Lecture 5 & 6:

GPU Architecture & CUDA Programming

Parallel Computer Architecture and Programming

CMU 15-418/15-618, Fall 2025

When It All Began, There Were Games

3D Rendering

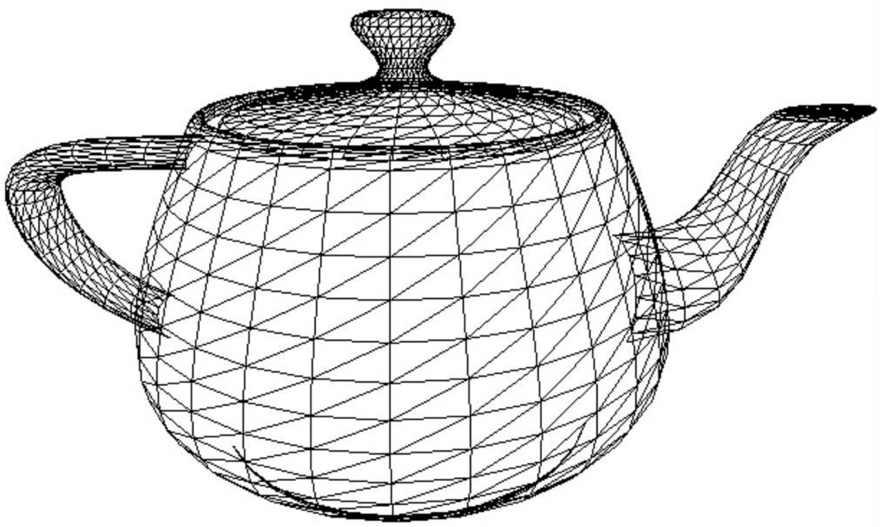


Image credit: Henrik Wann Jensen

Input: description of a scene:

3D surface geometry (e.g., triangle mesh)
surface materials, lights, camera, etc.

Output: image of the scene

Simple definition of rendering task: computing how each triangle in 3D mesh contributes to appearance of each pixel in the image?

3D Graphics Today

Early graphics programming (OpenGL API)

- Graphics programming APIs provided programmer mechanisms to set parameters of scene lights and materials

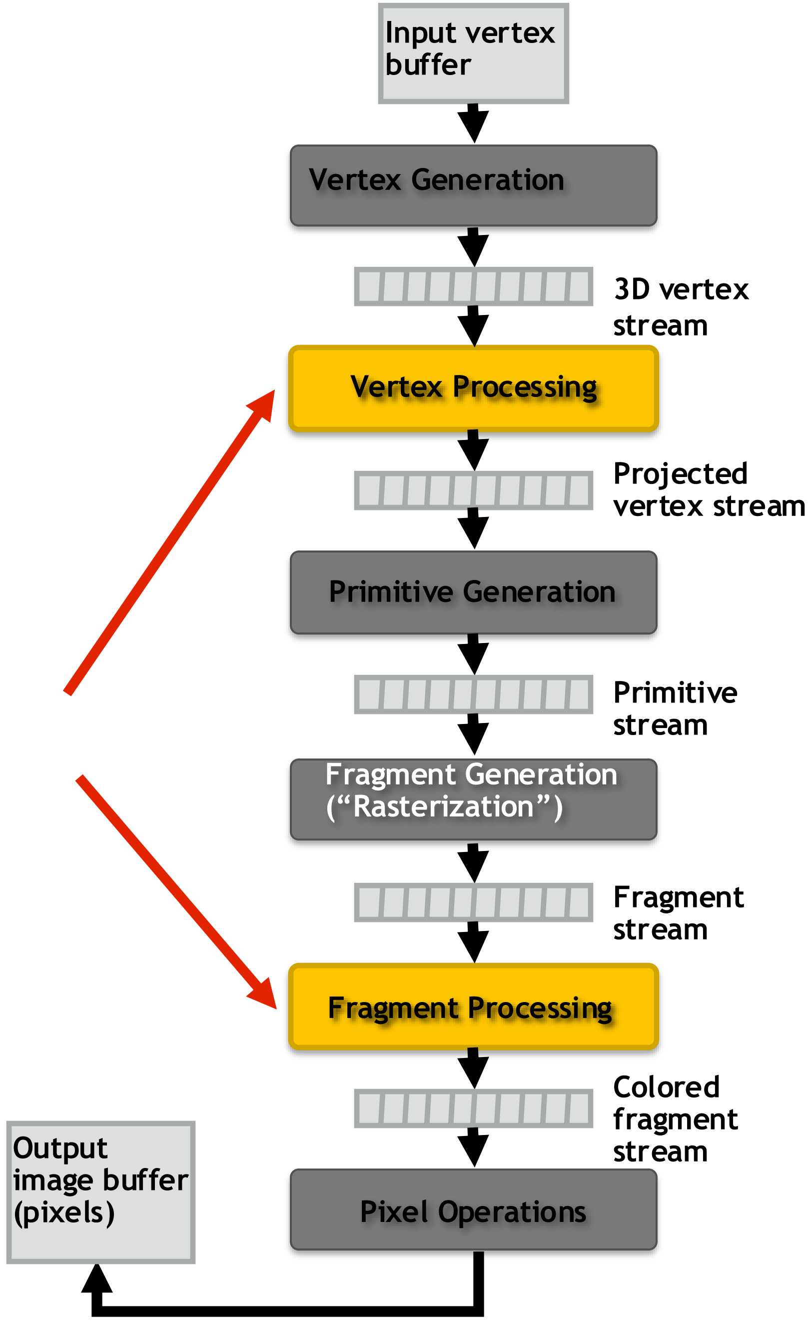
  - `glLight(light_id, parameter_id, parameter_value)`
    - Examples of light parameters: color, position, direction

  - `glMaterial(face, parameter_id, parameter_value)`
    - Examples of material parameters: color, shininess

Graphics shading languages

- Allow application to extend the functionality of the graphics pipeline by specifying materials and lights programmatically!
  - Support diversity in materials
  - Support diversity in lighting conditions

- Programmer provides mini-programs ("shaders") that define pipeline logic for certain stages
  - Pipeline maps shader function onto all elements of input stream



Example fragment shader program *

Run once per fragment (per pixel covered by a triangle)

OpenGL shading language (GLSL) shader program: defines behavior of fragment processing stage

```
uniform sampler2D myTexture;
uniform float3 lightDir;
varying vec3 norm;
varying vec2 uv;

void myFragmentShader()
{
  vec3 kd = texture2D(myTexture, uv);
  kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
  return vec4(kd, 1.0);
}
```

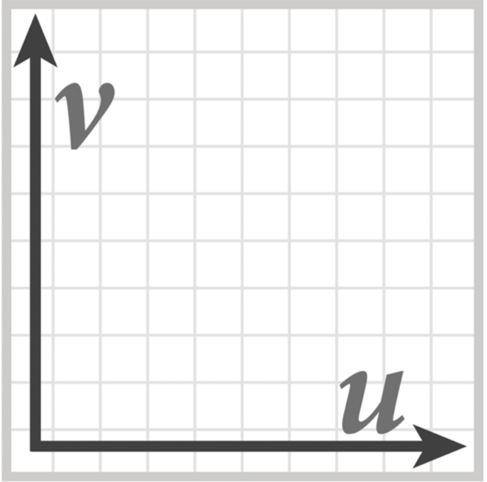
read-only global variables (uniform sampler2D myTexture; uniform float3 lightDir;)

per-fragment inputs (varying vec3 norm; varying vec2 uv;)

"fragment shader" (a.k.a kernel function mapped onto input fragment stream)

per-fragment output: RGBA surface color at pixel

`myTexture` is a texture map



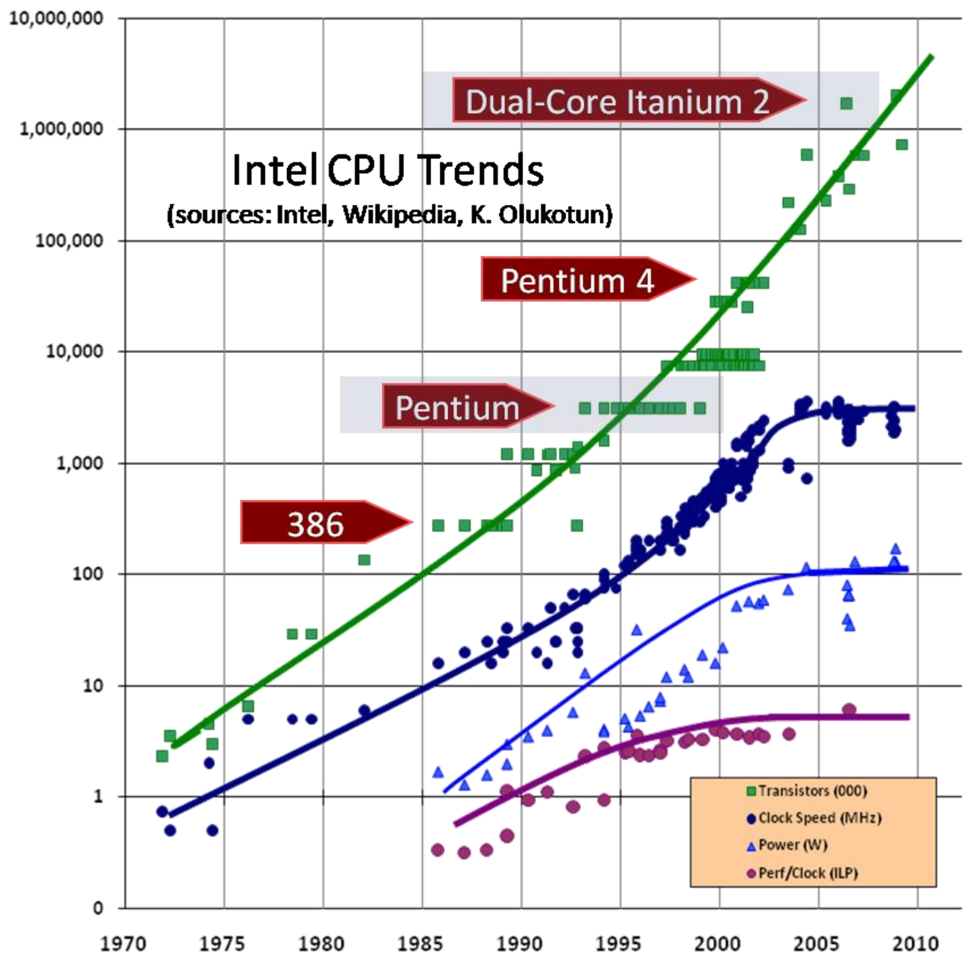
* Syntax/details of this code not important to 15-418.
What is important is that it's a kernel function operating on a stream of inputs.


ROBO
RECALL
UNREAL
FORTNITE

Observation circa 2001-2003

GPUs are very fast processors for performing the same computation (shader programs) on large collections of data (streams of vertices, fragments, and pixels)

Wait a minute! That sounds a lot like data-parallelism to me!



Hack! early GPU-based scientific computation

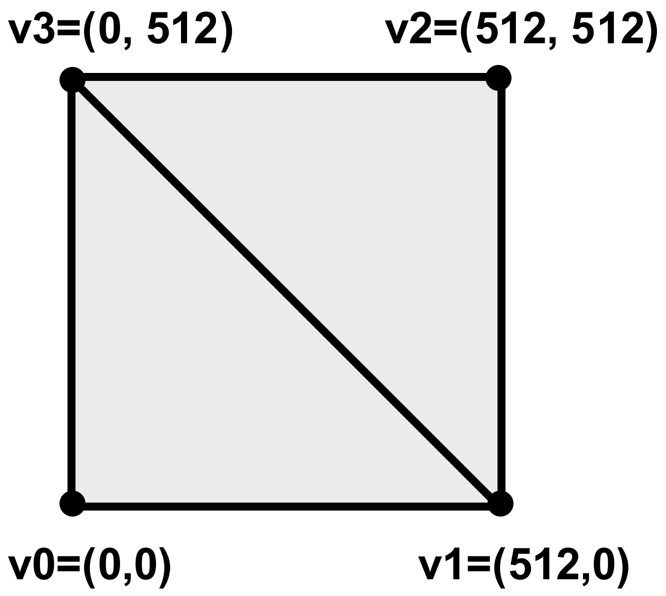
Set OpenGL output image size to be output array size (e.g., 512 x 512)

Render 2 triangles that exactly cover screen
(one shader computation per pixel = one shader computation output element)

We now can use the GPU like a data-parallel programming system.

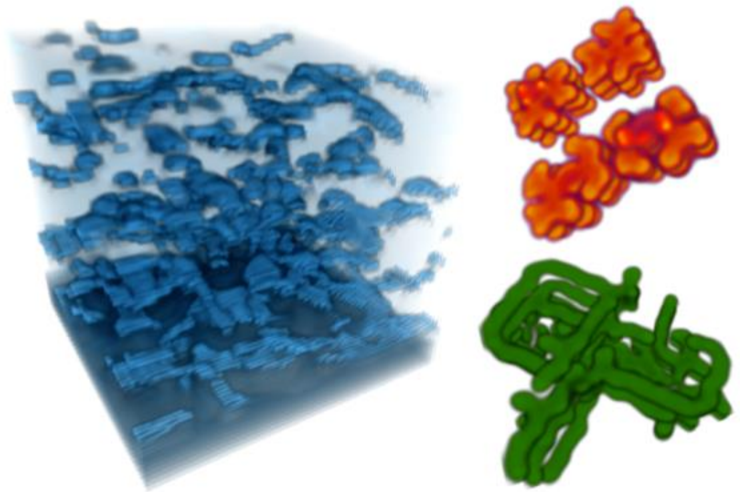
Fragment shader function is mapped over 512 x 512 element collection.

Hack!

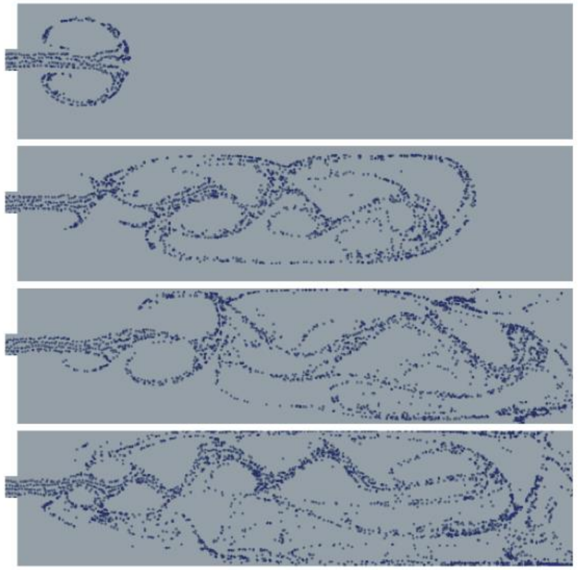


“GPGPU” 2002-2003

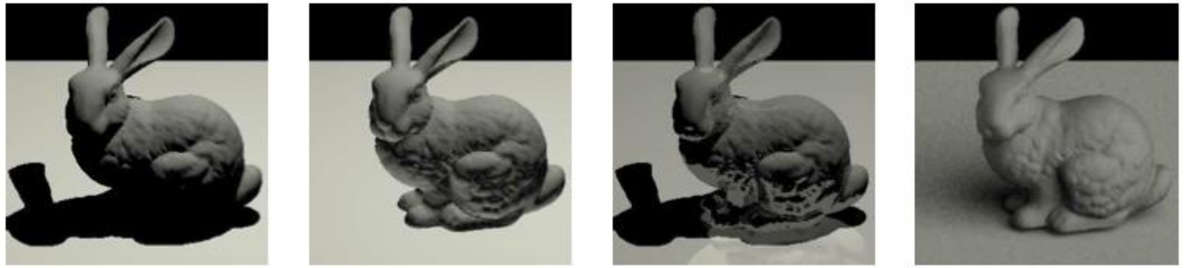
GPGPU = “general purpose” computation on GPUs



Coupled Map Lattice Simulation [Harris 02]

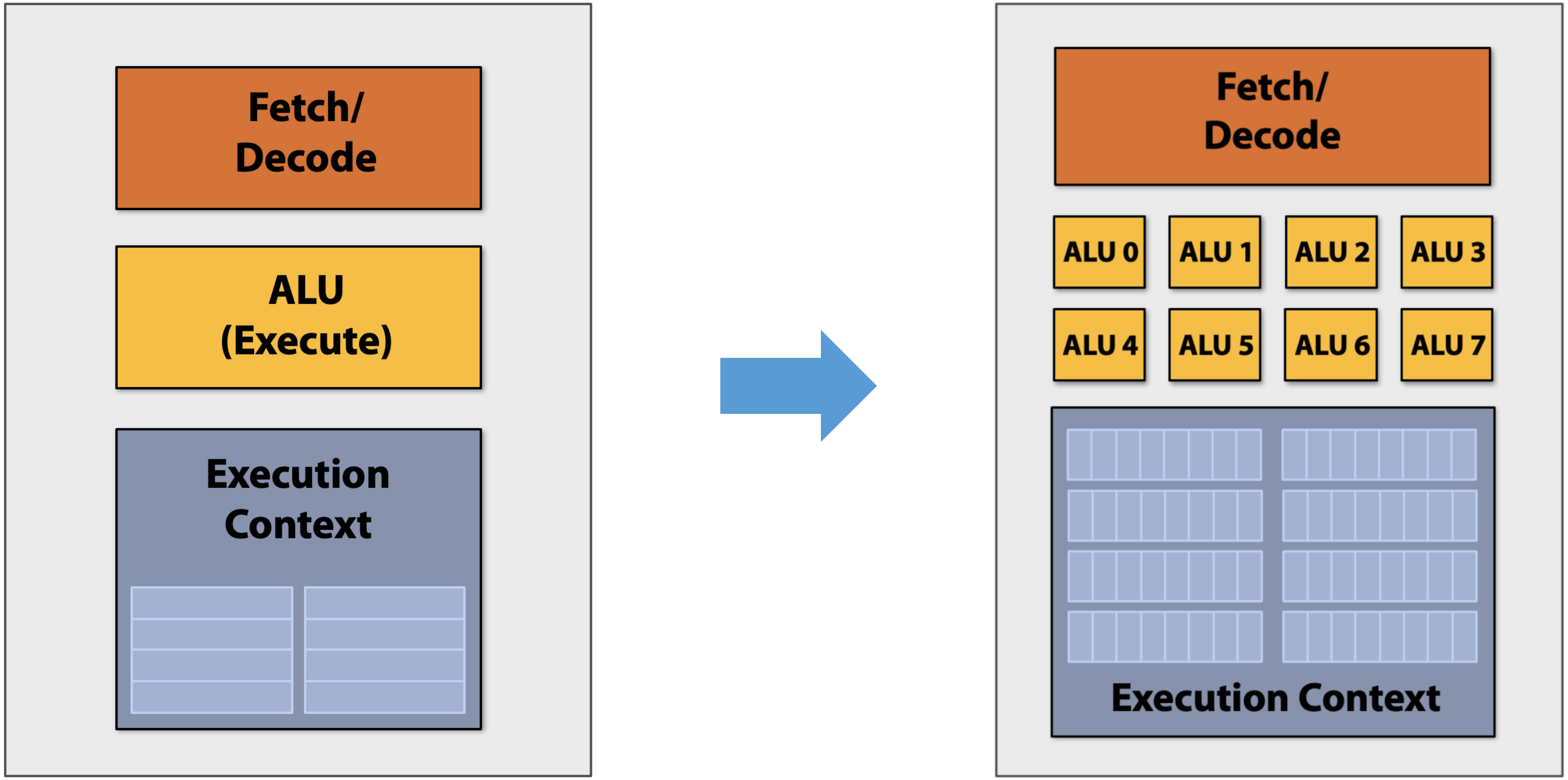


Sparse Matrix Solvers [Bolz 03]



Ray Tracing on Programmable Graphics Hardware [Purcell 02]

Single Instruction, Multiple Data



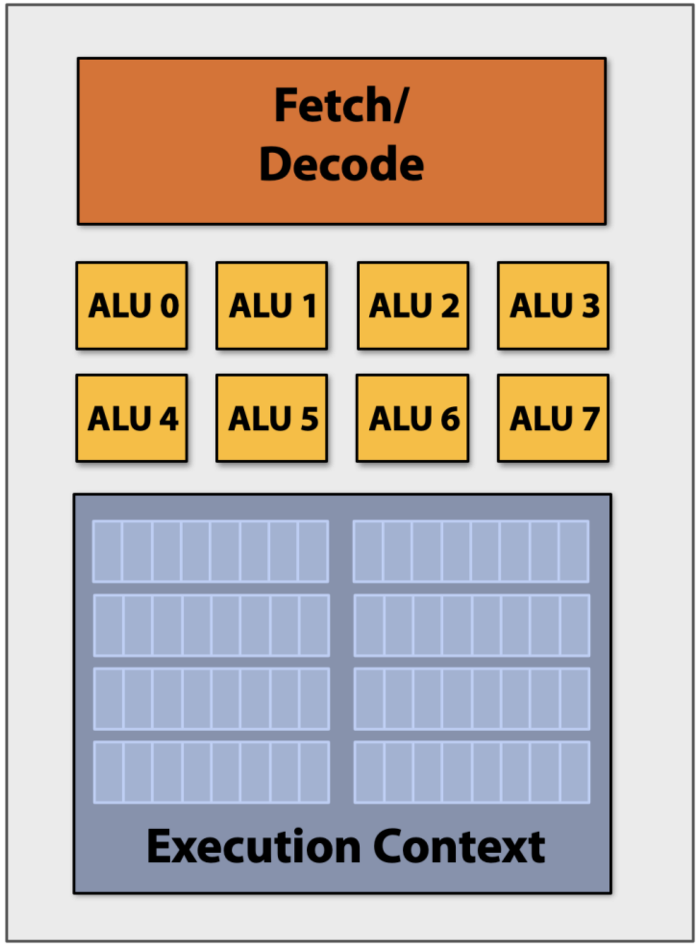
Conventional single instruction, single data processor

Modern single instruction, multiple data processor

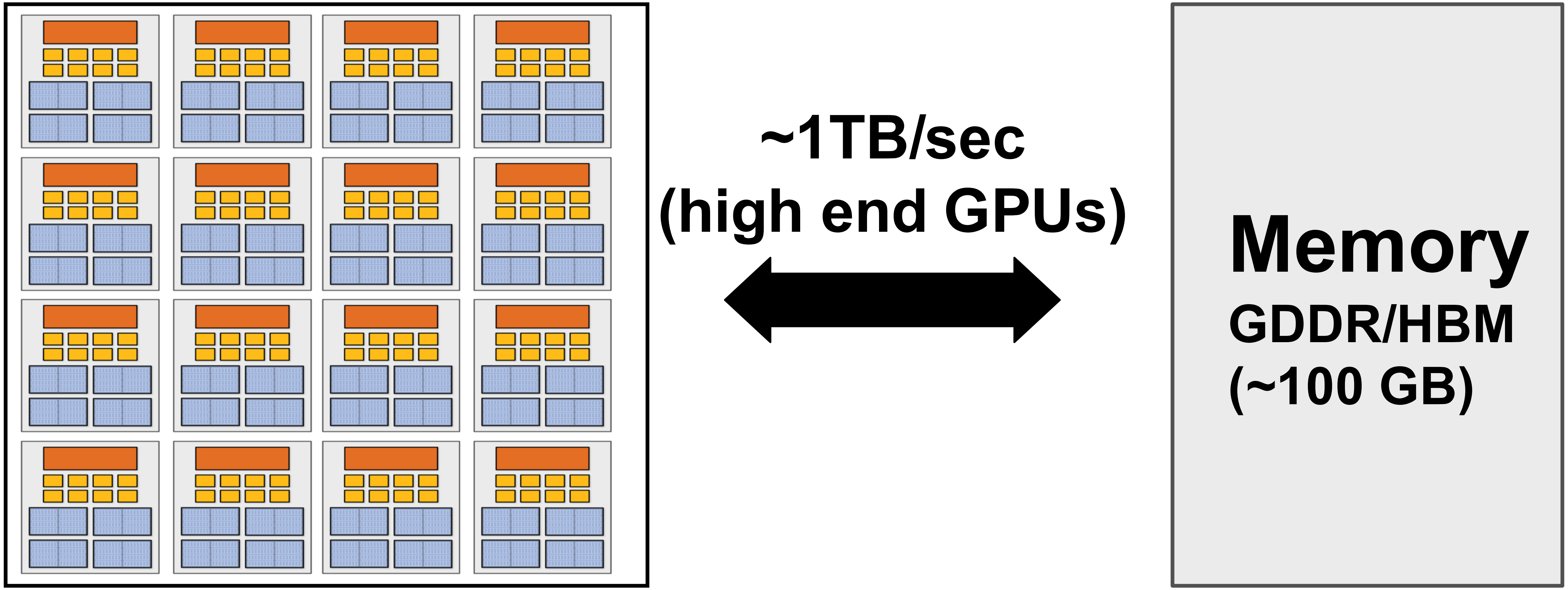
Single Instruction, Multiple Data

- Same instruction broadcast and executed in parallel on all ALUs

- Add ALUs to increase compute capability

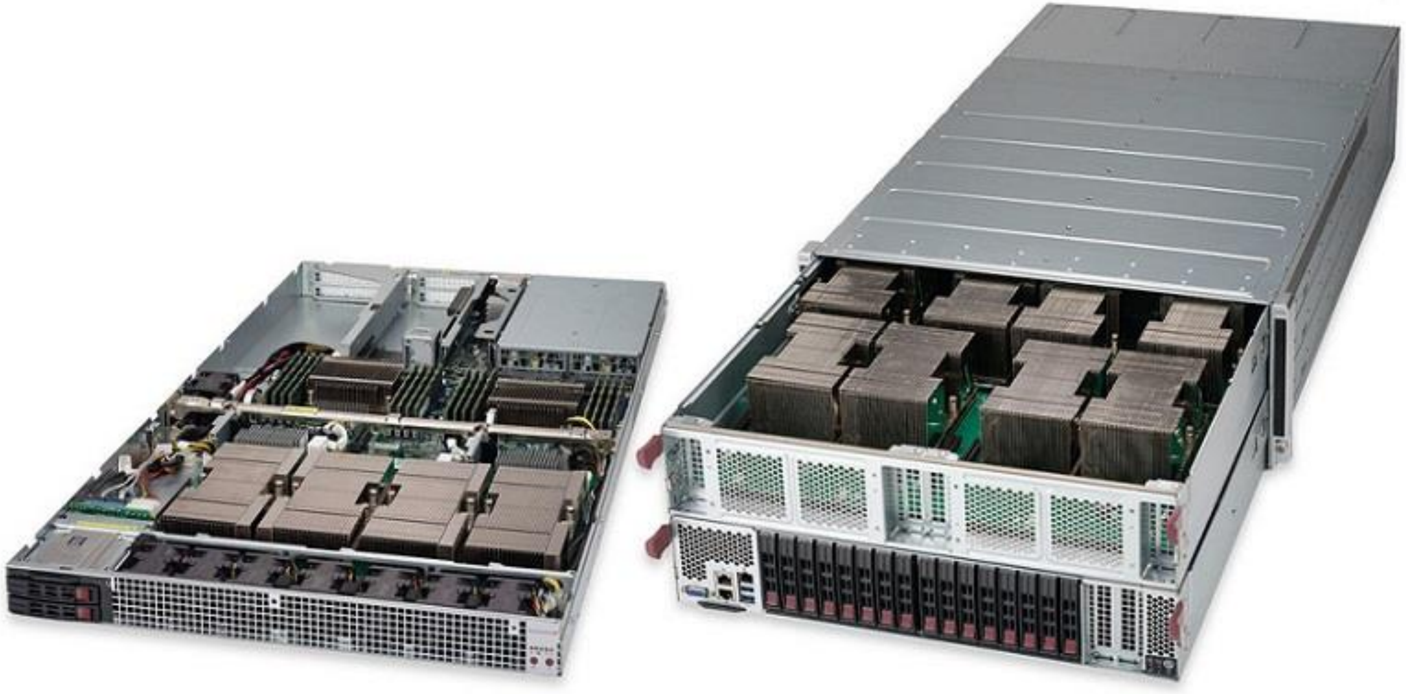


Basic GPU Architecture



- **Multi-core chip**
- **SIMD execution within a single core (many execution units performing the same instruction)**
- **Multi-threaded execution on a single core (multiple threads executed concurrently by a core)**

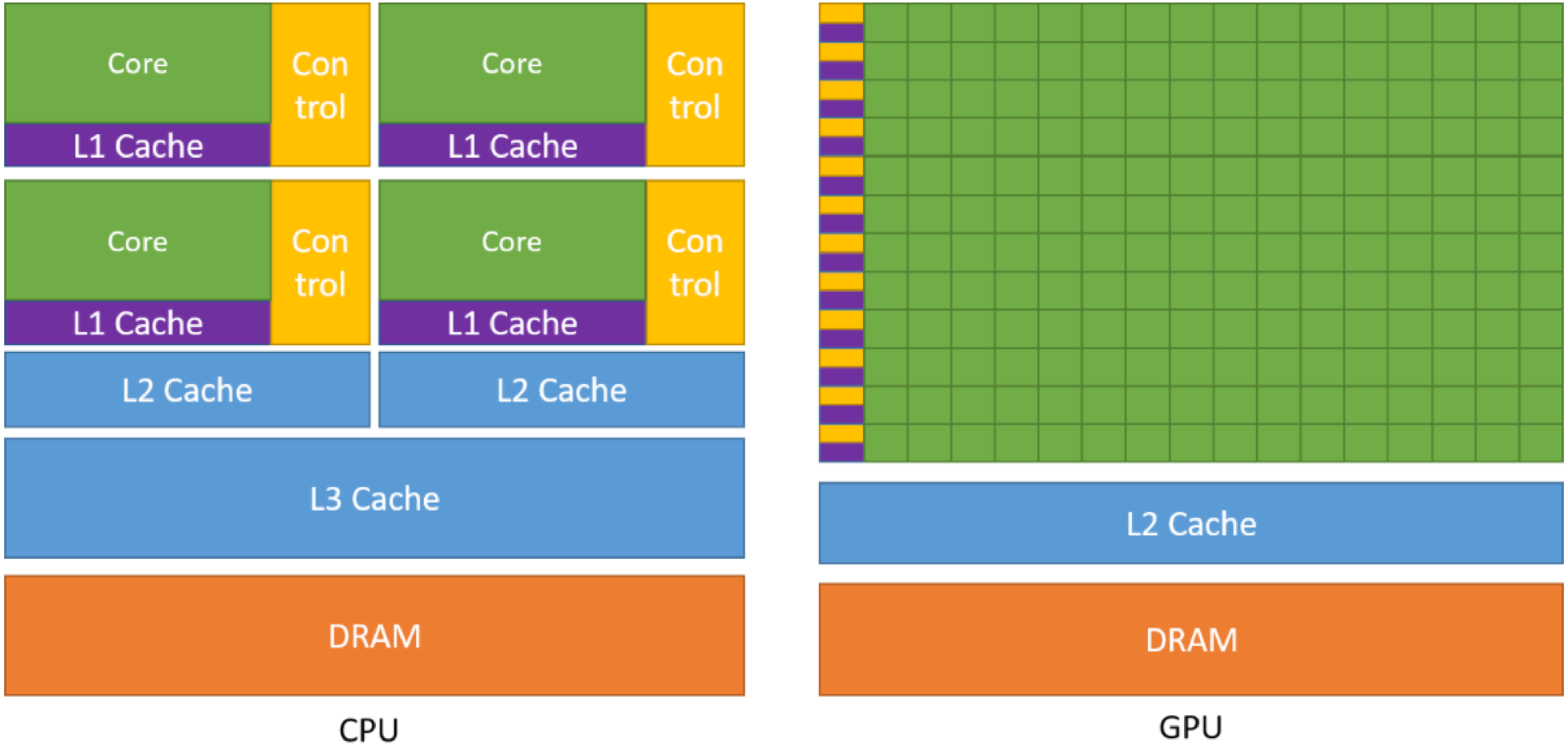
CPU-GPU Setup



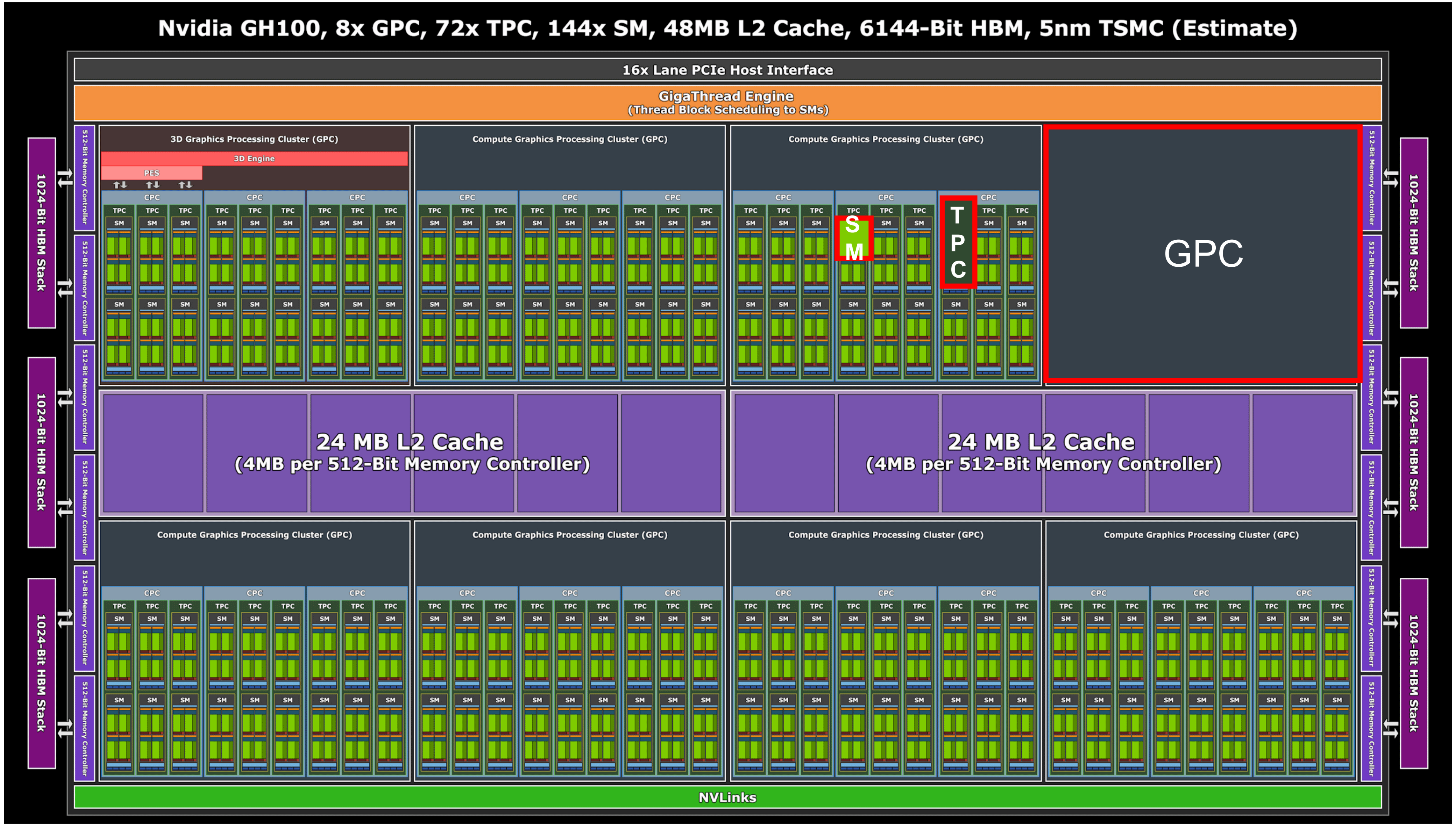
1U/4GPU Server

4U/8GPU Server

Massive Parallel Computing Units



H100 Architecture

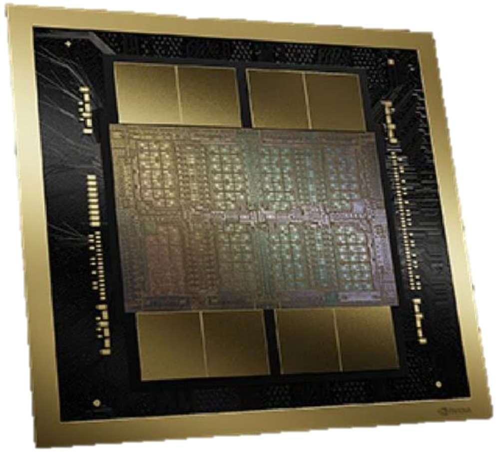


SM Architecture

- Execution units
  - 192 simple FUs (int and single-precision FP)
  - 64 double-precision FUs
  - 32 load-store FUs
  - 32 special-function FUs (e.g., sqrt, sin, cos, …)
- Memory structures
  - 64K 32-bit registers
  - 64KB data memory, split between shared memory (scratchpad) and L1
  - 48KB read-only data/texture cache

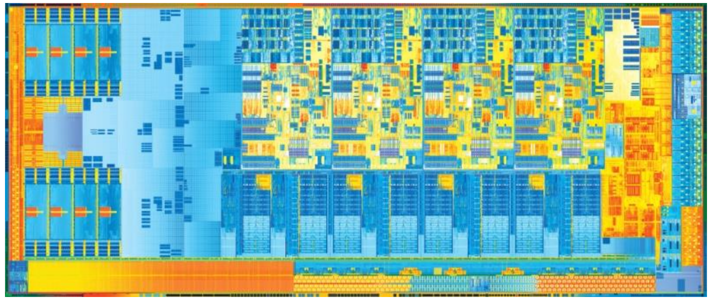
GPU Architectures

- Discrete GPUs
  - PCIe-based data-parallel processor
  - Separate GPU memory

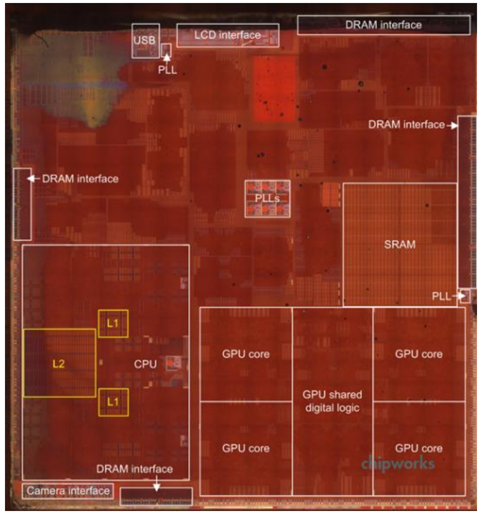


Nvidia Blackwell 5nm, 2x 750mm^2

- Integrated GPUs
  - CPU and GPU on same die
  - Shared main memory and last-level cache



Intel Ivy Bridge, 22nm 160mm^2



Apple A7, 28nm TSMC, 102mm^2

- Pros/cons?

Outline

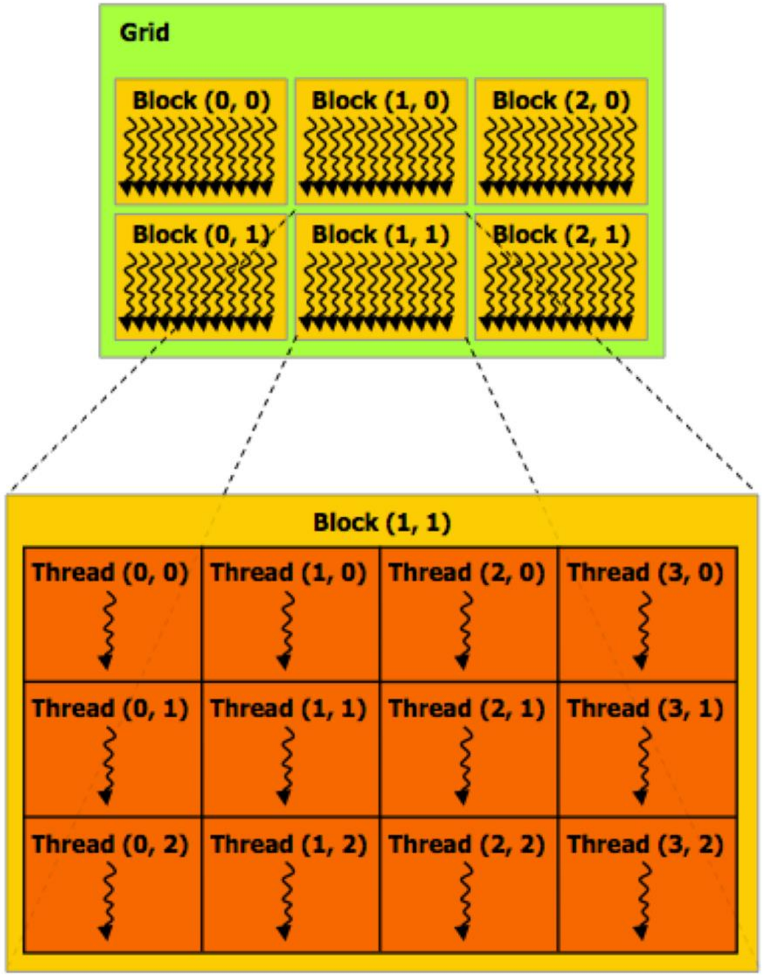
- CUDA Programming Abstractions
- GPU Architectures
- Cast Study 1: Matrix Multiplication in CUDA
- Cast Study 2: Parallel Reduction in CUDA

CUDA Programming Language

- Introduced in 2007 with NVIDIA Tesla architecture
- ”C-like” languages for programming on GPUs
- CUDA’s abstractions closely match the capabilities/performance characteristics of modern GPUs
- Design goal: maintain low abstraction distance

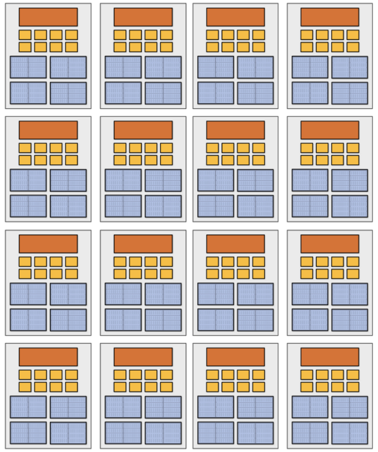
CUDA Programs Consist of a Hierarchy of Threads

- Thread IDs are up to 3-dimensional (2D example below)

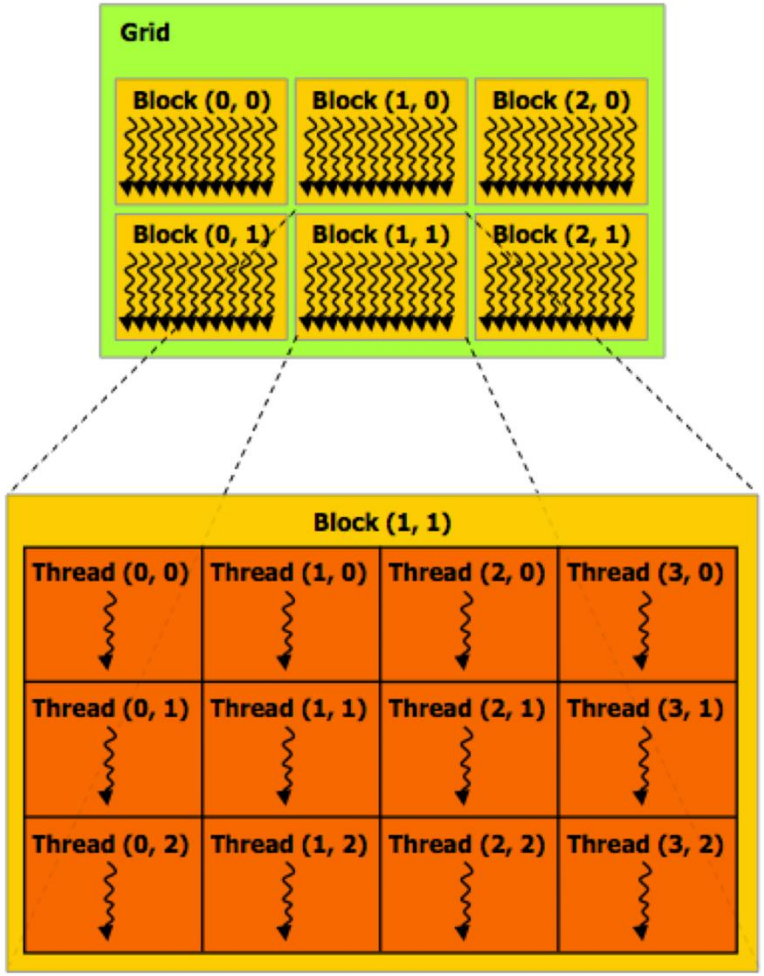


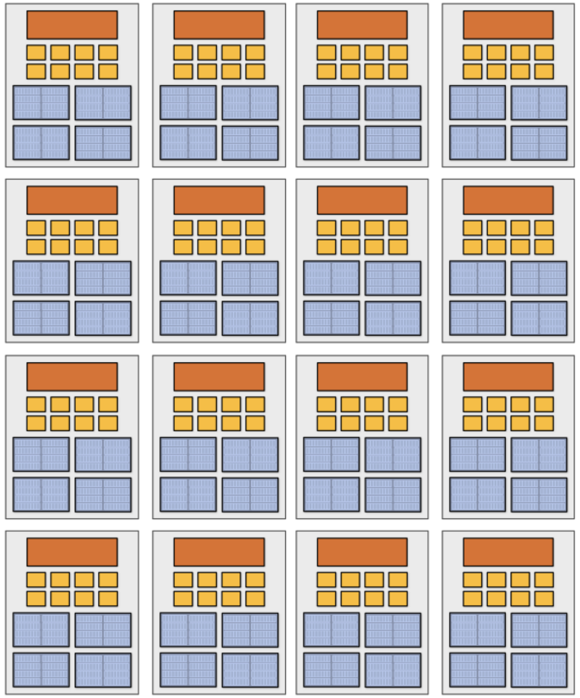

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Run on CPU



CUDA Blocks Map to GPU Cores (Streaming Multiprocessors)



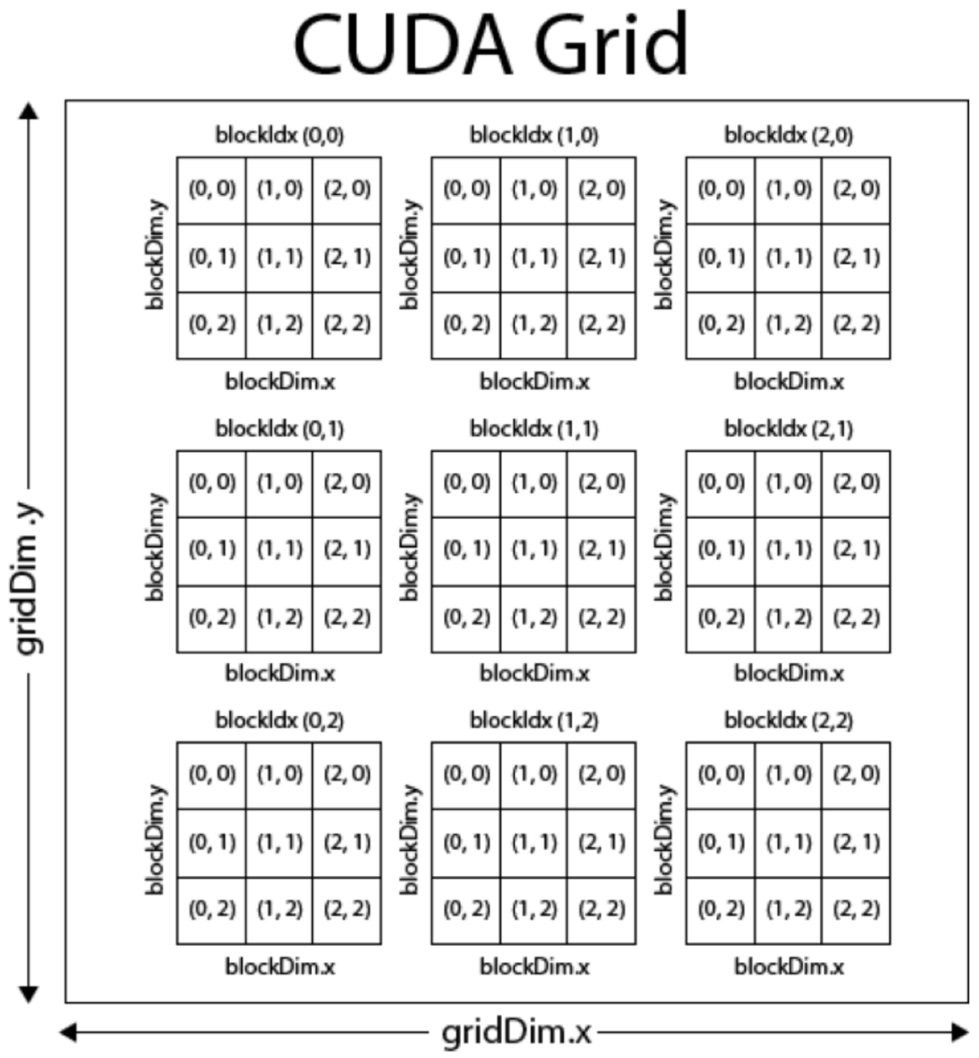


GPU

Grid, Block, and Thread

- gridDim: The dimensions of the grid
- blockIdx: The block index within the grid
- blockDim: The dimensions of a block
- threadIdx: The thread index within a block

We always have: (default kernel<<<1,1>>>)
gridDim = (1, 1, 1)
blockIdx = (1, 1, 1)
threadIdx = (1, 1, 1)



Basic CUDA syntax

Serial execution: running as part of normal C/C++ application on CPU

Host

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Bulk launch of many CUDA threads
"launch a grid of CUDA thread blocks"
Call returns when all threads have terminated

Device

```
// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  C[j][i] = A[j][i] + B[j][i];
}
```

__global__ denotes a CUDA kernel function runs on GPU

Each thread computes its overall grid thread id from its position in its block (threadIdx) and its block's position in the grid (blockIdx)

CUDA kernel: executed in parallel on multiple CUDA cores

Clear Separation of Host and Device Code

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
          Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

```
__device__ float doubleValue(float x)
{
  return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(float A[Ny][Nx],
                        float B[Ny][Nx],
                        float C[Ny][Nx])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  C[j][i] = A[j][i] + doubleValue(B[j][i]);
}
```

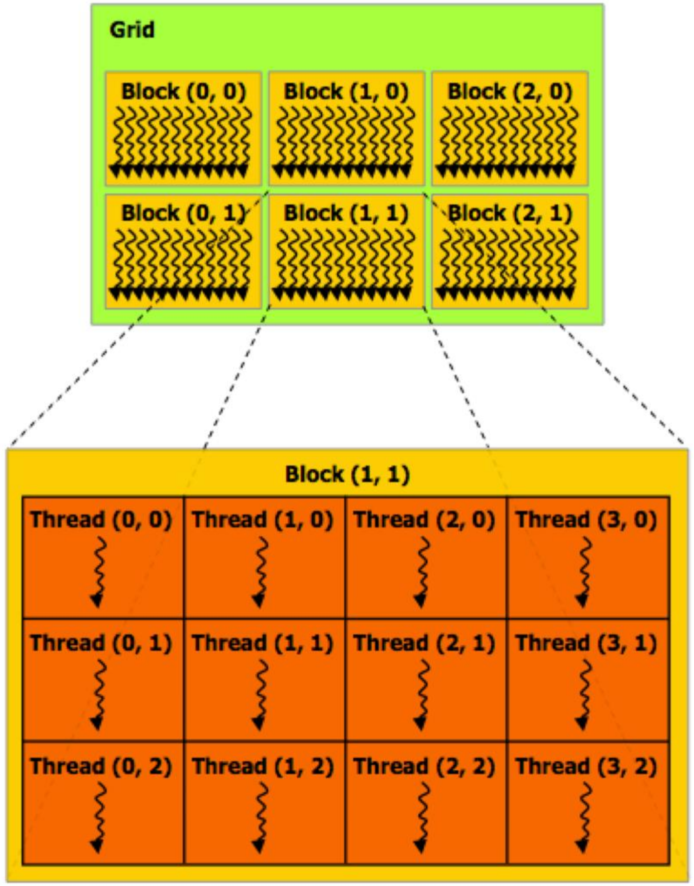
Separation of execution into host and device code is performed statically by the programmer

“Host” code : serial execution on CPU

“Device” code (SIMD execution on GPU)

Number of SIMD Threads is Explicit in Program

Number of kernel invocations is not determined by size of data collection

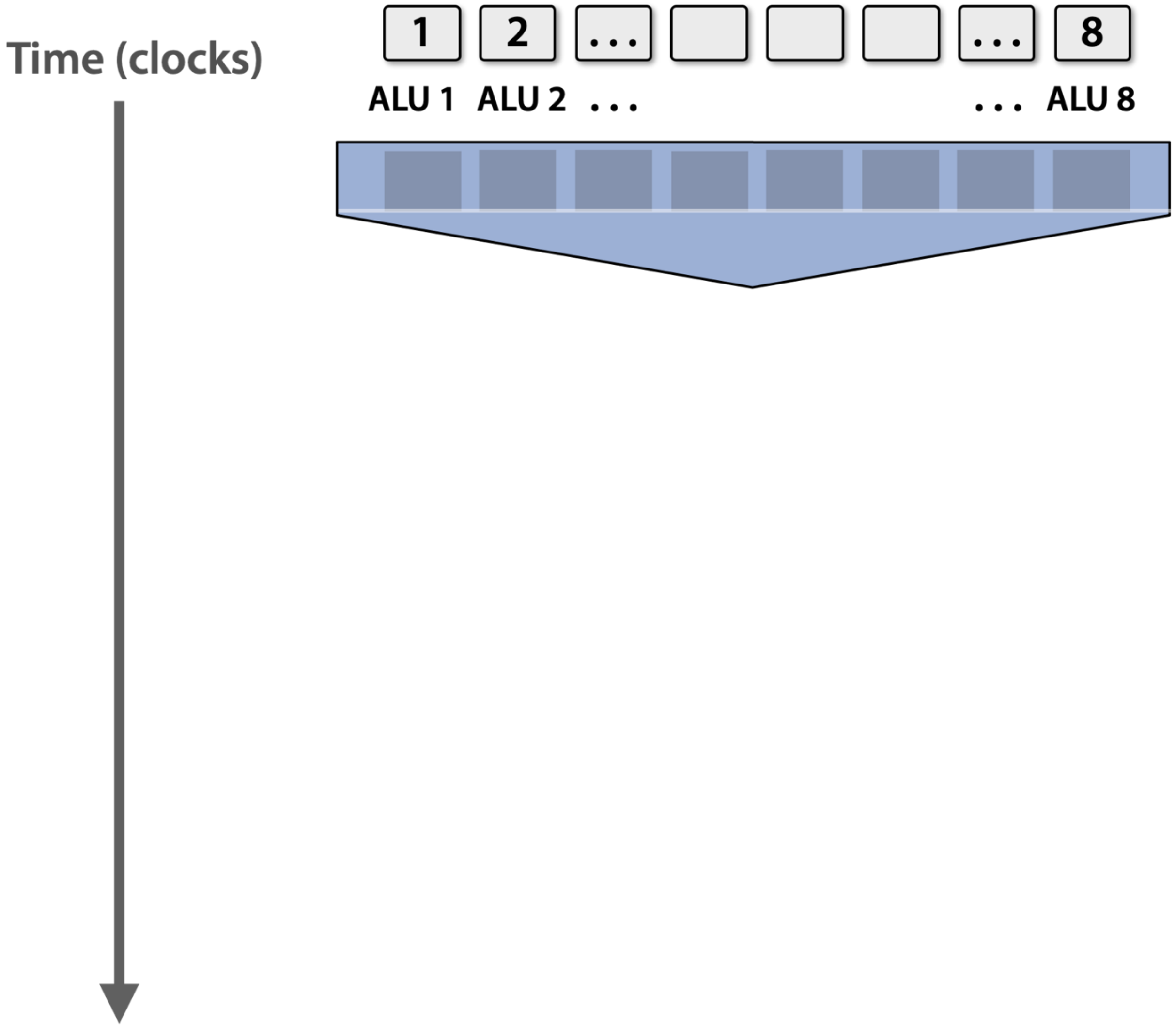



```
const int Nx = 11; // not a multiple of threadsPerBlk.x
const int Ny = 5; // not a multiple of threadsPerBlk.y

dim3 threadsPerBlk(4, 3, 1);
dim3 numBlocks(3, 2, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlk>>>(A, B, C);
```

Guard access

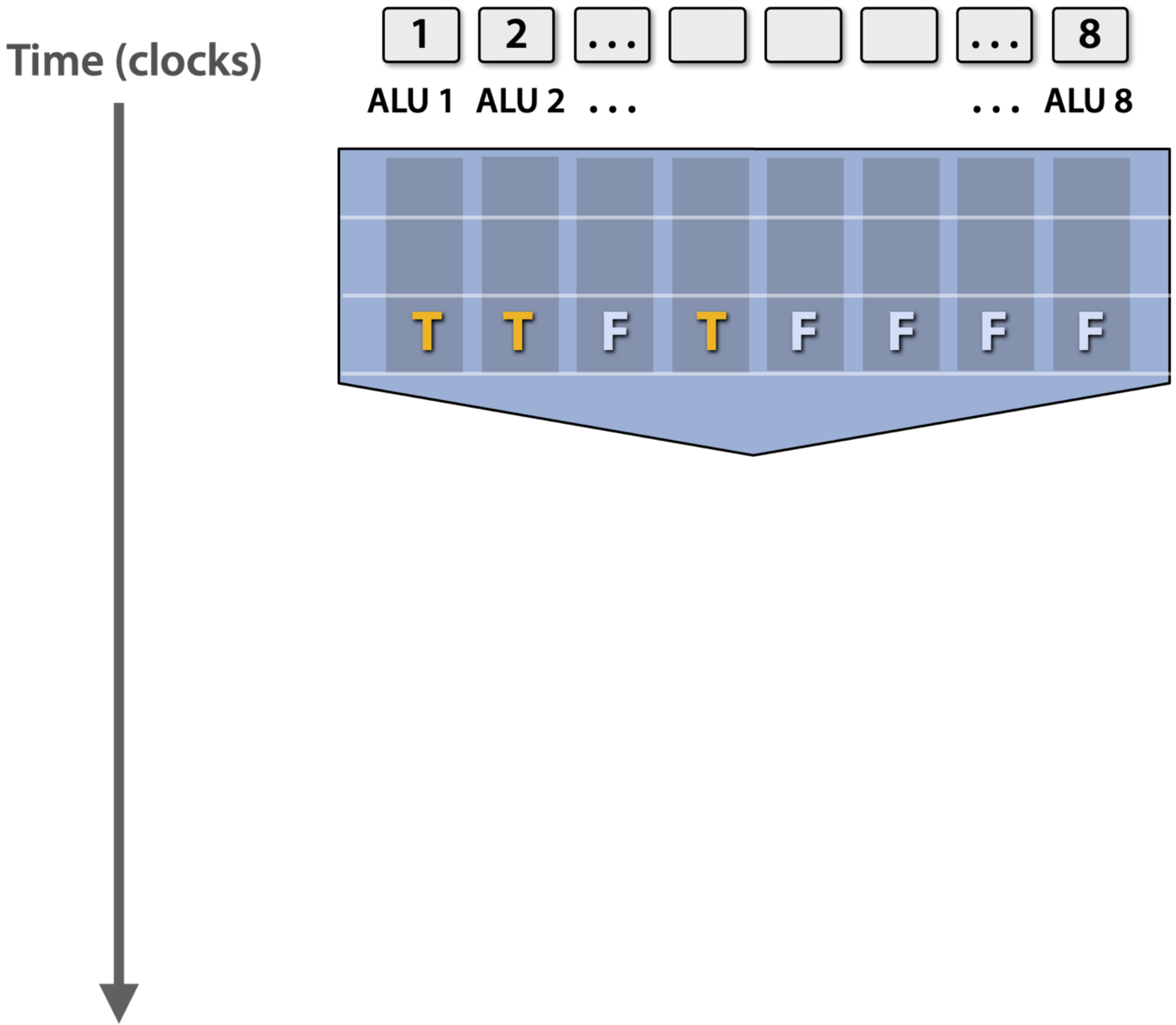
```
// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  // guard against out of bounds array access
  if (i < Nx && j < Ny)
    C[j][i] = A[j][i] + B[j][i];
}
```

What about Conditional Execution?



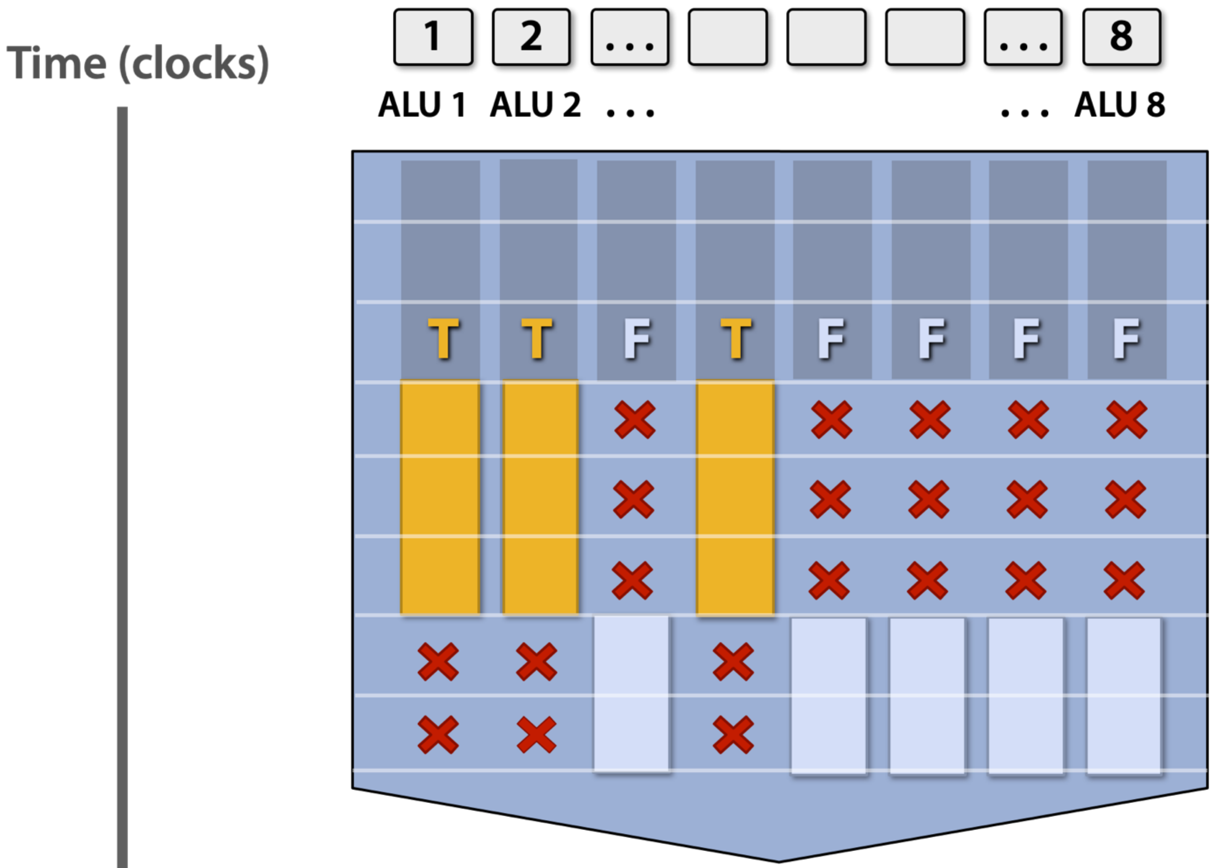

```
// kernel definition
__global__ void f(float A[N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = A[i];
  if (x > 0) {
    x = 2.0f * x;
  } else {
    x = exp(x, 5.0f);
  }
  A[i] = x;
}
```

What about Conditional Execution?




```
// kernel definition
__global__ void f(float A[N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = A[i];
  if (x > 0) {
    x = 2.0f * x;
  } else {
    x = exp(x, 5.0f);
  }
  A[i] = x;
}
```

Mask (discard) Output of ALU

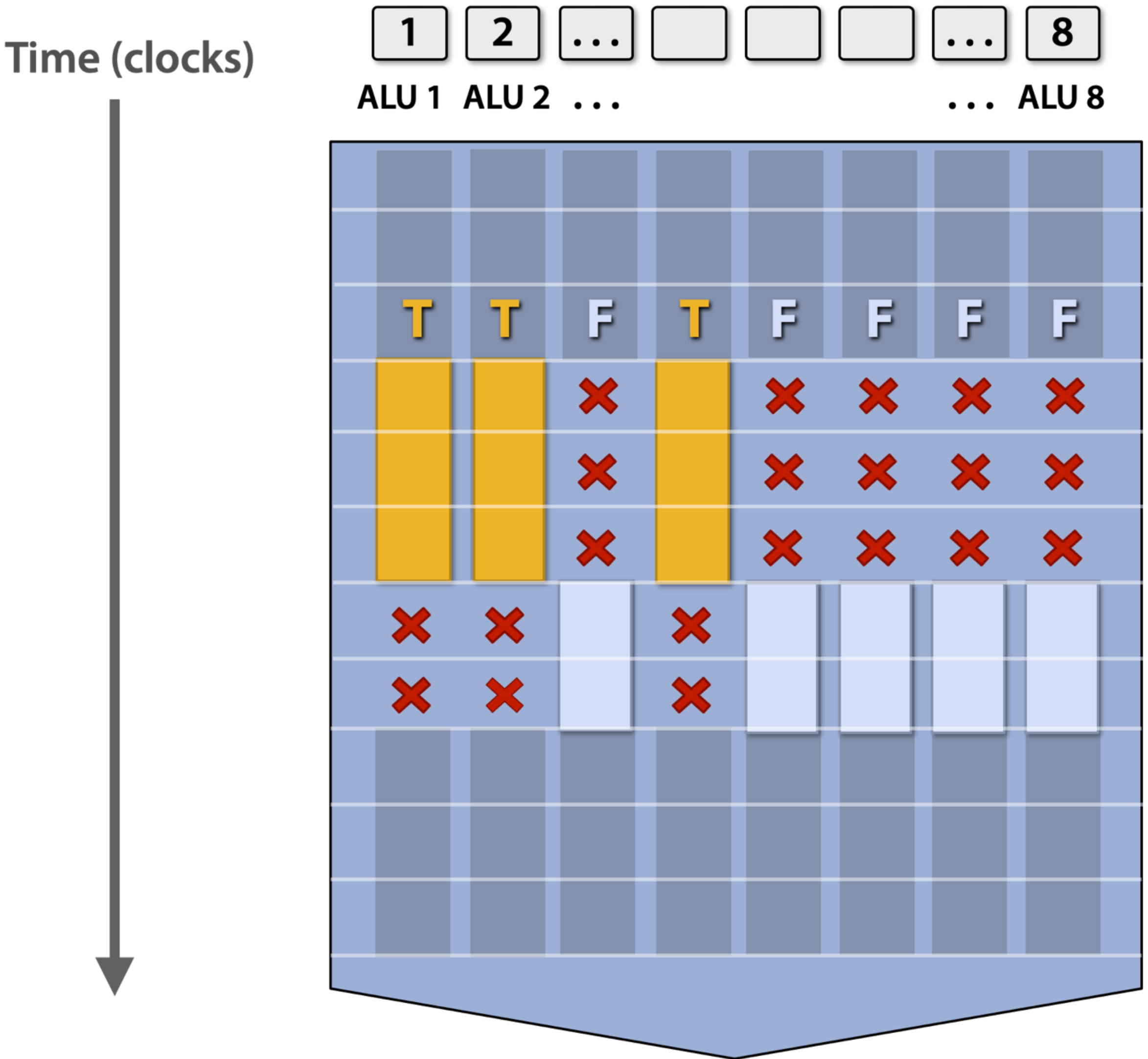


Not all ALUs do useful work!

Worst case: 1/8 peak performance

```
// kernel definition
__global__ void f(float A[N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = A[i];
  if (x > 0) {
        x = 2.0f * x;
  } else {
        x = exp(x, 5.0f);
  }
  A[i] = x;
}
```

After Branch: Continue at Full Performance




```
// kernel definition
__global__ void f(float A[N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = A[i];
  if (x > 0) {
          x = 2.0f * x;
  } else {
          x = exp(x, 5.0f);
  }
  A[i] = x;
}
```

Terminology

Coherence execution

- Same instruction sequence applies to all elements
- Necessary for efficient use of GPUs

Divergent execution

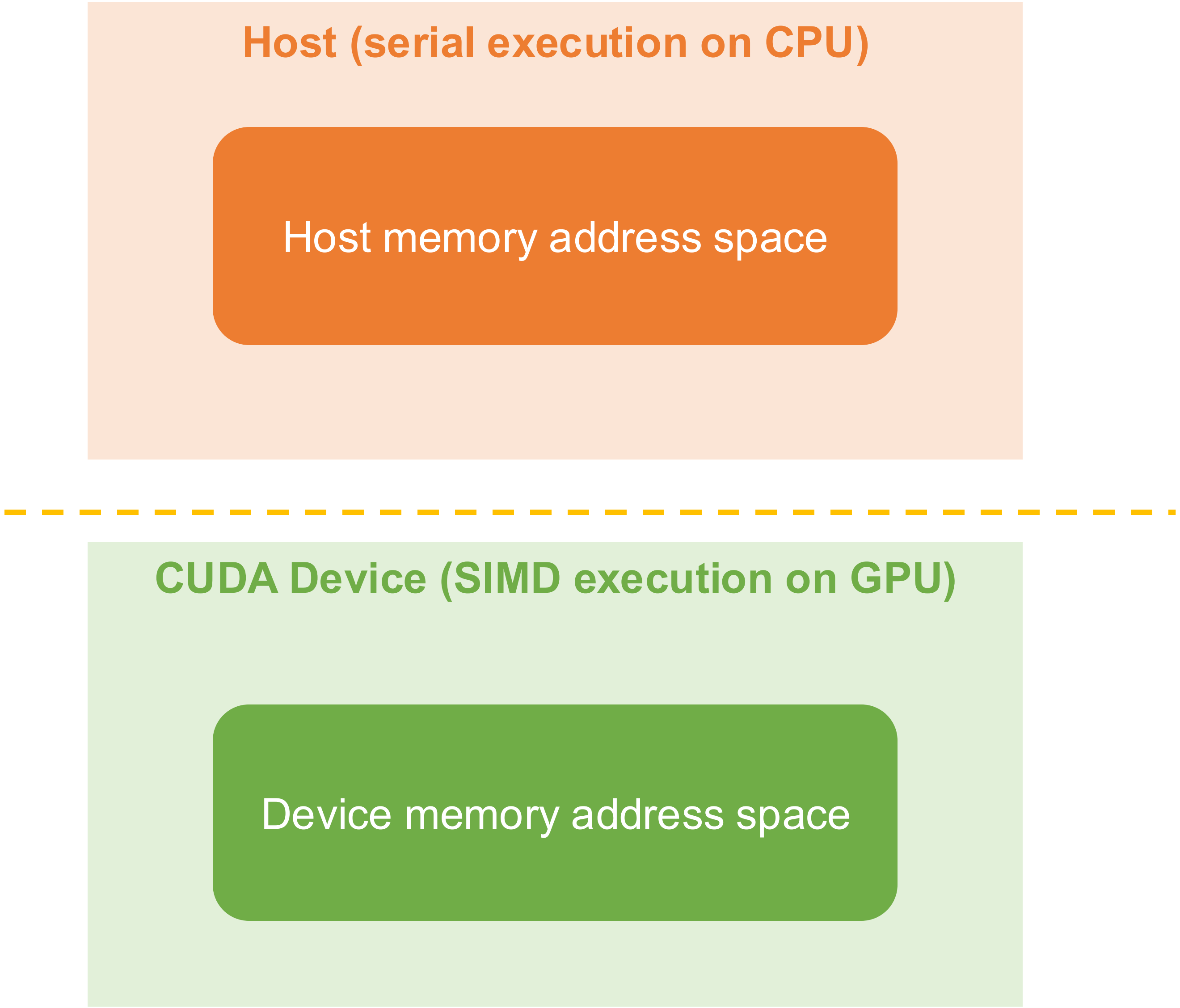
- A lack of coherence execution
- Should be minimized in CUDA programs

CUDA Memory Model

Host (serial execution on CPU)

CUDA Device (SIMD execution on GPU)

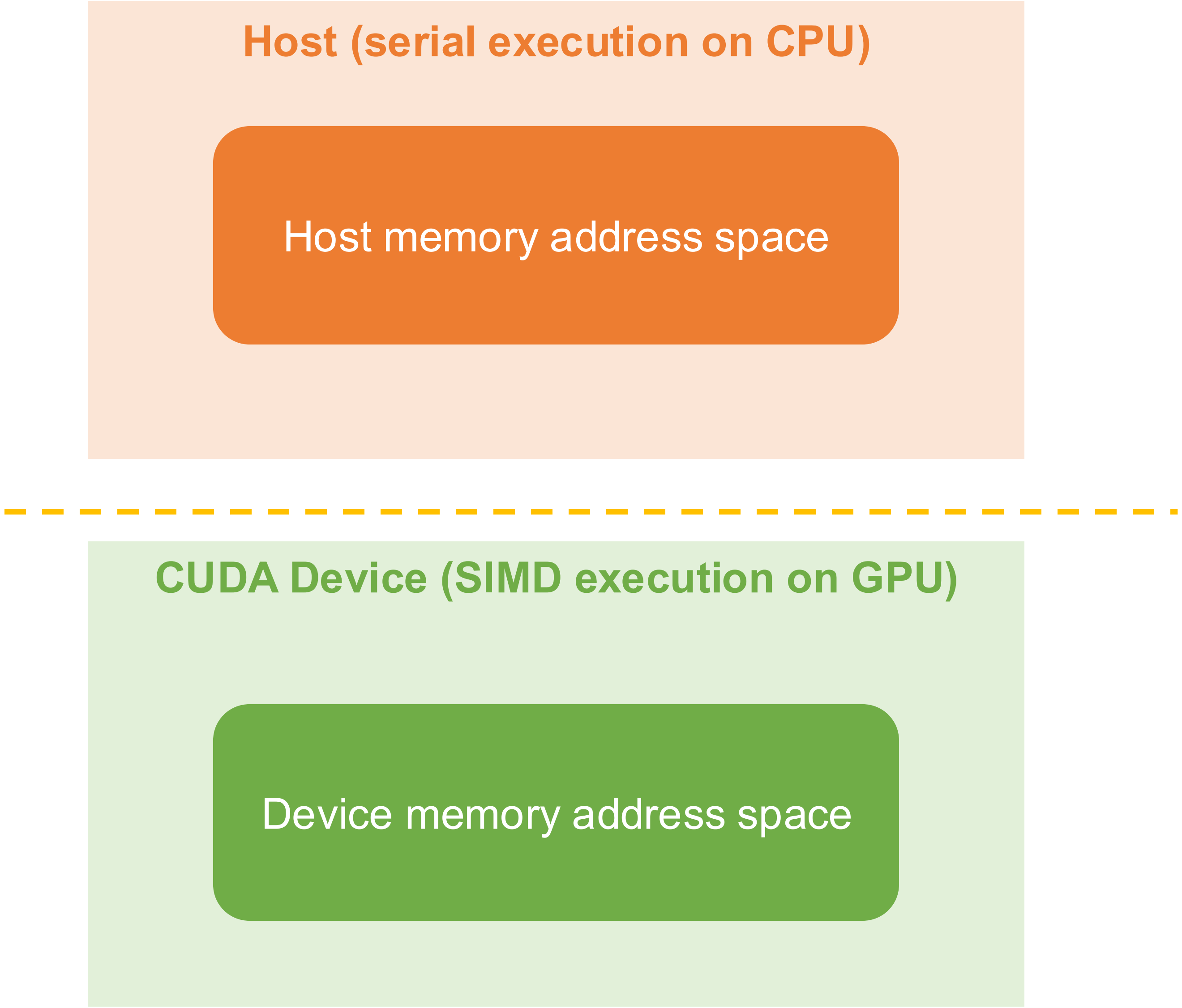
CUDA Memory Model



Distinct host and device address spaces

- Cannot access host memory from device
- Cannot access device memory from host

cudaMemcpy: Move Data Between Host and Device



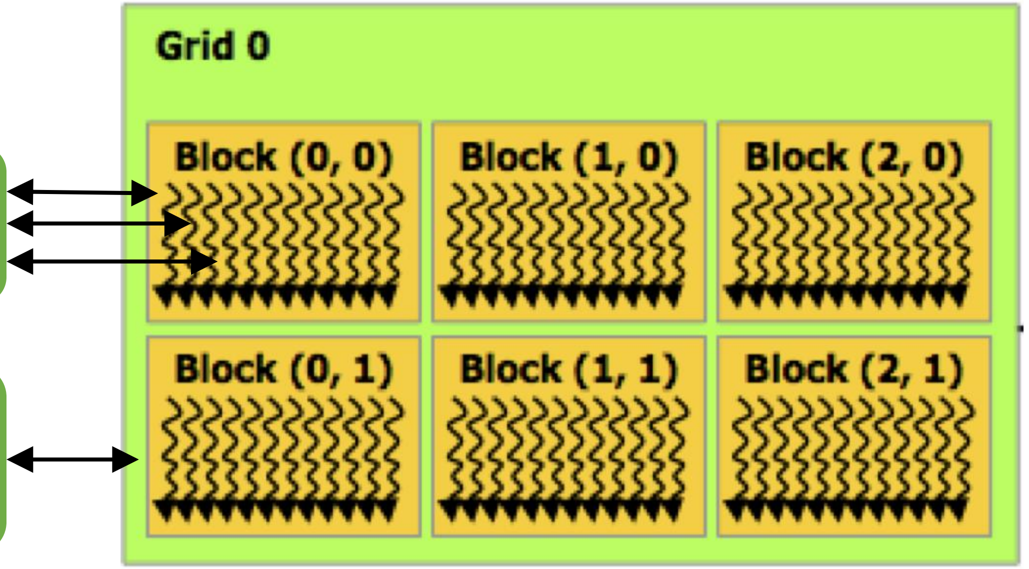

```
float* A = new float[N];

// populate host address space pointer A
for (int i=0 i<N; i++)
   A[i] = (float)i;

int bytes = sizeof(float) * N
float* deviceA;                // allocate buffer in
cudaMalloc(&deviceA, bytes);   // device address space

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);

// note: deviceA[i] is an invalid operation here (cannot
// manipulate contents of deviceA directly from host.
// Only from device code.)
```

CUDA Device Memory Model

- Three distinct types of memory available to kernels

Per-thread private memory (readable/writable by thread)

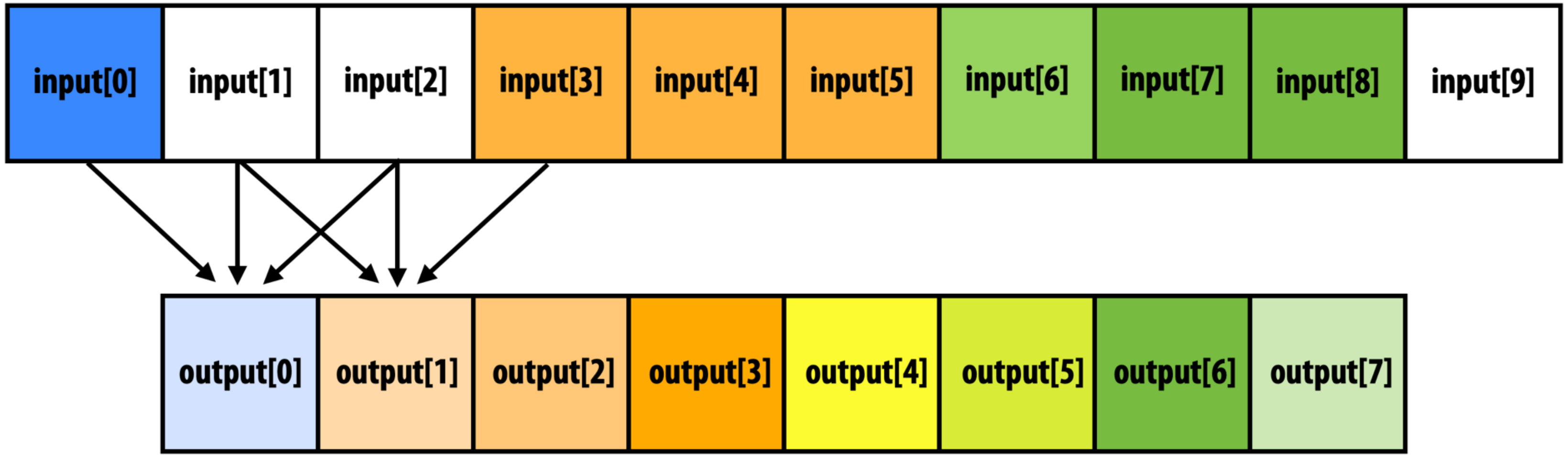
Per-block shared memory (readable/writable by all threads in a block)



Device global memory (readable/writable by all threads)

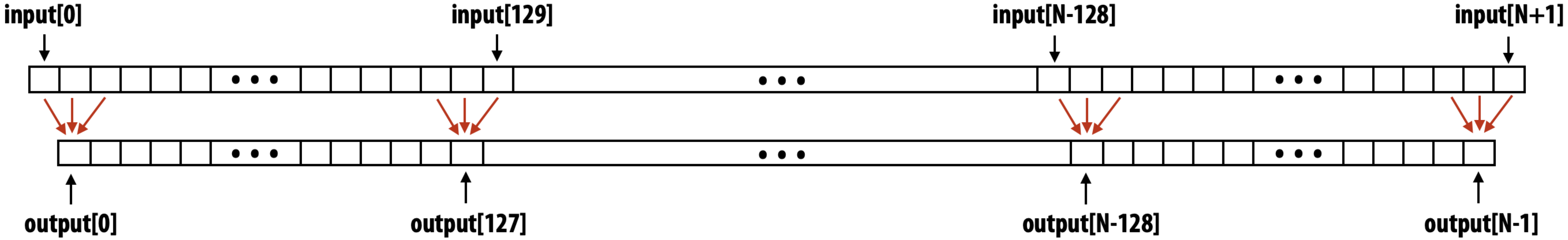
Why shared memory?
Enable cooperation across threads in a block

CUDA Programming Example: 1D Convolution



output[i] = (input[i] + input[i+1] + input[i+2]) / 3.f;

1D Convolution (Version 1)




```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );  // allocate array in device memory
cudaMalloc(&devOutput, sizeof(float) * N);      // allocate array in device memory

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

  int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

  float result = 0.0f;  // thread-local variable
  for (int i=0; i<3; i++)
    result += input[index + i];

  output[index] = result / 3.f;
}
```

each thread computes result for one element

1D Convolution (Reused Shared Memory)

```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );  // allocate array in device memory
cudaMalloc(&devOutput, sizeof(float) * N);      // allocate array in device memory

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

  int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

  __shared__ float support[THREADS_PER_BLK+2];        // per-block allocation
  support[threadIdx.x] = input[index];
  if (threadIdx.x < 2) {
    support[THREADS_PER_BLK + threadIdx.x] = input[index+THREADS_PER_BLK];
  }

  __syncthreads();

  float result = 0.0f;  // thread-local variable
  for (int i=0; i<3; i++)
    result += support[threadIdx.x + i];

  output[index] = result / 3.f;
}
```

All threads cooperatively load block's support region from global into shared memory (total of 130 loads instead of 3 * 128 loads)

barrier (all threads in block)

each thread computes result for one element

CUDA Synchronization Primitives

- __syncthreads(): wait for **all threads in a block** to arrive at this point
- Atomic operations
  - e.g., float atomicAdd(float* addr, float amount)
  - Atomic operations on both global and shared memory
- Host/device synchronization
  - Implicit barrier across all threads at return of kernel

CUDA Abstractions

- Execution: thread hierarchy
  - Bulk launch of many threads
  - Two-level hierarchy: threads are grouped into thread blocks
- Distributed address space
  - Memcpy primitives to copy between host and device address spaces
  - Three different types of device address spaces
  - Per thread, per block (“shared”), or per program (“global”)
- Barrier synchronization primitive for threads in thread block
- Atomic primitives for additional synchronization (shared and global variables)

CUDA Compilation

```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );
cudaMalloc(&devOutput, sizeof(float) * N);

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

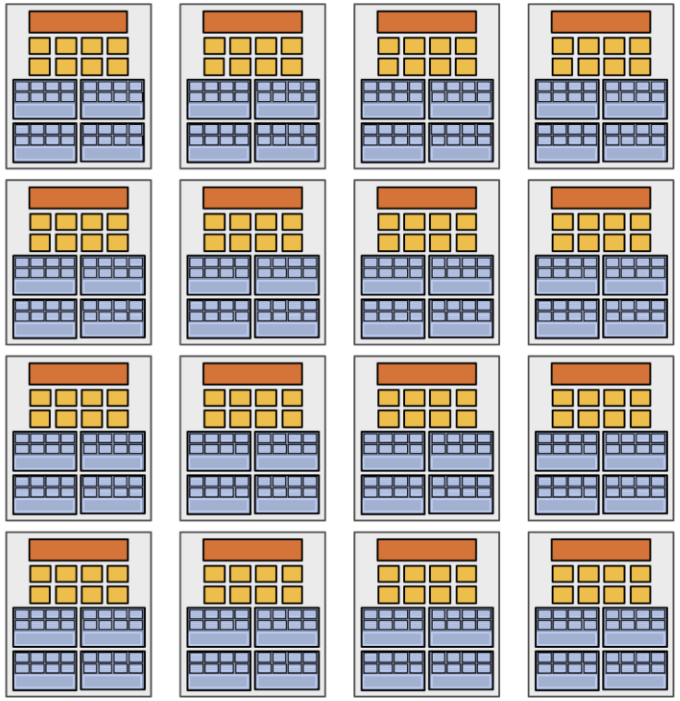
Launch 8K thread blocks

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

  int index = blockIdx.x * blockDim.x + threadIdx.x;

  __shared__ float support[THREADS_PER_BLK+2];
  support[threadIdx.x] = input[index];
  if (threadIdx.x < 2) {
    support[THREADS_PER_BLK + threadIdx.x] = input[index+THREADS_PER_BLK];
  }

  __syncthreads();

  float result = 0.0f;  // thread-local variable
  for (int i=0; i<3; i++)
    result += support[threadIdx.x + i];

  output[index] = result / 3.f;
}
```

CUDA Compilation

- Goal: run a CUDA program on various GPUs

Mid-range GPU (6 cores)



High-end GPU (16 cores)

CUDA Compilation

```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );
cudaMalloc(&devOutput, sizeof(float) * N);

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

Launch 8K thread blocks

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

  int index = blockIdx.x * blockDim.x + threadIdx.x;

  __shared__ float support[THREADS_PER_BLK+2];
  support[threadIdx.x] = input[index];
  if (threadIdx.x < 2) {
    support[THREADS_PER_BLK + threadIdx.x] = input[index+THREADS_PER_BLK];
  }

  __syncthreads();

  float result = 0.0f;  // thread-local variable
  for (int i=0; i<3; i++)
    result += support[threadIdx.x + i];

  output[index] = result / 3.f;
}
```

A compiled CUDA device binary includes:

Program text (instructions)
Information about required resources:

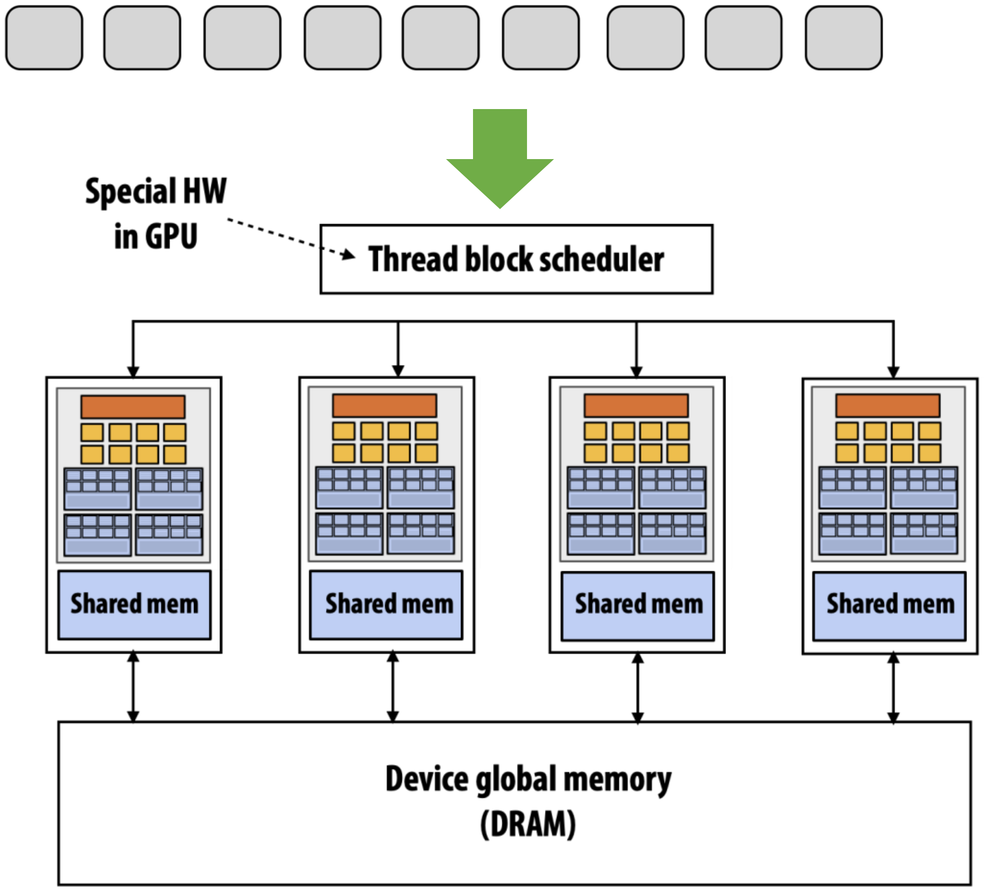
- 128 threads per block
- 8 bytes of local data per thread
- 130 floats (520 bytes) of shared space per thread block

CUDA Thread Block Scheduling

- **Major CUDA assumption**: threadblocks can be executed in any order (no dependencies between threadblocks)

- GPU maps threadblocks to cores using a dynamic scheduling policy that respects resource requirements

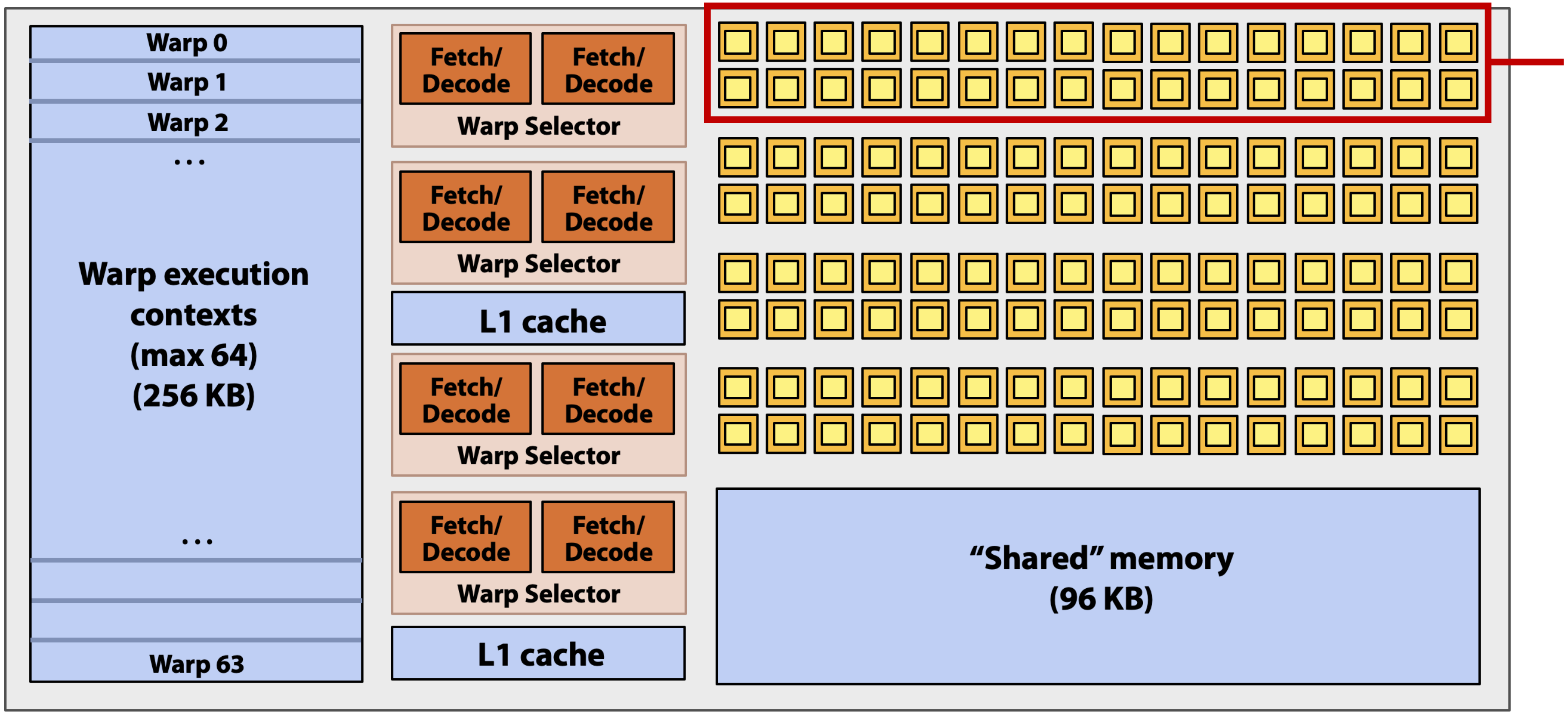
Grid of 8K convolve thread blocks (specified by kernel launch)
Block requirements:
- 128 threads
- 520 bytes of shared memory
- 1024 bytes of local memory



A GPU Core: Streaming Multiprocessor

SMM resource limits:

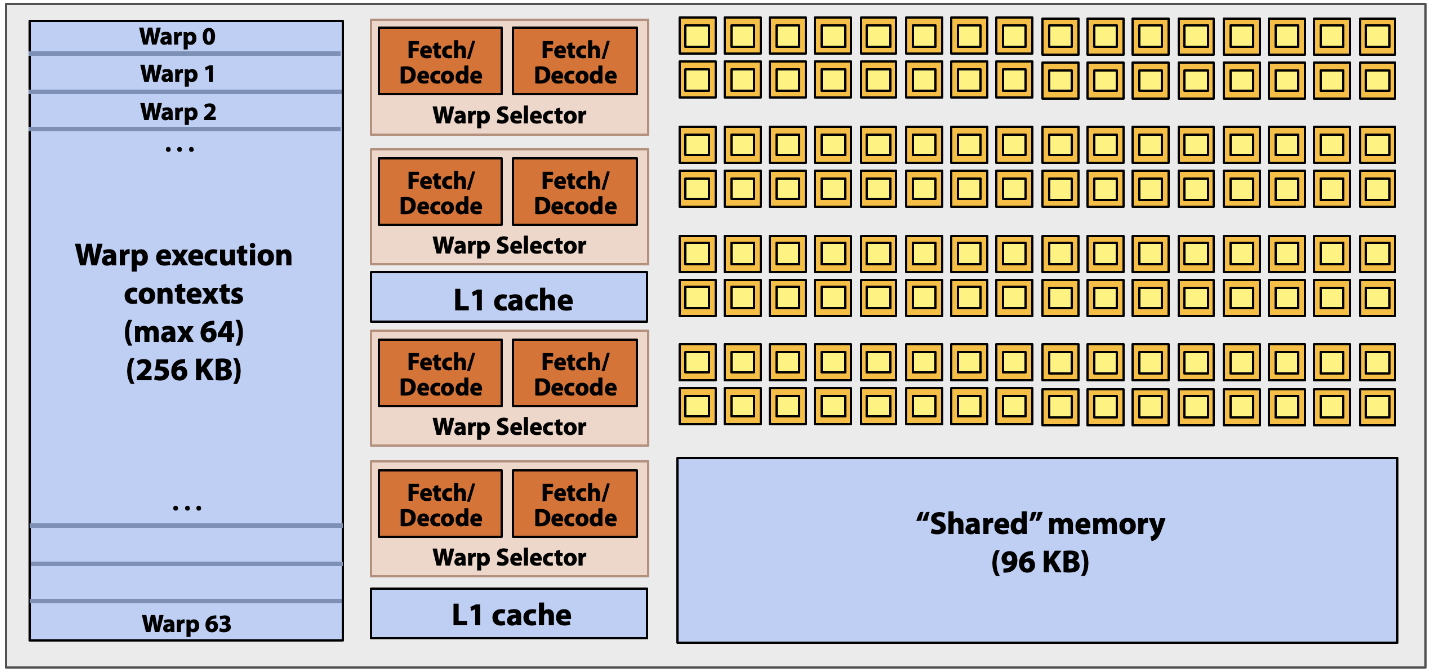
- Max warp execution contexts: 64 (up to 64 * 32 = 2K total CUDA threads)
- 96 KB of shared memory



SIMD functional unit, control shared across 32 units (1 MUL-ADD per clock)

Running a Thread Block on an SMM

- Warp: A group of 32 CUDA threads shared an instruction stream.
  - A convolve thread block is executed by 4 warps (4 warps x 32 threads / warp = 128 threads)
- SMM operation each clock:
  - Select up to four runnable warps from 64 resident on an SMM (thread-level parallelism)
  - Select up to two runnable instructions per warp (instruction-level parallelism)

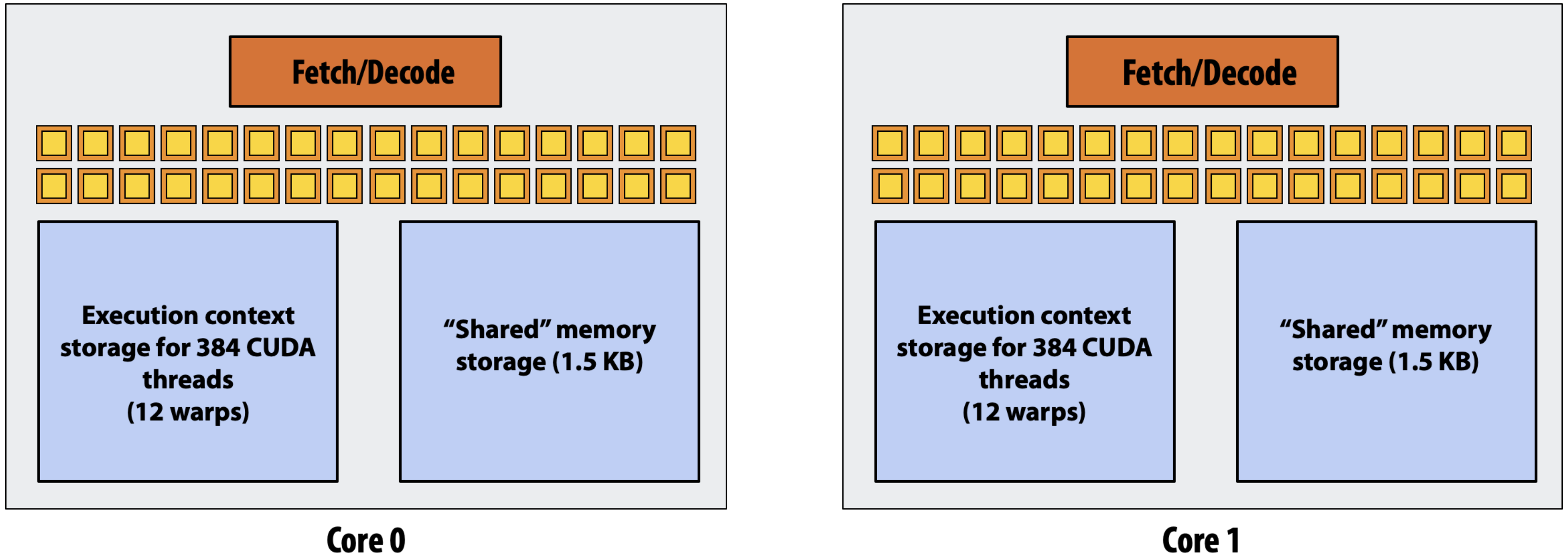


Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Assume the host side launches 1000 thread blocks

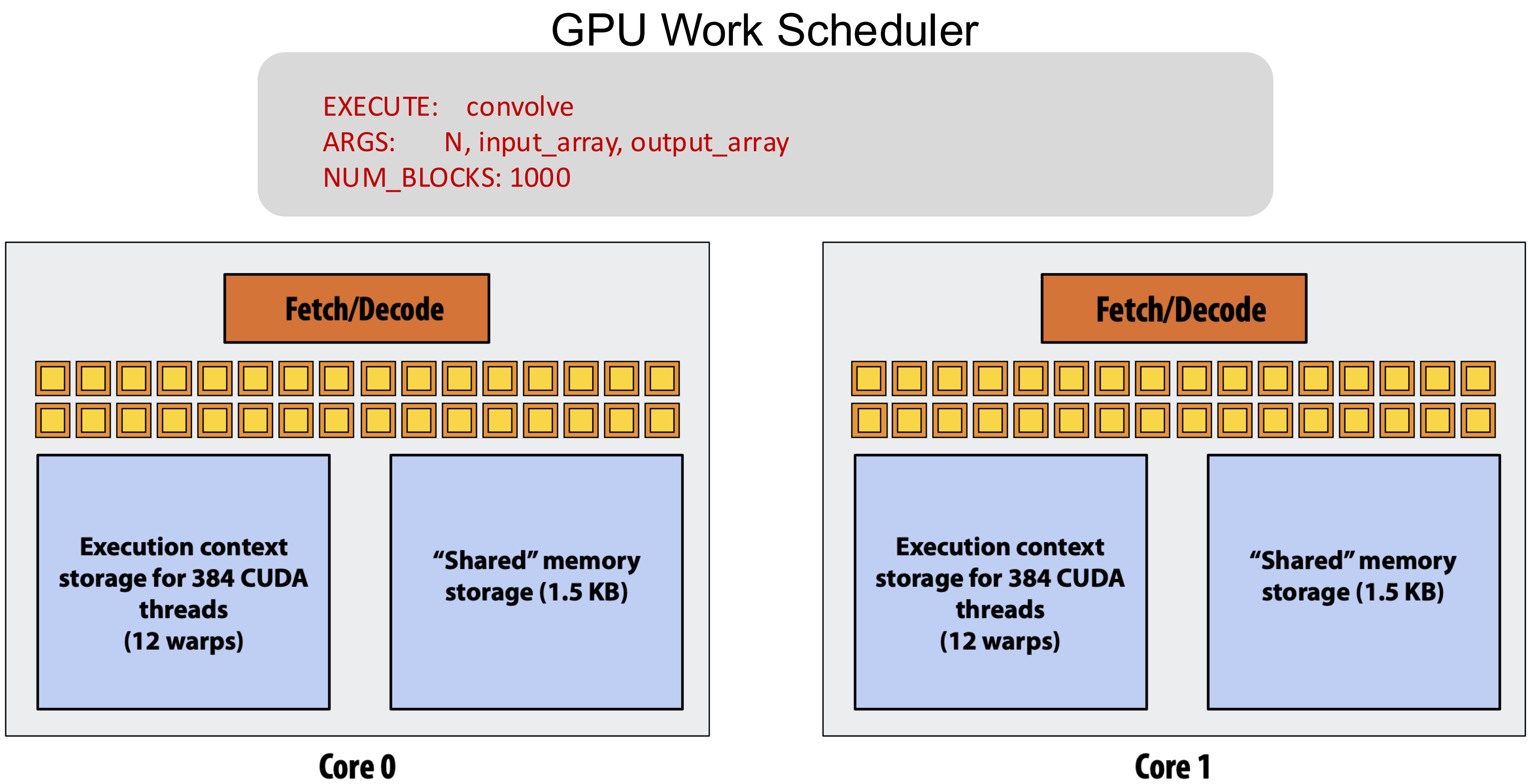
```
#define THREADS_PER_BLK 128
convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, input_array, output_array);
```

- Run the program on a two-SMM GPU



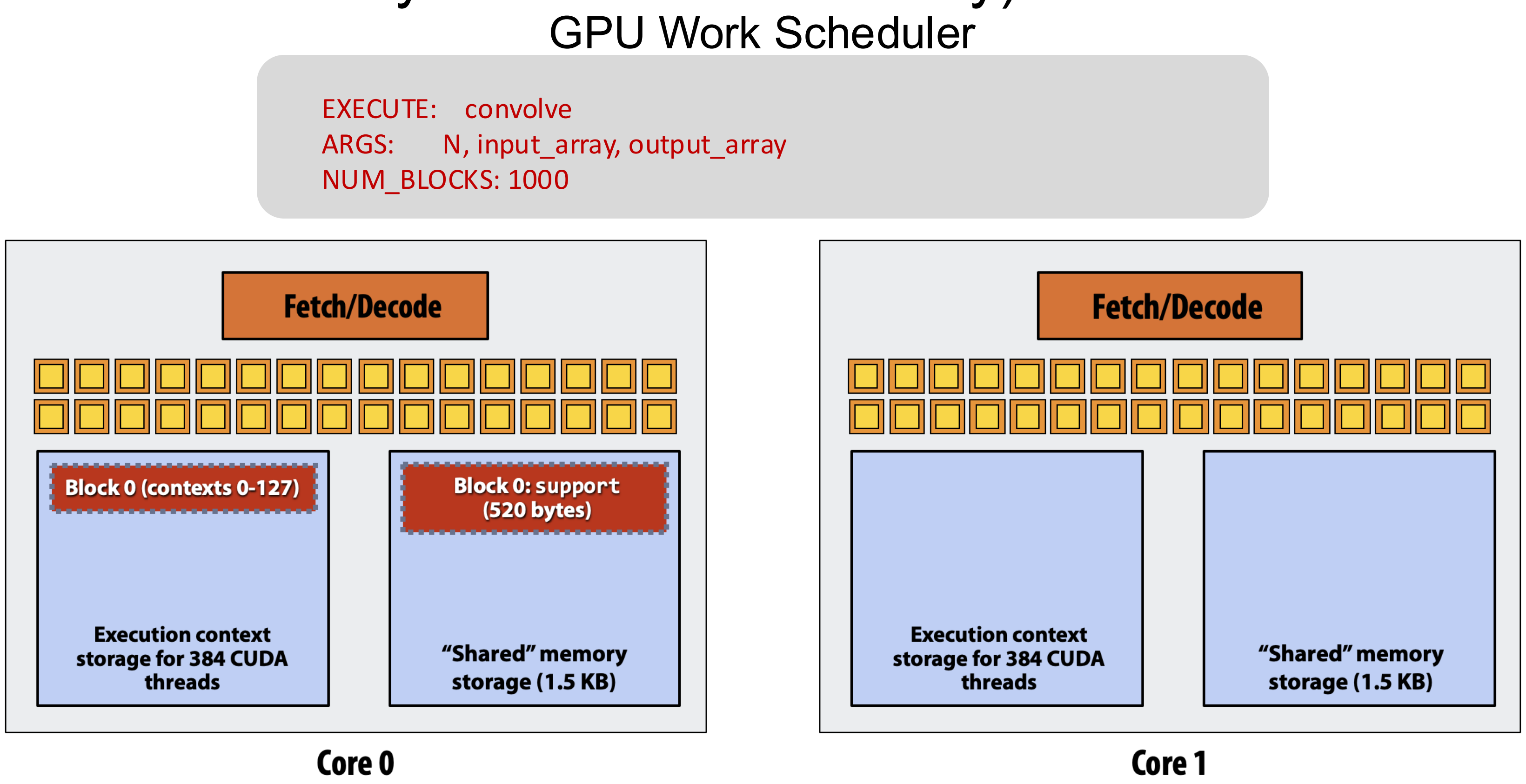
Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Step 1: host sends CUDA kernel to device



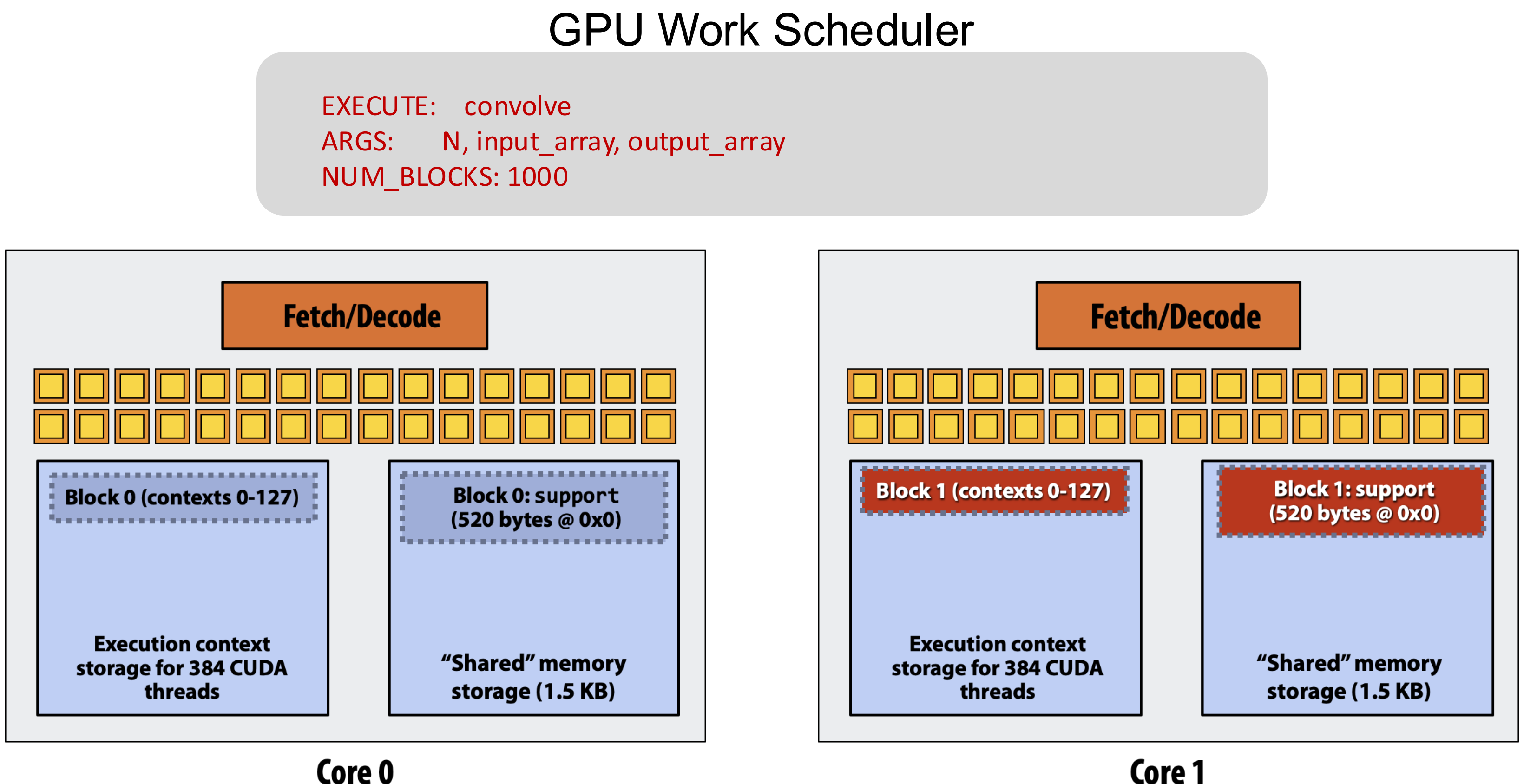
Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Step 2: scheduler maps block 0 to core 0 (reserves execution contexts for 128 threads and 520 bytes of shared memory)



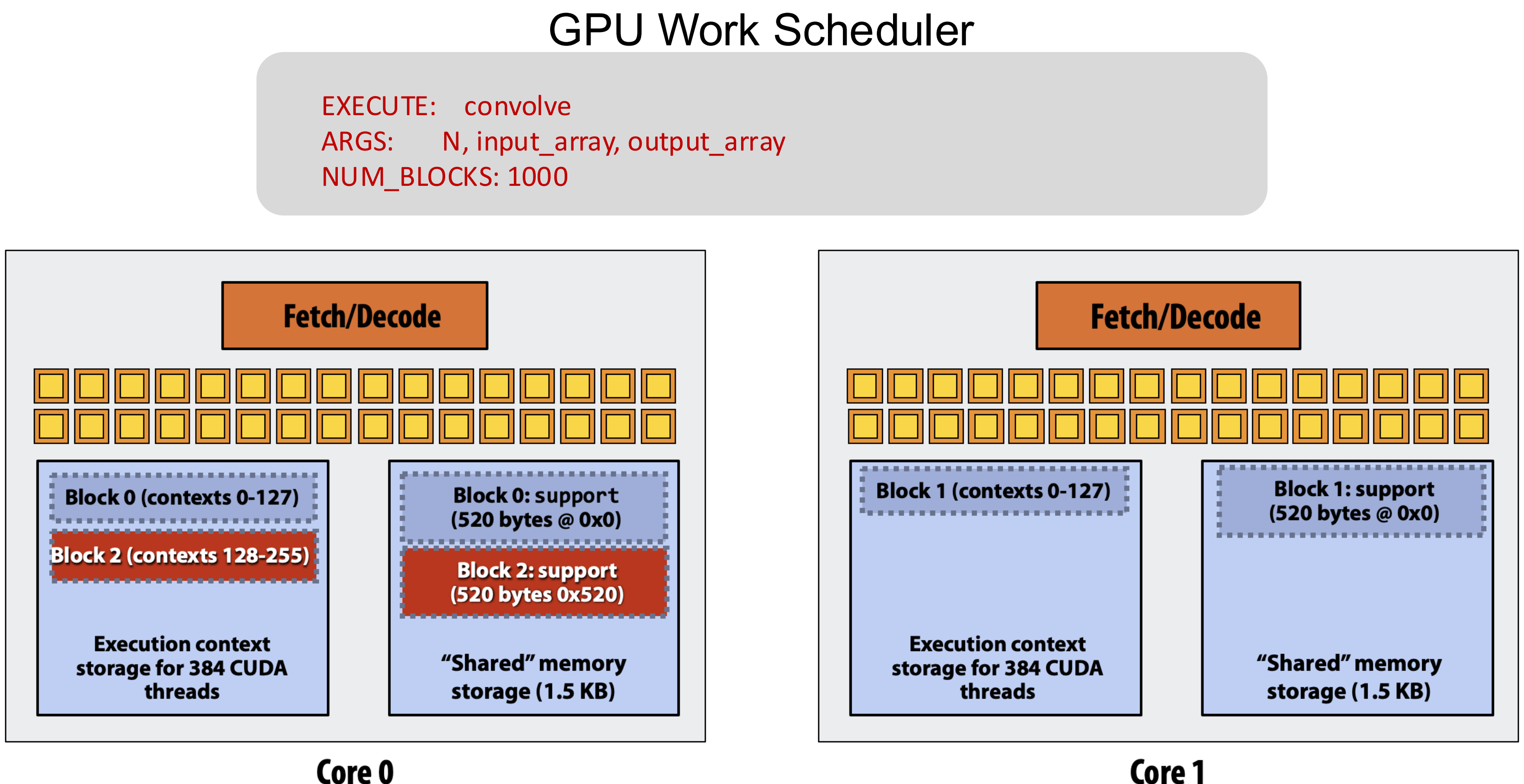
Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Step 3: scheduler continues to map blocks to available execution contexts



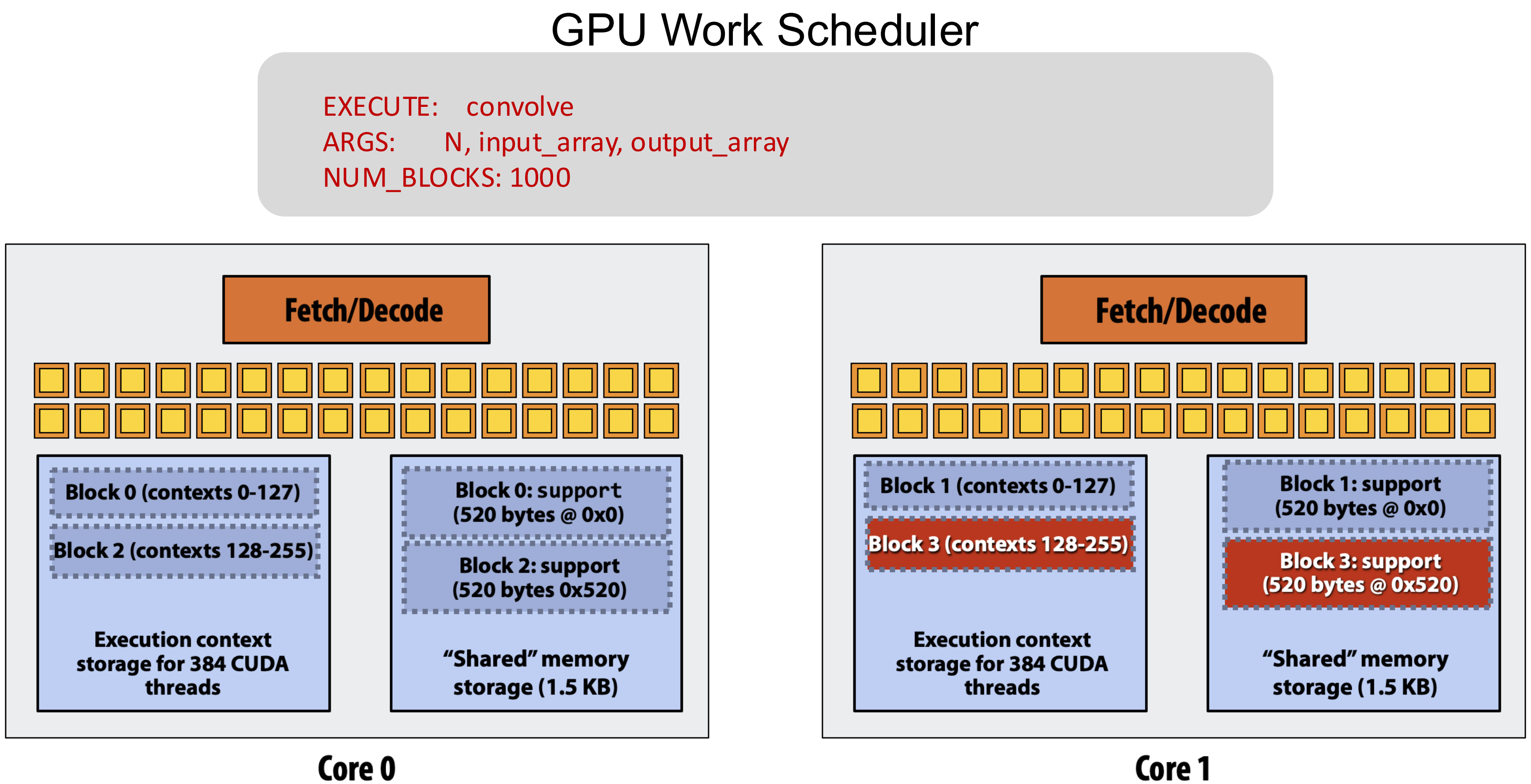
Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Step 3: scheduler continues to map blocks to available execution contexts



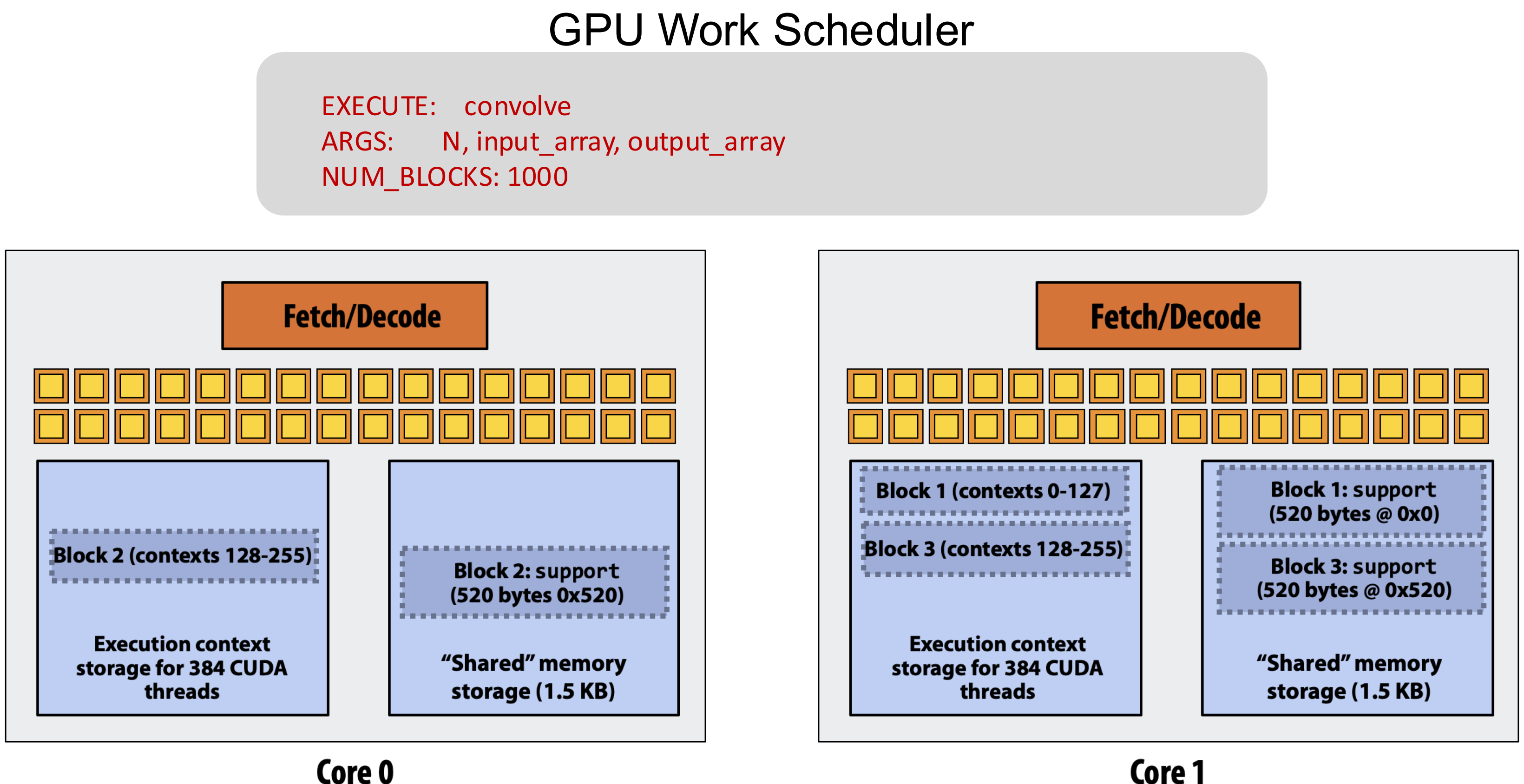
Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- (third block won't fit due to insufficient shared storage 3 x 520B > 1.5KB)



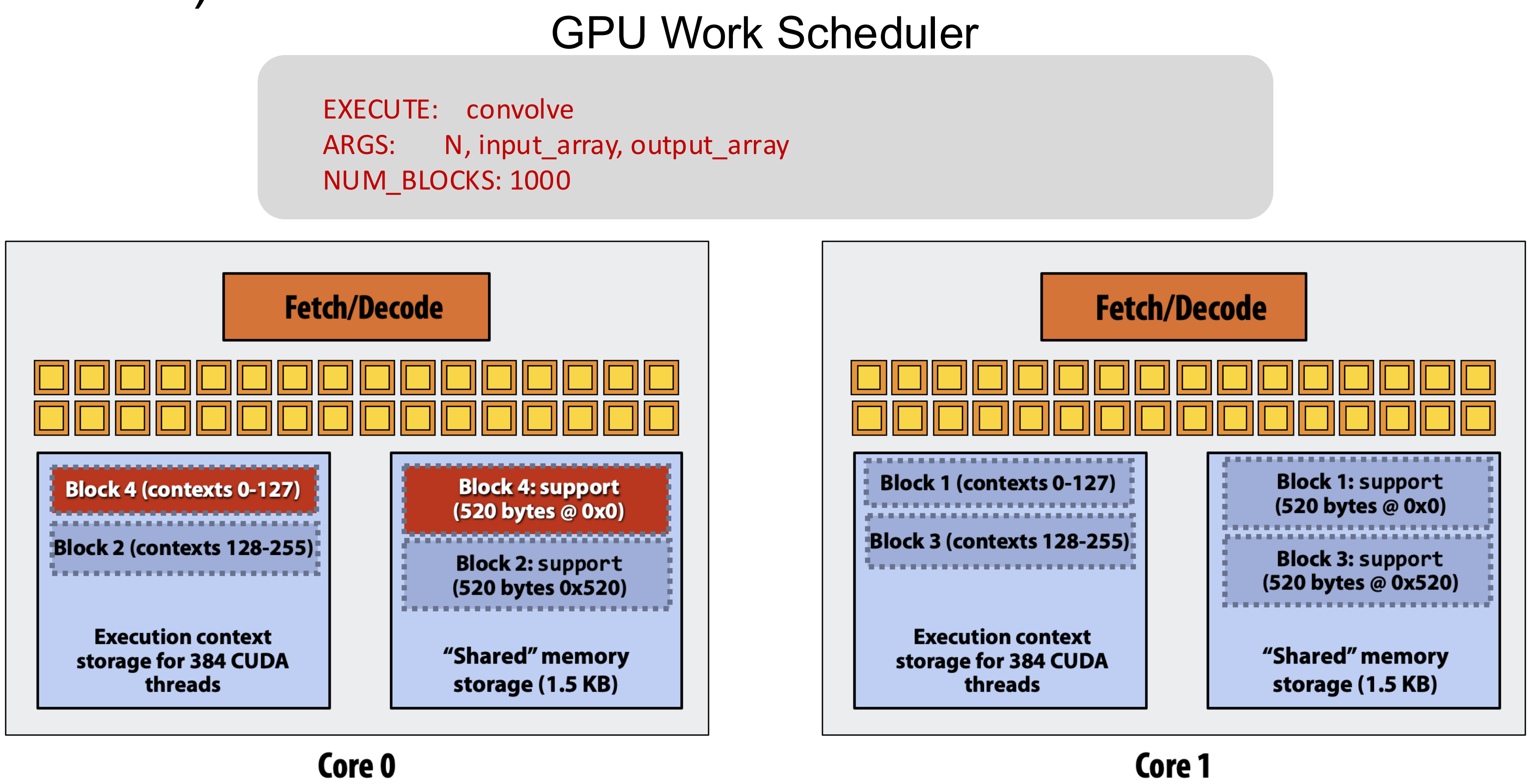
Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Step 4: thread block 0 completes on core 0



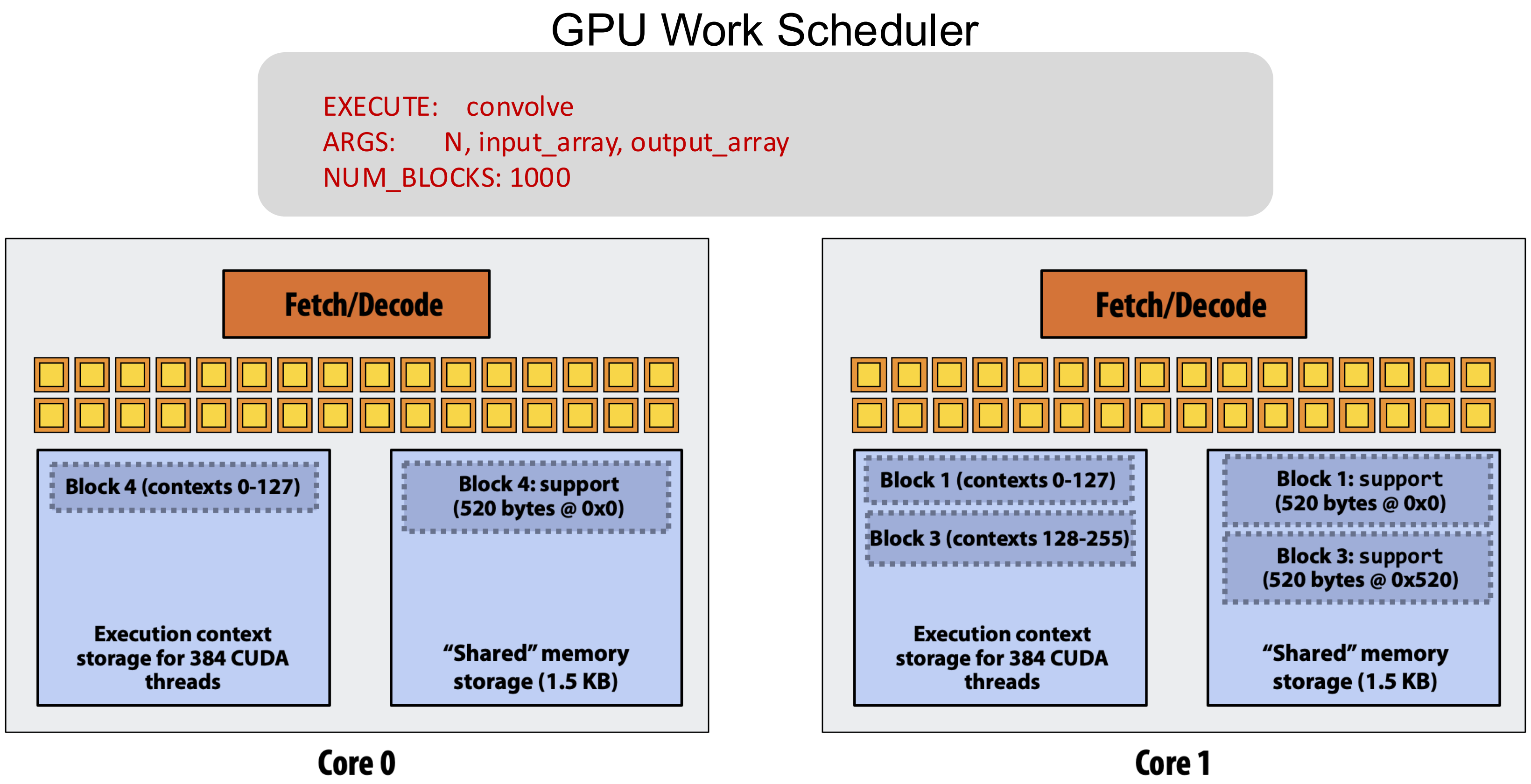
Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Step 5: thread block 4 is scheduled on core 0 (mapped to execution contexts 0-127)



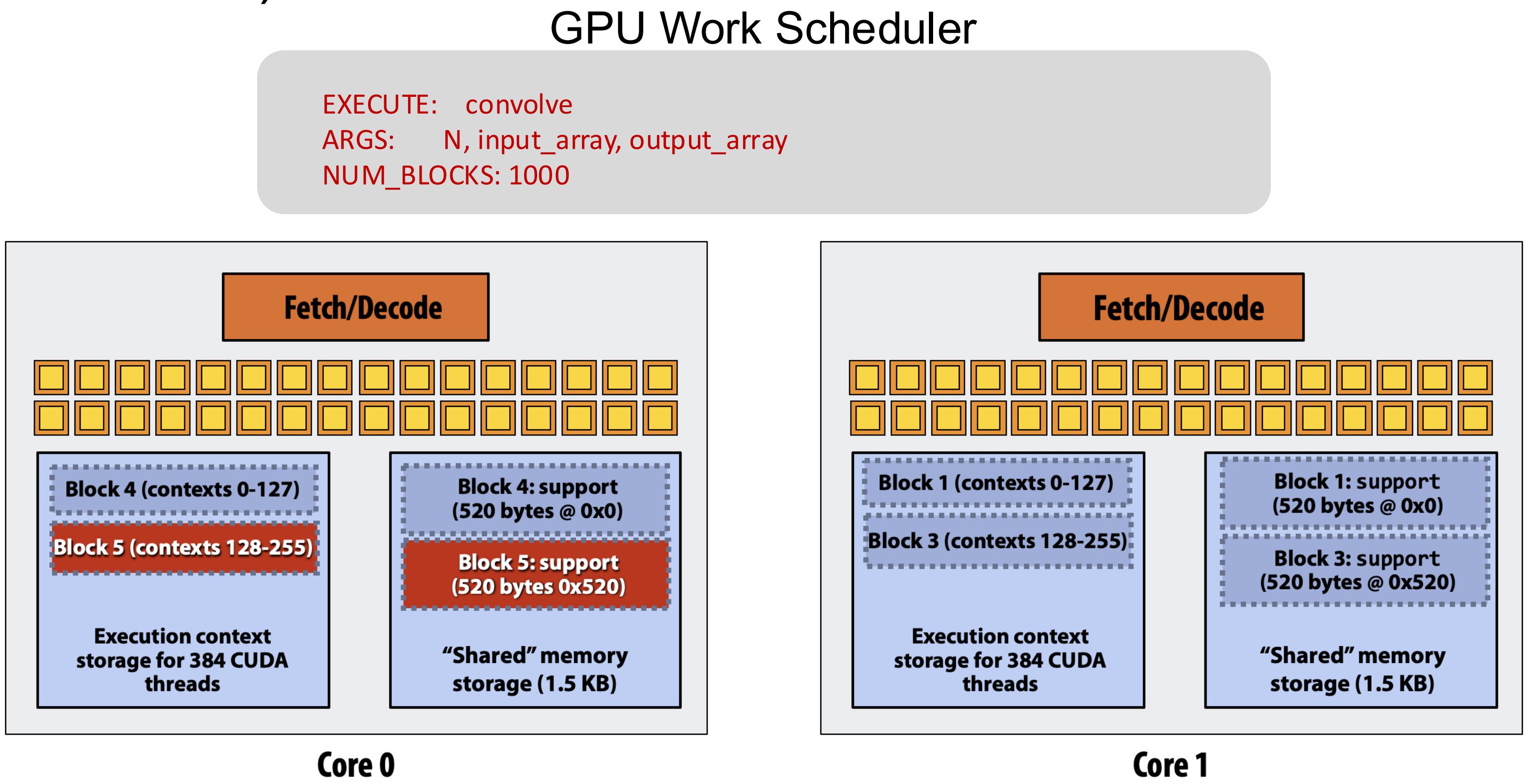
Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Step 6: thread block 2 completes on core 0



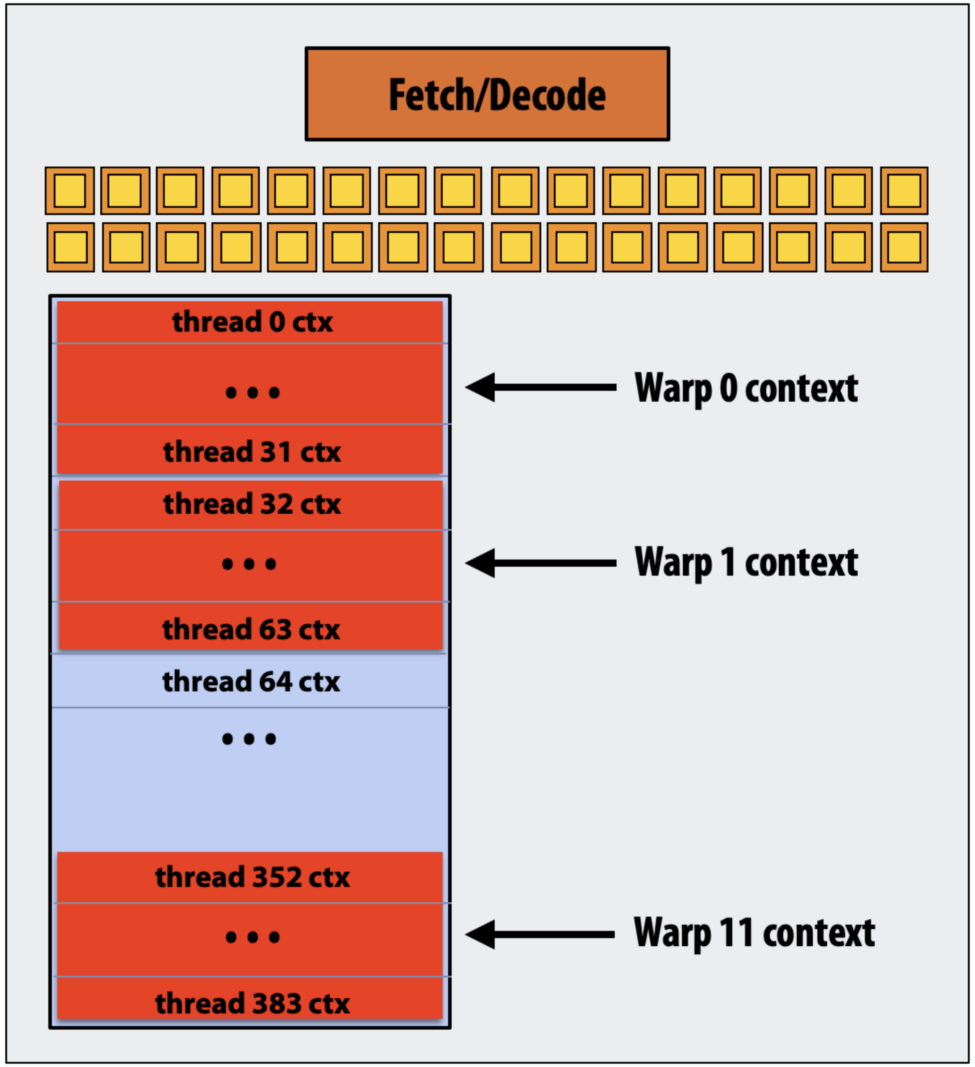
Running a CUDA Kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Step 7: thread block 5 is scheduled on core 0 (mapped to execution contexts 128-255)



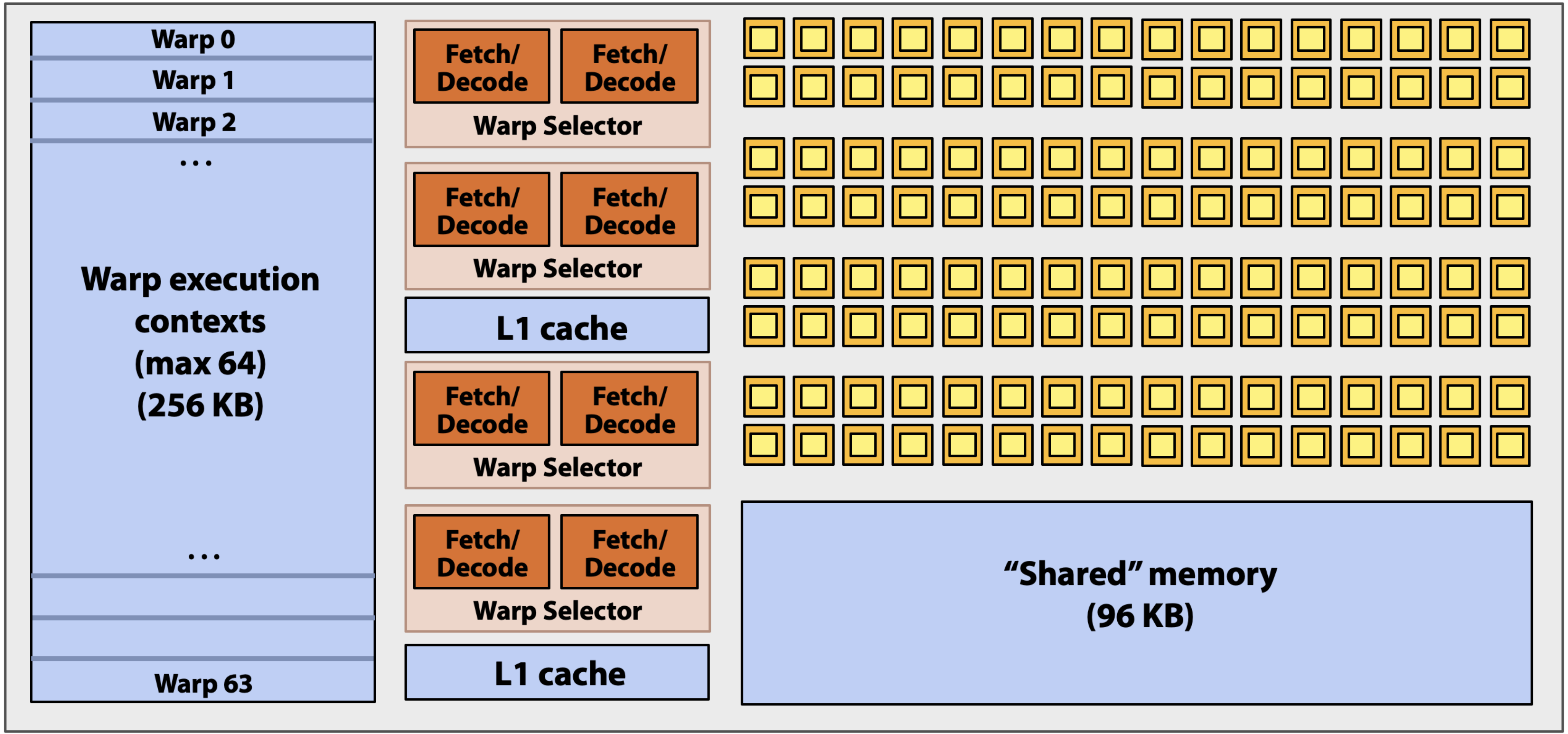
What is a warp?

- A warp is a CUDA implementation detail on NVIDIA GPUs
- On modern NVIDIA GPUs, groups of 32 CUDA threads in a thread block are executed simultaneously using 32-wide SIMD execution
- Each SMM can execute (GTX 980) up to 64 warps

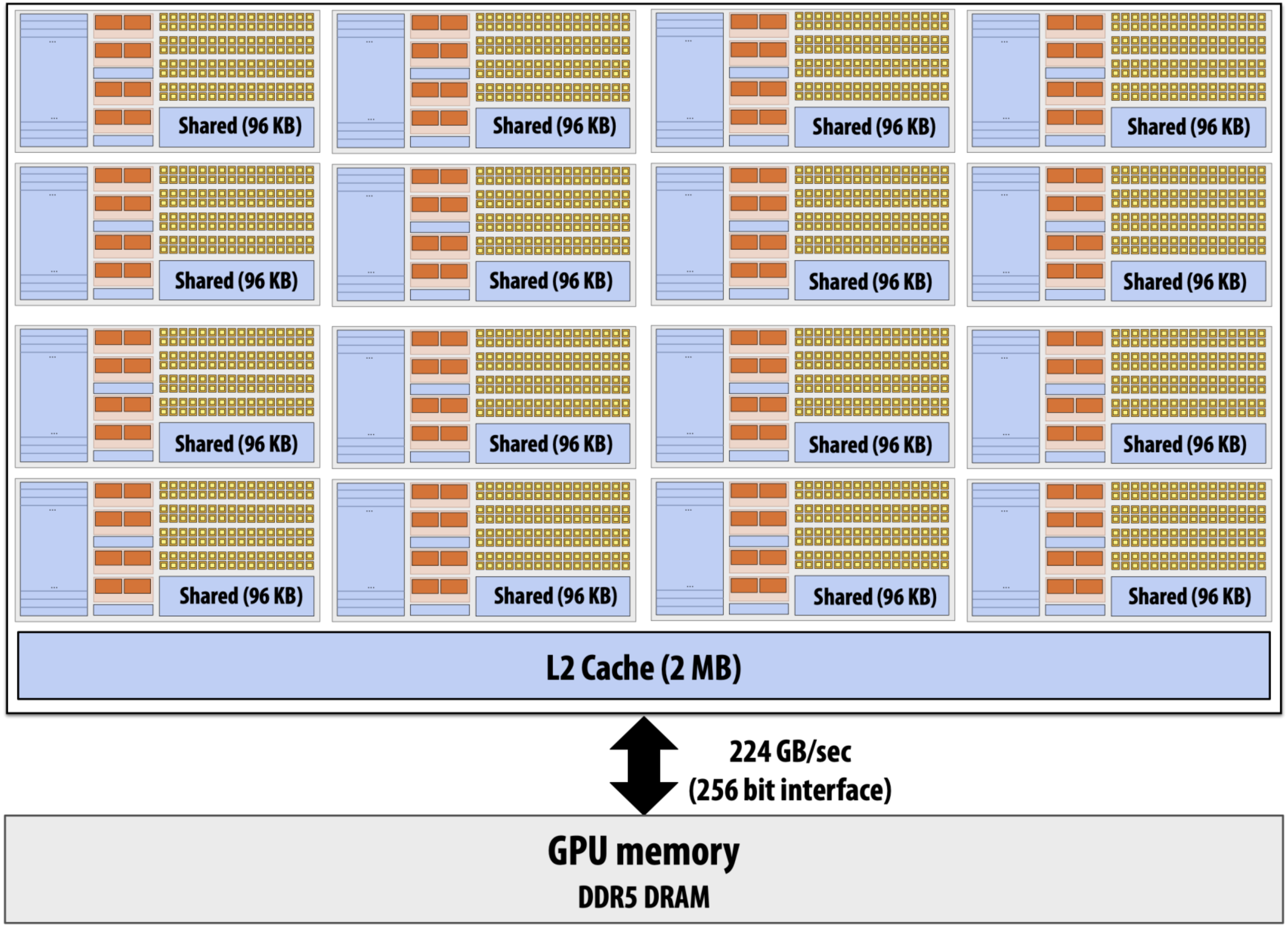


Recall: An SMM on a NVIDIA GTX 980 (2014)

- SMM resource:
  - Map warp execution contexts: 64 (64 * 32 = 2048 total CUDA threads)
  - 96 KB of shared memory



NVIDIA GTX 980 Contains 16 SMMs



1.1 GHz clock

16 SMM cores per chip

16 x 4 warps x 32 threads / warp
= 2048 SIMD mul-add ALUs
= 4.6 TFLOPs

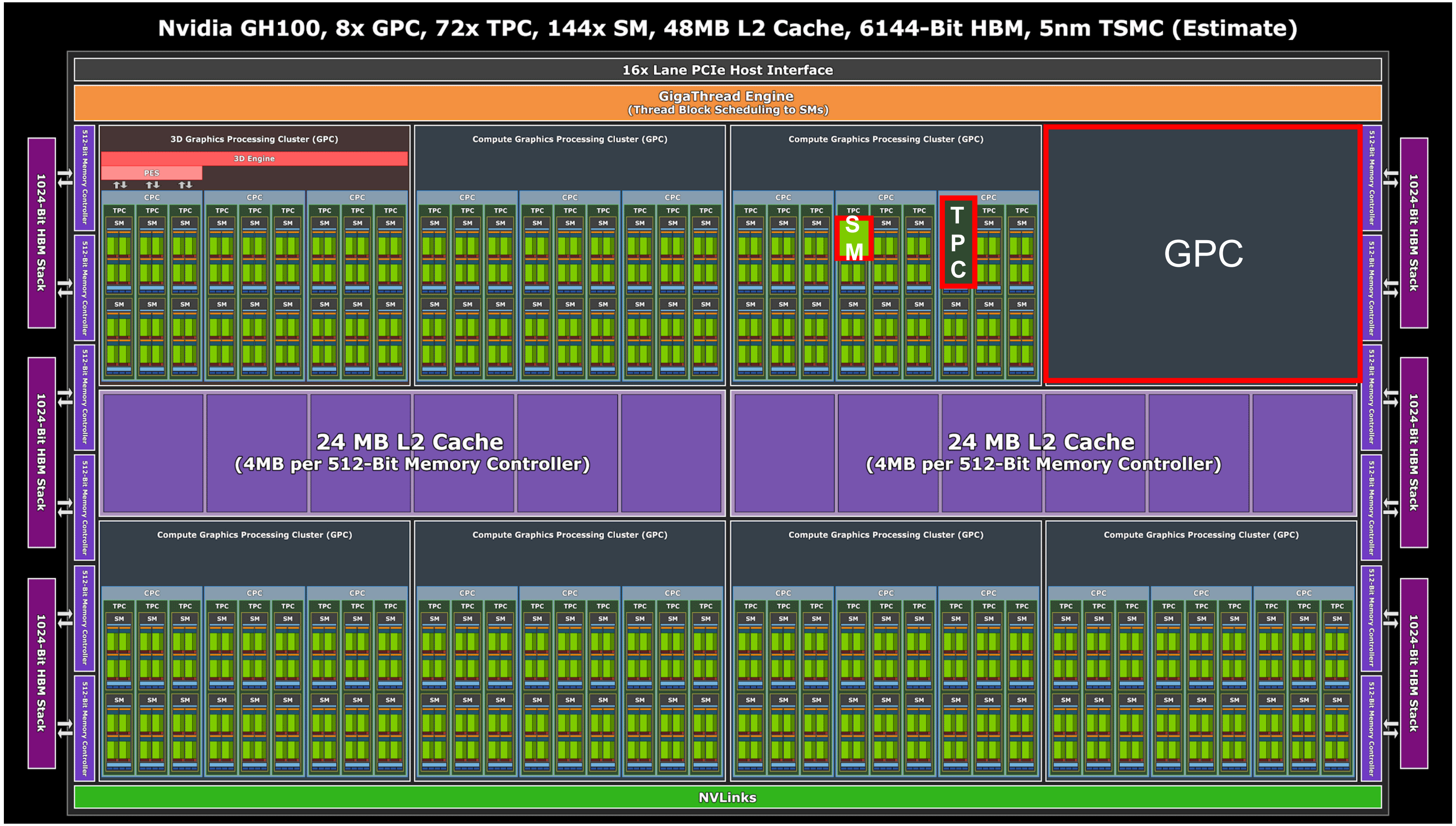
Up to 16 x 64 = 1024 interleaved warps per chip
(32,768 CUDA threads / chip)

GTX 980 (2014) -> H100 (2022)

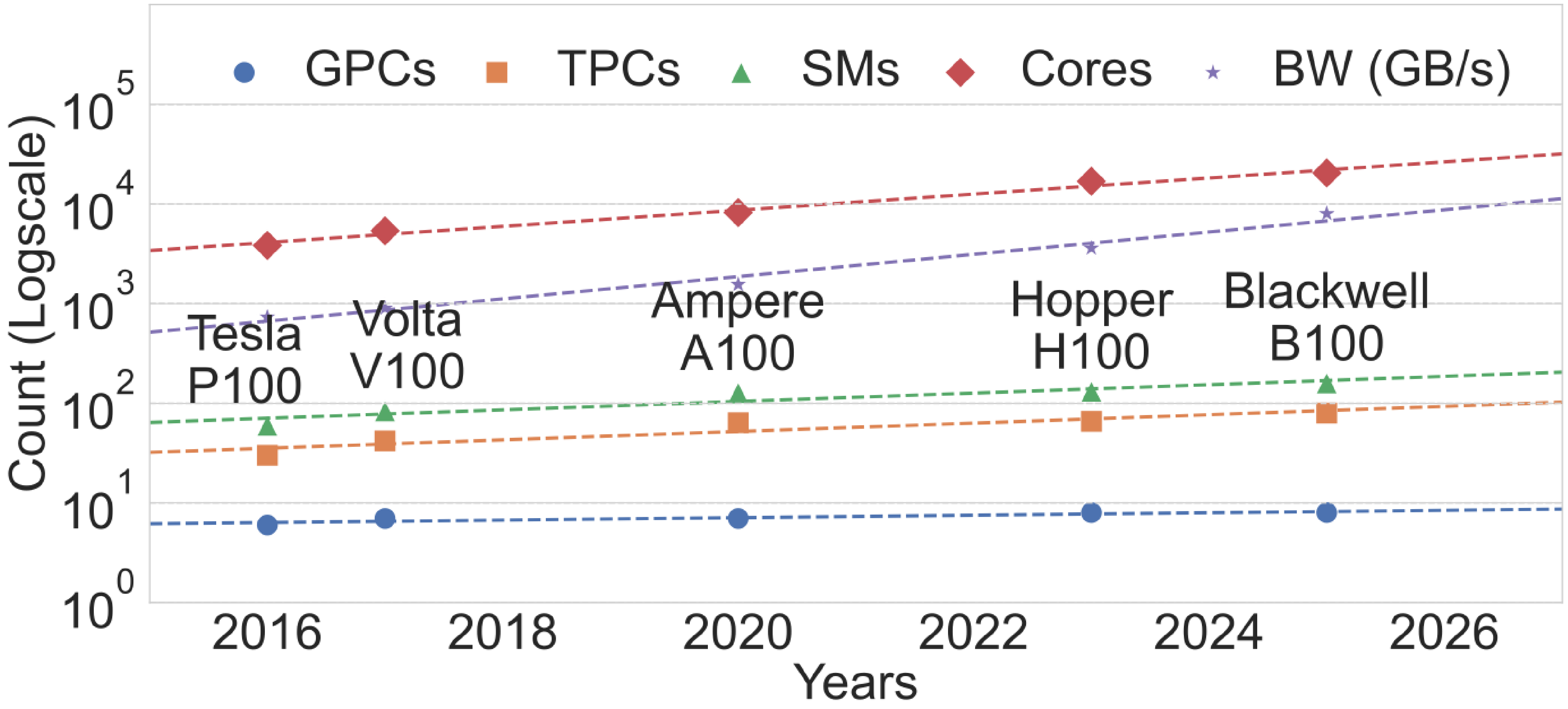
- SMMs remain the same
  - Clock speed: 1064 MHz -> 1110 MHz
  - Map warps per SMM: 64 -> 64
  - Threads per warp: 32 -> 32
  - Shared memory per SMM: 96 KB -> 168 KB (A100) -> 256 KB (H100)

- Streaming multiprocessors: 16 SMMs -> 132 SMMs
- Peak performance 4.6 TFLOPs -> 1000 TFLOPs (mainly because of tensor cores)

H100 Architecture



Evolution of GPUs



P. Coppock et al. LithOS: An Operating System for Efficient Machine Learning on GPUs – SOSP'25

CUDA Summary

- **Execution semantics**
  - Partitioning of problem into thread blocks is in the spirit of the data-parallel model (intended to be machine independent: system schedules blocks onto any number of cores)
  - Threads in a thread block actually do run concurrently (they have to, since they cooperate)
    - Inside a single thread block: SPMD shared address space programming
  - There are subtle, but notable differences between these models of execution. Make sure you understand it. (And ask yourself what semantics are being used whenever you encounter a parallel programming system)
- **Memory semantics**
  - Distributed address space: host/device memories
  - Thread local/block shared/global variables within device memory
    - Loads/stores move data between them (so it is correct to think about local/shared/global memory as being distinct address spaces)
- **Key implementation details:**
  - Threads in a thread block are scheduled onto same GPU core to allow fast communication through shared memory
  - Threads in a thread block are are grouped into warps for SIMD execution on GPU hardware

Outline

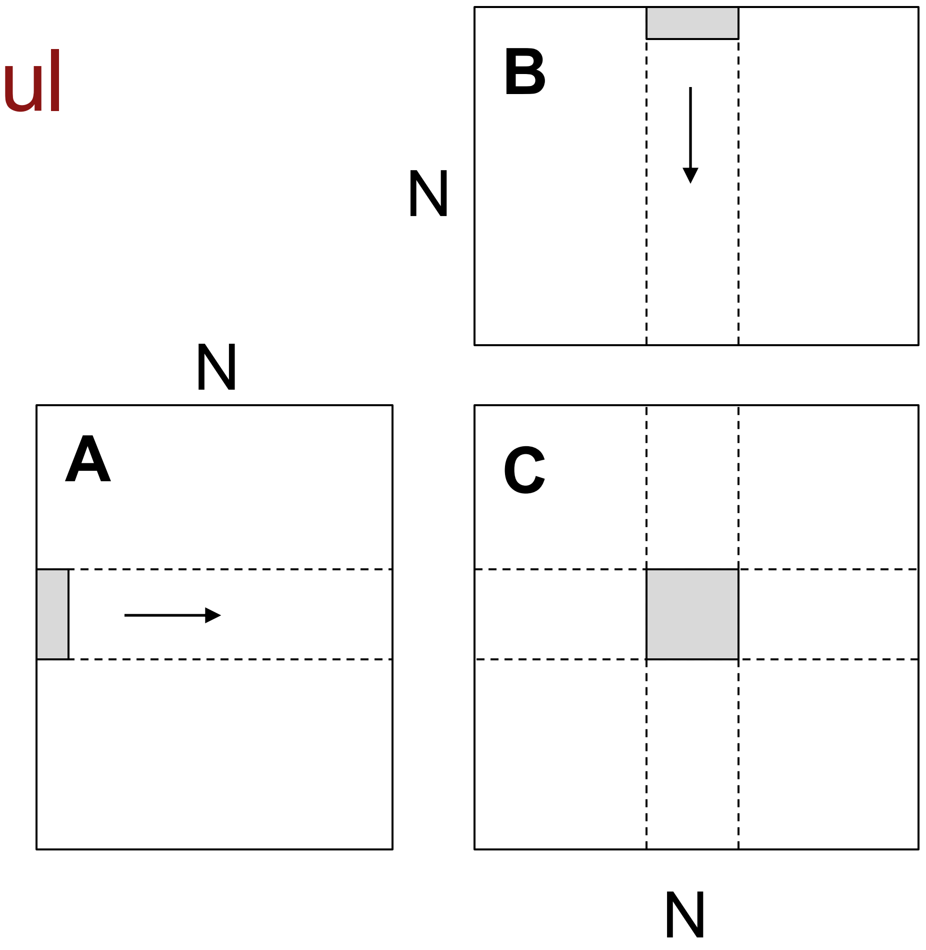
- CUDA Programming Abstractions
- CUDA Implementation on Modern GPUs
- Cast Study 1: Matrix Multiplication in CUDA
- Cast Study 2: Parallel Reduction in CUDA

Strawman Implementation of Matmul

- Compute $C = A \times B$
- Each thread computes one element

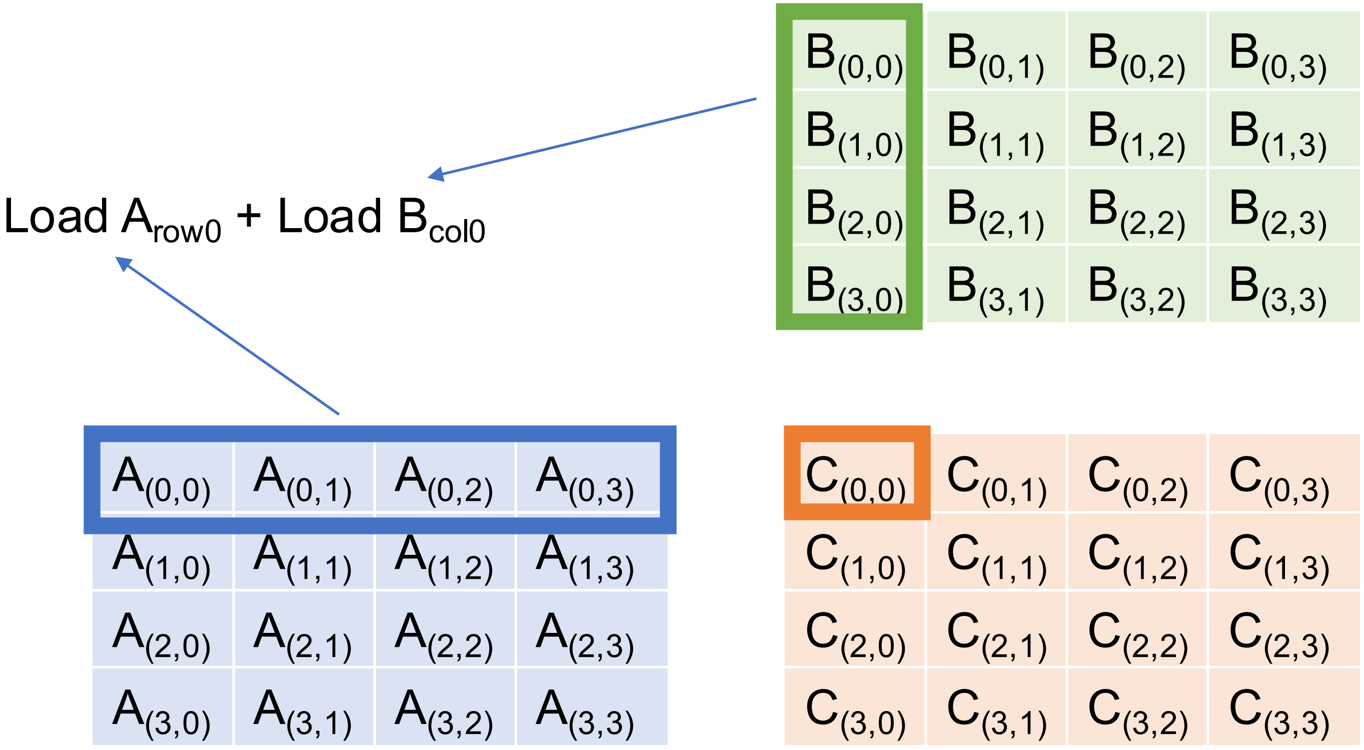
```
int N = 1024;
dim3 threadsPerBlock(32, 32, 1);
dim3 numBlocks(N/32, N/32, 1);

matmul<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  result = 0;
  for (int k = 0; k < N; ++k) {
    result += A[x][k] * B[k][y];
  }
  C[x][y] = result;
}
```

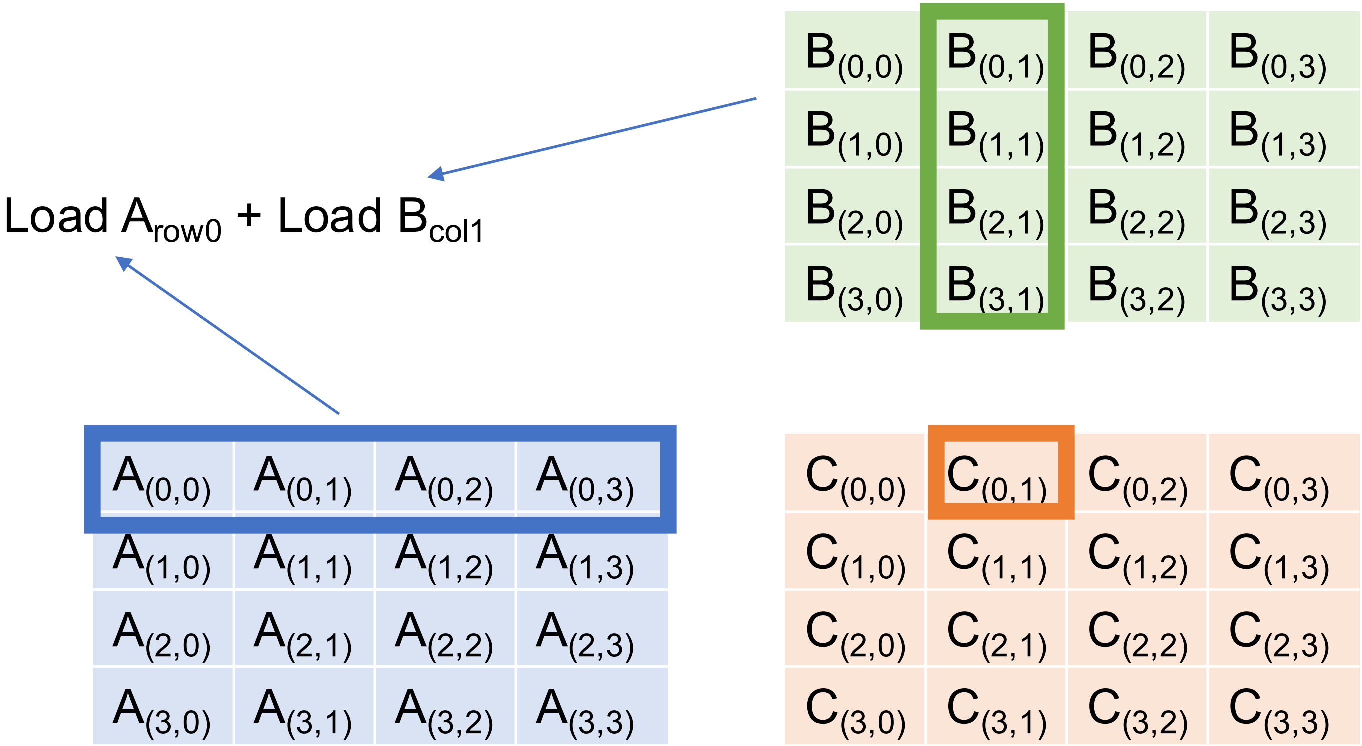


Global memory access per thread: 2*N
Number of threads: N^2
Total global memory access: $2N^3$

Example


Load A_{row0} + Load B_{col0}
$B_{(0,0)}$ $B_{(0,1)}$ $B_{(0,2)}$ $B_{(0,3)}$
$B_{(1,0)}$ $B_{(1,1)}$ $B_{(1,2)}$ $B_{(1,3)}$
$B_{(2,0)}$ $B_{(2,1)}$ $B_{(2,2)}$ $B_{(2,3)}$
$B_{(3,0)}$ $B_{(3,1)}$ $B_{(3,2)}$ $B_{(3,3)}$
$A_{(0,0)}$ $A_{(0,1)}$ $A_{(0,2)}$ $A_{(0,3)}$
$A_{(1,0)}$ $A_{(1,1)}$ $A_{(1,2)}$ $A_{(1,3)}$
$A_{(2,0)}$ $A_{(2,1)}$ $A_{(2,2)}$ $A_{(2,3)}$
$A_{(3,0)}$ $A_{(3,1)}$ $A_{(3,2)}$ $A_{(3,3)}$
$C_{(0,0)}$ $C_{(0,1)}$ $C_{(0,2)}$ $C_{(0,3)}$
$C_{(1,0)}$ $C_{(1,1)}$ $C_{(1,2)}$ $C_{(1,3)}$
$C_{(2,0)}$ $C_{(2,1)}$ $C_{(2,2)}$ $C_{(2,3)}$
$C_{(3,0)}$ $C_{(3,1)}$ $C_{(3,2)}$ $C_{(3,3)}$

Example


Load A_row0 + Load B_col1
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)
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)
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)

Example

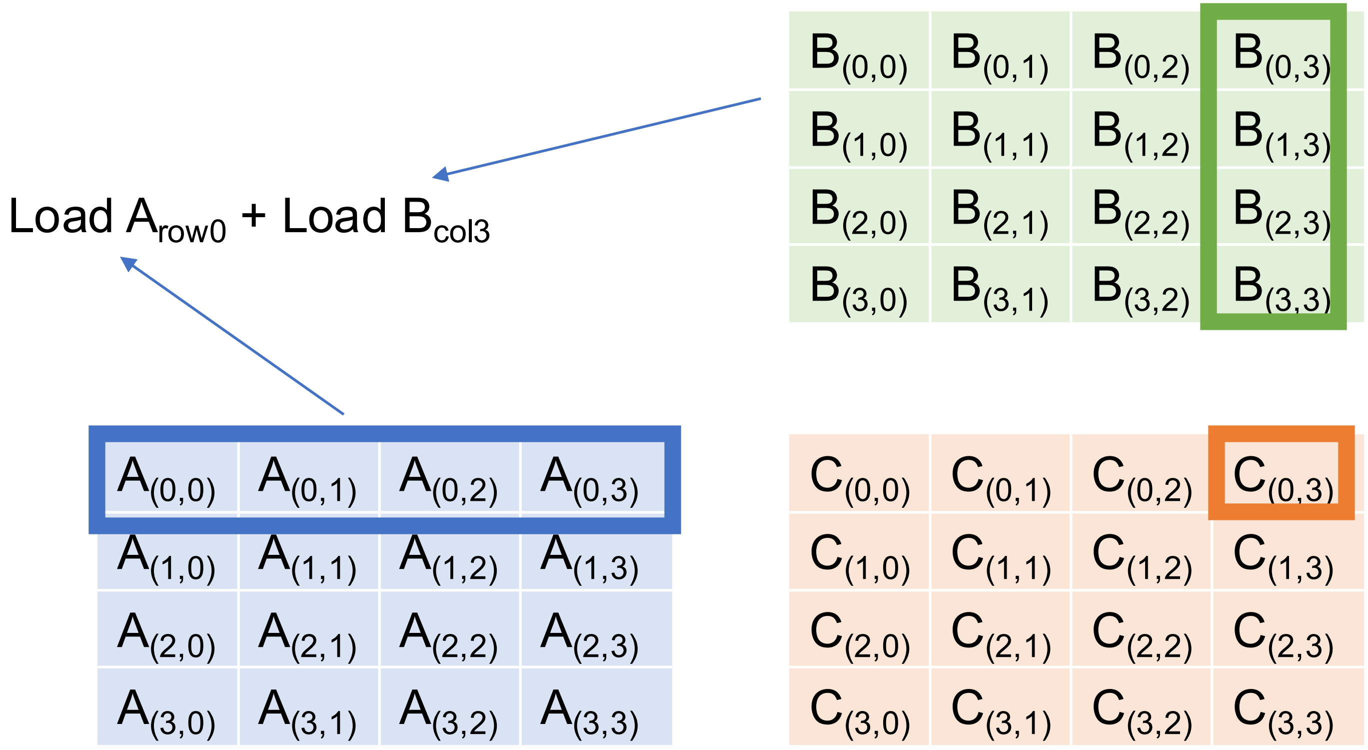
Load A_{row0} + Load B_{col2}

$B_{(0,0)}$	$B_{(0,1)}$	$B_{(0,2)}$	$B_{(0,3)}$
$B_{(1,0)}$	$B_{(1,1)}$	$B_{(1,2)}$	$B_{(1,3)}$
$B_{(2,0)}$	$B_{(2,1)}$	$B_{(2,2)}$	$B_{(2,3)}$
$B_{(3,0)}$	$B_{(3,1)}$	$B_{(3,2)}$	$B_{(3,3)}$

$A_{(0,0)}$	$A_{(0,1)}$	$A_{(0,2)}$	$A_{(0,3)}$
$A_{(1,0)}$	$A_{(1,1)}$	$A_{(1,2)}$	$A_{(1,3)}$
$A_{(2,0)}$	$A_{(2,1)}$	$A_{(2,2)}$	$A_{(2,3)}$
$A_{(3,0)}$	$A_{(3,1)}$	$A_{(3,2)}$	$A_{(3,3)}$

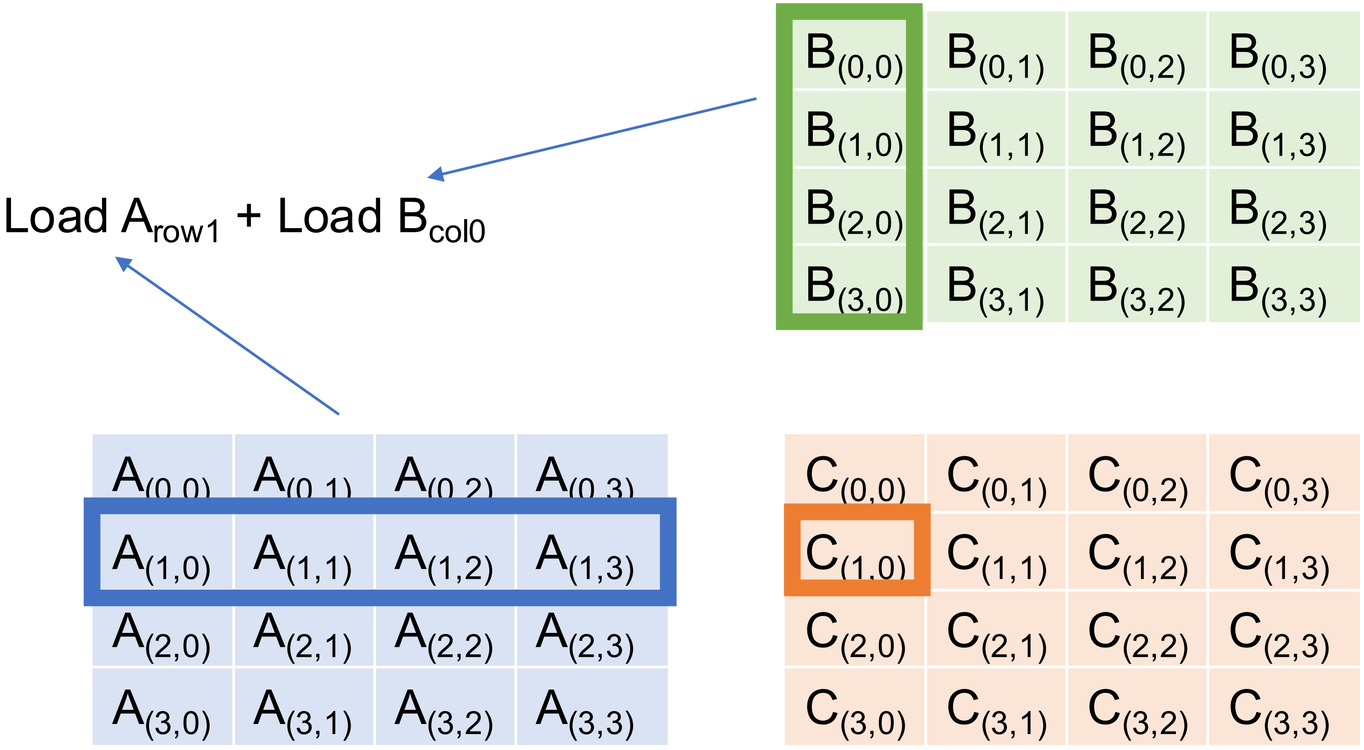
$C_{(0,0)}$	$C_{(0,1)}$	$C_{(0,2)}$	$C_{(0,3)}$
$C_{(1,0)}$	$C_{(1,1)}$	$C_{(1,2)}$	$C_{(1,3)}$
$C_{(2,0)}$	$C_{(2,1)}$	$C_{(2,2)}$	$C_{(2,3)}$
$C_{(3,0)}$	$C_{(3,1)}$	$C_{(3,2)}$	$C_{(3,3)}$

Example



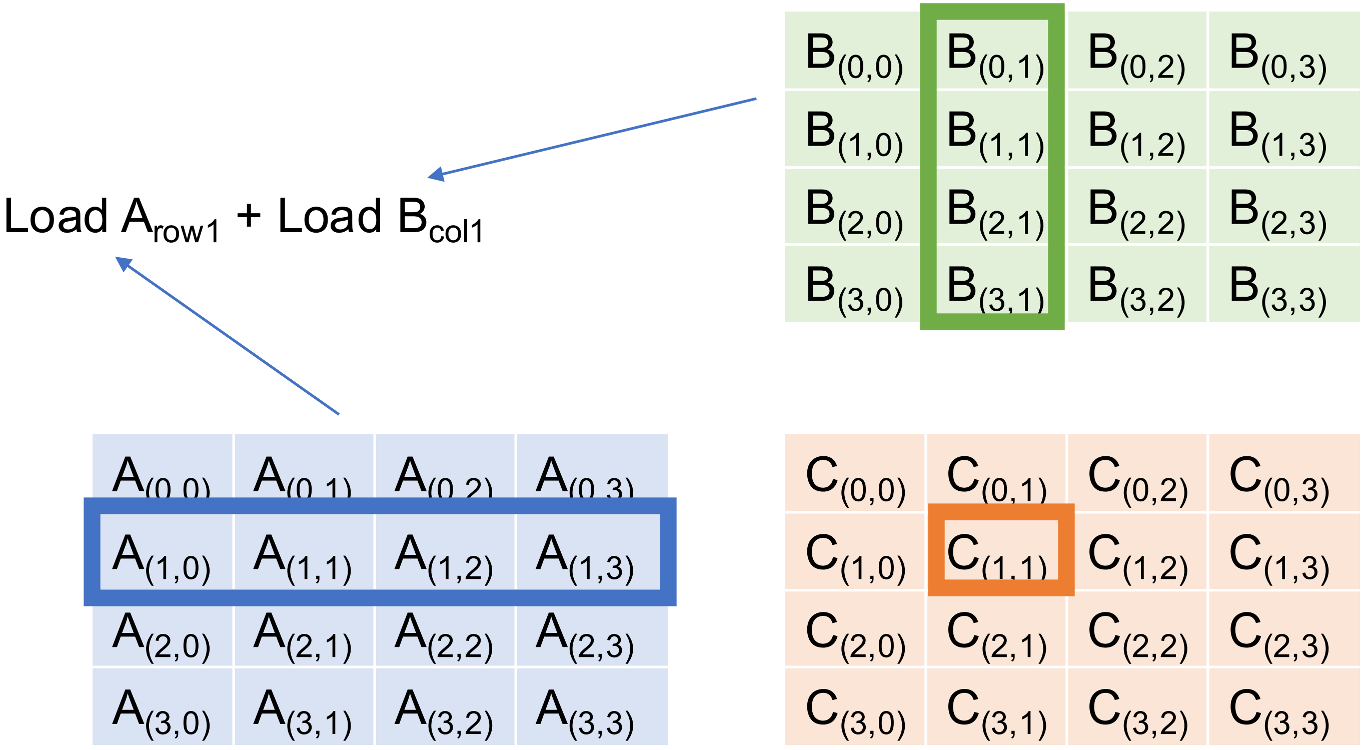
Load A_row0 + Load B_col3
B(0,0) B(0,1) B(0,2) B(0,3)
B(1,0) B(1,1) B(1,2) B(1,3)
B(2,0) B(2,1) B(2,2) B(2,3)
B(3,0) B(3,1) B(3,2) B(3,3)
A(0,0) A(0,1) A(0,2) A(0,3)
A(1,0) A(1,1) A(1,2) A(1,3)
A(2,0) A(2,1) A(2,2) A(2,3)
A(3,0) A(3,1) A(3,2) A(3,3)
C(0,0) C(0,1) C(0,2) C(0,3)
C(1,0) C(1,1) C(1,2) C(1,3)
C(2,0) C(2,1) C(2,2) C(2,3)
C(3,0) C(3,1) C(3,2) C(3,3)

Example



Load A_{row1} + Load B_{col0}
$B_{(0,0)}$ $B_{(0,1)}$ $B_{(0,2)}$ $B_{(0,3)}$
$B_{(1,0)}$ $B_{(1,1)}$ $B_{(1,2)}$ $B_{(1,3)}$
$B_{(2,0)}$ $B_{(2,1)}$ $B_{(2,2)}$ $B_{(2,3)}$
$B_{(3,0)}$ $B_{(3,1)}$ $B_{(3,2)}$ $B_{(3,3)}$
$A_{(0,0)}$ $A_{(0,1)}$ $A_{(0,2)}$ $A_{(0,3)}$
$A_{(1,0)}$ $A_{(1,1)}$ $A_{(1,2)}$ $A_{(1,3)}$
$A_{(2,0)}$ $A_{(2,1)}$ $A_{(2,2)}$ $A_{(2,3)}$
$A_{(3,0)}$ $A_{(3,1)}$ $A_{(3,2)}$ $A_{(3,3)}$
$C_{(0,0)}$ $C_{(0,1)}$ $C_{(0,2)}$ $C_{(0,3)}$
$C_{(1,0)}$ $C_{(1,1)}$ $C_{(1,2)}$ $C_{(1,3)}$
$C_{(2,0)}$ $C_{(2,1)}$ $C_{(2,2)}$ $C_{(2,3)}$
$C_{(3,0)}$ $C_{(3,1)}$ $C_{(3,2)}$ $C_{(3,3)}$

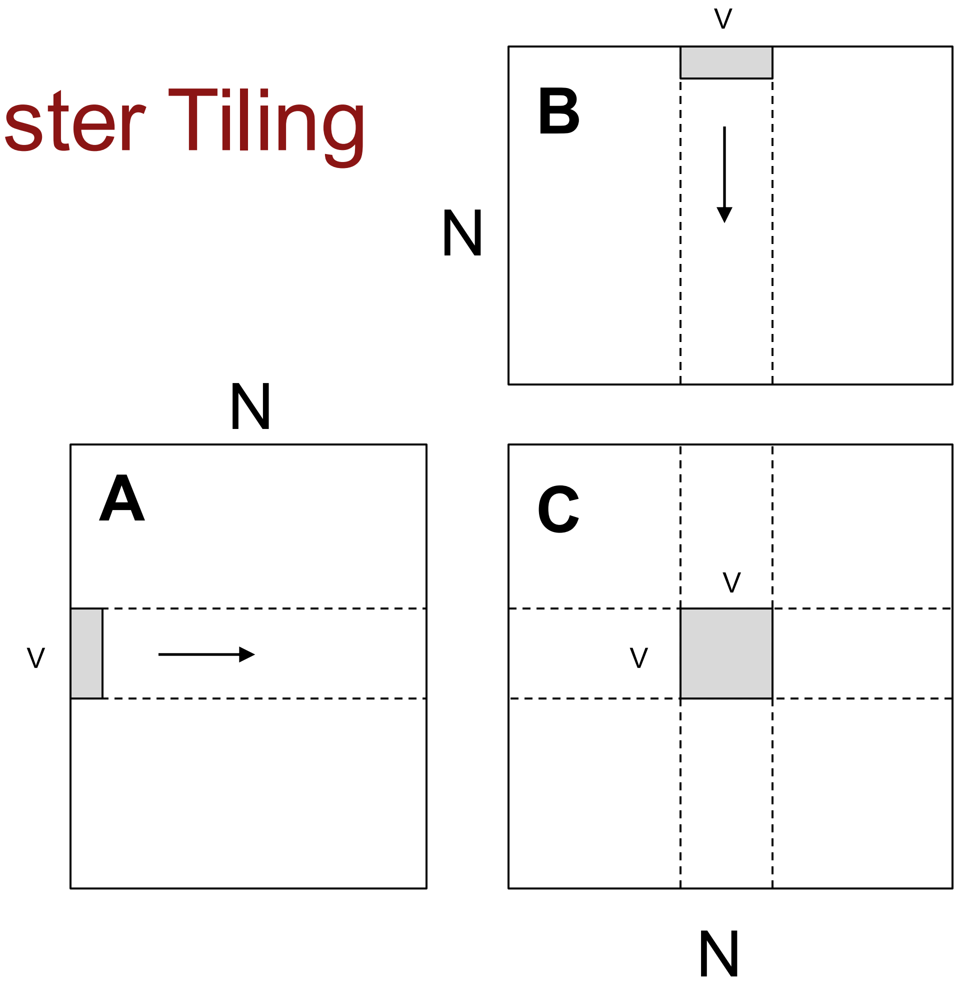
Example


Load A row1 + Load B col1
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)
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)
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)

Optimization 1: Thread-Level Register Tiling

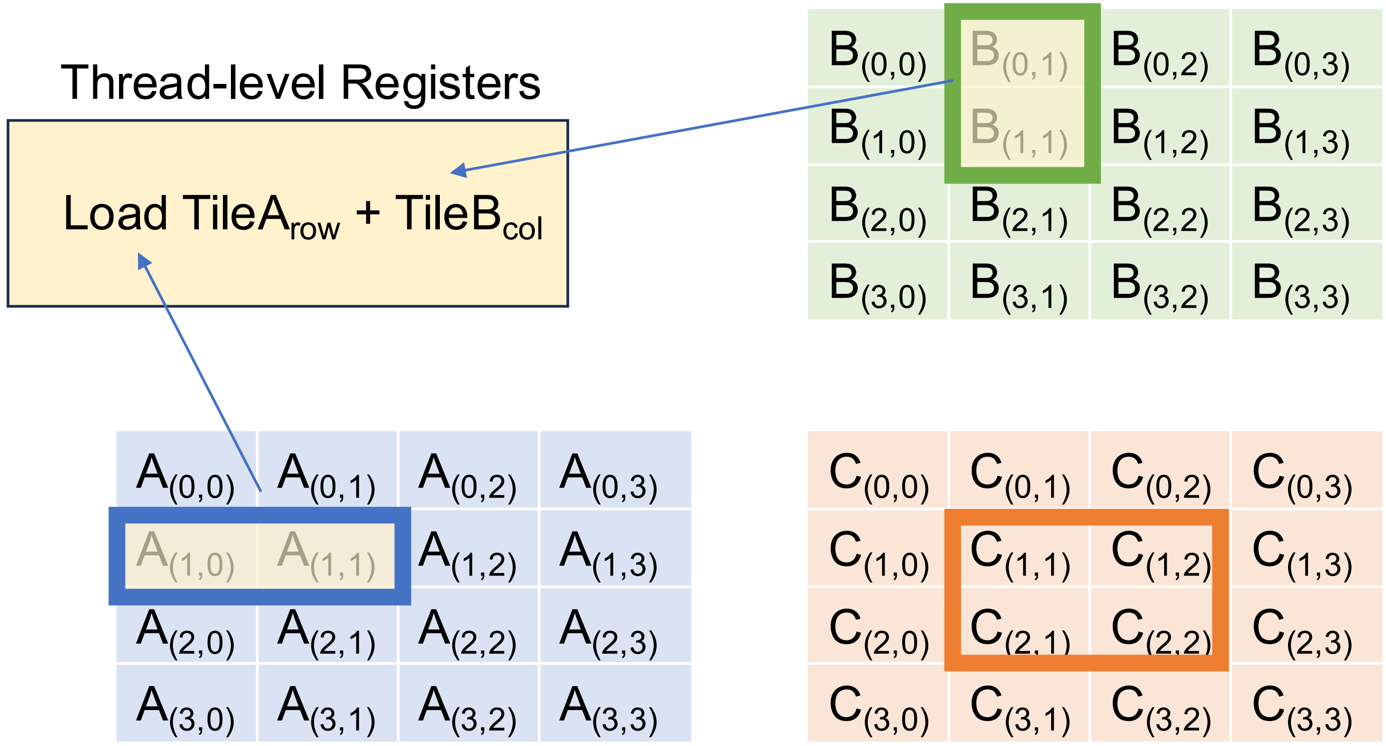
- Compute $C = A \times B$
- Each thread computes a V x V submatrix

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;

  float c[V][V] = {0};
  float a[V], b[V];
  for (int k = 0; k < N; ++k) {
    a[:] = A[xbase*V : xbase*V + V, k];
    b[:] = B[k, ybase*V : ybase*V + V];
    for (int y = 0; y < V; ++y) {
      for (int x = 0; x < V; ++x) {
        c[x][y] += a[x] * b[y];
      }
    }
  }
  C[xbase * V : xbase*V + V, ybase*V : ybase*V + V] = c[:];
}
```



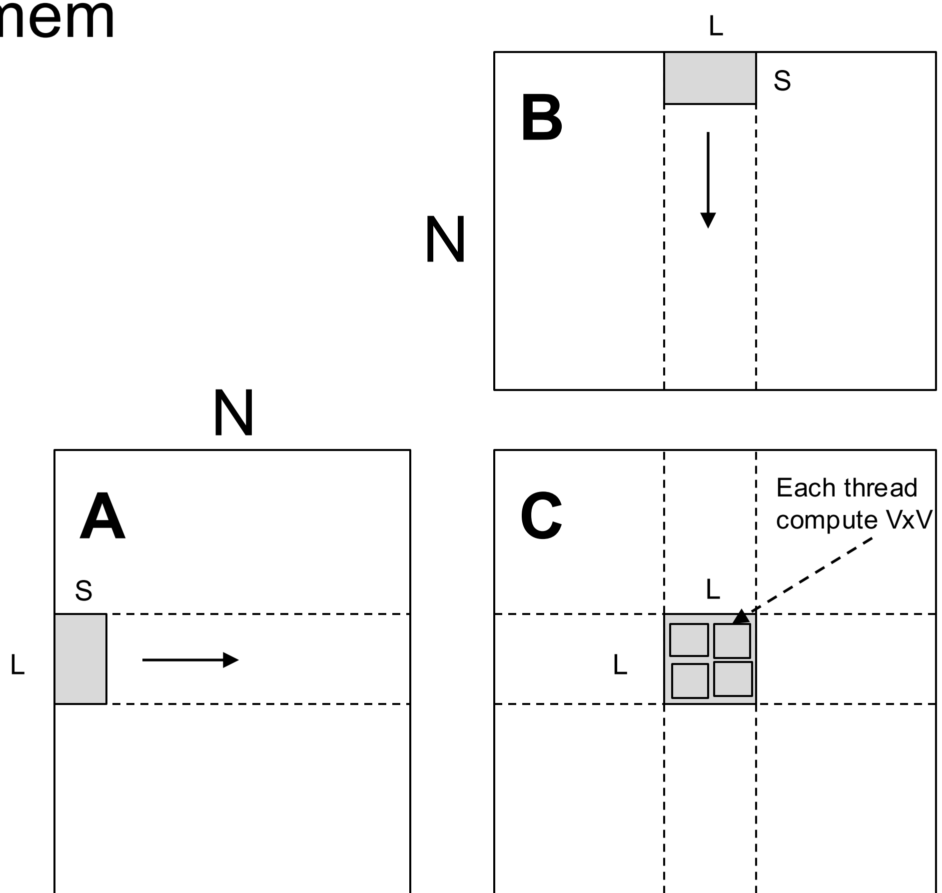
Global memory access per thread: $2NV$
Number of threads: N^2/V^2
Total global memory access: $2N^3/V$

Example


Thread-level Registers
Load TileA$_{row}$ + TileB$_{col}$
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)
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)
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)

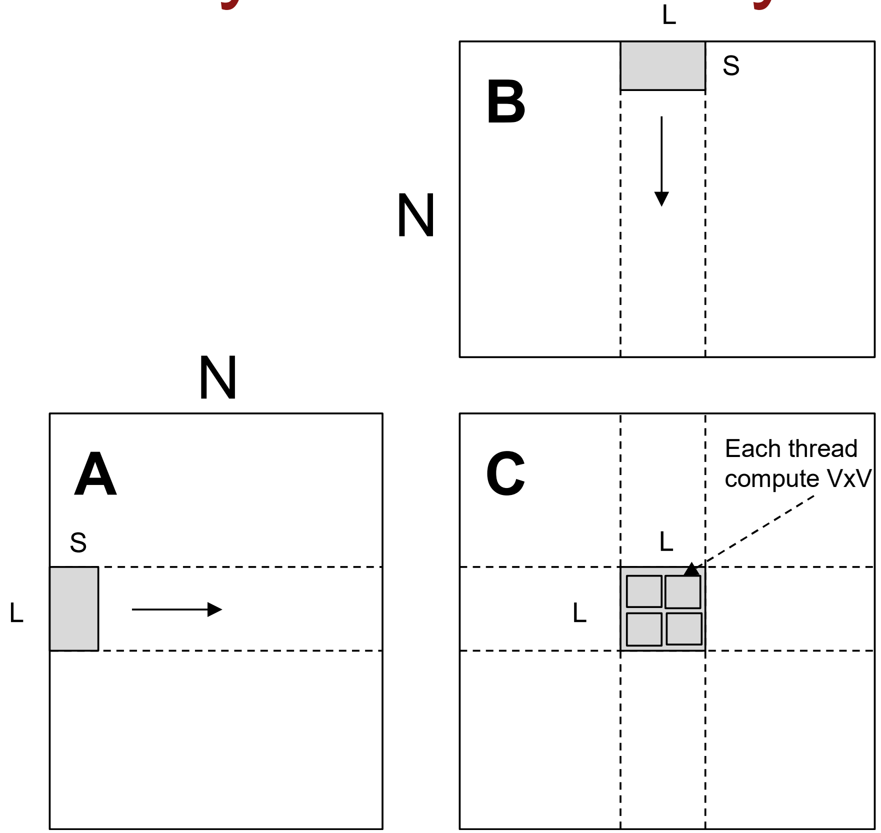
Optimization 2: Block-Level Shared Memory Tiling

- A block computes a L x L submatrix
- A thread computes a V x V submatrix and reuses the matrices in shared mem




```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  __shared__ float sA[S][L], sB[S][L];
  float c[V][V] = {0};
  float a[V], b[V];
  int yblock = blockIdx.y;
  int xblock = blockIdx.x;

  for (int k = 0; k < N; k += S) {
    __syncthreads();
    // needs to be implemented by thread cooperative fetching
    sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];
    sB[:, :] = B[k : k + S, xblock * L : xblock * L + L];
    __syncthreads();
    for (int ki = 0; ki < S; ++ ki) {
      a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];
      b[:] = sB[ki, threadIdx.x * V : threadIdx.x * V + V];
      for (int y = 0; y < V; ++y) {
        for (int x = 0; x < V; ++x) {
          c[x][y] += a[x] * b[y];
        }
      }
    }
  }
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;
  C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];
}
```

Analysis of Memory Reuse

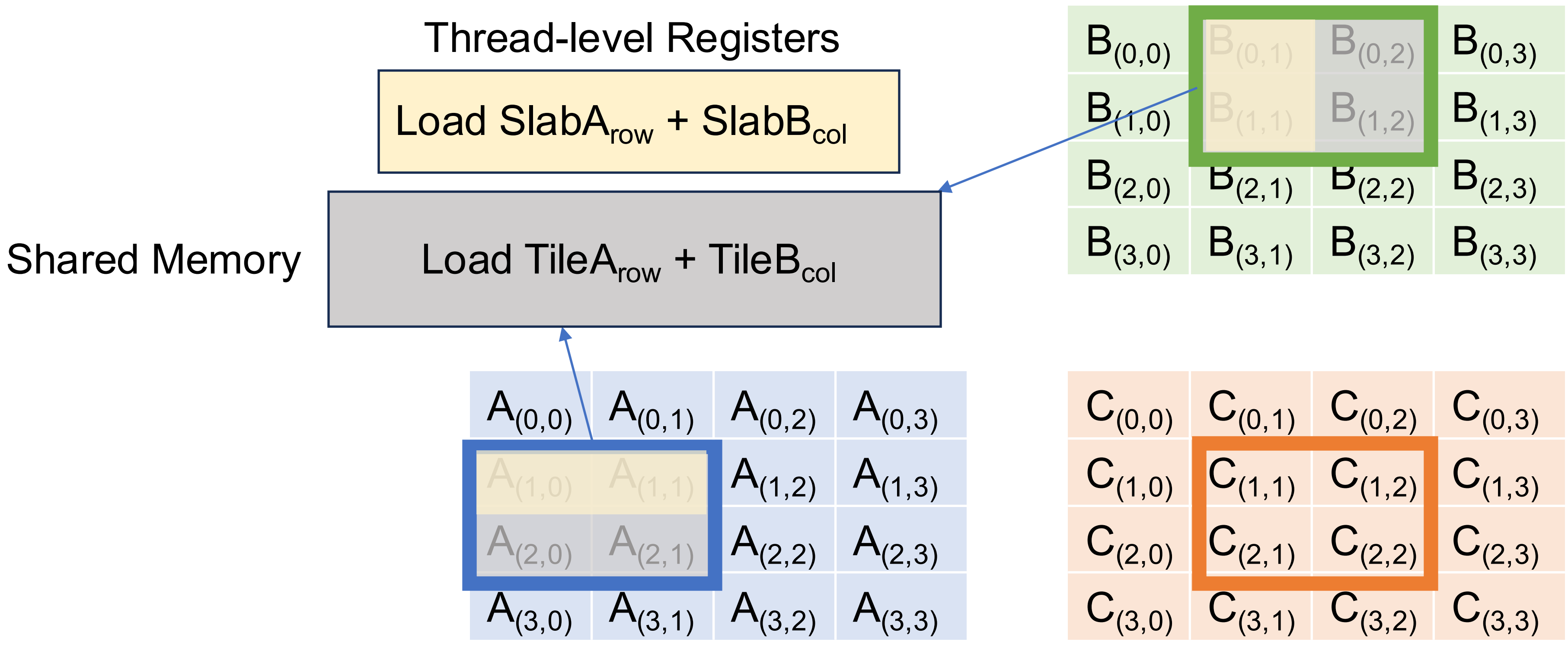


Global memory access per thread block: $2NL$
Number of thread blocks: N^2/L^2
Total global memory access: $2N^3/L$

Shared memory access per thread: $2NV$
Number of threads: N^2/V^2
Total shared memory access: $2N^3/V$

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  __shared__ float sA[S][L], sB[S][L];
  float c[V][V] = {0};
  float a[V], b[V];
  int yblock = blockIdx.y;
  int xblock = blockIdx.x;

  for (int ko = 0; ko < N; ko += S) {
    __syncthreads();
    // needs to be implemented by thread cooperative fetching
    sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];
    sB[:, :] = B[k : k + S, xblock * L : xblock * L + L];
    __syncthreads();
    for (int ki = 0; ki < S; ++ ki) {
      a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];
      b[:] = sB[ki, threadIdx.x * V : threadIdx.x * V + V];
      for (int y = 0; y < V; ++y) {
        for (int x = 0; x < V; ++x) {
          c[x][y] += a[x] * b[y];
        }
      }
    }
  }
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;
  C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];
}
```

Example



Thread-level Registers
Load SlabA$_{row}$ + SlabB$_{col}$
Shared Memory
Load TileA$_{row}$ + TileB$_{col}$
$B_{(0,0)}$ $B_{(0,1)}$ $B_{(0,2)}$ $B_{(0,3)}$
$B_{(1,0)}$ $B_{(1,1)}$ $B_{(1,2)}$ $B_{(1,3)}$
$B_{(2,0)}$ $B_{(2,1)}$ $B_{(2,2)}$ $B_{(2,3)}$
$B_{(3,0)}$ $B_{(3,1)}$ $B_{(3,2)}$ $B_{(3,3)}$
$A_{(0,0)}$ $A_{(0,1)}$ $A_{(0,2)}$ $A_{(0,3)}$
$A_{(1,0)}$ $A_{(1,1)}$ $A_{(1,2)}$ $A_{(1,3)}$
$A_{(2,0)}$ $A_{(2,1)}$ $A_{(2,2)}$ $A_{(2,3)}$
$A_{(3,0)}$ $A_{(3,1)}$ $A_{(3,2)}$ $A_{(3,3)}$
$C_{(0,0)}$ $C_{(0,1)}$ $C_{(0,2)}$ $C_{(0,3)}$
$C_{(1,0)}$ $C_{(1,1)}$ $C_{(1,2)}$ $C_{(1,3)}$
$C_{(2,0)}$ $C_{(2,1)}$ $C_{(2,2)}$ $C_{(2,3)}$
$C_{(3,0)}$ $C_{(3,1)}$ $C_{(3,2)}$ $C_{(3,3)}$

Coorperative Fetching

```
sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];
```

↓

```
int nthreads = blockDim.y * blockDim.x;
int tid = threadIdx.y * blockDim.x + threadIdx.x;

for(int j = 0; j < L * S / nthreads; ++j) {
  int y = (j * nthreads + tid) / L;
  int x = (j * nthreads + tid) % L;
  s[y, x] = A[k + y, yblock * L + x];
}
```

Thread IDs

	L = 3	**S = 2**
tid	**y = tid / L**	**x = tid % L**
0	0	0
1	0	1
2	0	2
3	1	0
4	1	1
5	1	2

		S	
		0	1
	0	(0, 0)	(0, 1)
L	1	(1, 0)	(1, 1)
	2	(2, 0)	(2, 1)

Slab Accesses per Thread

Practice

For a small N, perform the calculations of matrix multiply with pen & paper

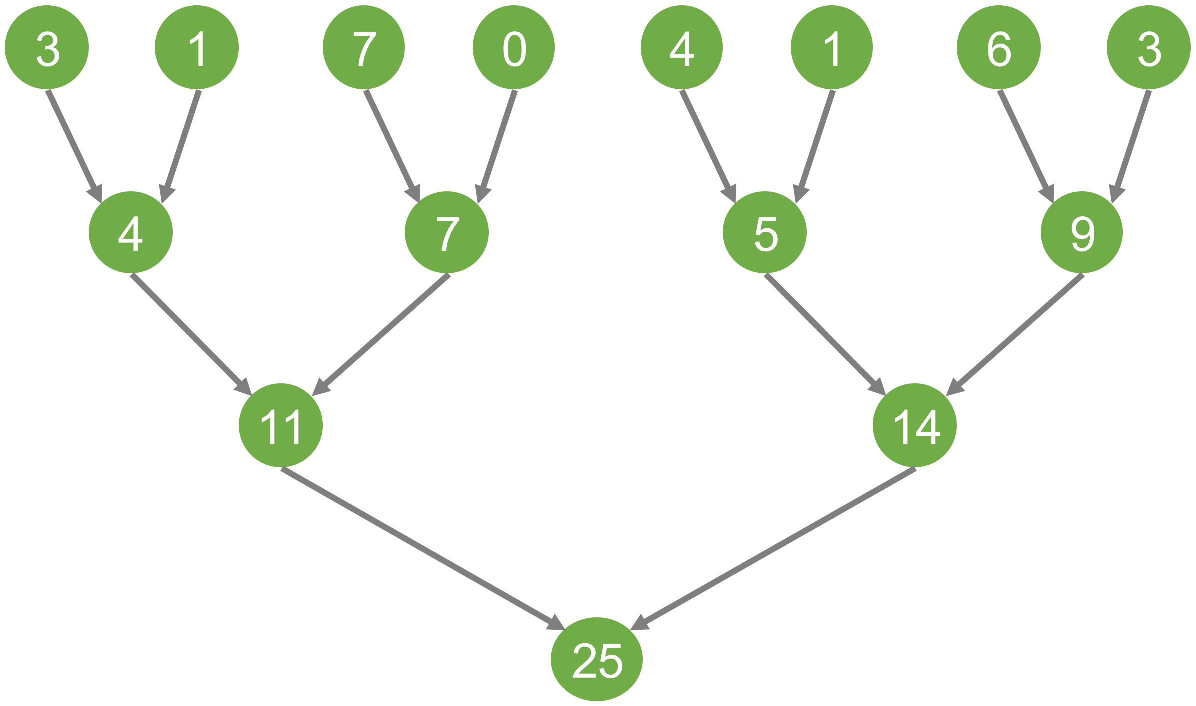
- Add optimization 1
- Add optimization 2
- Calculate memory reuse for all cases

Outline

- CUDA Programming Abstractions
- CUDA Implementation on Modern GPUs
- Cast Study 1: Matrix Multiplication in CUDA
- Cast Study 2: Parallel Reduction in CUDA

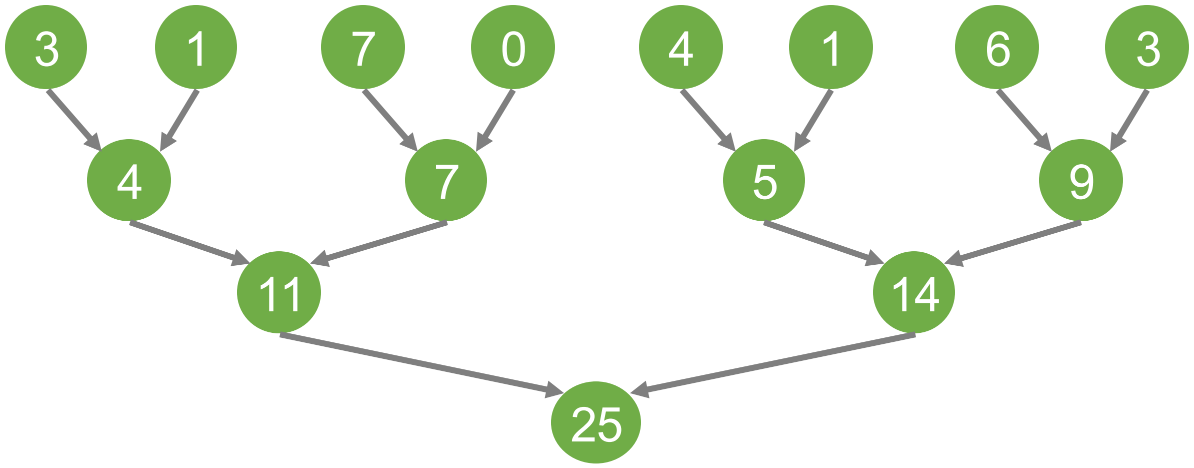
Parallel Reduction

- Common and important primitive used by many MLSys operators: normalization, softmax, etc
- Tree-based approach to reduce elements within each thread block



Challenges of Parallel Reduction in CUDA

- Task: for a large array of n elements, compute $\sum_{i=1}^{n} A[i]$
- To achieve high GPU utilization
  - Need to use multiple thread blocks (since a block is assigned to one SMM)
  - Each thread block reduces a portion of the array
- How to communicate partial results between thread blocks?



Problem: CUDA has no Global Synchronization

Recall CUDA assumption: thread blocks can be executed in any order, cannot sync between them

Why?

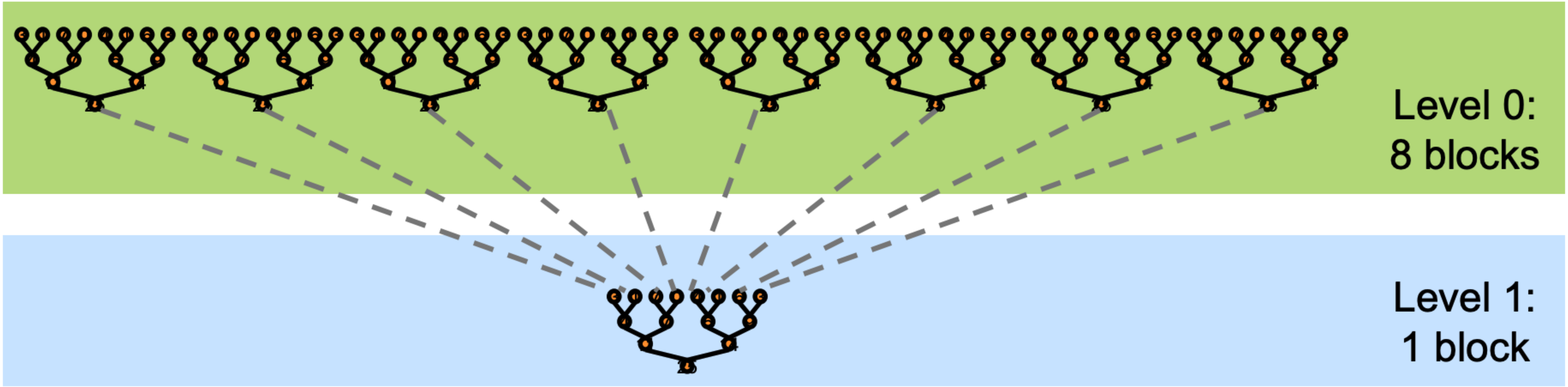
- Expensive to build hardware for GPUs with high processor count
- Potential deadlock when # blocks > # multiprocessors * # resident blocks

Solution: decompose into multiple kernels

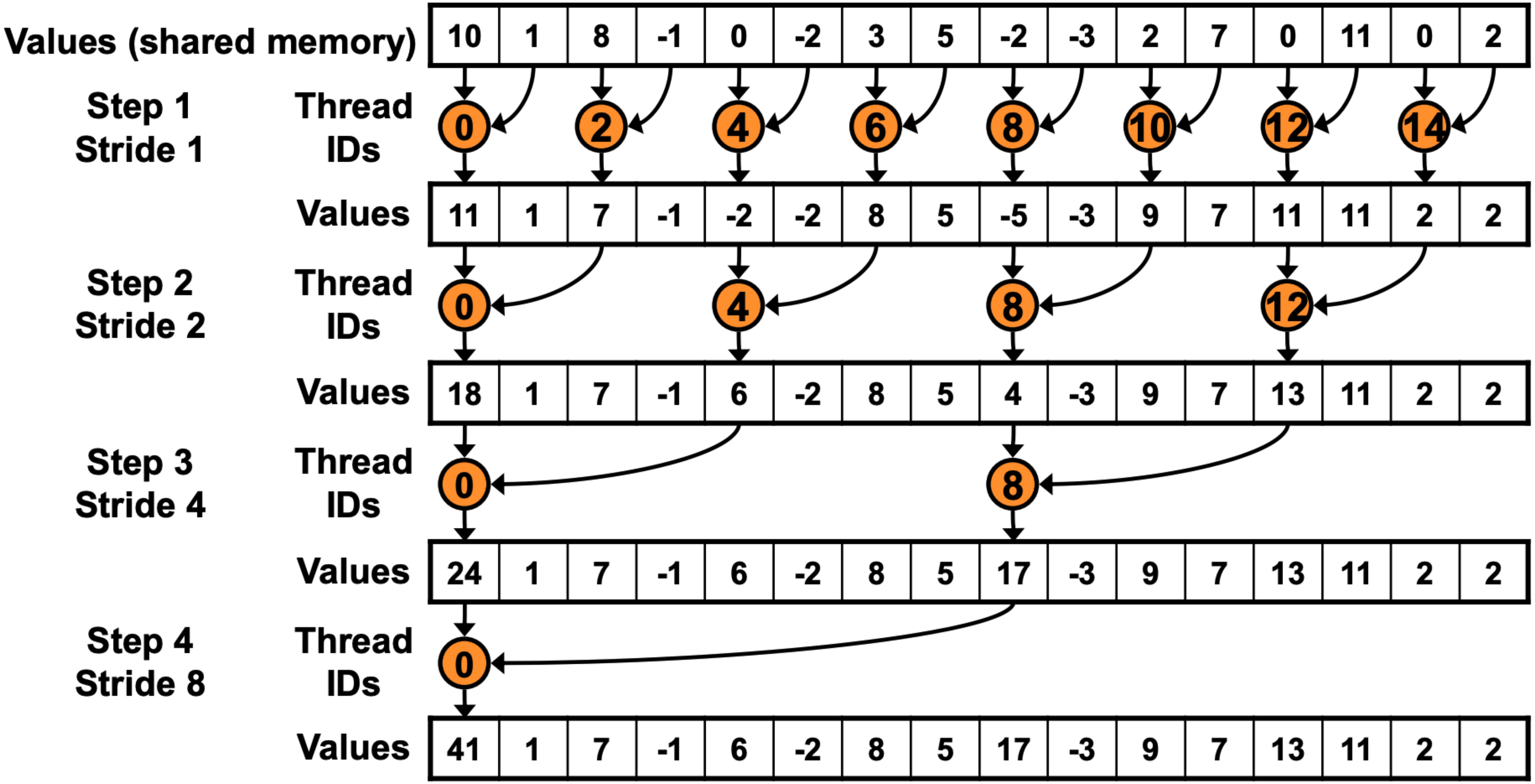
- Kernel launch serves a global synchronization
- Kernel launch has very low hardware/software overhead

Solution: Kernel Decomposition

- Avoid global synchronization by decompose computation into multiple kernel invocations
- Code for all levels is the same



Version 1: Interleaved Addressing



Version 1: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
   if (tid % (2*s) == 0) {
     sdata[tid] += sdata[tid + s];
   }
   __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Why we need the two __syncthreads?

Version 1: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

4M elements
Time: 8.054 ms
BW usage: 2.083 GB/s

Problem: highly divergent warps

Version 1: Divergent Warps

```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0)
    sdata[tid] += sdata[tid + s];
}
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s=1	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
s=2	T	F	F	F	T	F	F	F	T	F	F	F	T	F	F	F
s=4	T	F	F	F	F	F	F	F	T	F	F	F	F	F	F	F

Version 2: Strided Index and Non-divergent warp

Original Version

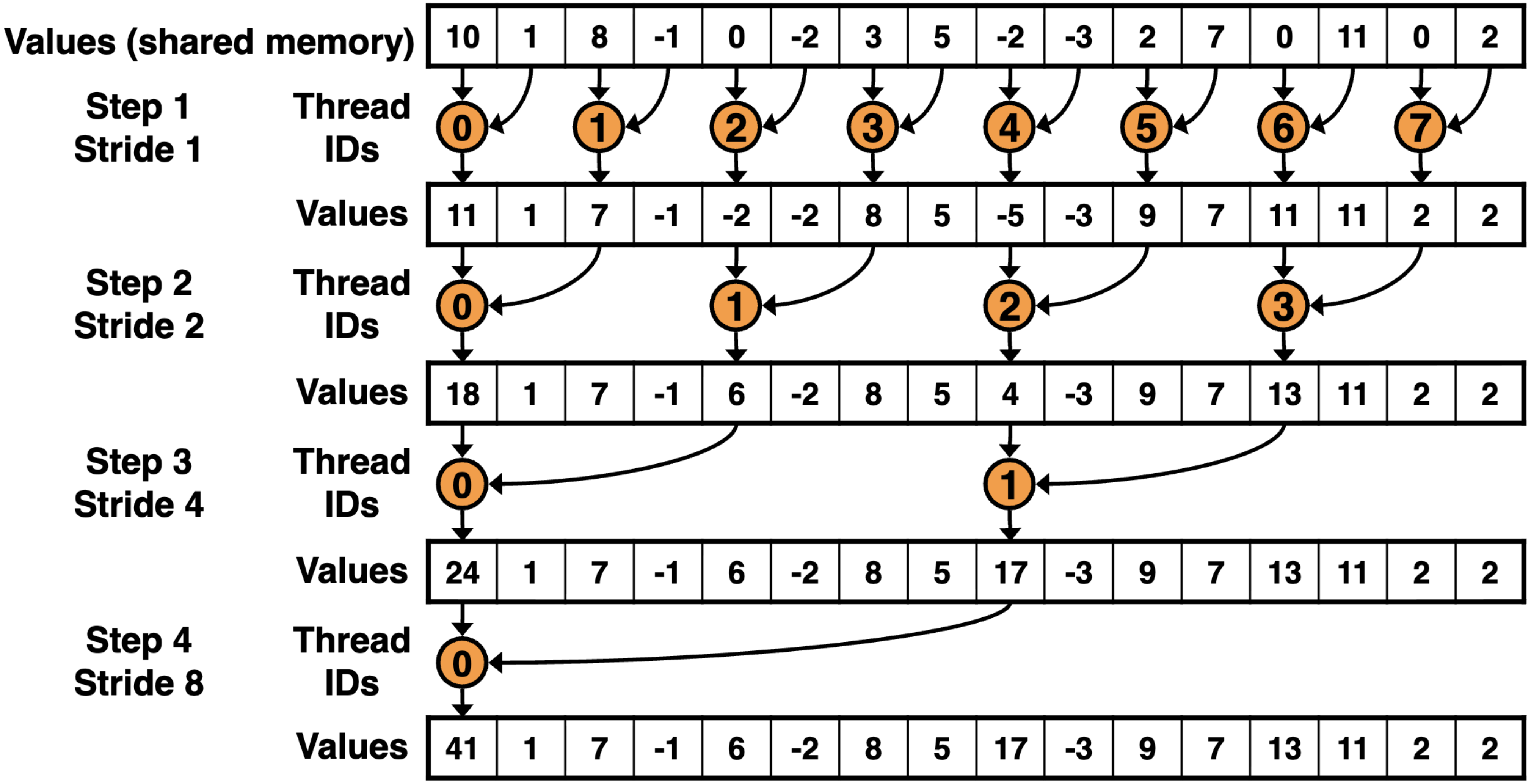
```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

Replace divergent branch with strided index and non-divergent branch

Optimized Version

```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 *s * threadIdx.x;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

Version 1: Interleaved Addressing



Version 2: Strided Index and Non-divergent warp

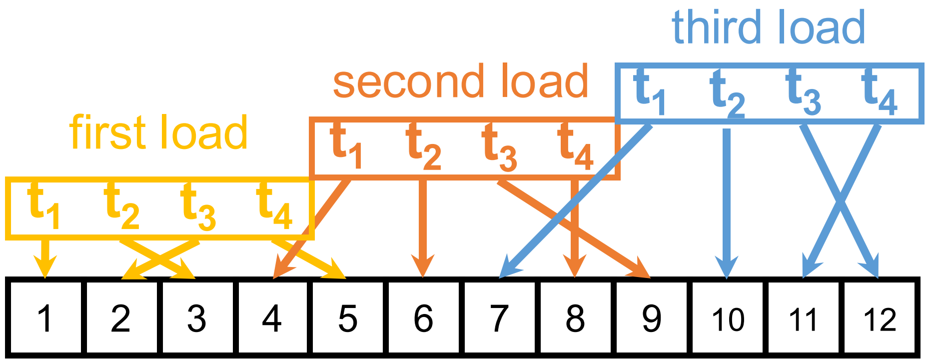
```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 *s * threadIdx.x;
  if (index < blockDim.x)
    sdata[index] += sdata[index + s];
}
```

4M elements (2.3x speedup)
Time: 3.4 ms
BW usage: 4.8 GB/s

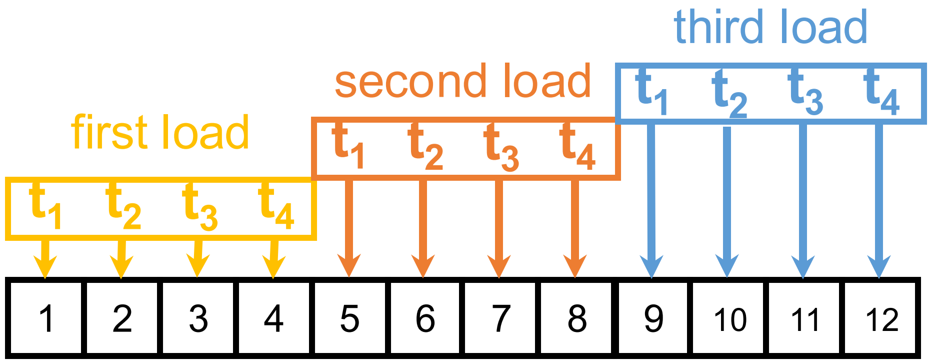
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s=1	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
s=2	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F
s=4	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F

Coalesced Memory Access

- Multiple GPU threads access **consecutive** memory addresses
- Maximize GPU memory usage
- Avoid Bank conflicts

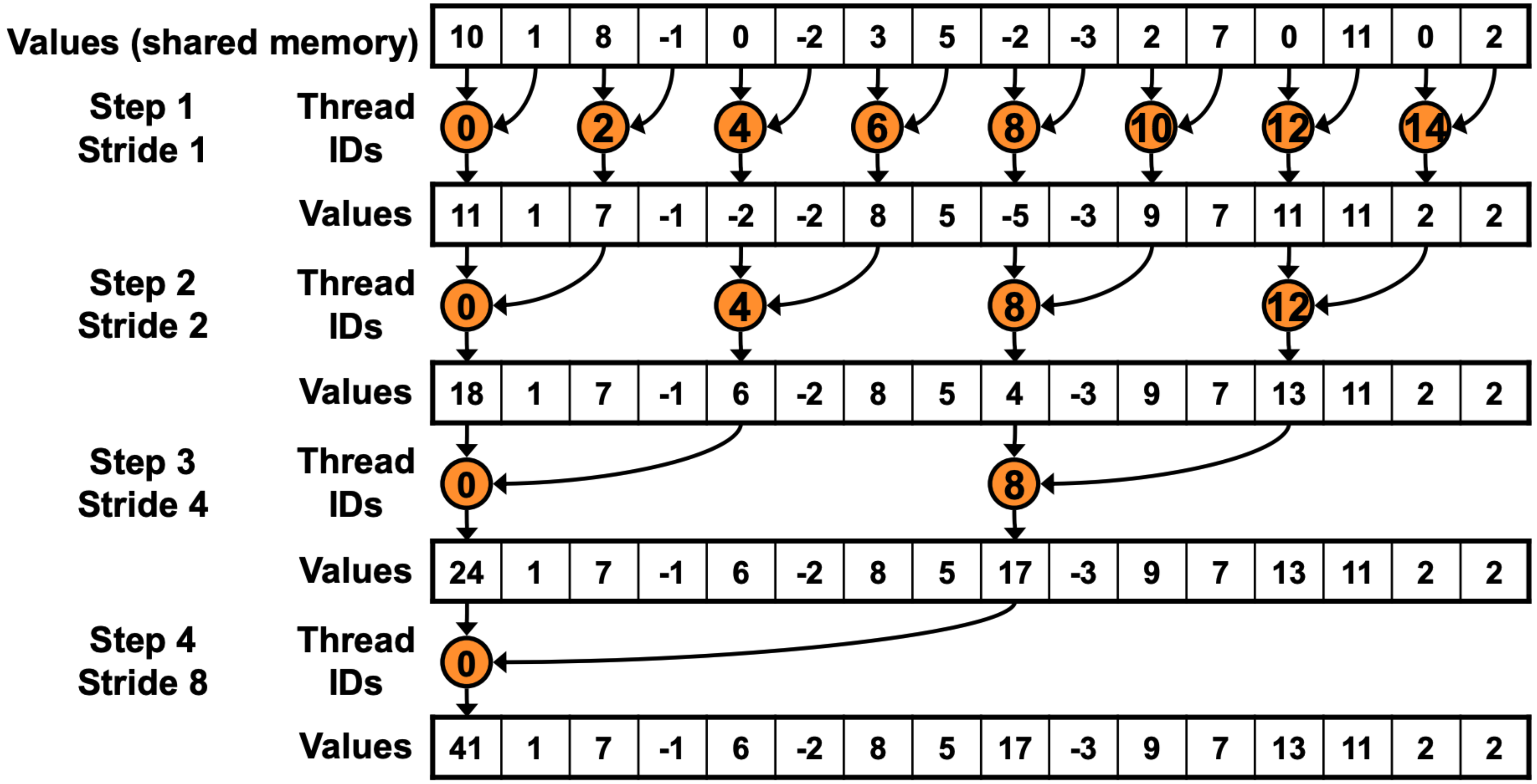


Non-coalesced access (suboptimal usage)



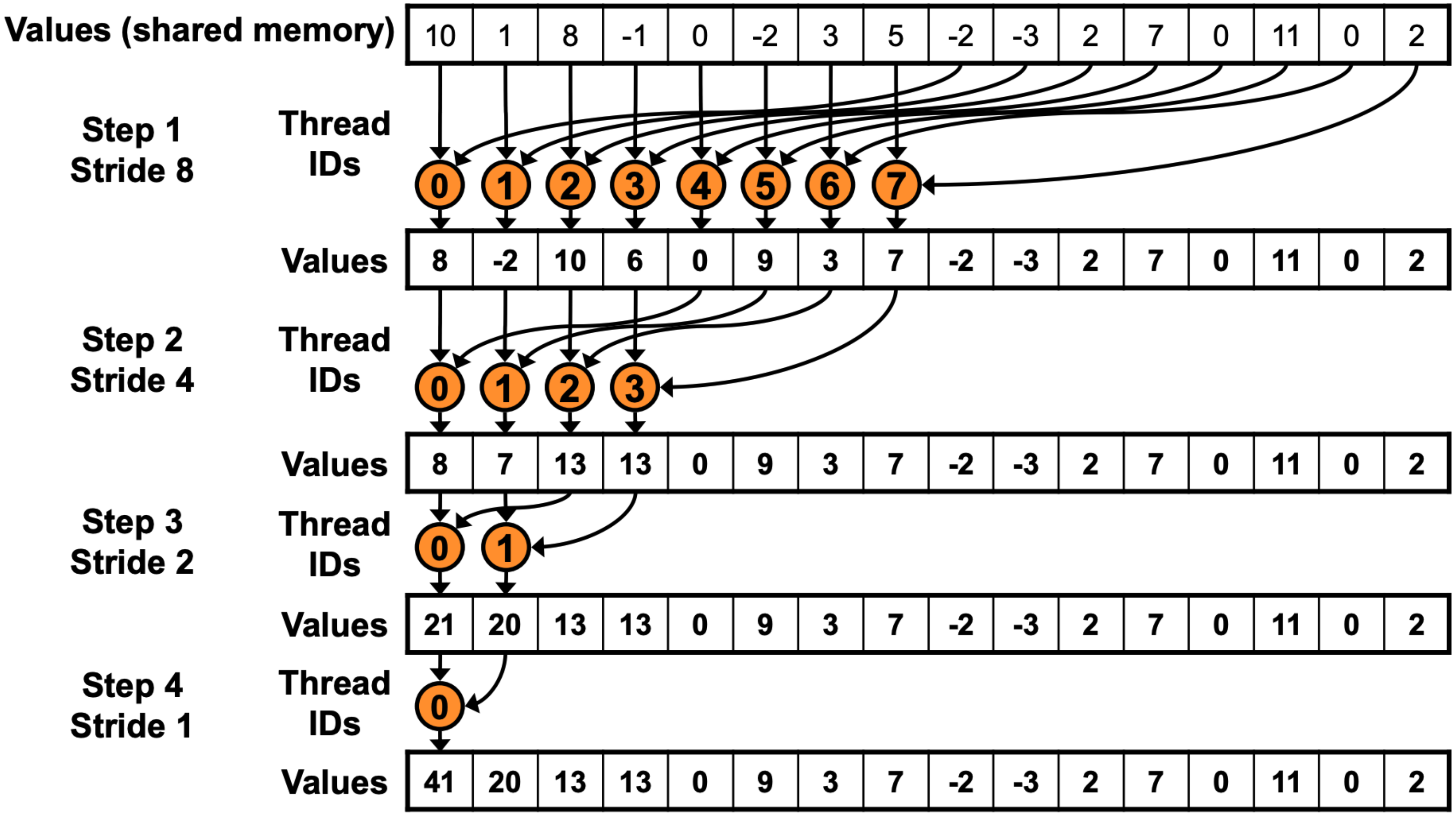
coalesced access (optimal usage)

Version 1: Interleaved Addressing



Suboptimal memory accesses

Version 2: Sequential Addressing



Fully coalesced memory access

Version 2: Sequential Addressing

Original Version

```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 *s * threadIdx.x;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

4M elements (4.6x speedup)
Time: 1.7 ms
BW usage: 9.7 GB/s

Replace strided index with reversed loop and threadId-based index

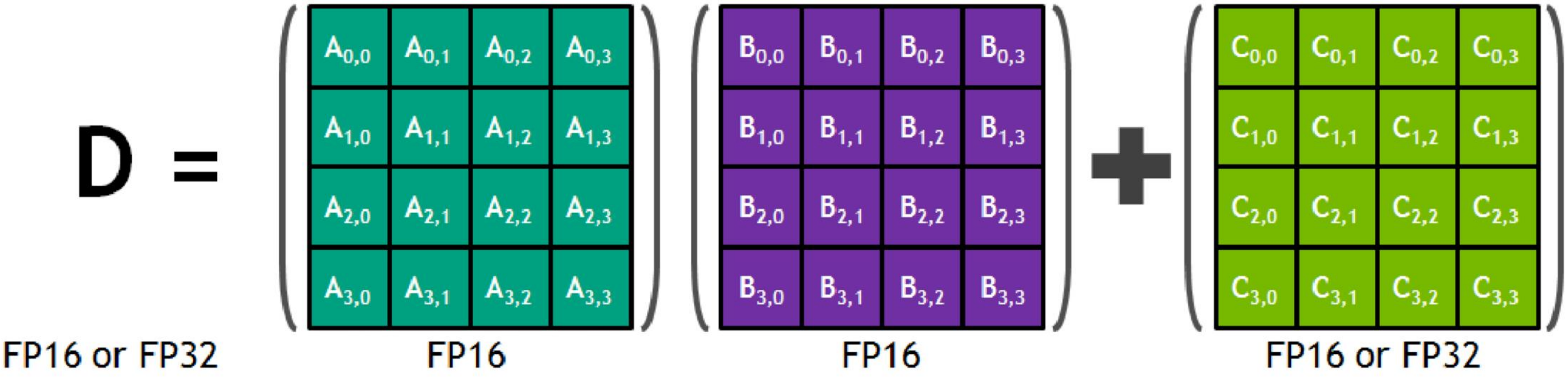
Original Version

```
for(unsigned int s=blockDim.x / 2; s > 0; s /= 2) {
  if (threadIdx.x < s) {
    sdata[threadIdx.x] += sdata[threadIdx.x + s];
  }
  __syncthreads();
}
```

H100 Architecture with Tensor Cores

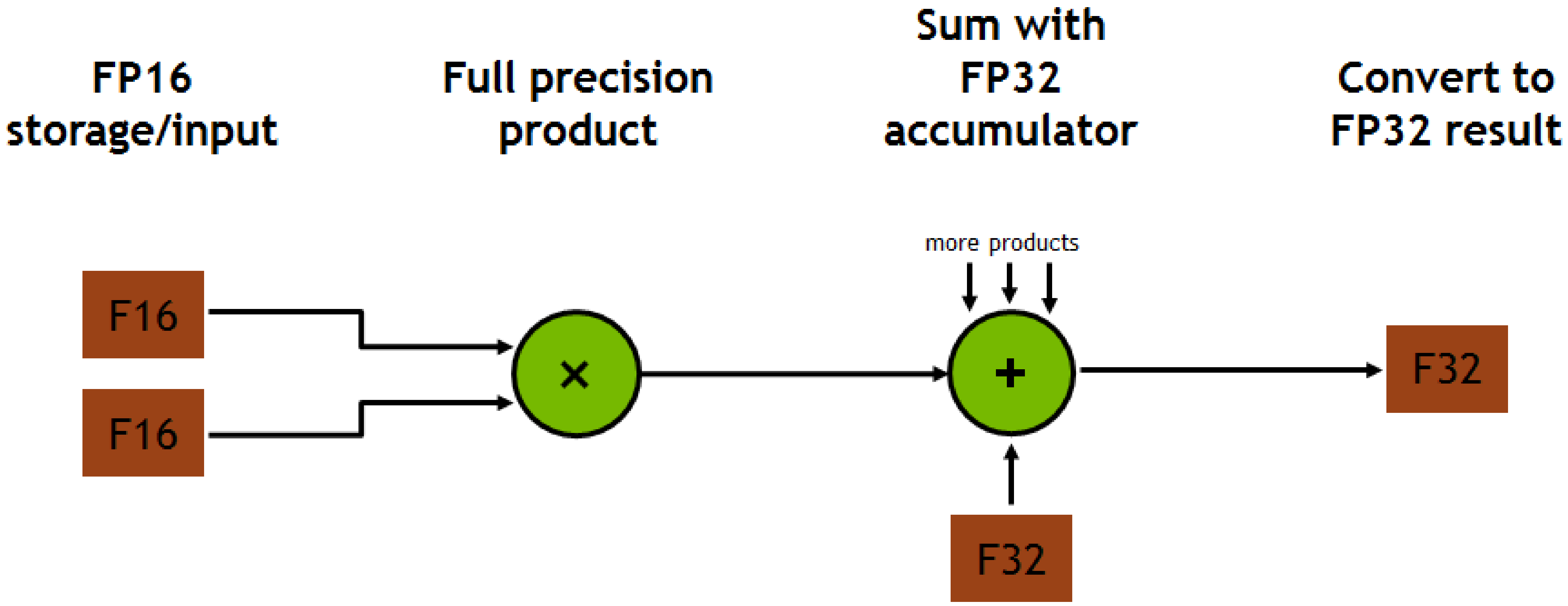
Tensor Cores

- Matrix multiplication unit in SMM

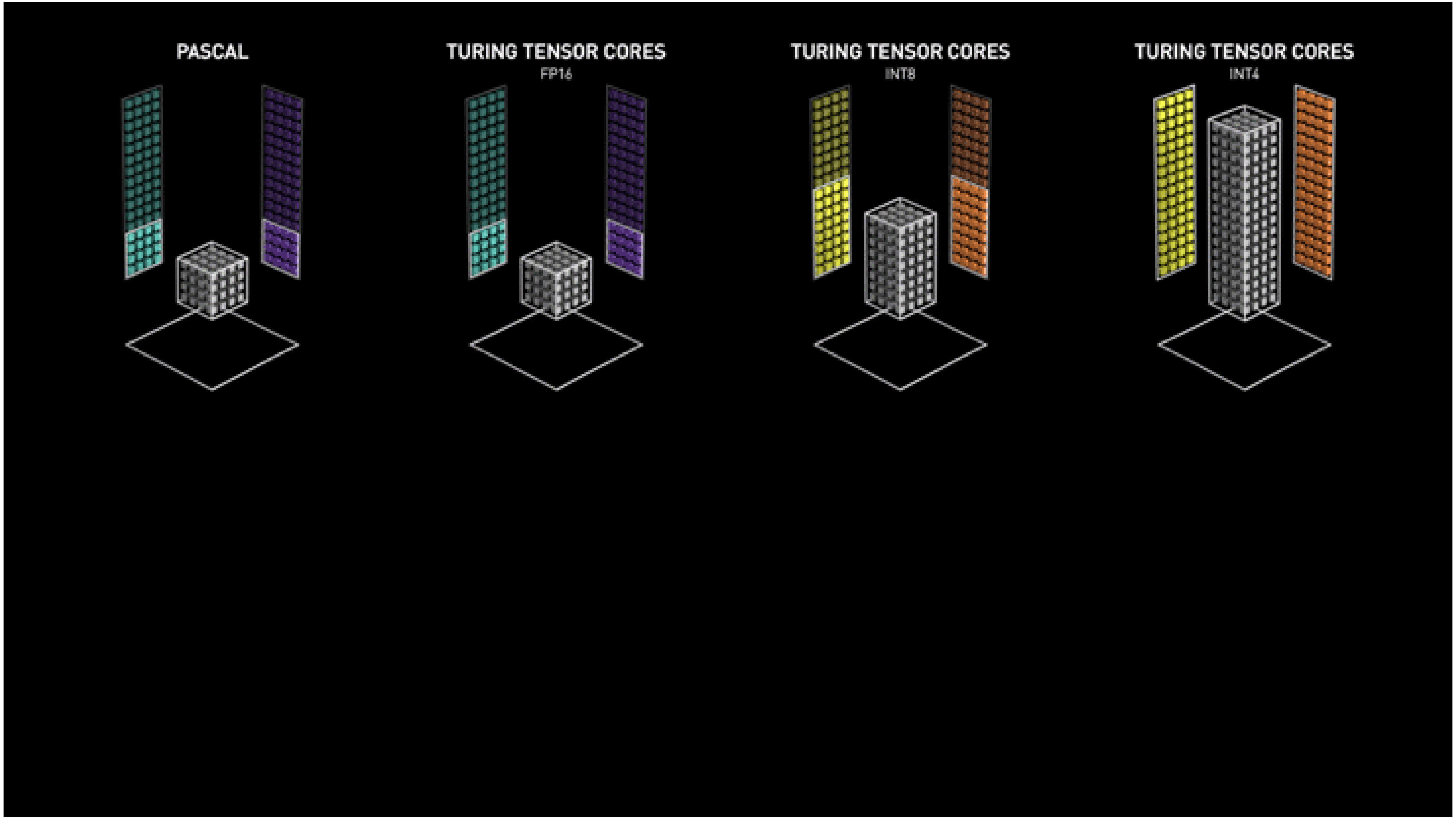


Tensor Cores

- Matrix multiplication unit in SMM



Tensor Cores



Recap

- CUDA programming and GPU architecture
- GPU optimization techniques:
  - Coherence warps
  - Coalesced memory access
  - Shared memory bank conflict
  - Warp level optimizations
  - Tensor core