Computer Architecture
Lecture 26: GPU Programming

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

06 January 2023

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
- Collaborative computing

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

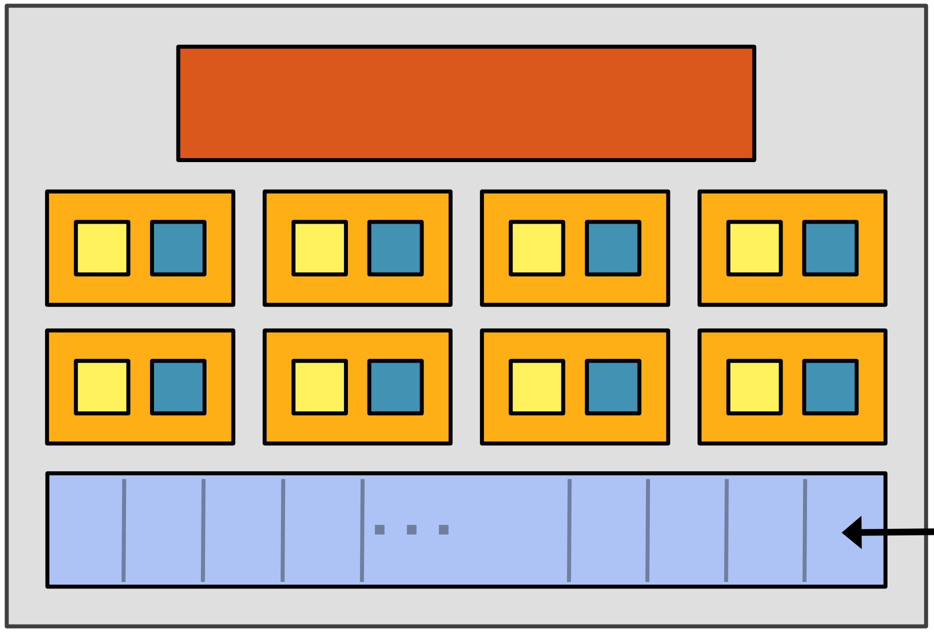
Recall: NVIDIA GeForce GTX 285

- 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

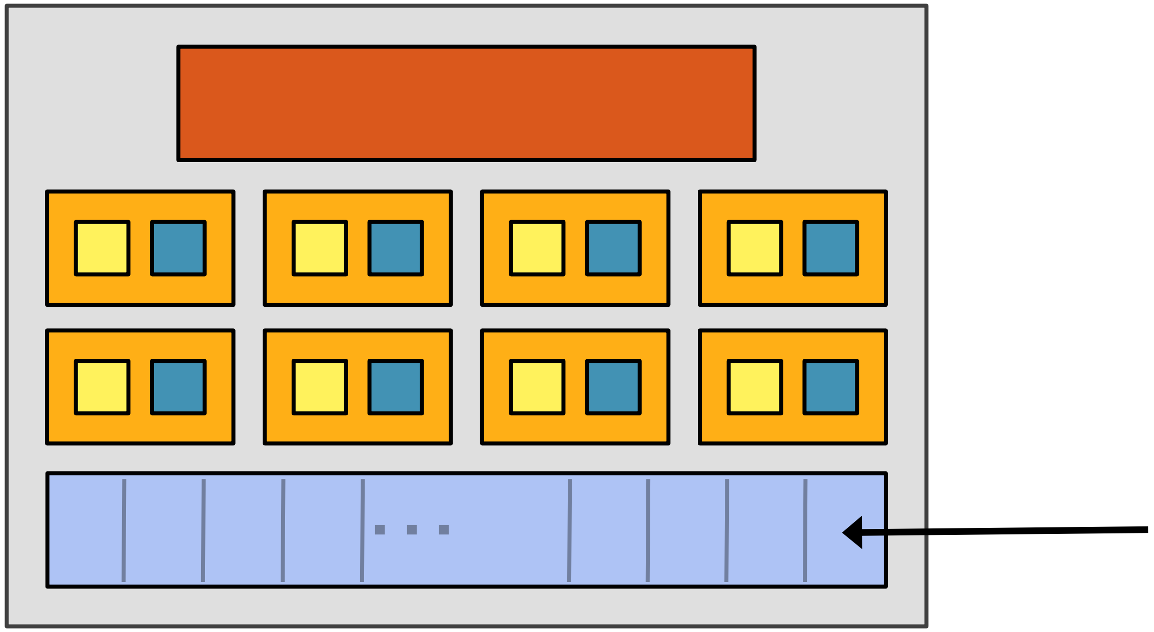
🟦 = multiply

🟥 = instruction stream decode

🟪 = execution context storage

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285 "core"

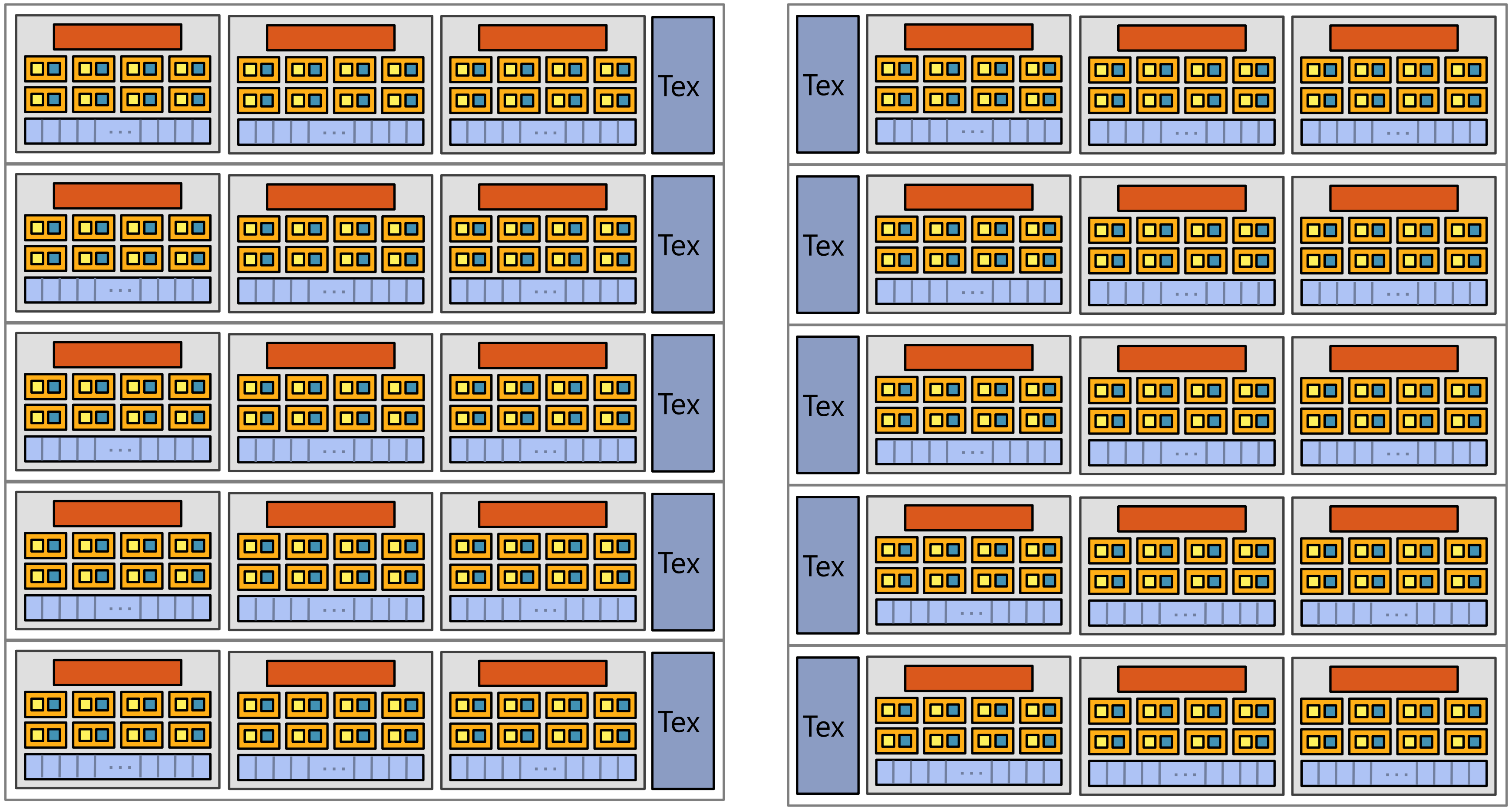


64 KB of storage for thread contexts (registers)

- 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

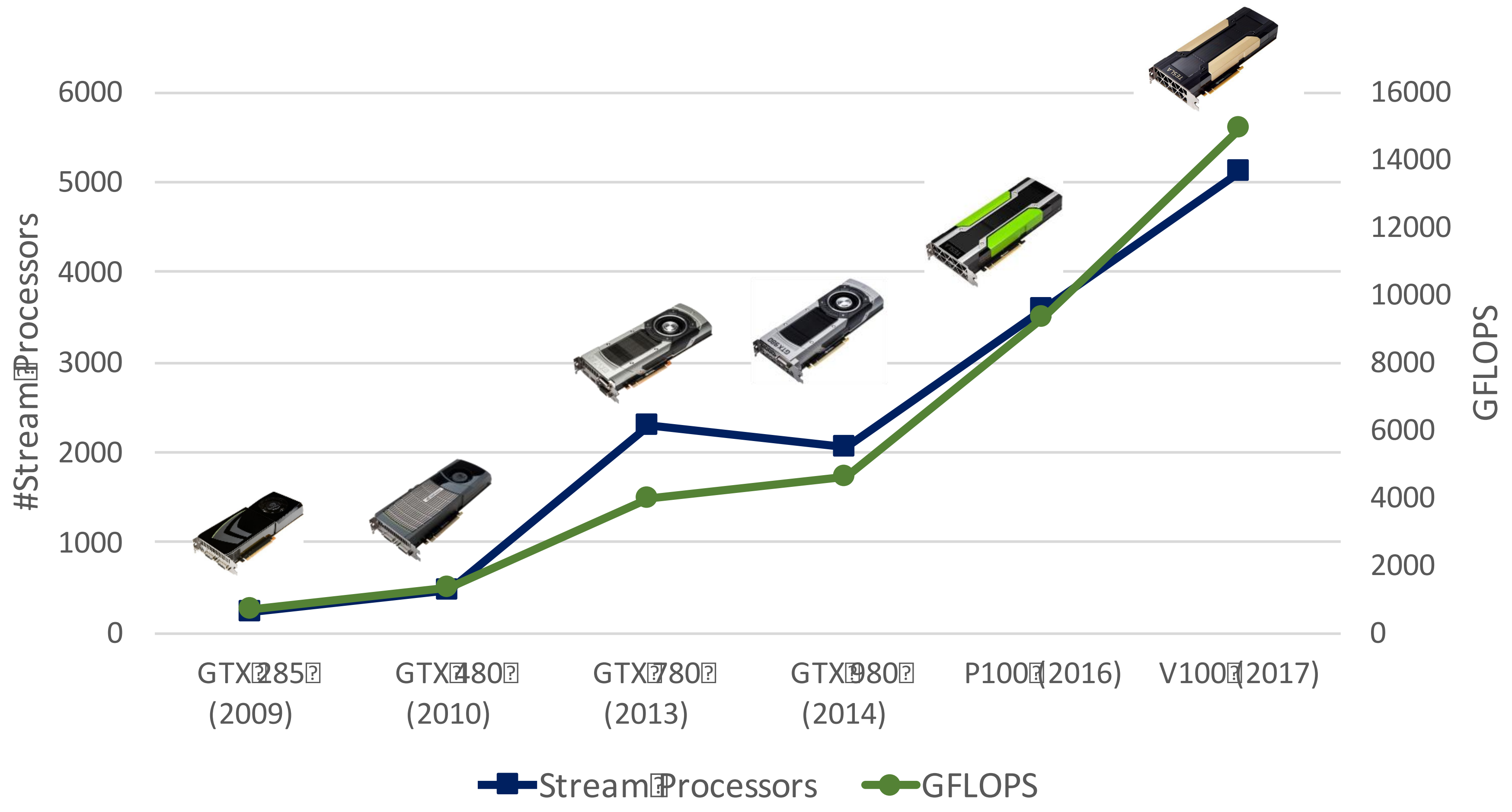
NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian

Recall: Evolution of NVIDIA GPUs

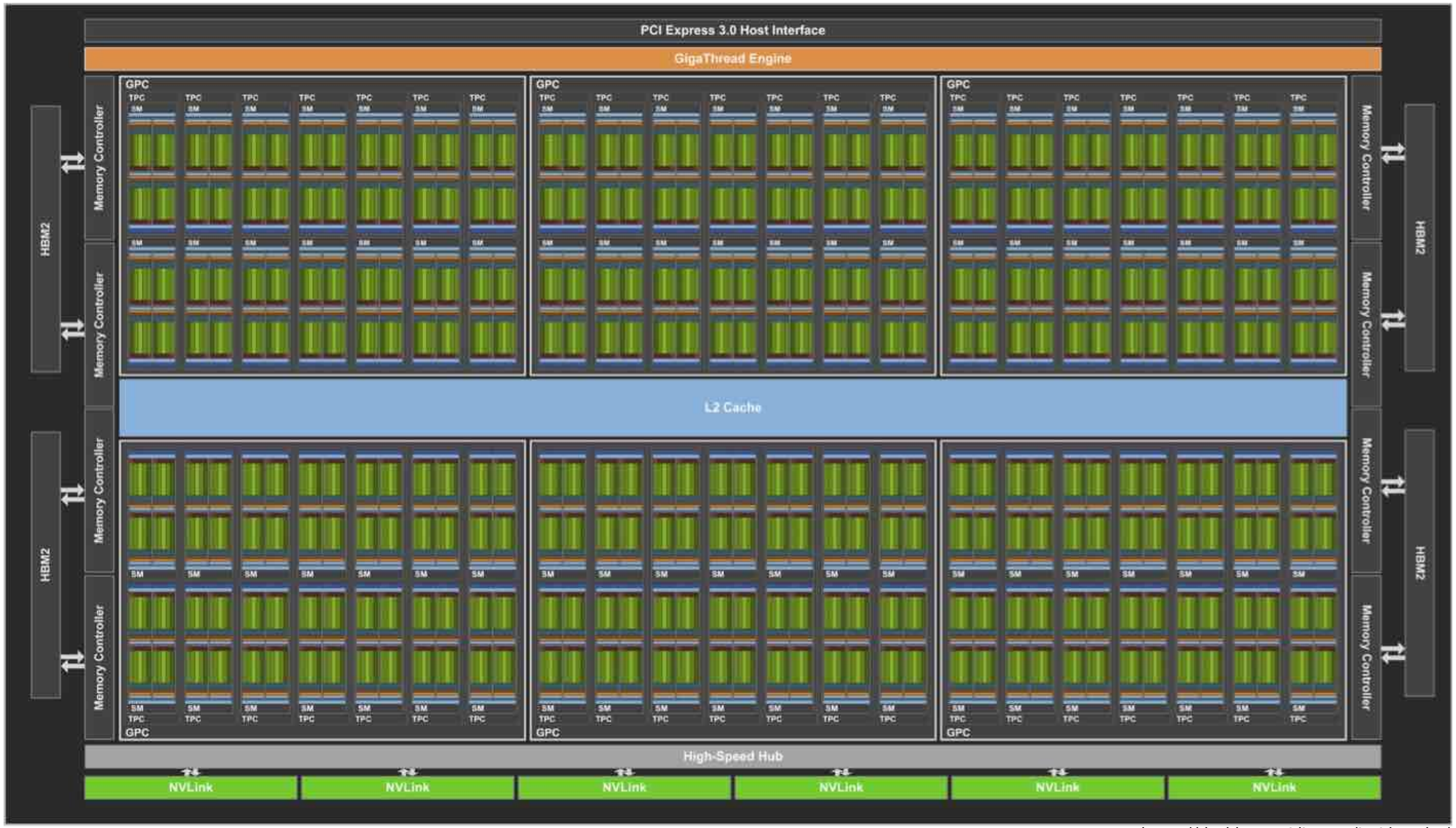


Recall: NVIDIA V100

- 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)

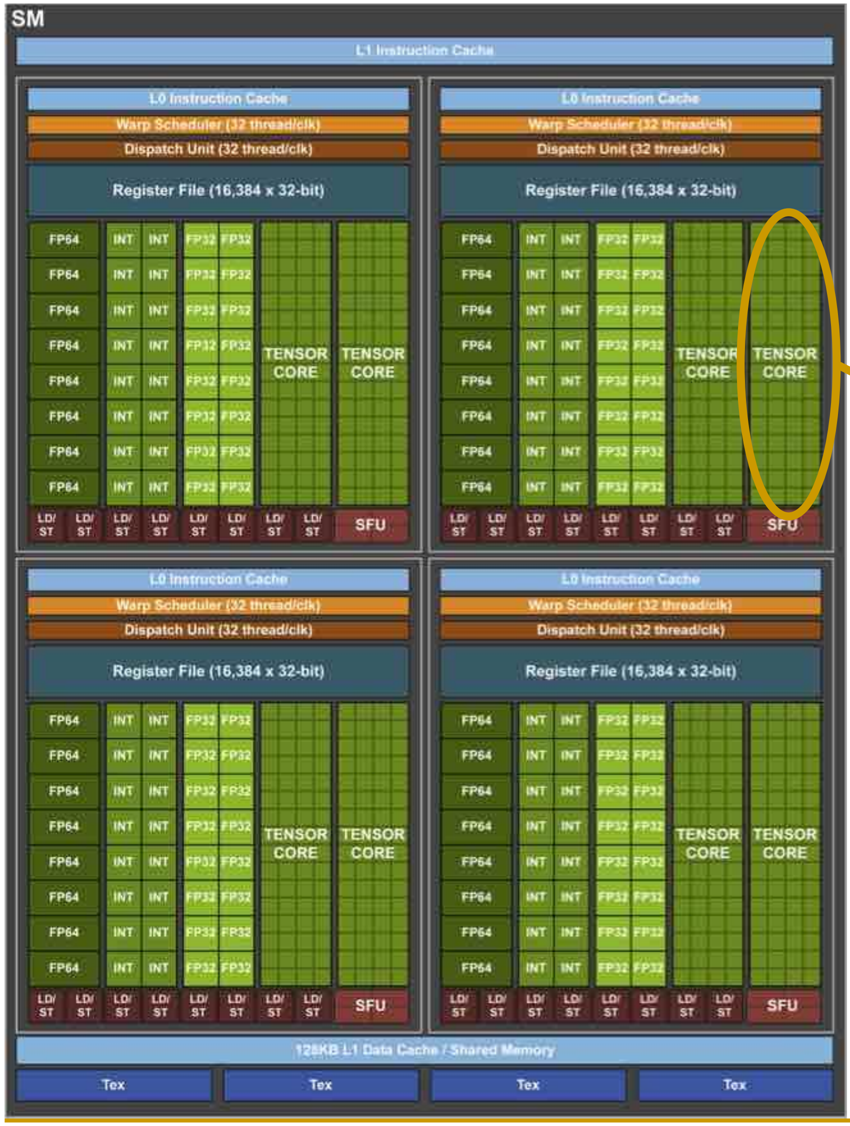
Recall: NVIDIA V100 Block Diagram



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

80 cores on the V100

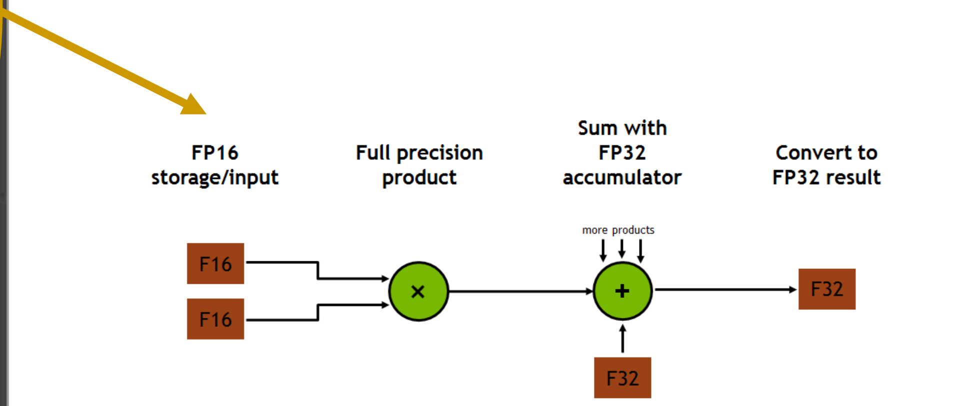
Recall: NVIDIA V100 Core


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

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/

Food for Thought

- What is the main bottleneck in GPU programs?

- "Tensor cores":
  - Can you think about other operations than matrix multiplication?
  - What other applications could benefit from specialized cores?

- Compare and contrast GPUs vs other accelerators (e.g., systolic arrays)

  - Which one is better for machine learning?

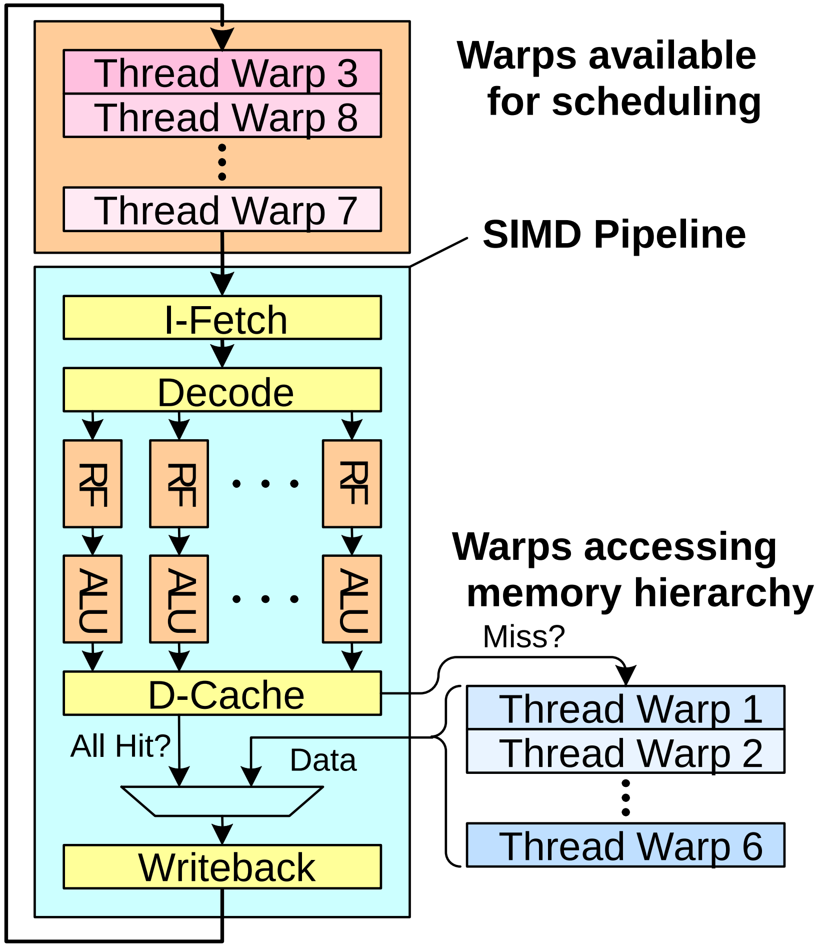
  - Which one is better for image/vision processing?

  - What types of parallelism each one exploits?

  - What are the tradeoffs?

Recall: Latency Hiding via Warp-Level FGMT

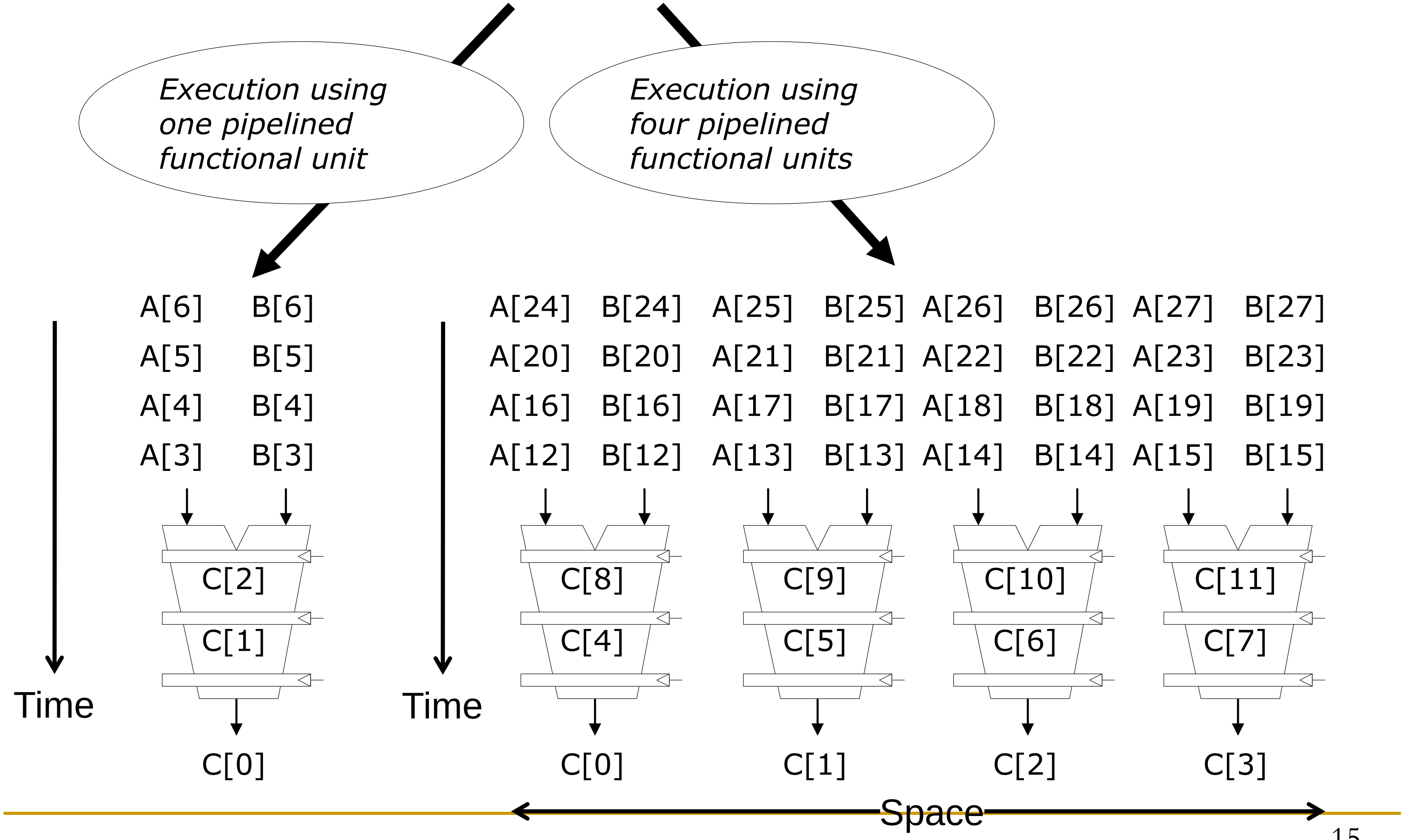
- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables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



Slide credit: Tor Aamodt

Recall: Warp Execution

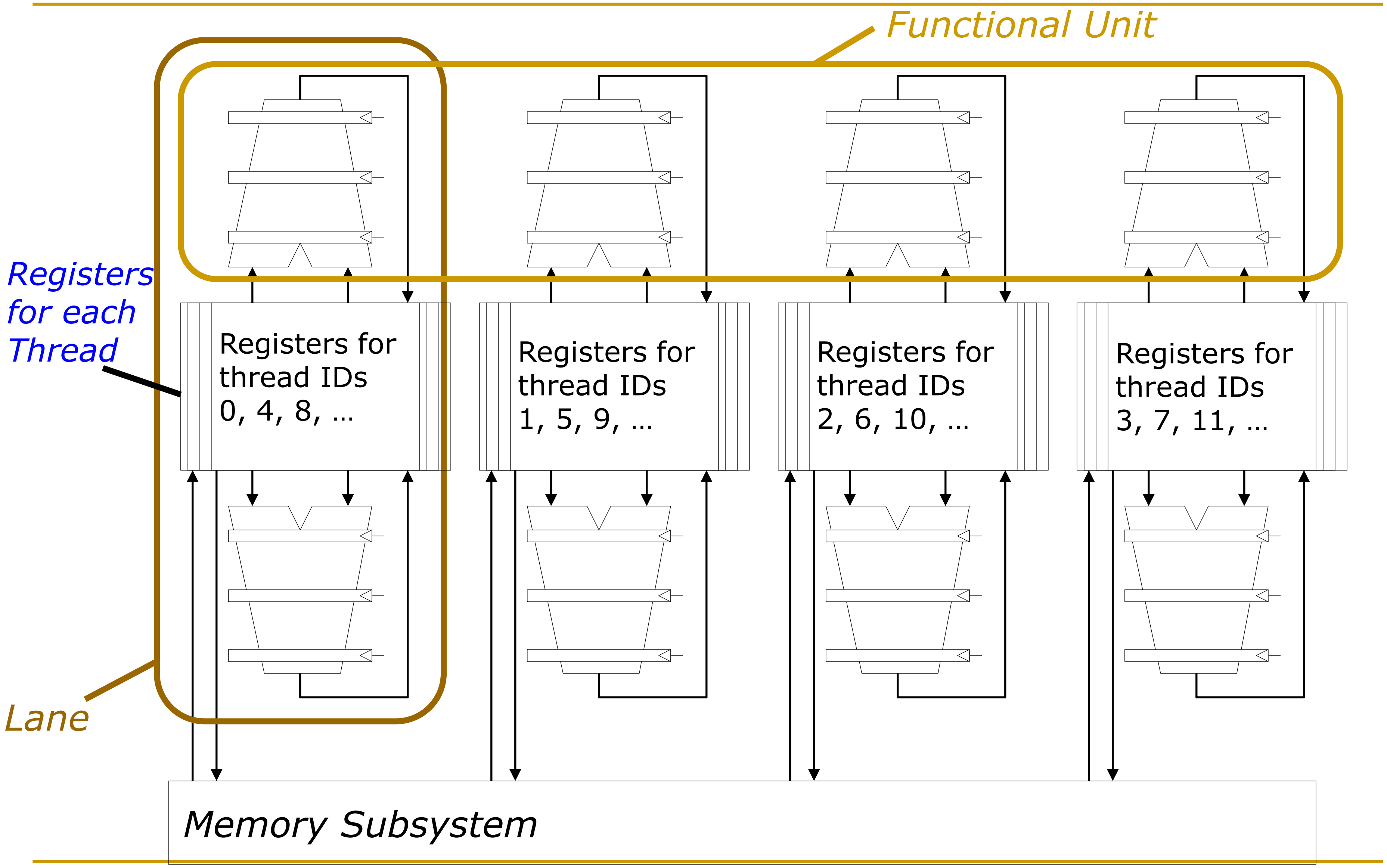
32-thread warp executing ADD A[tid],B[tid] → C[tid]



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

Slide credit: Krste Asanovic

Recall: SIMD Execution Unit Structure



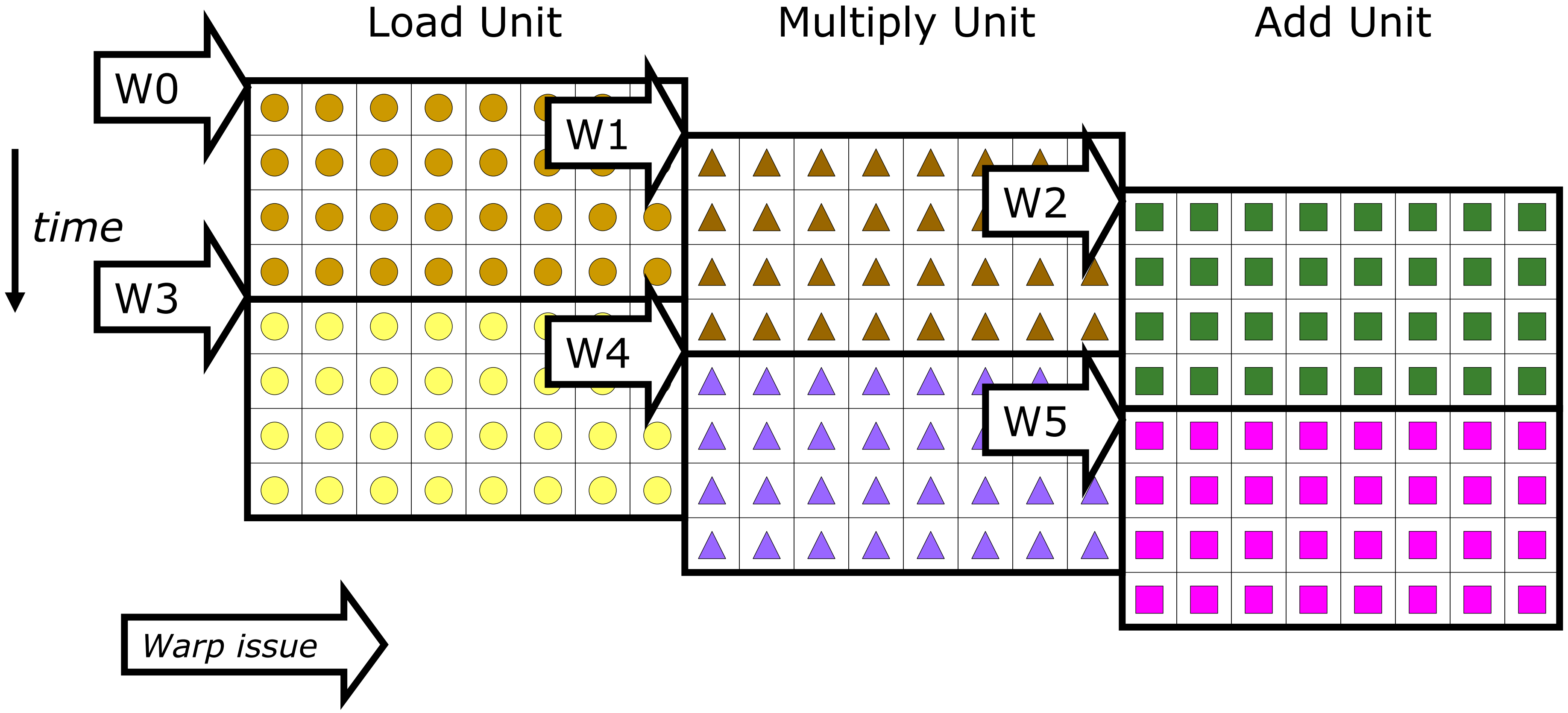
Functional Unit
Registers for each Thread
Registers for thread IDs 0, 4, 8, …
Registers for thread IDs 1, 5, 9, …
Registers for thread IDs 2, 6, 10, …
Registers for thread IDs 3, 7, 11, …
Lane
Memory Subsystem

Slide credit: Krste Asanovic

Recall: 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



Slide credit: Krste Asanovic

Clarification of some GPU Terms

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

GPU Programming

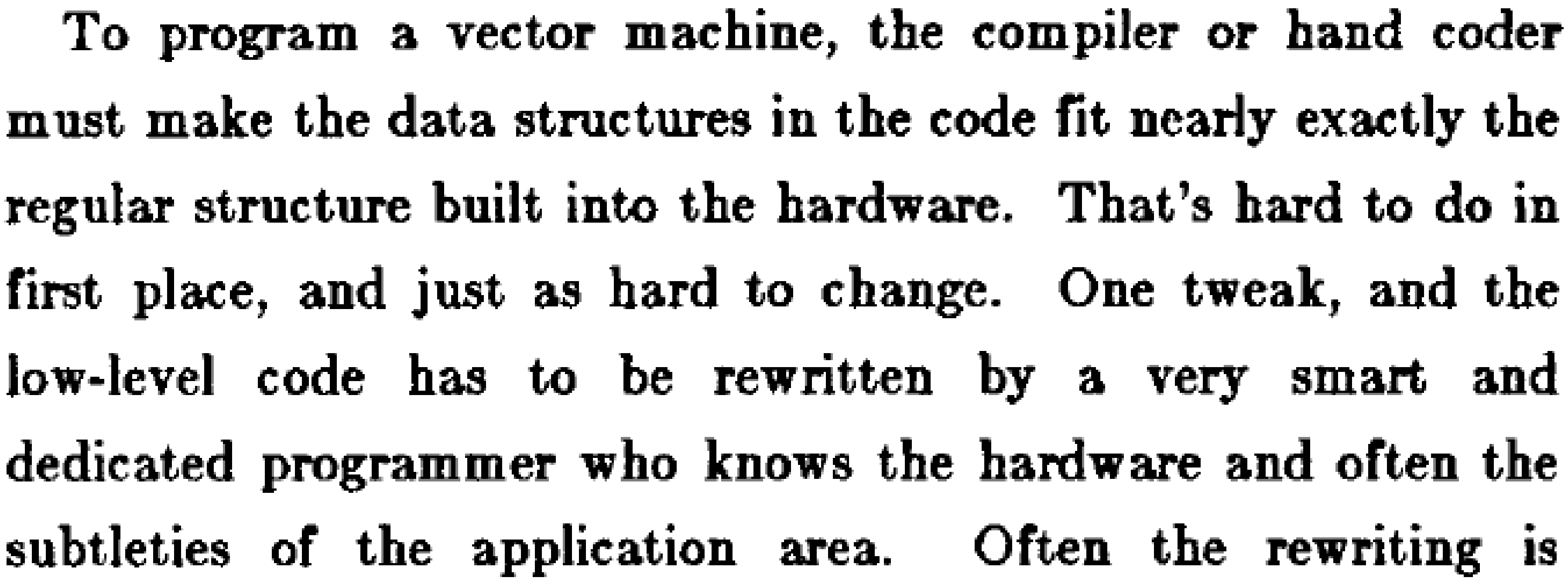
Recall: Vector Processor Disadvantages

-- 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

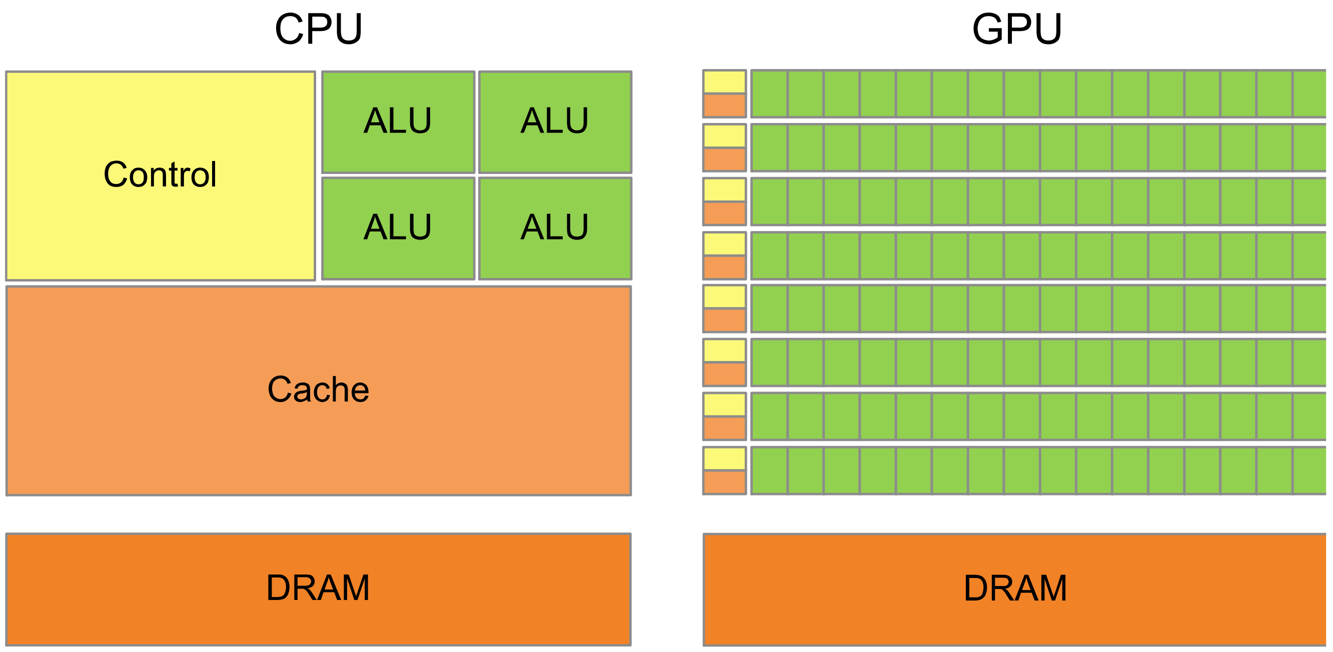
Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

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
  - Deep neural networks
- 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, GDDR6, HBM2)
    - Data layout

CPU vs. GPU

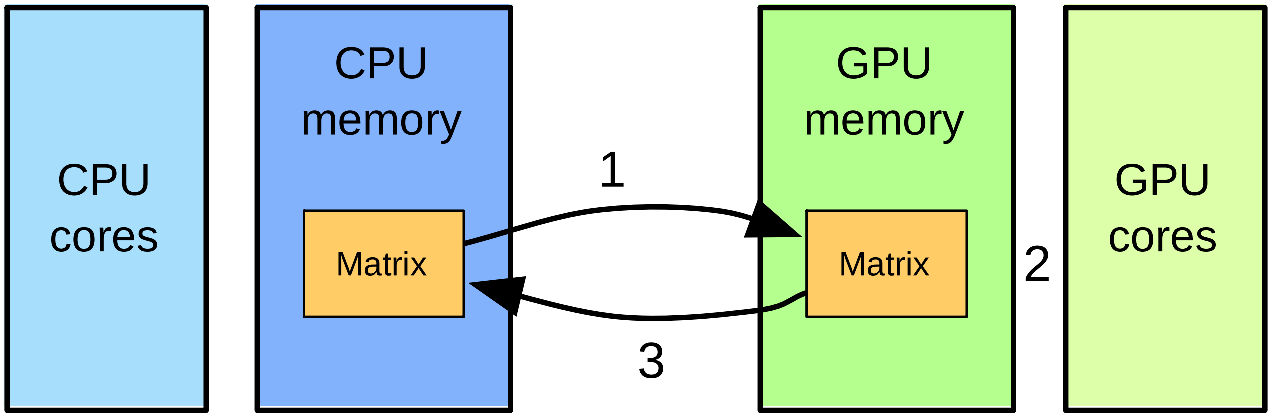
- Different design philosophies
  - CPU: A few out-of-order cores
  - GPU: Many in-order FGMT cores



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

Serial Code (host)

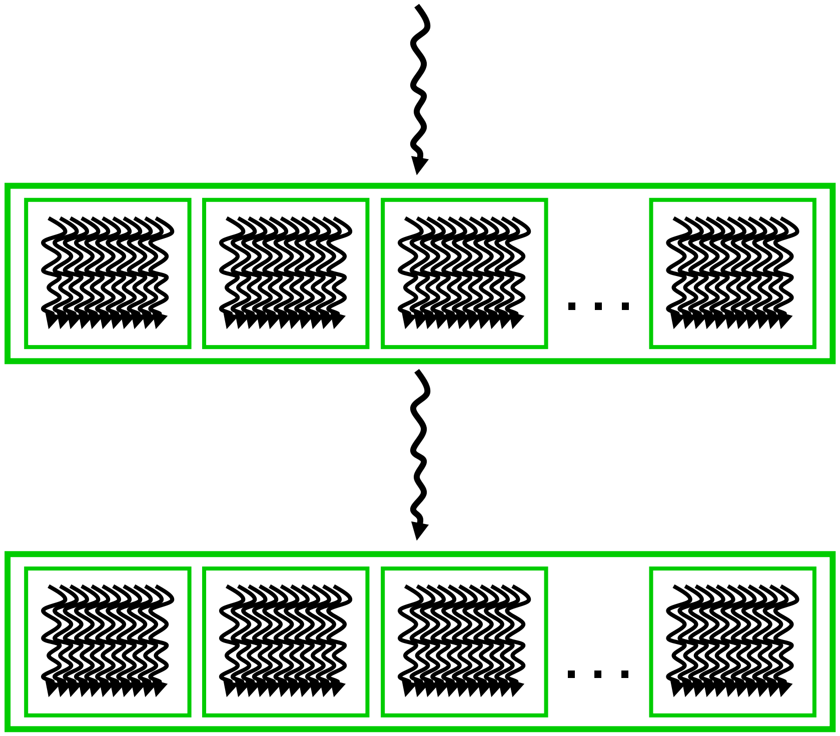
Parallel Kernel (device)

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

Serial Code (host)

Parallel Kernel (device)

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



Slide credit: Hwu & Kirk

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

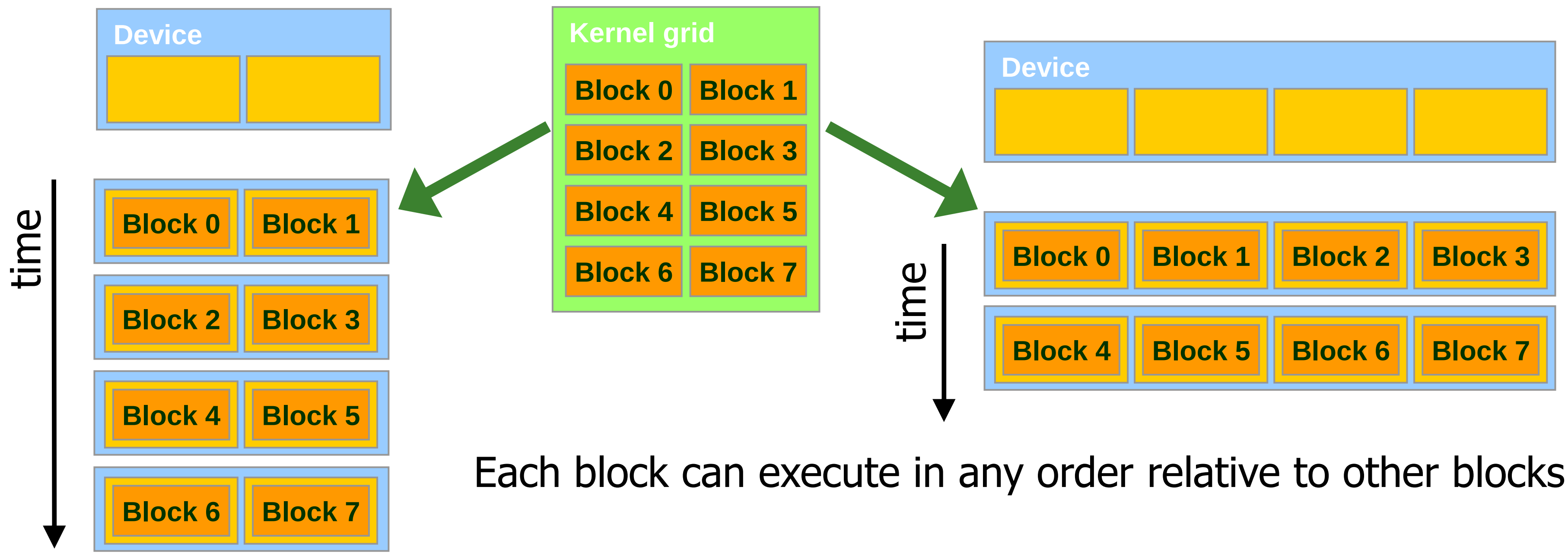
- Essentially, multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed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

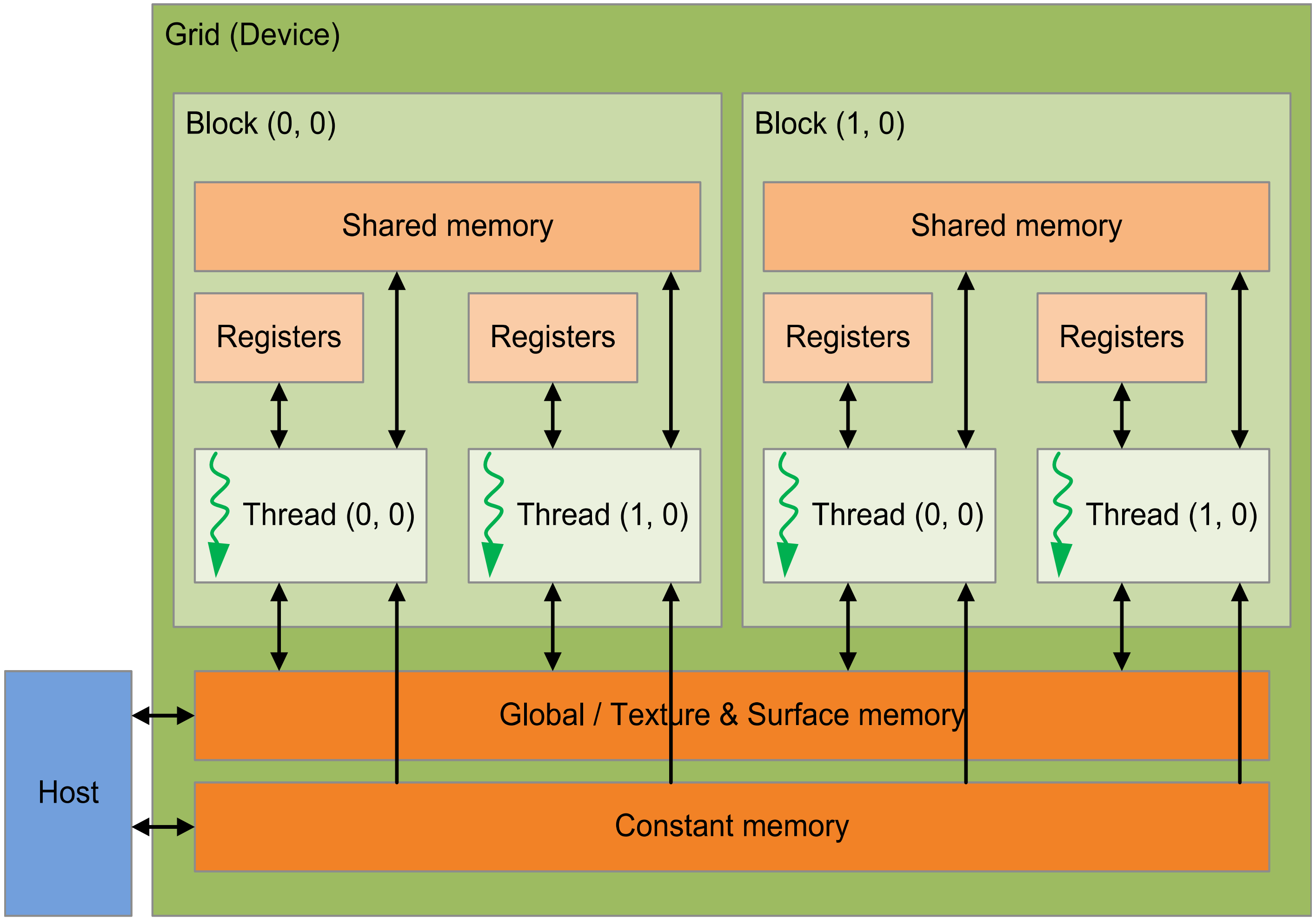
- Hardware is free to schedule thread blocks



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

Slide credit: Hwu & Kirk

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();`

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

- Images are 2D data structures
  - height x width
  - Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width

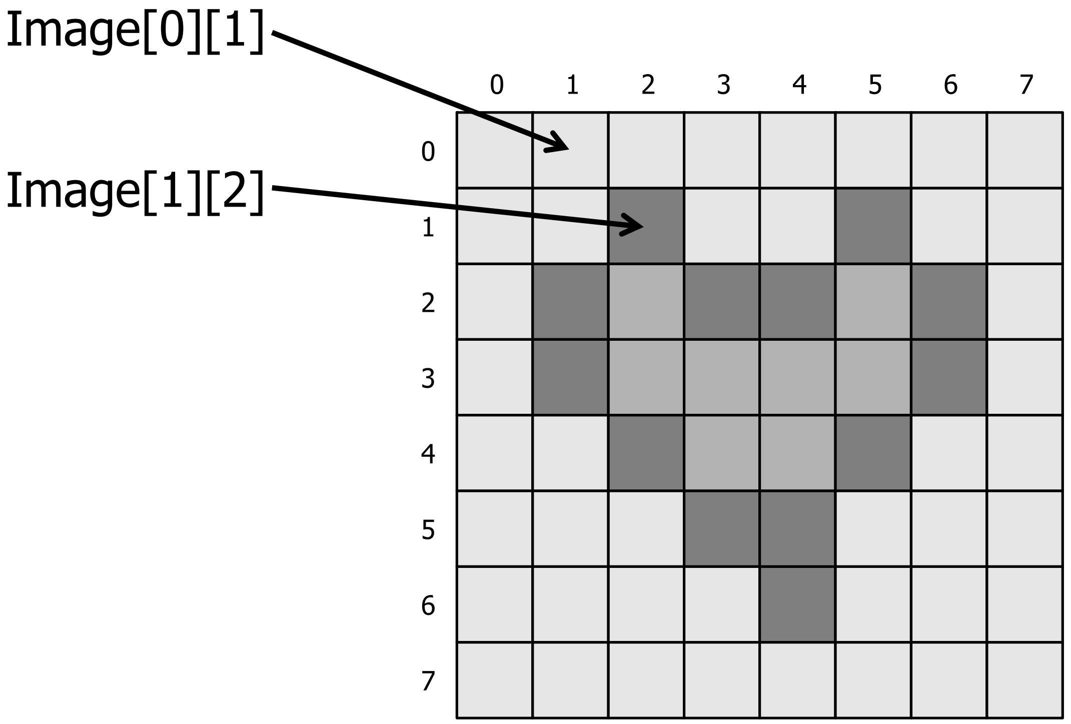
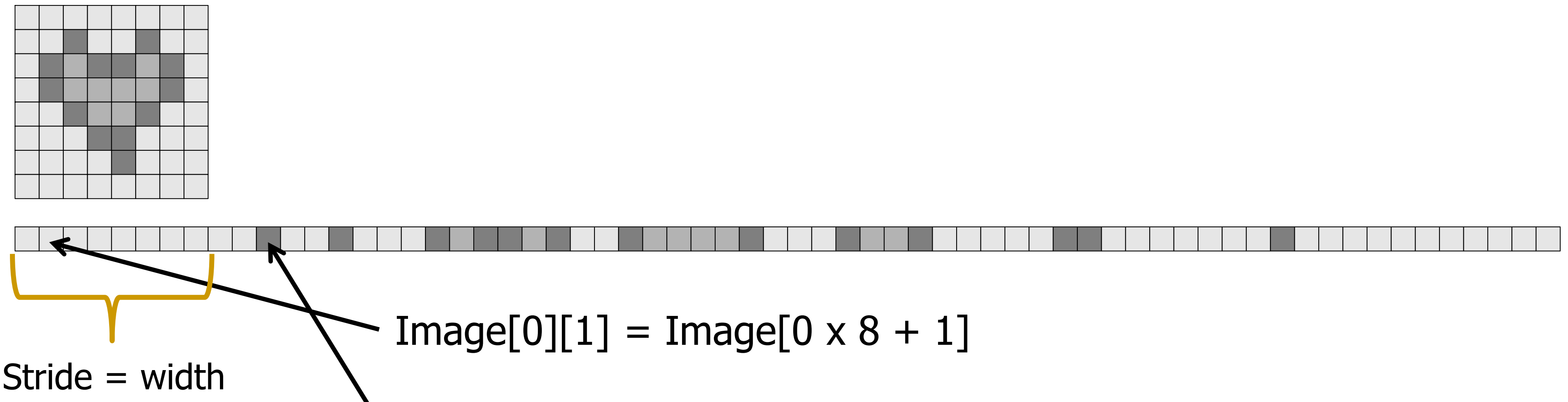


Image Layout in Memory

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

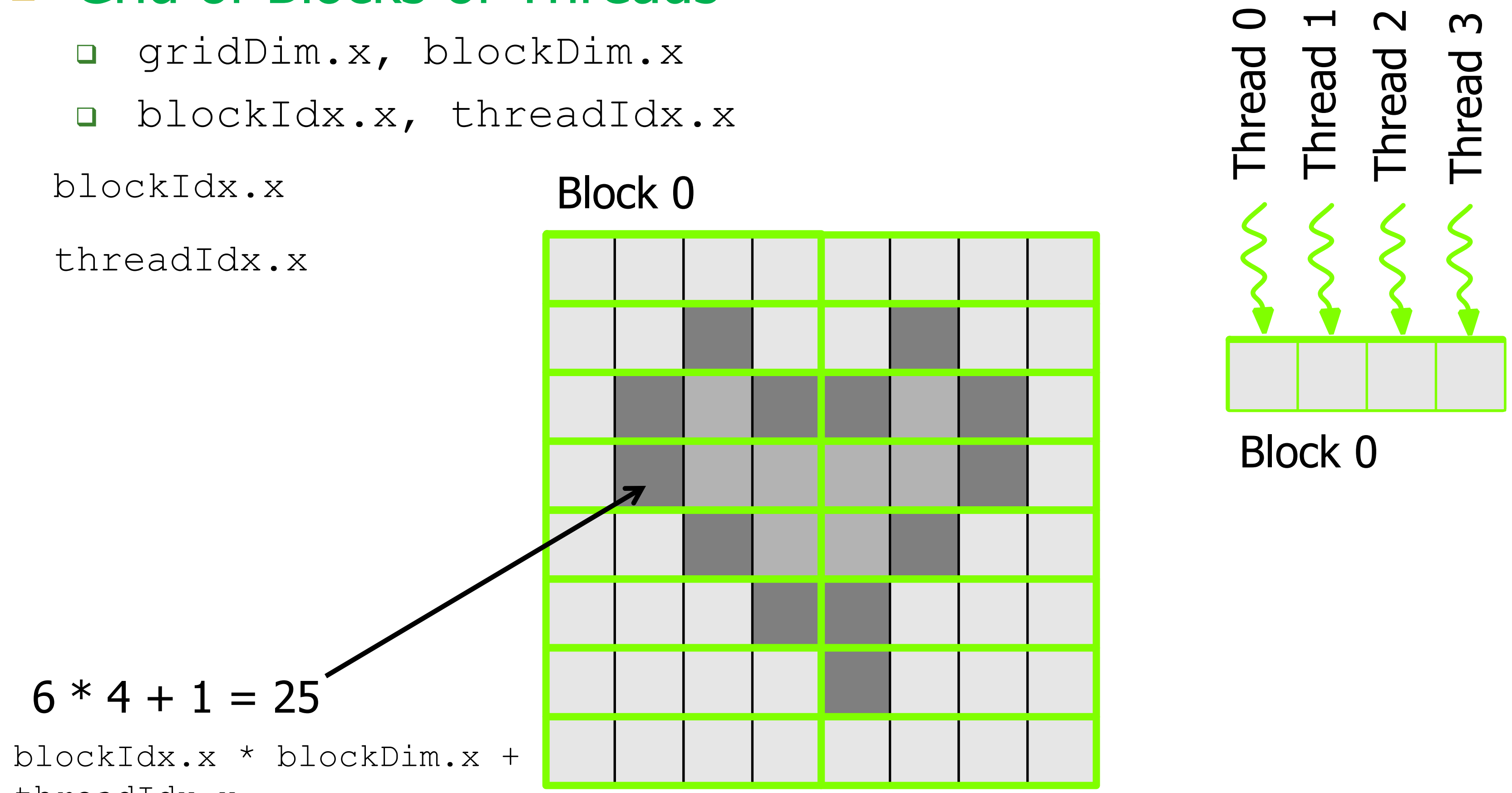

Image[0][1] = Image[0 x 8 + 1]
Image[1][2] = Image[1 x 8 + 2]
Stride = width

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`

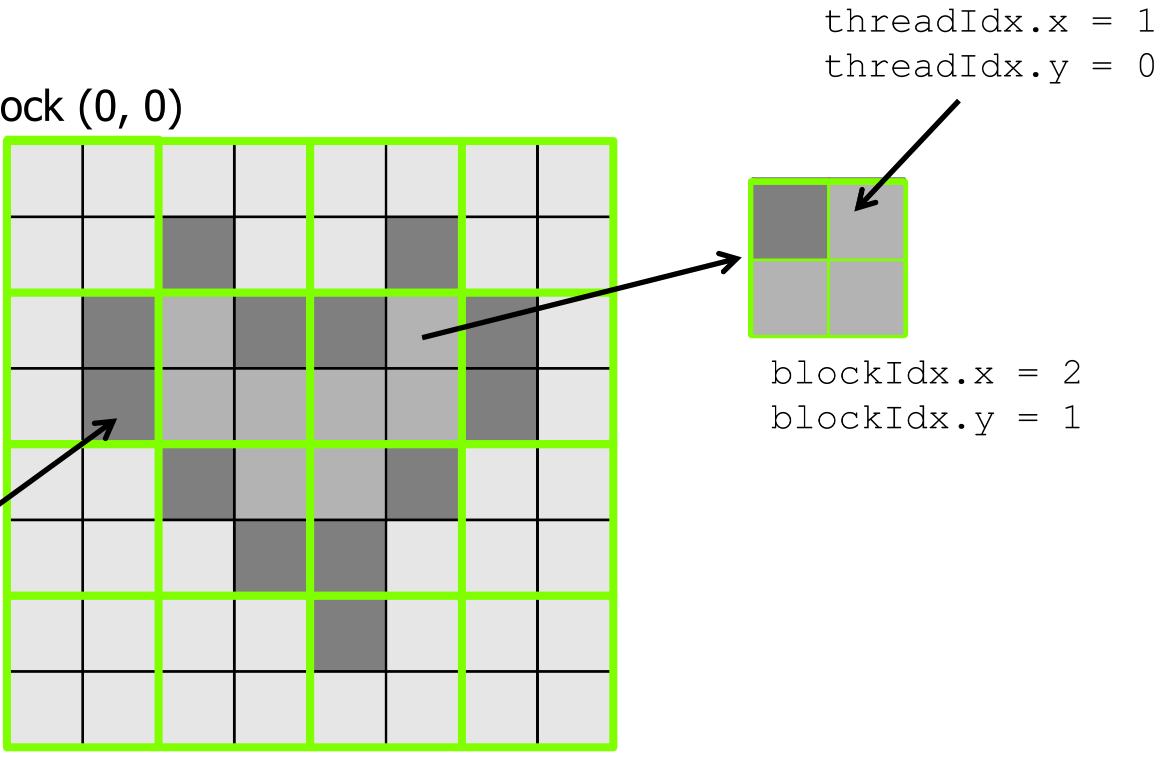
`threadIdx.x`



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

Indexing and Memory Access: 2D Grid

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



```
Row = blockIdx.y *
blockDim.y + threadIdx.y

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

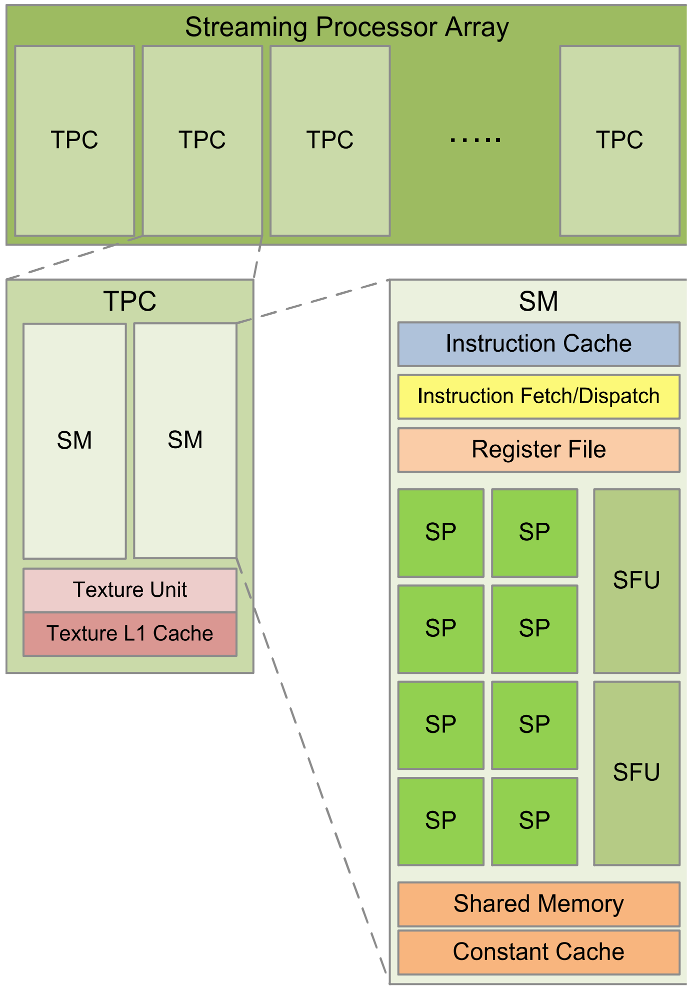
Row = 1 * 2 + 1 = 3

Col = 0 * 2 + 1 = 1

Image[3][1] = Image[3 * 8 + 1]

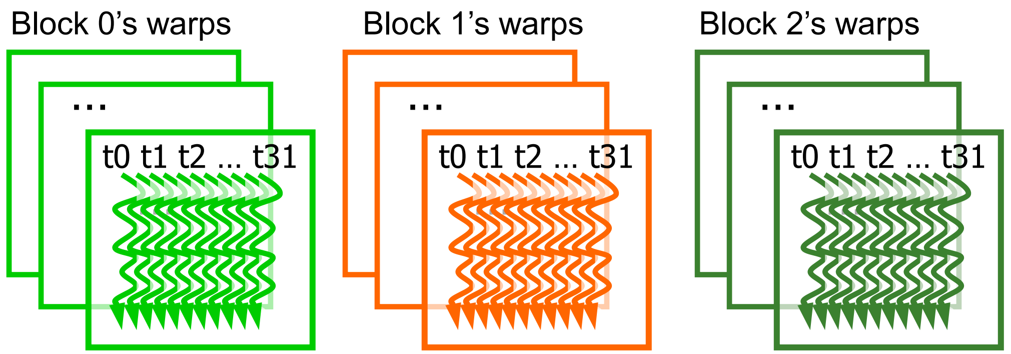
Brief Review of GPU Architecture (I)

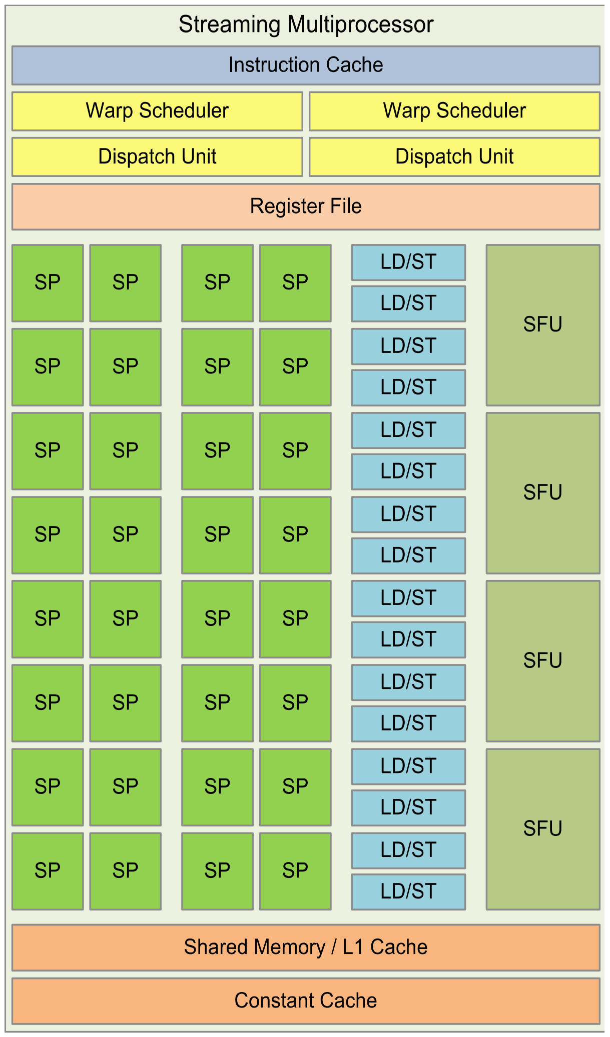
- Streaming Processor Array
  - Tesla architecture (G80/GT200)



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

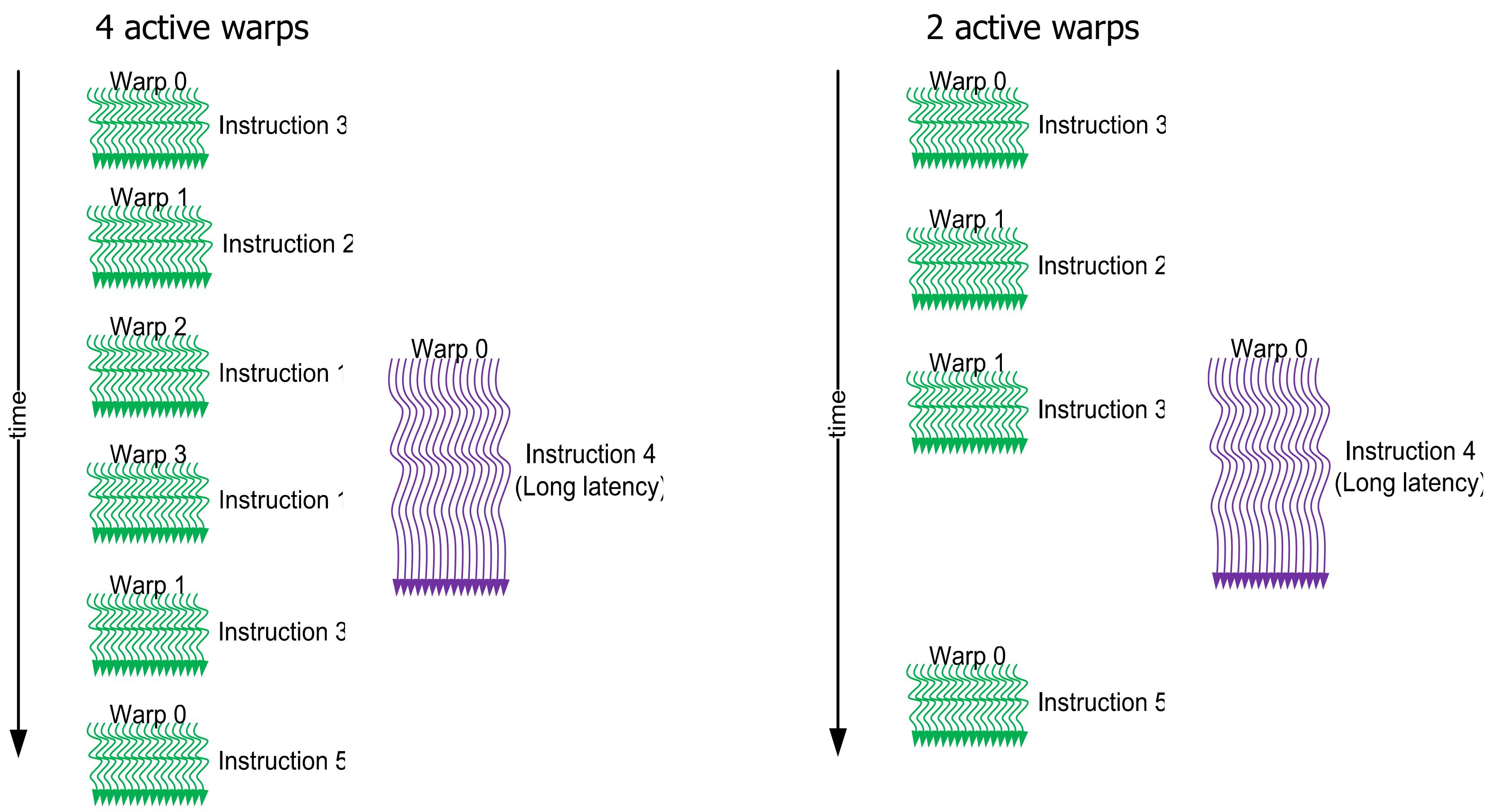
Performance Considerations

- 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



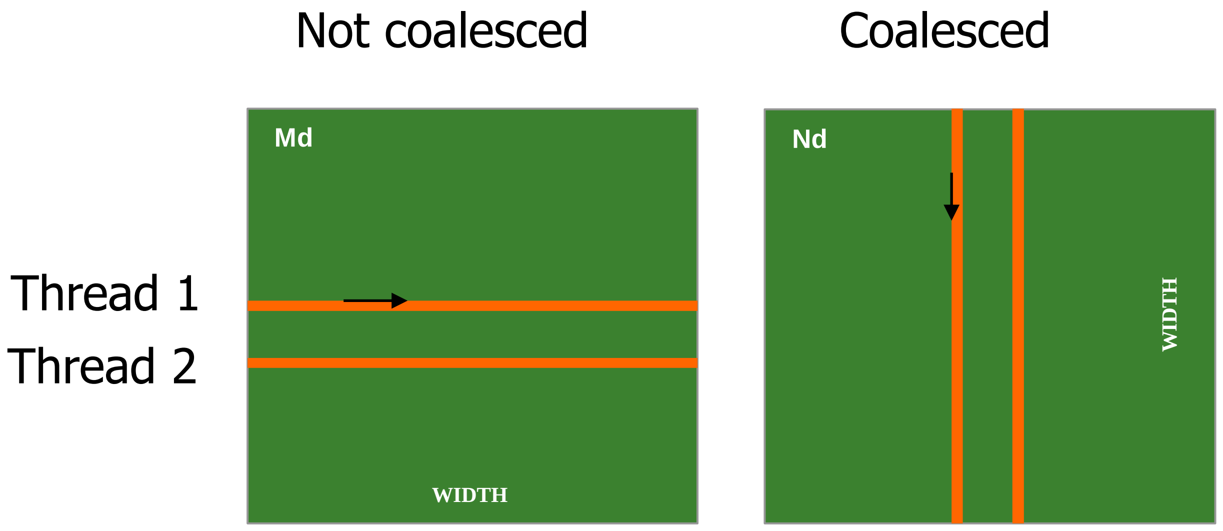
Occupancy

- 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)

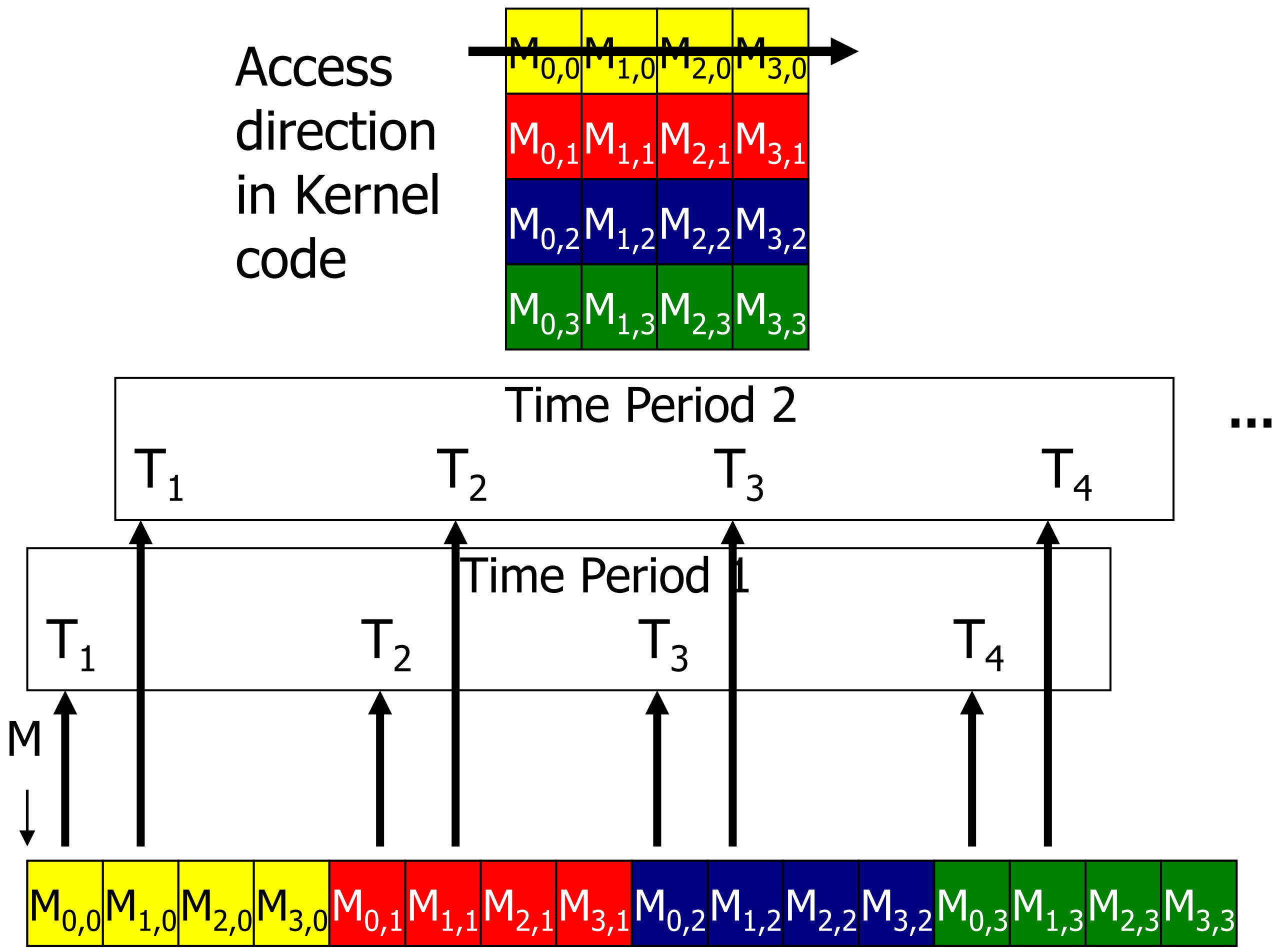
Memory Coalescing

- 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



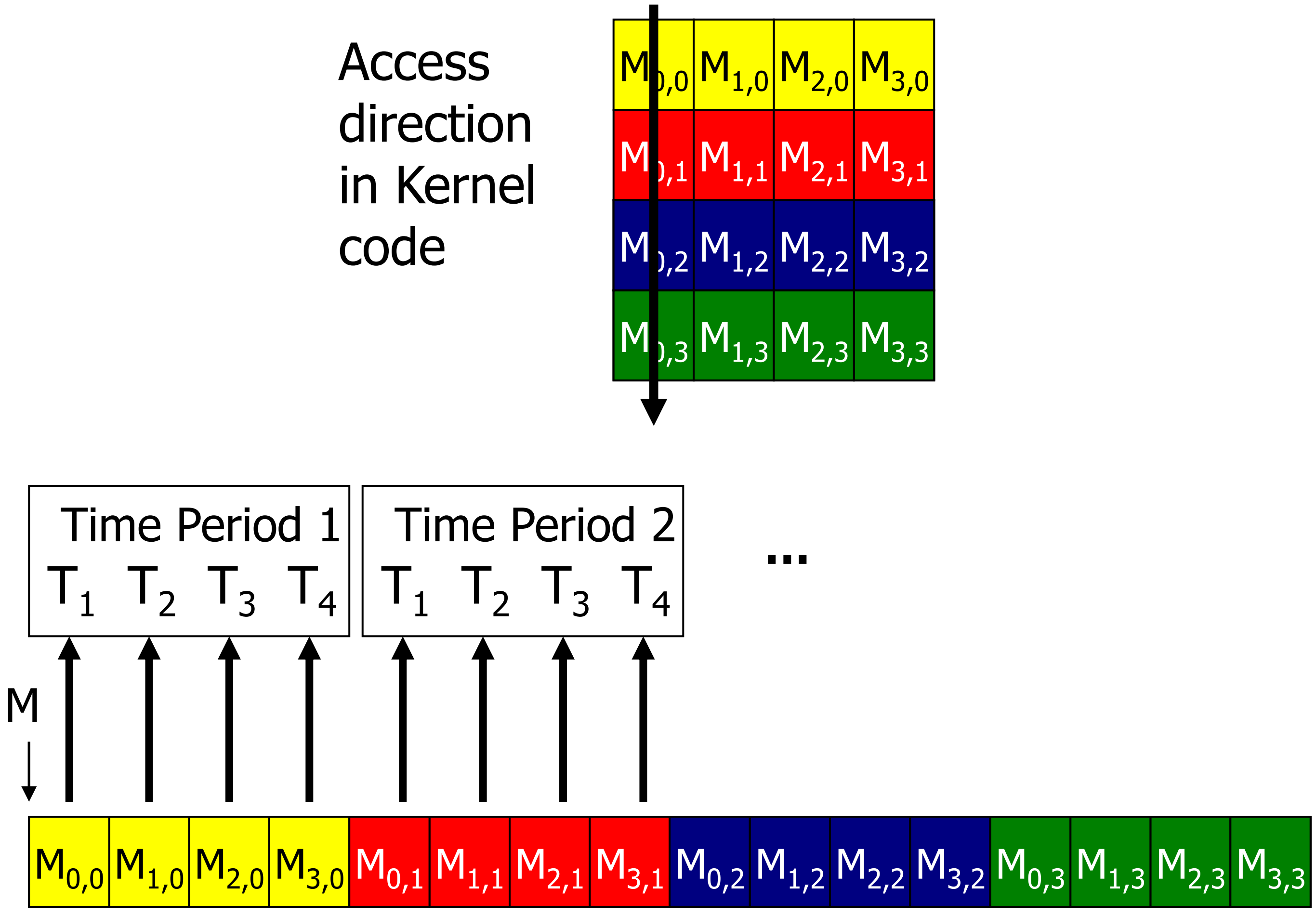
Slide credit: Hwu & Kirk

Uncoalesced Memory Accesses


Access direction in Kernel code
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
Time Period 2
T_1 T_2 T_3 T_4
...
Time Period 1
T_1 T_2 T_3 T_4
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

Slide credit: Hwu & Kirk

Coalesced Memory Accesses


Access direction in Kernel code
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
Time Period 1
T_1 T_2 T_3 T_4
Time Period 2
T_1 T_2 T_3 T_4
…
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

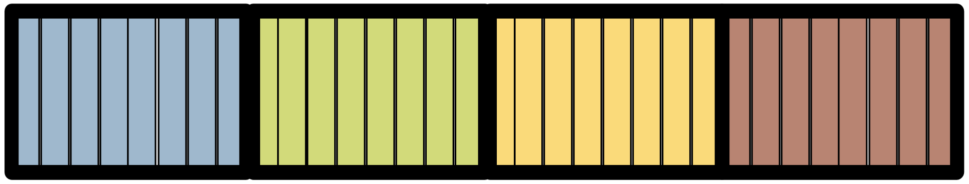
Slide credit: Hwu & Kirk

AoS vs. SoA

- Array of Structures vs. Structure of Arrays

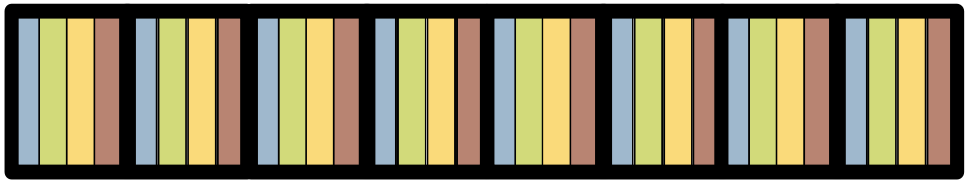
Structure of Arrays (SoA)

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



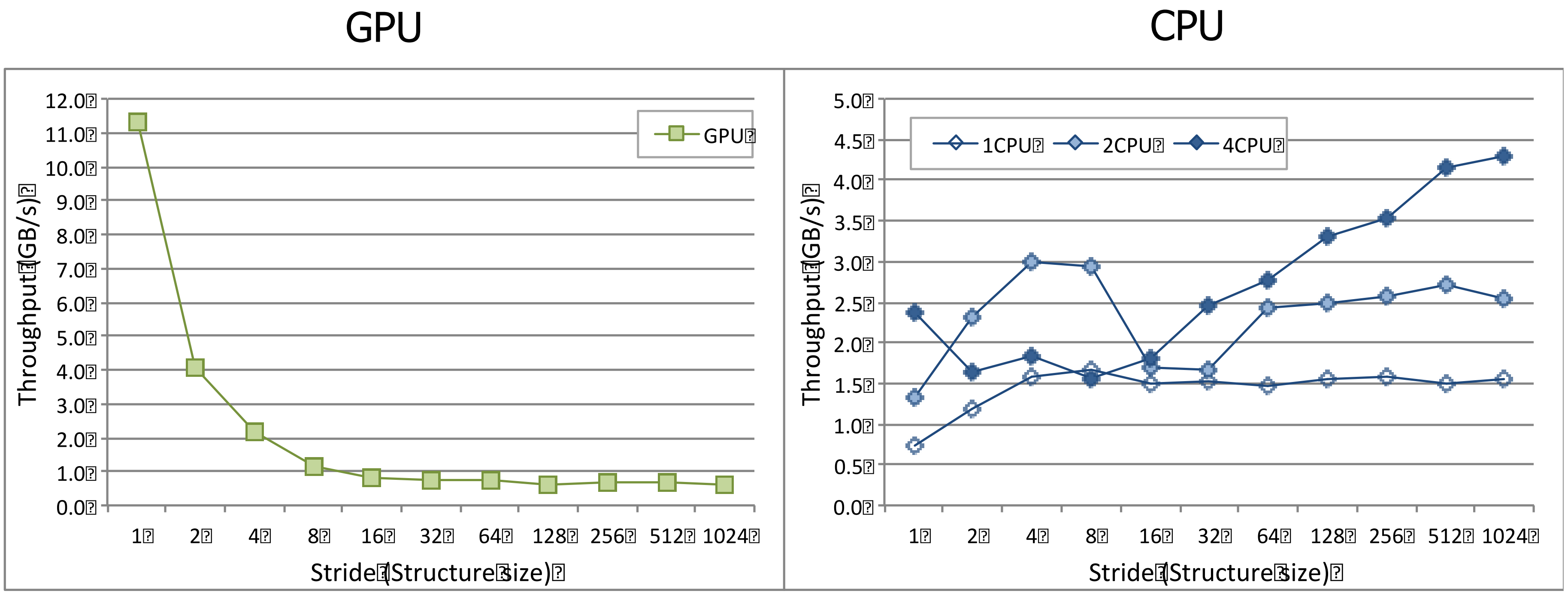
Array of Structures (AoS)

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



CPUs Prefer AoS, GPUs Prefer SoA

- Linear and strided accesses

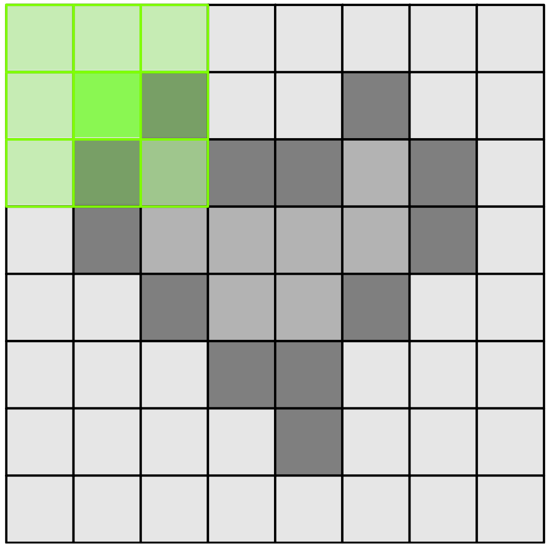


AMD Kaveri A10-7850K

Sung+, "DL: A data layout transformation system for heterogeneous computing," INPAR 2012

Data Reuse

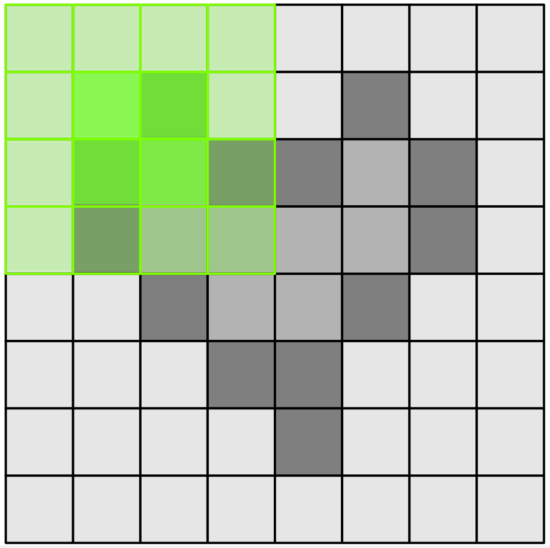
- Same memory locations accessed by neighboring threads



```
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

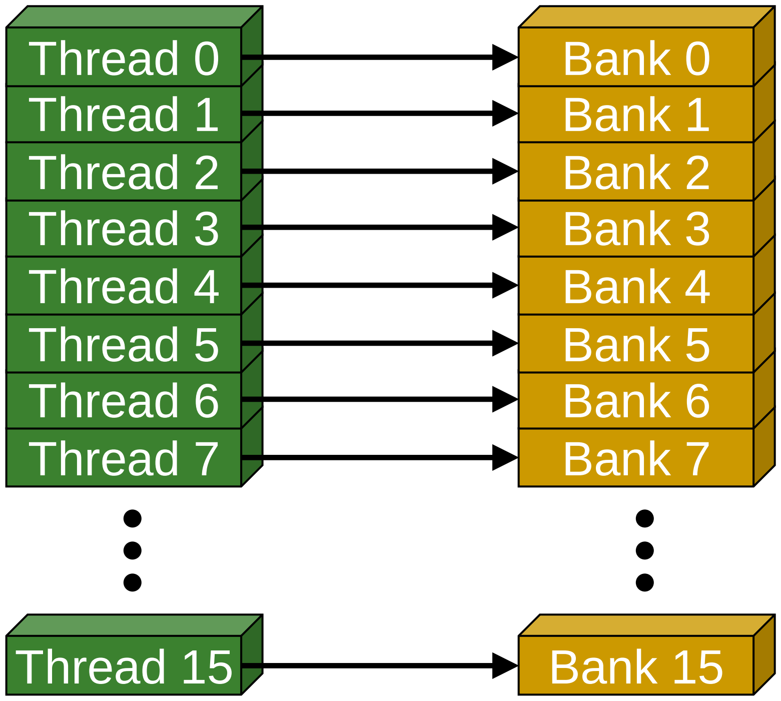
- Shared memory is an interleaved (banked) memory
  - Each bank can service one address per cycle

- Typically, 32 banks in NVIDIA GPUs
  - Successive 32-bit words are assigned to successive banks
    - Bank = Address % 32

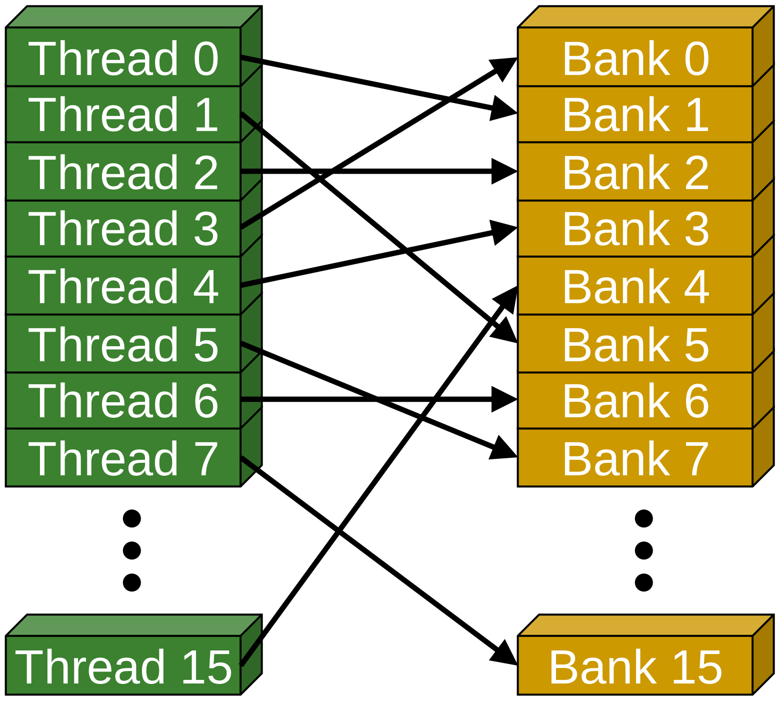
- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps

Shared Memory Bank Conflicts (I)

- Bank conflict free



Linear addressing: stride = 1

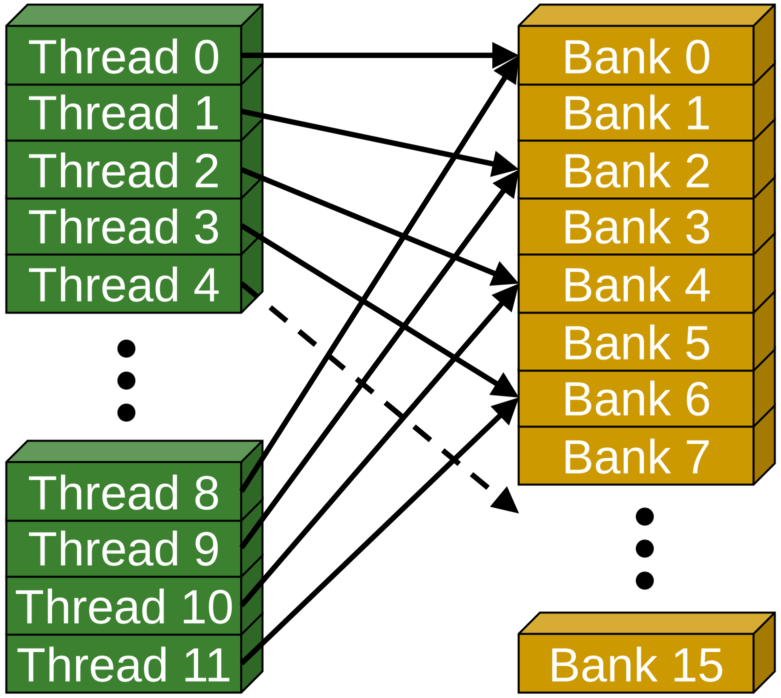


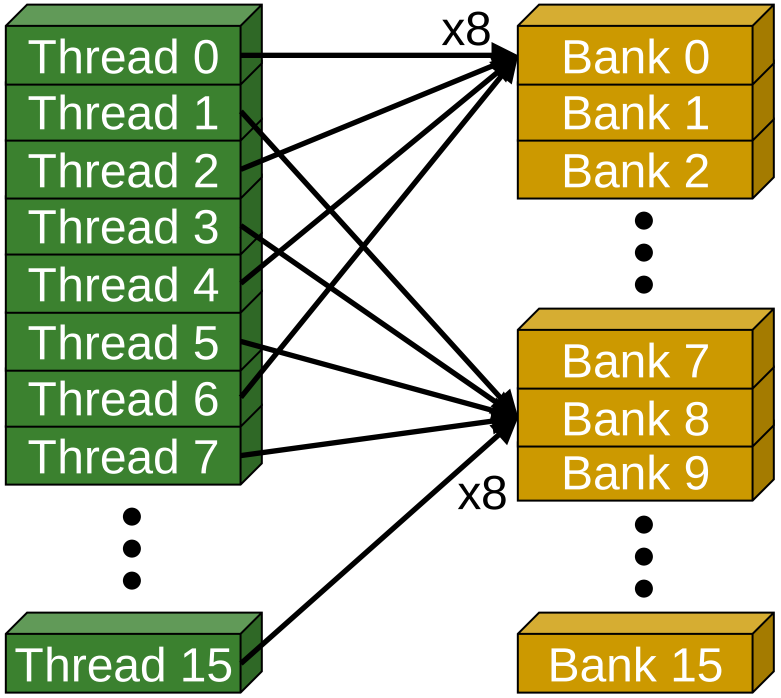
Random addressing 1:1

Slide credit: Hwu & Kirk

Shared Memory Bank Conflicts (II)

- N-way bank conflicts





2-way bank conflict: stride = 2

8-way bank conflict: stride = 8

Slide credit: Hwu & Kirk

Reducing Shared Memory Bank Conflicts

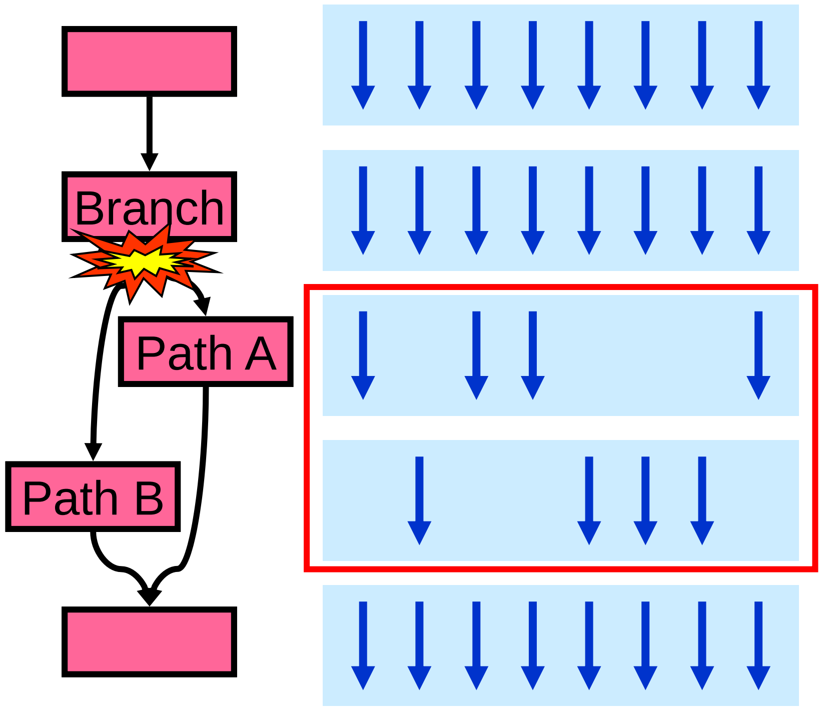
- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps

- 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

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



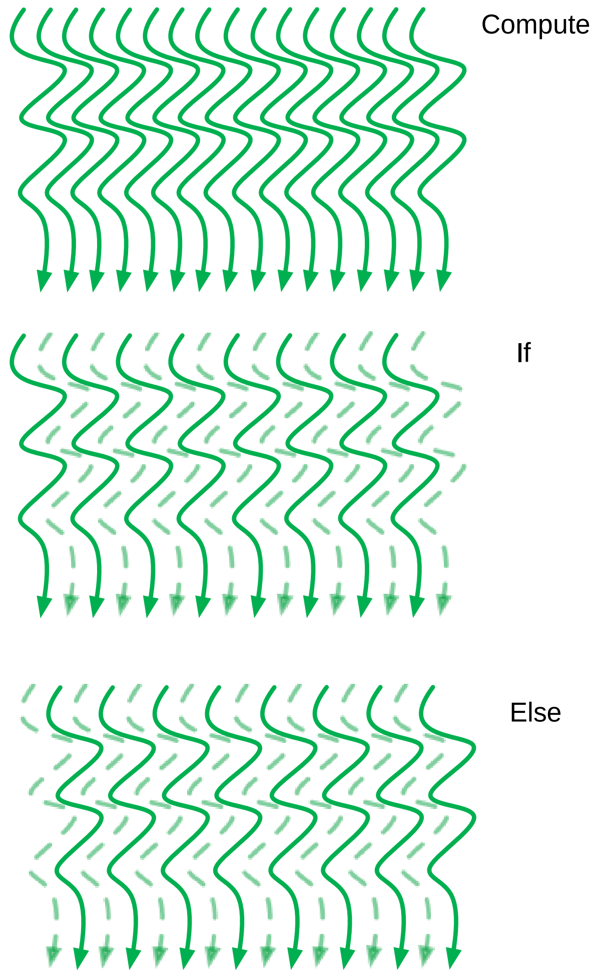
This is the same as conditional/predicated/masked execution. Recall the Vector Mask and Masked Vector Operations?

Slide credit: Tor Aamodt

SIMD Utilization

- Intra-warp divergence

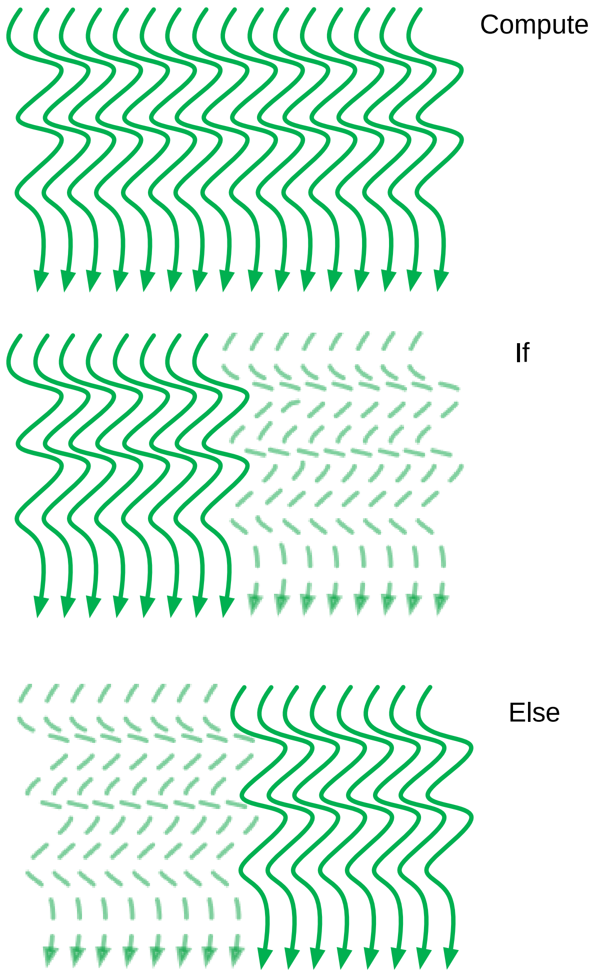
```
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
  Do_this(threadIdx.x);
}
else{
  Do_that(threadIdx.x);
}
```



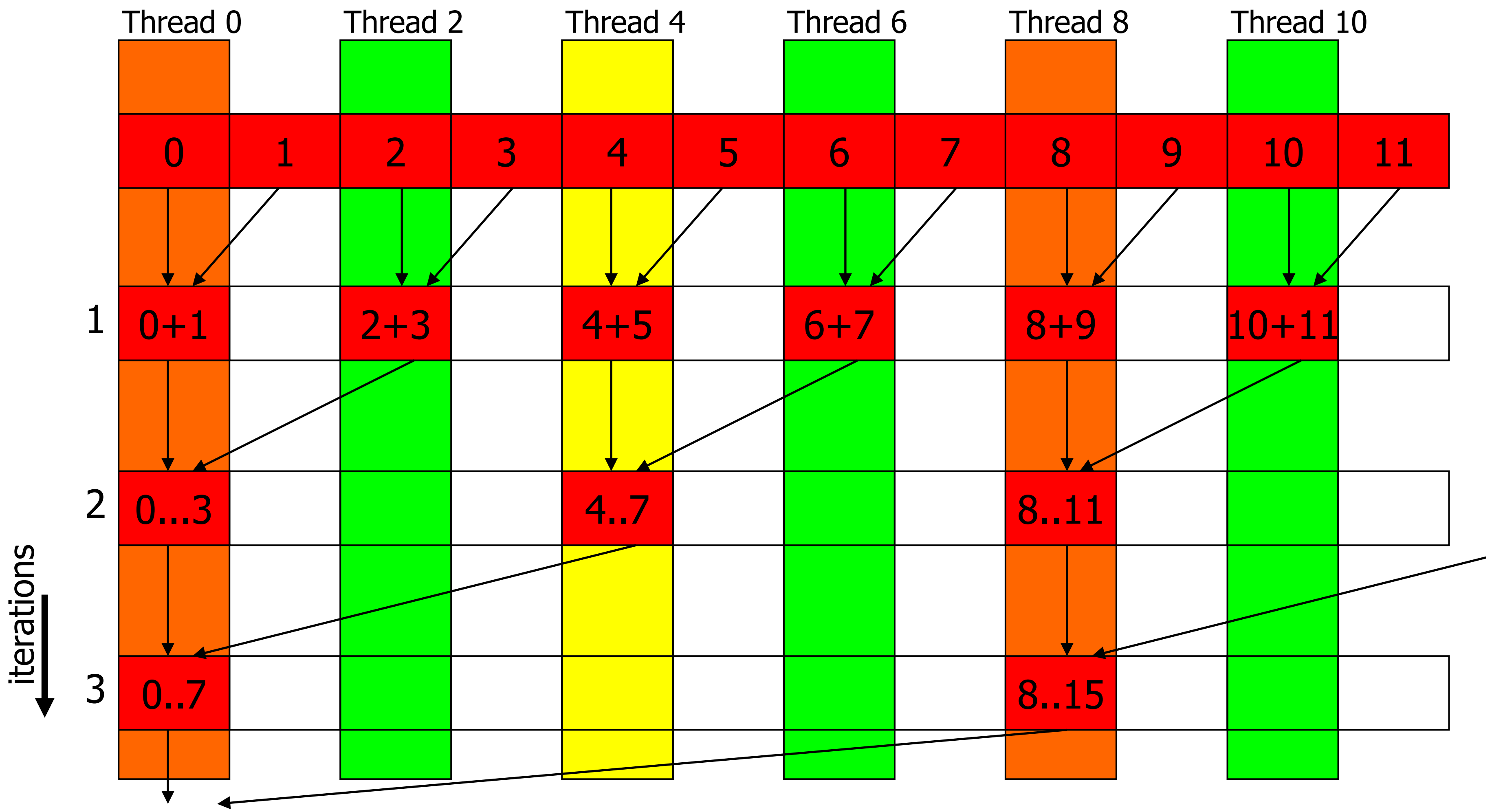
Increasing SIMD Utilization

- 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

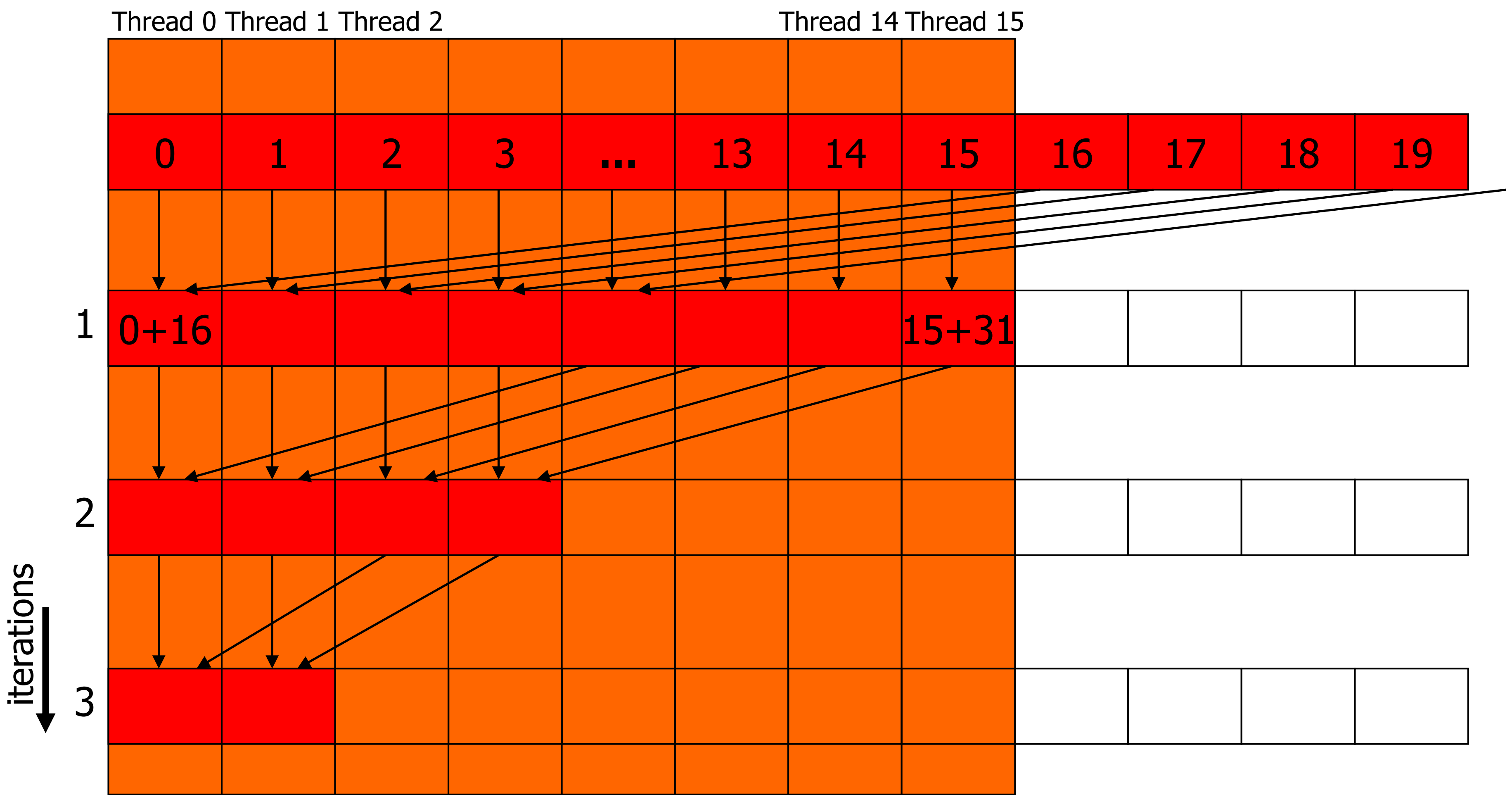
Vector Reduction: Naïve Mapping (II)

- 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;
```

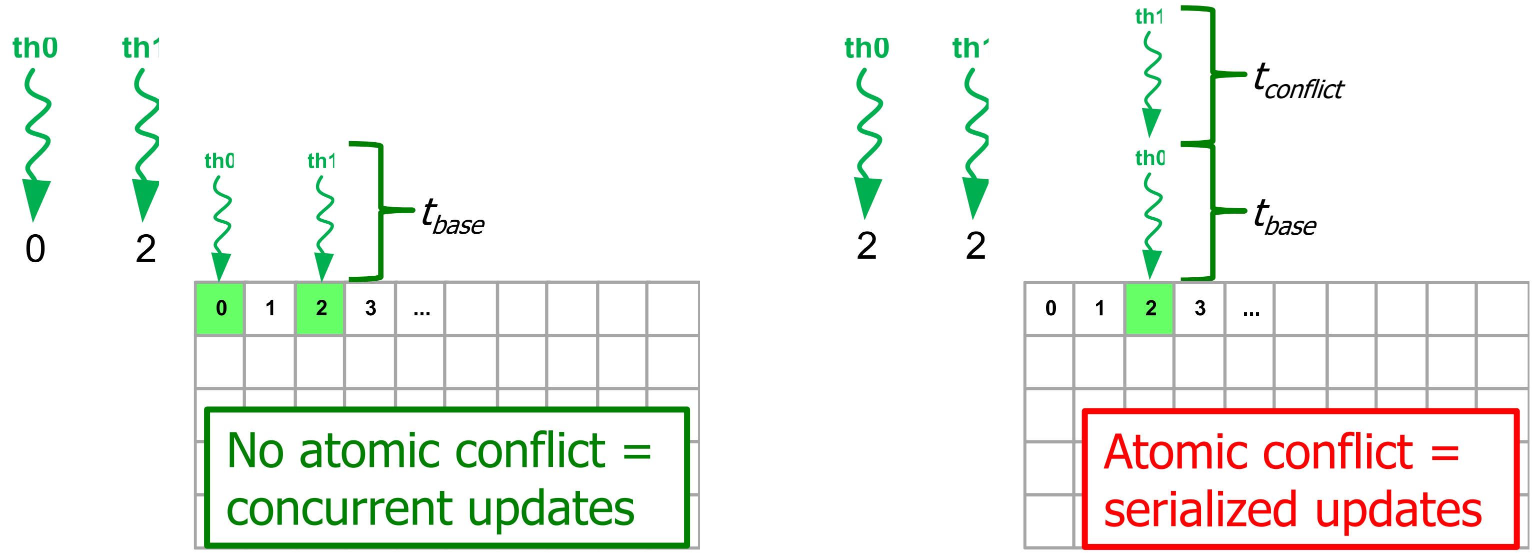
Maxwell, Pascal, Volta

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

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

Atomic Conflicts

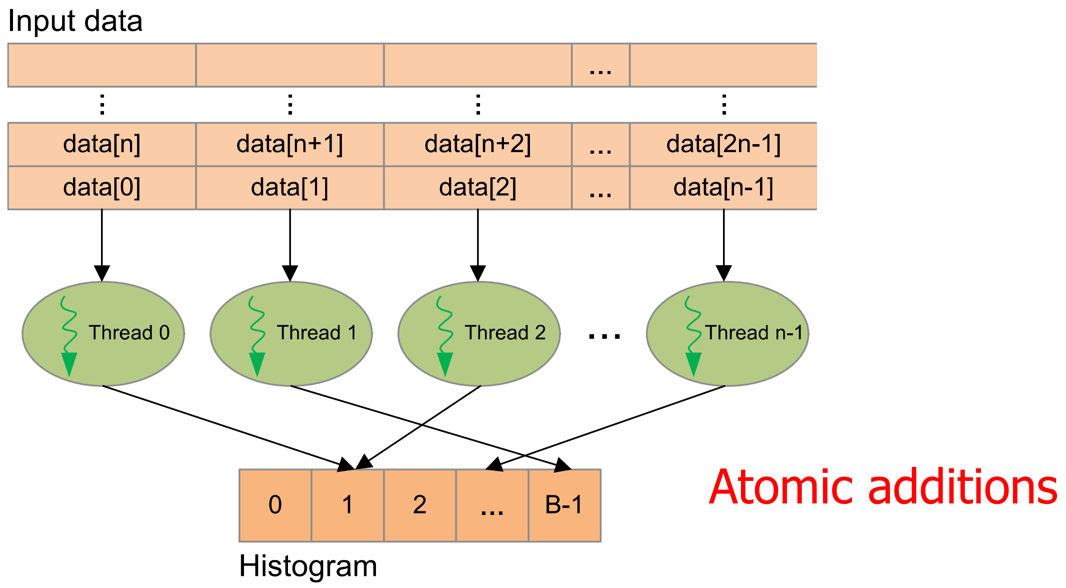
- We define the intra-warp conflict degree as the number of threads in a warp that update the same memory position
- The conflict degree can be between 1 and 32



Histogram Calculation

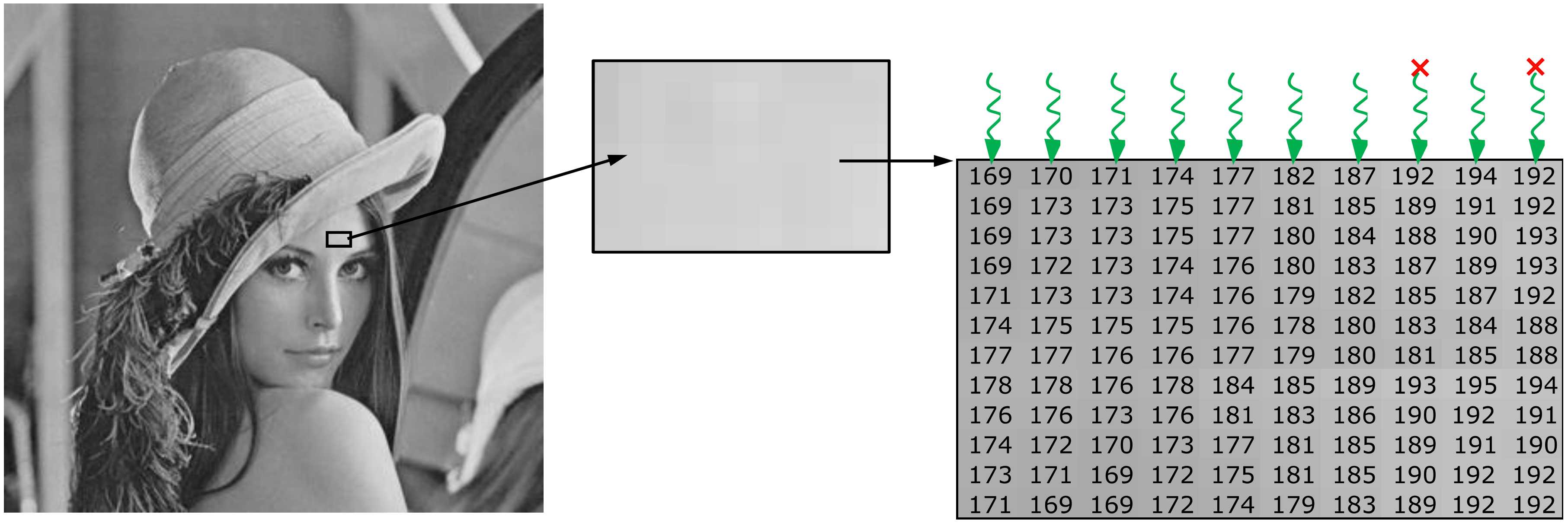
- Histograms count the number of data instances in disjoint categories (bins)

```
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

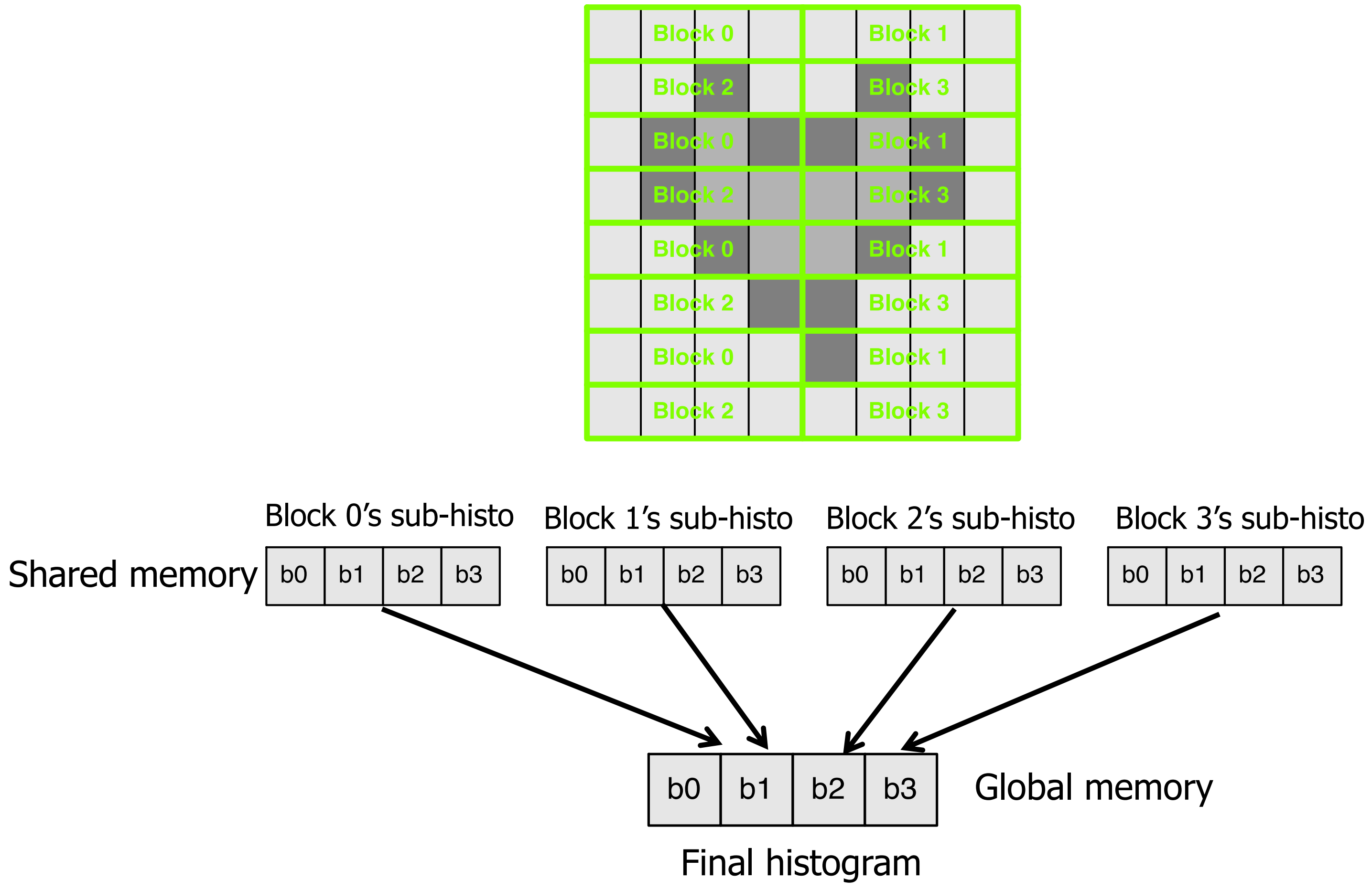
- <span>Frequent conflicts</span> in natural images



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

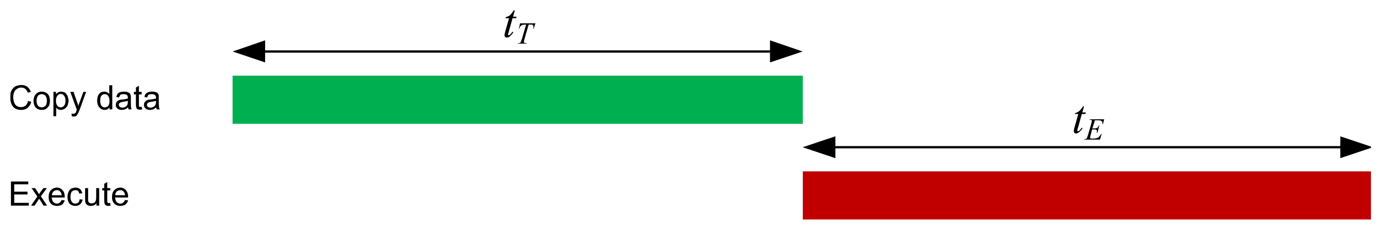


Gomez-Luna+, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," IEEE TPDS, 2013.

Data Transfers between CPU and GPU

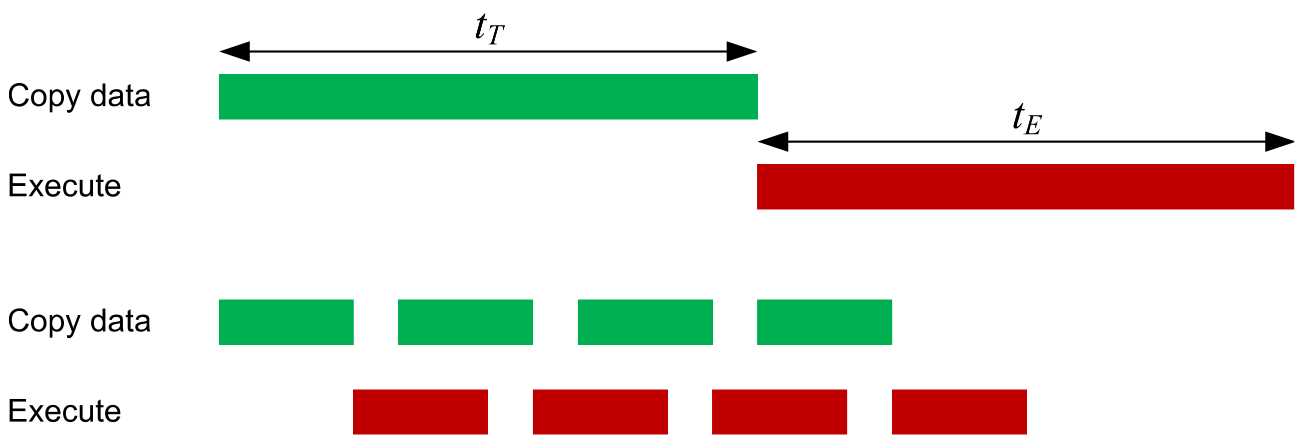
Data Transfers

- 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

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

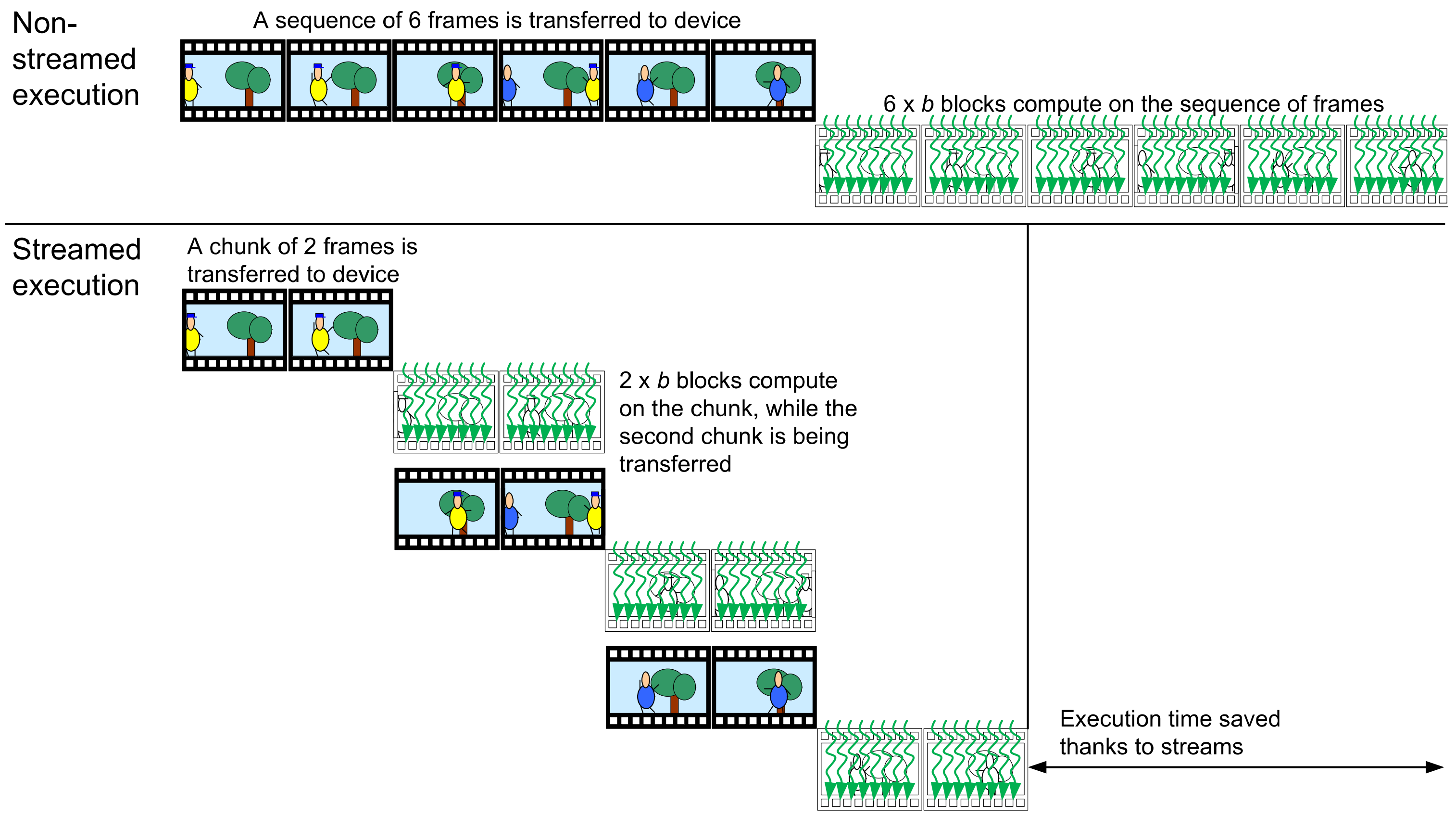
$t_E >= t_T$ (dominant kernel)

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

$t_T > t_E$ (dominant transfers)

Overlap of Communication and Computation

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


Non-streamed execution
A sequence of 6 frames is transferred to device
6 x *b* blocks compute on the sequence of frames
Streamed execution
A chunk of 2 frames is transferred to device
2 x *b* blocks compute on the chunk, while the second chunk is being transferred
Execution time saved thanks to streams

Gomez-Luna+, "Performance models for asynchronous data transfers on consumer Graphics Processing Units," JPDC, 2012.

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

Collaborative Computing

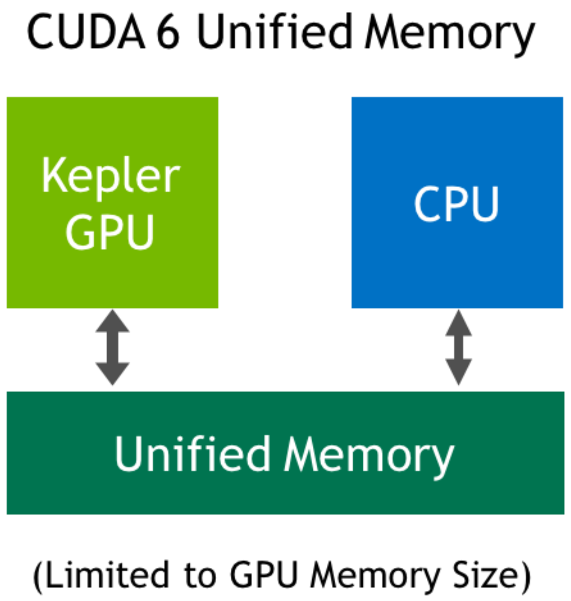
Review

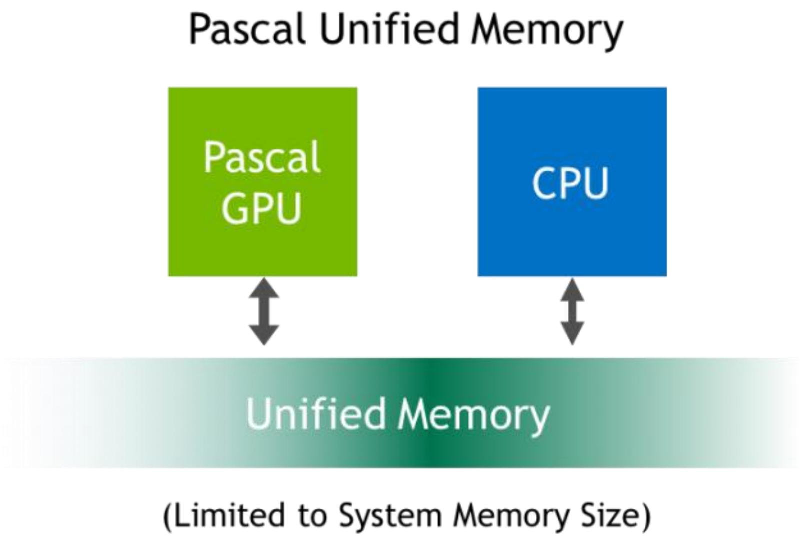
- Device allocation, CPU-GPU transfer, and GPU-CPU transfer
  - `cudaMalloc();`
  - `cudaMemcpy();`

```
// Allocate input
malloc(input, ...);
cudaMalloc(d_input, ...);
cudaMemcpy(d_input, input, ..., HostToDevice); // Copy to device memory

// Allocate output
malloc(output, ...);
cudaMalloc(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();

// Copy output to host memory
cudaMemcpy(output, d_output, ..., DeviceToHost);
```

Unified Memory (I)

- Unified Virtual Address
- Since CUDA 6.0: Unified memory
- Since CUDA 8.0 + Pascal: GPU page faults





Unified Memory (II)

- Easier programming with Unified Memory
  - `cudaMallocManaged();`

```
// Allocate input
malloc(input, ...);
cudaMallocManaged(d_input, ...);
memcpy(d_input, input, ...); // Copy to managed memory

// Allocate output
cudaMallocManaged(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();
```

Collaborative Computing Algorithms

- Case studies using CPU and GPU
- Kernel launches are asynchronous
  - CPU can work while waits for GPU to finish
  - Traditionally, this is the most efficient way to exploit heterogeneity

```
// Allocate input
malloc(input, ...);
cudaMalloc(d_input, ...);
cudaMemcpy(d_input, input, ..., HostToDevice); // Copy to device memory

// Allocate output
malloc(output, ...);
cudaMalloc(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// CPU can do things here

// Synchronize
cudaDeviceSynchronize();

// Copy output to host memory
cudaMemcpy(output, d_output, ..., DeviceToHost);
```

Fine-Grained Heterogeneity

- Fine-grain heterogeneity becomes possible with Pascal/Volta architecture
- Pascal/Volta Unified Memory
  - CPU-GPU memory coherence
  - System-wide atomic operations

```
// Allocate input
cudaMallocManaged(input, ...);

// Allocate output
cudaMallocManaged(output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (output, input, ...);

// CPU can do things here
output[x] = input[y];

output[x+1].fetch_add(1);
```

Since CUDA 8.0

- Unified memory

```
cudaMallocManaged(&h_in, in_size);
```

- System-wide atomics

```
old = atomicAdd_system(&h_out[x], inc);
```

Since OpenCL 2.0

- Shared virtual memory

```
 XYZ * h_in = (XYZ *)clSVMAlloc(
             ocl.clContext, CL_MEM_SVM_FINE_GRAIN_BUFFER, in_size, 0);
```

- More flags:

```
CL_MEM_READ_WRITE
CL_MEM_SVM_ATOMICS
```

- C++11 atomic operations

  (`memory_scope_all_svm_devices`)

```
old = atomic_fetch_add(&h_out[x], inc);
```

C++AMP (HCC)

- Unified memory space (HSA)

```
XYZ *h_in = (XYZ *)malloc(in_size);
```

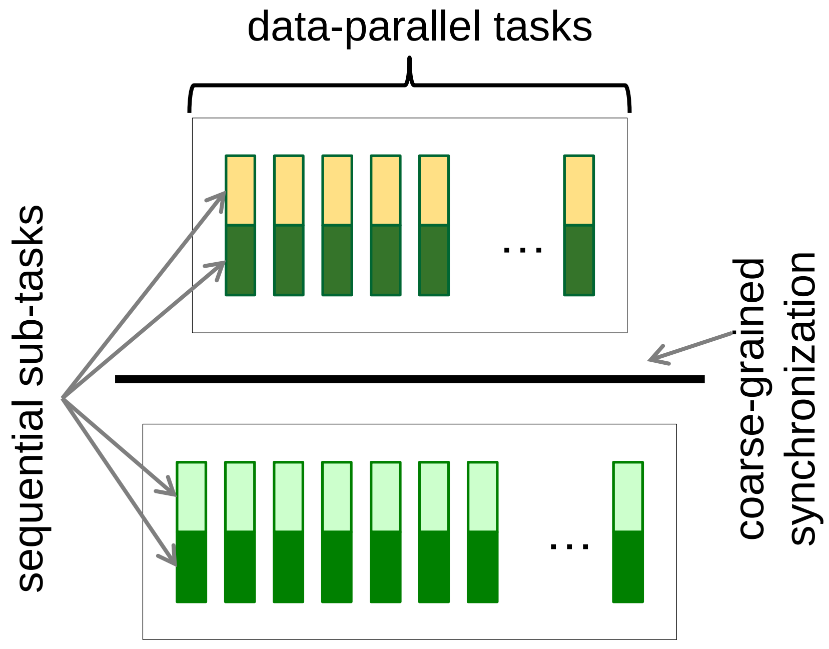
- C++11 atomic operations

  (`memory_scope_all_svm_devices`)

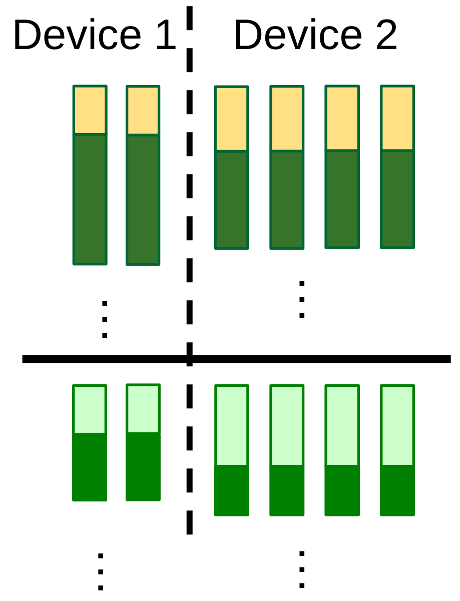
  - Platform atomics (HSA)

```
old = atomic_fetch_add(&h_out[x], inc);
```

Collaborative Patterns (I)

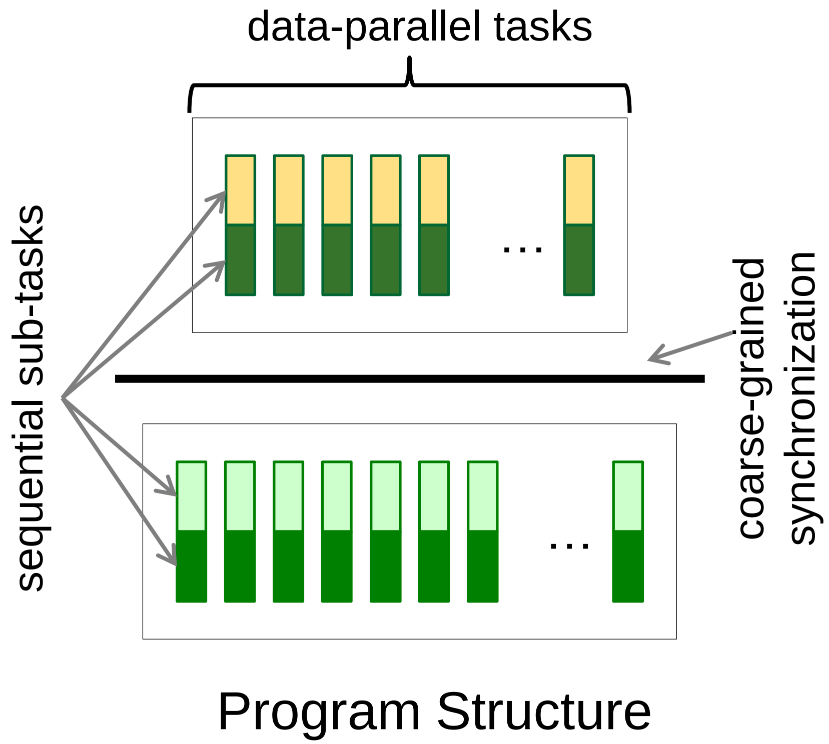


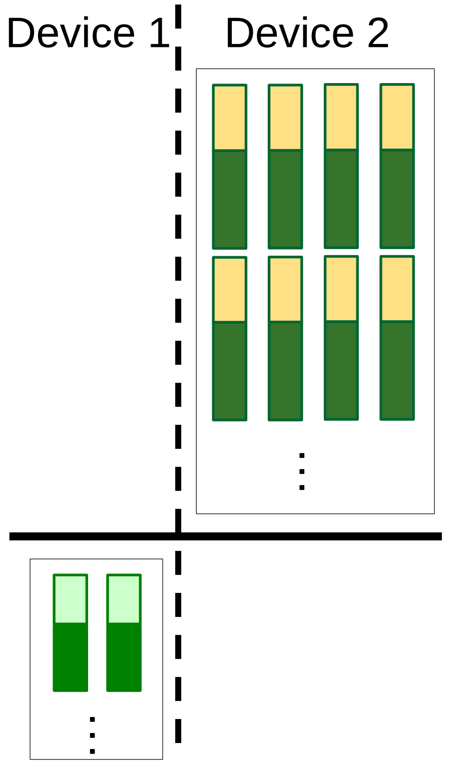
Program Structure



Data Partitioning

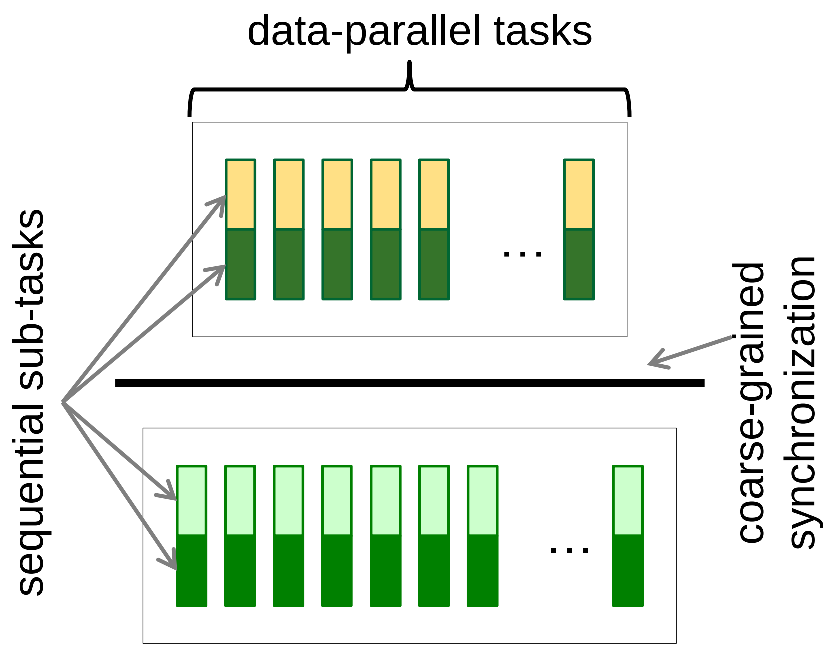
Collaborative Patterns (II)



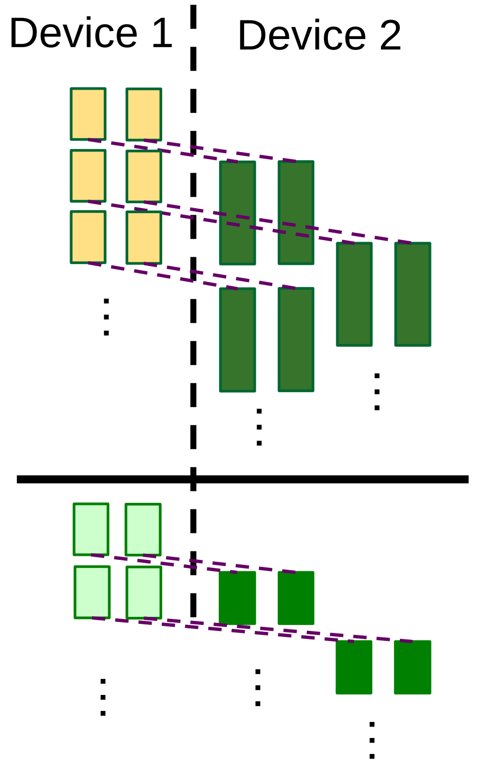


Coarse-grained Task Partitioning

Collaborative Patterns (III)



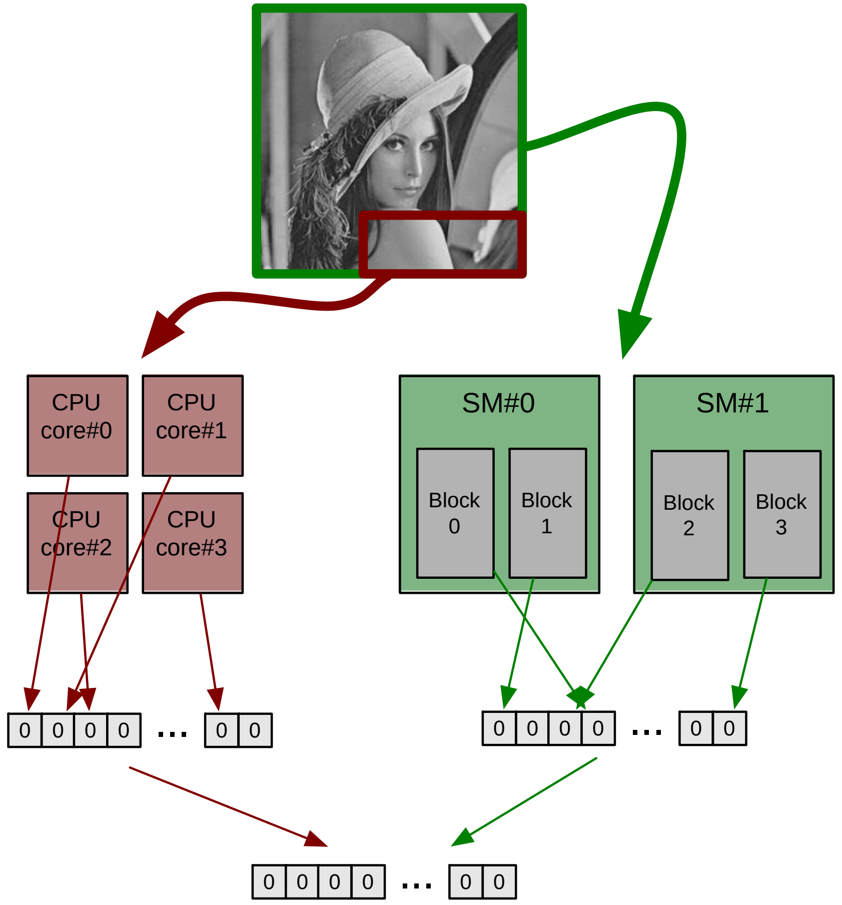
Program Structure



Fine-grained Task Partitioning

Histogram (I)

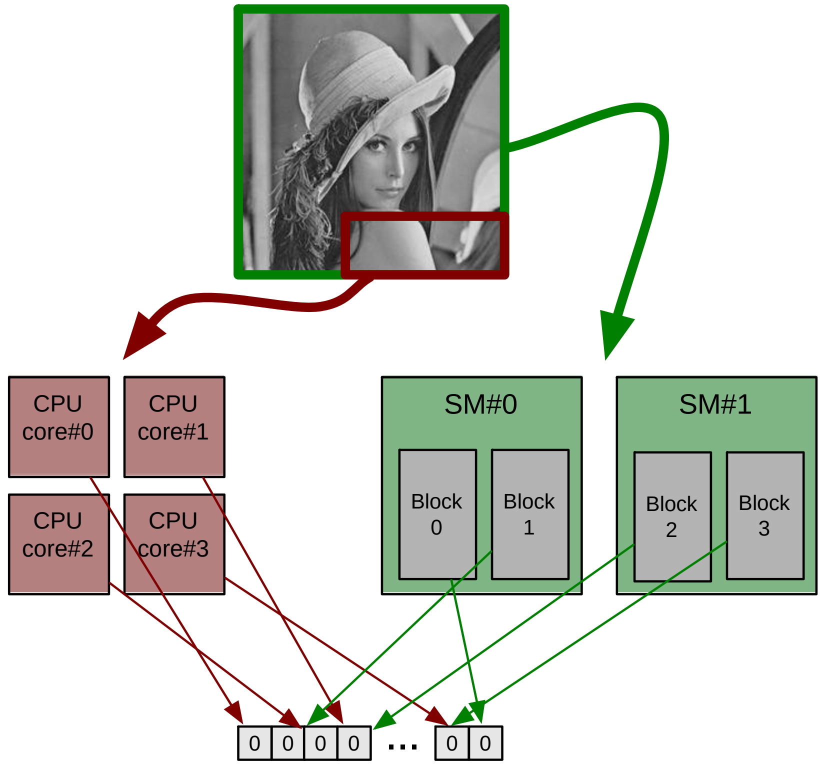
- Previous generations: separate CPU and GPU histograms are merged at the end



```
malloc(CPU image);
cudaMalloc(GPU image);
cudaMemcpy(GPU image, CPU image, ...,
           Host to Device);
malloc(CPU histogram);
memset(CPU histogram, 0);
cudaMalloc(GPU histogram);
cudaMemset(GPU histogram, 0);

// Launch CPU threads
// Launch GPU kernel

cudaMemcpy(GPU histogram, DeviceToHost);

// Launch CPU threads for merging
```

Histogram (II)

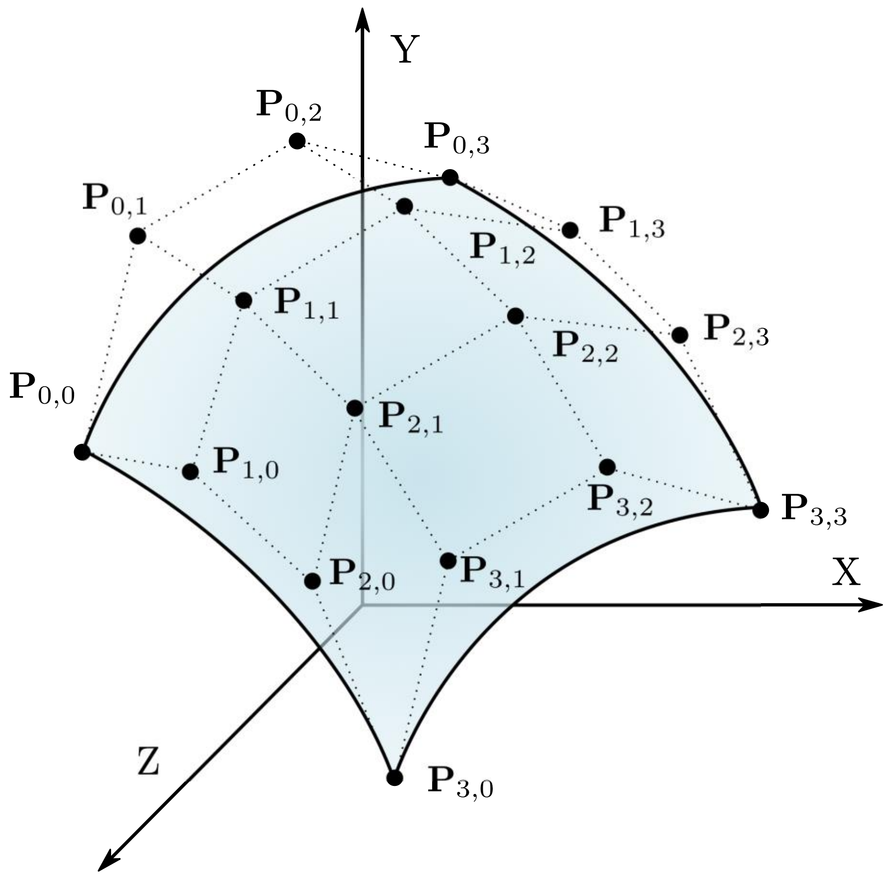
- System-wide atomic operations: one single histogram



```
cudaMallocManaged(Histogram);
cudaMemset(Histogram, 0);

// Launch CPU threads
// Launch GPU kernel (atomicAdd_system)
```

Bézier Surfaces (I)

- Bézier surface: 4x4 net of control points



Bézier Surfaces (II)

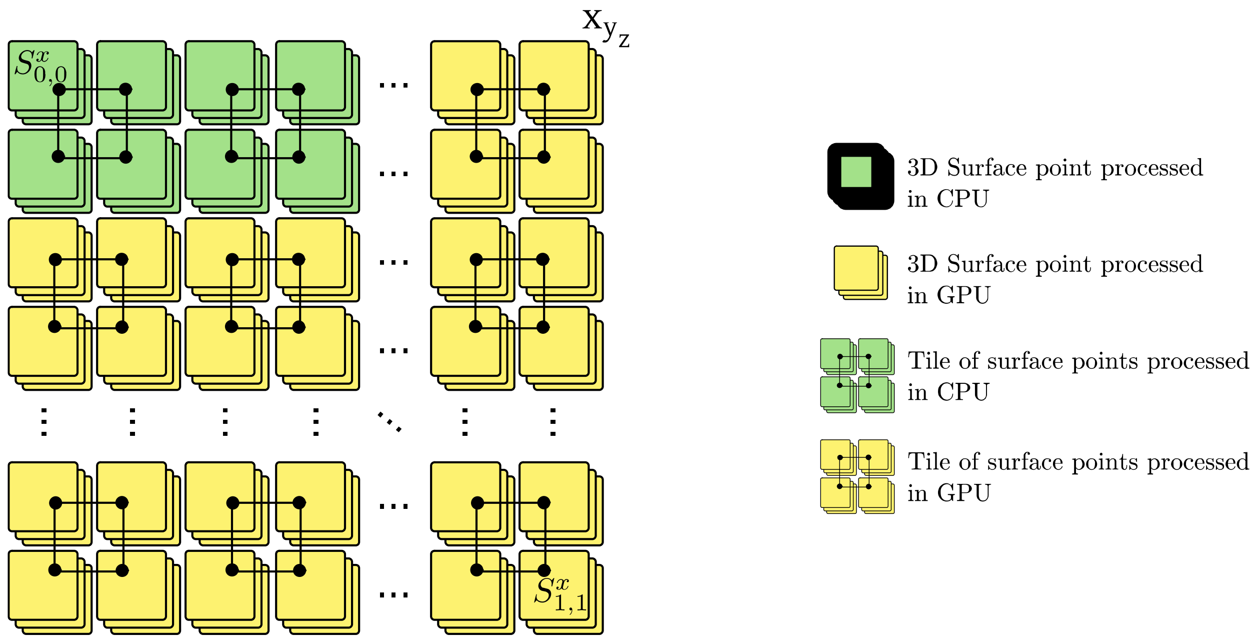
- Parametric non-rational formulation
  - Bernstein polynomials
  - Bi-cubic surface $m = n = 3$

$$\mathbf{S}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{P}_{i,j} B_{i,m}(u) B_{j,n}(v), \qquad (1)$$

$$B_{i,m}(u) = \binom{m}{i} (1-u)^{(m-i)} u^{i}, \qquad (2)$$

Bézier Surfaces (III)

- Collaborative implementation
  - Tiles calculated by GPU blocks or CPU threads
  - Static distribution



$S^x_{0,0}$
$S^x_{1,1}$
x y z
3D Surface point processed in CPU
3D Surface point processed in GPU
Tile of surface points processed in CPU
Tile of surface points processed in GPU

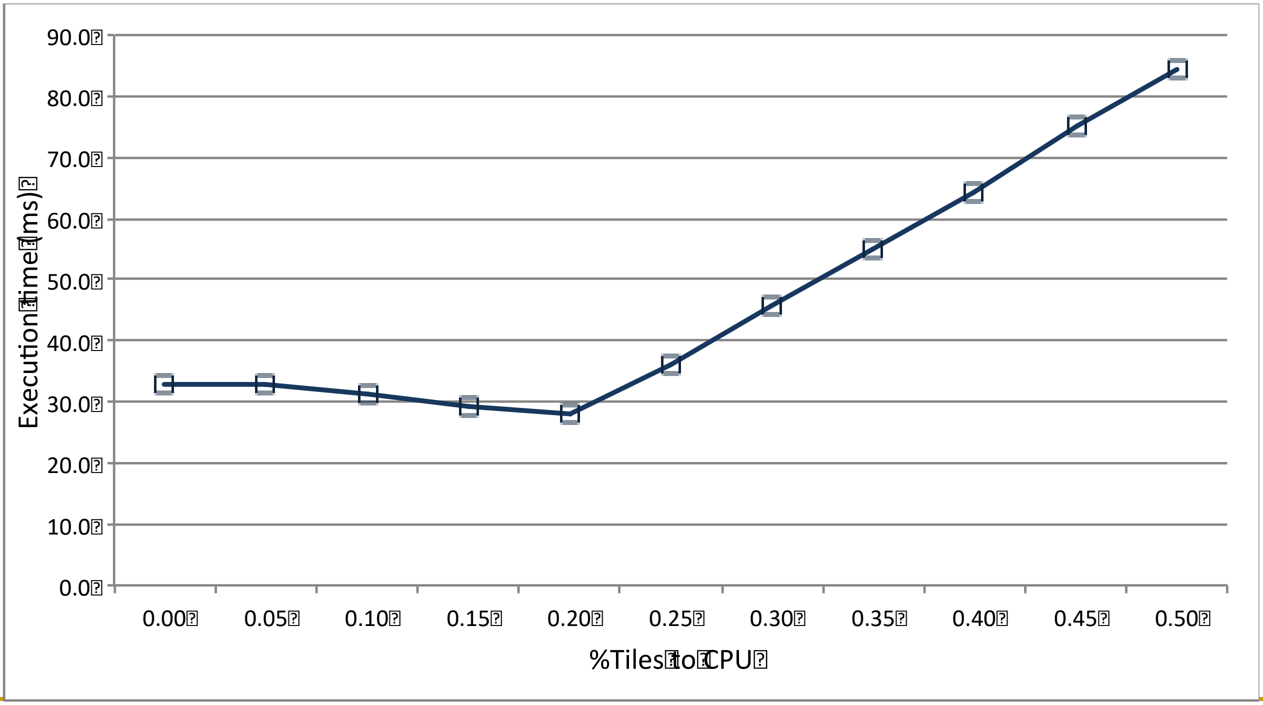
Bézier Surfaces (IV)

- Without Unified Memory

```
// Allocate control points
malloc(control_points, ...);
generate_cp(control_points);
cudaMalloc(d_control_points, ...);
cudaMemcpy(d_control_points, control_points, ..., HostToDevice); // Copy to device memory

// Allocate surface
malloc(surface, ...);
cudaMalloc(d_surface, ...);

// Launch CPU threads
std::thread main_thread (run_cpu_threads, control_points, surface, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_surface, d_control_points, ...);

// Synchronize
main_thread.join();
cudaDeviceSynchronize();

// Copy gpu part of surface to host memory
cudaMemcpy(&surface[end_of_cpu_part], d_surface, ..., DeviceToHost);
```

Bézier Surfaces (V)

- Execution results
  - Bezier surface: 300x300, 4x4 control points
  - %Tiles to CPU
  - NVIDIA Jetson TX1 (4 ARMv8 + 2 SMX): 17% speedup wrt GPU only



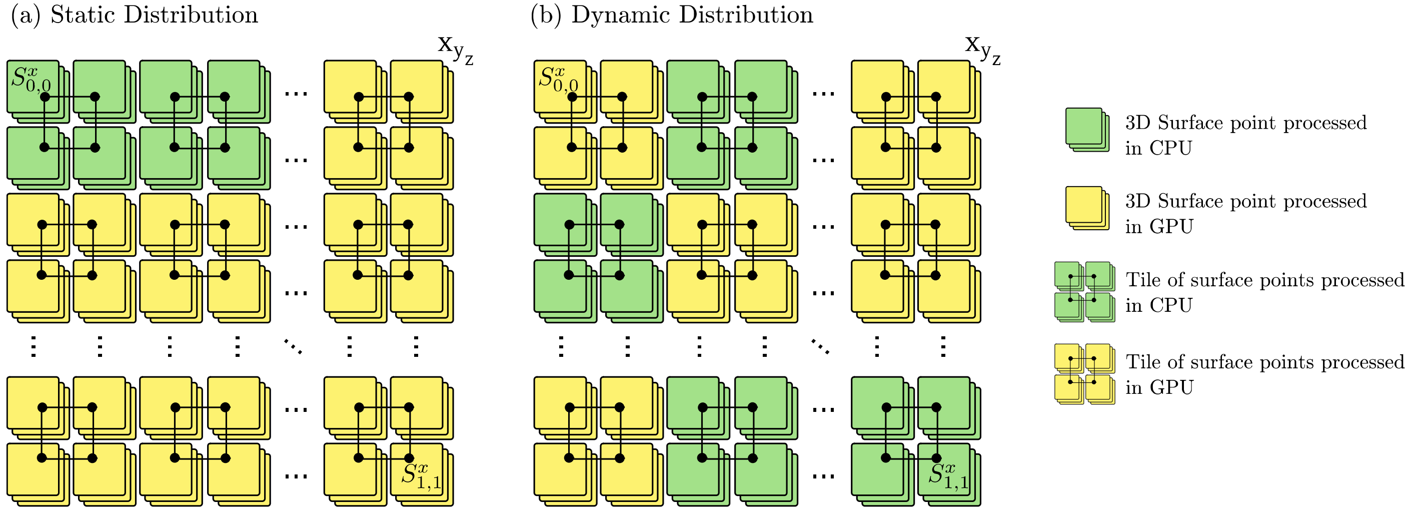
Bézier Surfaces (VI)

- With Unified Memory (Pascal/Volta)

```
// Allocate control points
malloc(control_points, ...);
generate_cp(control_points);
cudaMalloc(d_control_points, ...);
cudaMemcpy(d_control_points, control_points, ..., HostToDevice); // Copy to device memory

// Allocate surface
cudaMallocManaged(surface, ...);

// Launch CPU threads
std::thread main_thread (run_cpu_threads, control_points, surface, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (surface, d_control_points, ...);

// Synchronize
main_thread.join();
cudaDeviceSynchronize();
```

Bézier Surfaces (VII)

- Static vs. dynamic implementation



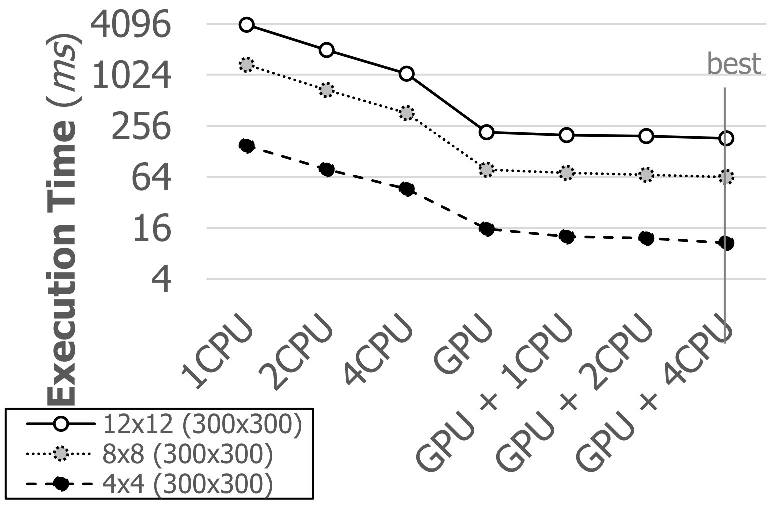
(a) Static Distribution
(b) Dynamic Distribution
x_{y_z}
$S^x_{0,0}$
$S^x_{1,1}$
3D Surface point processed in CPU
3D Surface point processed in GPU
Tile of surface points processed in CPU
Tile of surface points processed in GPU

  - Pascal/Volta Unified Memory: system-wide atomic operations

```
while(true){
    if(threadIdx.x == 0)
        my_tile = atomicAdd_system(tile_num, 1);  // my_tile in shared memory; tile_num in UM

    __syncthreads();  // Synchronization

    if(my_tile >= number_of_tiles) break;  // Break when all tiles processed
...
}
```

Benefits of Collaboration

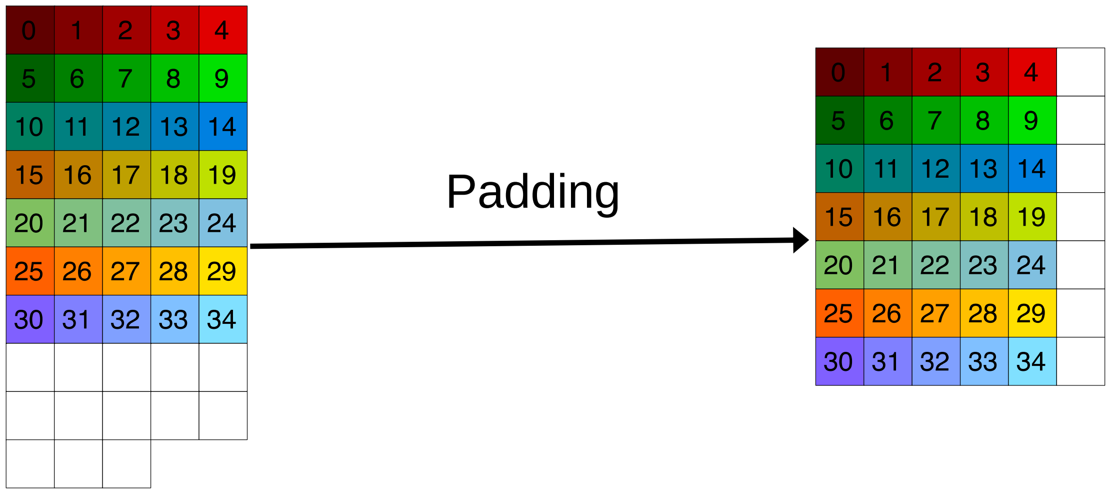
- Data partitioning improves performance
  - AMD Kaveri (4 CPU cores + 8 GPU CUs)



Bézier Surfaces
(up to 47% improvement over GPU only)

Padding (I)

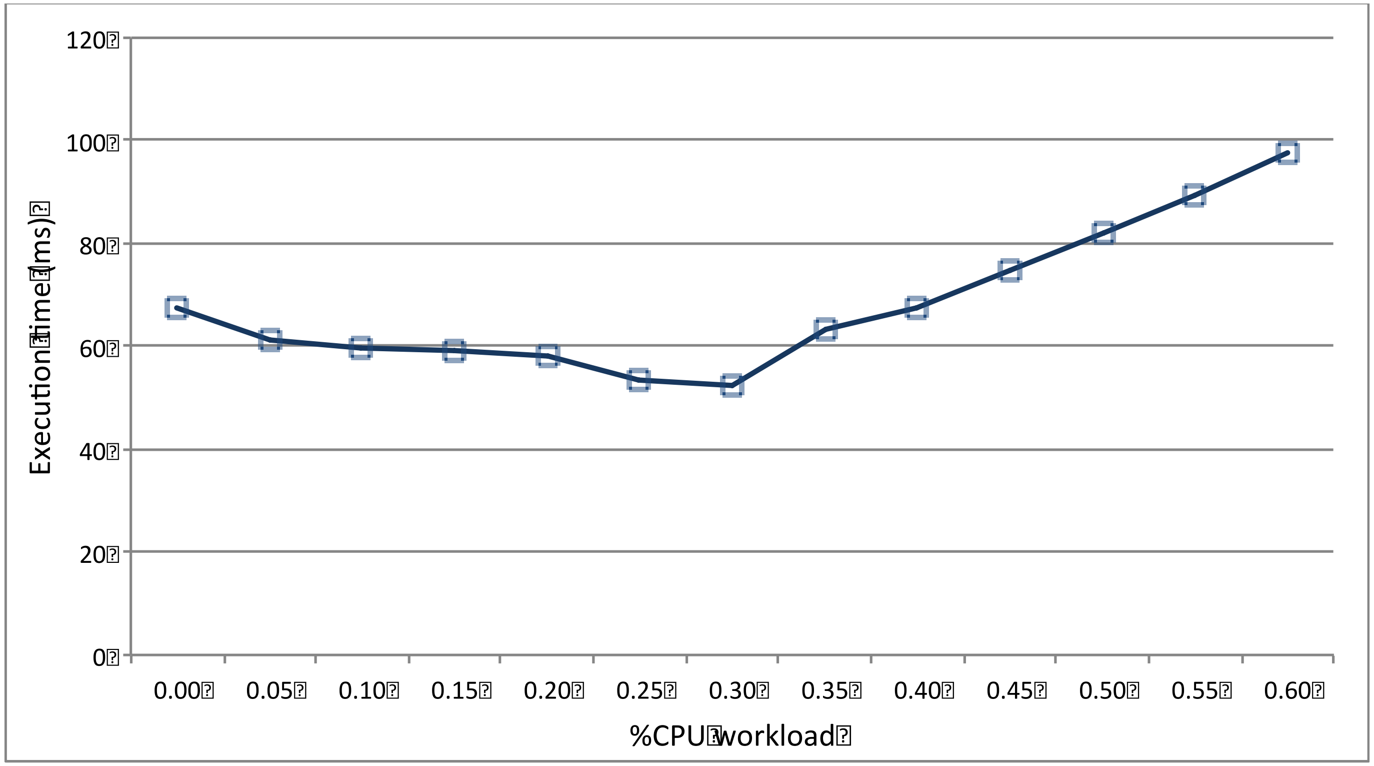
- Matrix padding
  - Memory alignment
  - Transposition of near-square matrices



- Traditionally, it can only be performed out-of-place

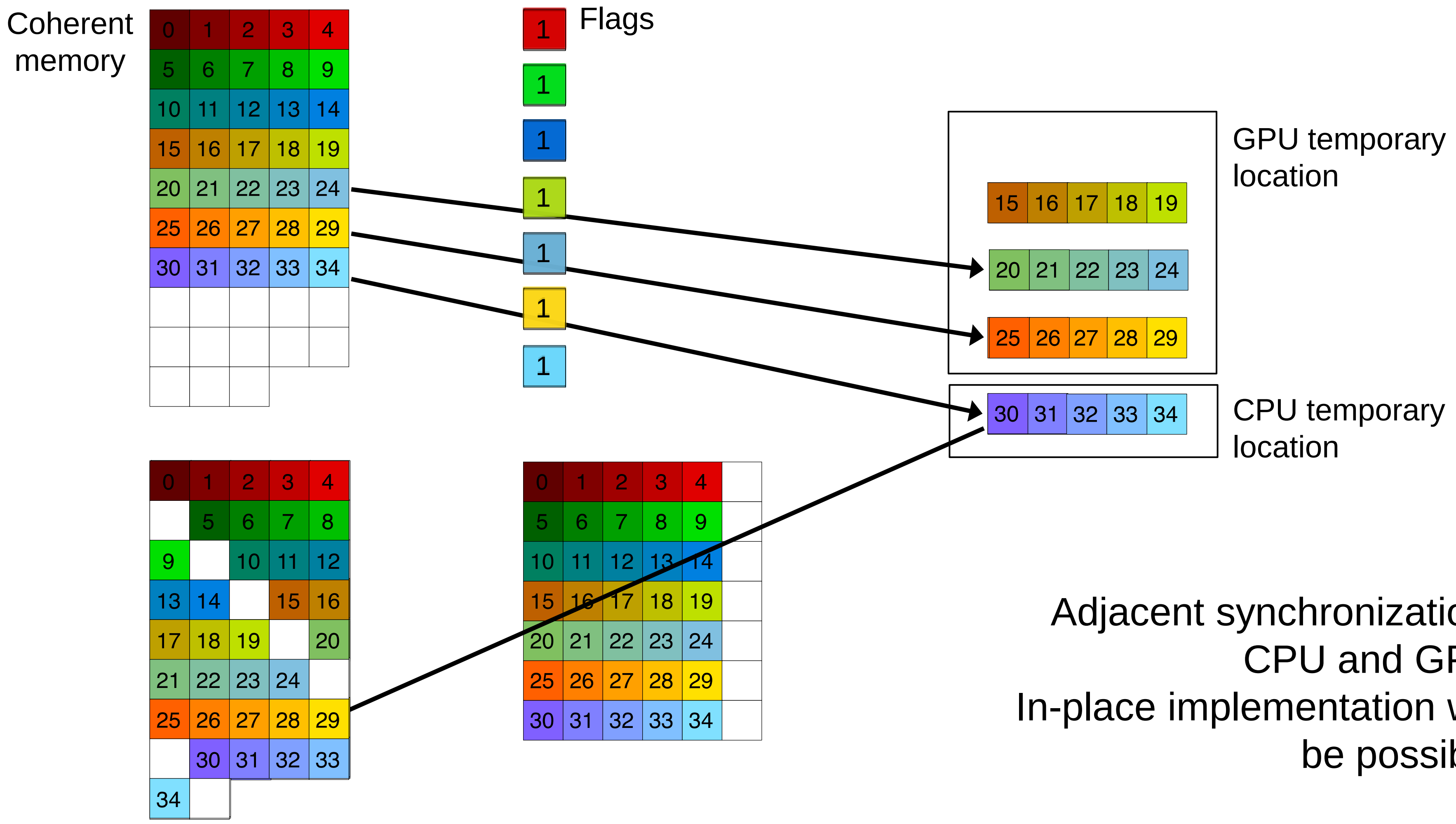
Padding (II)

- Execution results
  - Matrix size: 4000x4000, padding = 1
  - NVIDIA Jetson TX1 (4 ARMv8 + 2 SMX): 29% speedup wrt GPU only



In-Place Padding

- Pascal/Volta Unified Memory

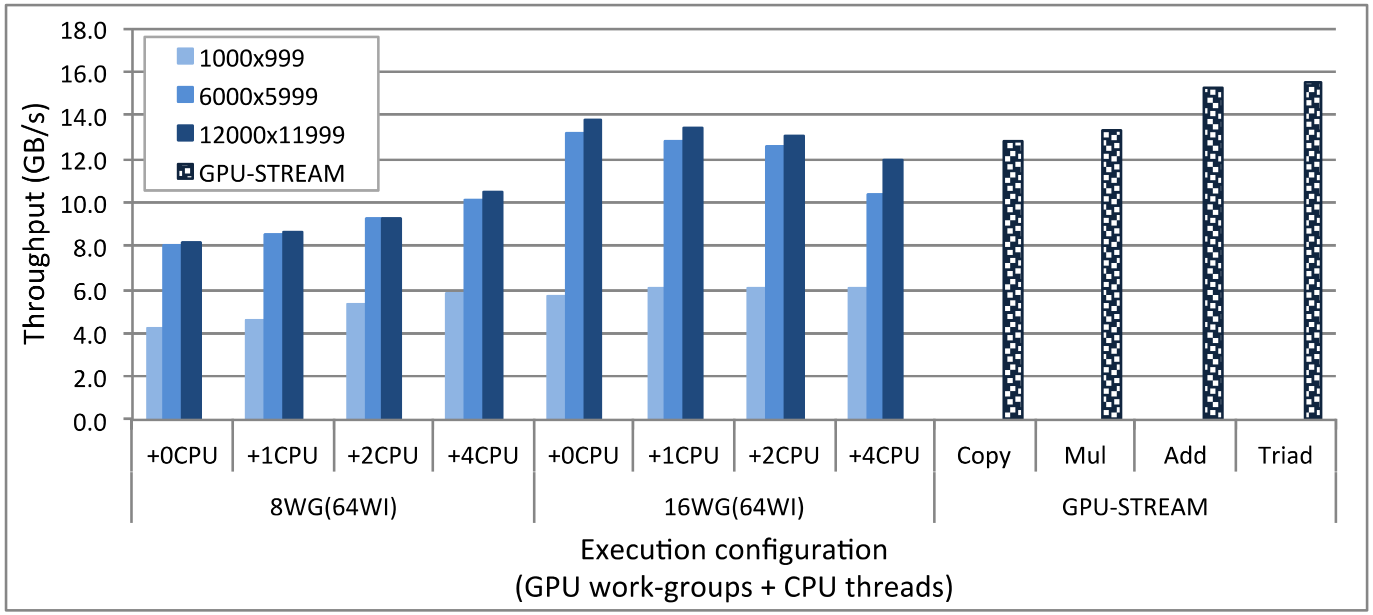


Adjacent synchronization: CPU and GPU

In-place implementation will be possible

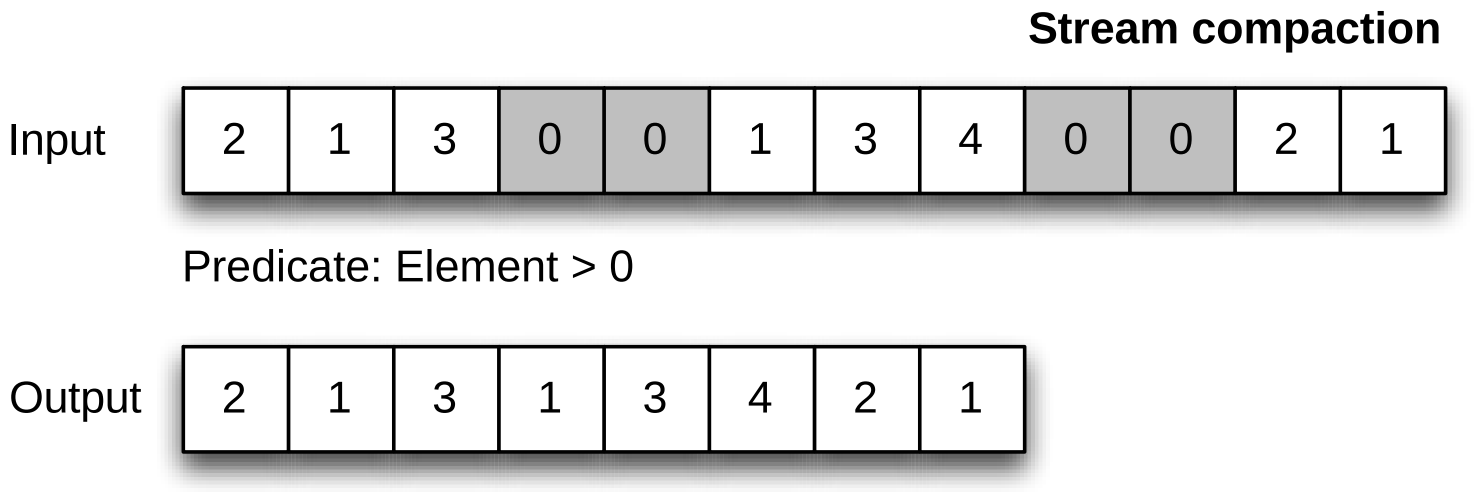
Benefits of Collaboration

- Optimal number of devices is not always max
  - AMD Kaveri (4 CPU cores + 8 GPU CUs)



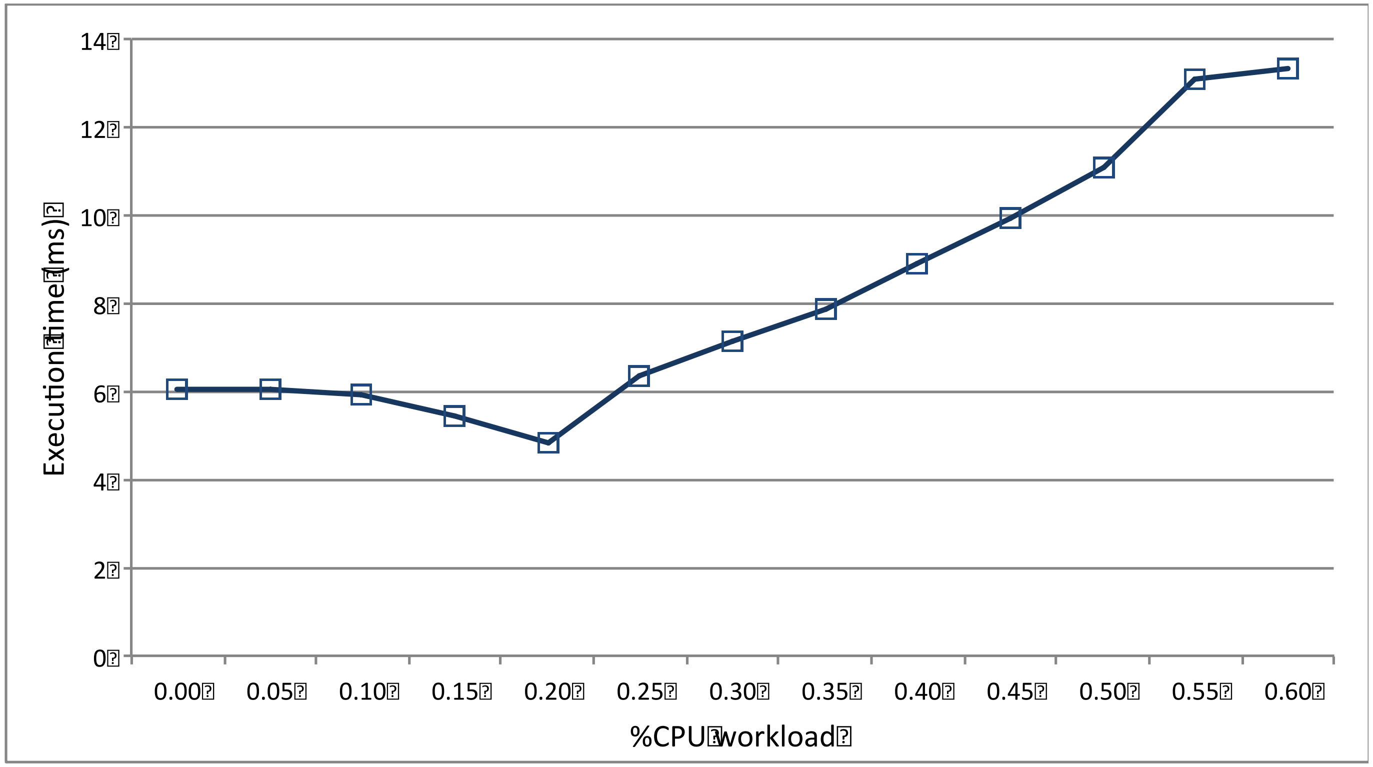
Stream Compaction (I)

- Stream compaction
  - Saving memory storage in sparse data
  - Similar to padding, but local reduction result (non-zero element count) is propagated



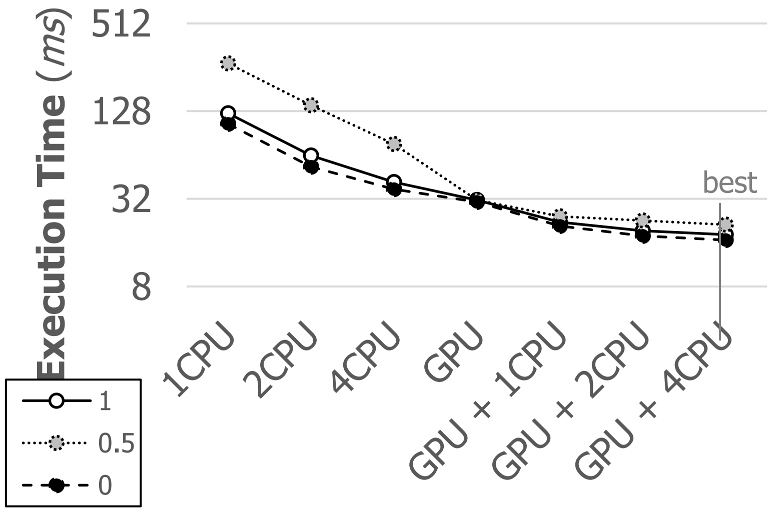
Stream Compaction (II)

- Execution results
  - Array size: 2 MB, Filtered items = 50%
  - NVIDIA Jetson TX1 (4 ARMv8 + 2 SMX): 25% speedup wrt GPU only



Benefits of Collaboration

- Data partitioning improves performance
  - AMD Kaveri (4 CPU cores + 8 GPU CUs)



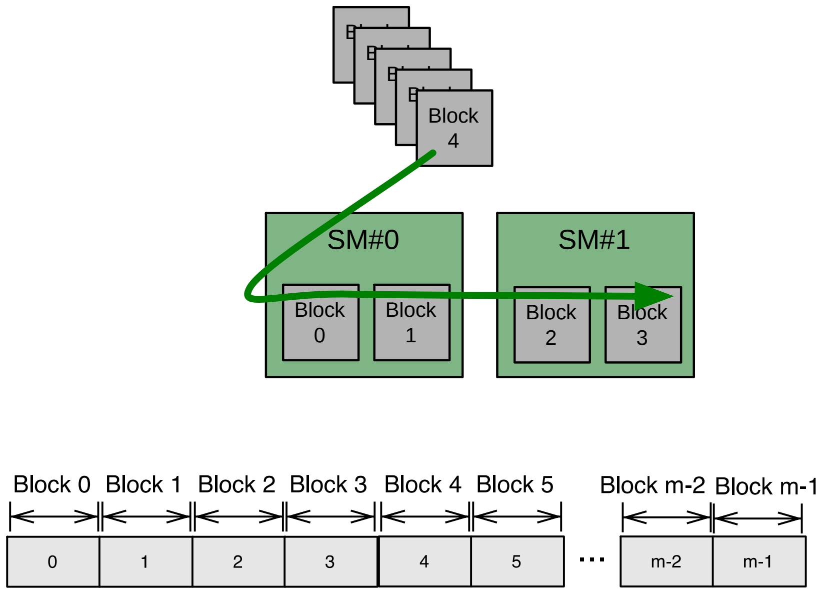
Stream Compaction
(up to 82% improvement over GPU only)

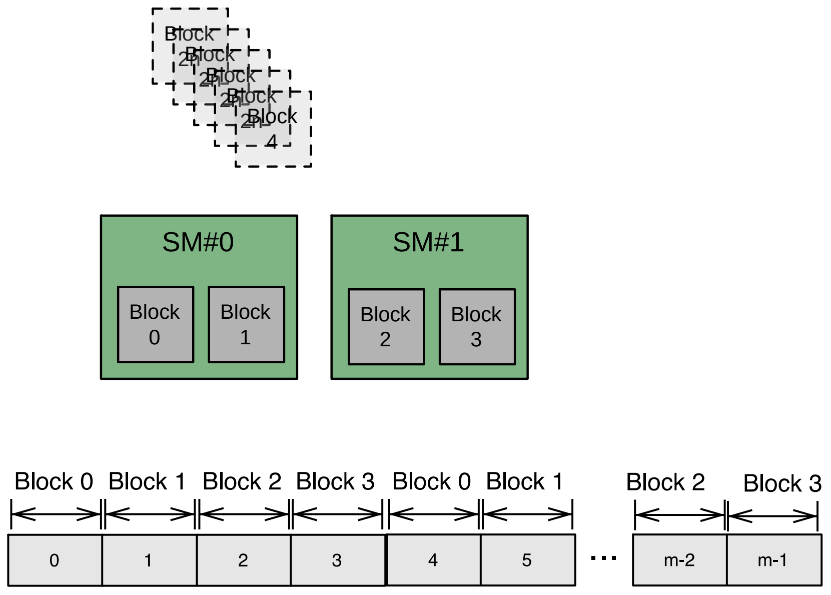
Breadth-First Search

- Small-sized and big-sized frontiers
  - Top-down approach
  - Kernel 1 and Kernel 2
- Atomic-based block synchronization
  - Avoids kernel re-launch
- Very small frontiers
  - Underutilize GPU resources
- Collaborative implementation

Atomic-Based Block Synchronization (I)

- Combine Kernel 1 and Kernel 2
- We can avoid kernel re-launch
- We need to use persistent thread blocks
  - Kernel 2 launches (frontier_size / block_size) blocks
  - Persistent blocks: up to (number_SMs x max_blocks_SM)





Atomic-Based Block Synchronization (II)

- Code (simplified)

```
// GPU kernel
const int gtid = blockIdx.x * blockDim.x + threadIdx.x;

while(frontier_size != 0){

    for(node = gtid; node < frontier_size; node += blockDim.x*gridDim.x){

      // Visit neighbors
      // Enqueue in output queue if needed (global or local queue)

    }

    // Update frontier_size

    // Global synchronization
}
```

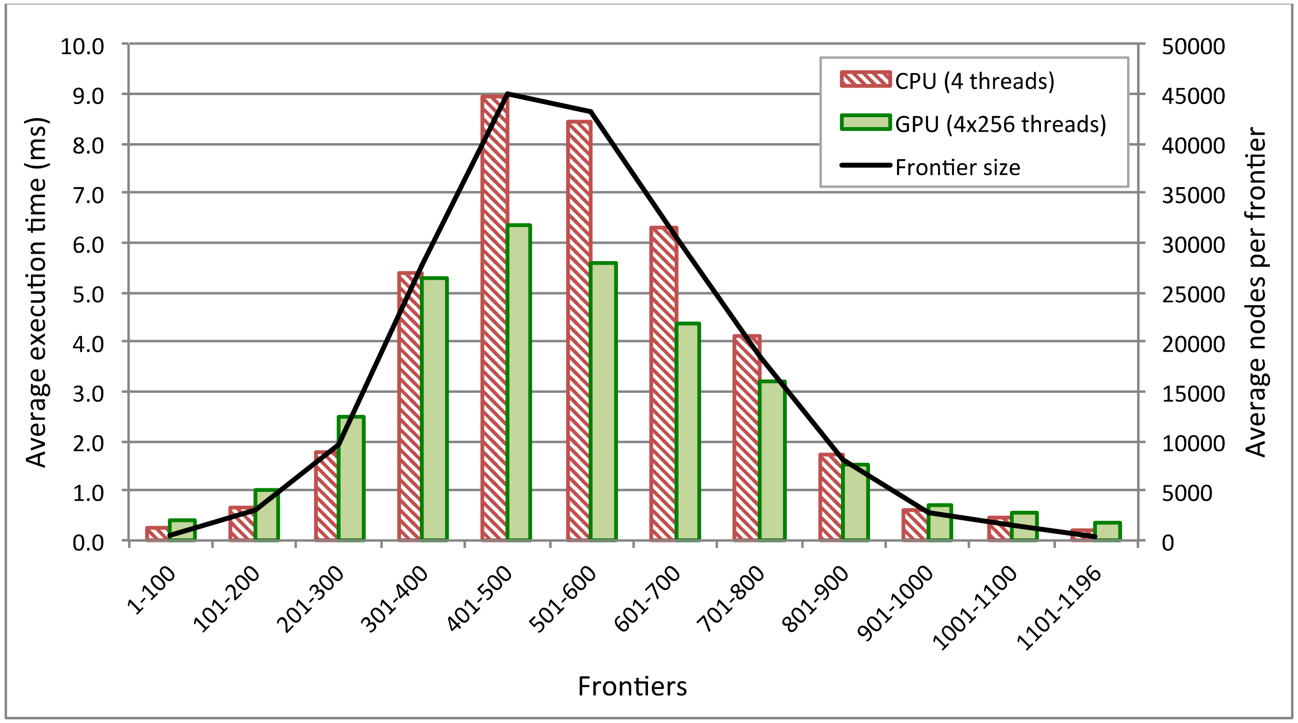
Atomic-Based Block Synchronization (III)

- Global synchronization (simplified)
  - At the end of each iteration

```
const int tid = threadIdx.x;
const int gtid = blockIdx.x * blockDim.x + threadIdx.x;
atomicExch(ptr_threads_run, 0);
atomicExch(ptr_threads_end, 0);
int frontier = 0;
 ...

frontier++;

if(tid == 0){
    atomicAdd(ptr_threads_end, 1);   // Thread block finishes iteration
}

if(gtid == 0){
    while(atomicAdd(ptr_threads_end, 0) != gridDim.x){;}  // Wait until all blocks finish

    atomicExch(ptr_threads_end, 0);   // Reset
    atomicAdd(ptr_threads_run, 1);   // Count iteration
}

if(tid == 0 && gtid != 0){
    while(atomicAdd(ptr_threads_run, 0) < frontier){;}  // Wait until ptr_threads_run is updated
}

__syncthreads();   // Rest of threads wait here

...
```

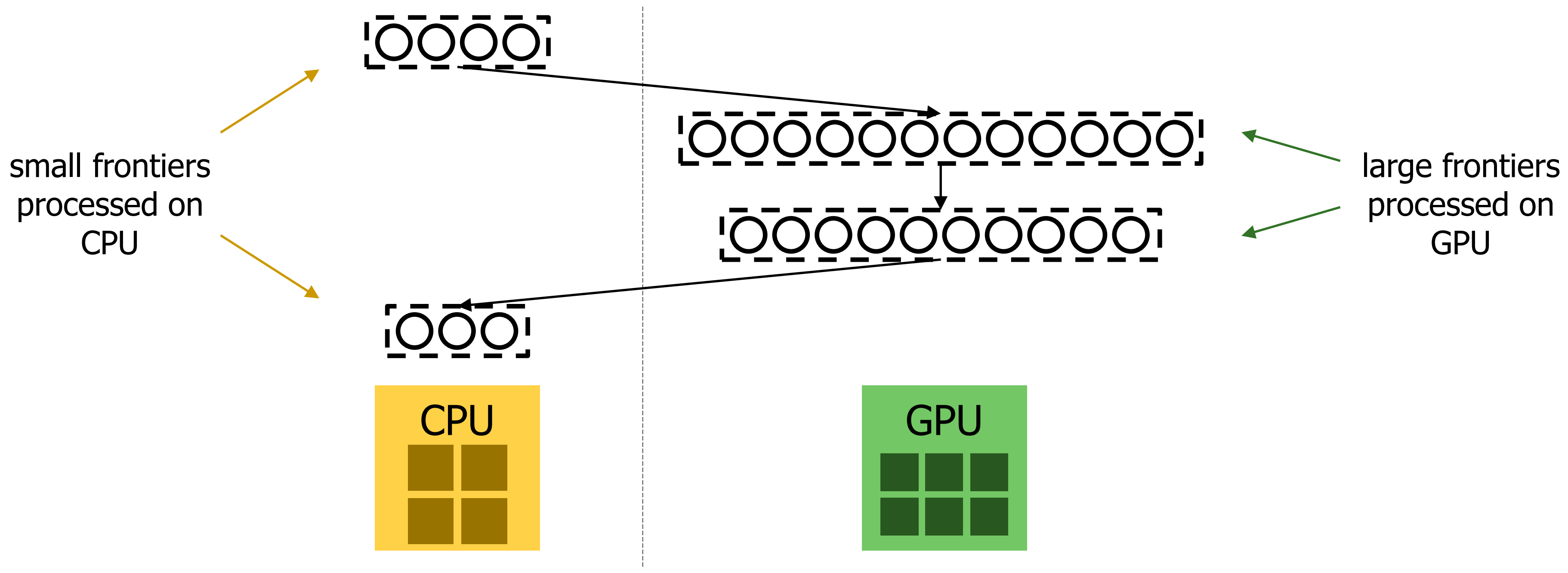
Collaborative Implementation (I)

- Motivation
  - Small-sized frontiers underutilize GPU resources
    - NVIDIA Jetson TX1 (4 ARMv8 CPUs + 2 SMXs)
    - New York City roads



Collaborative Implementation (II)

- Choose the most appropriate device


small frontiers processed on CPU
large frontiers processed on GPU
CPU
GPU

Collaborative Implementation (III)

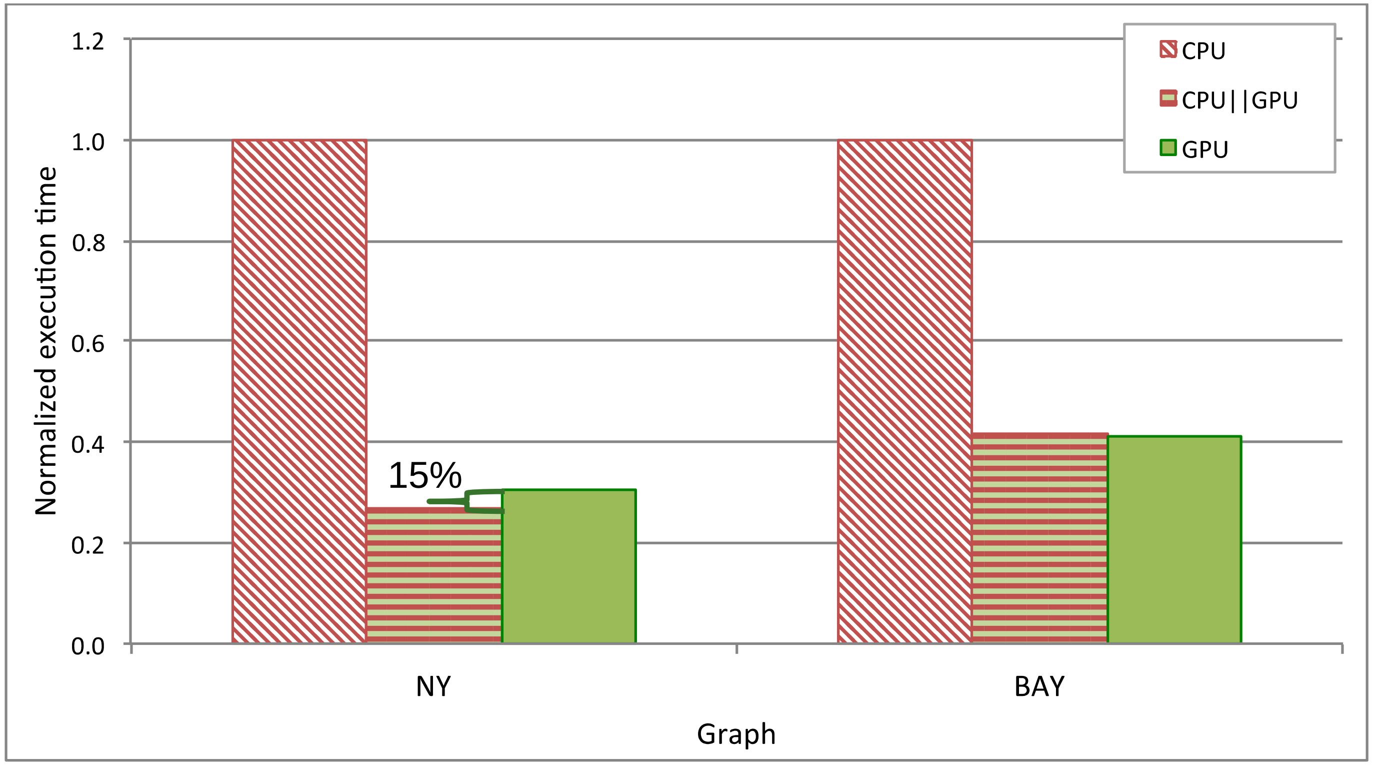
- Choose CPU or GPU depending on frontier size

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads

    }
    else{

        // Launch GPU kernel

    }

}
```

- CPU threads or GPU kernel keep running while the condition is satisfied

Collaborative Implementation (IV)

- Execution results



Collaborative Implementation (V)

- Without Unified Memory
  - Explicit memory copies

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads

    }
    else{

        // Copy from host to device (queues and synchronization variables)

        // Launch GPU kernel

        // Copy from device to host (queues and synchronization variables)

    }

}
```

Collaborative Implementation (VI)

- Unified Memory
  - `cudaMallocManaged();`
  - Easier programming
  - No explicit memory copies

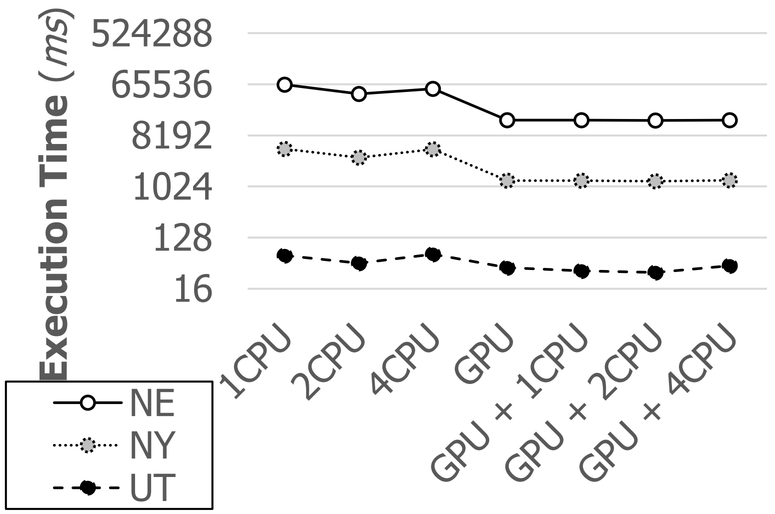
```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads

    }
    else{

        // Launch GPU kernel

        cudaDeviceSynchronize();

    }

}
```

Collaborative Implementation (VII)

- Pascal/Volta Unified Memory
  - CPU/GPU coherence
  - System-wide atomic operations
  - No need to re-launch kernel or CPU threads
  - Possibility of CPU and GPU working on the same frontier

Benefits of Collaboration

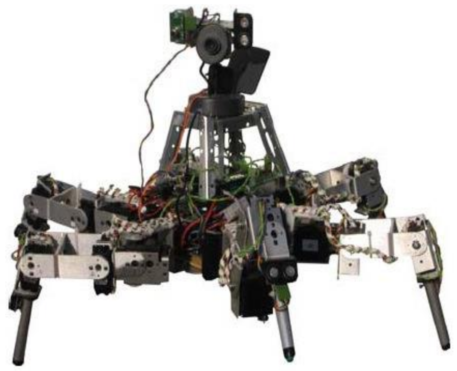
- SSSP performs more computation than BFS



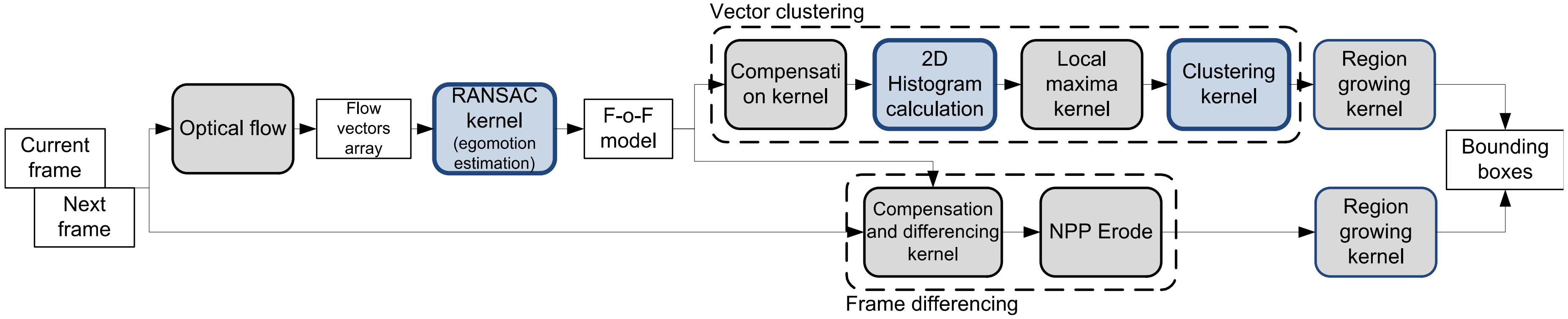
Single Source Shortest Path
(up to 22% improvement over GPU only)

Egomotion Compensation and Moving Objects Detection (I)

- Hexapod robot OSCAR
  - Rescue scenarios
  - Strong egomotion on uneven terrains

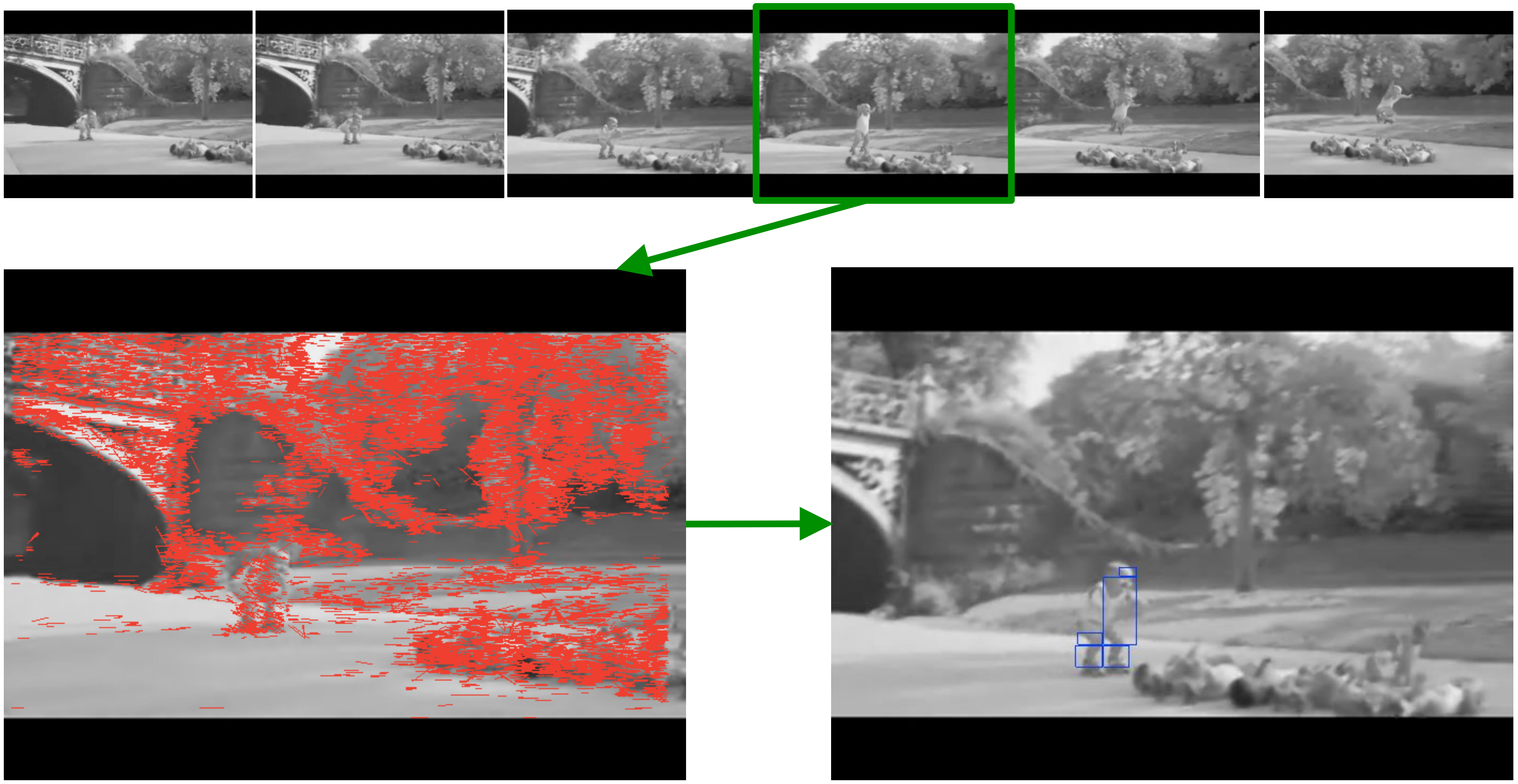


- Algorithm
  - Random Sample Consensus (RANSAC): F-o-F model



Egomotion Compensation and Moving Objects Detection (II)

Fast moving object in strong egomotion scenario detected by vector clustering

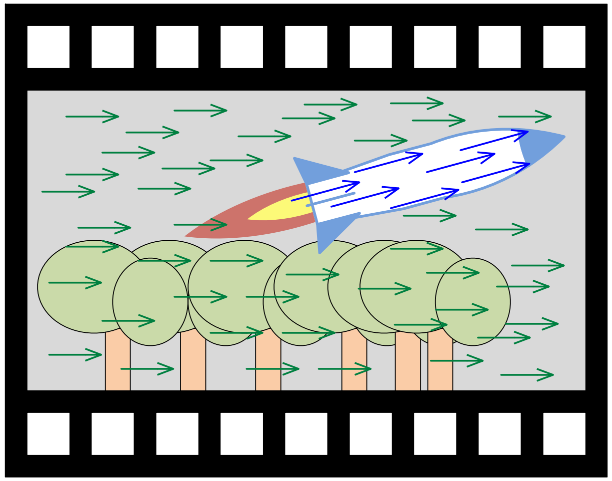


SISD and SIMD phases

- RANSAC (Fischler *et al.* 1981)

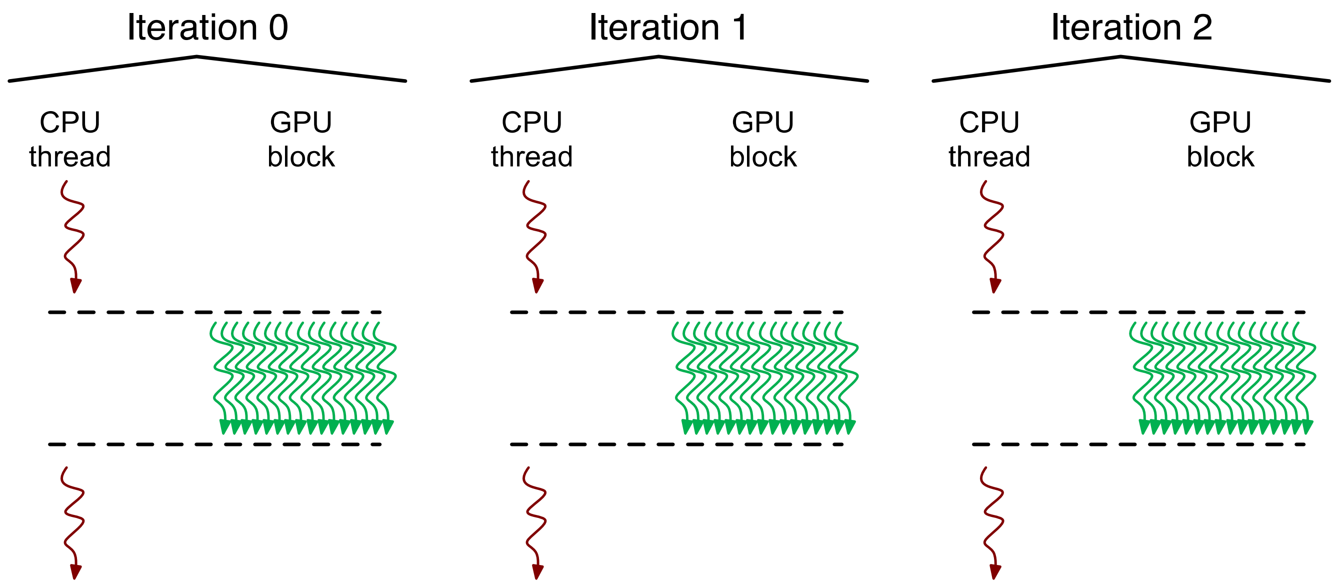
```
While (iteration < MAX_ITER){
     Fitting stage (Compute F-o-F model)          // SISD phase

     Evaluation stage (Count outliers)            // SIMD phase

     Comparison to best model                     // SISD phase

     Check if best model is good enough and iteration >= MIN_ITER     // SISD phase
}
```

  - Fitting stage picks two flow vectors randomly
  - Evaluation generates motion vectors from F-o-F model, and compares them to real flow vectors



Collaborative Implementation

- Randomly picked vectors: Iterations are independent
  - We assign one iteration to one CPU thread and one GPU block



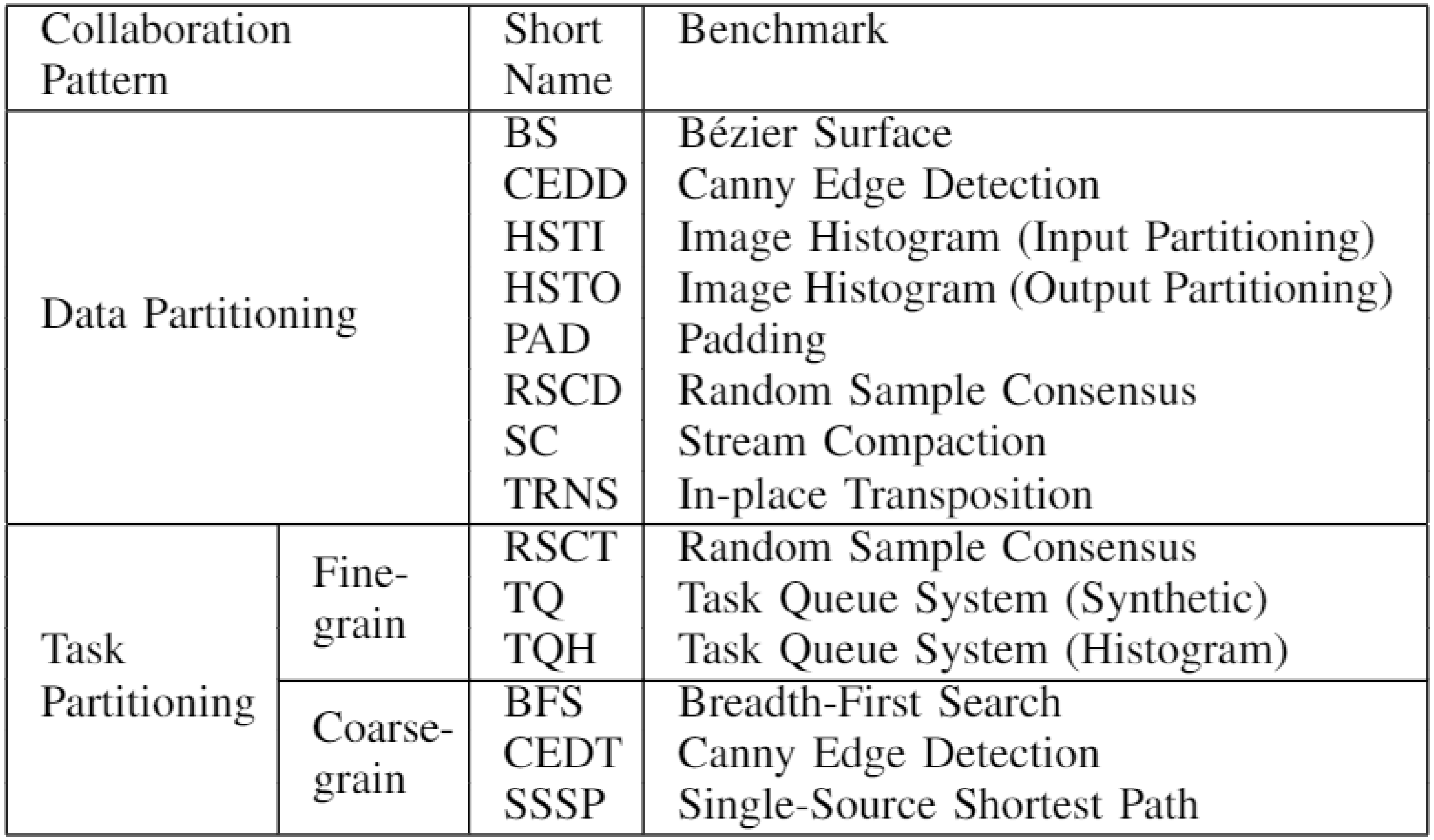
Chai Benchmark Suite (I)

- Collaboration patterns
  - 8 data partitioning benchmarks
  - 3 coarse-grain task partitioning benchmarks
  - 3 fine-grain task partitioning benchmarks

https://chai-benchmarks.github.io

CHAI

Chai Benchmark Suite (II)



Collaboration Pattern		Short Name	Benchmark
Data Partitioning		BS	Bézier Surface
		CEDD	Canny Edge Detection
		HSTI	Image Histogram (Input Partitioning)
		HSTO	Image Histogram (Output Partitioning)
		PAD	Padding
		RSCD	Random Sample Consensus
		SC	Stream Compaction
		TRNS	In-place Transposition
Task Partitioning	Fine-grain	RSCT	Random Sample Consensus
		TQ	Task Queue System (Synthetic)
		TQH	Task Queue System (Histogram)
	Coarse-grain	BFS	Breadth-First Search
		CEDT	Canny Edge Detection
		SSSP	Single-Source Shortest Path

Computer Architecture
Lecture 26: GPU Programming

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

06 January 2023