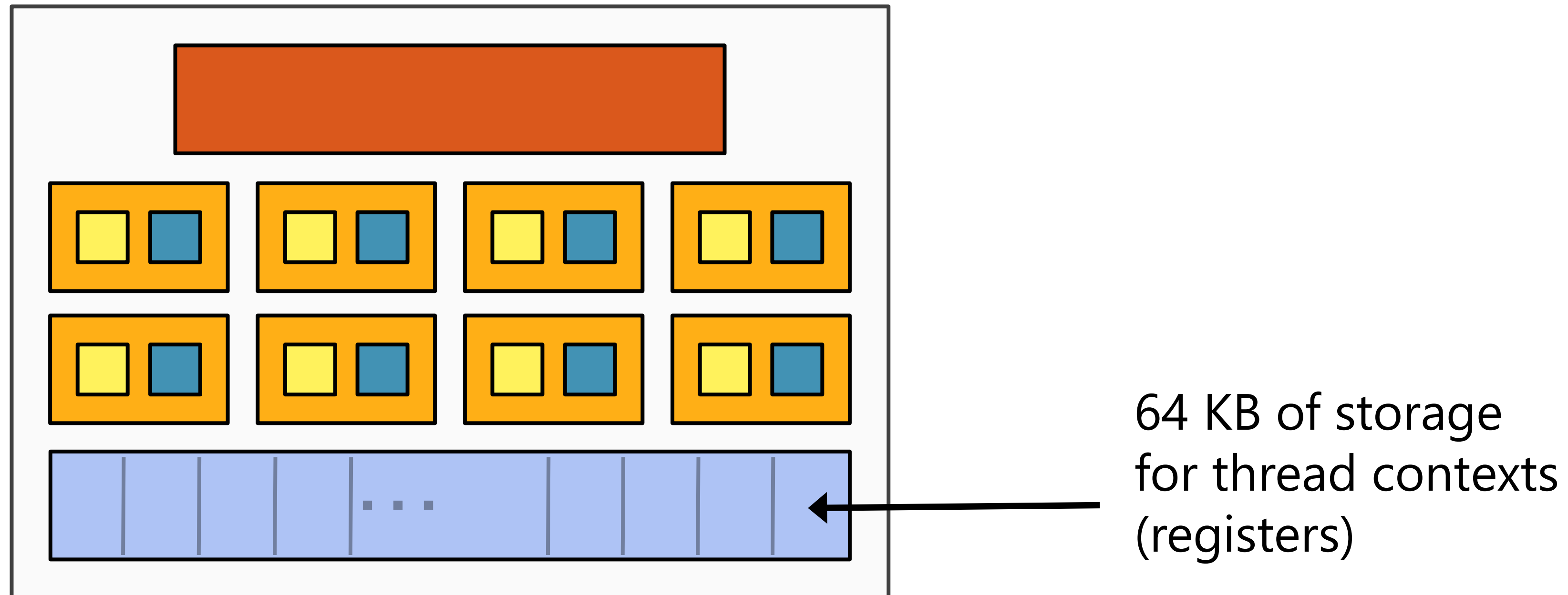
- NVIDIA-terminology:
  - 240 stream processors
  - "SIMT execution"
- Generic classification:
  - 30 cores
  - 8 SIMD functional units per core

NVIDIA GeForce GTX 285 "core"



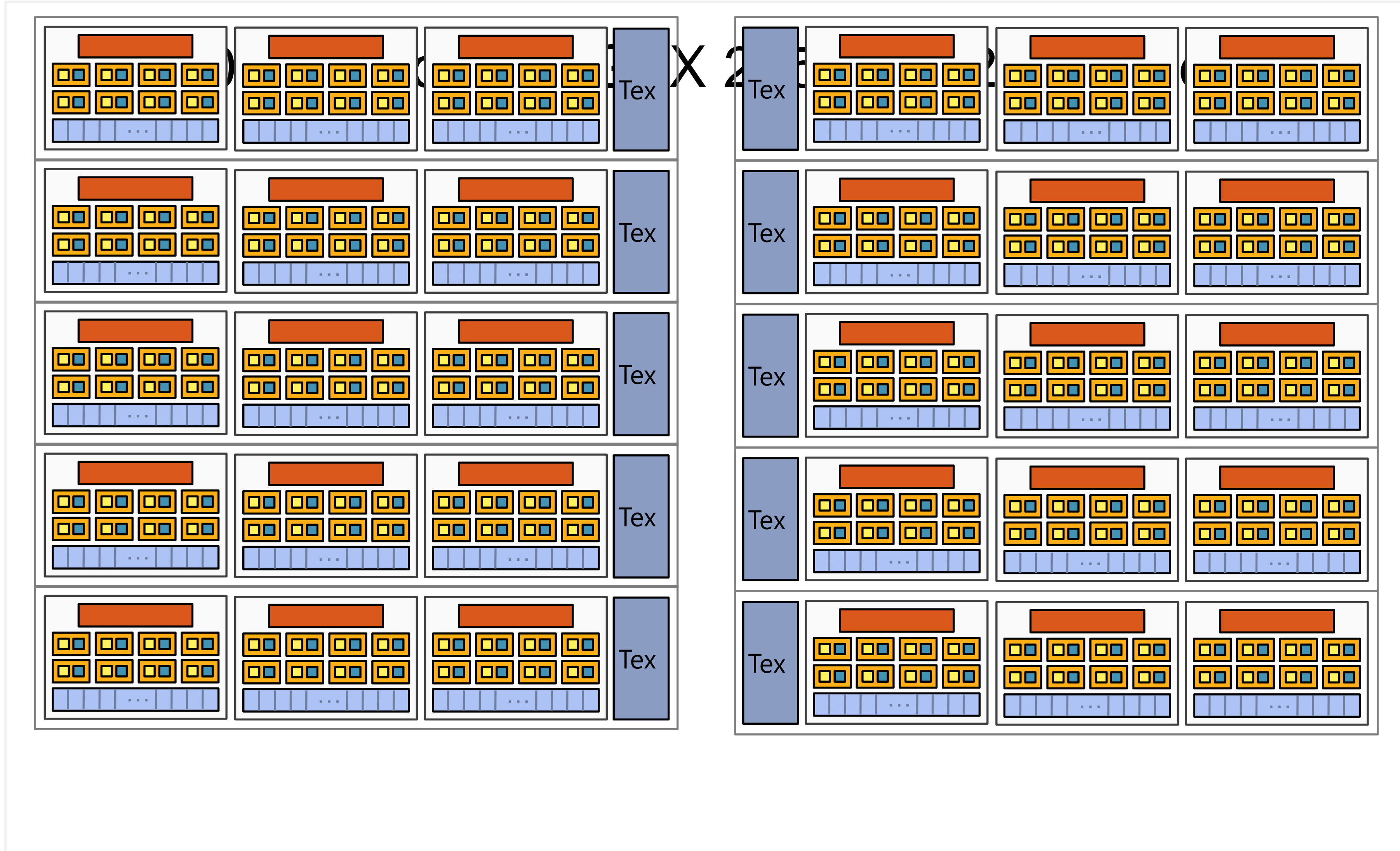


NVIDIA GeForce GTX 285 “core”

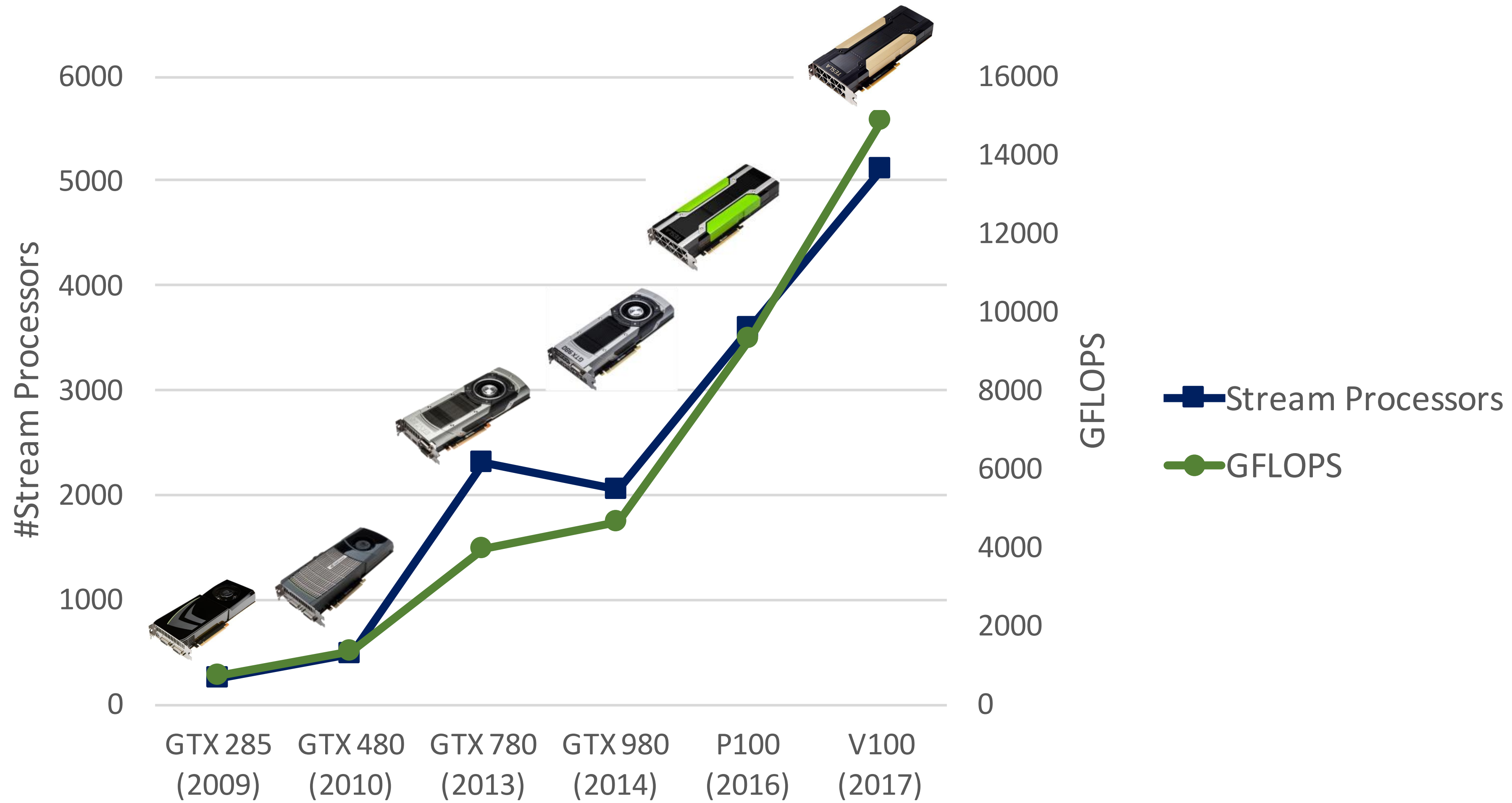


- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



Evolution of NVIDIA GPUs



NVIDIA V100

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

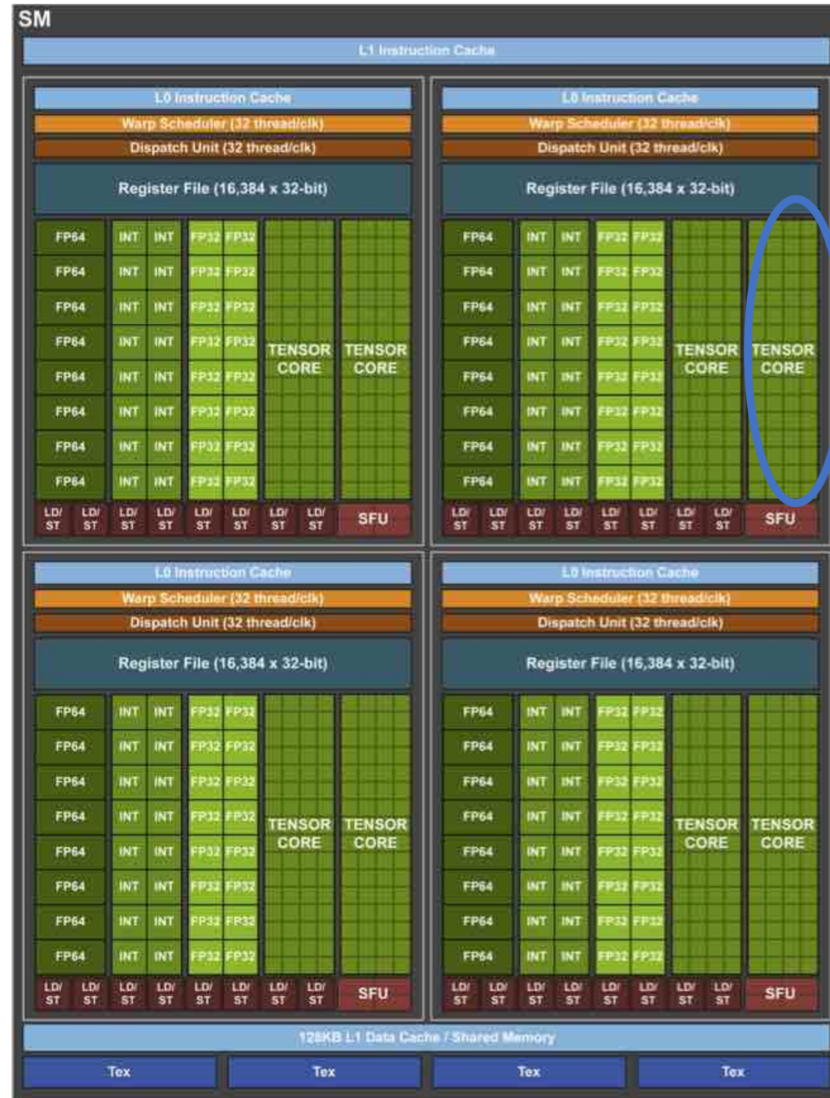
- Generic classification:
  - 80 cores
  - 64 SIMD functional units per core
  - Tensor cores for Machine Learning

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

NVIDIA V100 Block Diagram

80 cores on the V100

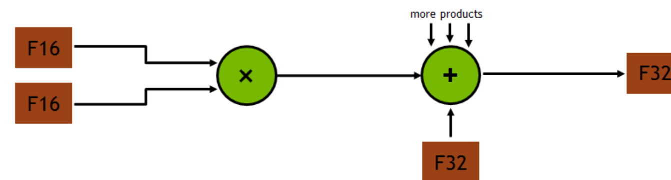
NVIDIA V100 Core



15.7 TFLOPS Single Precision

7.8 TFLOPS Double Precision

125 TFLOPS for Deep Learning (Tensor cores)



$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 — FP16 — FP16 — FP16 or FP32

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