Computer Architecture

Lecture 9: GPUs and GPGPU Programming

Prof. Onur Mutlu
ETH Zürich

Fall 2017

19 October 2017

Agenda for Today

- GPUs

- Introduction to GPU Programming

Digitaltechnik (Spring 2017) YouTube videos
Lecture 21: GPUs
https://youtu.be/MUPTdxl3JKs?t=23m17s

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)

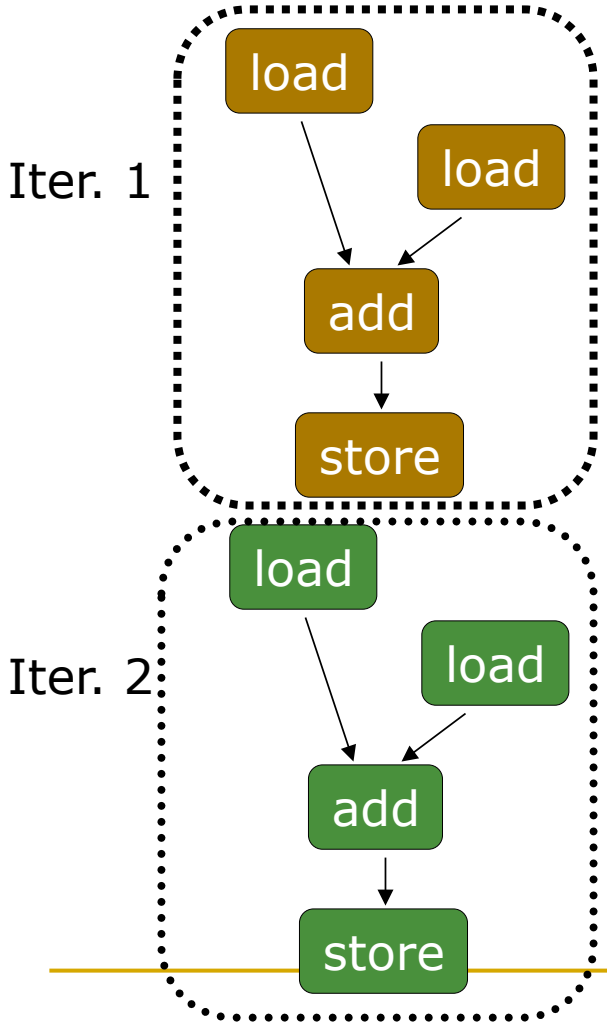
Programming Model vs. Hardware Execution Model

- 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:

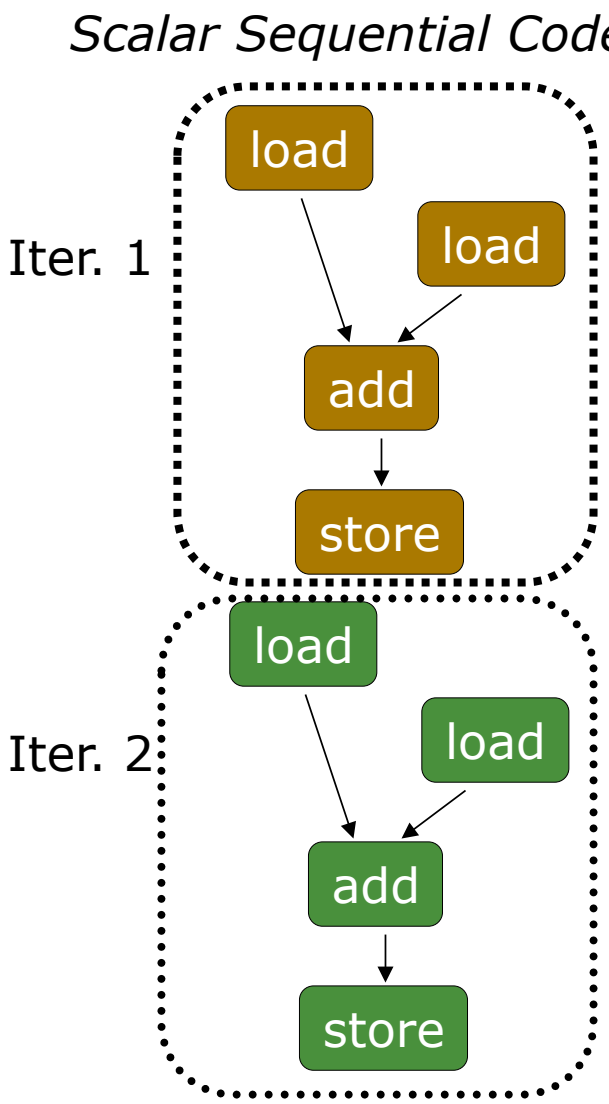
1. Sequential (SISD)

2. Data-Parallel (SIMD)

3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

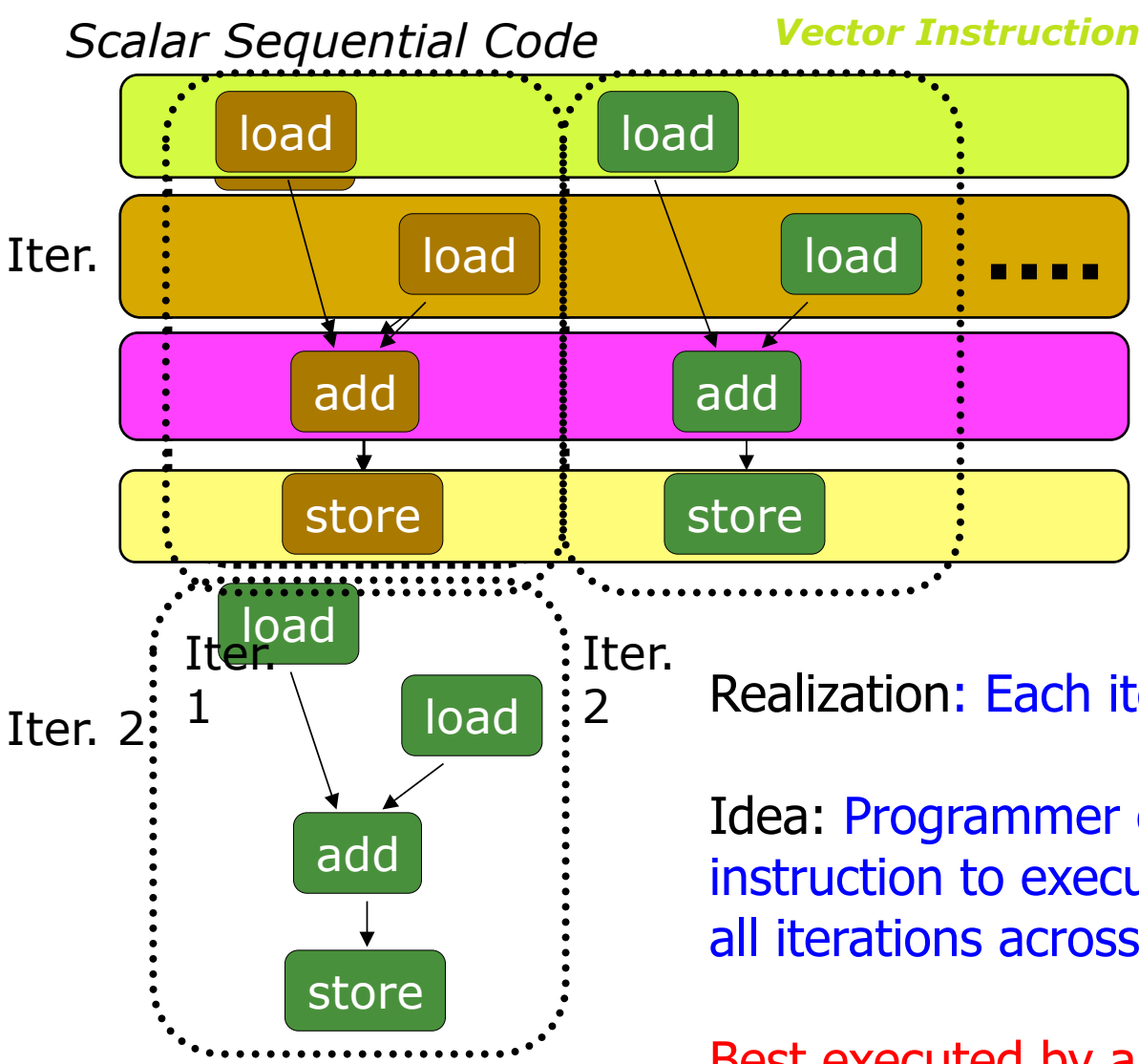
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



- 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

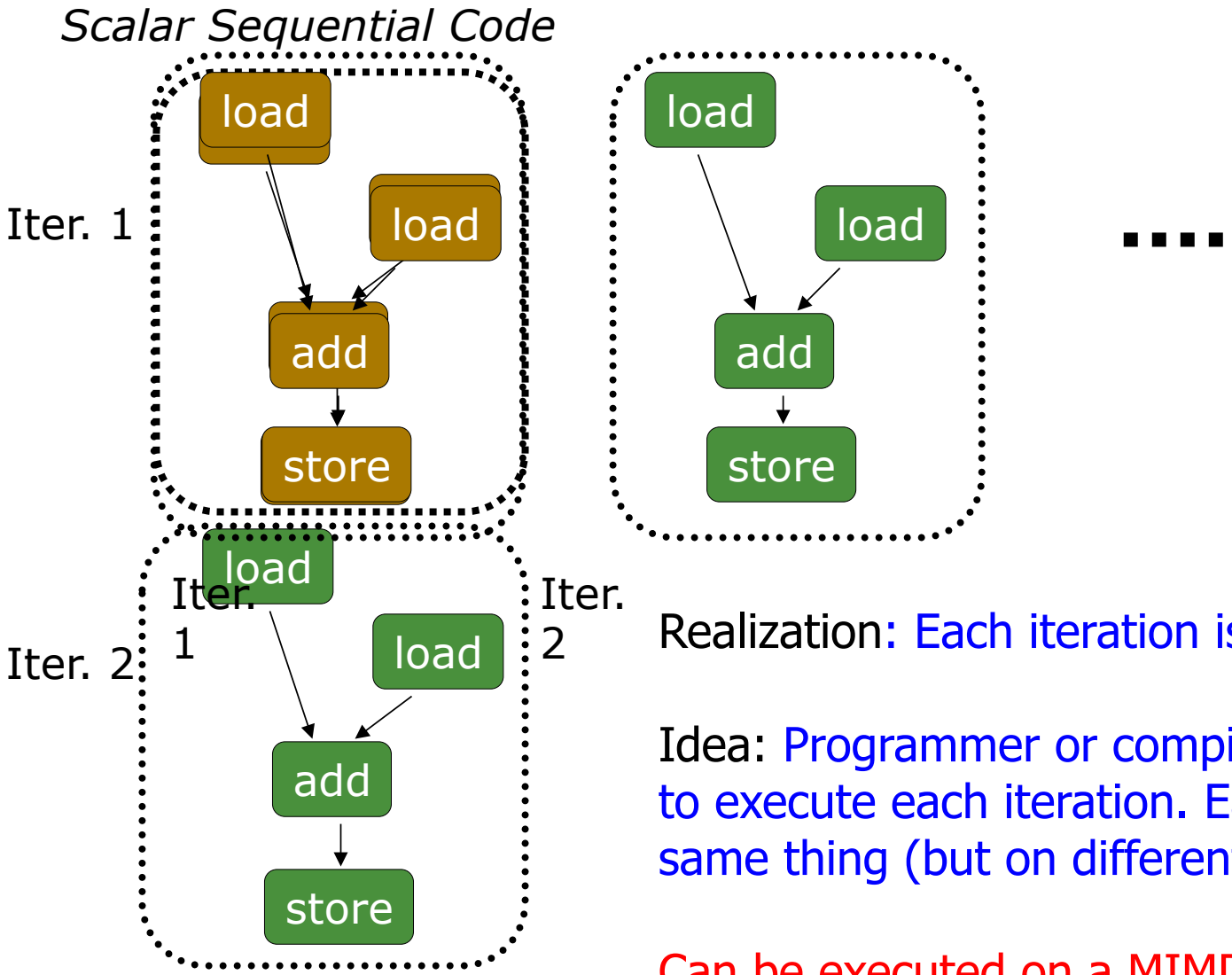
Realization: Each iteration is independent

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];
```



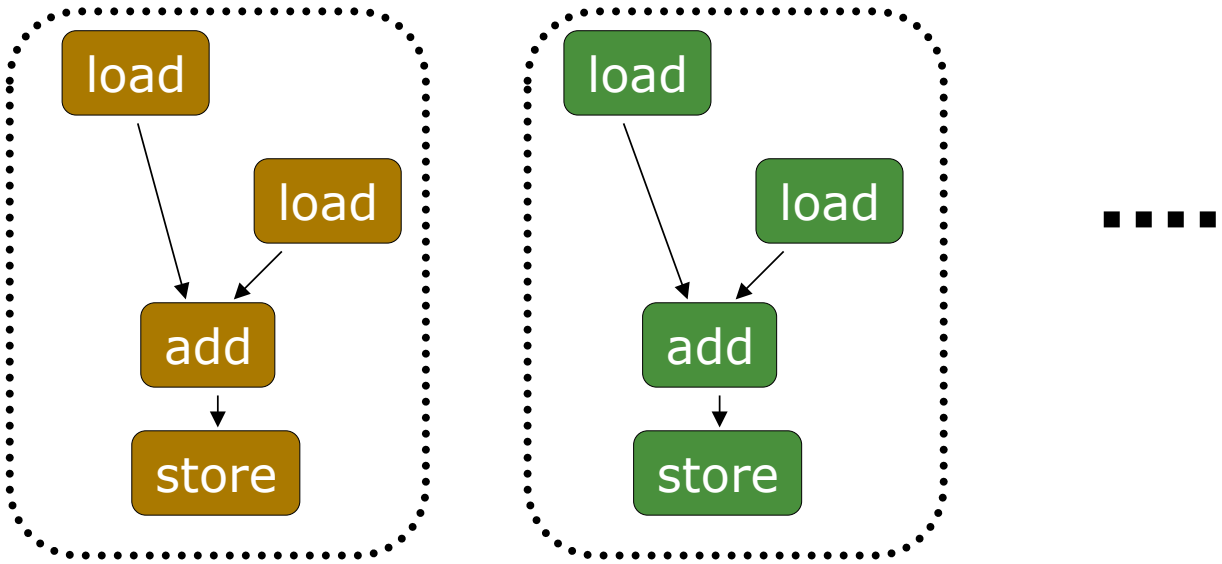
Realization: Each iteration is independent

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

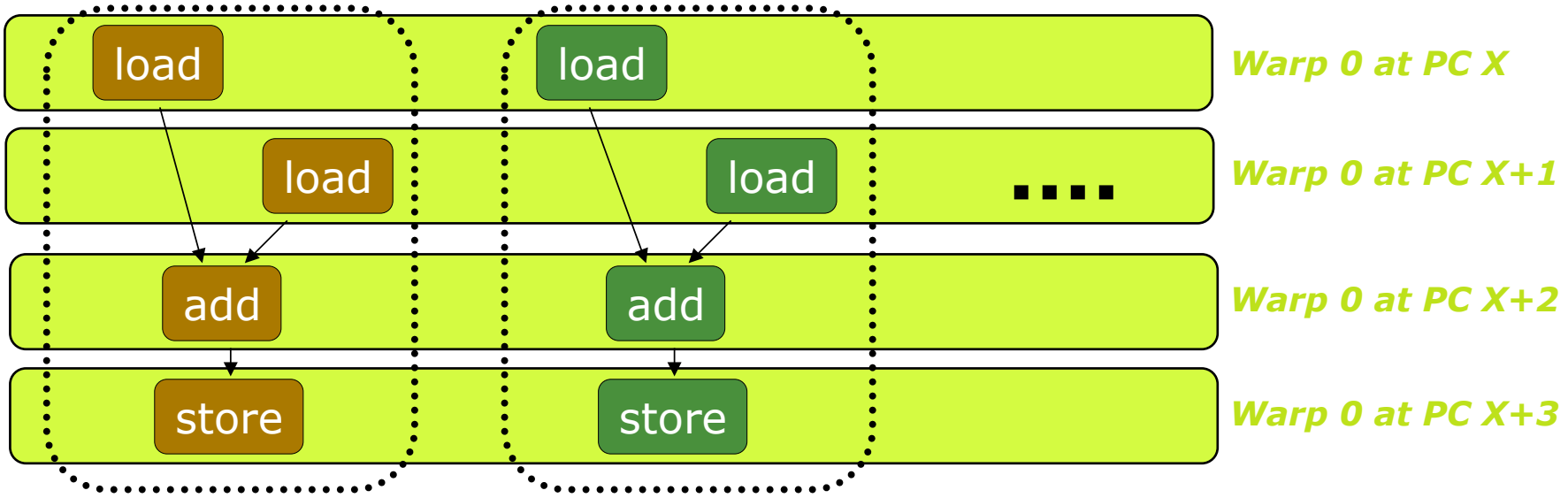
- Except it is **not** programmed using SIMD instructions

- 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

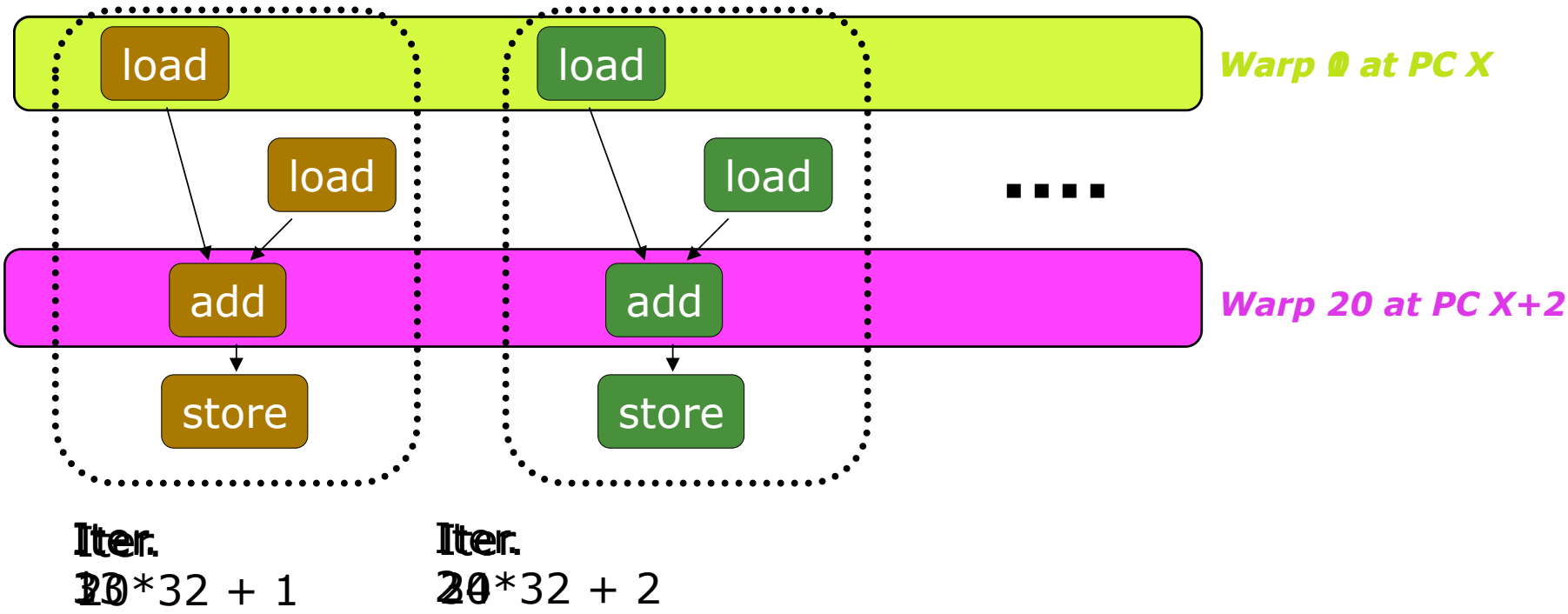
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- 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

Multithreading of Warps

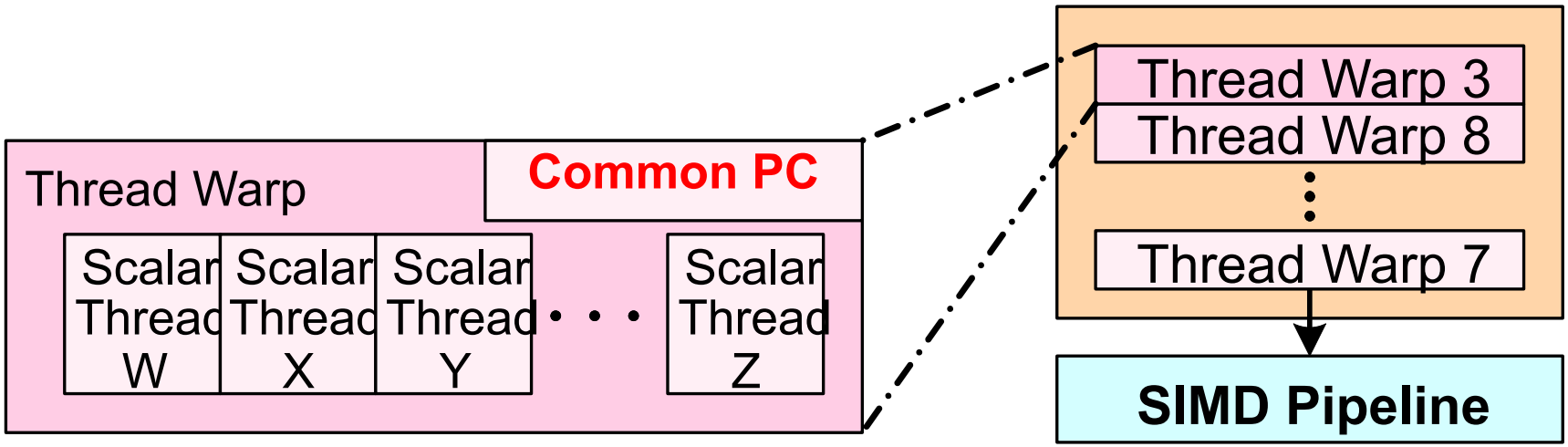
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

- 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

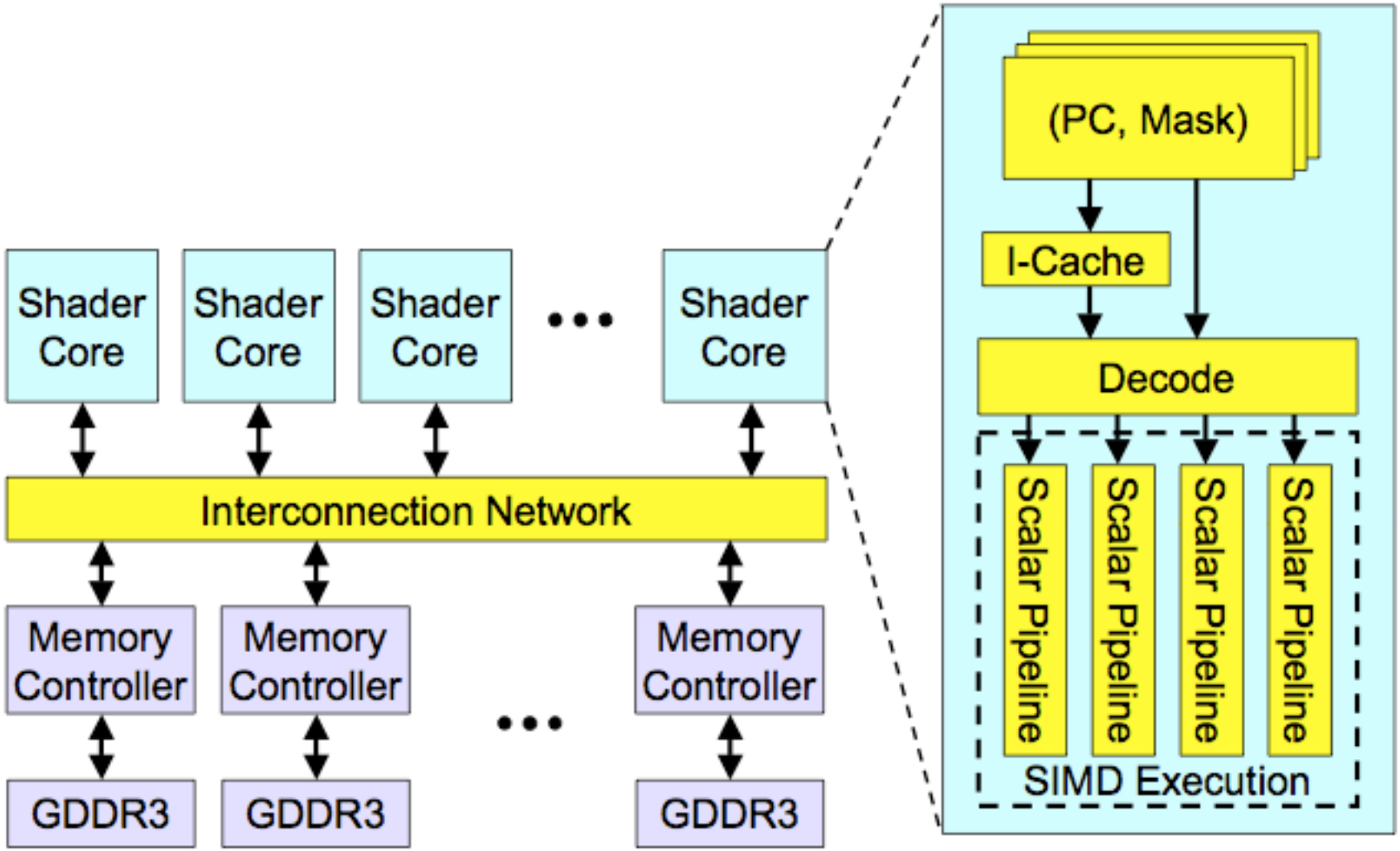


Warps and Warp-Level FGMT

- 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 …

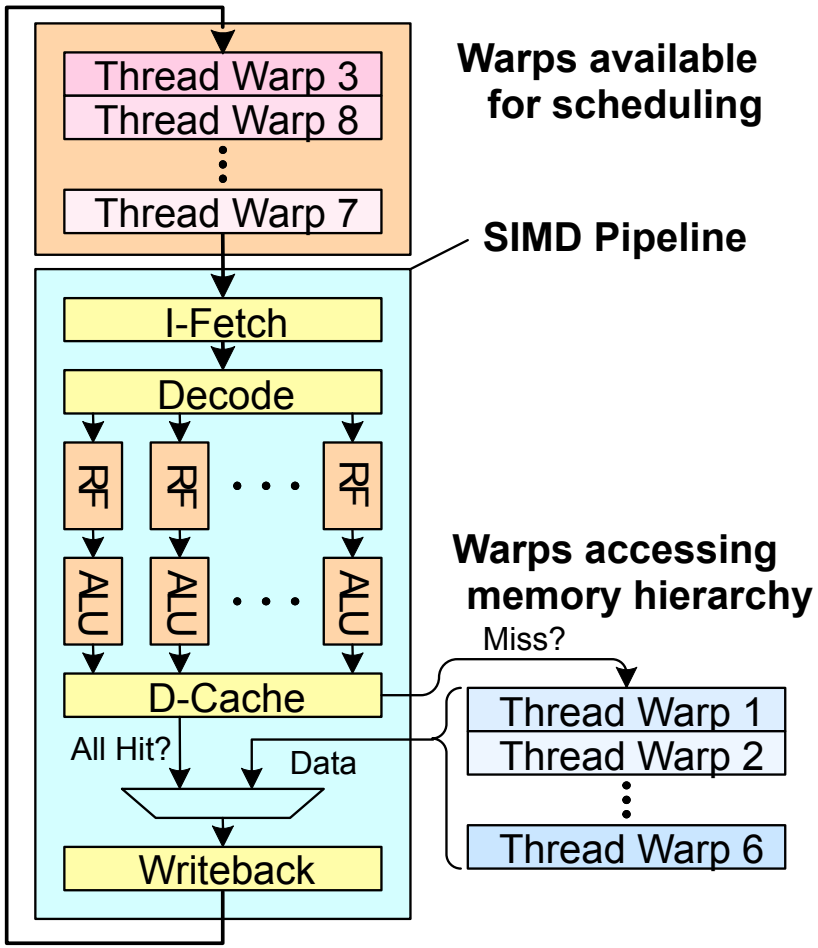


High-Level View of a GPU

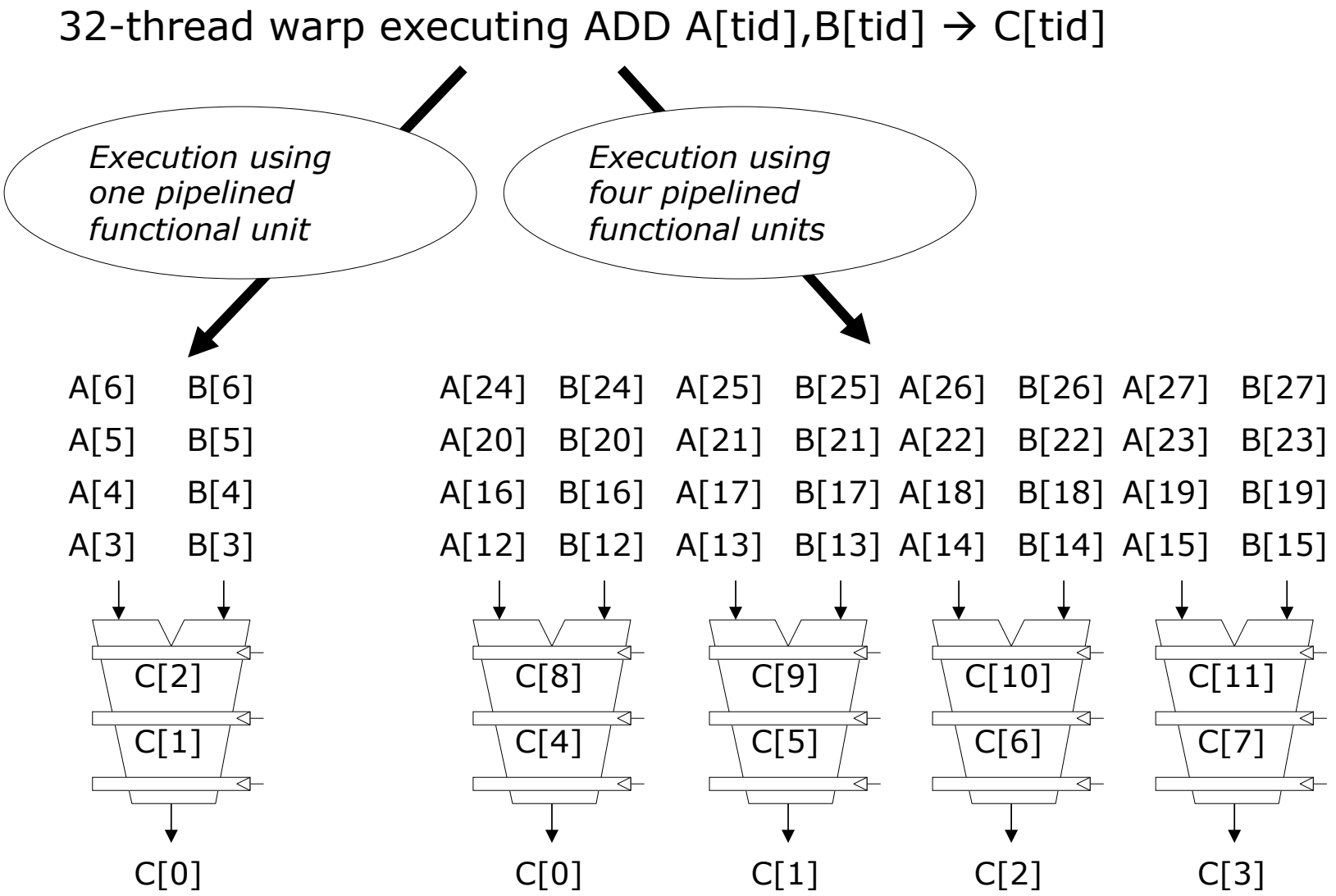


Latency Hiding via Warp-Level FGMT

- 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 long latency tolerance
  - Millions of pixels

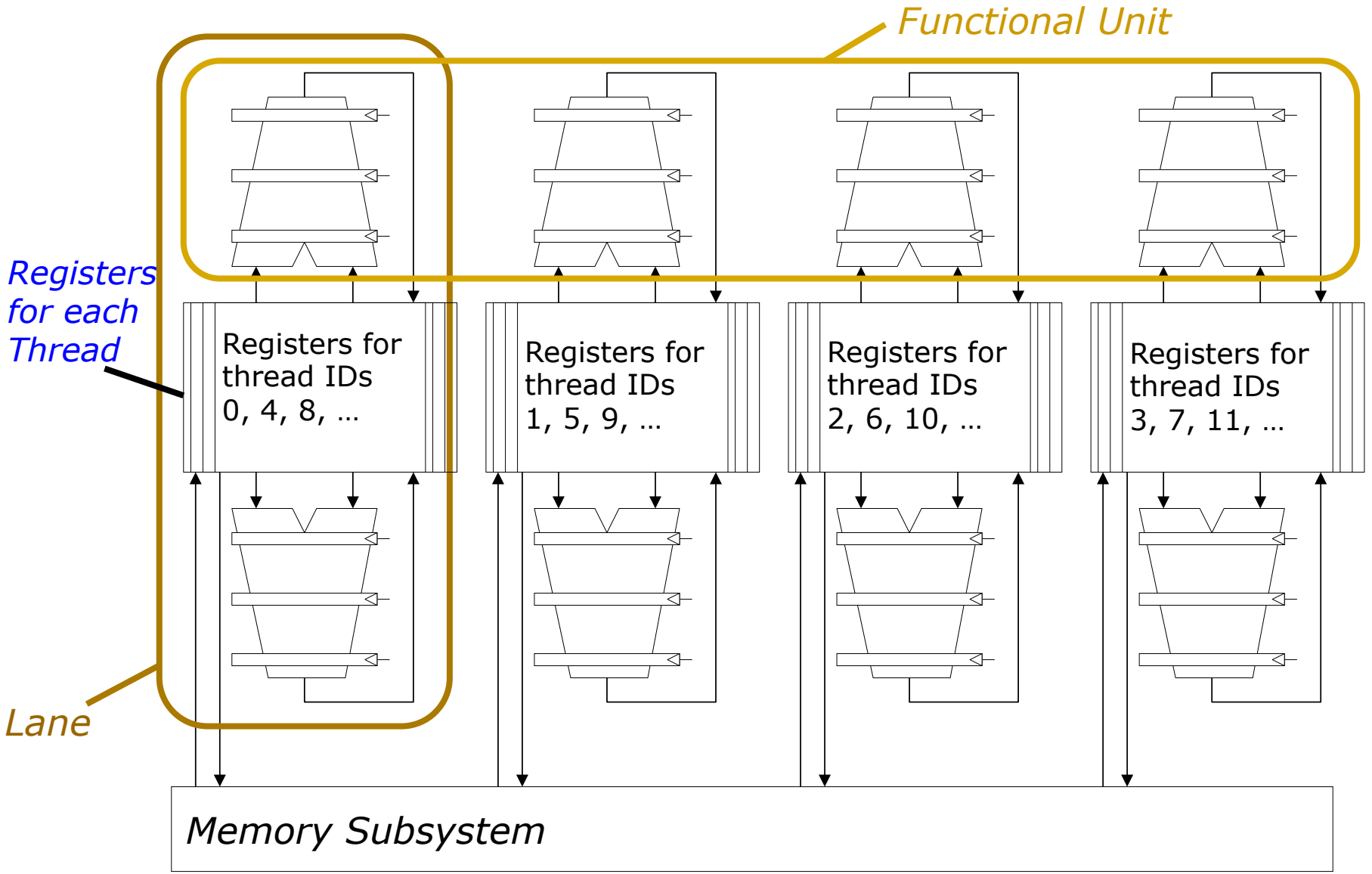


Warp Execution (Recall the Slide)



Slide credit: Krste Asanovic

SIMD Execution Unit Structure

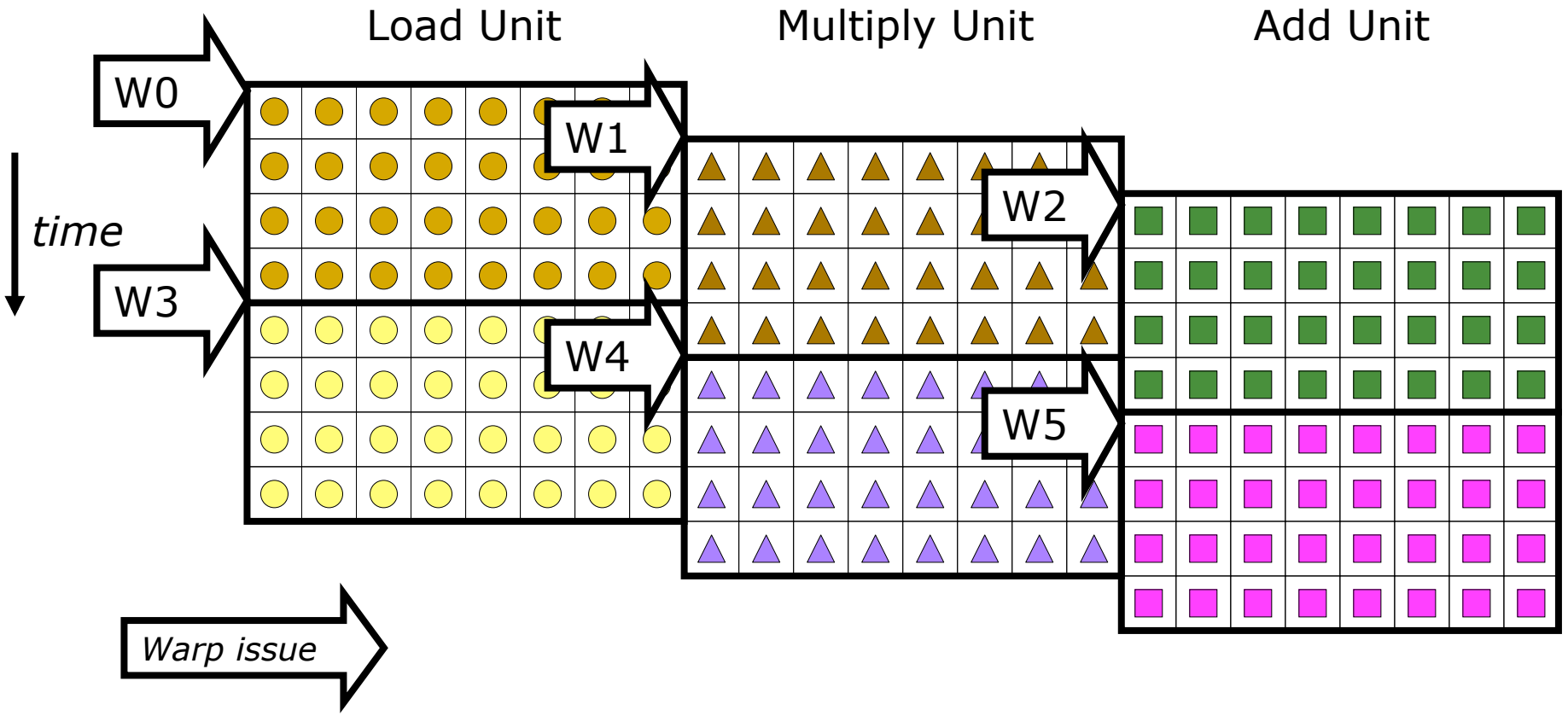


Slide credit: Krste Asanovic

Warp Instruction Level Parallelism

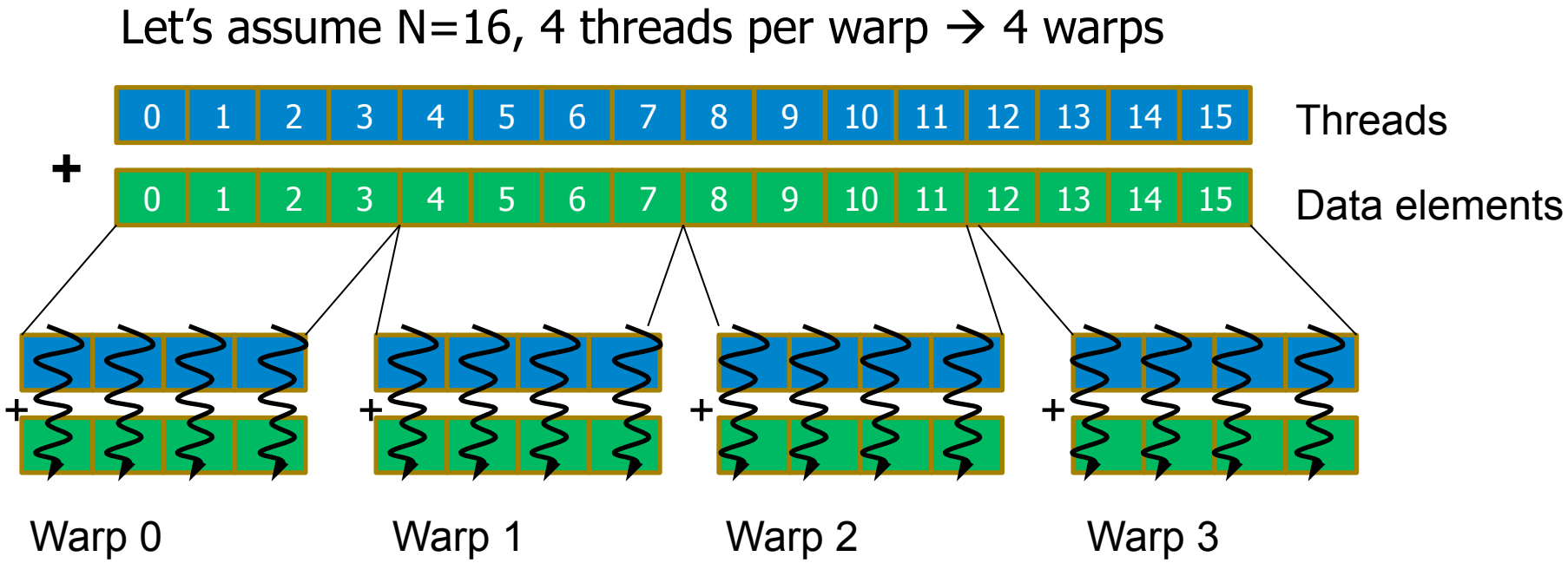
Can overlap execution of multiple instructions

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



SIMT Memory Access

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



Slide credit: Hyesoon Kim

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

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

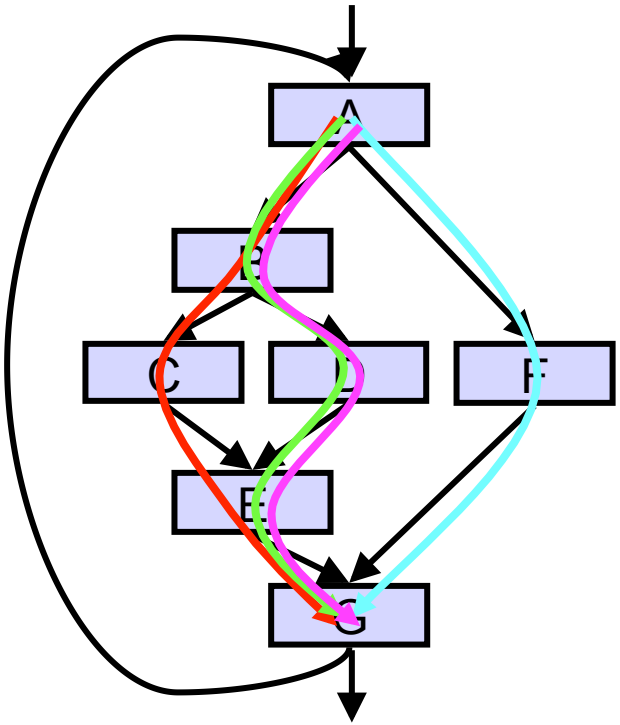
- SIMD: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

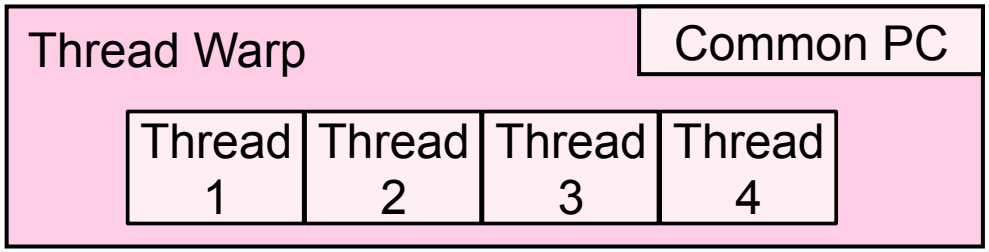
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- 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

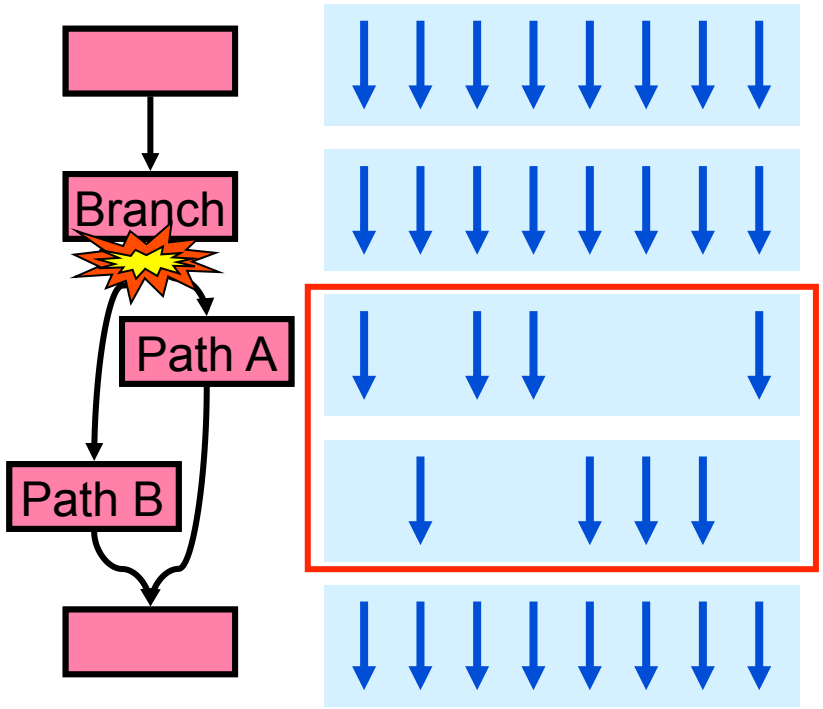




Control Flow Problem in GPUs/SIMT

- 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?

Slide credit: Tor Aamodt

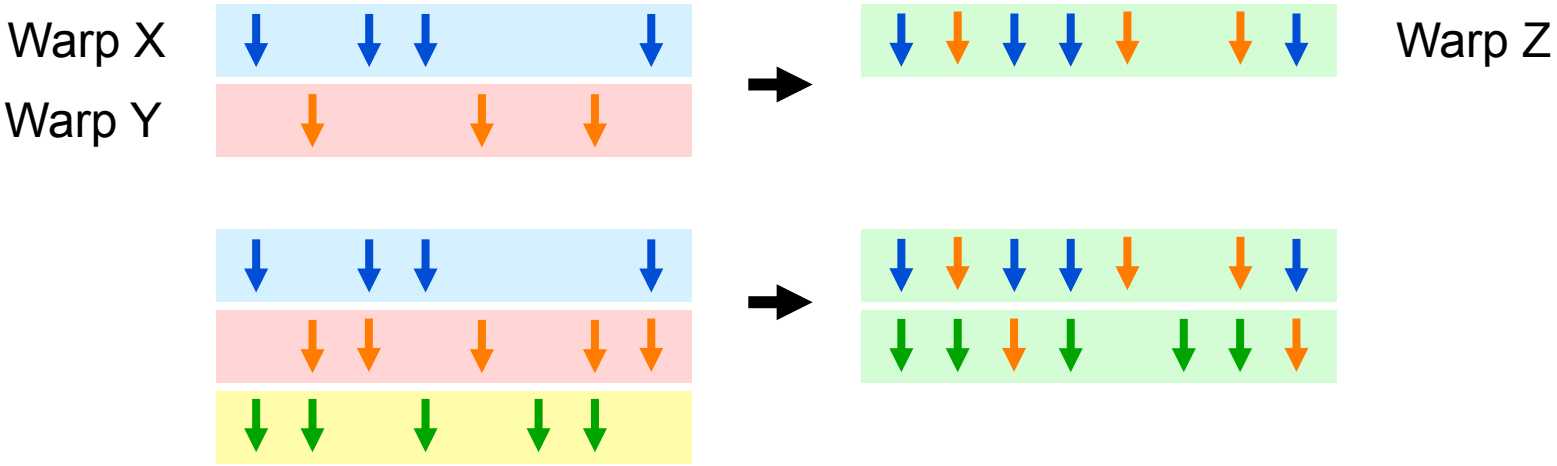
Remember: Each Thread Is Independent

- 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)

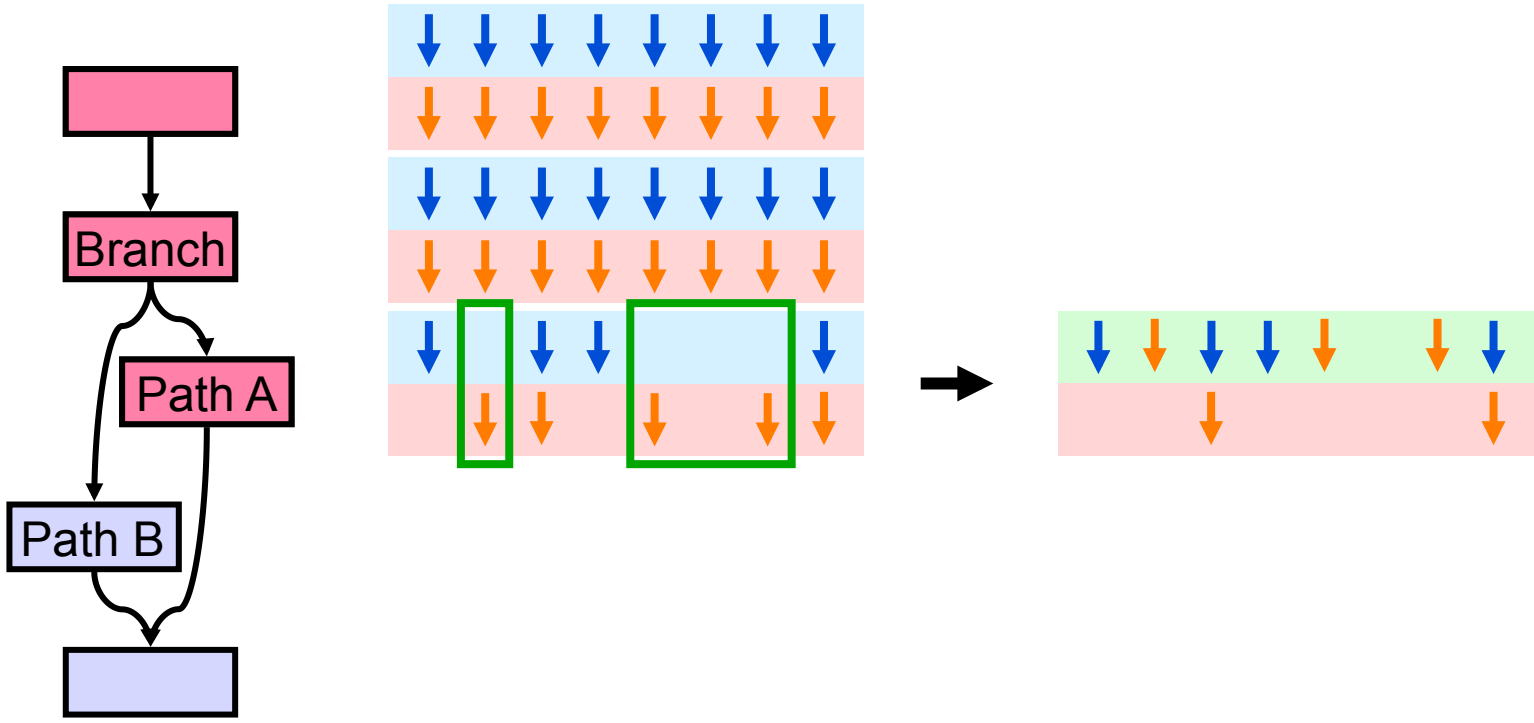
Dynamic Warp Formation/Merging

- 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



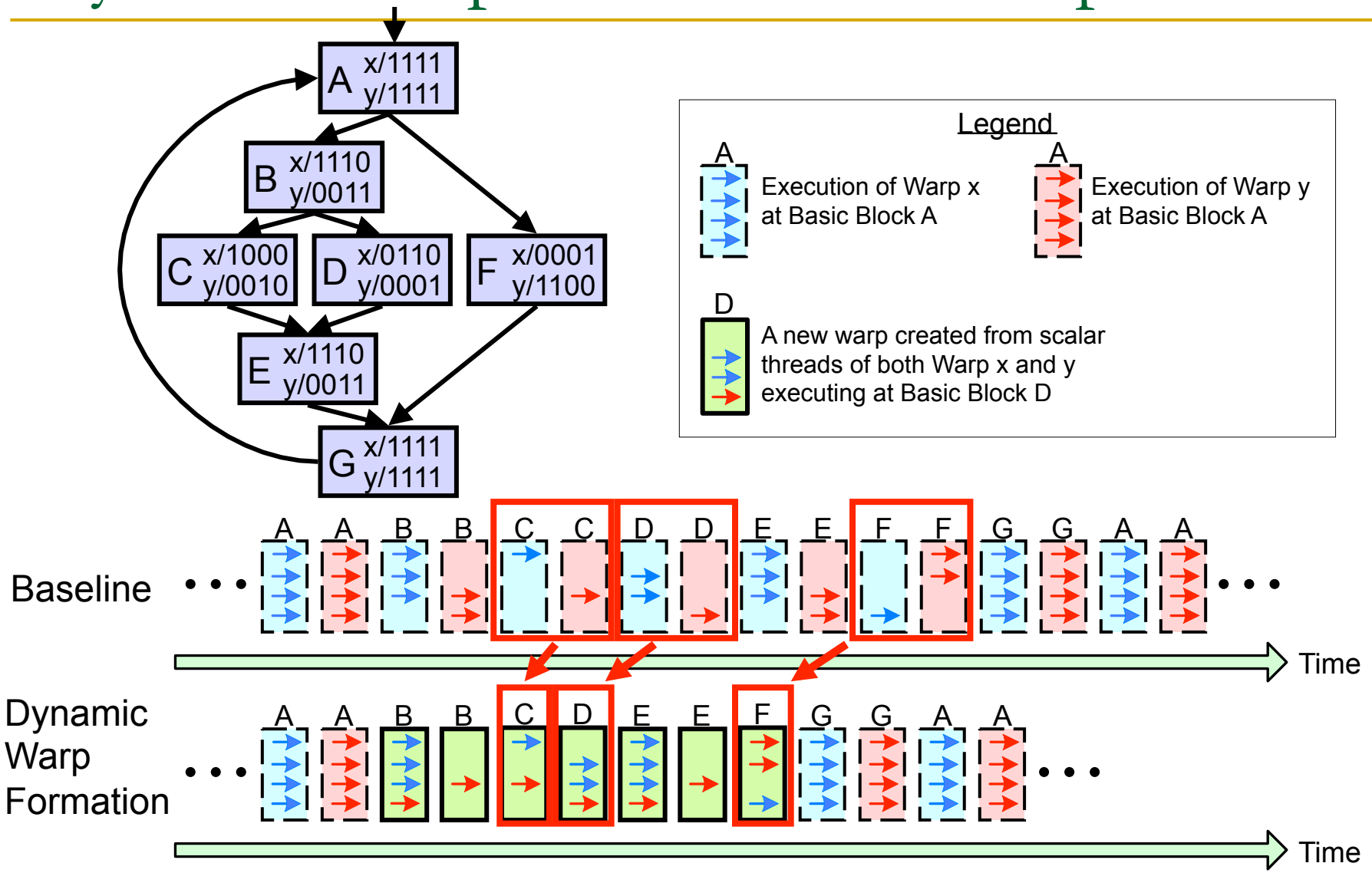
Dynamic Warp Formation/Merging

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

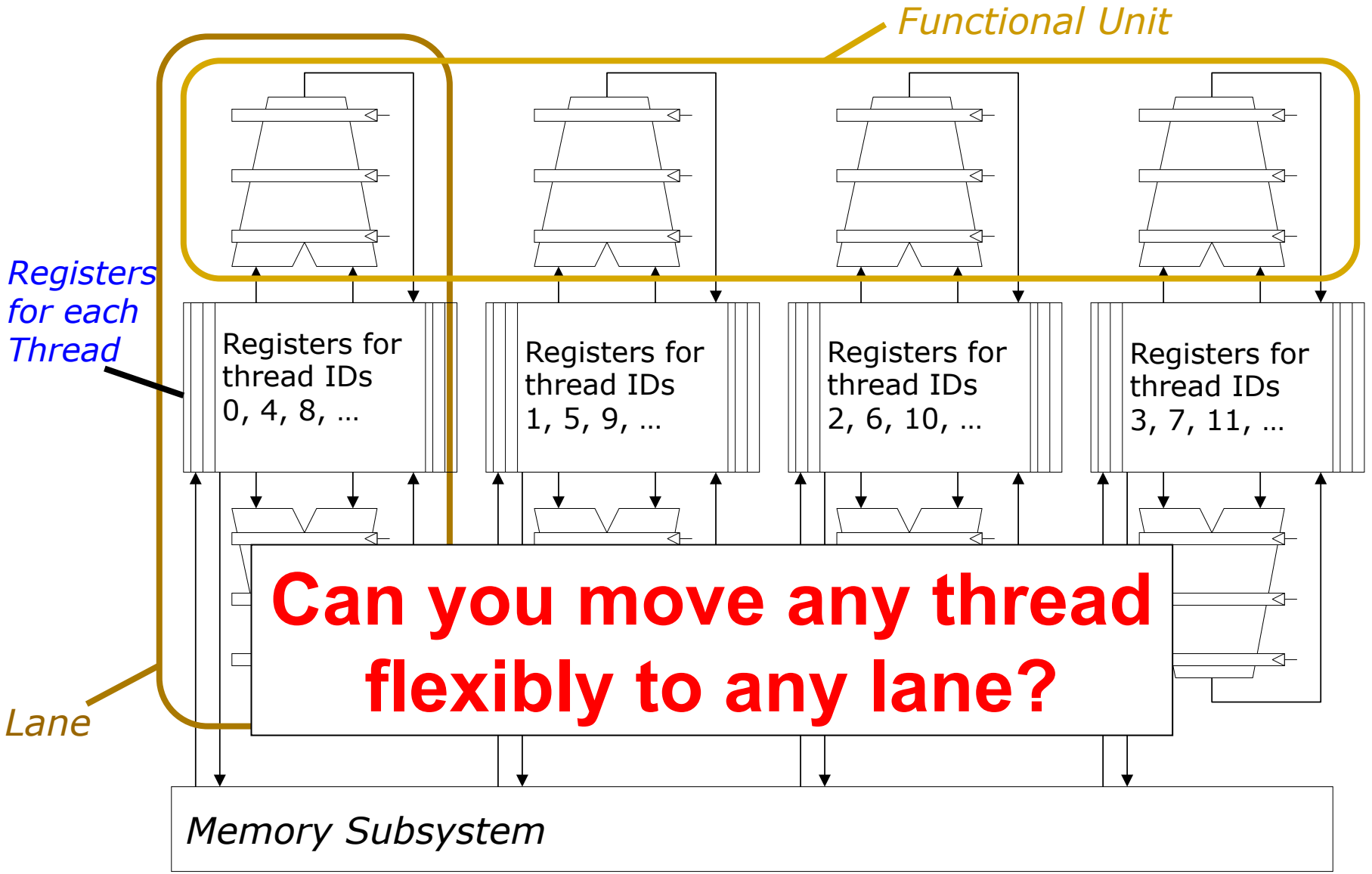


- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

Dynamic Warp Formation Example



Hardware Constraints Limit Flexibility of Warp Grouping



Slide credit: Krste Asanovic

An Example GPU

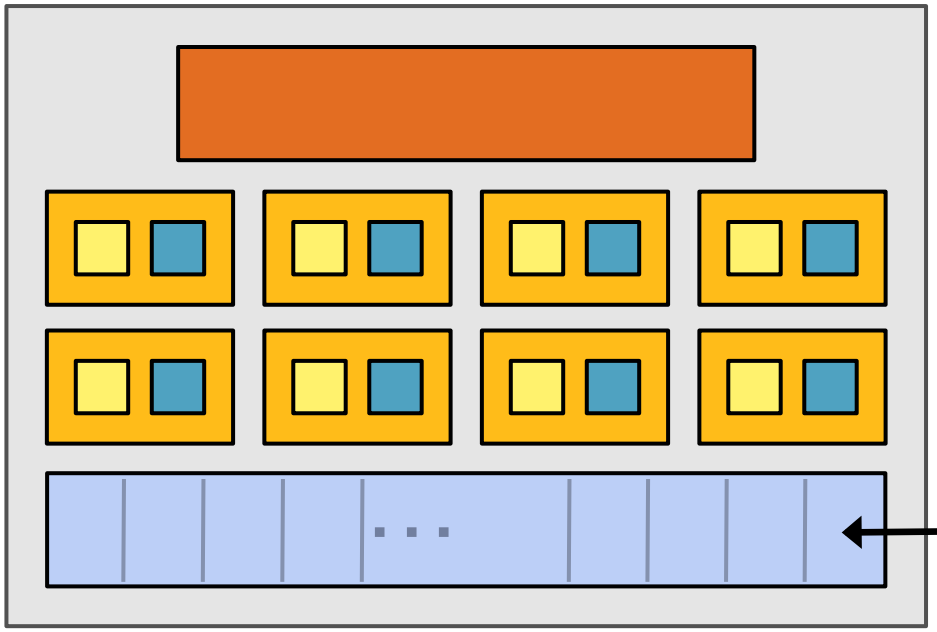
NVIDIA GeForce GTX 285

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

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

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

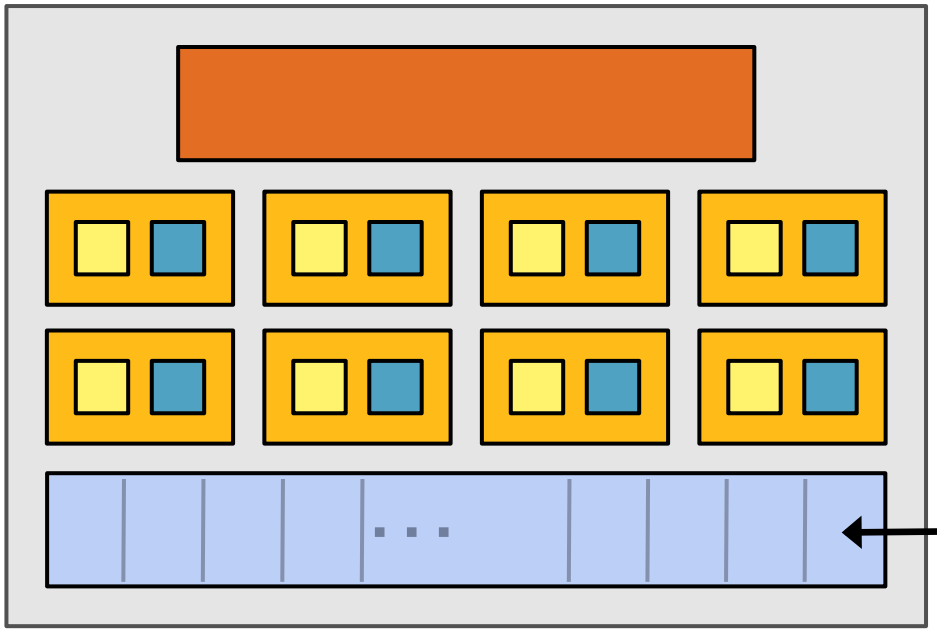
= multiply

= instruction stream decode

= execution context storage

Slide credit: Kayvon Fatahalian

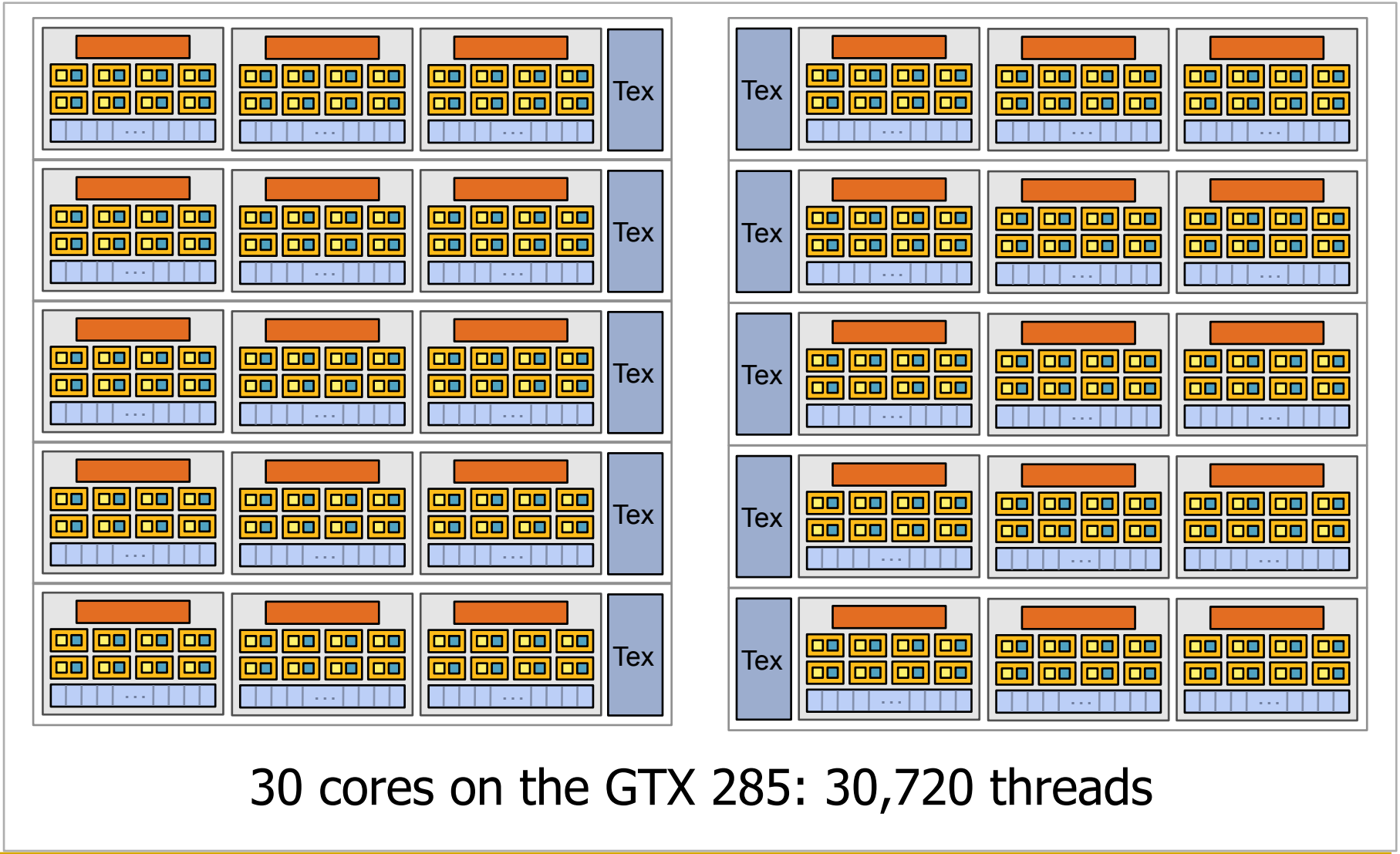
NVIDIA GeForce GTX 285 "core"



- 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

Introduction to GPGPU Programming

ETH Zürich
Fall 2017
19 October 2017

SAFARI

Agenda for Today

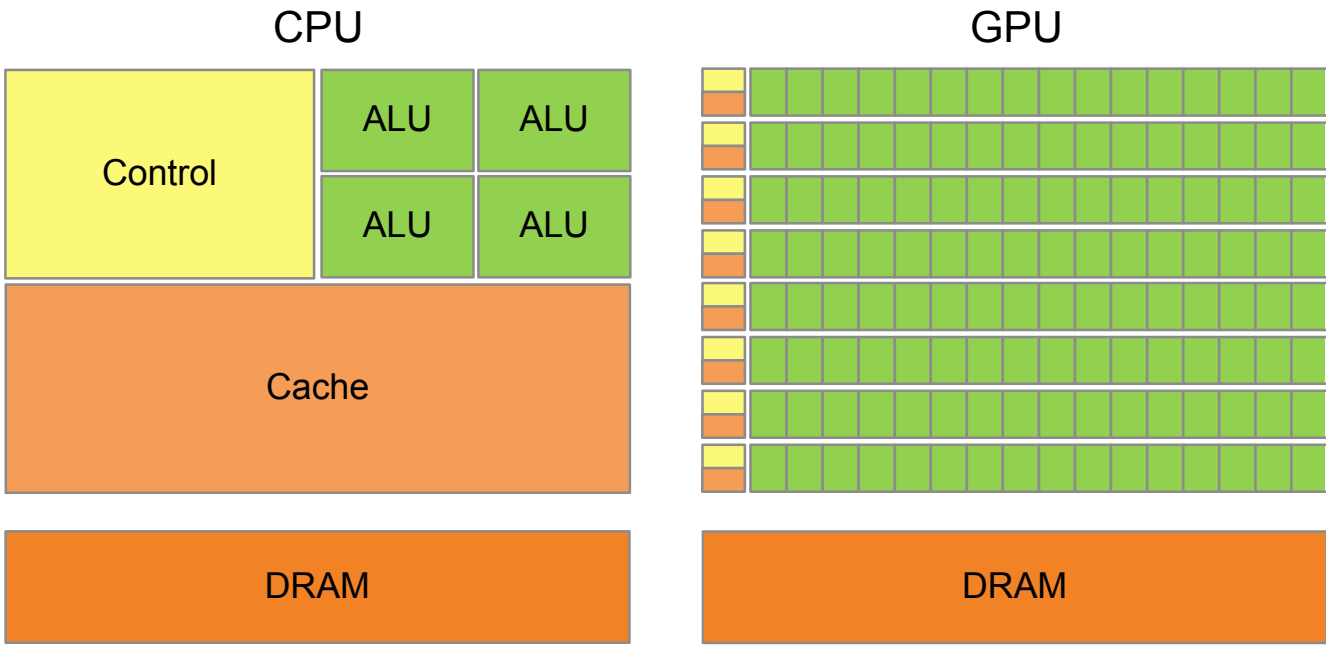
- Traditional accelerator model
  - Program structure
    - Bulk synchronous programming model
  - Memory hierarchy and memory management
  - Performance considerations
    - Memory access
    - SIMD utilization
    - Atomic operations
    - Data transfers
- New programming features
  - Dynamic parallelism
  - Collaborative computing

General Purpose Processing on GPU

- GPUs have democratized HPC
  - Great FLOP/$, massively parallel chip on a commodity PC
- However, this is not for free
  - New programming model
  - New challenges
- Algorithms need to be re-implemented and rethought
- Many workloads exhibit inherent parallelism
  - Matrices
  - Image processing
- Main bottlenecks
  - CPU-GPU data transfers (PCIe, NVLINK)
  - DRAM memory (GDDR5, HBM2)

CPU vs. GPU

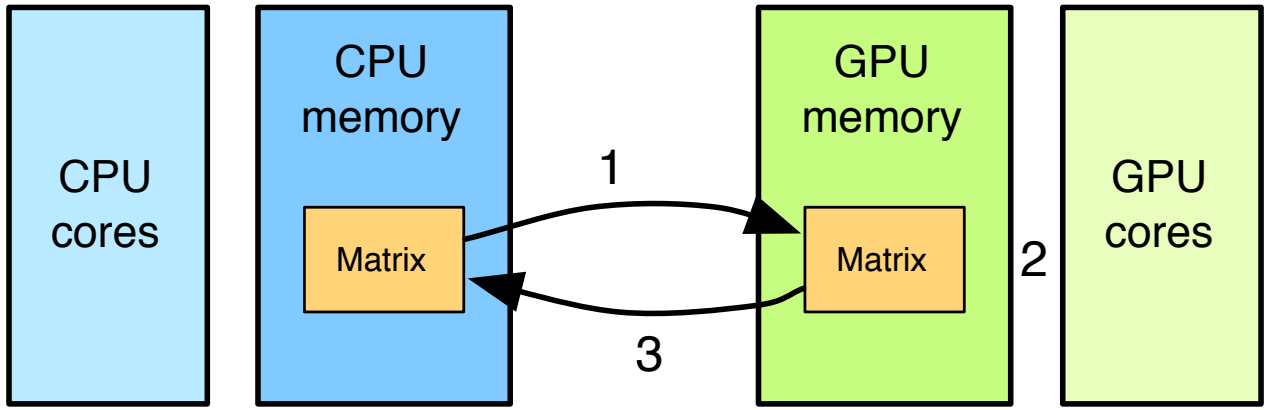
- Different design philosophies
  - CPU: A few out-of-order cores
  - GPU: Many in-order cores



Slide credit: Hwu & Kirk

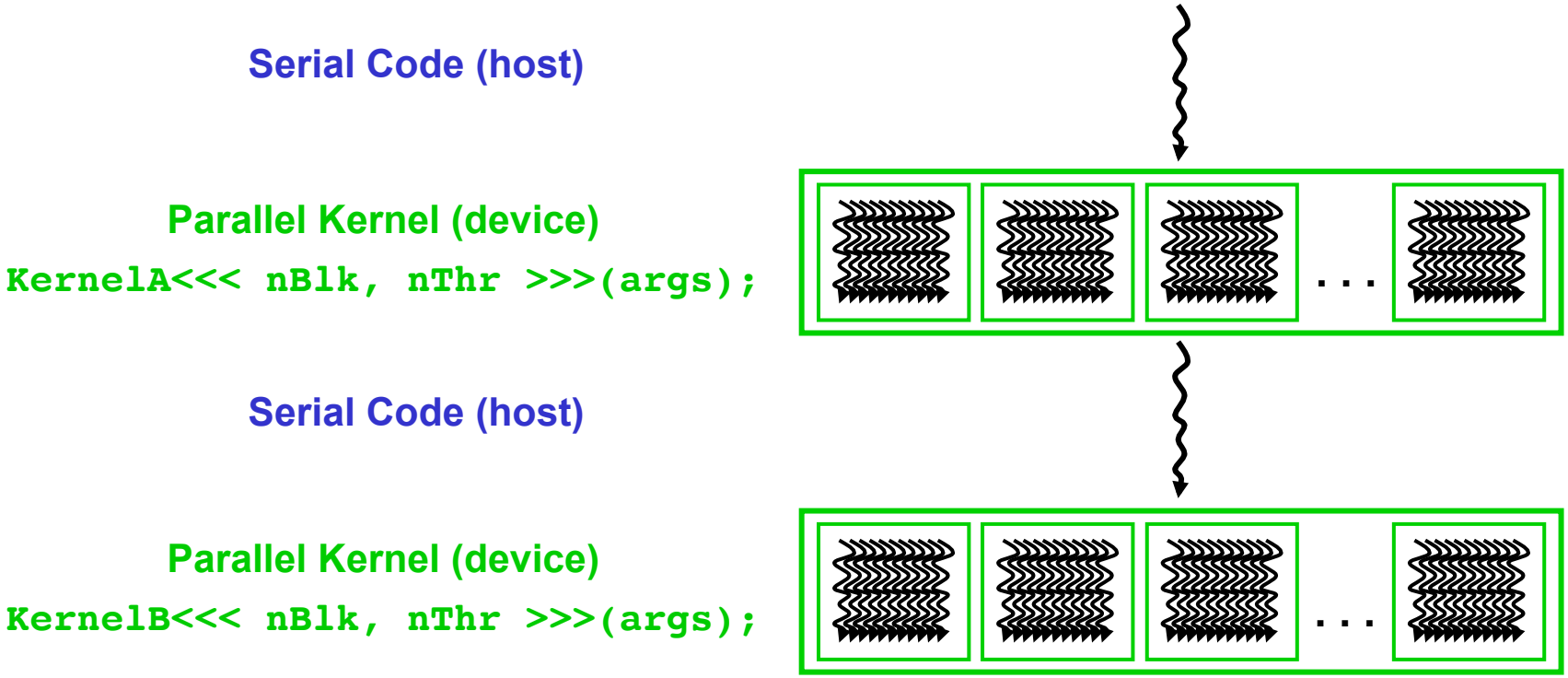
GPU Computing

- Computation is offloaded to the GPU
- Three steps
  - CPU-GPU data transfer (1)
  - GPU kernel execution (2)
  - GPU-CPU data transfer (3)



Traditional Program Structure

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



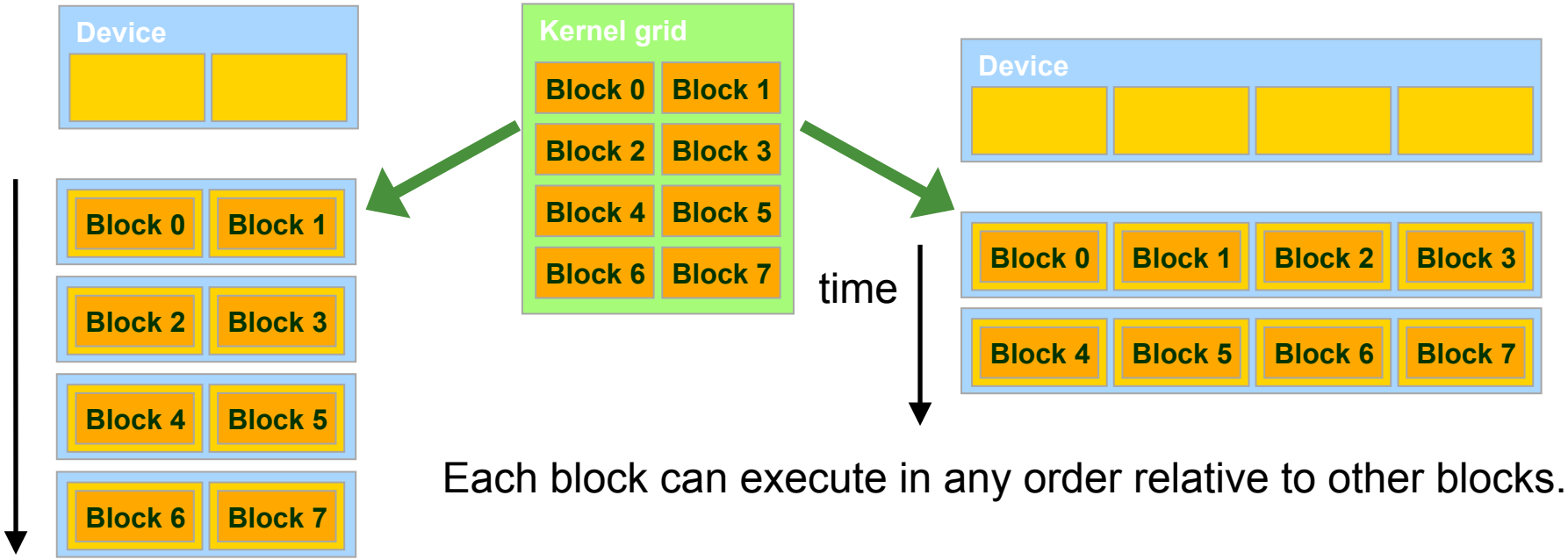
Slide credit: Hwu & Kirk

CUDA/OpenCL Programming Model

- SIMT or SPMD
- Bulk synchronous programming
  - Global (coarse-grain) synchronization between kernels
- The host (typically CPU) allocates memory, copies data, and launches kernels
- The device (typically GPU) executes kernels
  - Grid (NDRange)
  - Block (work-group)
    - Within a block, shared memory and synchronization
  - Thread (work-item)

Transparent Scalability

- Hardware is free to schedule thread blocks

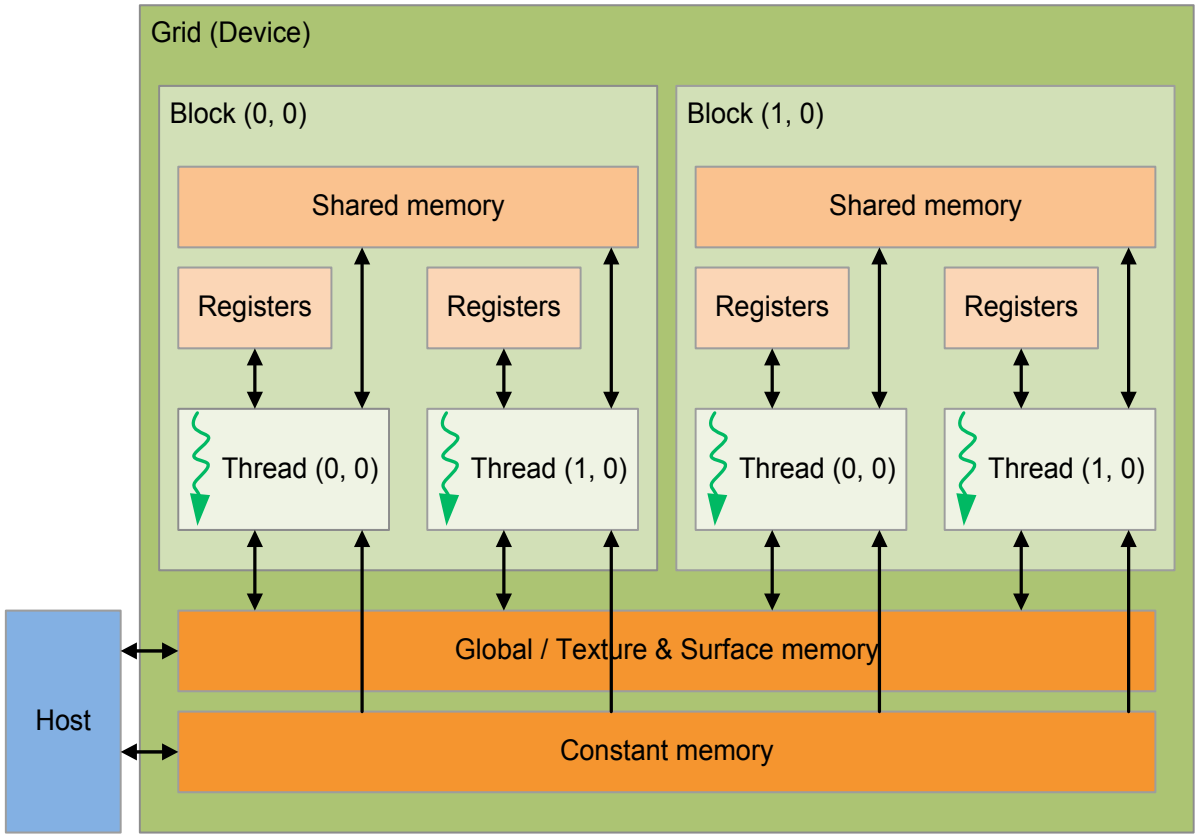


Each block can execute in any order relative to other blocks.

Slide credit: Hwu & Kirk

CUDA/OpenCL Programming Model

- Memory hierarchy

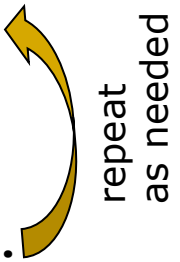


Traditional Program Structure

- Function prototypes

```
float serialFunction(…);
__global__ void kernel(…);
```

- `main()`
  - 1) Allocate memory space on the device – `cudaMalloc(&d_in, bytes);`
  - 2) Transfer data from host to device – `cudaMemCpy(d_in, h_in, …);`
  - 3) Execution configuration setup: #blocks and #threads
  - 4) Kernel call – `kernel<<<execution configuration>>>(args…);`
  - 5) Transfer results from device to host – `cudaMemCpy(h_out, d_out, …);`



- Kernel – `__global__ void kernel(type args,…)`
  - Automatic variables transparently assigned to registers
  - Shared memory – `__shared__`
  - Intra-block synchronization – `__syncthreads();`

Slide credit: Hwu & Kirk

CUDA Programming Language

- Memory allocation

```
cudaMalloc((void**)&d_in, #bytes);
```

- Memory copy

```
cudaMemcpy(d_in, h_in, #bytes,
                cudaMemcpyHostToDevice);
```

- Kernel launch

```
kernel<<< #blocks, #threads >>>(args);
```

- Memory deallocation

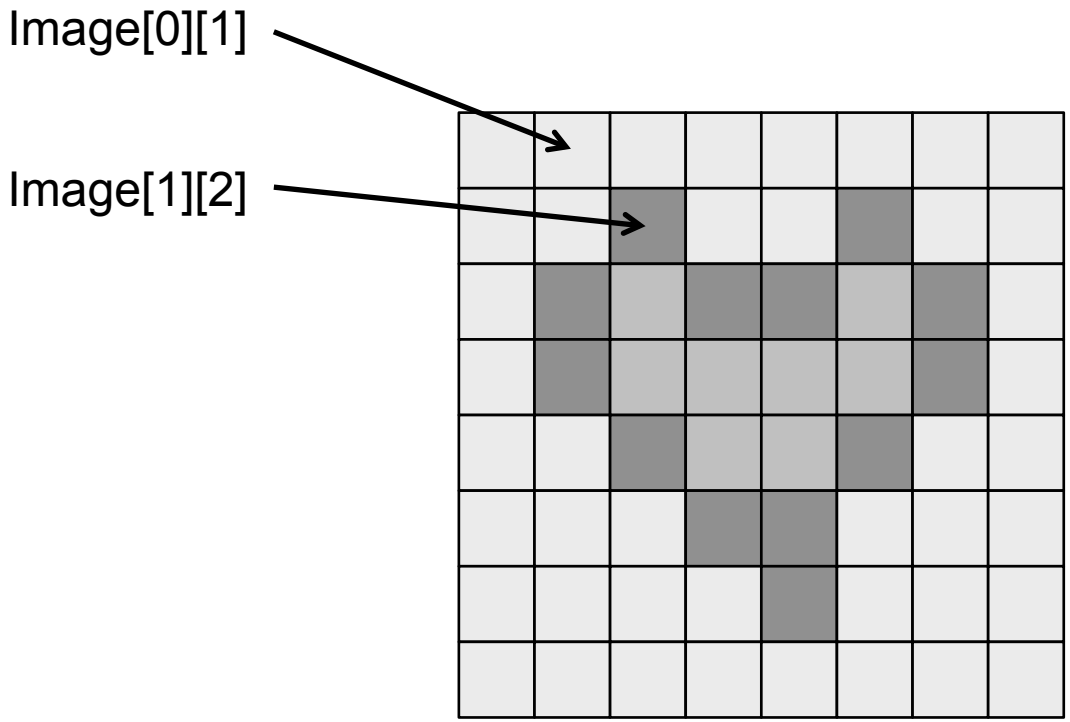
```
cudaFree(d_in);
```

- Explicit synchronization

```
cudaDeviceSynchronize();
```

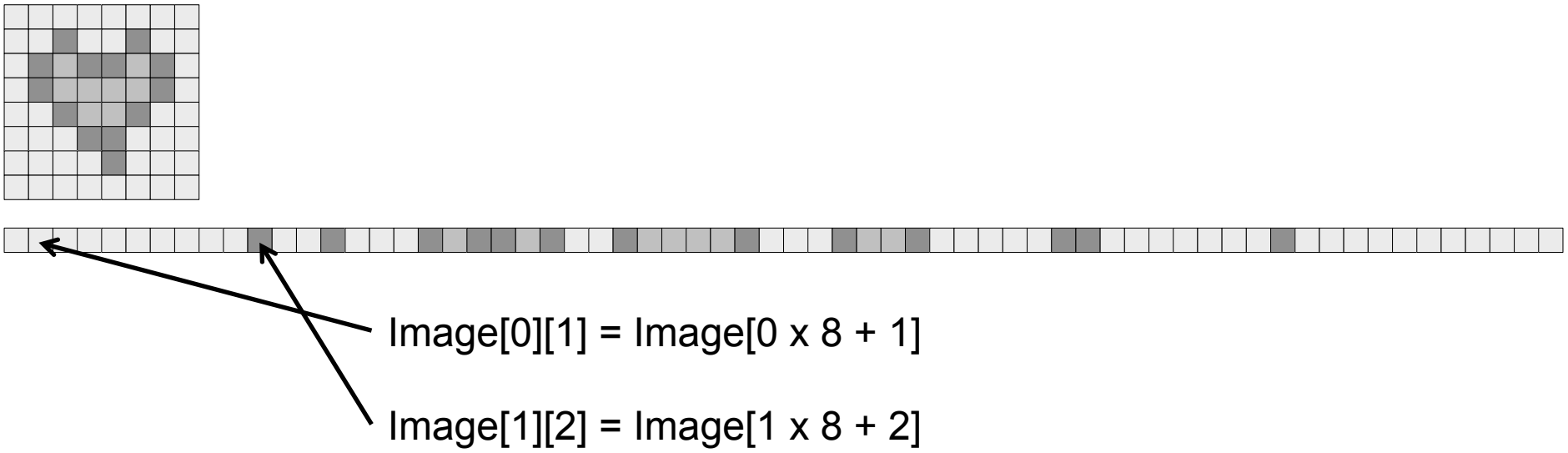
Indexing and Memory Access

- Image layout in memory
  - height x width
  - Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width



Indexing and Memory Access

- Image layout in memory
  - Row-major layout
  - Image[j][i] = Image[j x width + i]

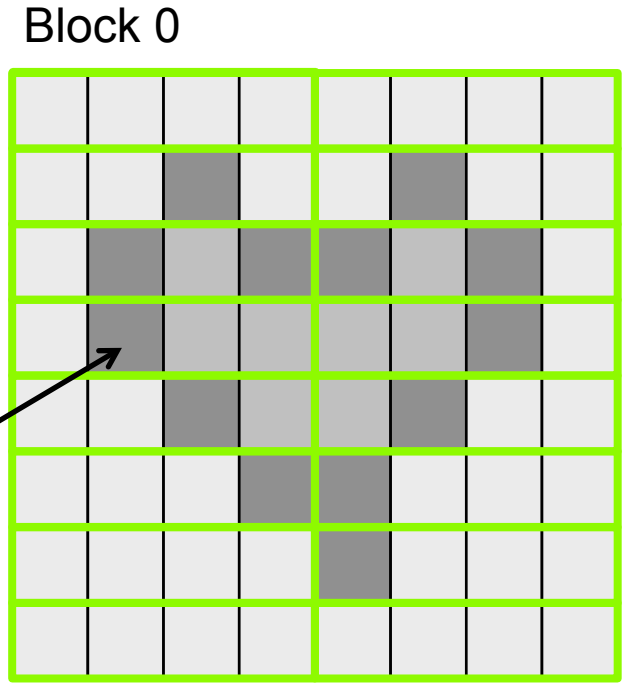


Indexing and Memory Access

- One GPU thread per pixel
- Grid of Blocks of Threads
  - `blockIdx.x, threadIdx.x`
  - `gridDim.x, blockDim.x`

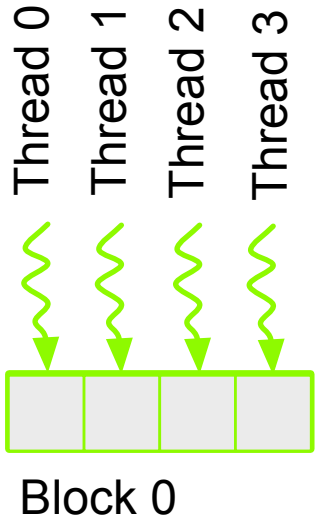
`blockIdx.x`

`threadIdx.x`



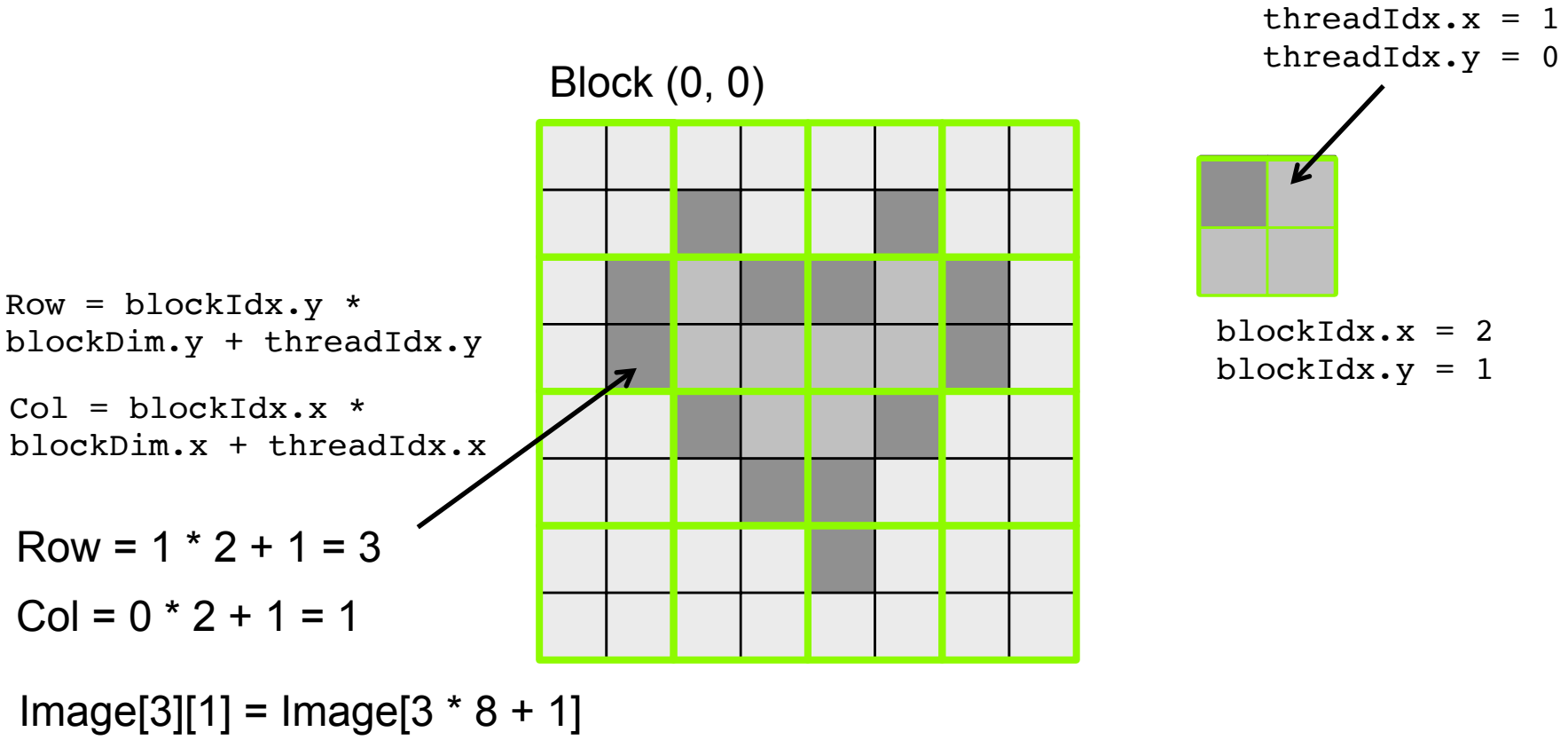
6 * 4 + 1 = 25

`blockIdx.x * blockDim.x + threadIdx.x`



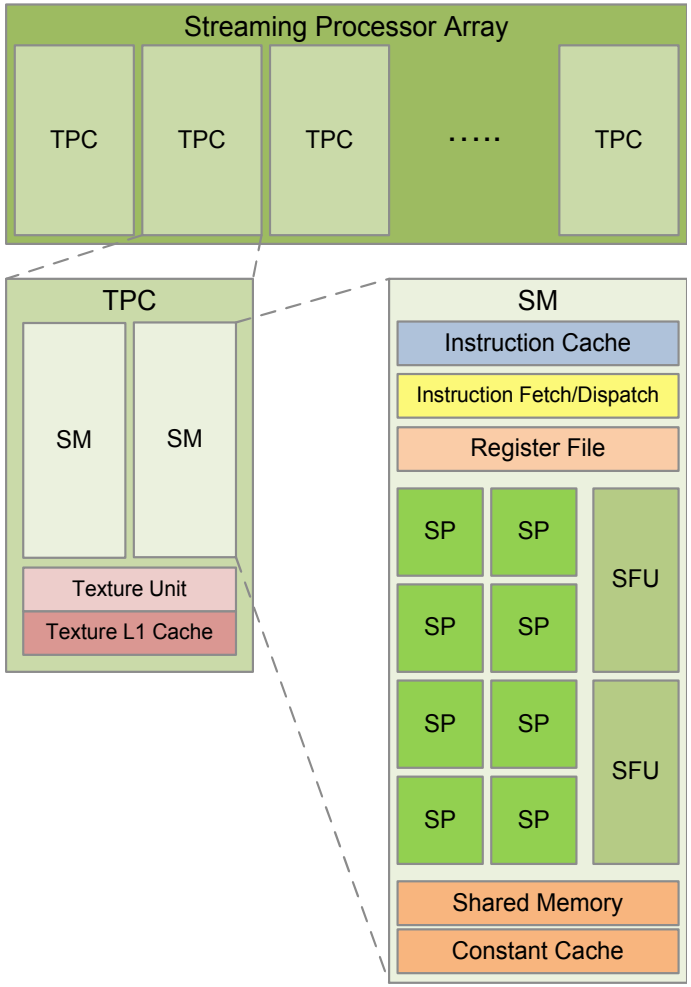
Indexing and Memory Access

- 2D blocks
  - gridDim.x, gridDim.y



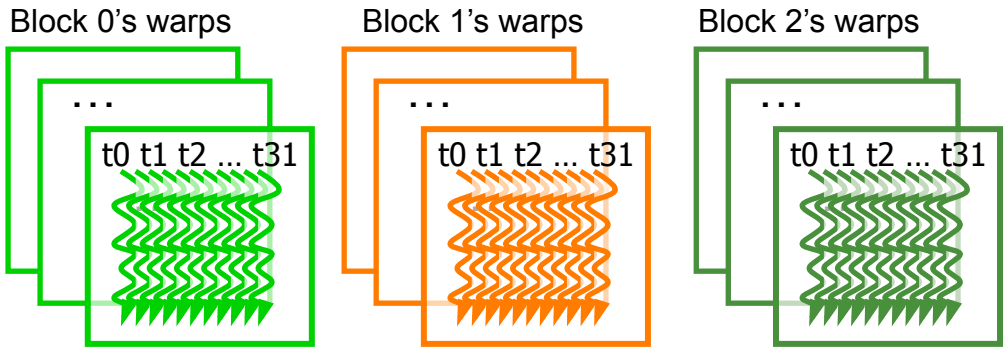
Brief Review of GPU Architecture

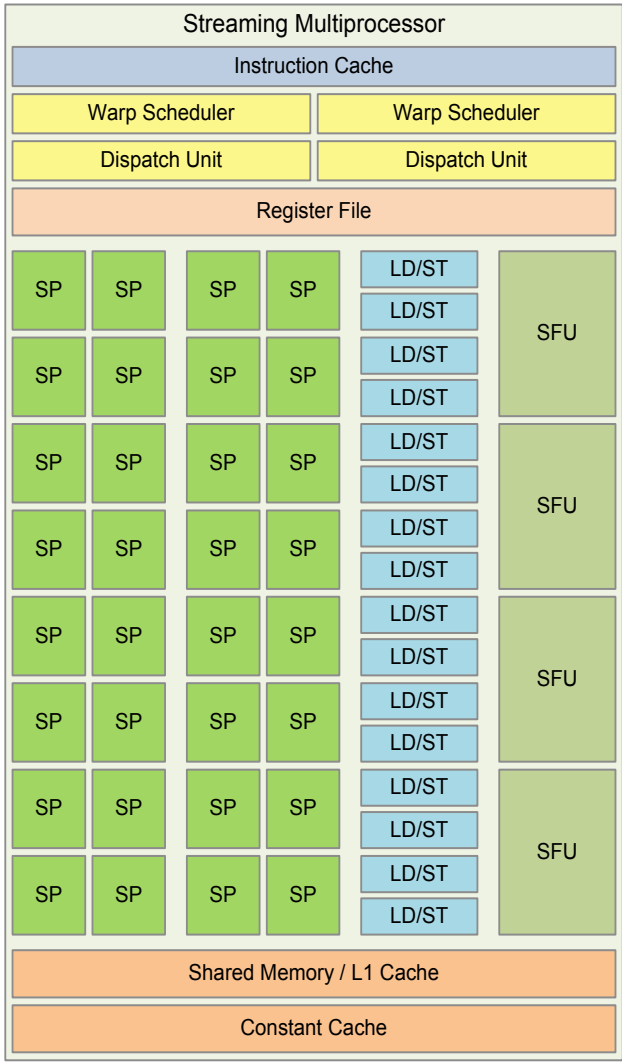
- Streaming Processor Array
  - Tesla architecture (G80/GT200)



Brief Review of GPU Architecture

- Blocks are divided into warps
  - SIMD unit (32 threads)
- Streaming Multiprocessors (SM)
  - Streaming Processors (SP)





Brief Review of GPU Architecture

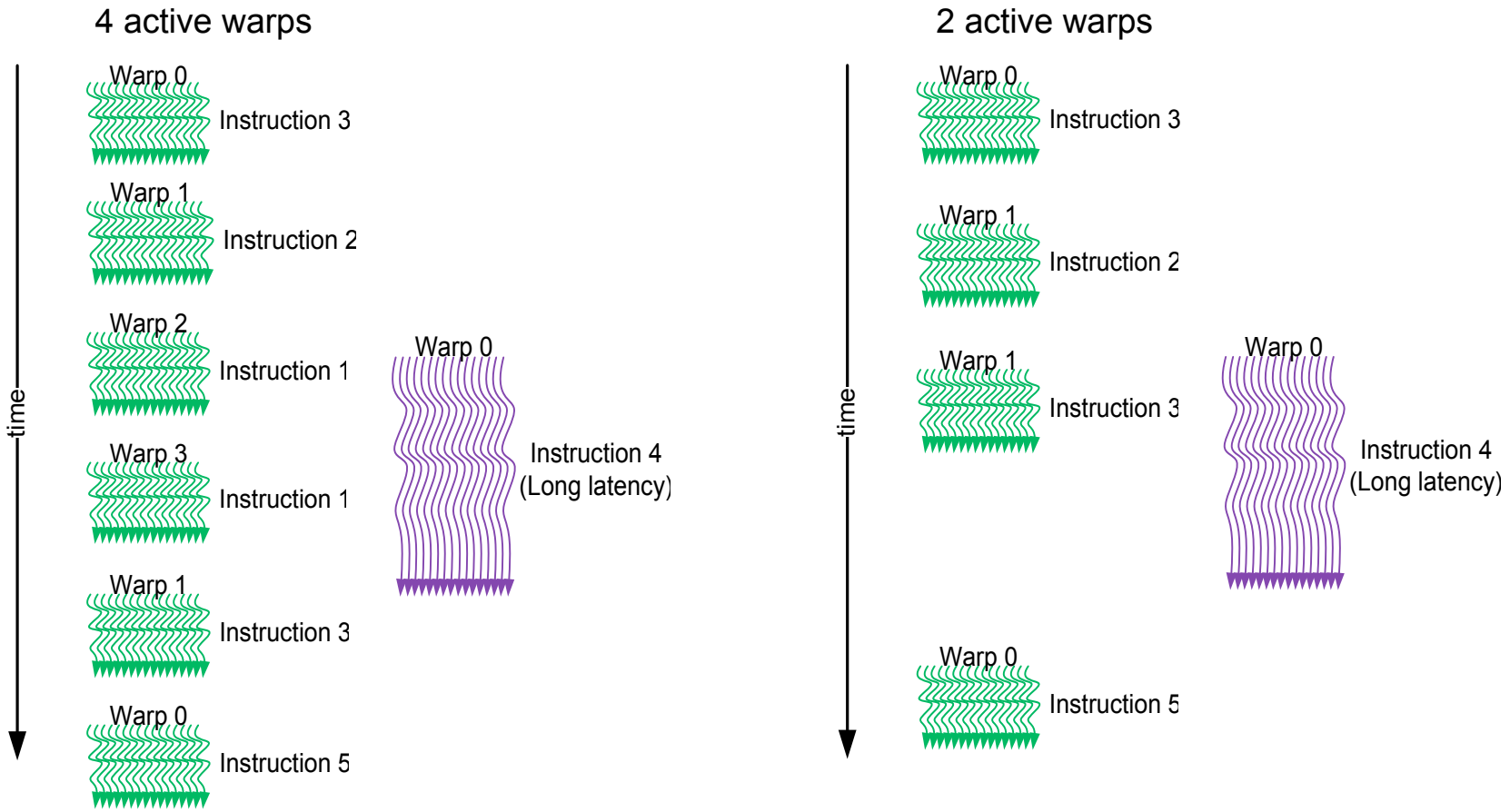
- Streaming Multiprocessors (SM)
  - Compute Units (CU)
- Streaming Processors (SP) or CUDA cores
  - Vector lanes
- Number of SMs x SPs
  - Tesla (2007): 30 x 8
  - Fermi (2010): 16 x 32
  - Kepler (2012): 15 x 192
  - Maxwell (2014): 24 x 128
  - Pascal (2016): 56 x 64
  - Volta (2017): 80 x 64

Performance Considerations

- Main bottlenecks
  - Global memory access
  - CPU-GPU data transfers
- Memory access
  - Latency hiding
    - Thread Level Parallelism (TLP)
    - Occupancy
  - Memory coalescing
  - Data reuse
    - Shared memory usage
- SIMD Utilization
- Atomic operations
- Data transfers between CPU and GPU
  - Overlap of communication and computation

Latency Hiding

- Occupancy: ratio of active warps
  - Not only memory accesses (e.g., SFU)

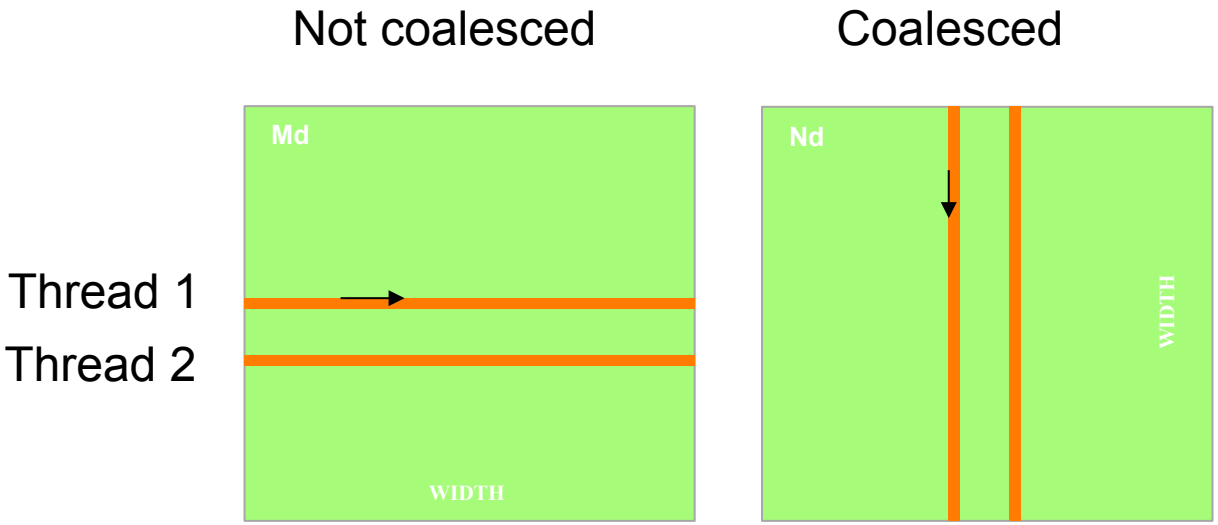


Occupancy

- SM resources (typical values)
  - Maximum number of warps per SM (64)
  - Maximum number of blocks per SM (32)
  - Register usage (256KB)
  - Shared memory usage (64KB)
- Occupancy calculation
  - Number of threads per block
  - Registers per thread
  - Shared memory per block
- The number of registers per thread is known in compile time

Memory Coalescing

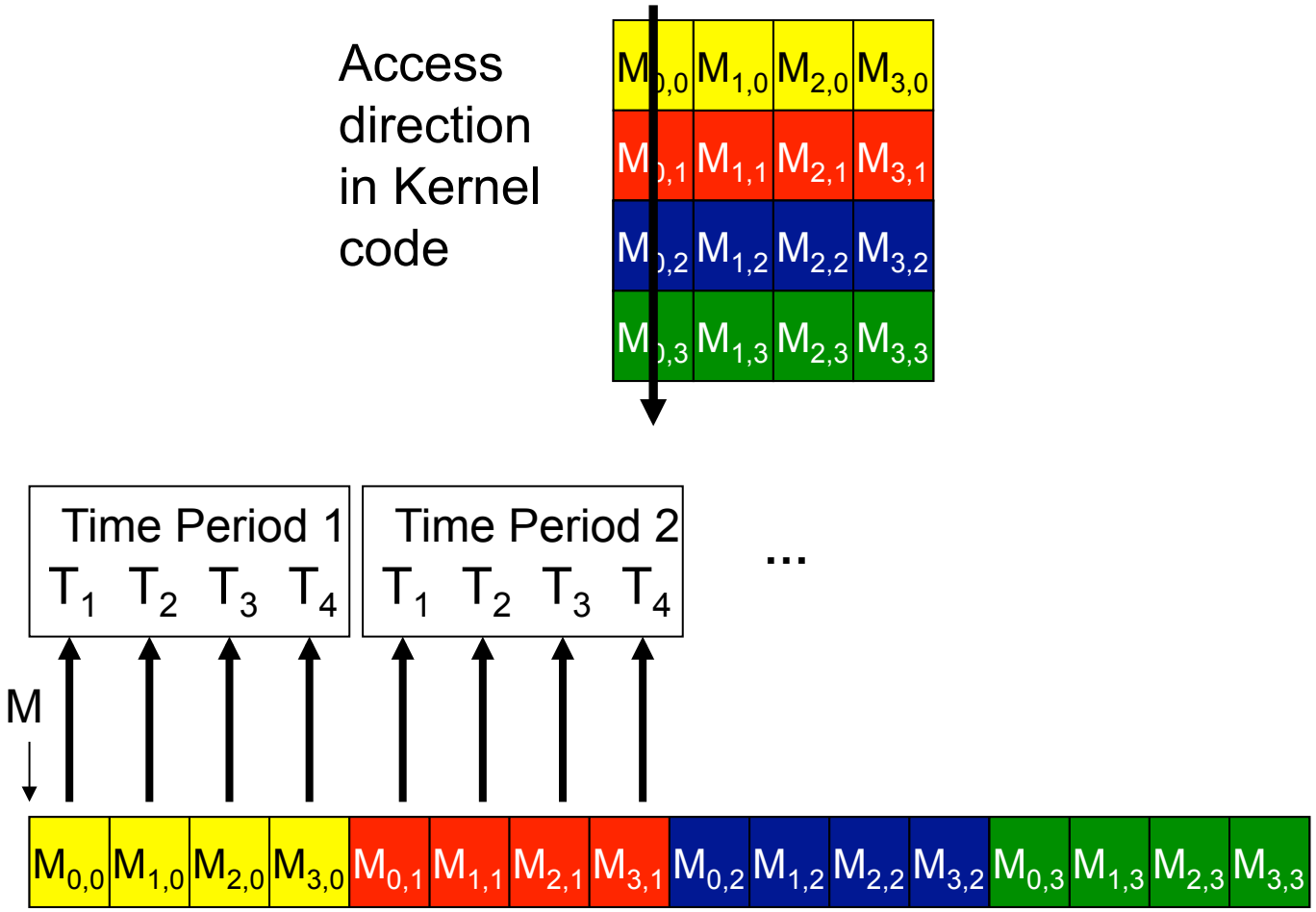
- When accessing global memory, peak bandwidth utilization occurs when all threads in a warp access one cache line



Slide credit: Hwu & Kirk

Memory Coalescing

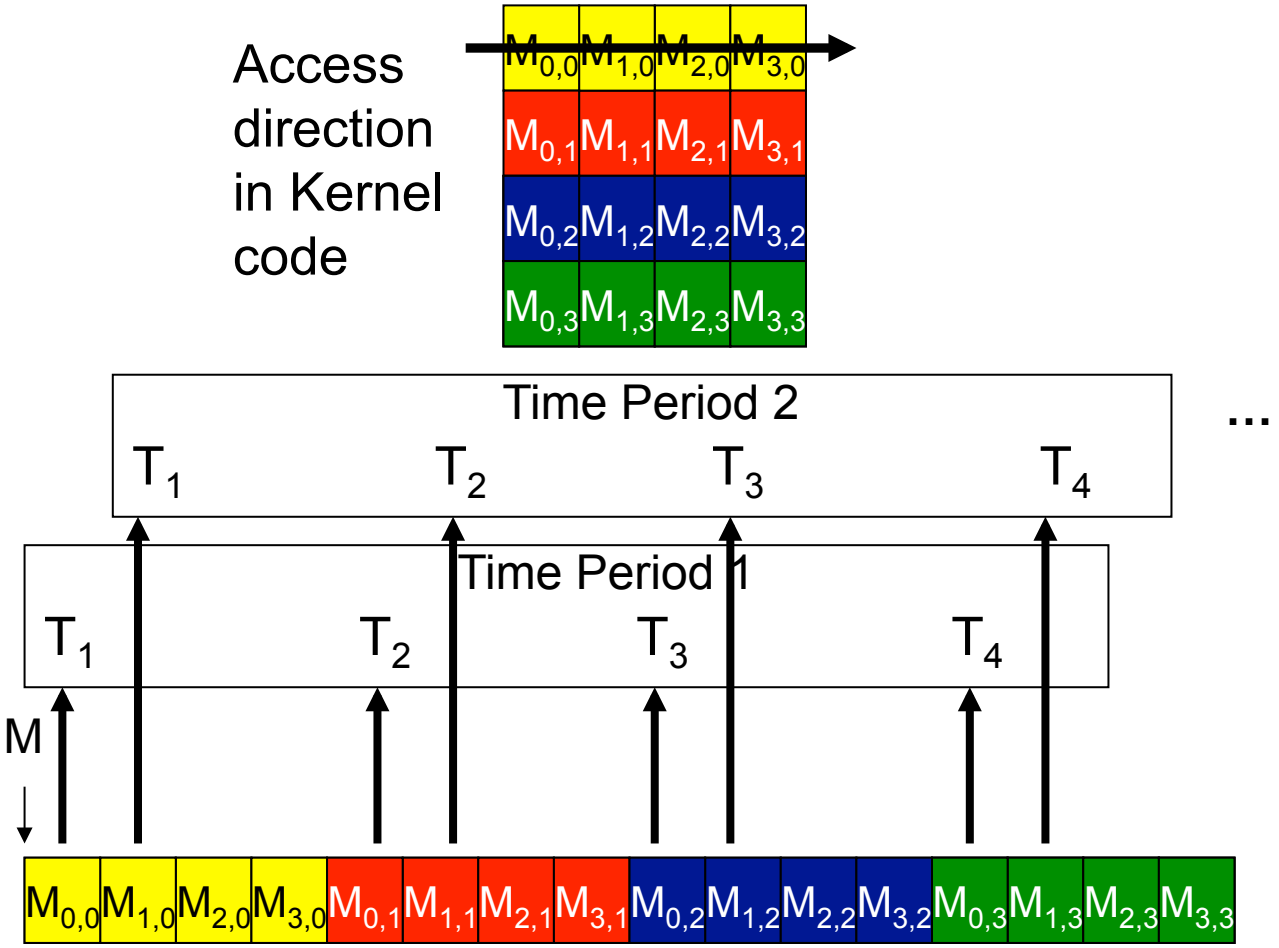
- Coalesced accesses



Slide credit: Hwu & Kirk

Memory Coalescing

- Uncoalesced accesses



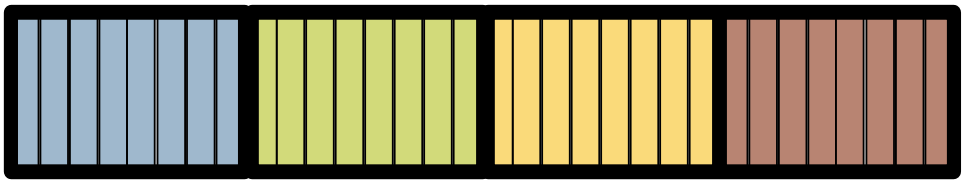
Slide credit: Hwu & Kirk

Memory Coalescing

- AoS vs. SoA

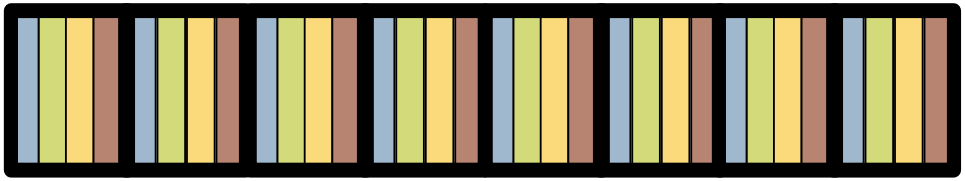
Structure of Arrays (SoA)

```
struct foo{
 float a[8];
 float b[8];
 float c[8];
 int d[8];
} A;
```



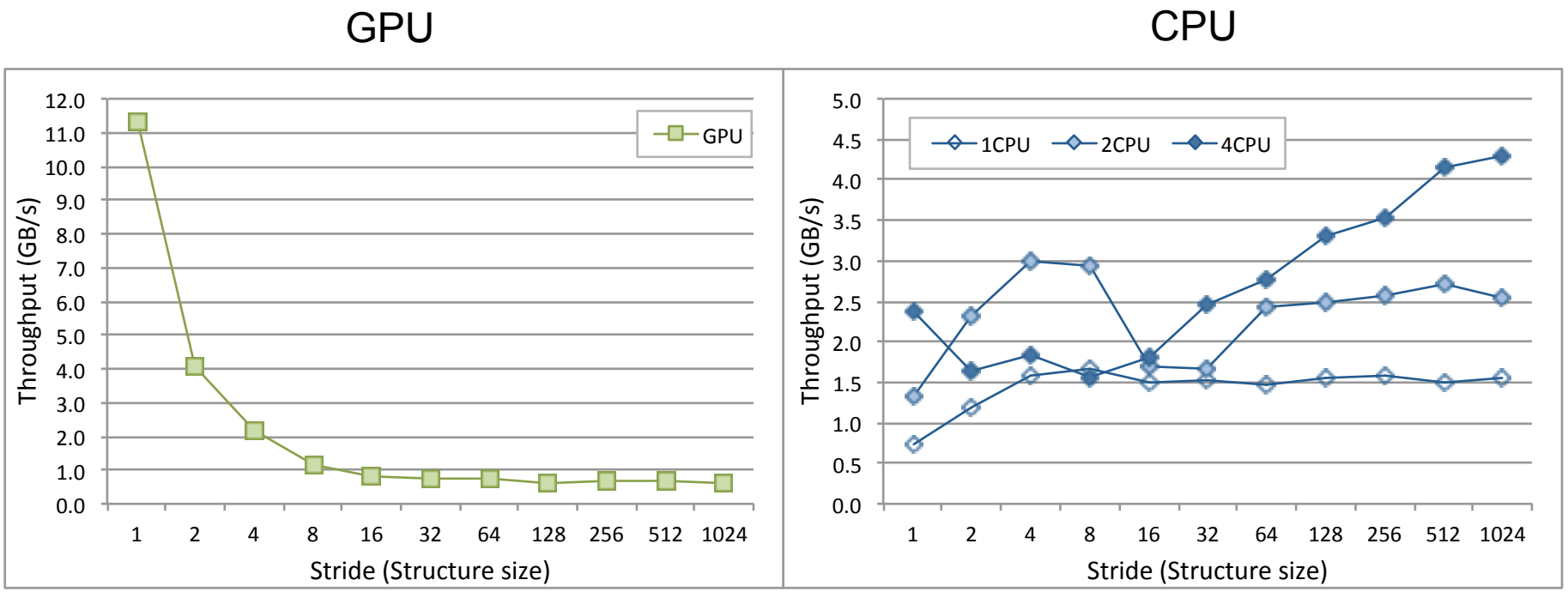
Array of Structures (AoS)

```
struct foo{
 float a;
 float b;
 float c;
 int d;
} A[8];
```



Memory Coalescing

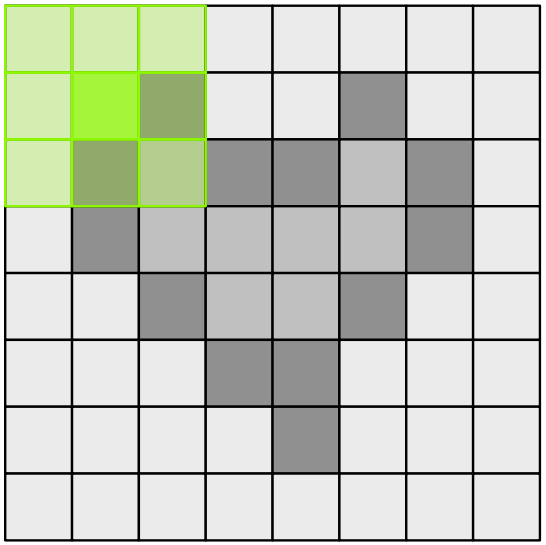
- Linear and strided accesses



AMD Kaveri A10-7850K

Data Reuse

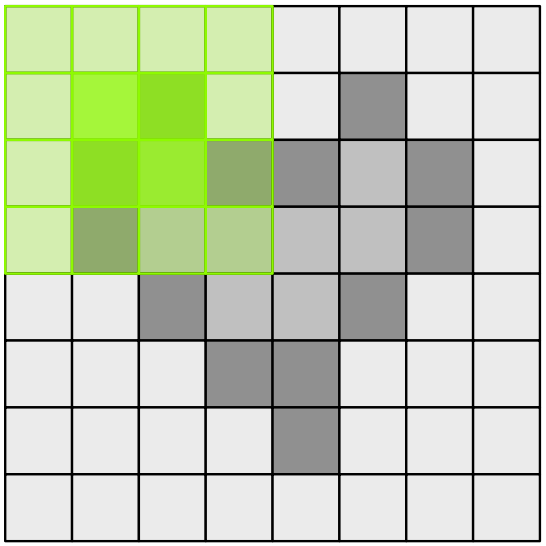
- Same memory locations accessed by neighboring threads



```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```

Data Reuse

- Shared memory tiling



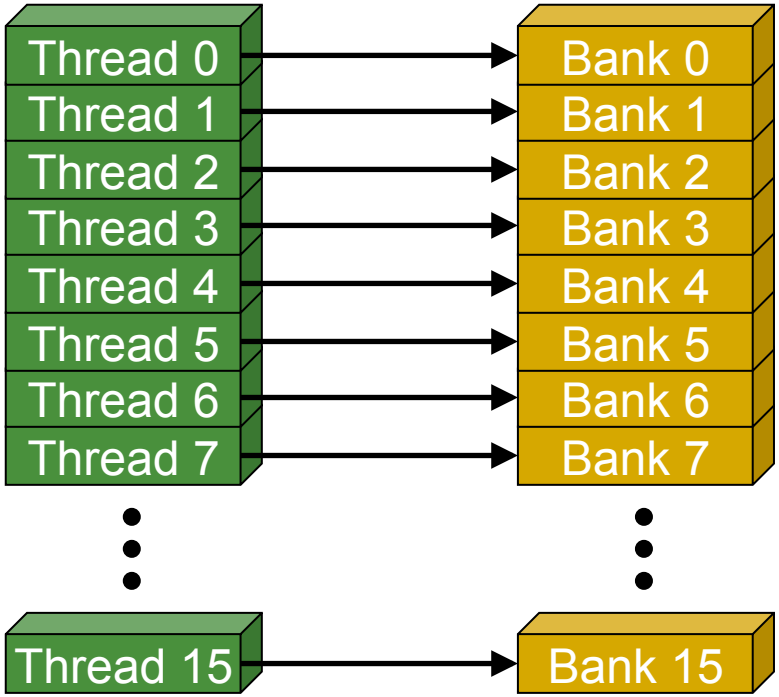
```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
…
Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

Shared Memory

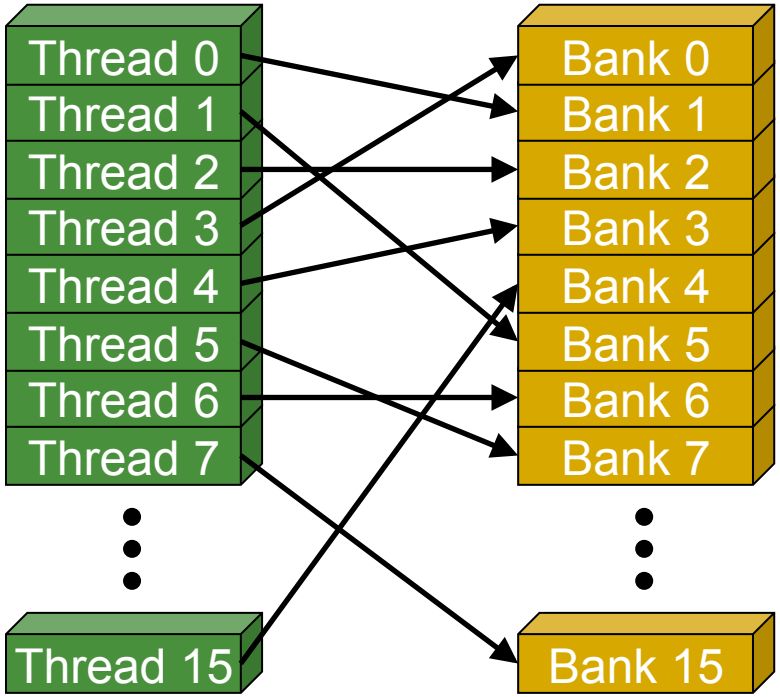
- Shared memory is an interleaved memory
  - Typically 32 banks
  - Each bank can service one address per cycle
  - Successive 32-bit words are assigned to successive banks
    - Bank = Address % 32
- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps

Shared Memory

- Bank conflict free



Linear addressing: stride = 1

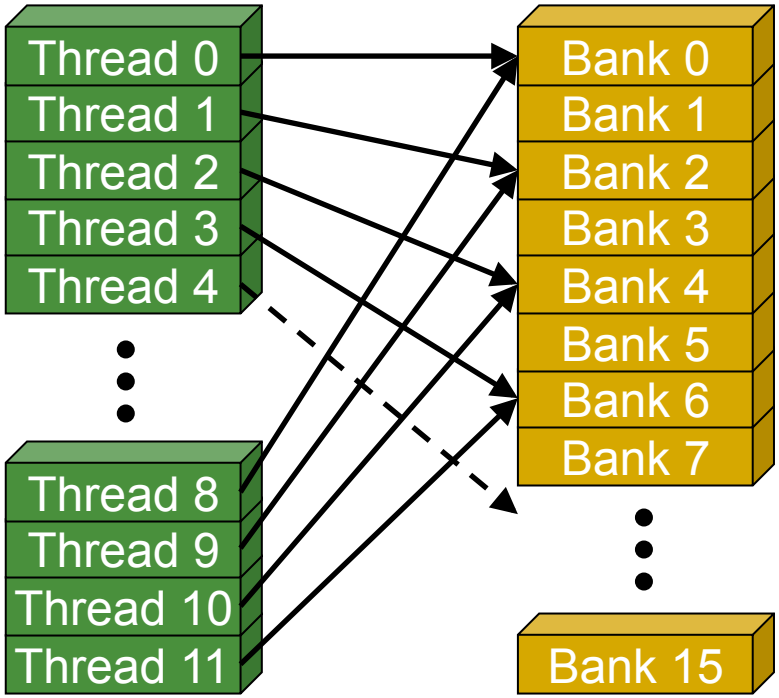


Random addressing 1:1

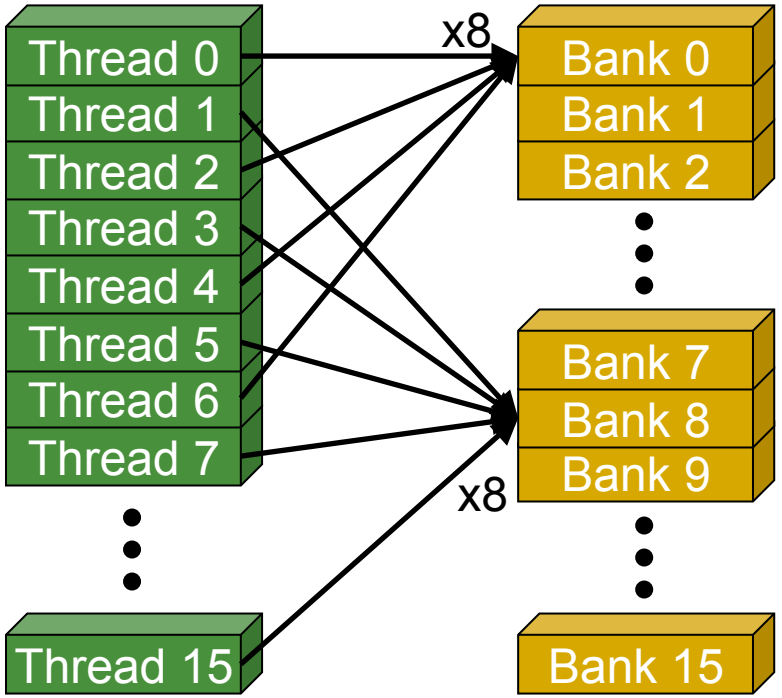
Slide credit: Hwu & Kirk

Shared Memory

- N-way bank conflicts



2-way bank conflict: stride = 2



8-way bank conflict: stride = 8

Slide credit: Hwu & Kirk

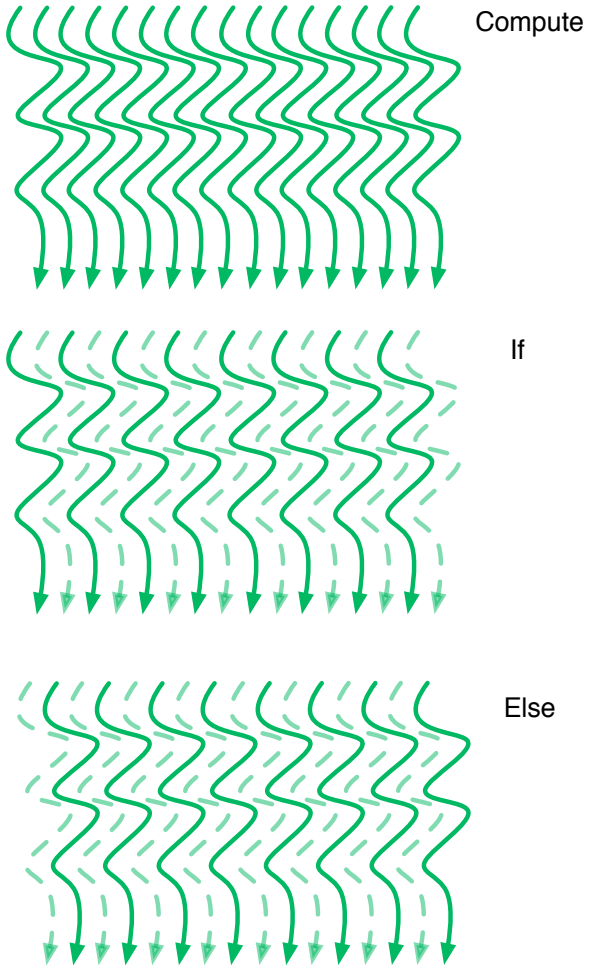
Shared Memory

- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps
- If strided accesses are needed, some optimization techniques can help
  - Padding
  - Hash functions

SIMD Utilization

- Intra-warp divergence

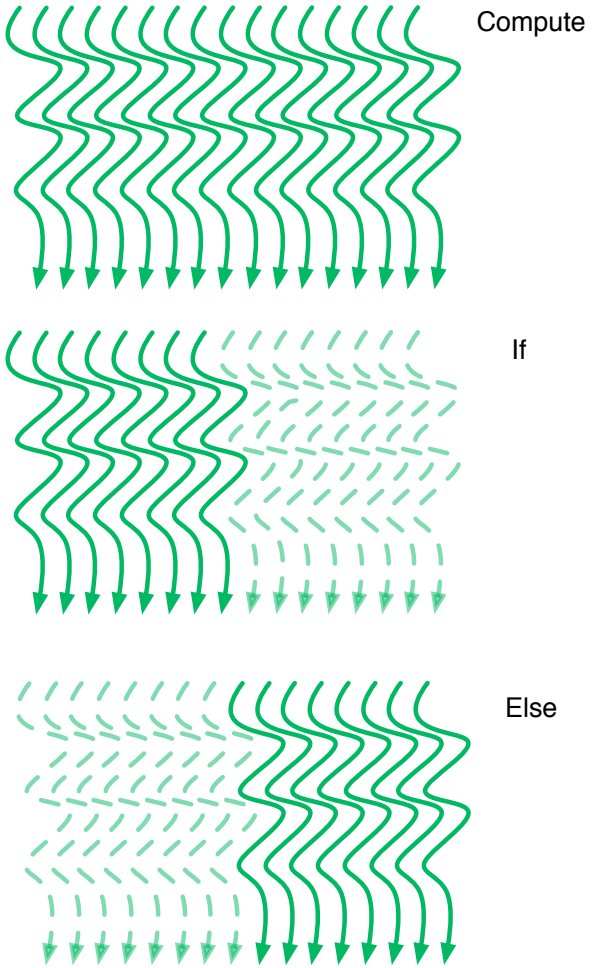
```
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
  Do_this(threadIdx.x);
}
else{
  Do_that(threadIdx.x);
}
```



SIMD Utilization

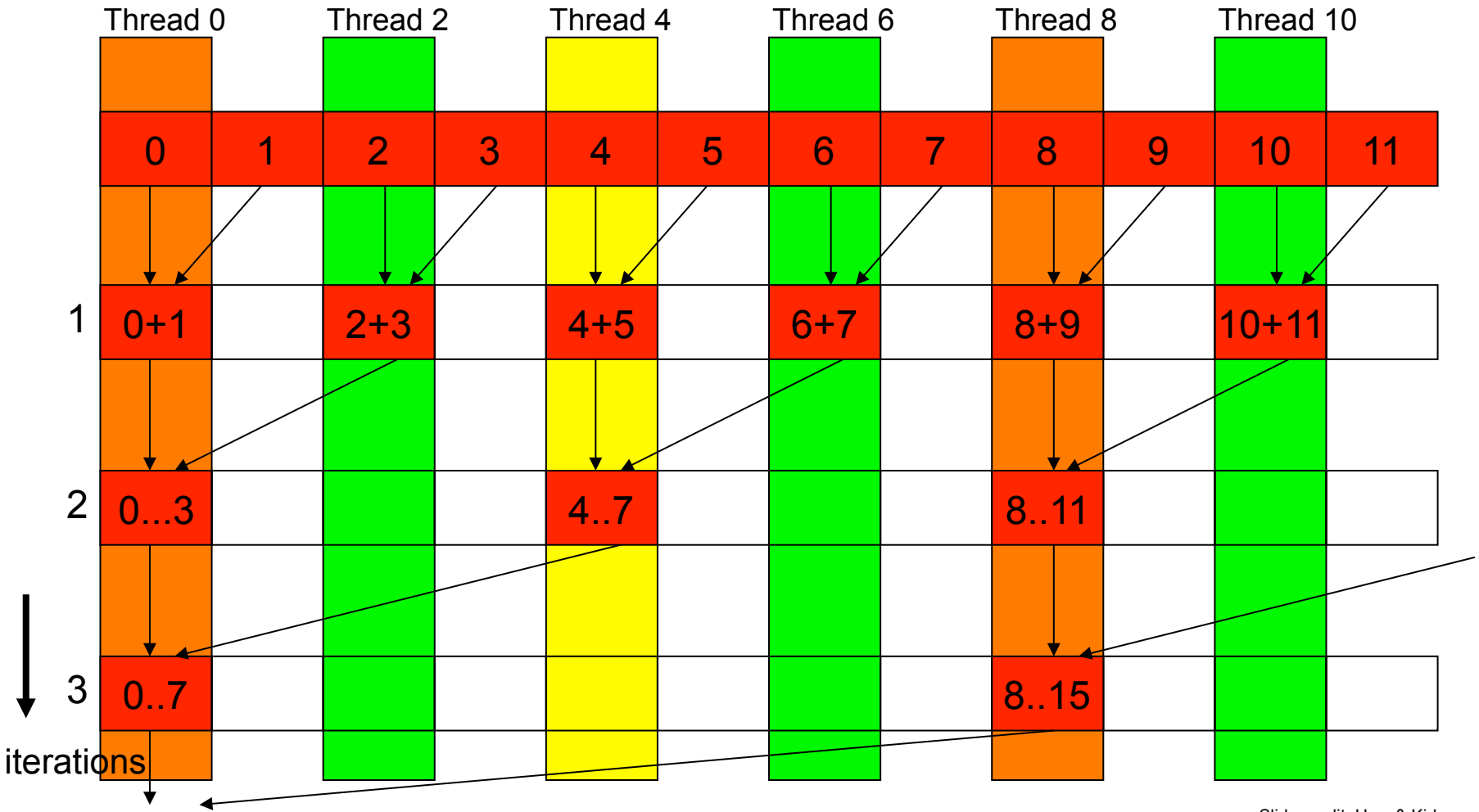
- Intra-warp divergence

```
Compute(threadIdx.x);
if (threadIdx.x < 32){
  Do_this(threadIdx.x * 2);
}
else{
  Do_that((threadIdx.x%32)*2+1);
}
```



Vector Reduction

- Naïve mapping



Slide credit: Hwu & Kirk

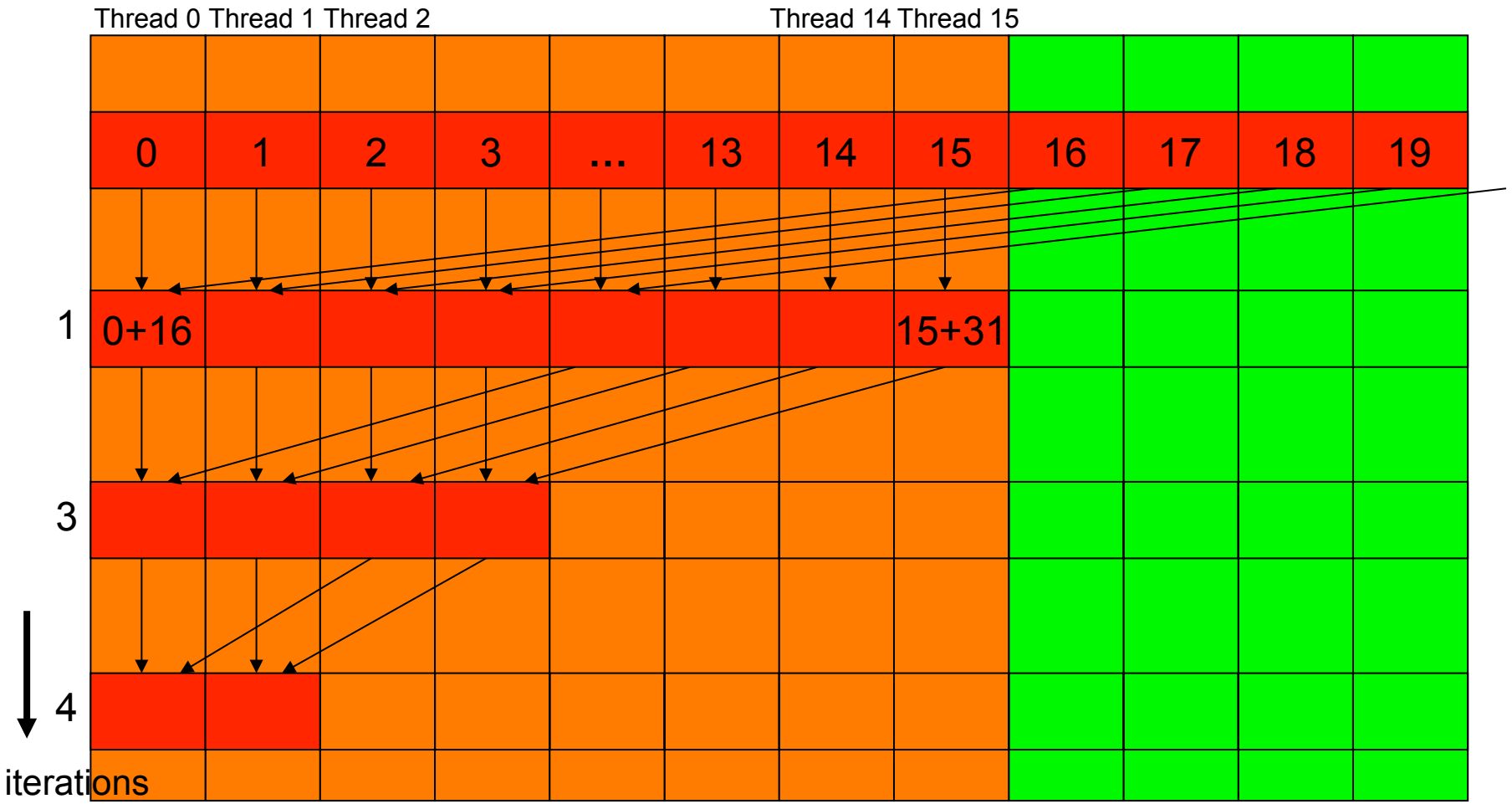
Vector Reduction

- Naïve mapping

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = 1; stride < blockDim.x; stride *= 2) {

  __syncthreads();

  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t + stride];

}
```

Vector Reduction

- Divergence-free mapping



Slide credit: Hwu & Kirk

Vector Reduction

- Divergence-free mapping

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 1;  stride >> 1){

  __syncthreads();

  if (t < stride)
    partialSum[t] += partialSum[t + stride];

}
```

We did not cover the following slides in lecture.
These are for your preparation for the next lecture.

Atomic Operations

- Shared memory atomic operations
  - CUDA: `int atomicAdd(int*, int);`
  - PTX: `atom.shared.add.u32 %r25, [%rd14], %r24;`
  - SASS:

Tesla, Fermi, Kepler

```
/*00a0*/ LDSLK P0, R9, [R8];
/*00a8*/ @P0 IADD R10, R9, R7;
/*00b0*/ @P0 STSCUL P1, [R8], R10;
/*00b8*/ @!P1 BRA 0xa0;
```

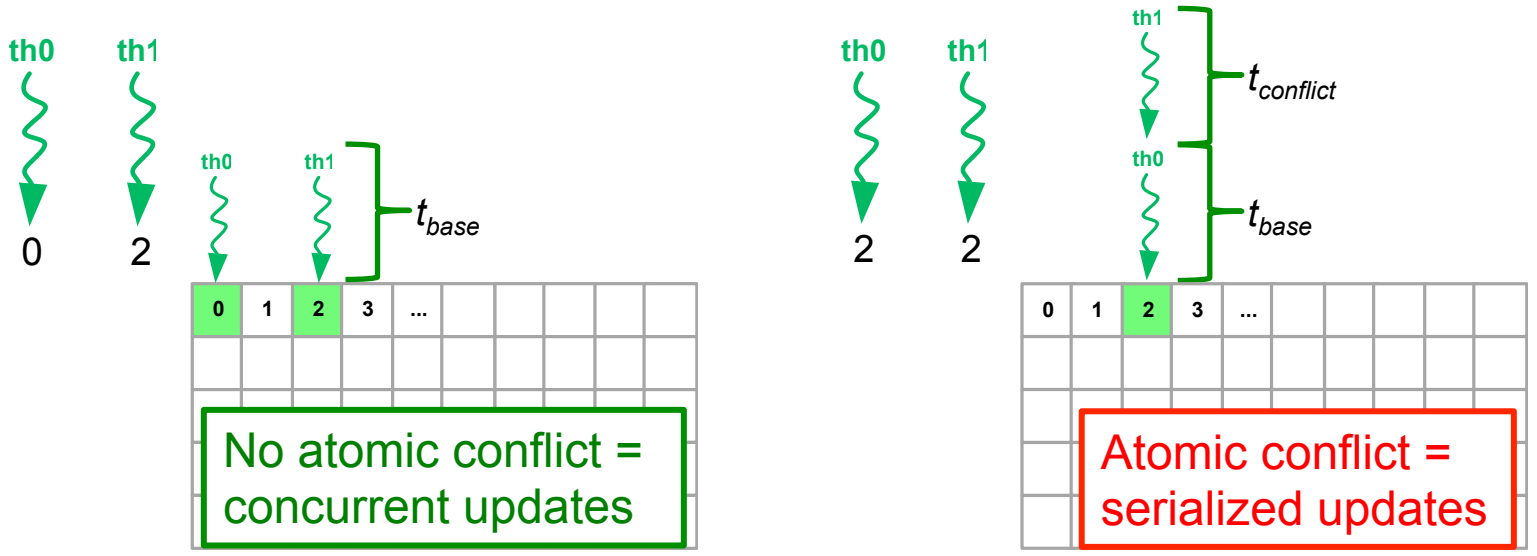
Maxwell

```
/*01f8*/ ATOMS.ADD RZ, [R7], R11;
```

Native atomic operations for 32-bit integer, and 32-bit and 64-bit atomicCAS

Atomic Operations

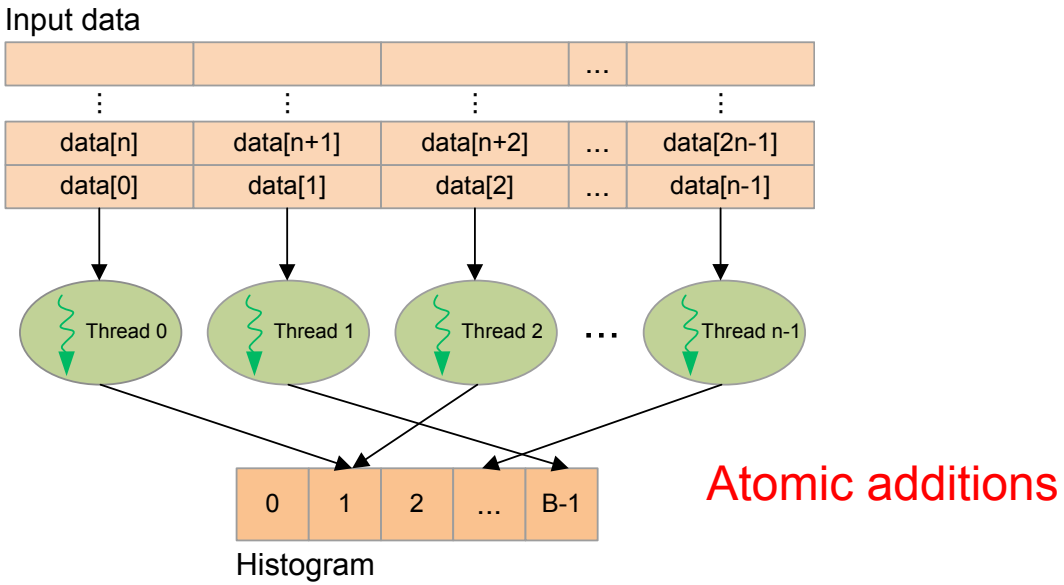
- Atomic conflicts
  - Intra-warp conflict degree from 1 to 32



Histogram Calculation

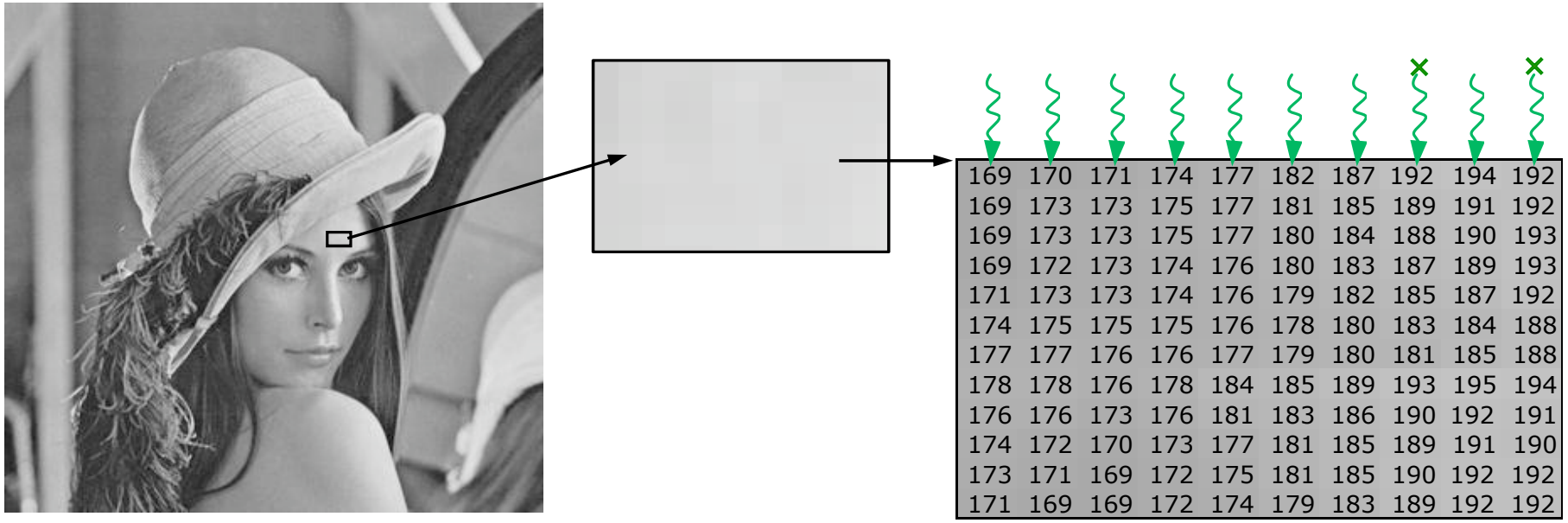
- Histograms count the number of data instances in disjoint categories (bins)

```
for (each pixel i in image I){
    Pixel = I[i]                    // Read pixel
    Pixel' = Computation(Pixel)     // Optional computation
    Histogram[Pixel']++             // Vote in histogram bin
}
```



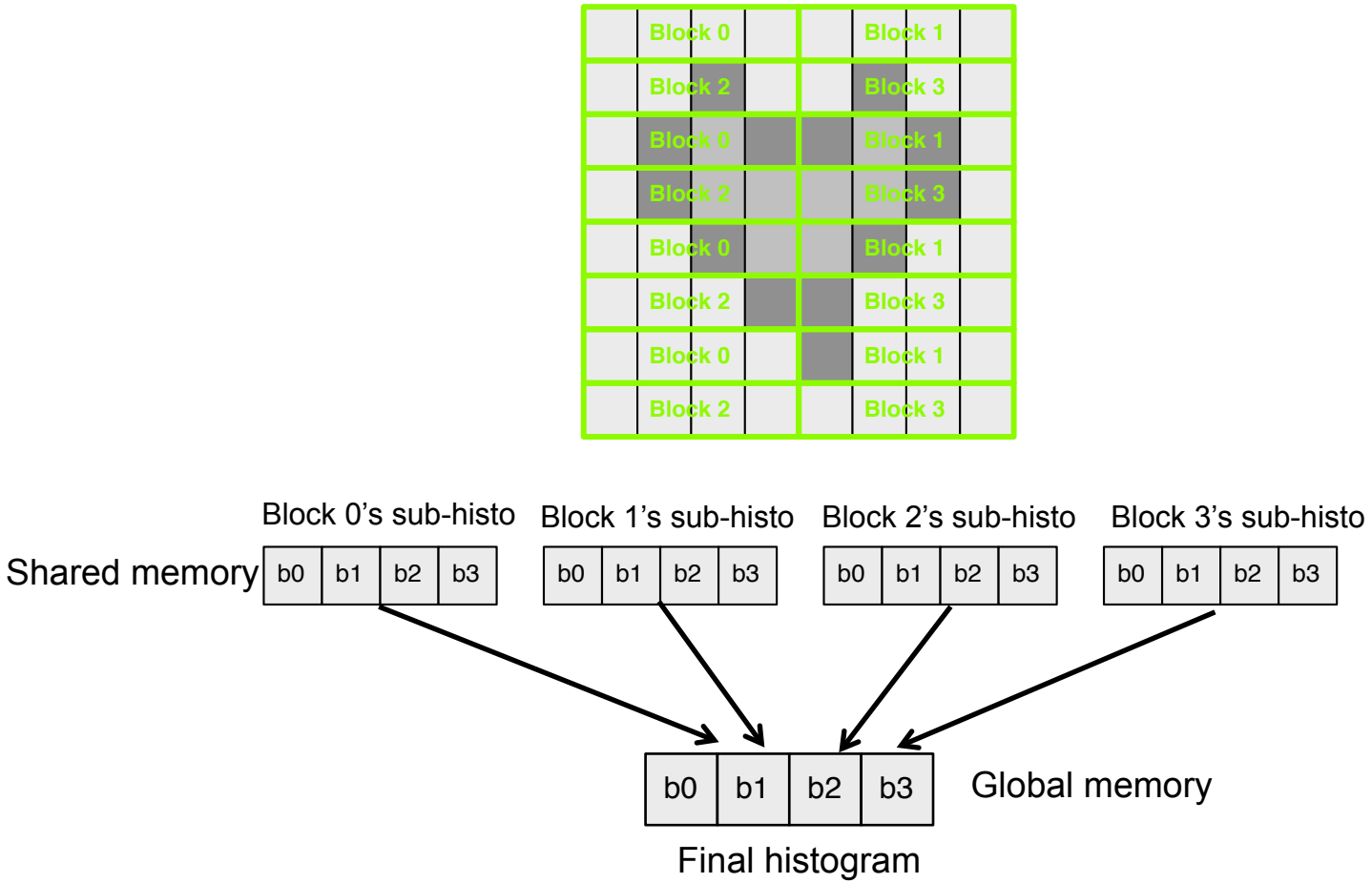
Histogram Calculation

- Frequent conflicts in natural images



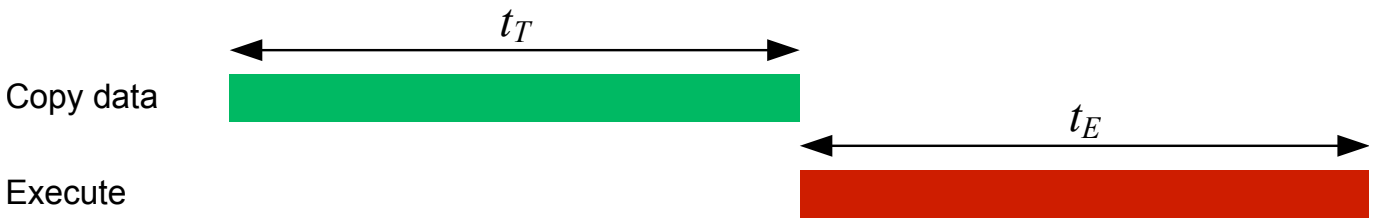
Histogram Calculation

- Privatization: Per-block sub-histograms in shared memory



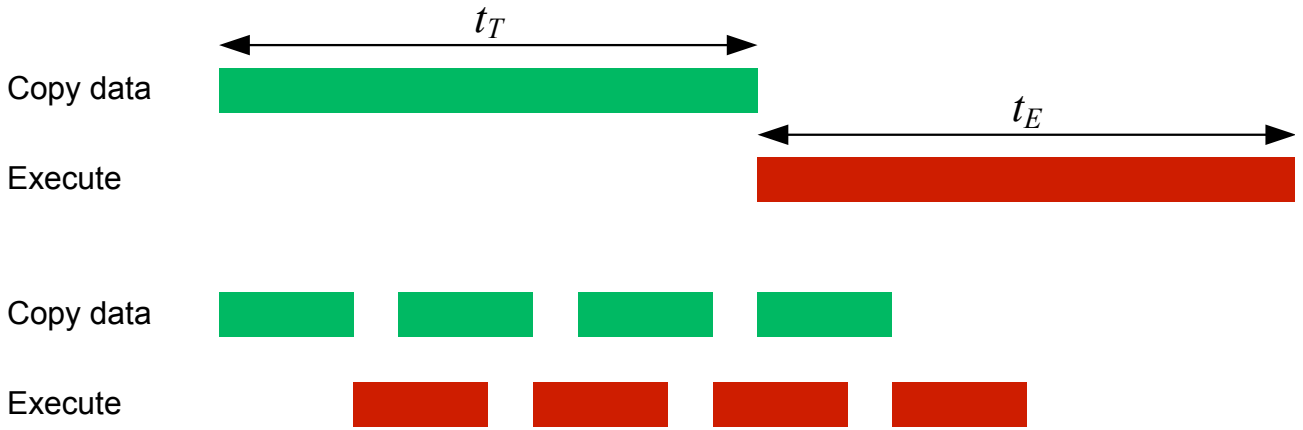
Data Transfers

- Synchronous and asynchronous transfers
- Streams (Command queues)
  - Sequence of operations that are performed in order
    - CPU-GPU data transfer
    - Kernel execution
      - D input data instances, B blocks
    - GPU-CPU data transfer
  - Default stream



Asynchronous Transfers

- Computation divided into nStreams
  - D input data instances, B blocks
  - nStreams
    - D/nStreams data instances
    - B/nStreams blocks



  - Estimates

$$t_E + \frac{t_T}{nStreams}$$

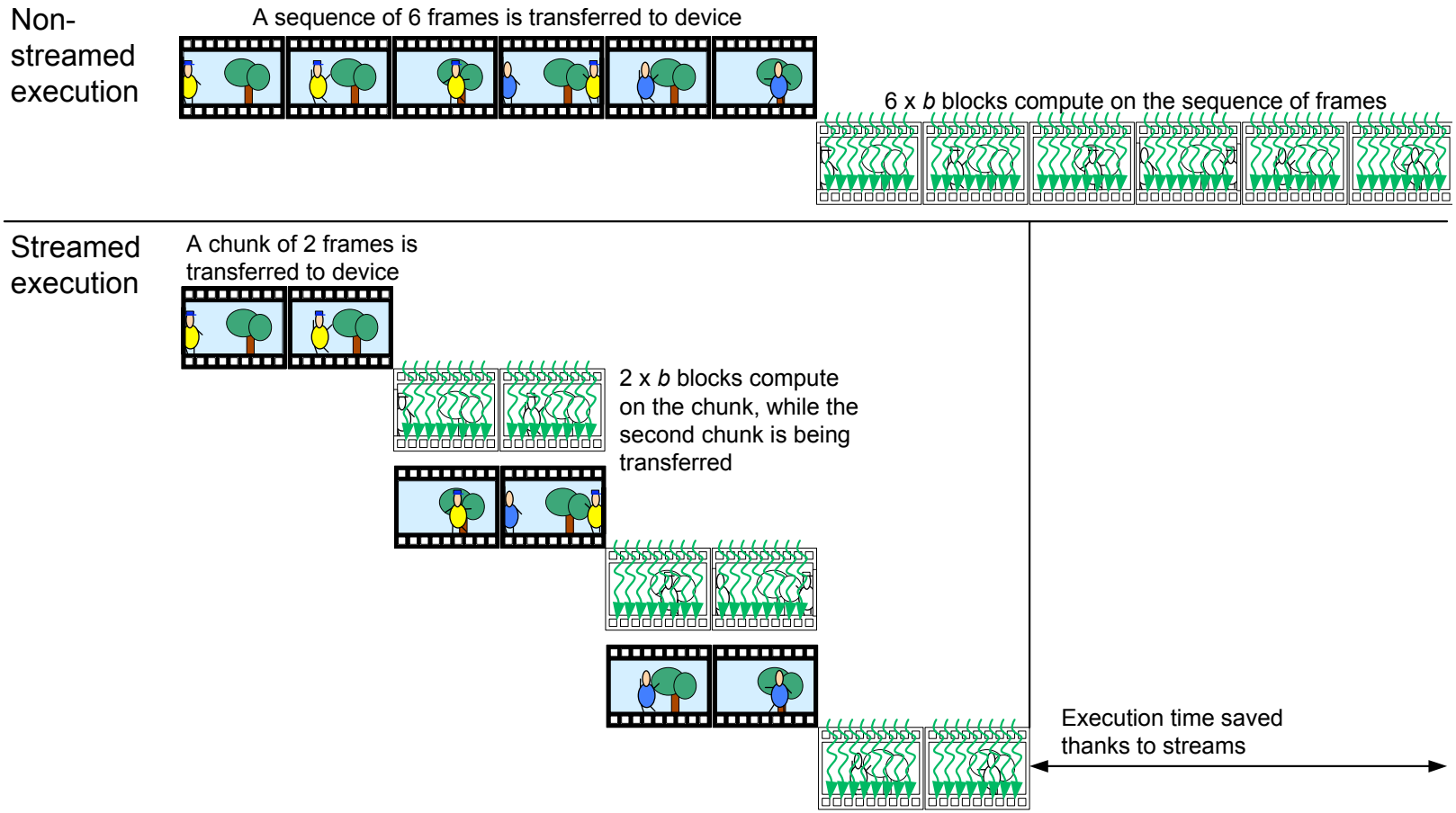
$t_E >= t_T$ (dominant kernel)

$$t_T + \frac{t_E}{nStreams}$$

$t_T > t_E$ (dominant transfers)

Asynchronous Transfers

- Overlap of communication and computation (e.g., video processing)



Summary

- Traditional accelerator model
  - Program structure
    - Bulk synchronous programming model
  - Memory hierarchy and memory management
  - Performance considerations
    - Memory access
      - Latency hiding: occupancy (TLP)
      - Memory coalescing
      - Data reuse: shared memory
    - SIMD utilization
    - Atomic operations
    - Data transfers

Computer Architecture

Lecture 9: GPUs and GPGPU Programming

Prof. Onur Mutlu
ETH Zürich

Fall 2017

19 October 2017