

Design of Digital Circuits

Lecture 22: GPU Programming

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zurich

Spring 2018

18 May 2018

Agenda for Today

- GPU as an accelerator
 - Program structure
 - Bulk synchronous programming model
 - Memory hierarchy and memory management
 - Performance considerations
 - Memory access
 - SIMD utilization
 - Atomic operations
 - Data transfers

Recommended Readings

- CUDA Programming Guide
 - <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- Hwu and Kirk, “Programming Massively Parallel Processors,” Third Edition, 2017

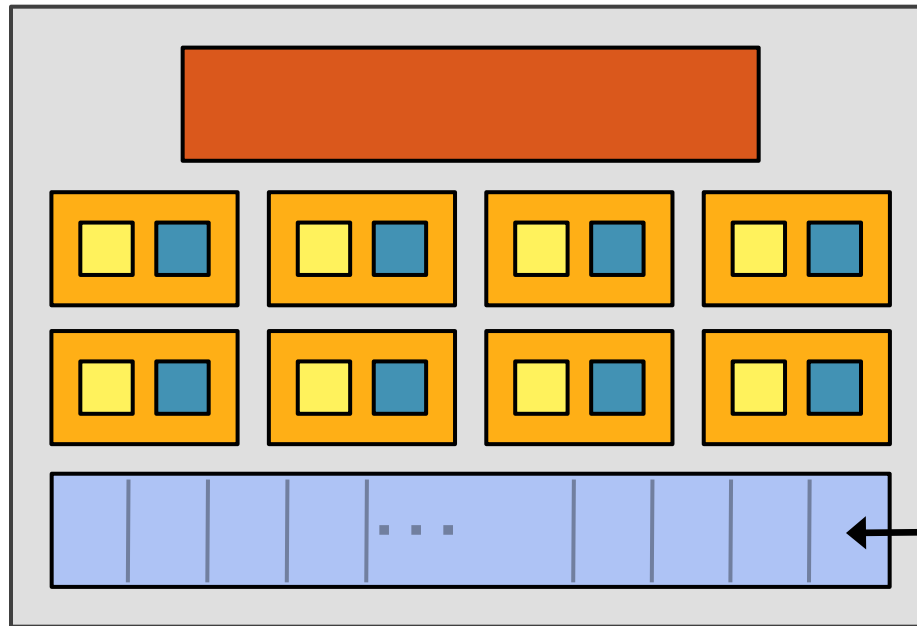
An Example GPU

NVIDIA GeForce GTX 285

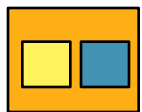
- NVIDIA-speak:
 - ❑ 240 stream processors
 - ❑ “SIMT execution”
- Generic speak:
 - ❑ 30 cores
 - ❑ 8 SIMD functional units per core



NVIDIA GeForce GTX 285 “core” (I)



64 KB of storage
for thread contexts
(registers)



= SIMD functional unit, control
shared across 8 units

■ = multiply-add
■ = multiply

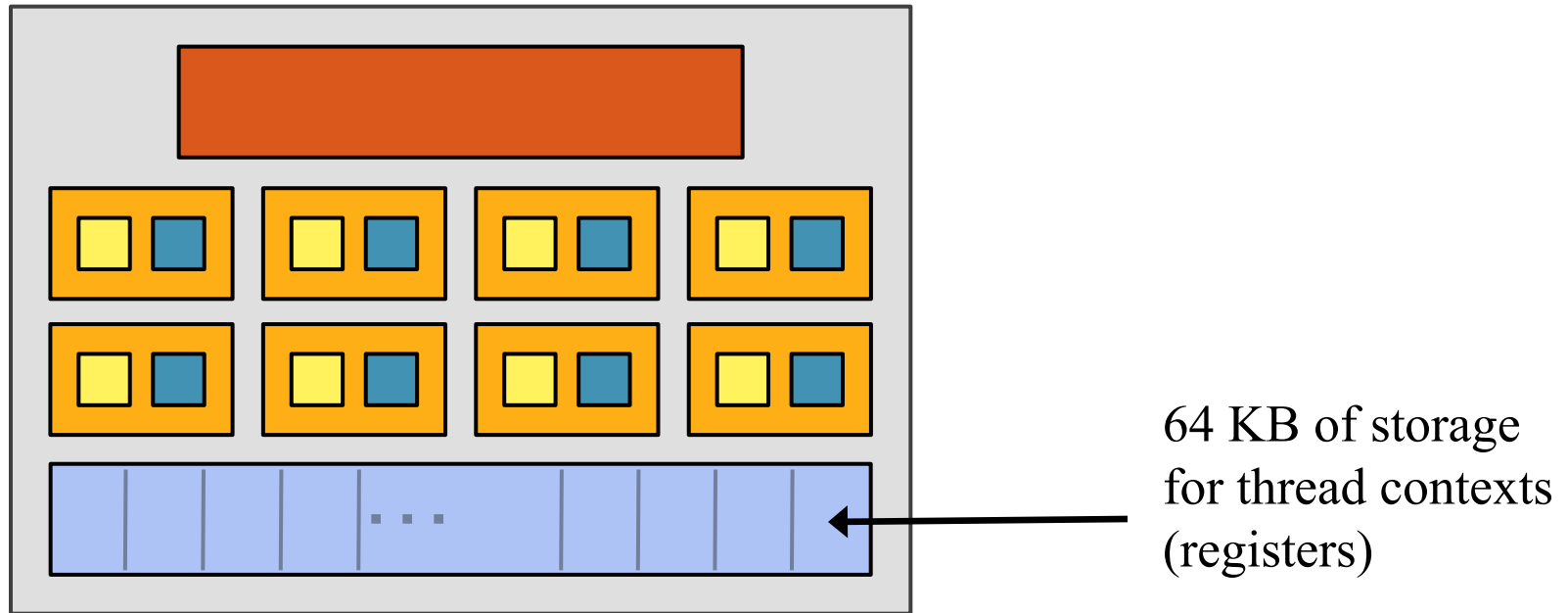


= instruction stream decode



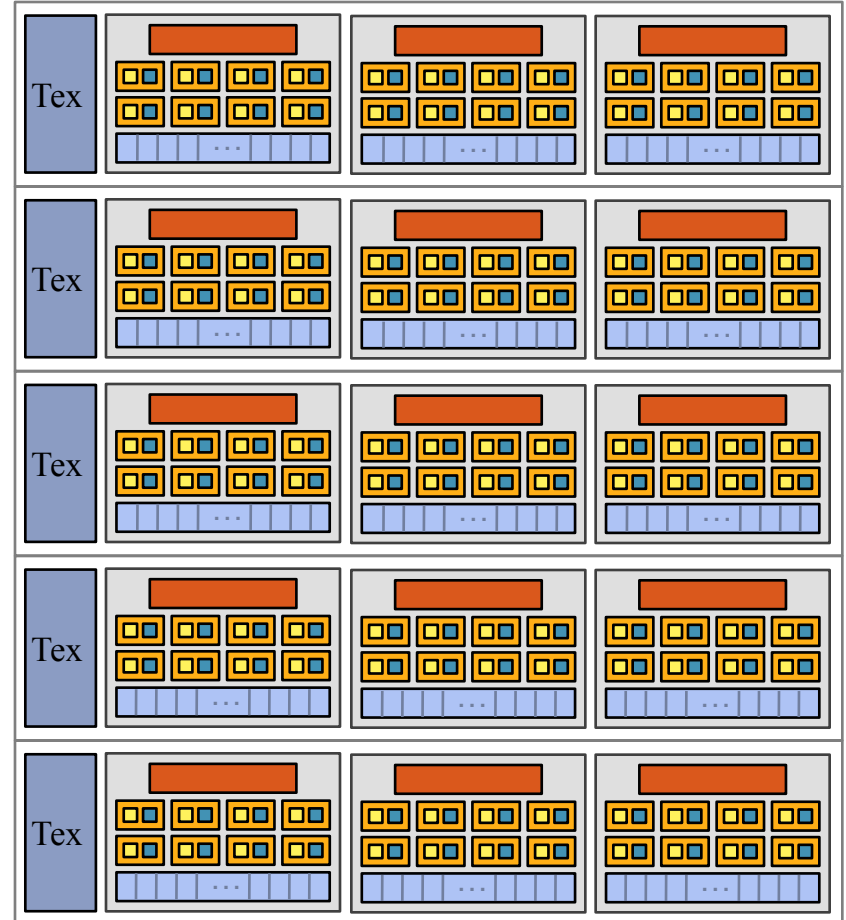
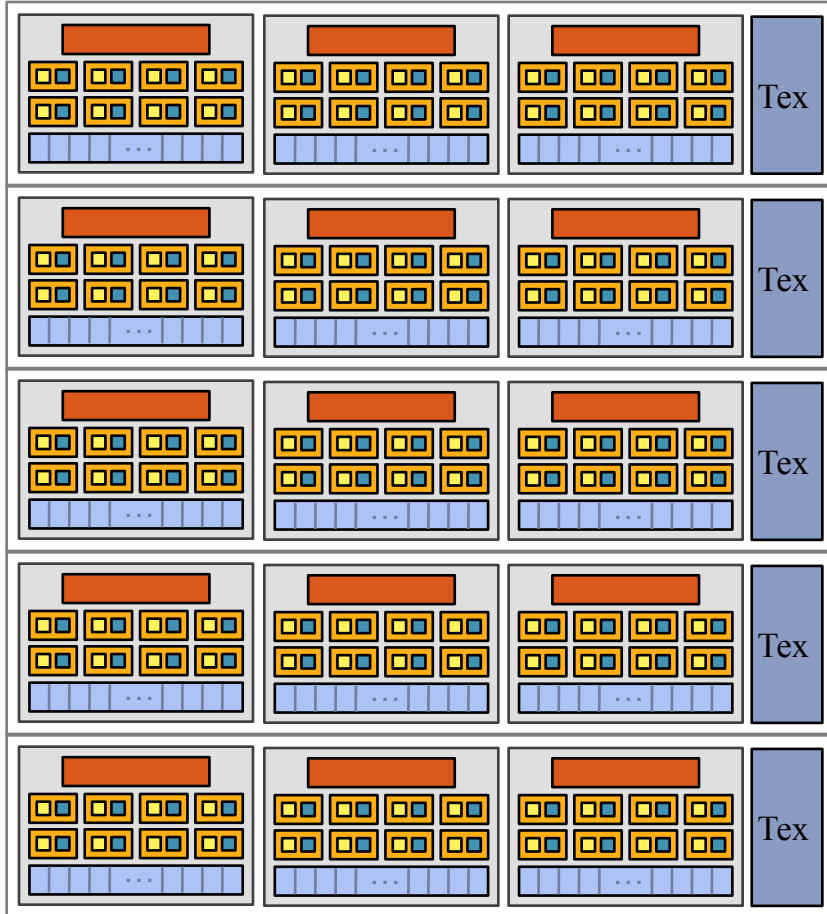
= execution context storage

NVIDIA GeForce GTX 285 “core” (II)



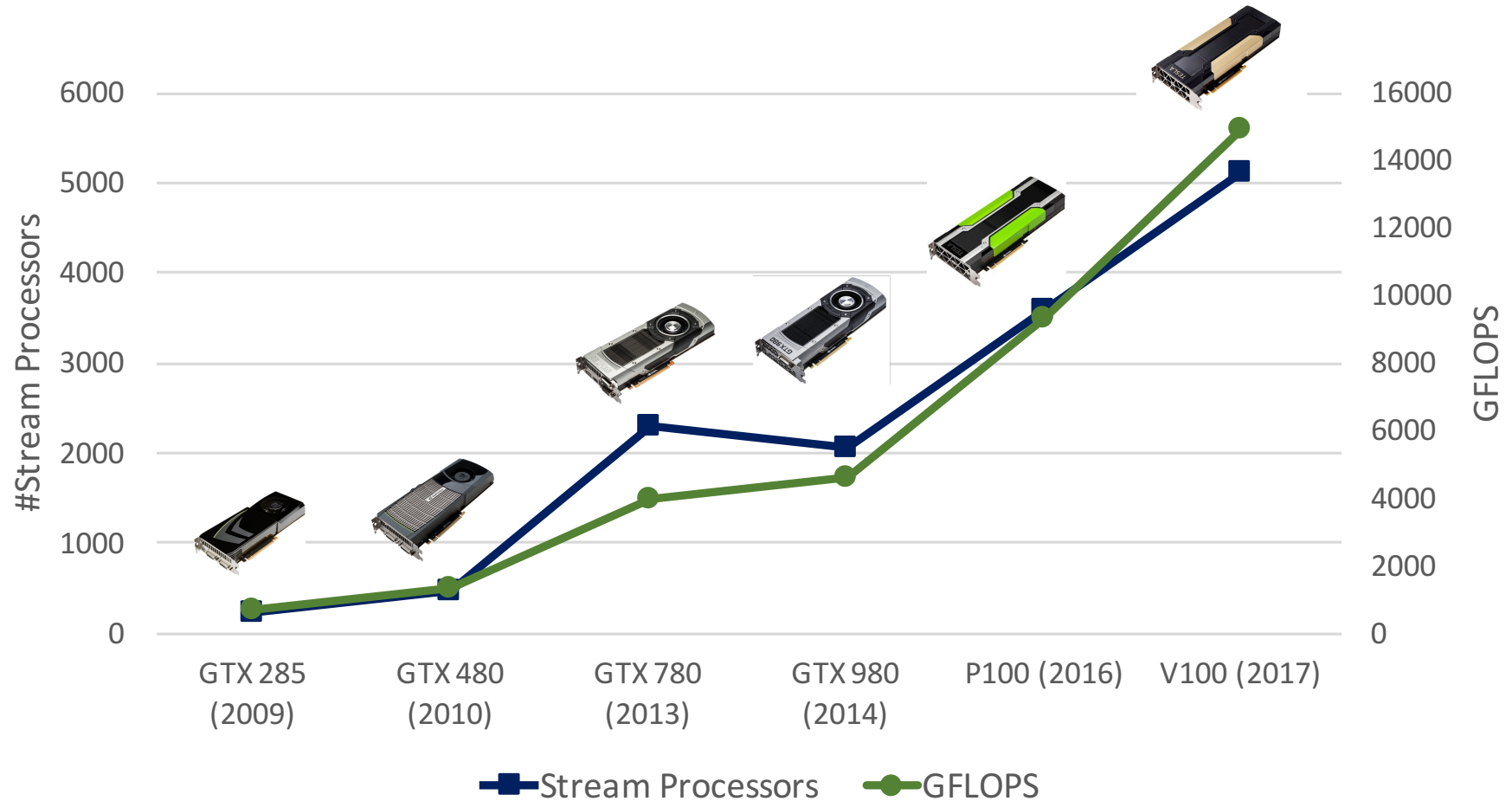
- Groups of 32 **threads** share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

Evolution of NVIDIA GPUs



NVIDIA V100

- NVIDIA-speak:
 - ❑ 5120 stream processors
 - ❑ “SIMT execution”
- Generic speak:
 - ❑ 80 cores
 - ❑ 64 SIMD functional units per core
 - ❑ Specialized Functional Units for Machine Learning (tensor “cores” in NVIDIA-speak)



NVIDIA V100 Block Diagram



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

80 cores on the V100

NVIDIA V100 Core

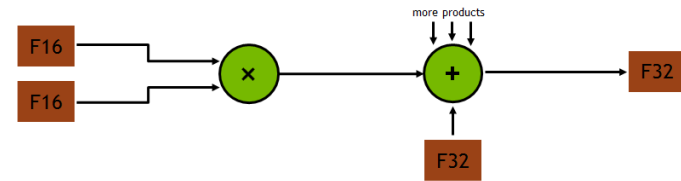


15.7 TFLOPS Single Precision

7.8 TFLOPS Double Precision

125 TFLOPS for Deep Learning (Tensor "cores")

FP16 storage/input Full precision product Sum with FP32 accumulator Convert to FP32 result



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

FP16 or FP32 FP16 FP16 FP16 or FP32

GPU Programming

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?

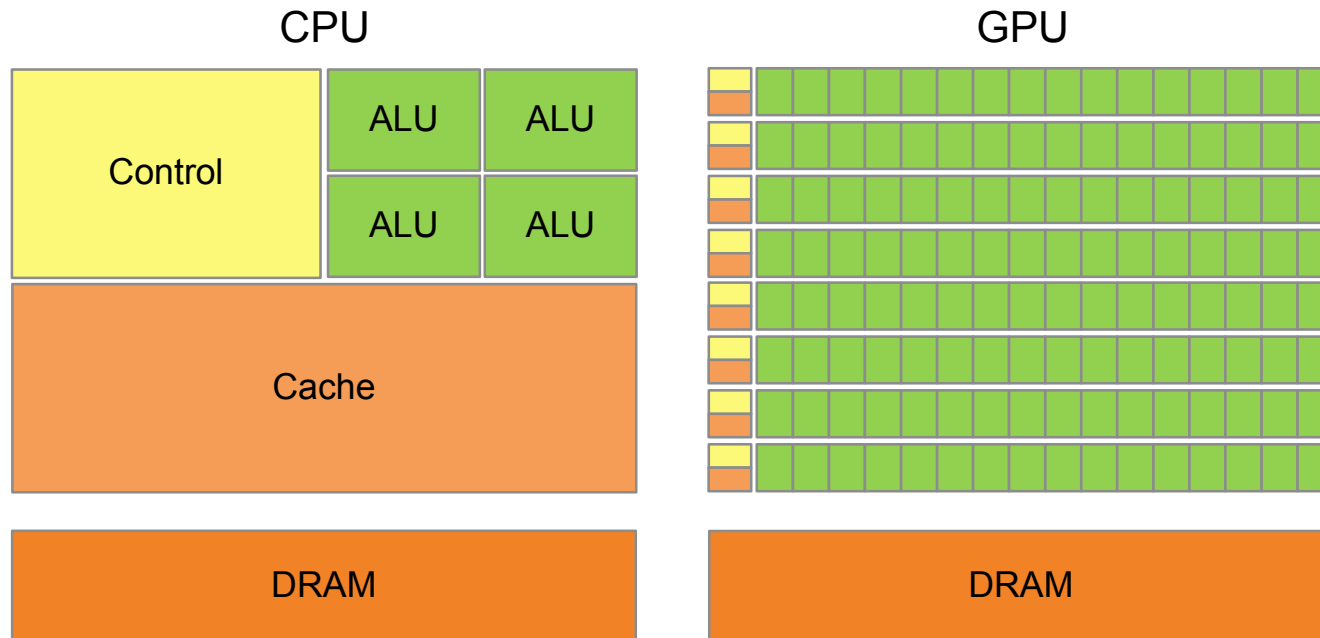
To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

General Purpose Processing on GPU

- Easier programming of SIMD processors with SPMD
 - GPUs have democratized High Performance Computing (HPC)
 - Great FLOPS/\$, massively parallel chip on a commodity PC
- Many workloads exhibit inherent parallelism
 - Matrices
 - Image processing
- However, this is not for free
 - New programming model
 - Algorithms need to be re-implemented and rethought
- Still some bottlenecks
 - CPU-GPU data transfers (PCIe, NVLINK)
 - DRAM memory bandwidth (GDDR5, HBM2)
 - Data layout

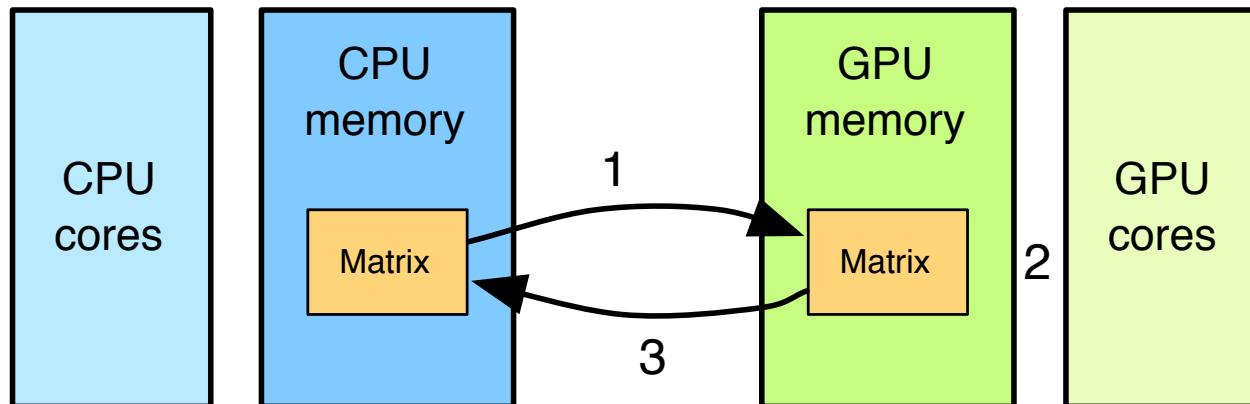
CPU vs. GPU

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



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

Serial Code (host)

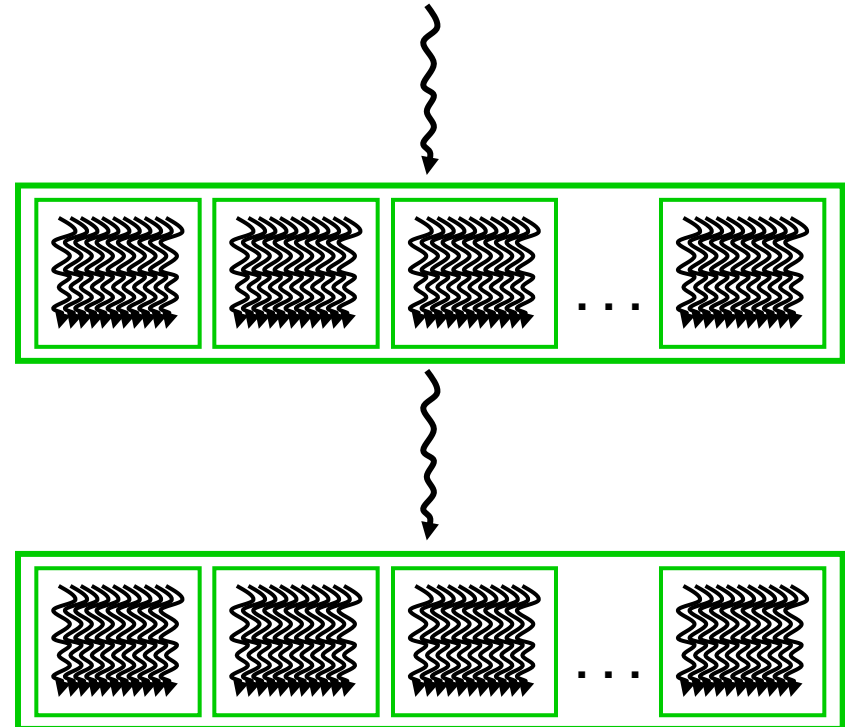
Parallel Kernel (device)

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

Serial Code (host)

Parallel Kernel (device)

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



Recall: SPMD

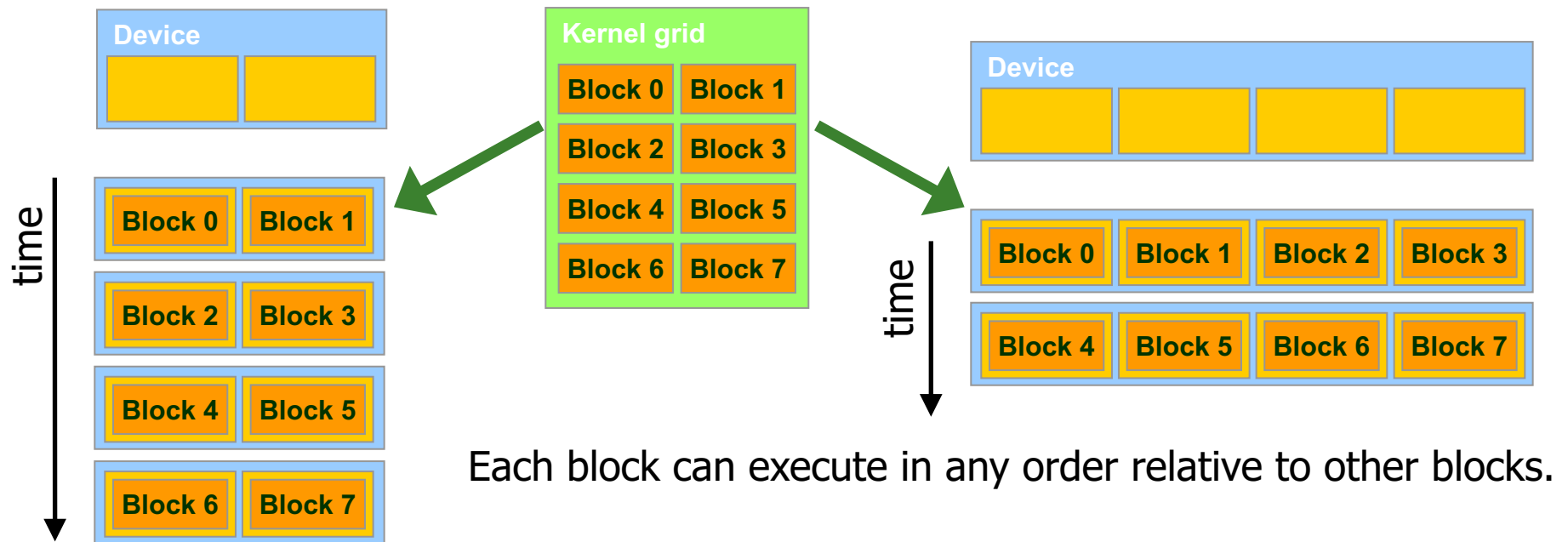
- Single procedure/program, multiple data
 - This is a programming model rather than computer organization
- Each processing element executes the same procedure, except on different data elements
 - Procedures can synchronize at certain points in program, e.g. barriers
- Essentially, multiple instruction 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

CUDA/OpenCL Programming Model

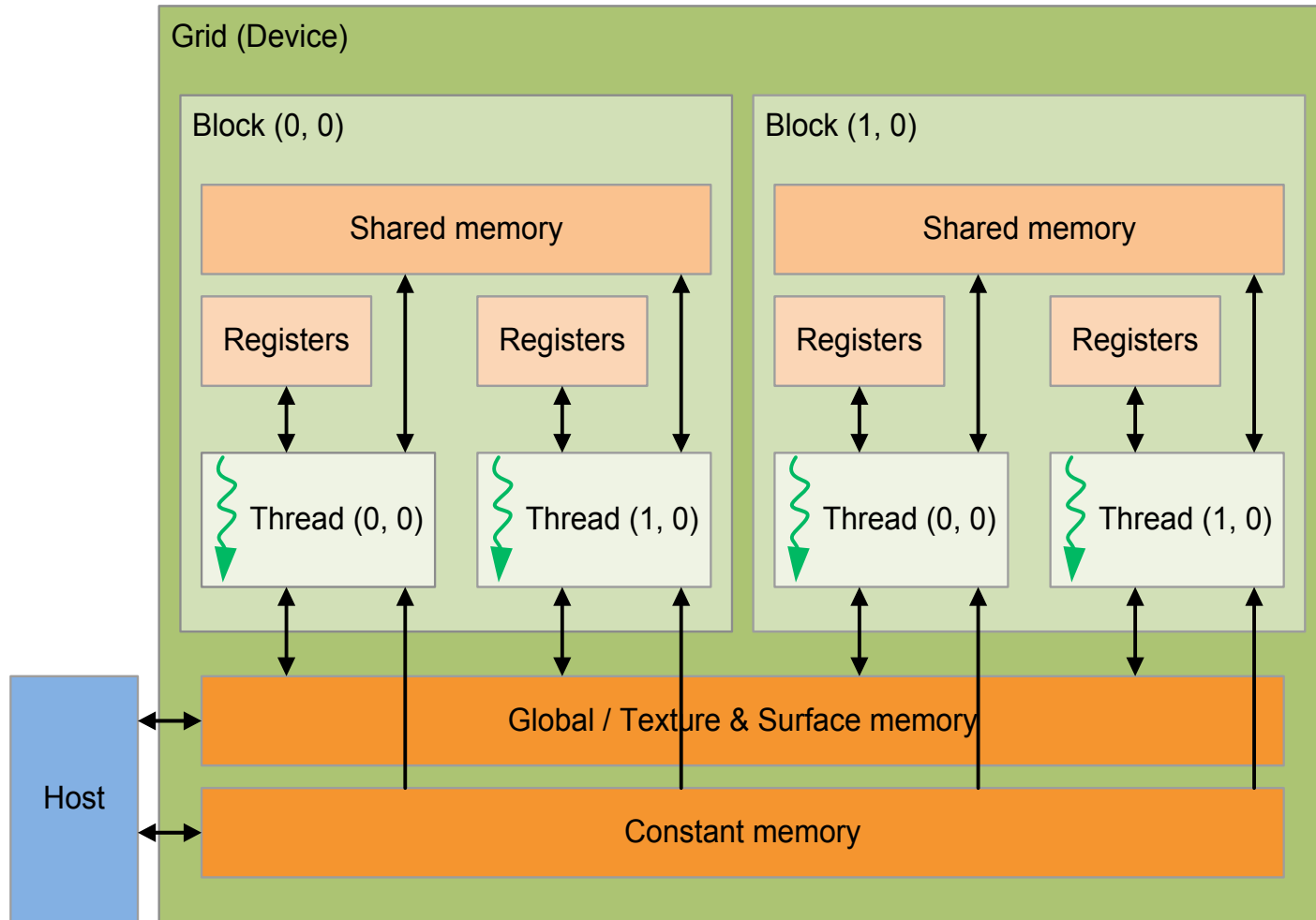
- SIMT or SPMD
- Bulk synchronous programming
 - Global (coarse-grain) synchronization between kernels
- The host (typically CPU) allocates memory, copies data, and launches kernels
- The device (typically GPU) executes kernels
 - Grid (NDRange)
 - Block (work-group)
 - Within a block, shared memory,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

Transparent Scalability

- Hardware is **free to schedule** thread blocks



Memory Hierarchy




Traditional Program Structure in CUDA

■ 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, ...);`



repeat
as needed

■ Kernel – `__global__ void kernel(type args,...)`

- ❑ Automatic variables transparently assigned to **registers**
- ❑ **Shared memory**: `__shared__`
- ❑ Intra-block **synchronization**: `__syncthreads()`;

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

- Images are 2D data structures
 - height x width
 - $\text{Image}[j][i]$, where $0 \leq j < \text{height}$, and $0 \leq i < \text{width}$

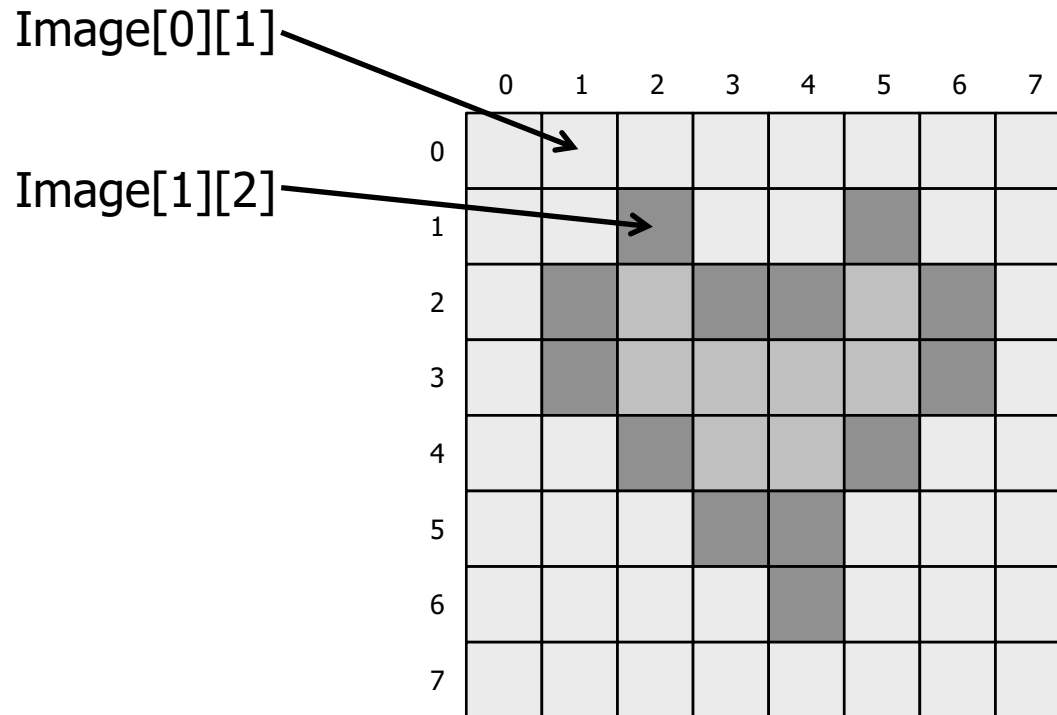
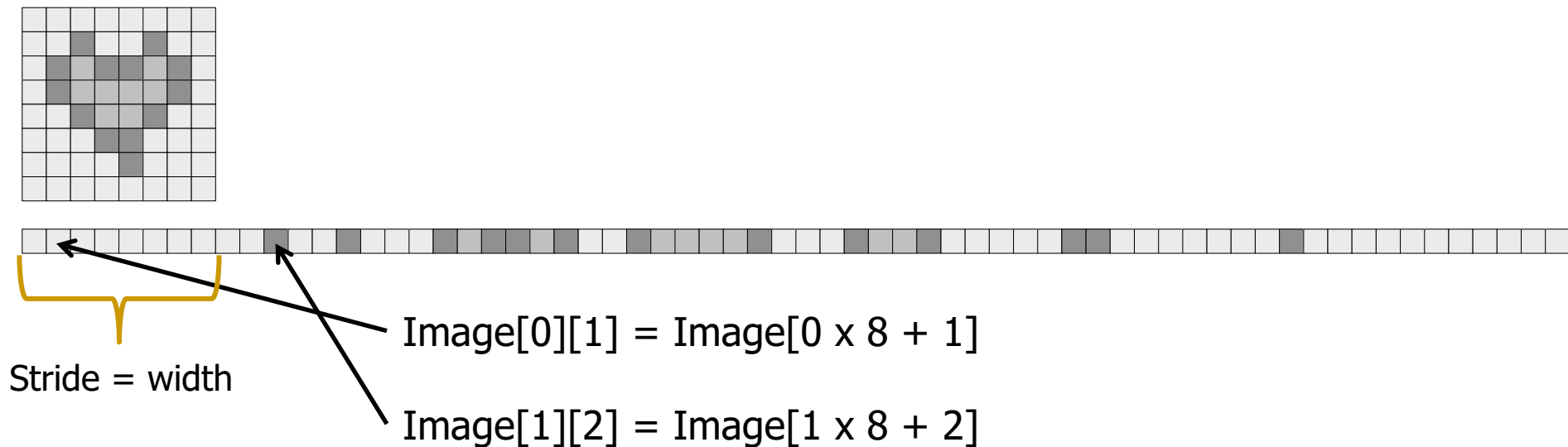


Image Layout in Memory

- Row-major layout
- $\text{Image}[j][i] = \text{Image}[j \times \text{width} + i]$



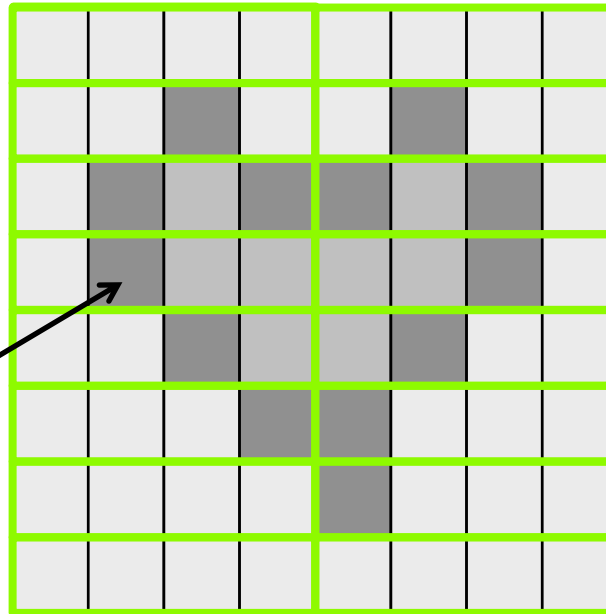
Indexing and Memory Access: 1D Grid

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

`blockIdx.x`

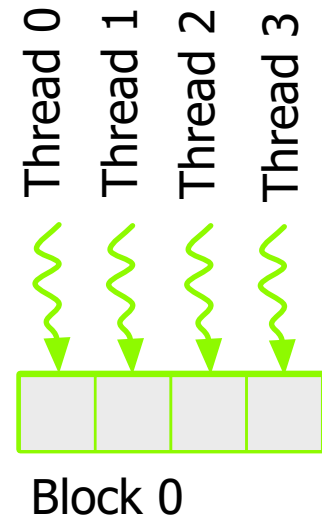
`threadIdx.x`

Block 0



$$6 * 4 + 1 = 25$$

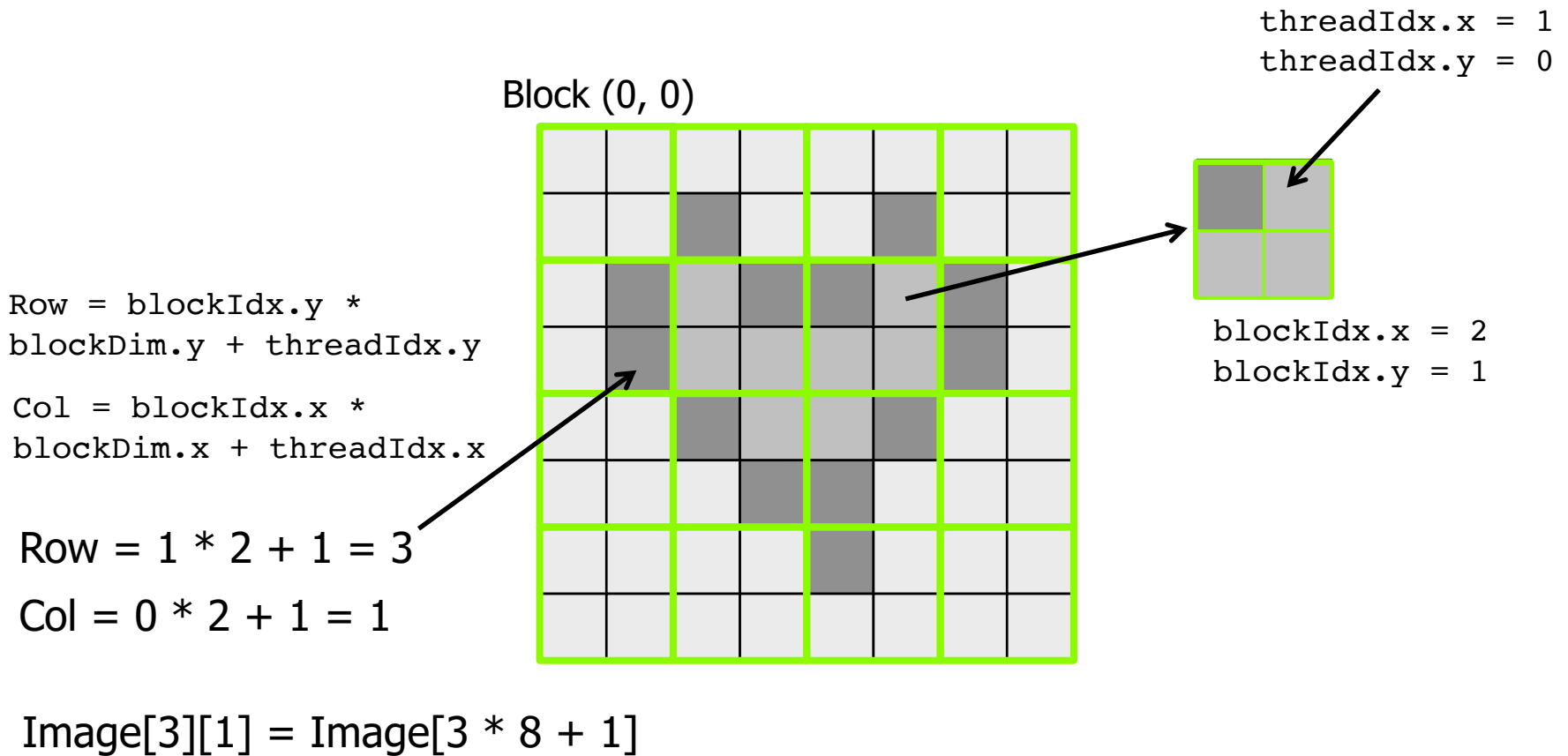
`blockIdx.x * blockDim.x + threadIdx.x`



Indexing and Memory Access: 2D Grid

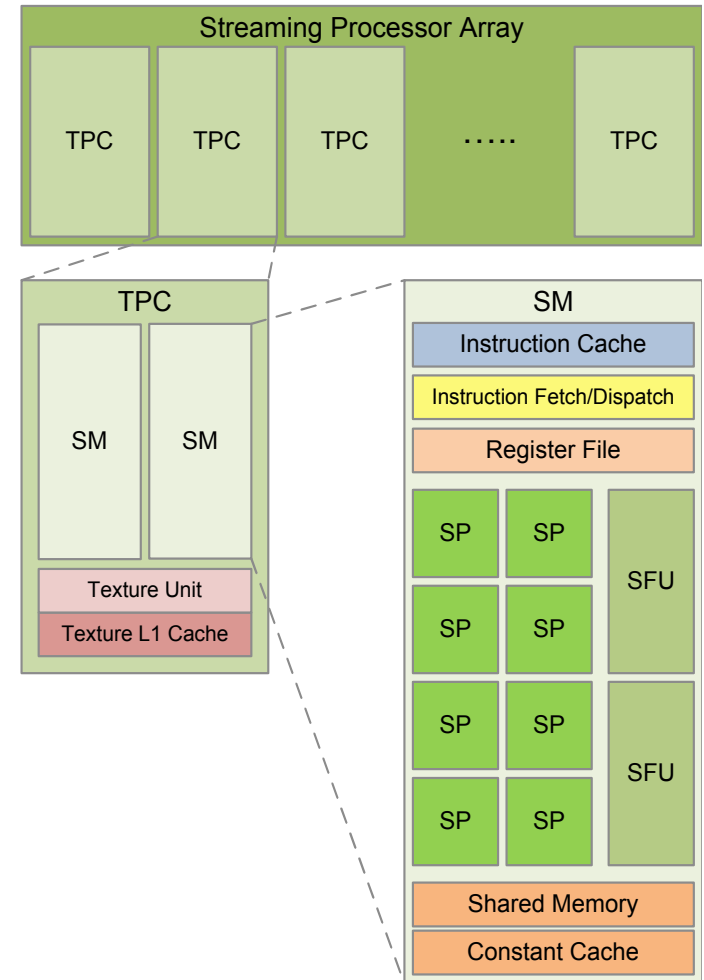
■ 2D blocks

□ `gridDim.x`, `gridDim.y`



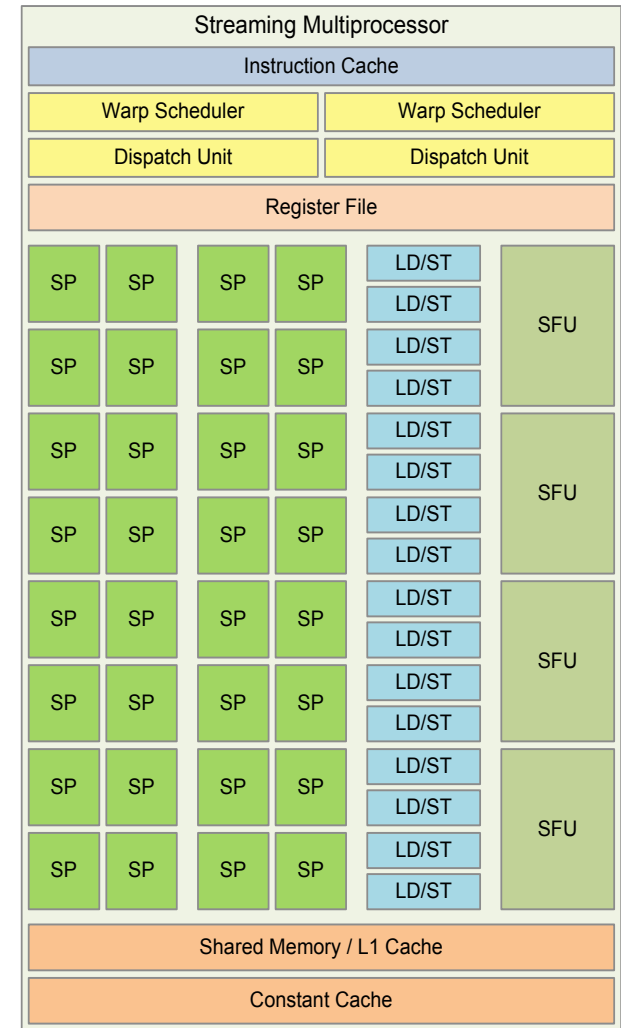
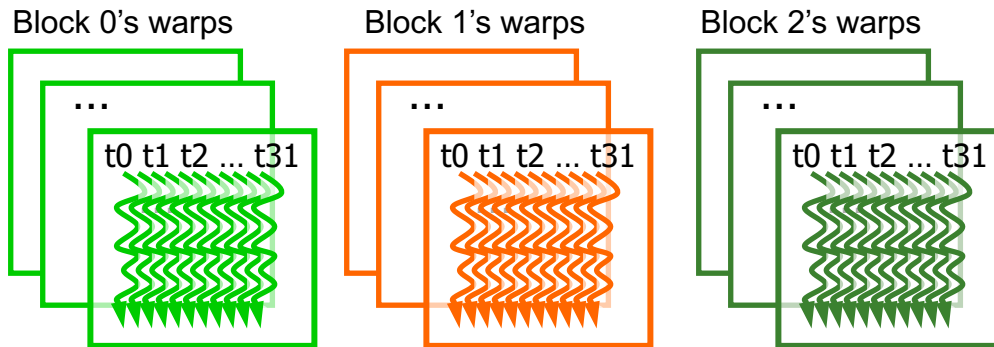
Brief Review of GPU Architecture (I)

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



Brief Review of GPU Architecture (II)

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



NVIDIA Fermi architecture

Brief Review of GPU Architecture (III)

- Streaming Multiprocessors (SM) or Compute Units (CU)
 - SIMD pipelines
- Streaming Processors (SP) or CUDA "cores"
 - Vector lanes
- Number of SMs x SPs across generations
 - 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

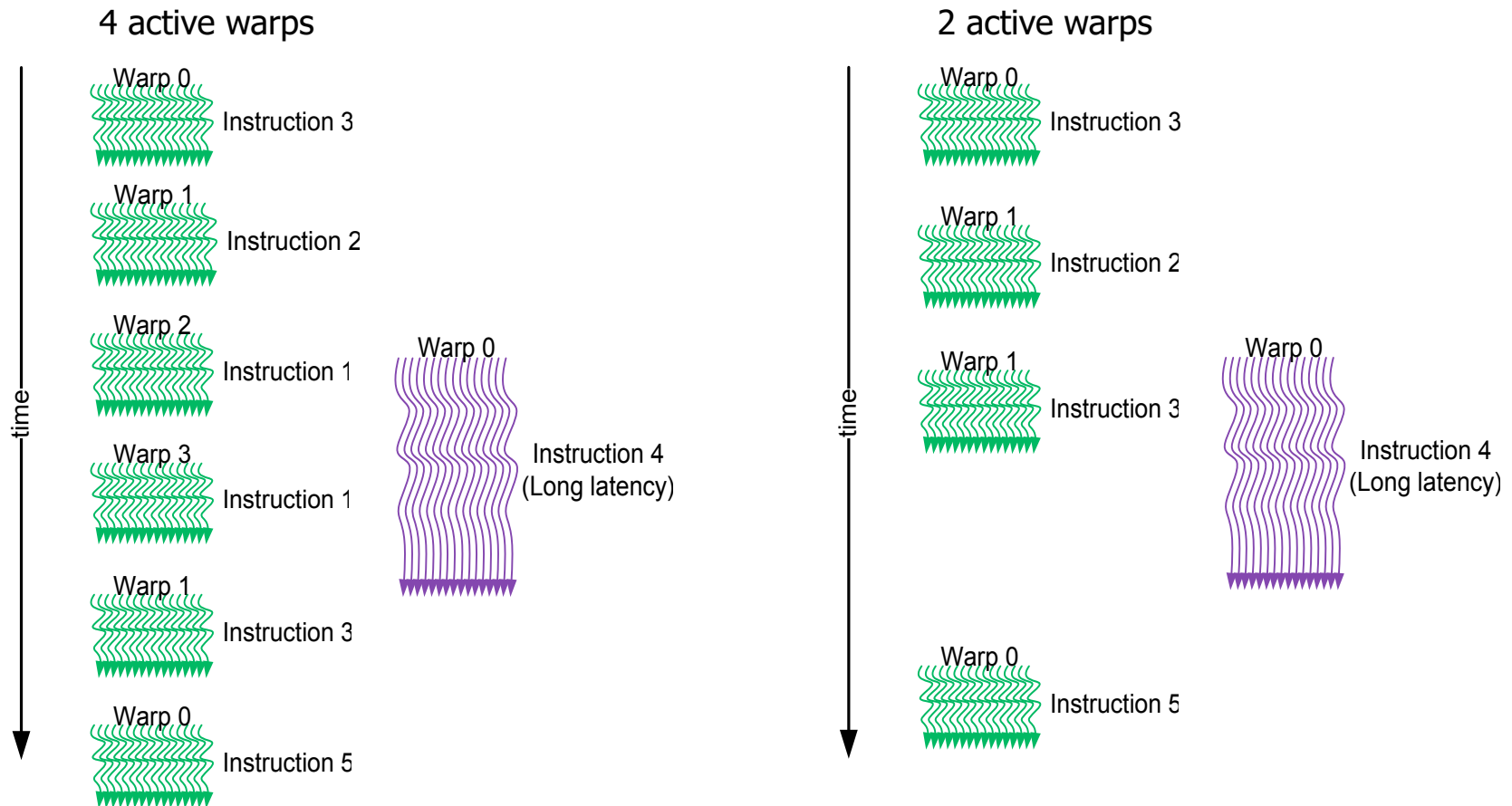
Performance Considerations

- Main bottlenecks
 - ❑ Global memory access
 - ❑ CPU-GPU data transfers
- Memory access
 - ❑ Latency hiding
 - Occupancy
 - ❑ Memory coalescing
 - ❑ Data reuse
 - Shared memory usage
- SIMD (Warp) Utilization: Divergence
- Atomic operations: Serialization
- Data transfers between CPU and GPU
 - ❑ Overlap of communication and computation

Memory Access

Latency Hiding

- FGMT can hide **long latency operations** (e.g., memory accesses)
- **Occupancy**: ratio of active warps

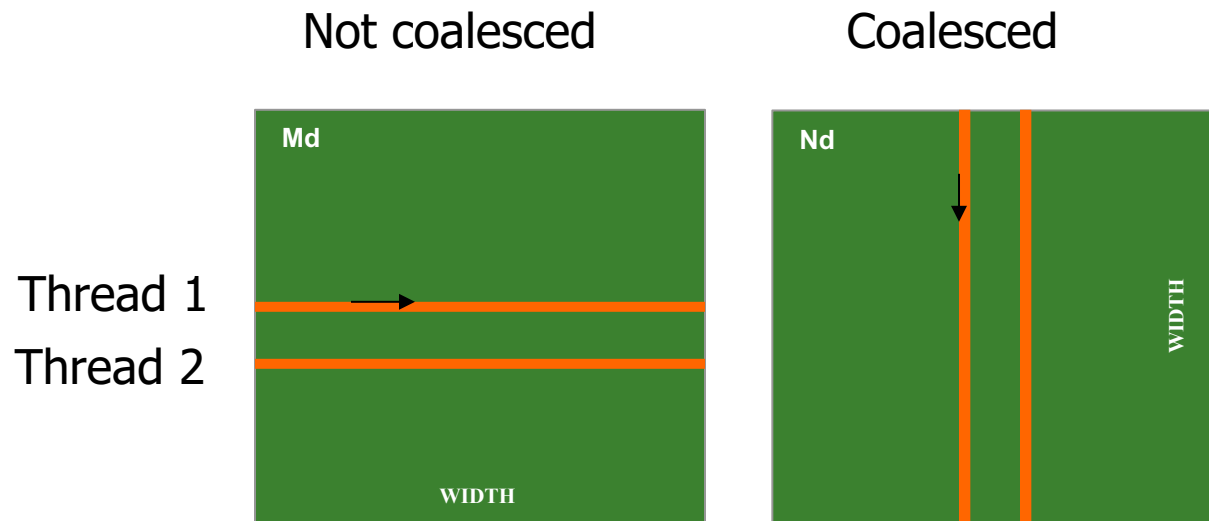


Occupancy

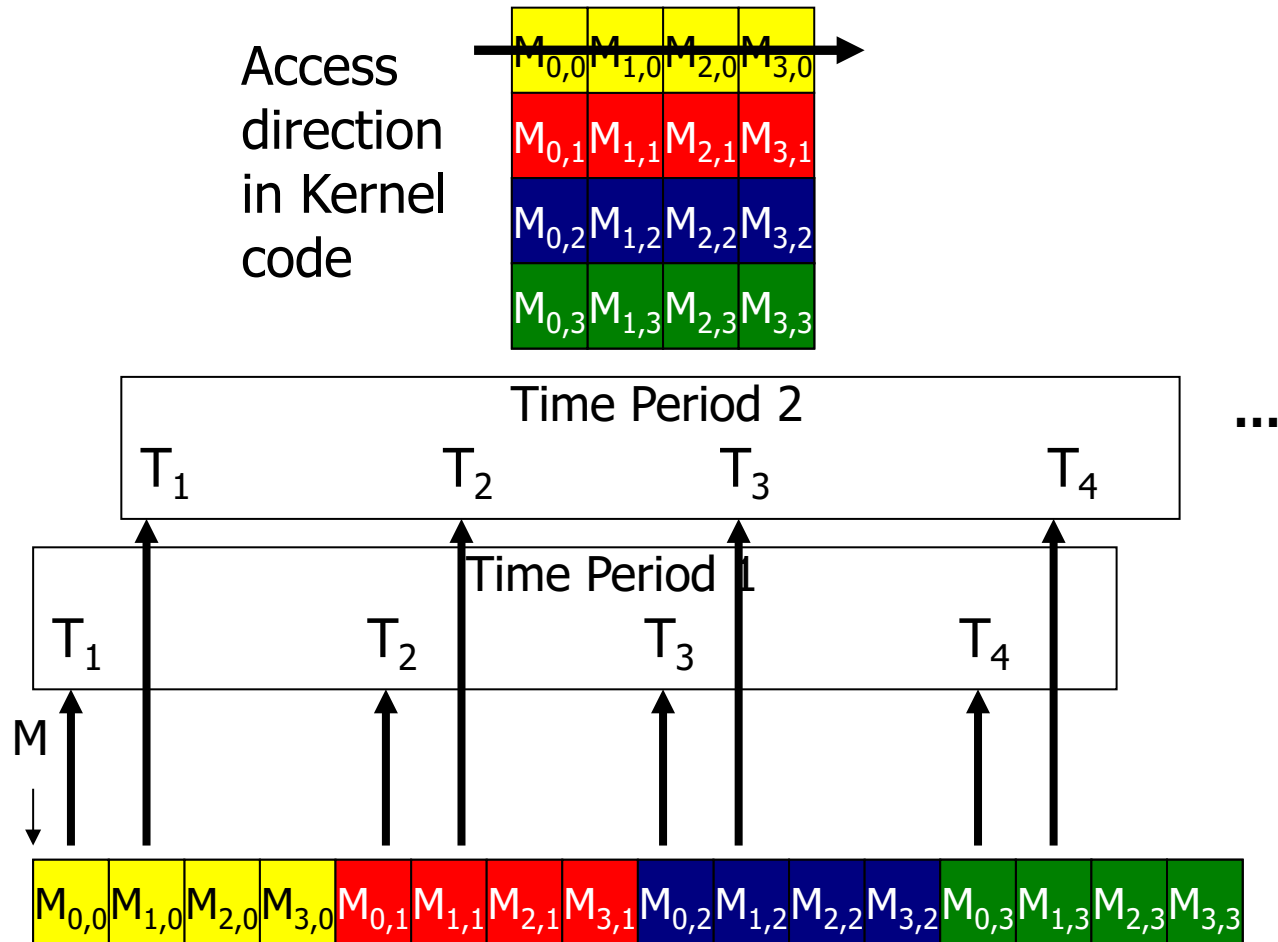
- SM resources (typical values)
 - ❑ Maximum number of warps per SM (64)
 - ❑ Maximum number of blocks per SM (32)
 - ❑ Register usage (256KB)
 - ❑ Shared memory usage (64KB)
- Occupancy calculation
 - ❑ Number of threads per block (defined by the programmer)
 - ❑ Registers per thread (known at compile time)
 - ❑ Shared memory per block (defined by the programmer)

Memory Coalescing

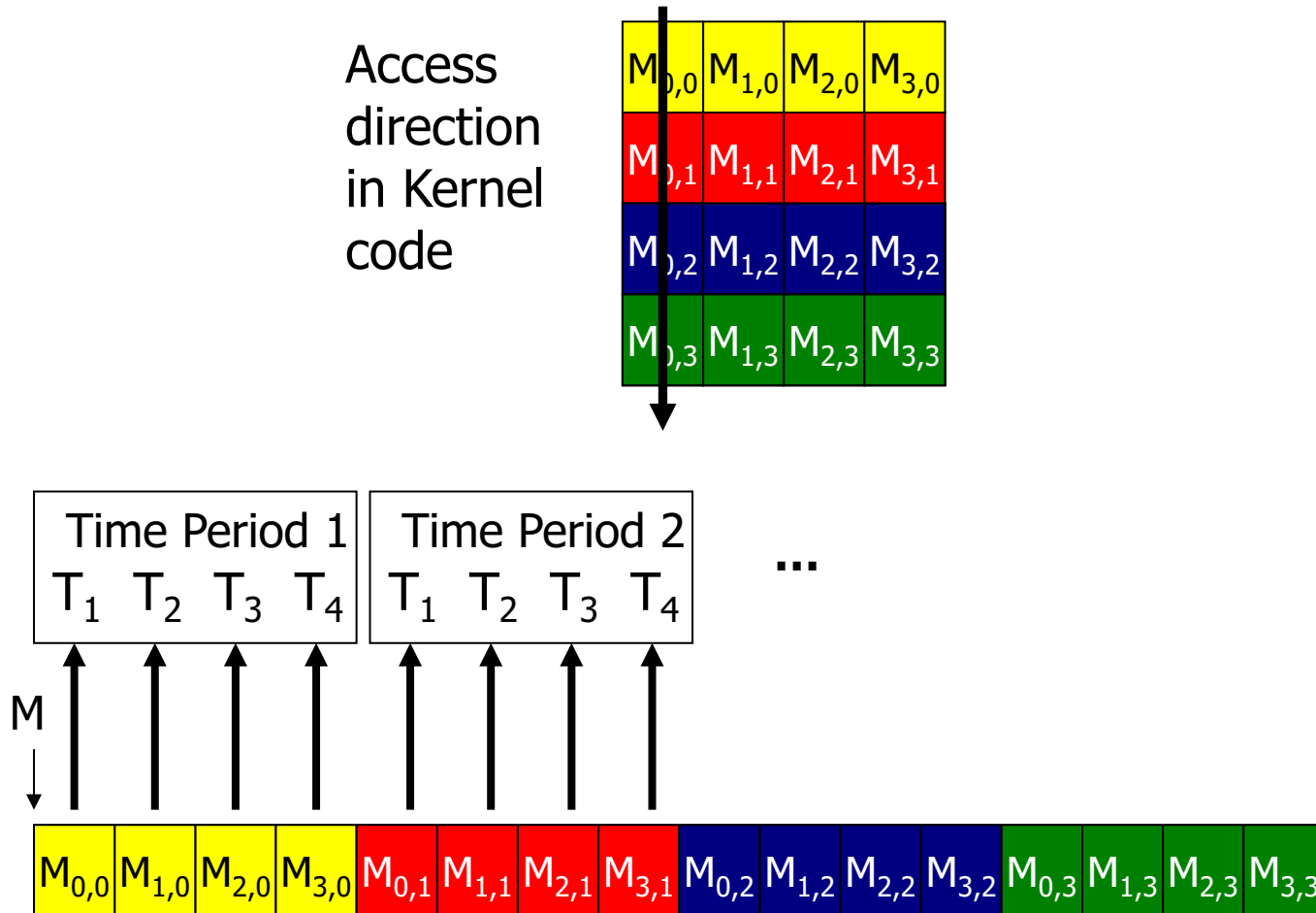
- When accessing global memory, we want to make sure that **concurrent threads access nearby memory locations**
- **Peak bandwidth** utilization occurs when all threads in a warp access **one cache line**



Uncoalesced Memory Accesses



Coalesced Memory Accesses

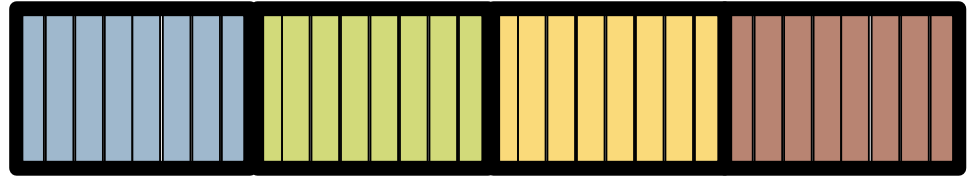


AoS vs. SoA

■ Array of Structures vs. Structure of Arrays

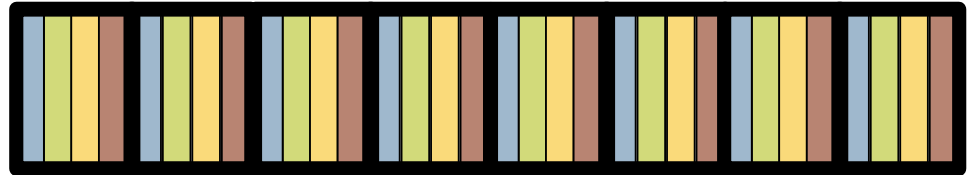
Structure of
Arrays
(SoA)

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



Array of
Structures
(AoS)

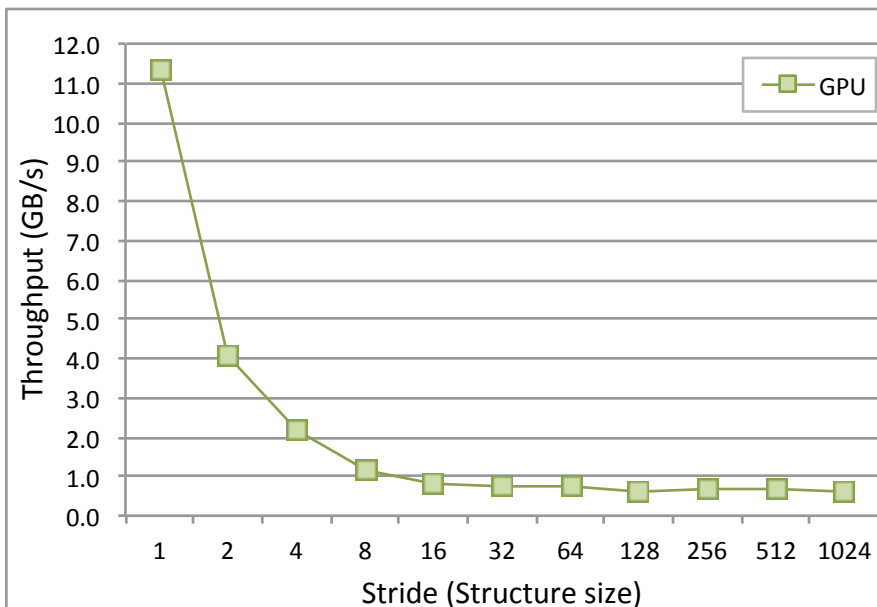
```
struct foo{  
    float a;  
    float b;  
    float c;  
    int d;  
} A[8];
```



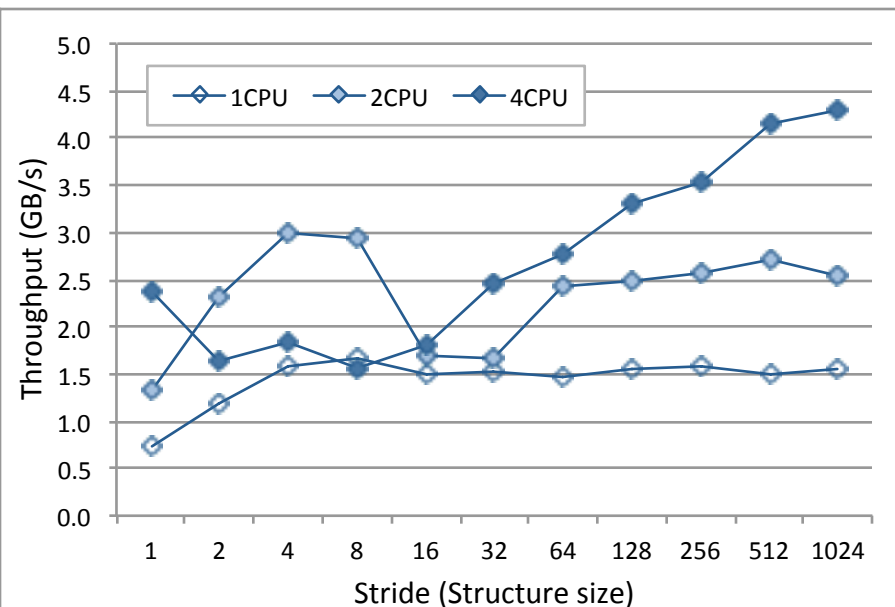
CPU's Prefer AoS, GPU's Prefer SoA

■ Linear and strided accesses

GPU



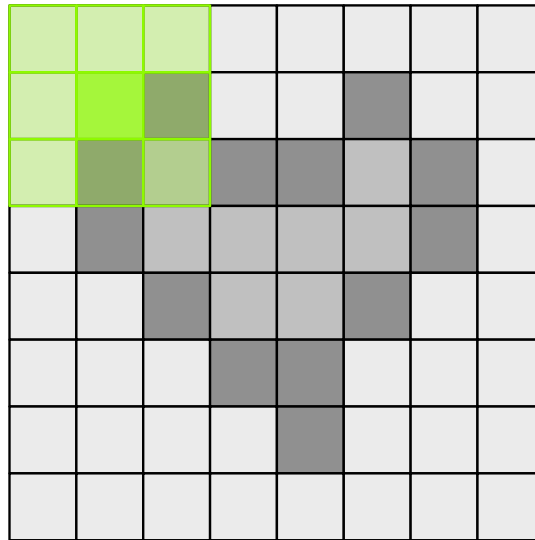
CPU



AMD Kaveri A10-7850K

Data Reuse

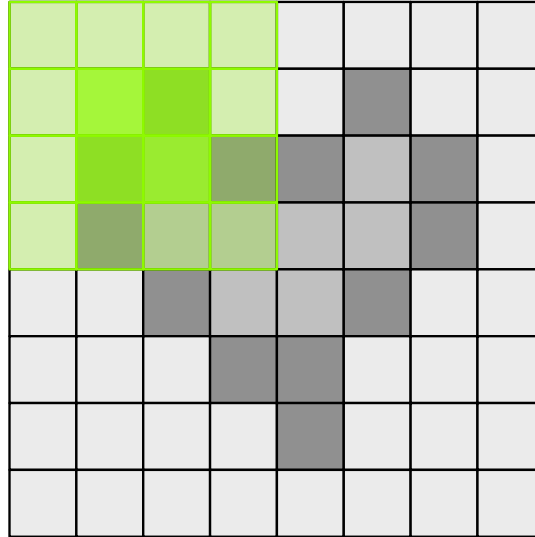
- Same memory locations accessed by neighboring threads



```
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: Tiling

- To take advantage of data reuse, we divide the input into **tiles** that can be loaded into **shared memory**



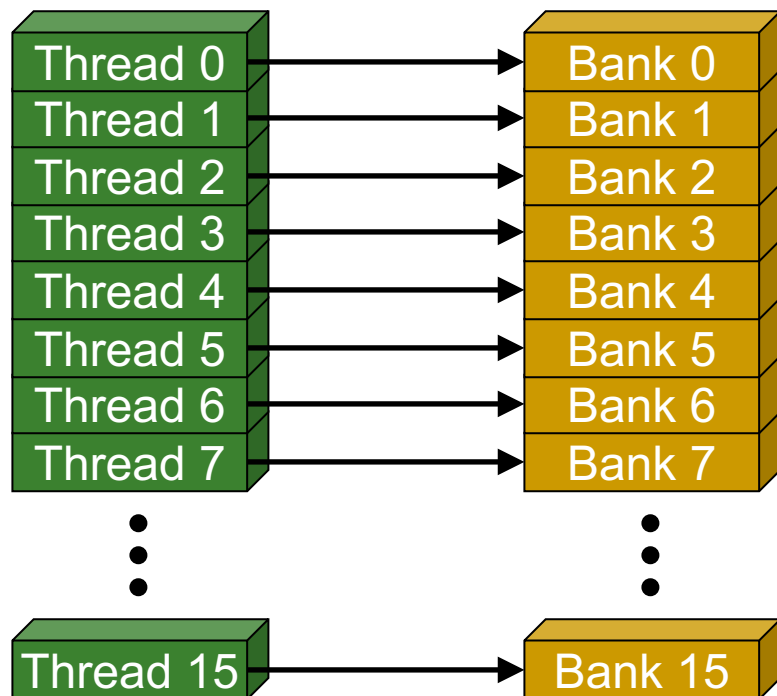
```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];  
...  
Load tile into shared memory  
__syncthreads();  
for (int i = 0; i < 3; i++){  
    for (int j = 0; j < 3; j++){  
        sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];  
    }  
}
```

Shared Memory

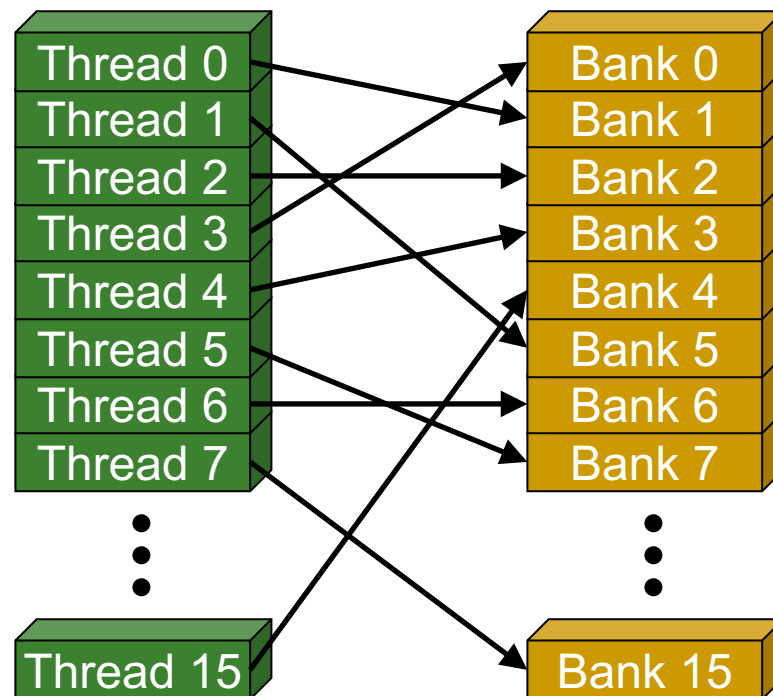
- Shared memory is an **interleaved (banked) memory**
 - Each bank can service one address per cycle
- Typically, 32 banks in NVIDIA GPUs
 - Successive 32-bit words are assigned to successive banks
 - $\text{Bank} = \text{Address} \% 32$
- Bank conflicts are **only possible within a warp**
 - No bank conflicts between different warps

Shared Memory Bank Conflicts (I)

- Bank conflict free



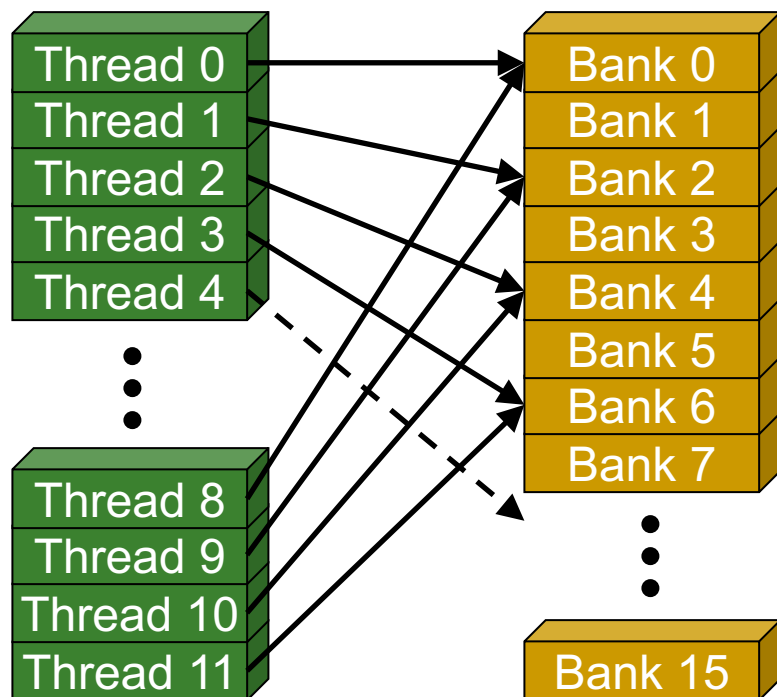
Linear addressing: stride = 1



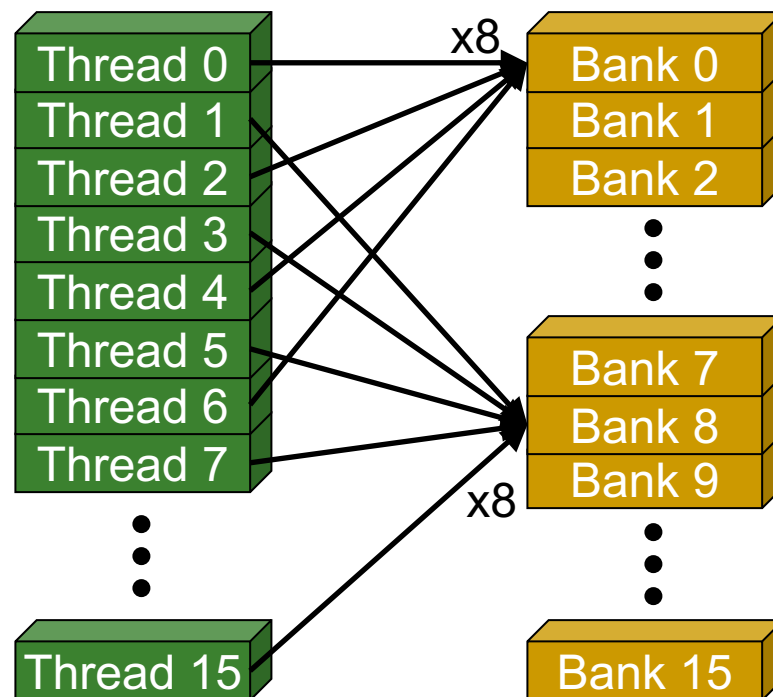
Random addressing 1:1

Shared Memory Bank Conflicts (II)

■ N-way bank conflicts



2-way bank conflict: stride = 2



8-way bank conflict: stride = 8

Reducing Shared Memory Bank Conflicts

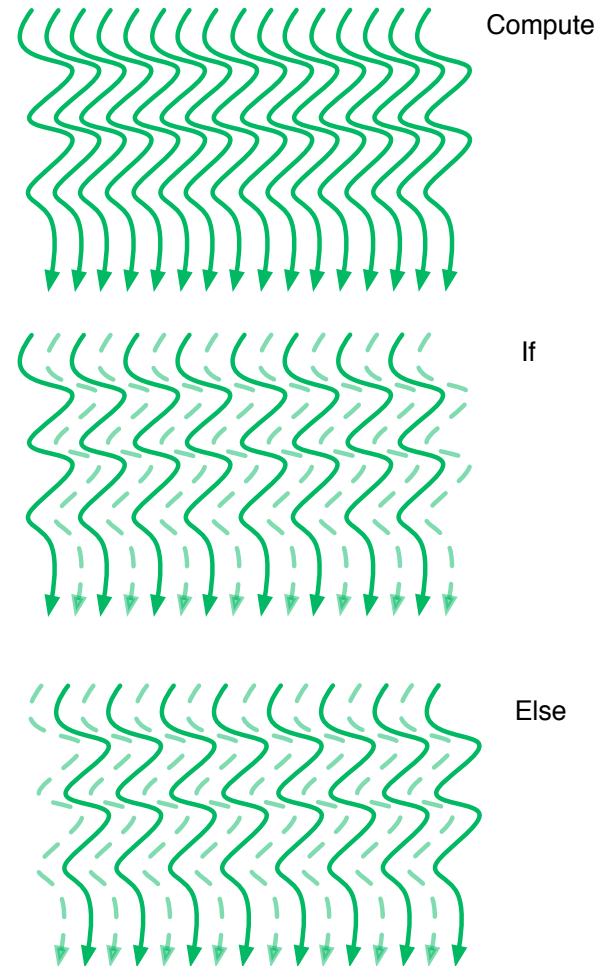
- Bank conflicts are only possible within a warp
 - No bank conflicts between different warps
- If strided accesses are needed, some optimization techniques can help
 - Padding
 - Randomized mapping
 - Rau, “Pseudo-randomly interleaved memory,” ISCA 1991
 - Hash functions
 - V.d.Braak+, “Configurable XOR Hash Functions for Banked Scratchpad Memories in GPUs,” IEEE TC, 2016

SIMD Utilization

SIMD Utilization

■ Intra-warp divergence

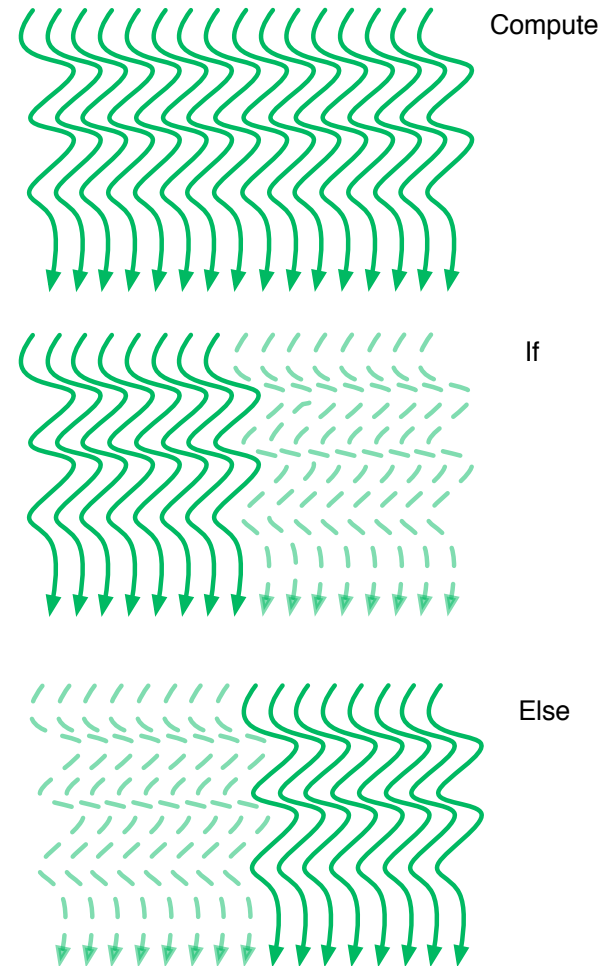
```
Compute(threadIdx.x);  
if (threadIdx.x % 2 == 0){  
    Do_this(threadIdx.x);  
}  
else{  
    Do_that(threadIdx.x);  
}
```



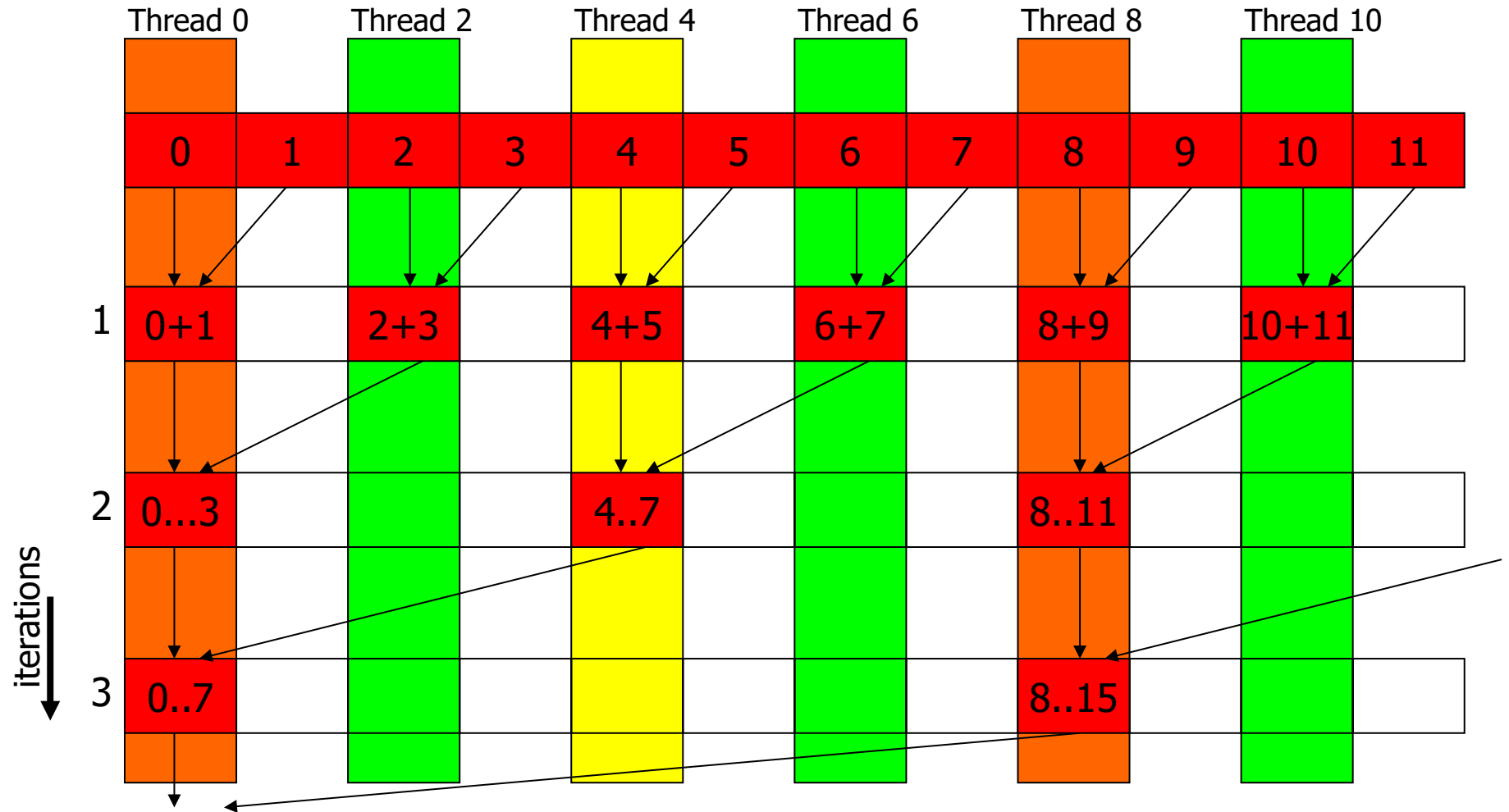
Increasing SIMD Utilization

■ Divergence-free execution

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



Vector Reduction: Naïve Mapping (I)



Slide credit: Hwu & Kirk

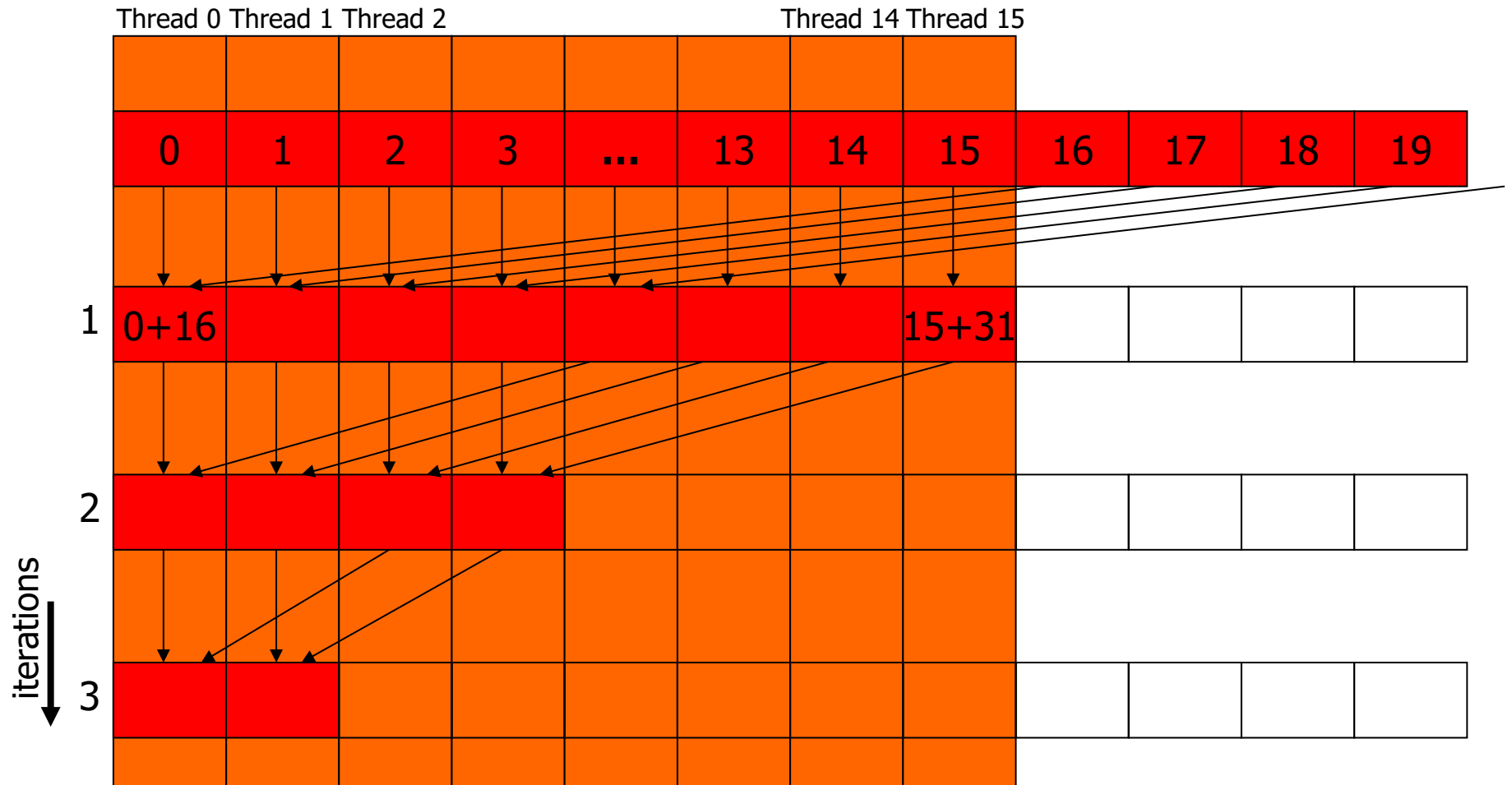
Vector Reduction: Naïve Mapping (II)

- Program with **low SIMD utilization**

```
__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];  
  
}
```

Divergence-Free Mapping (I)

- All active threads belong to the same warp



Slide credit: Hwu & Kirk

Divergence-Free Mapping (II)

- Program with high SIMD utilization

```
__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];  
  
}
```

Atomic Operations

Shared Memory Atomic Operations

- Atomic Operations are needed when threads might **update the same memory locations at the same time**
- 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;
```

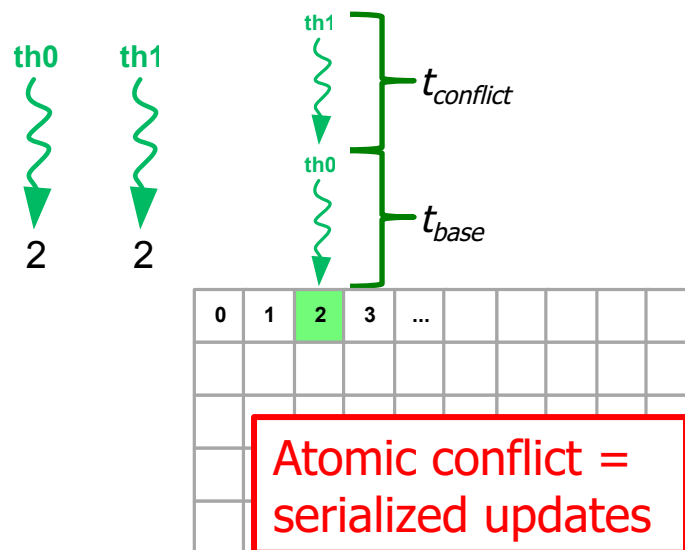
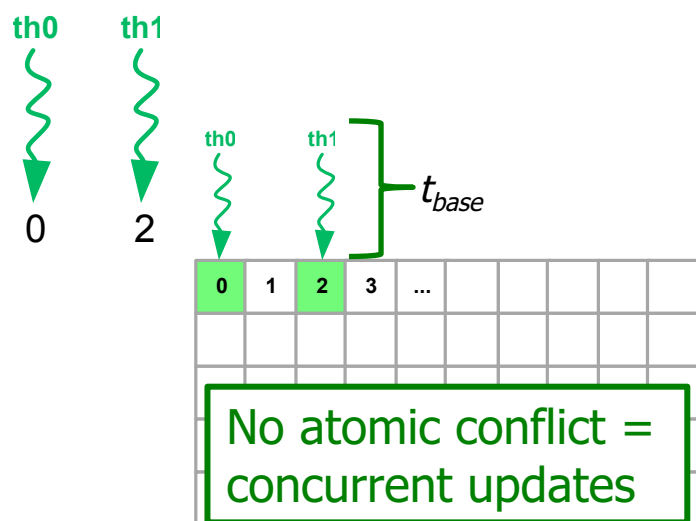
Maxwell, Pascal, Volta

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

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

Atomic Conflicts

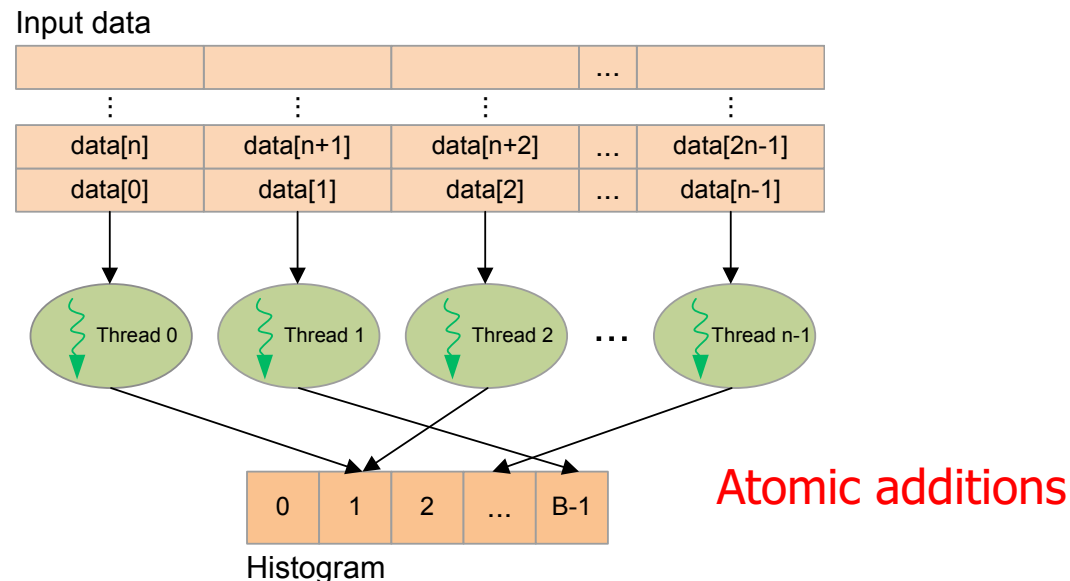
- We define the intra-warp **conflict degree** as the number of threads in a warp that update the same memory position
- The conflict degree can be between 1 and 32



Histogram Calculation

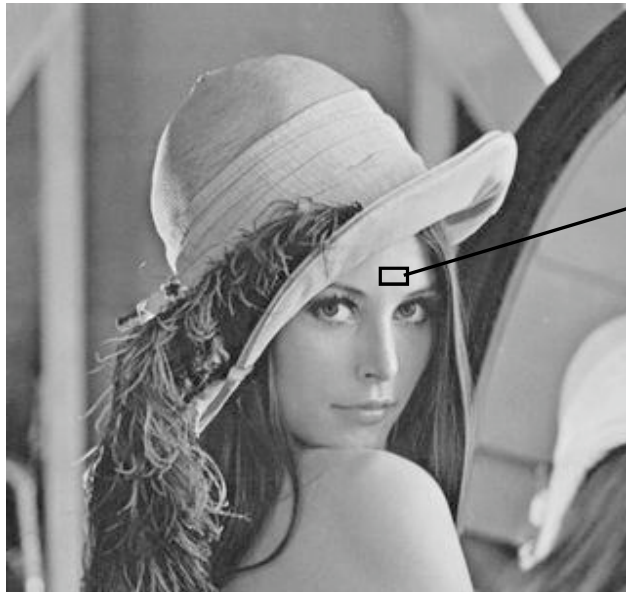
- Histograms count the number of data instances in disjoint categories (bins)

```
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 of Natural Images

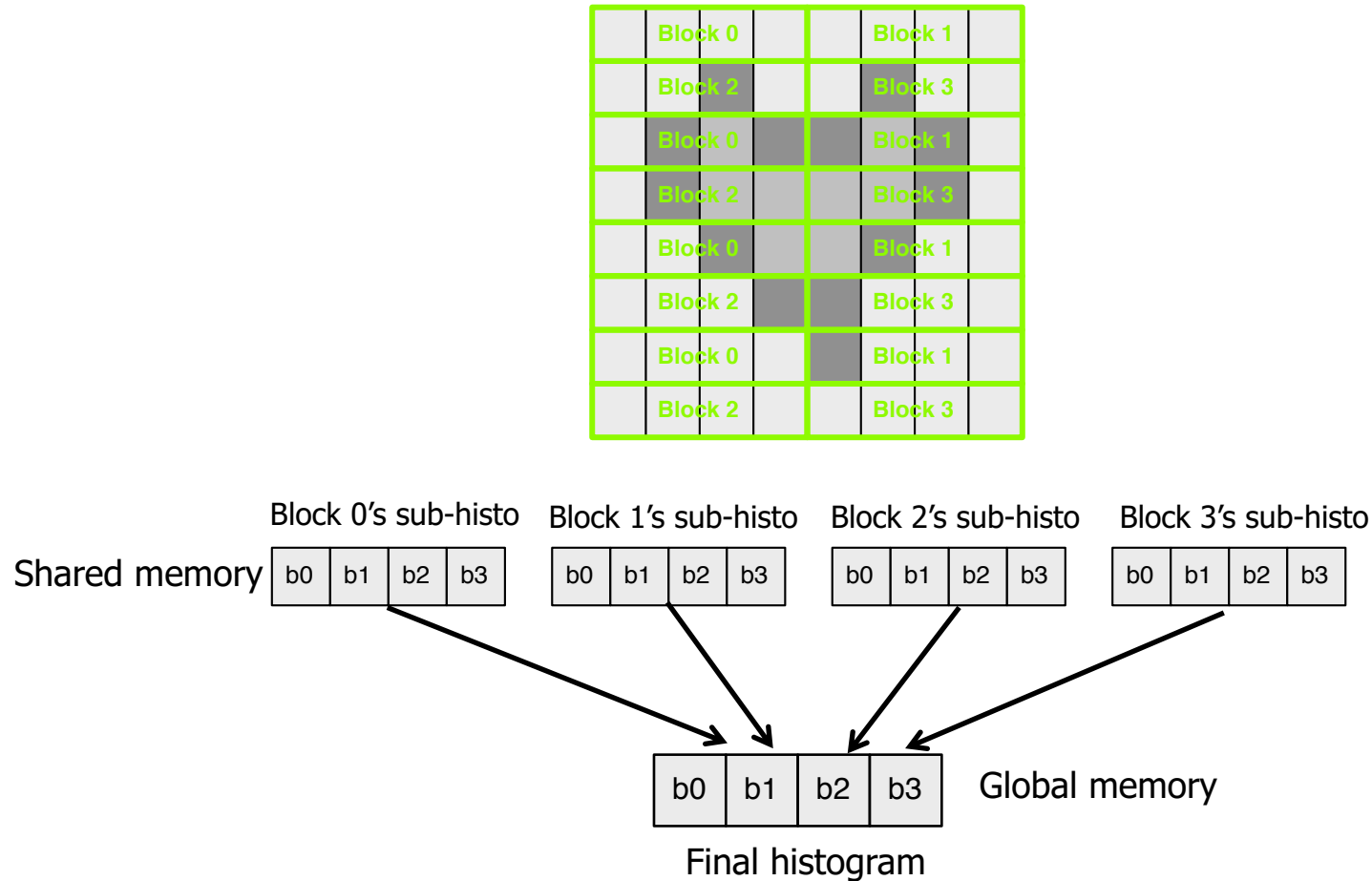
- Frequent conflicts in natural images



	↯	↯	↯	↯	↯	↯	↯	✗	↯	✗
169	170	171	174	177	182	187	192	194	192	
169	173	173	175	177	181	185	189	191	192	
169	173	173	175	177	180	184	188	190	193	
169	172	173	174	176	180	183	187	189	193	
171	173	173	174	176	179	182	185	187	192	
174	175	175	175	176	178	180	183	184	188	
177	177	176	176	177	179	180	181	185	188	
178	178	176	178	184	185	189	193	195	194	
176	176	173	176	181	183	186	190	192	191	
174	172	170	173	177	181	185	189	191	190	
173	171	169	172	175	181	185	190	192	192	
171	169	169	172	174	179	183	189	192	192	

Optimizing Histogram Calculation

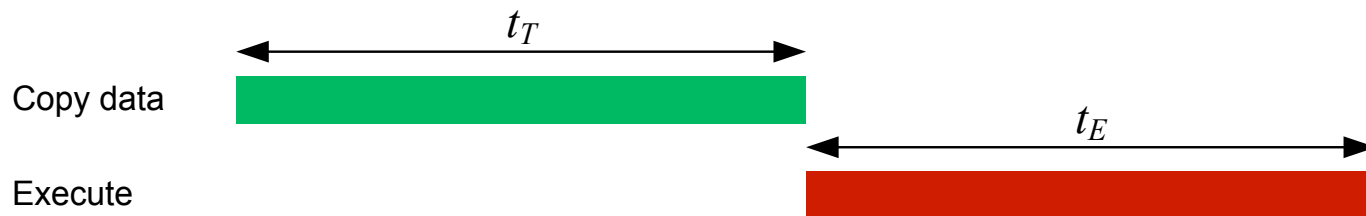
- **Privatization:** Per-block sub-histograms in shared memory



Data Transfers between CPU and GPU

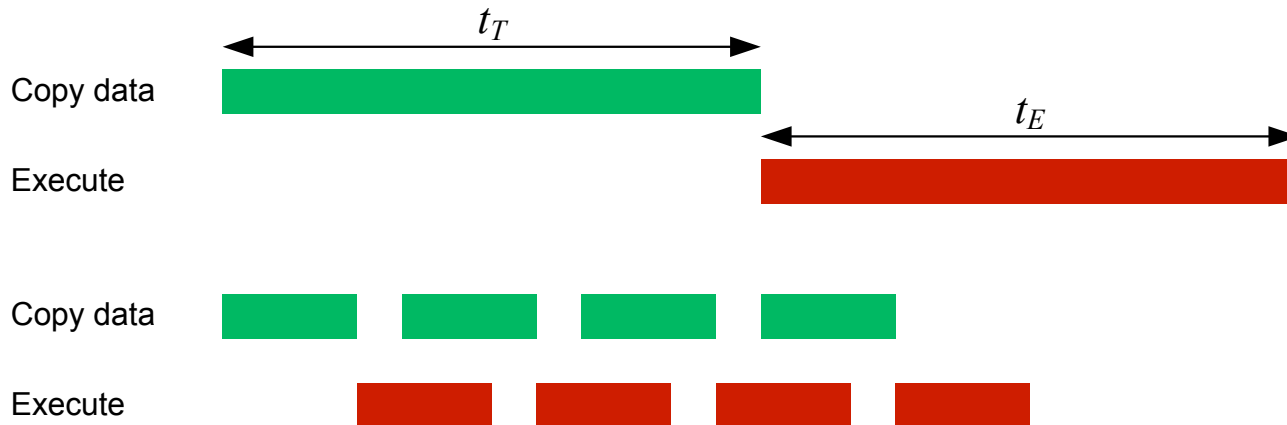
Data Transfers

- 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

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

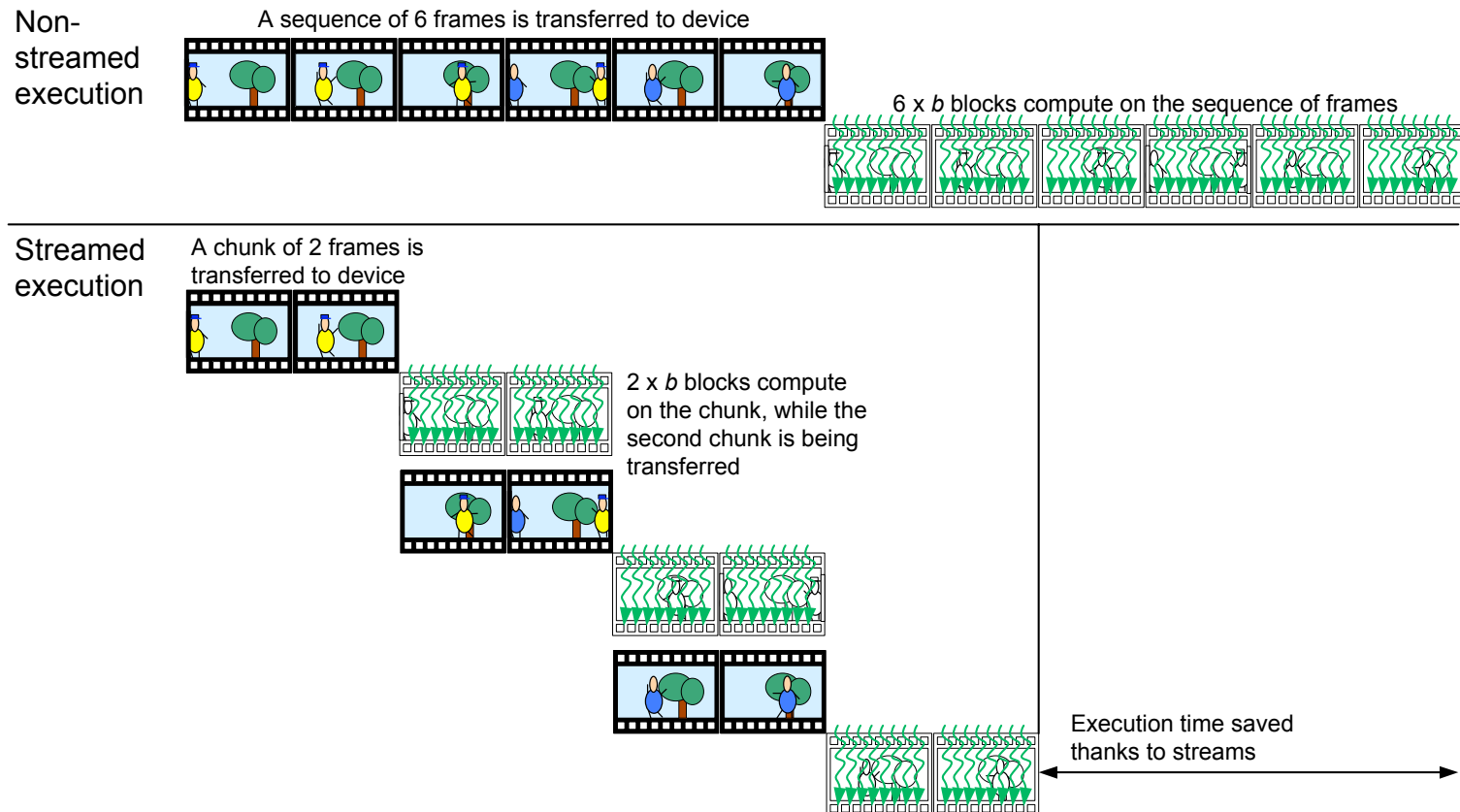
$t_E \geq t_T$ (dominant kernel)

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

$t_T > t_E$ (dominant transfers)

Overlap of Communication and Computation

- Applications with independent computation on different data instances can benefit from asynchronous transfers
- For instance, **video processing**

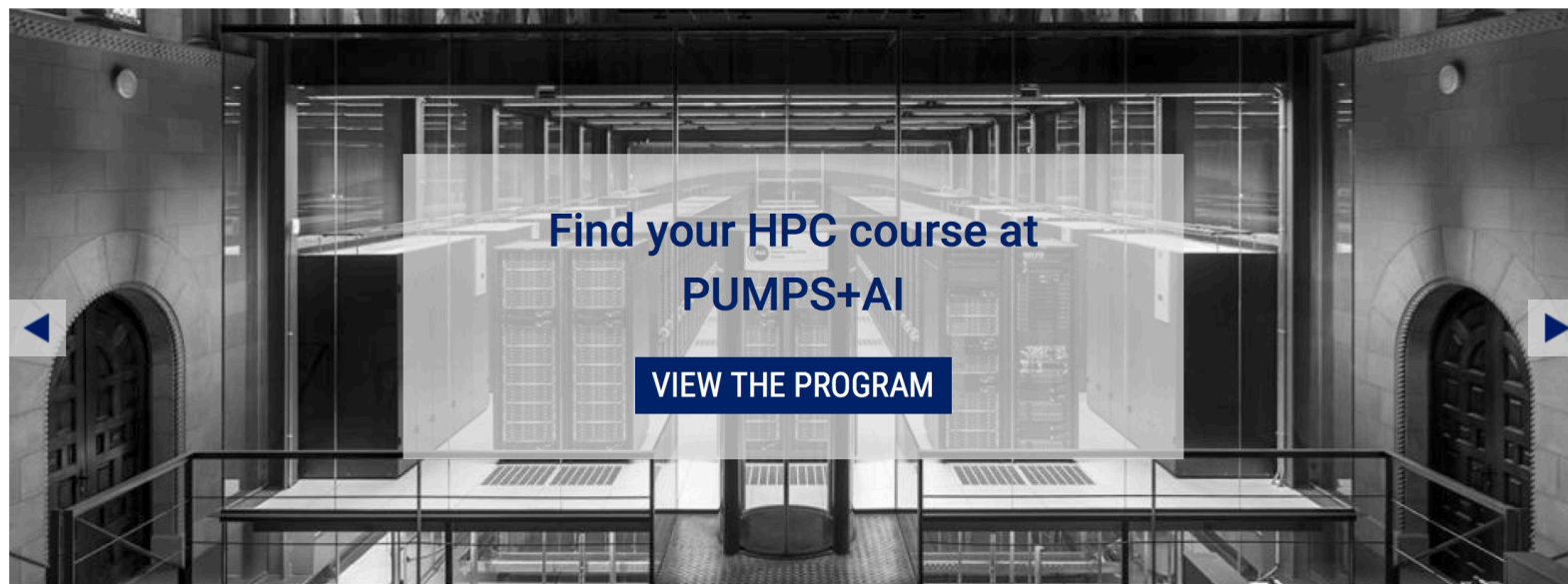


Summary

- GPU as an accelerator
 - 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

PUMPS+AI Summer School, Barcelona, July 16-20

- <https://pumps.bsc.es/2018/>



PUMPS+AI Summer School, 2018, July 16-20

The Barcelona Supercomputing Center (BSC) in association with Universitat Politècnica de Catalunya (UPC) has been awarded by NVIDIA as a GPU Center of Excellence. BSC and UPC currently offer a number of courses covering CUDA architecture and programming languages for parallel computing. Please contact us for possible collaborations.

The ninth edition of the Programming and Tuning Massively Parallel Systems + Artificial Intelligence summer school (PUMPS+AI) is aimed at enriching the skills of researchers, graduate students and teachers with cutting-edge technique and hands-on experience in developing applications for many-core processors with massively parallel computing resources like GPU accelerators.

- *Summer School Co-Directors:* **Mateo Valero** (BSC and UPC) and **Wen-mei Hwu** (University of Illinois at Urbana-Champaign)
- *Local Organizers:* **Antonio J. Peña** (responsible, BSC and UPC), and **Pau Farre** (BSC)
- *Dates:*

● Applications due: **May 31, 2018**

Design of Digital Circuits

Lecture 22: GPU Programming

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zurich

Spring 2018

18 May 2018