Lecture 4:

GPU Architecture & CUDA Programming

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2023

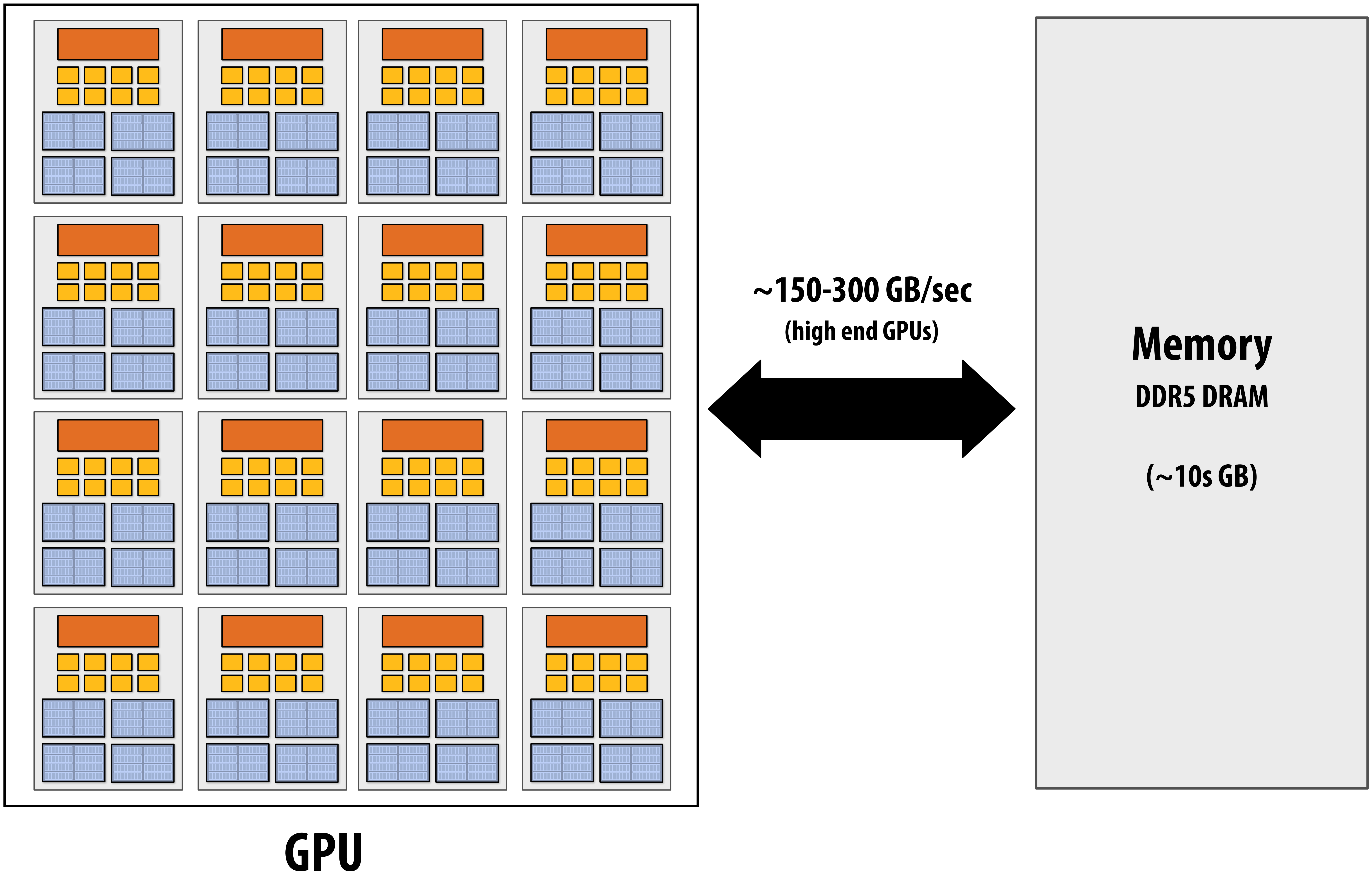
Today

- History: how graphics processors, originally designed to accelerate 3D games like Quake, evolved into highly parallel compute engines for a broad class of applications

- Programming GPUs using the CUDA language

- A more detailed look at GPU architecture

Recall 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)

Graphics 101 + GPU history
(for fun)

What GPUs were originally designed to do: 3D rendering

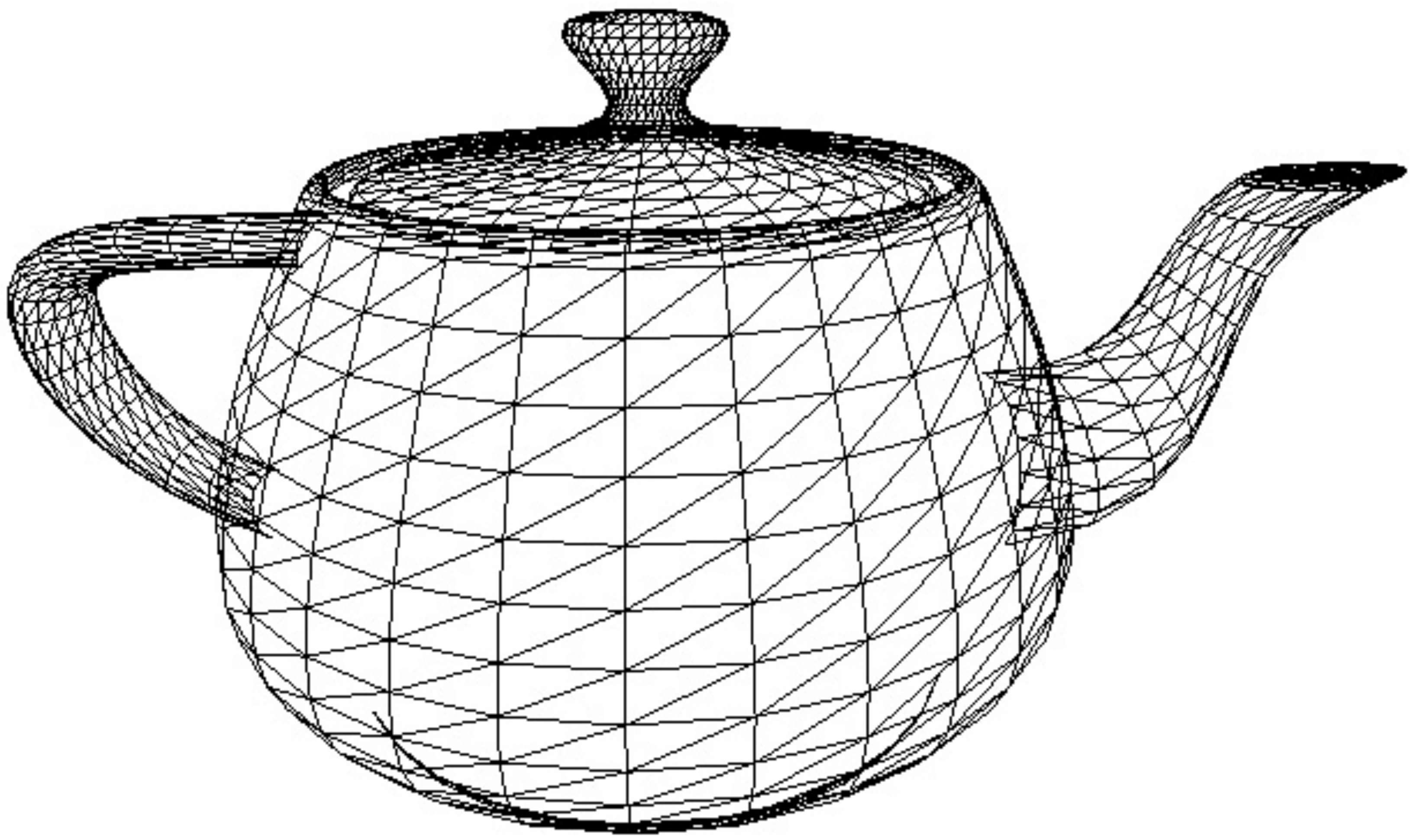


Image credit: Henrik Wann Jensen

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?

What GPUs are still designed to do

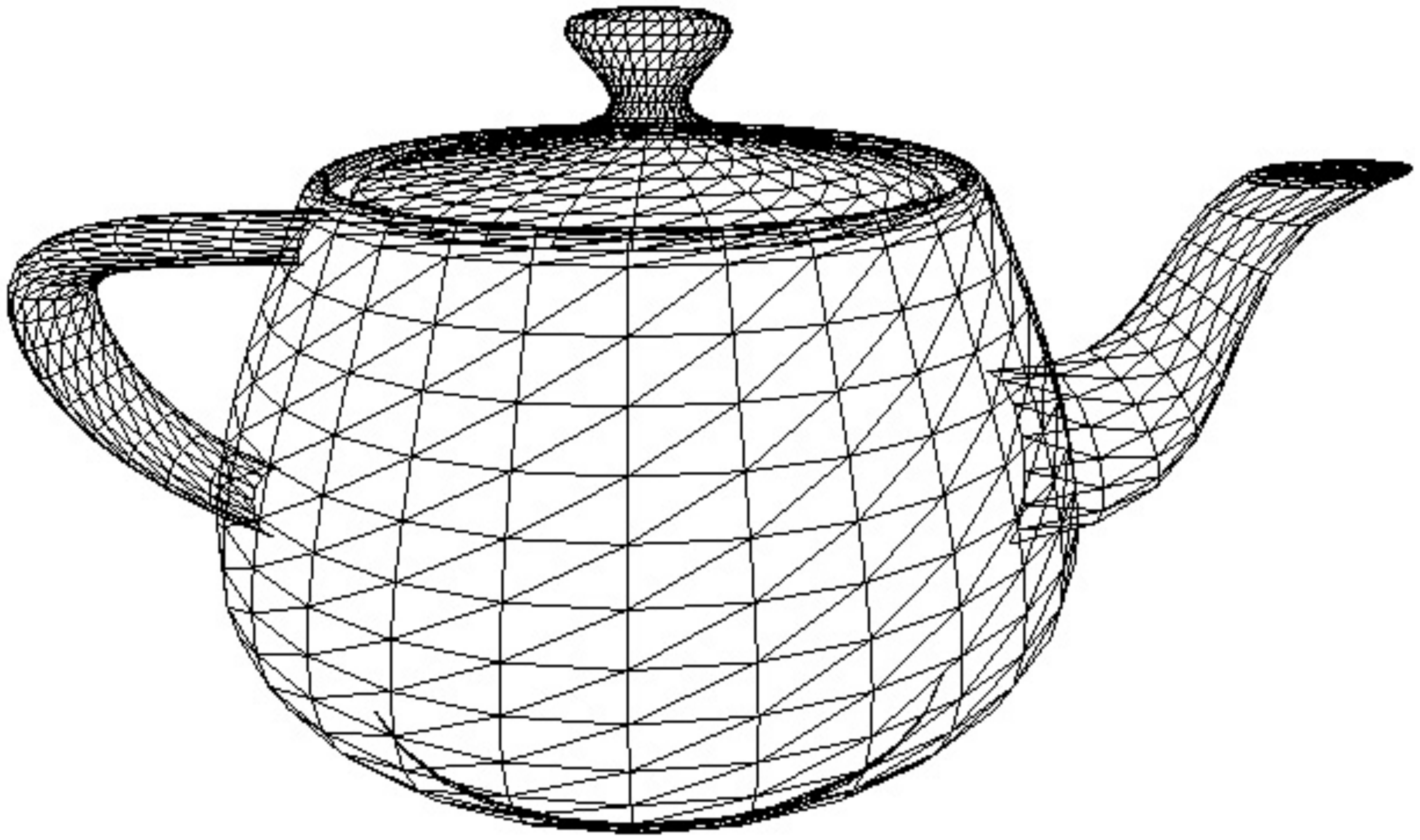
Unreal Engine Kite Demo (Epic Games 2015)

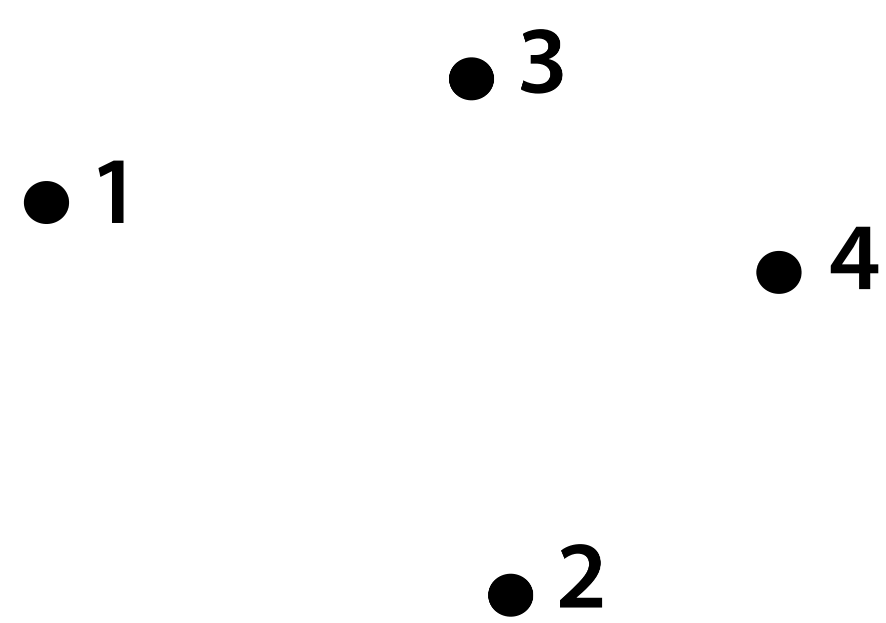
Tip: how to explain a system

- **Step 1: describe the <u>things</u> (key entities) that are manipulated**
  - **The nouns**

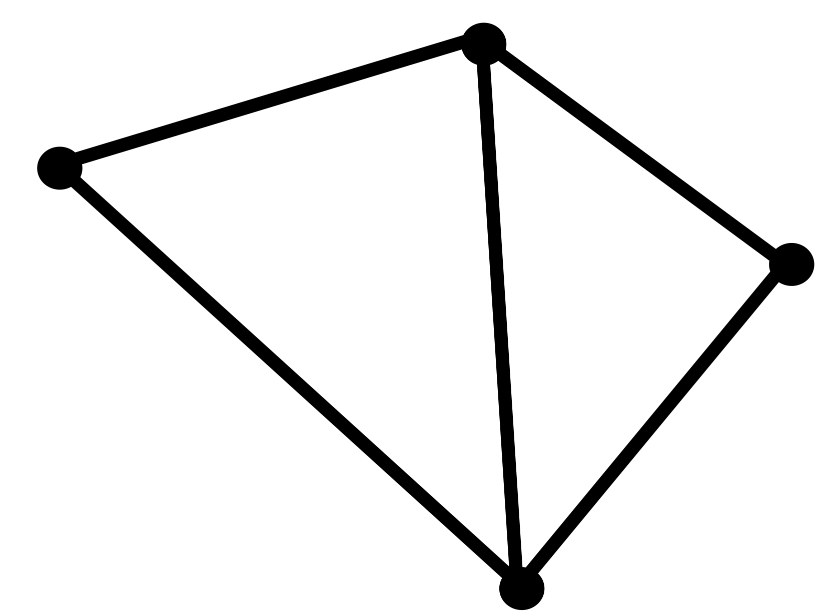
Real-time graphics primitives (entities)

Represent surface as a 3D triangle mesh



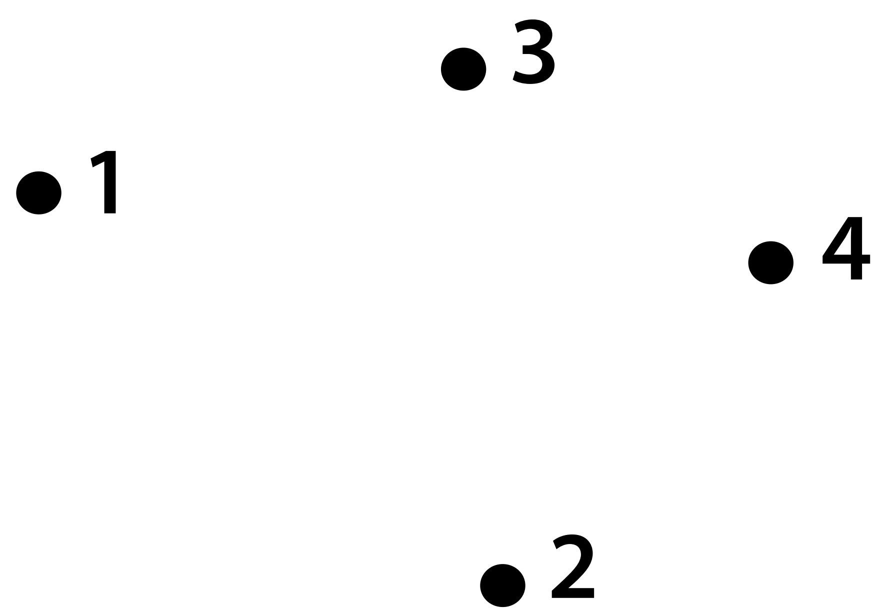


Vertices
(points in space)

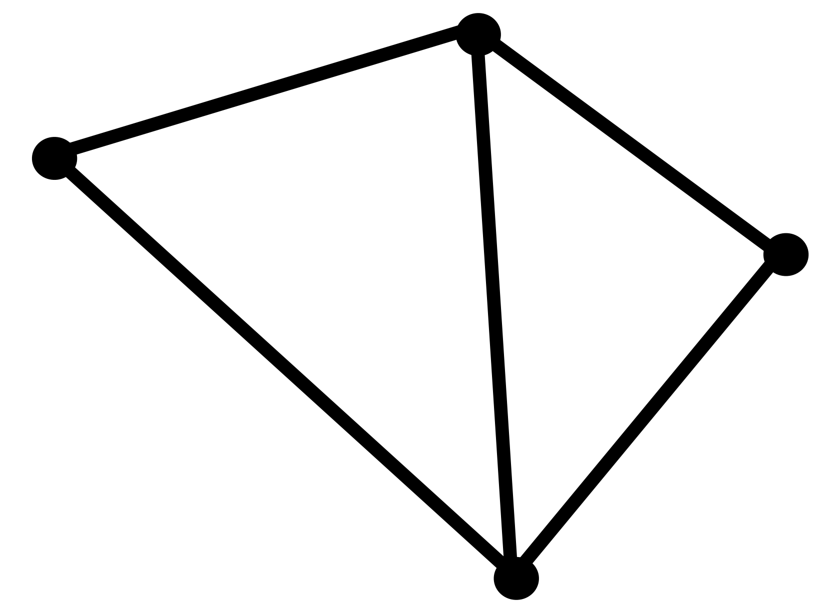


Primitives
(e.g., triangles, points, lines)

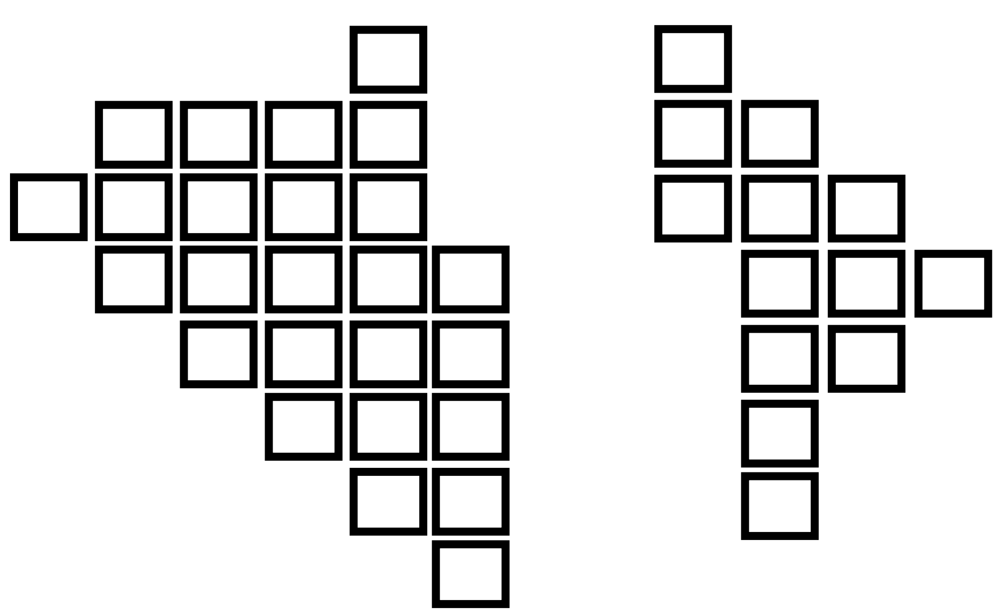
Real-time graphics primitives (entities)



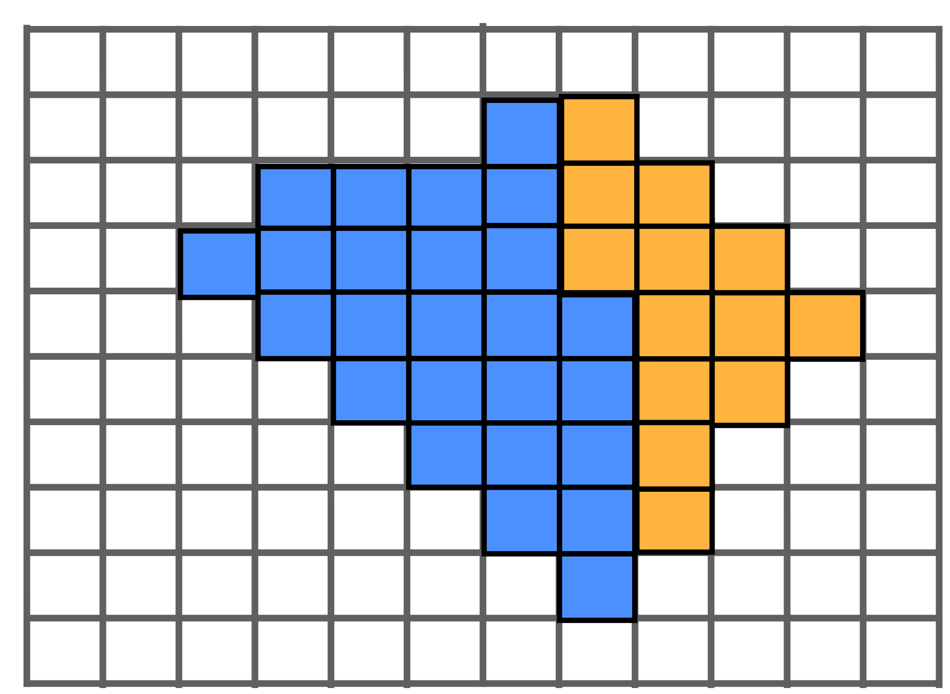
Vertices
(points in space)



Primitives
(e.g., triangles, points, lines)



Fragments



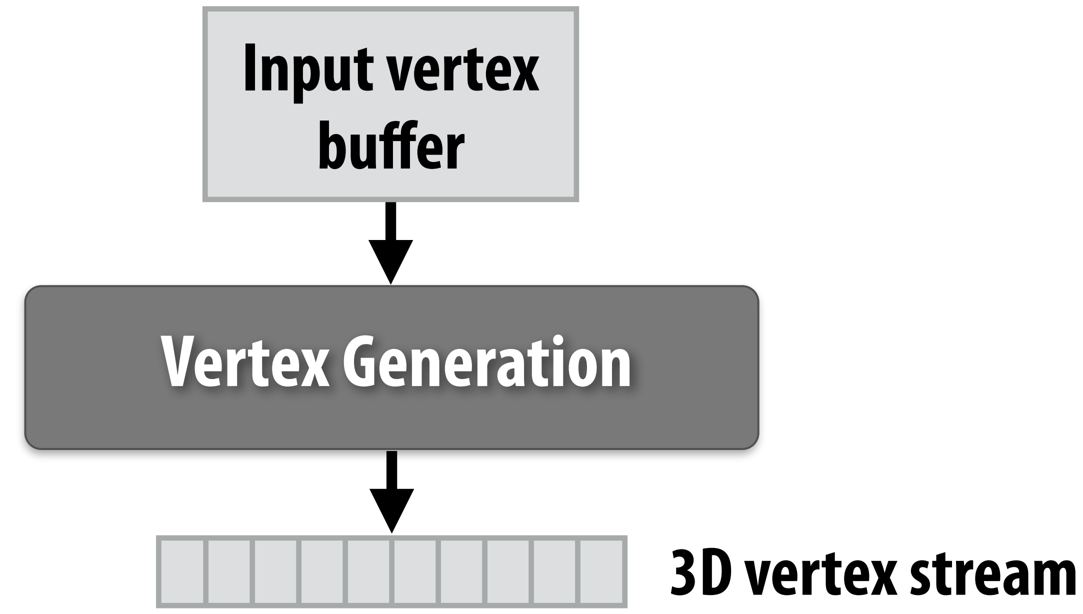
Pixels (in an image)

How to explain a system

- **Step 1: describe the <u>things</u> (key entities) that are manipulated**
  - **The nouns**

- **Step 2: describe operations the system performs on the entities**
  - **The verbs**

Rendering a picture



Input: a list vertices in 3D space (and their connectivity into primitives)

Example: every three vertices defines a triangle

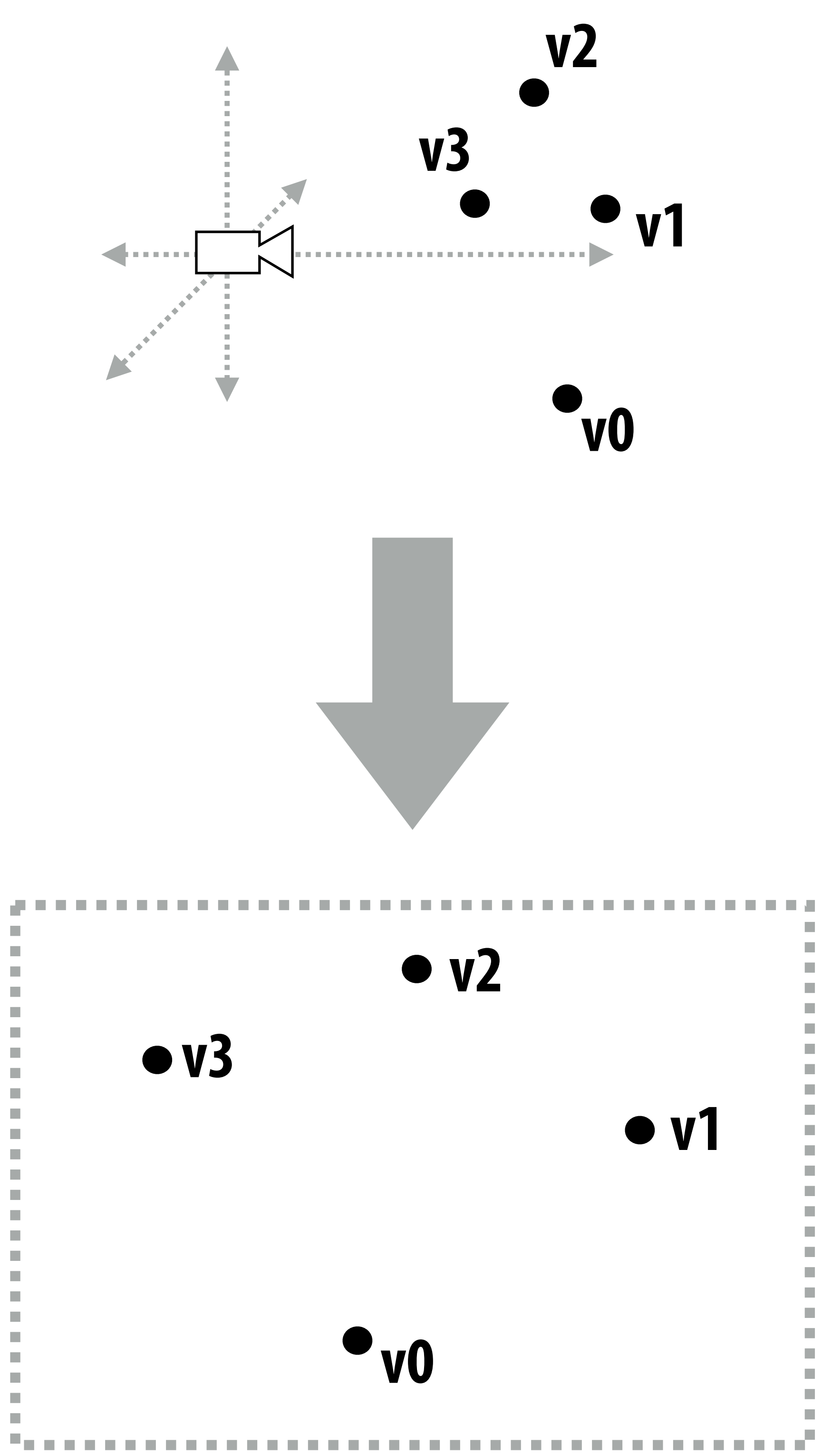
```
list_of_positions = {
    v0x, v0y, v0z,
    v1x, v1y, v1x,
    v2x, v2y, v2z,
    v3x, v3y, v3x
  };
```

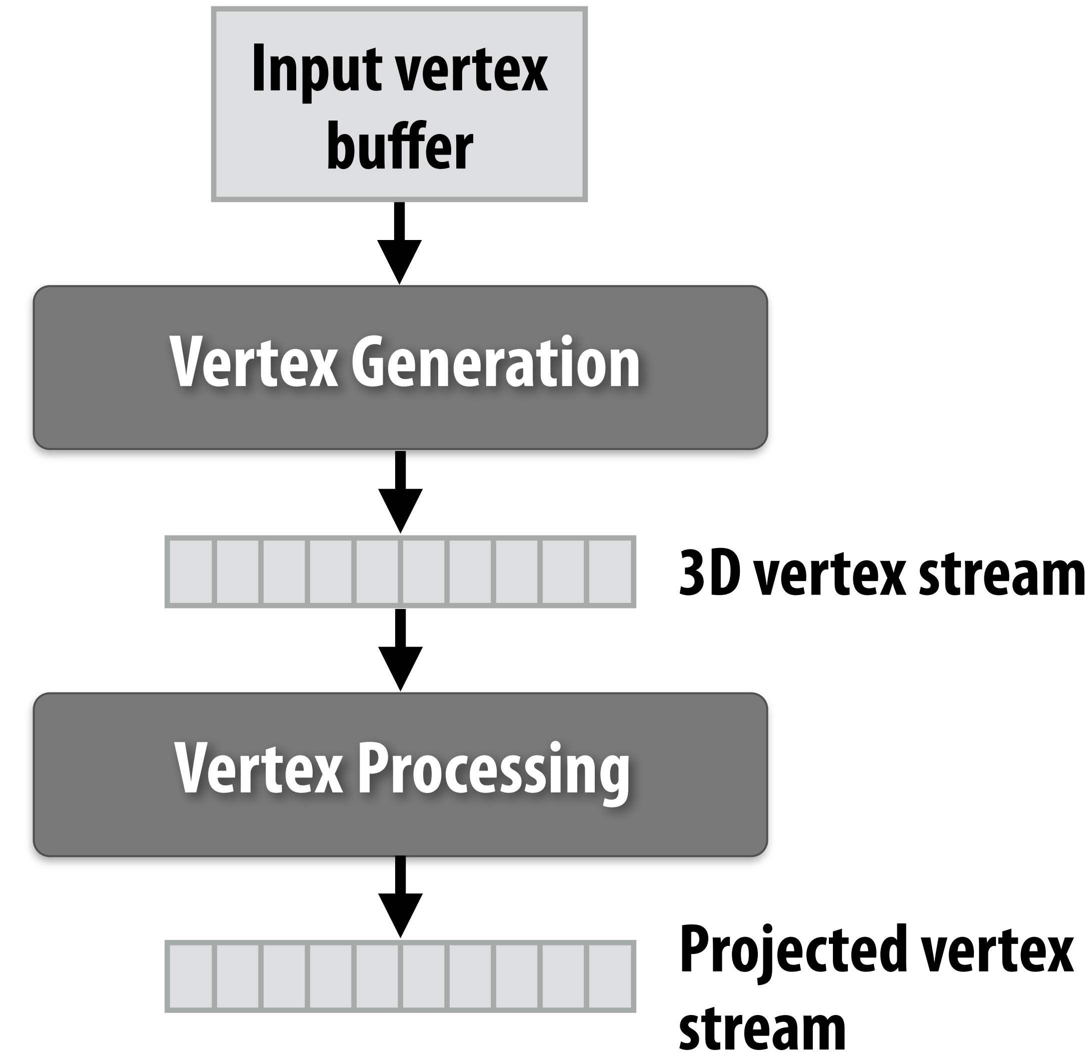
triangle 0 = {v0, v1, v2}

triangle 1 = {v1, v2, v3}

Rendering a picture

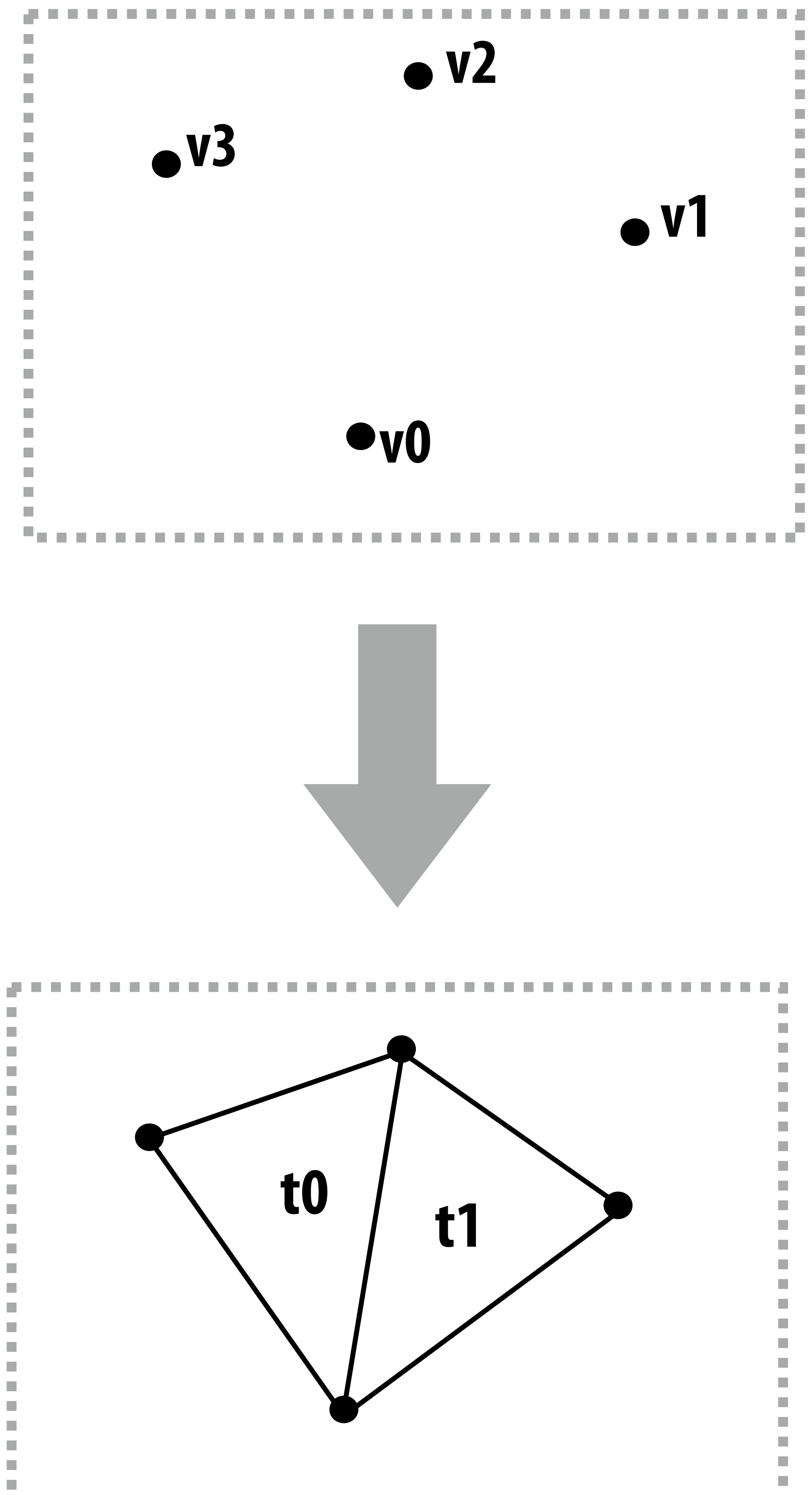
Step 1: given a scene camera position, compute where the vertices lie on screen

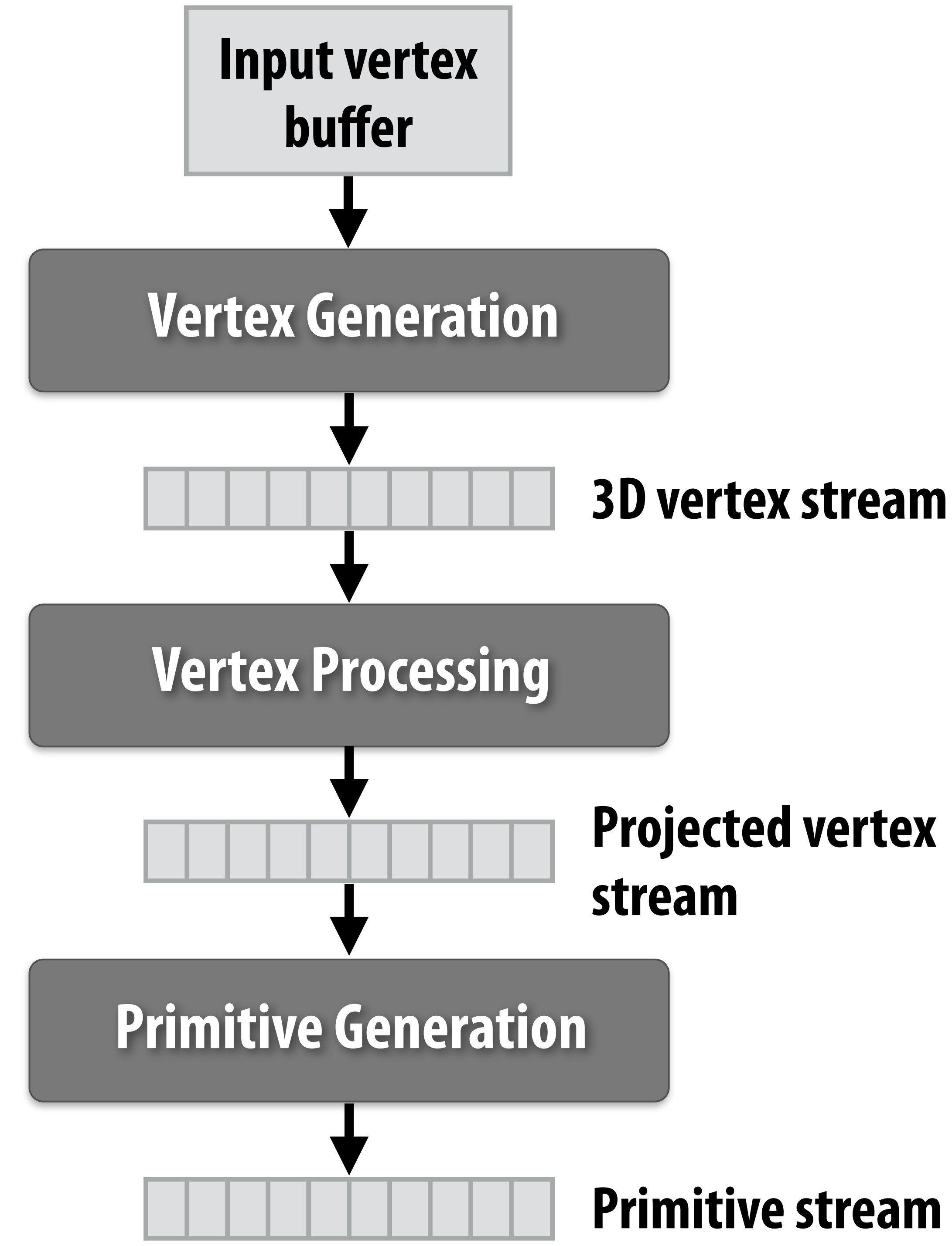




Rendering a picture

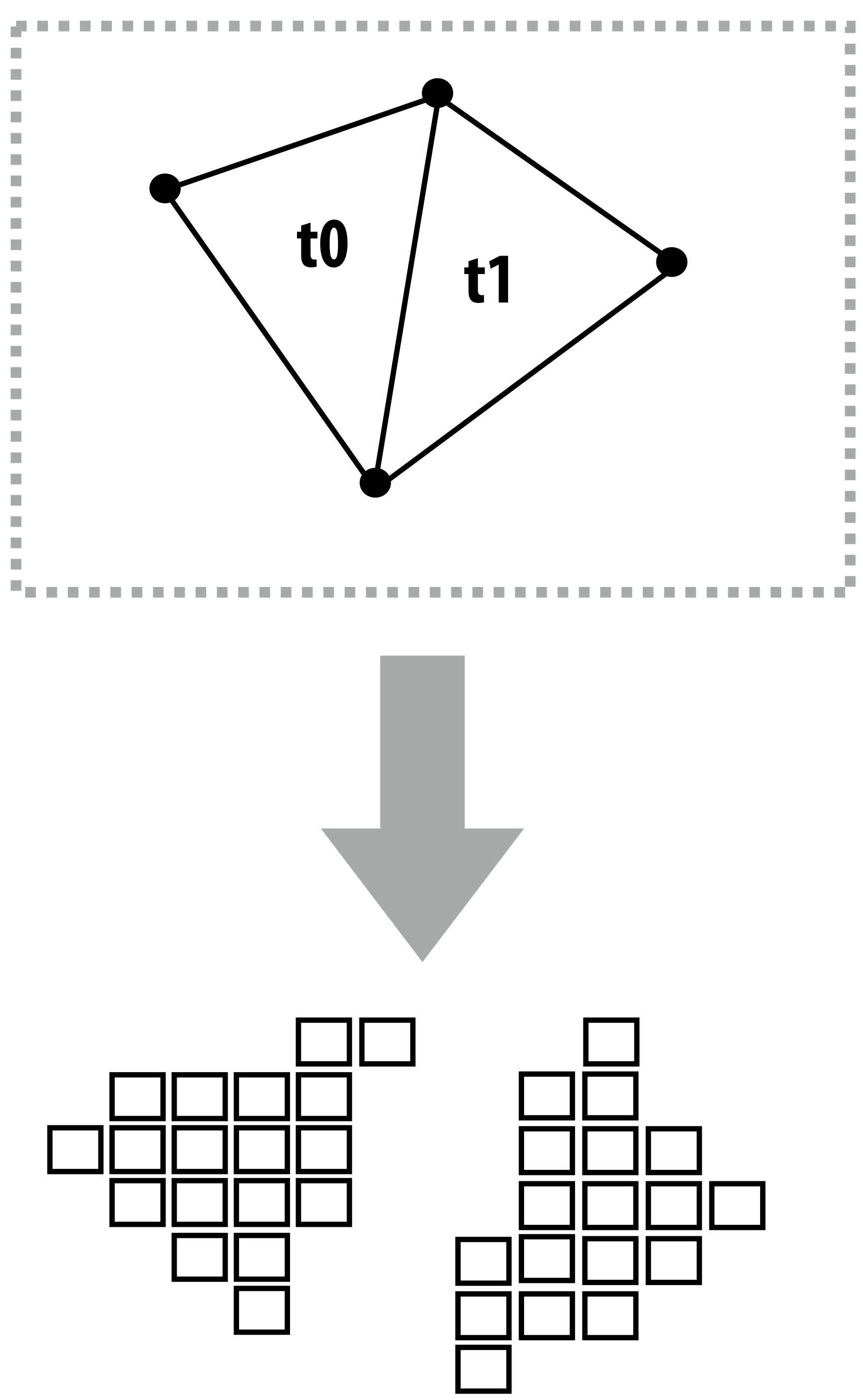
Step 2: group vertices into primitives

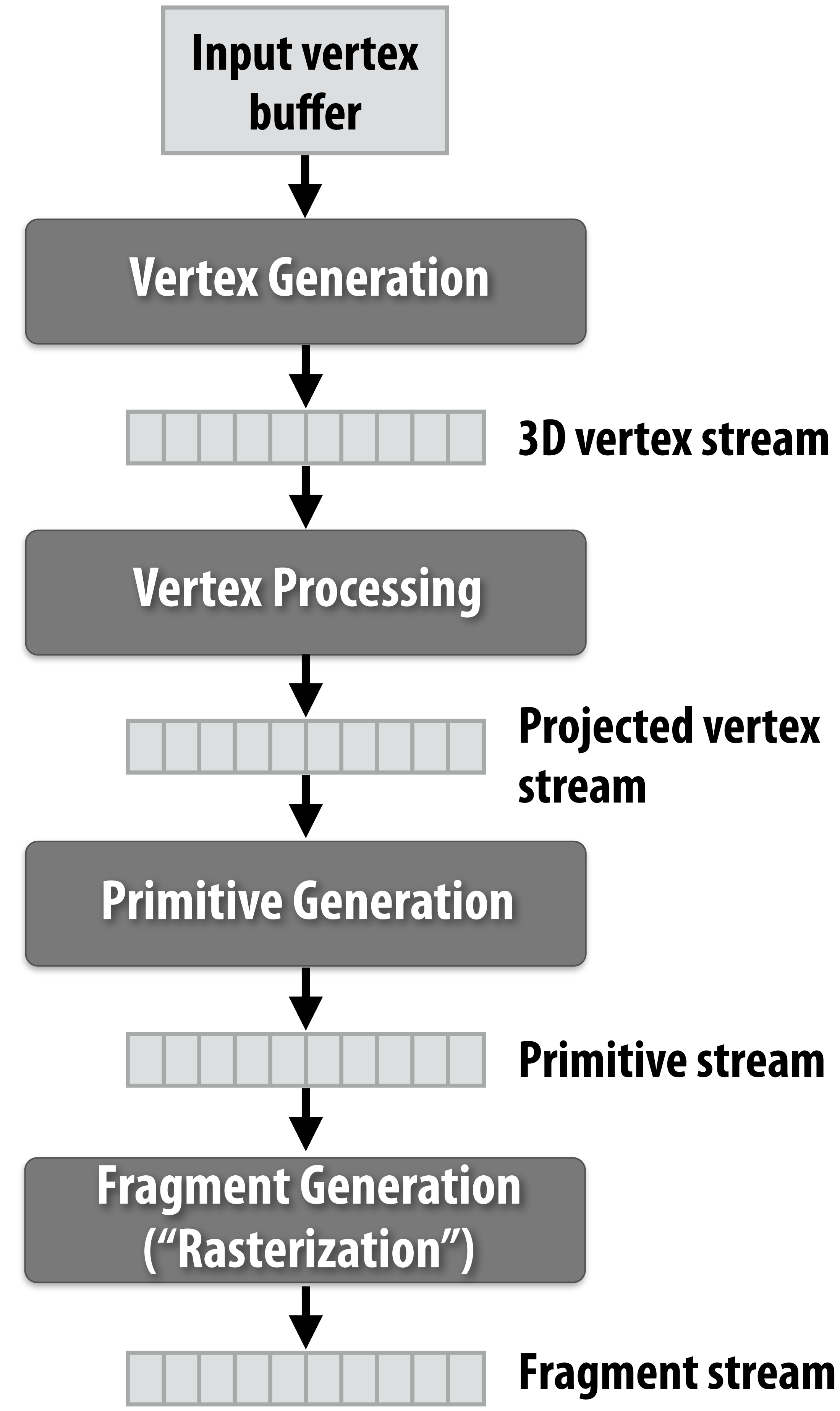




Rendering a picture

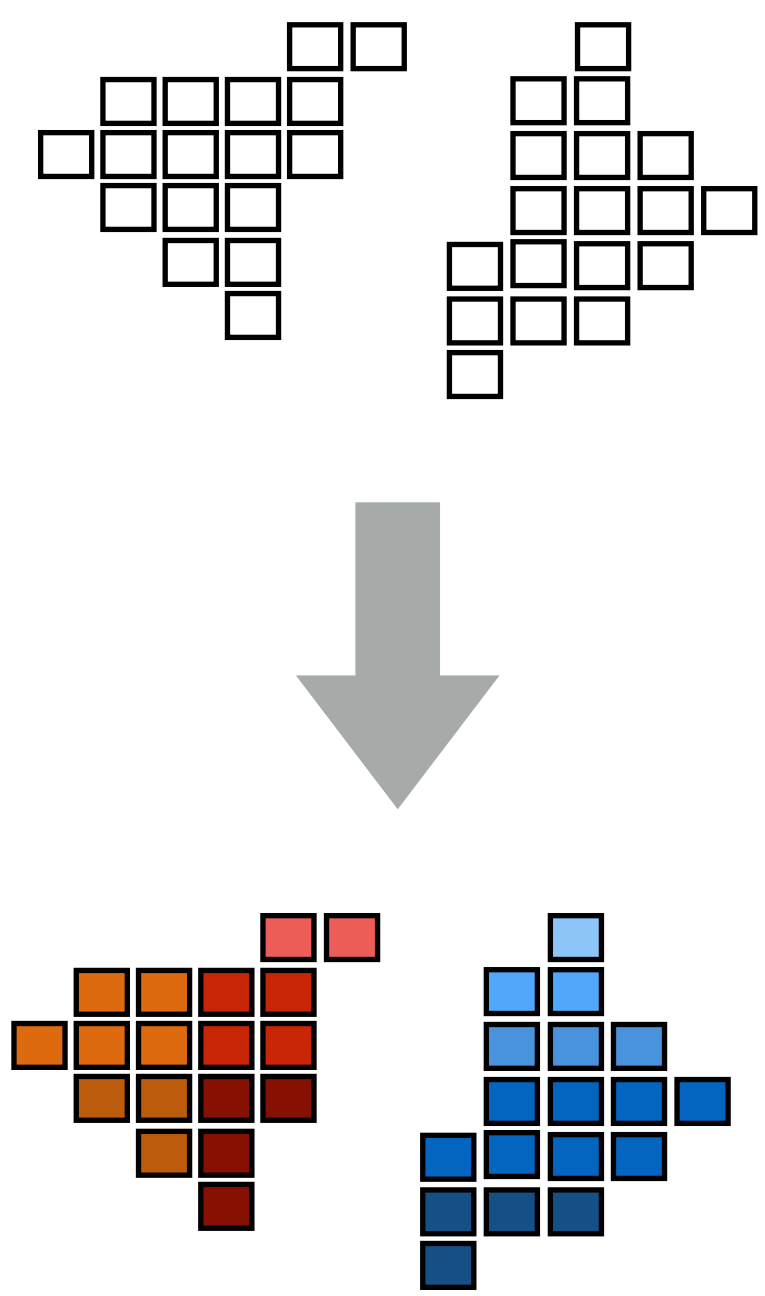
Step 3: generate one fragment for each pixel a primitive overlaps

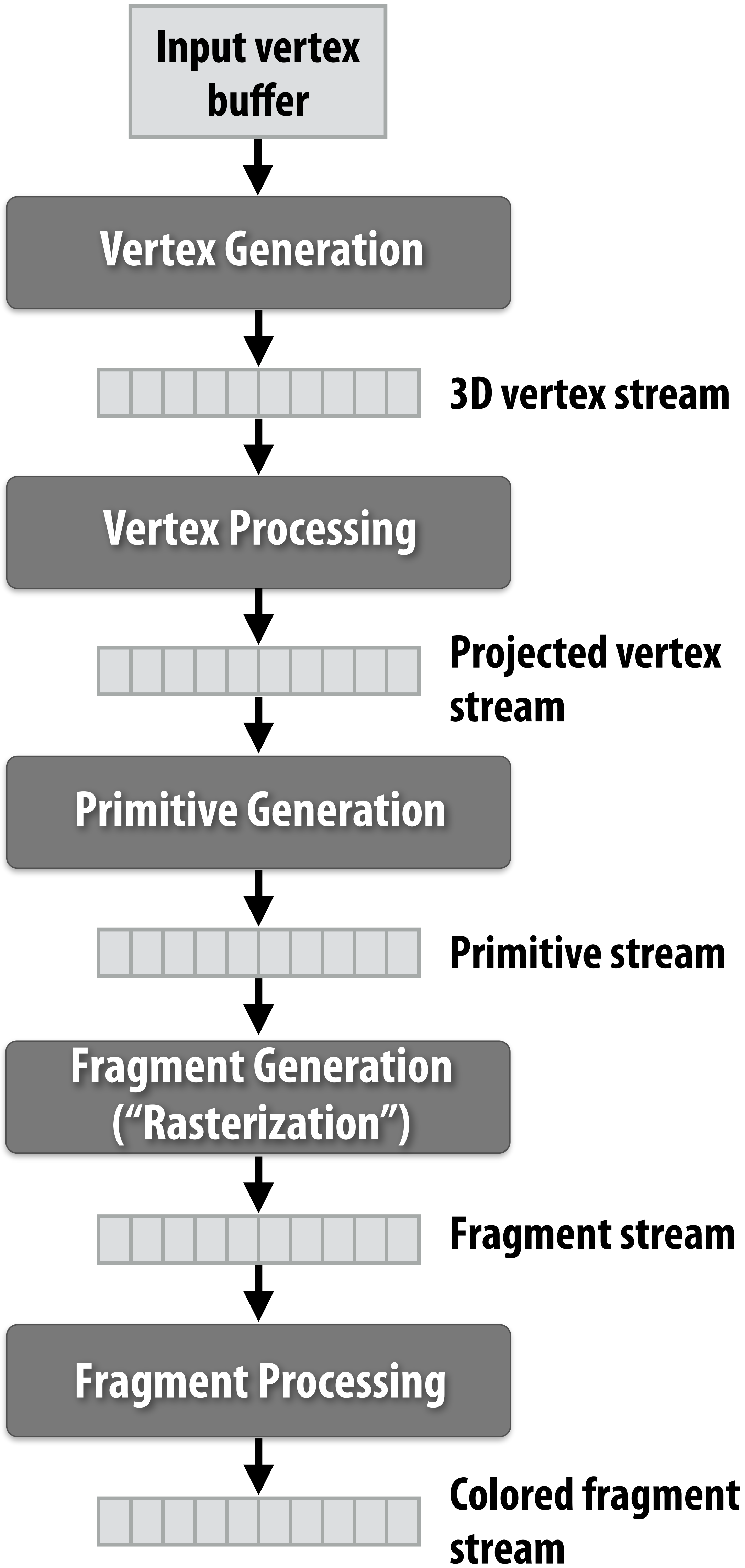




Rendering a picture

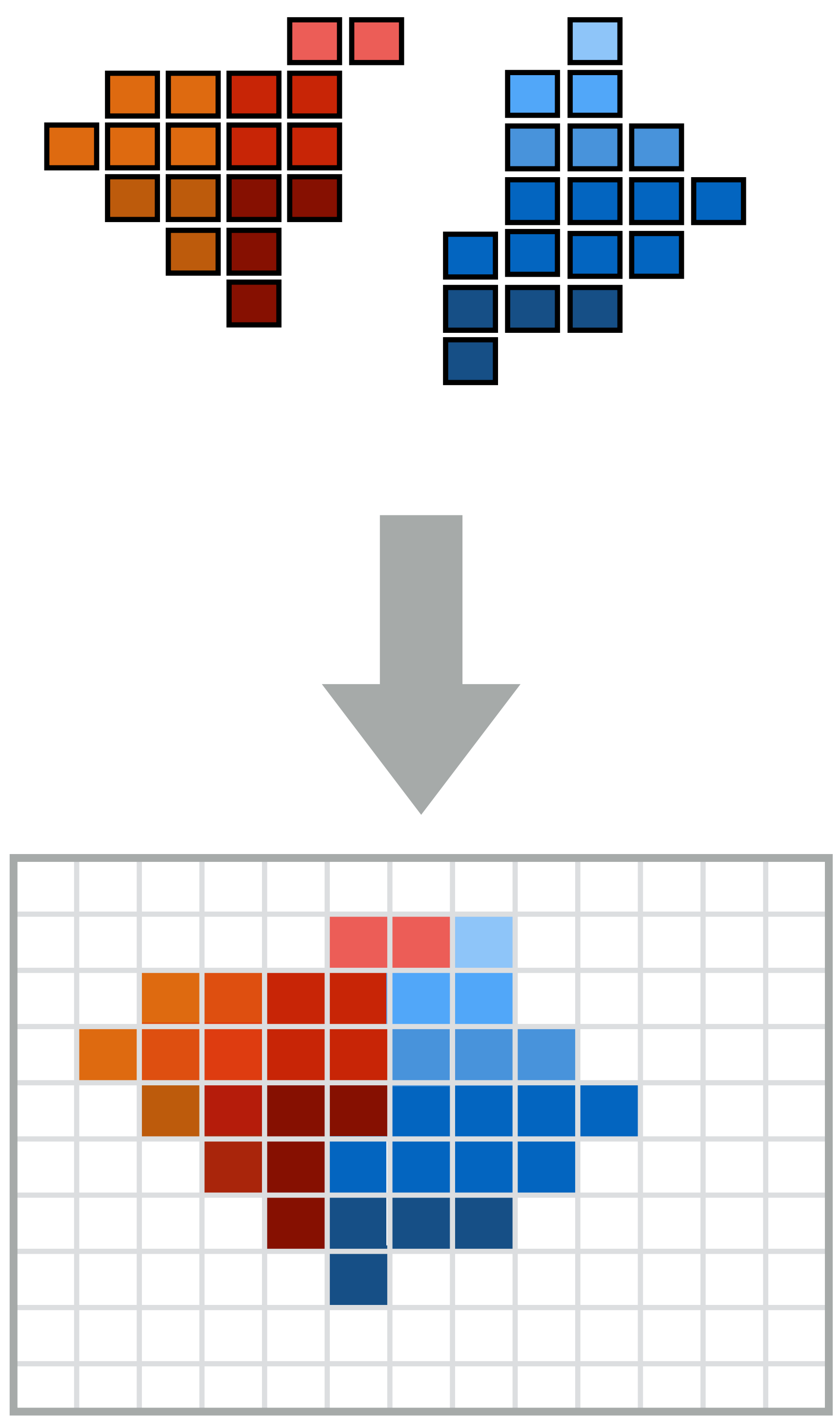
Step 4: compute color of primitive for each fragment (based on scene lighting and primitive material properties)

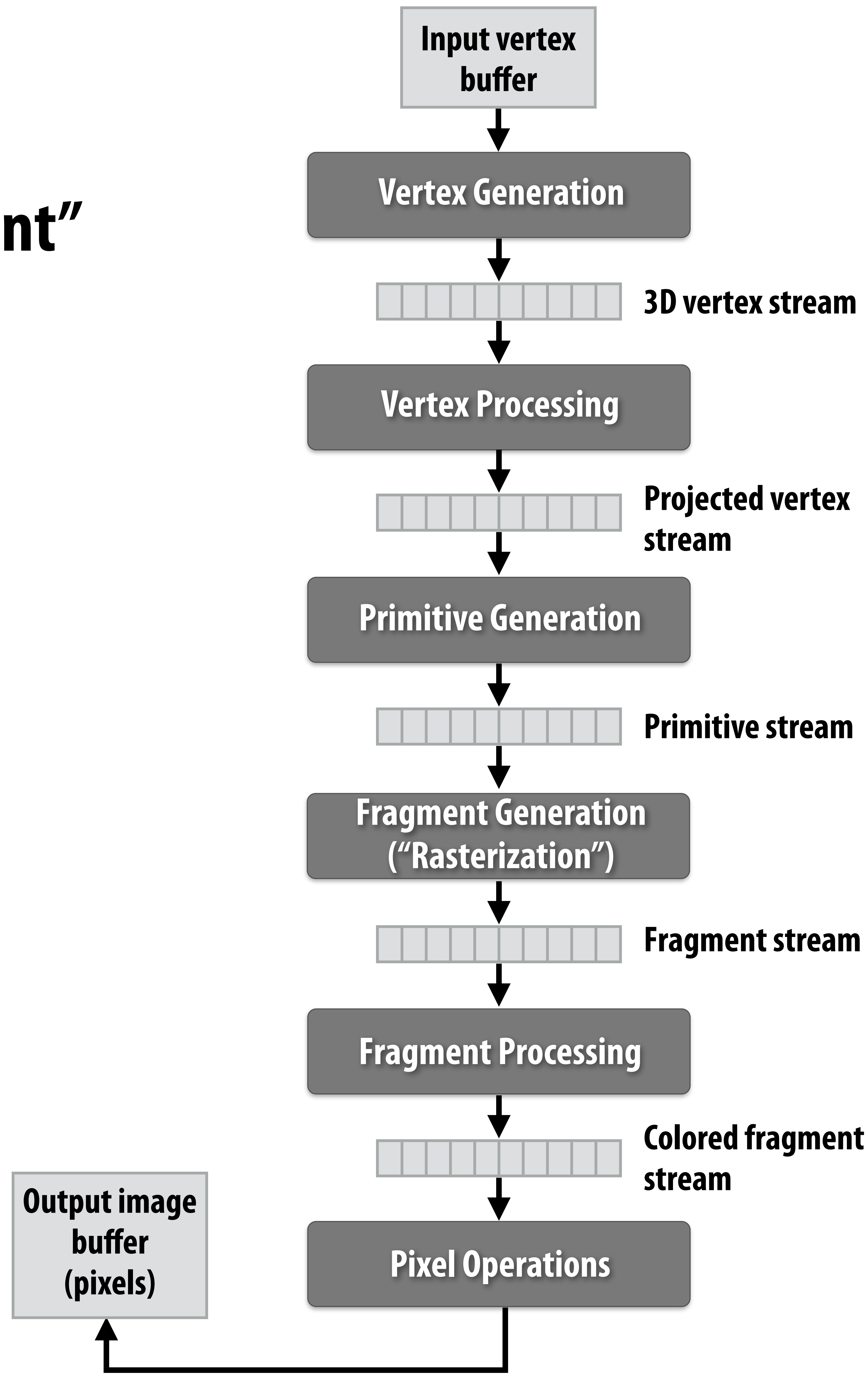




Rendering a picture

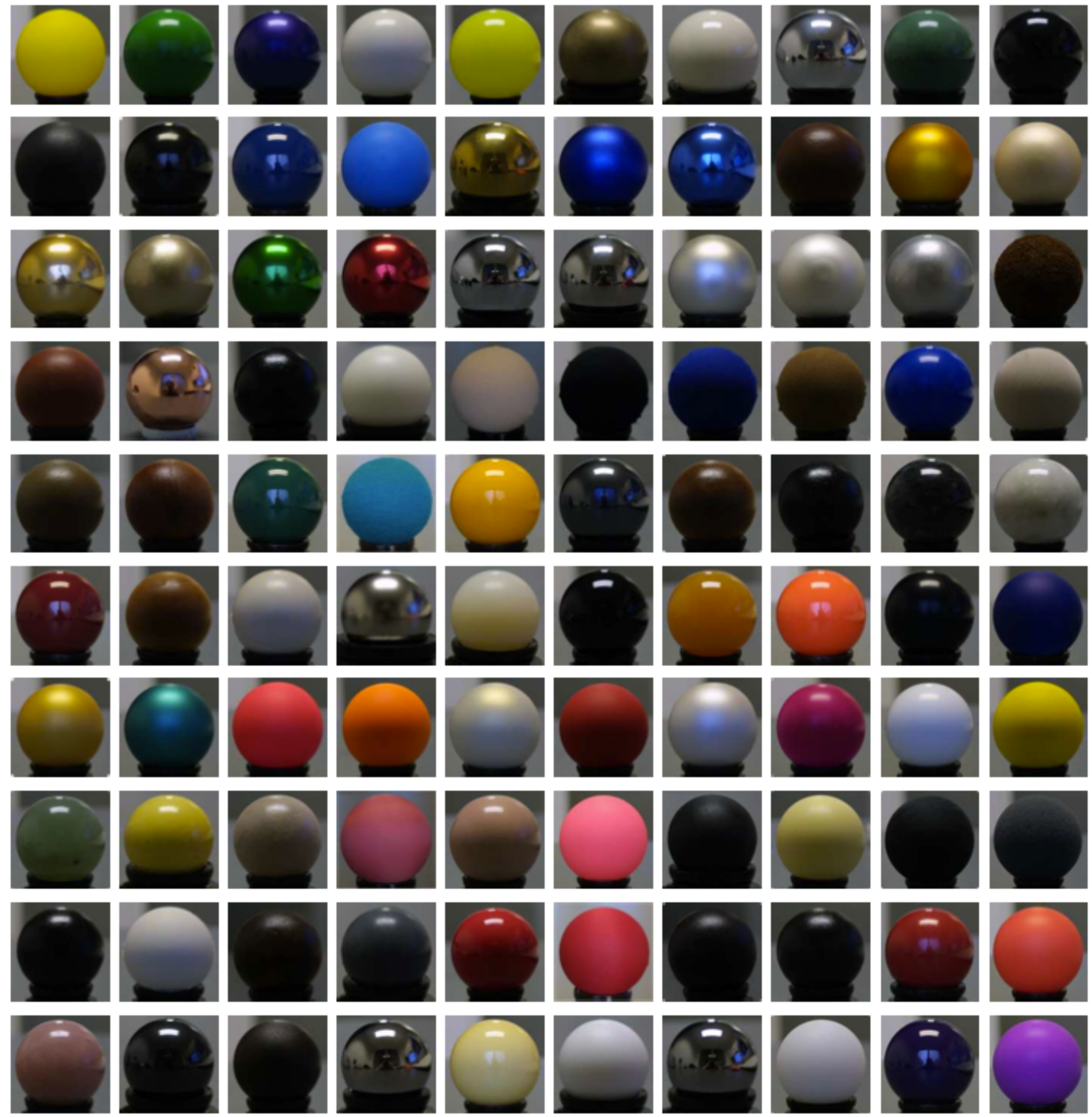
Step 5: put color of the "closest fragment" to the camera in the output image





Fragment processing computations simulate reflection of light off of real-world materials

Example materials:



Images from Matusik et al. SIGGRAPH 2003

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**

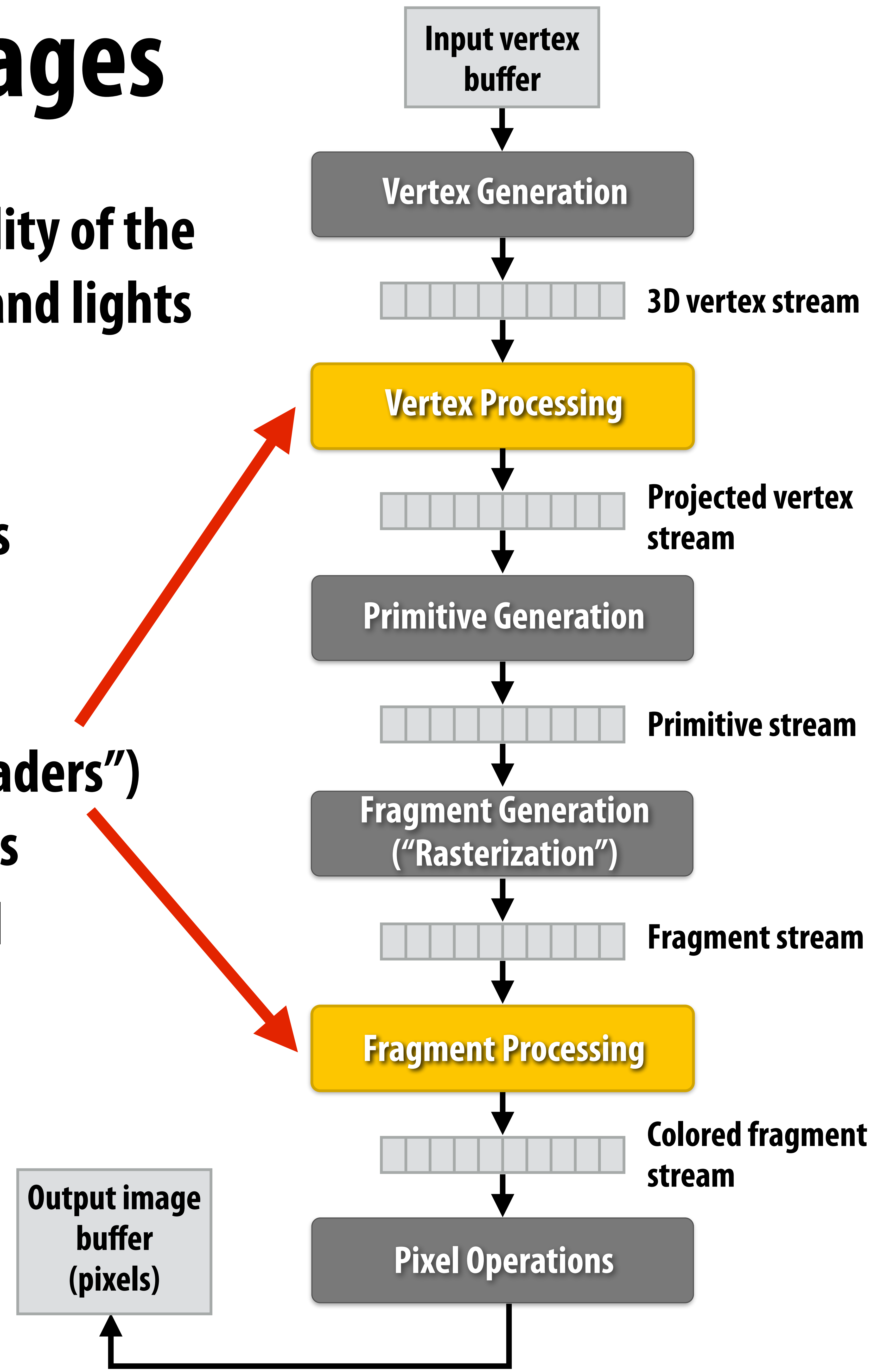
  - `glMaterial(face, parameter_id, parameter_value)`
    - **Examples of material parameters: color, shininess**

Great diversity of materials and lights in the world!

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);
}
```

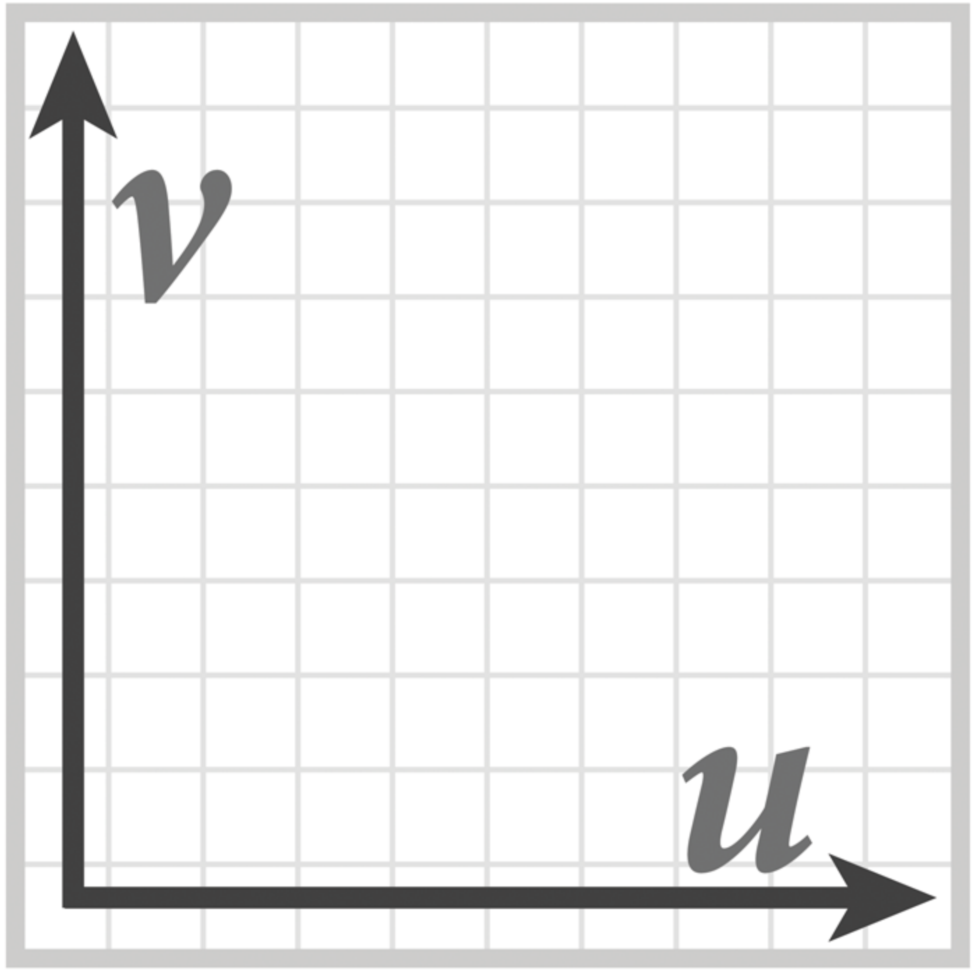
read-only global variables

per-fragment inputs

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

per-fragment output: RGBA surface color at pixel

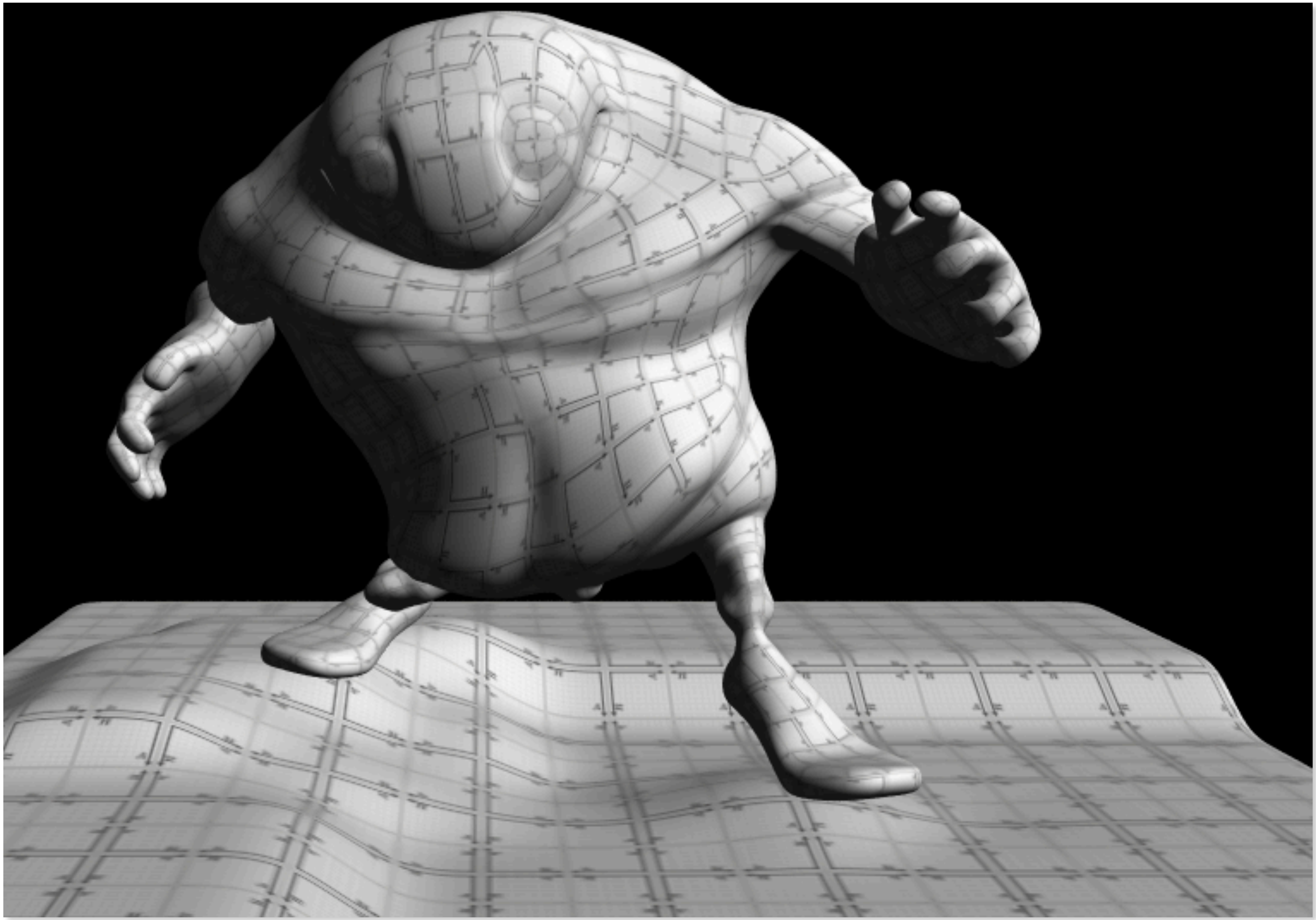
`myTexture` is a texture map



* 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.

Shaded result

Image contains output of `myFragmentShader` for each pixel covered by surface
(pixels covered by multiple surfaces contain output from surface closest to camera)



Can we use shader for other computation?

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

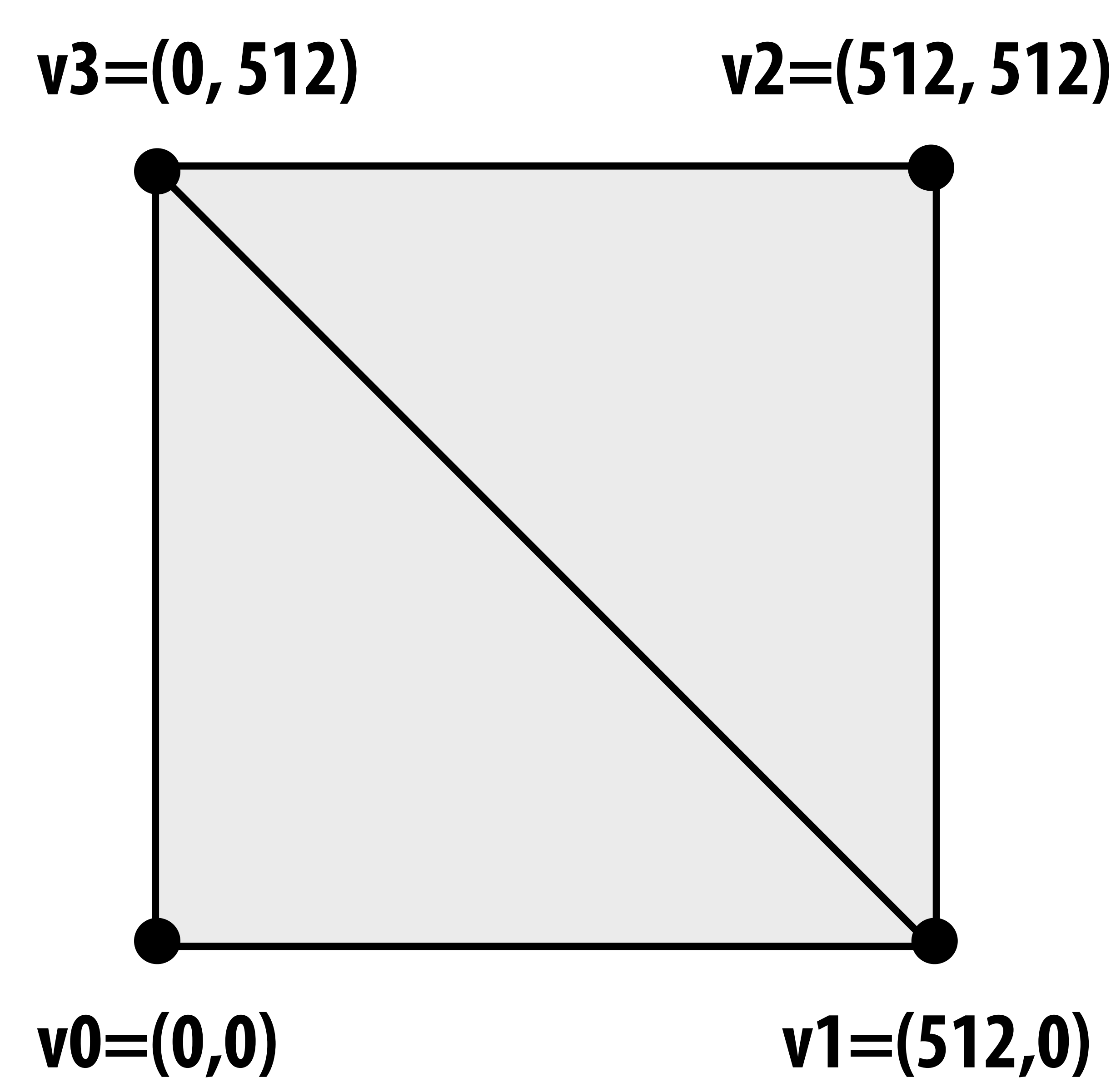
Render 2 triangles that exactly cover screen

(one shader computation per pixel = one shader computation output image element)

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

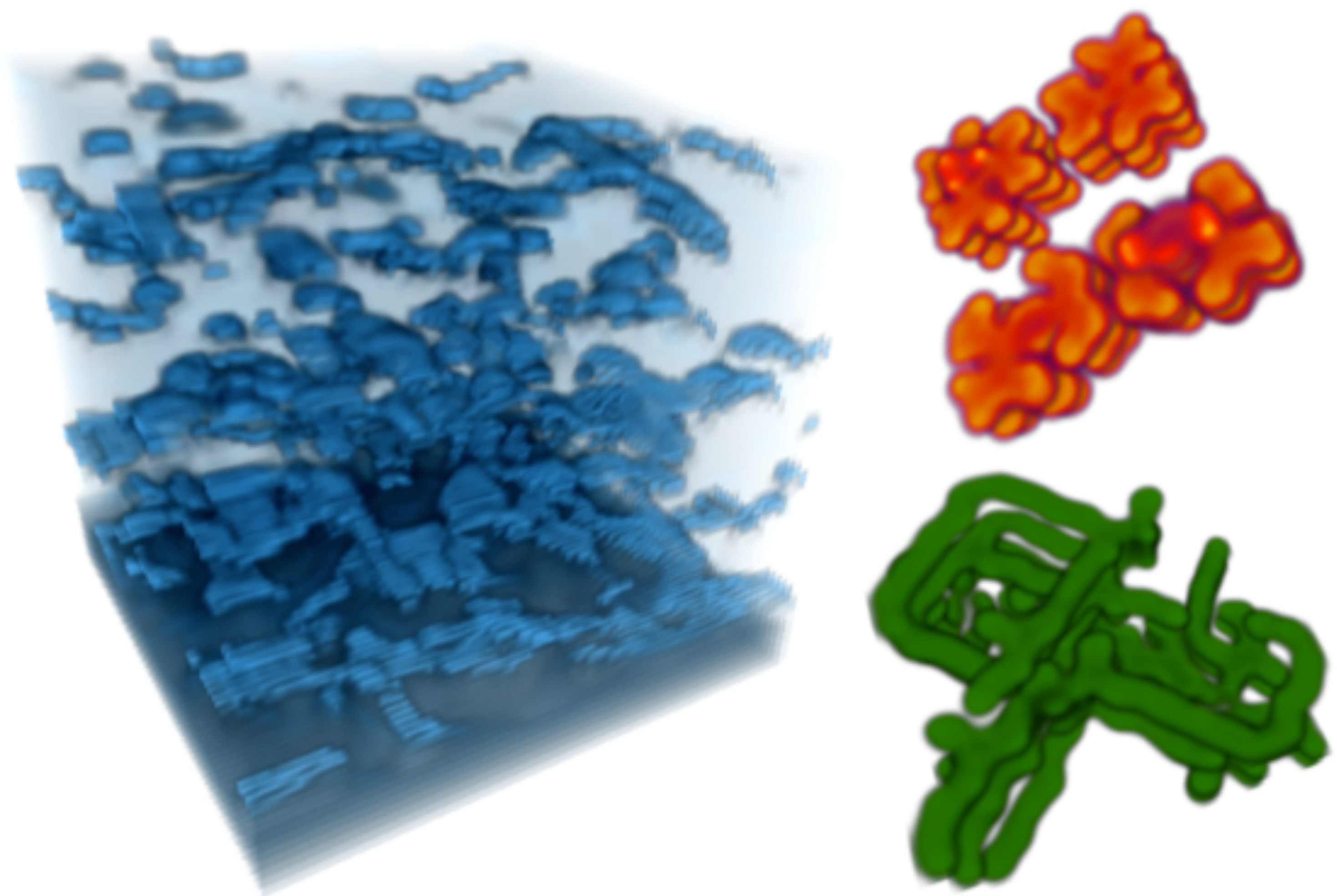
Fragment shader function is mapped over 512 x 512 element collection.

Hack!

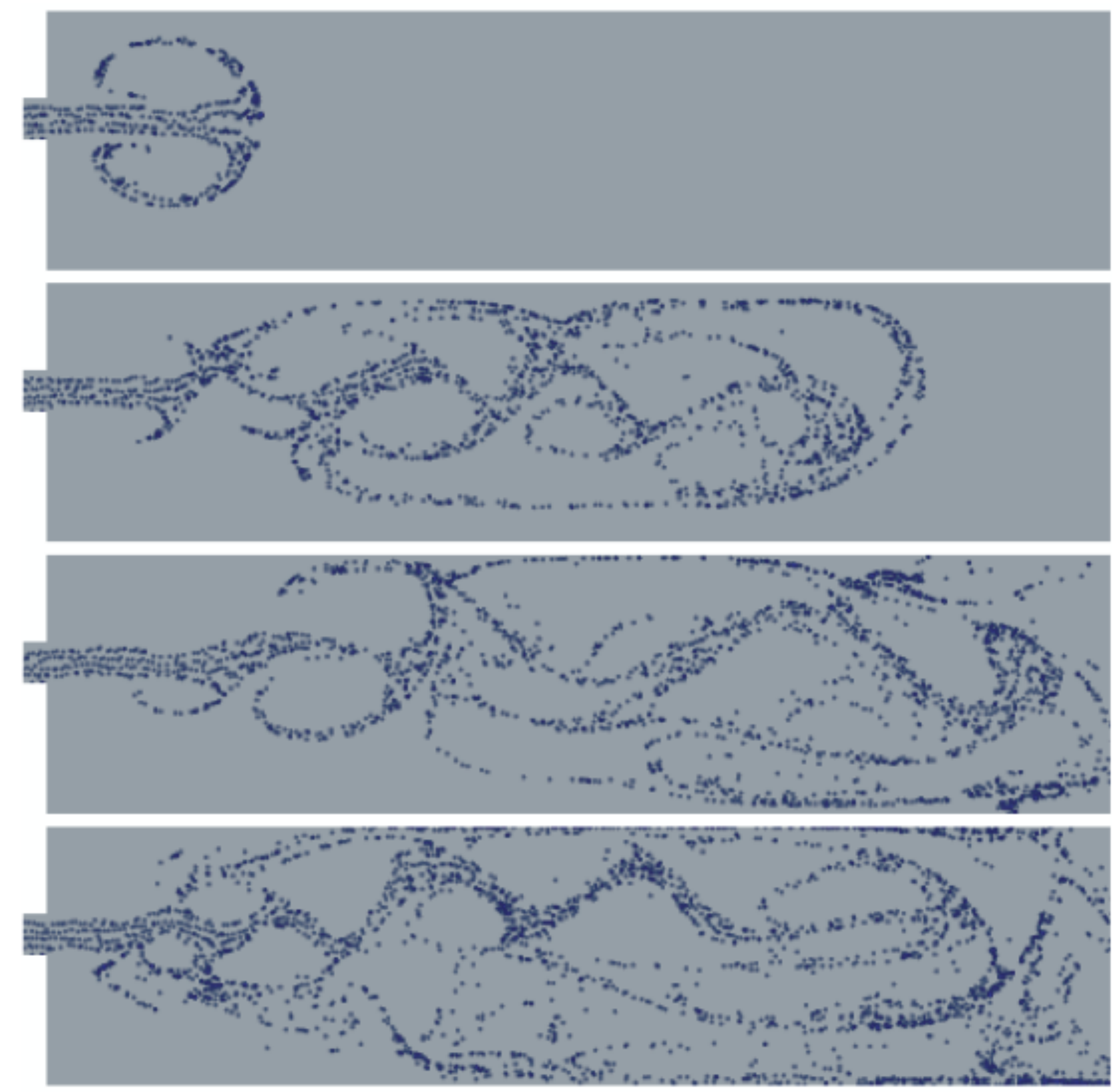


"GPGPU" 2002-2003

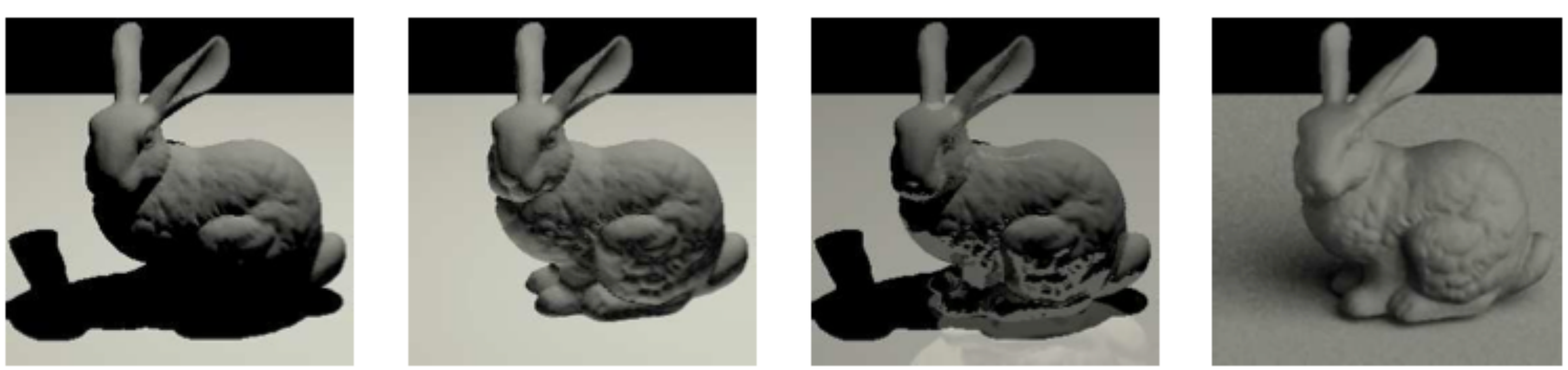
GPGPU = "general purpose" computation on GPUs



Coupled Map Lattice Simulation [Harris 02]



Sparse Matrix Solvers [Bolz 03]



Ray Tracing on Programmable Graphics Hardware [Purcell 02]

Brook stream programming language (2004)

- **Stanford graphics lab research project** [Buck 2004]
- **Abstract GPU hardware as data-parallel processor**

```
kernel void scale(float amount, float a<>, out float b<>)
{
   b = amount * a;
}

float scale_amount;
float input_stream<1000>;   // stream declaration
float output_stream<1000>;  // stream declaration

// omitting stream element initialization...

// map kernel onto streams
scale(scale_amount, input_stream, output_stream);
```

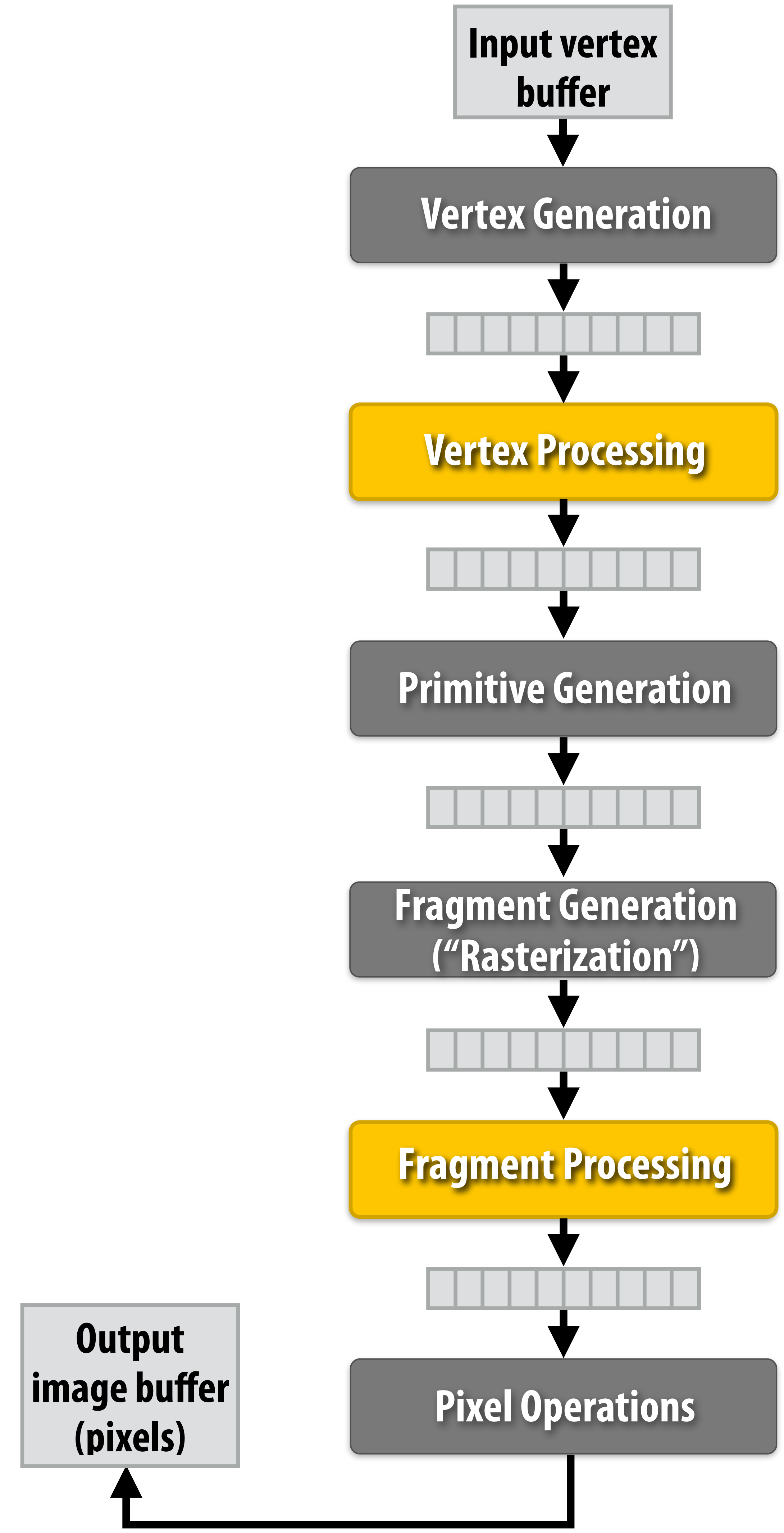
- **Brook compiler converted generic stream program into OpenGL commands such as drawTriangles() and a set of shader programs.**

How to run code on a GPU (prior to 2007)

Lets say a user wants to draw a picture using a GPU…

- Application (via graphics driver) provides GPU vertex and fragment shader program binaries
- Application sets graphics pipeline parameters (e.g., output image size)
- Application provides hardware a buffer of vertices
- Go! (drawPrimitives(vertex_buffer))

This was the only interface to GPU hardware. GPU hardware could only execute graphics pipeline computations.



CUDA Programming Language

CUDA programming language

- Introduced in 2007 with NVIDIA Tesla architecture
- "C-like" language to express programs that run on GPUs
- Relatively low-level: CUDA's abstractions closely match the capabilities/performance characteristics of modern GPUs (design goal: maintain low abstraction distance)

The plan

1. CUDA programming abstractions
2. CUDA implementation on modern GPUs
3. More detail on GPU architecture

Things to consider throughout this lecture:

- Is CUDA a data-parallel programming model?
- Is CUDA an example of the shared address space model?
- Or the message passing model?
- Can you draw analogies to ISPC instances and tasks? What about pthreads?

Clarification (here we go again...)

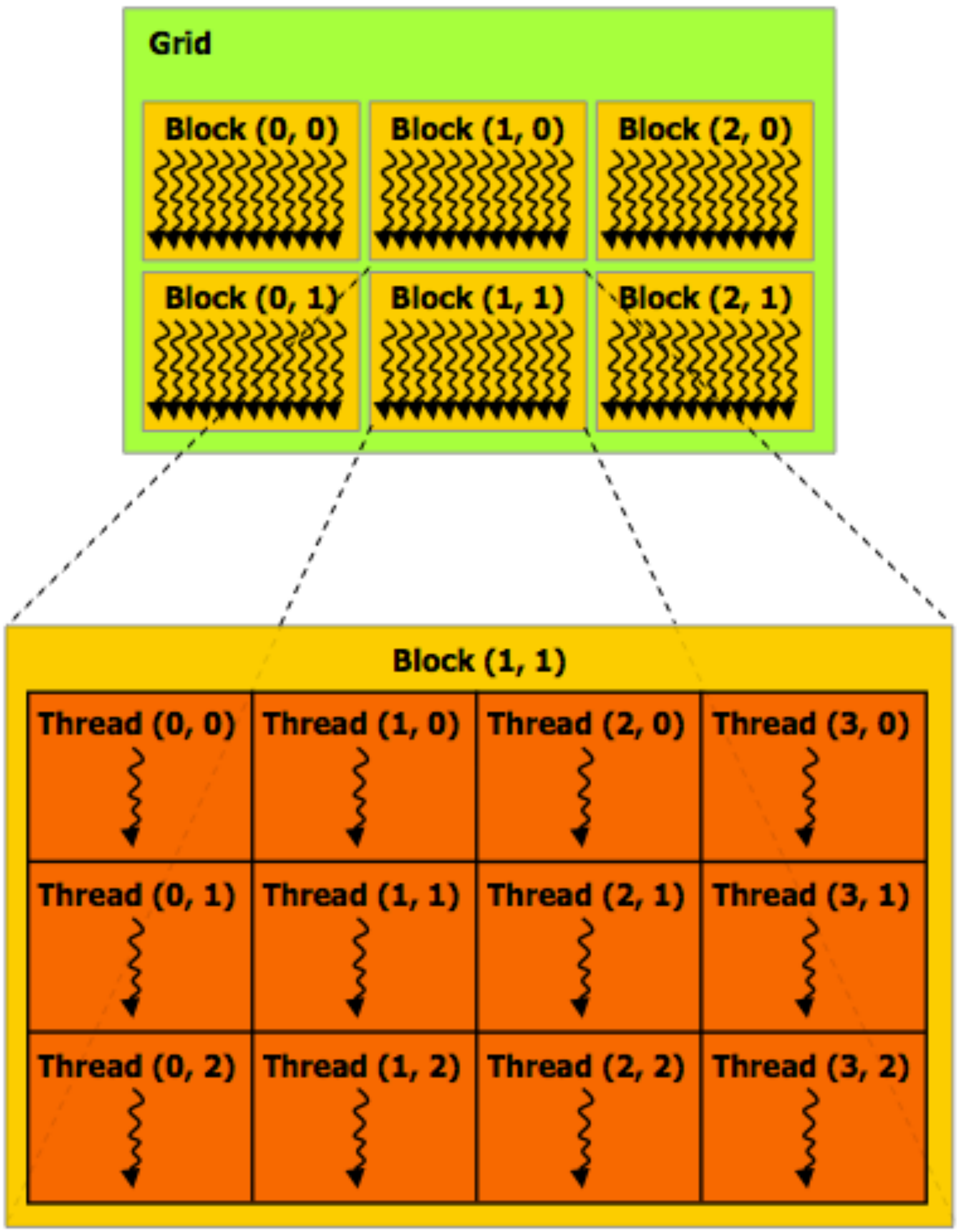
- **I am going to describe CUDA abstractions using CUDA terminology**

- **Specifically, be careful with the use of the term CUDA thread. A CUDA thread presents a similar abstraction as a pthread in that both correspond to logical threads of control, but the implement of a CUDA thread is <u>very different</u>**

- **We will discuss these differences at the end of the lecture**

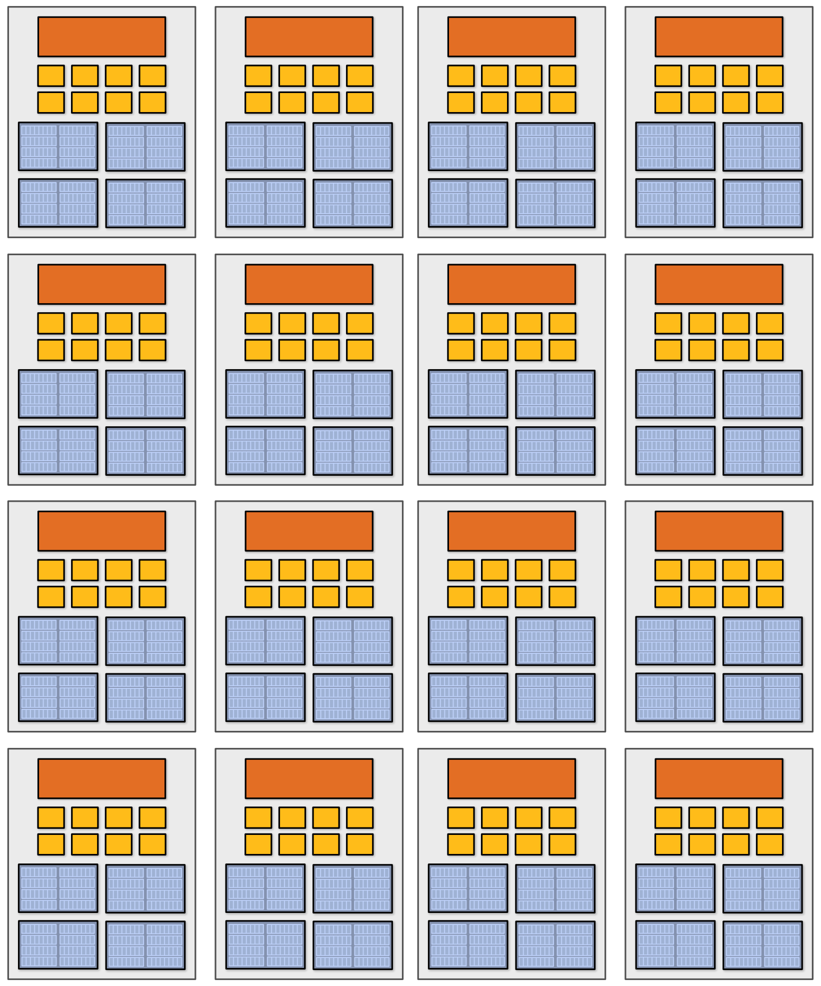
CUDA programs consist of a hierarchy of concurrent threads

Thread IDs can be up to 3-dimensional (2D example below)
Multi-dimensional thread ids are convenient for problems that are naturally N-D



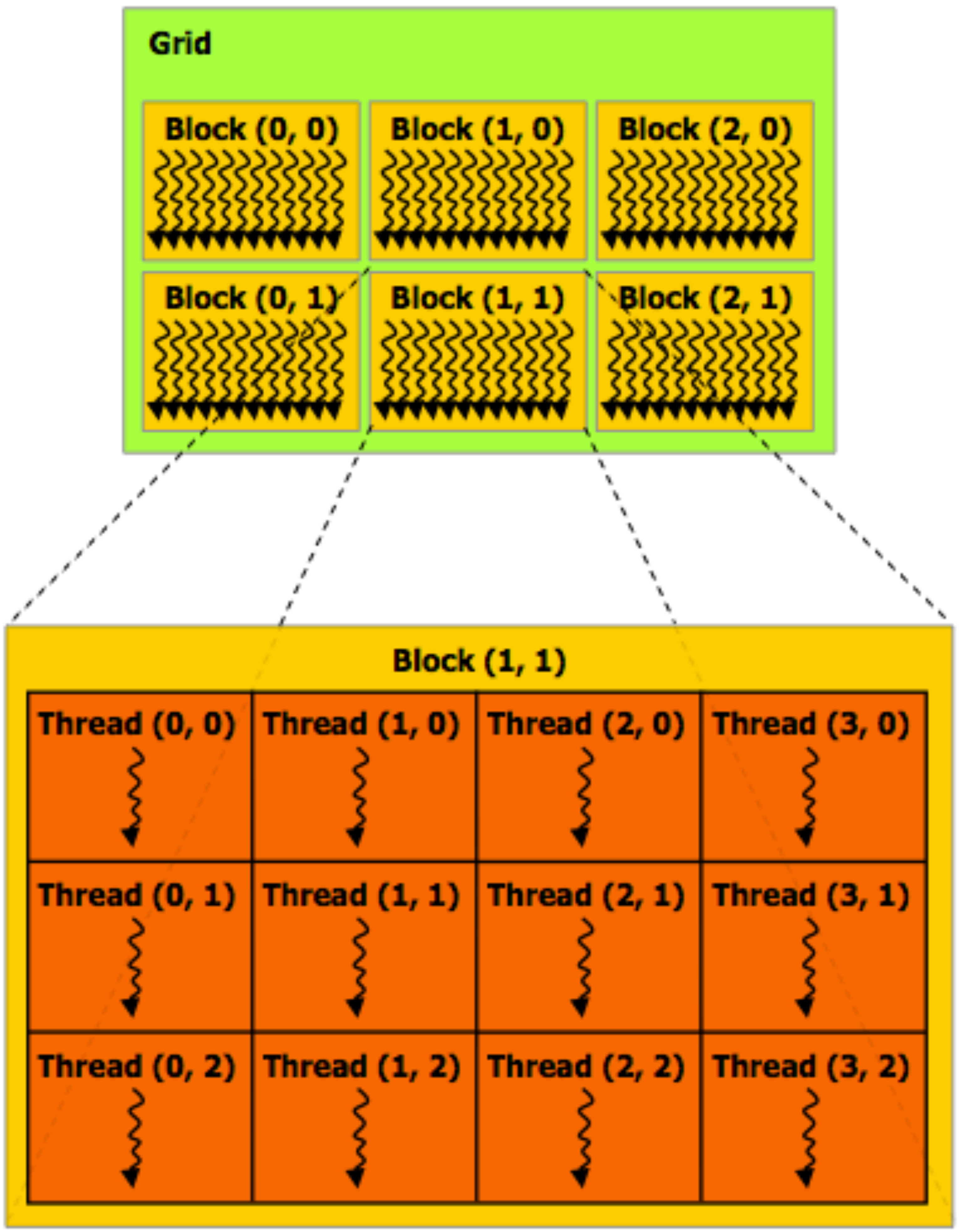
Regular application thread running on CPU (the "host")

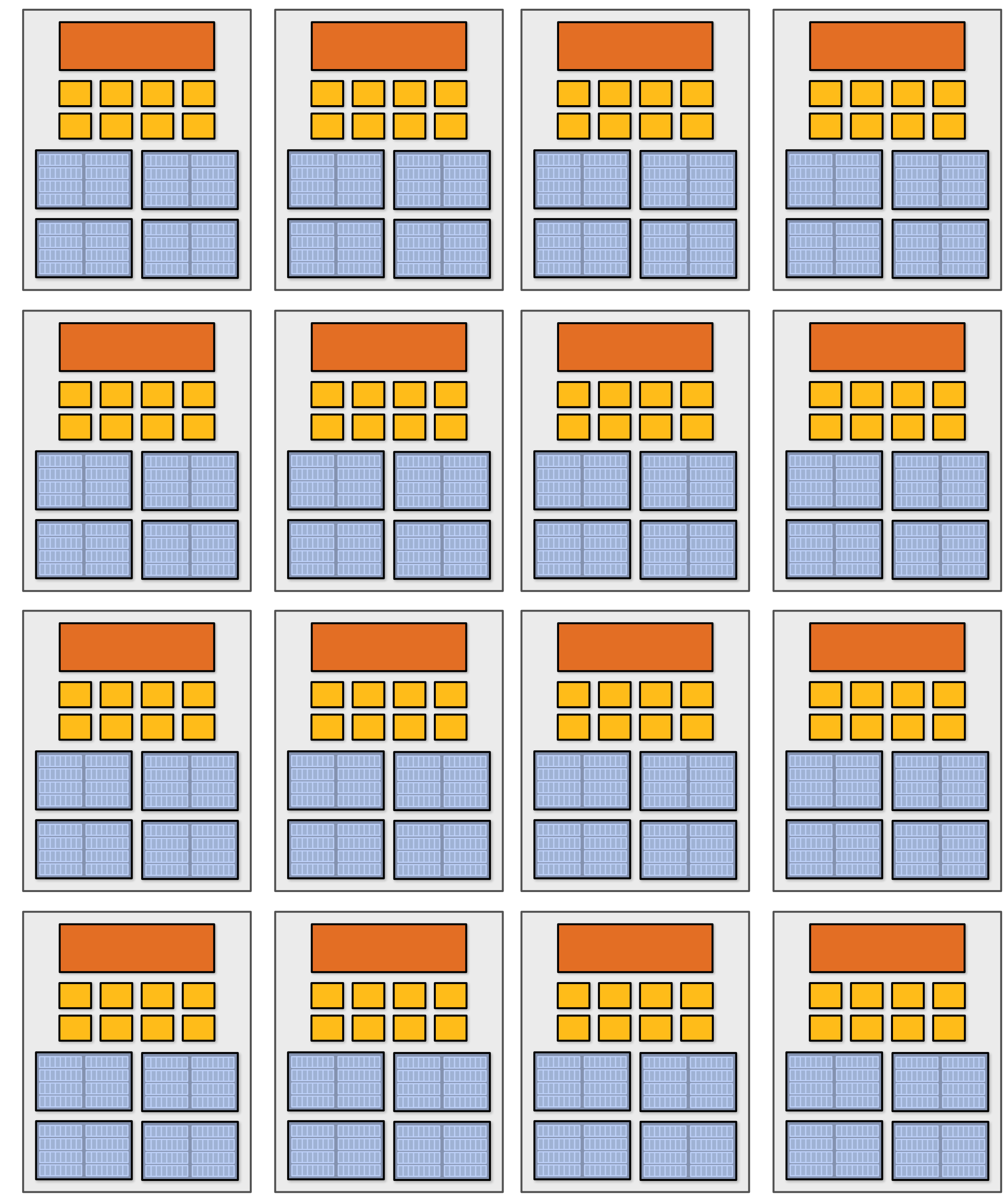
```
const int Nx = 12;
const int Ny = 6;

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);
```



GPU

CUDA blocks map to GPU cores (streaming multiprocessors)

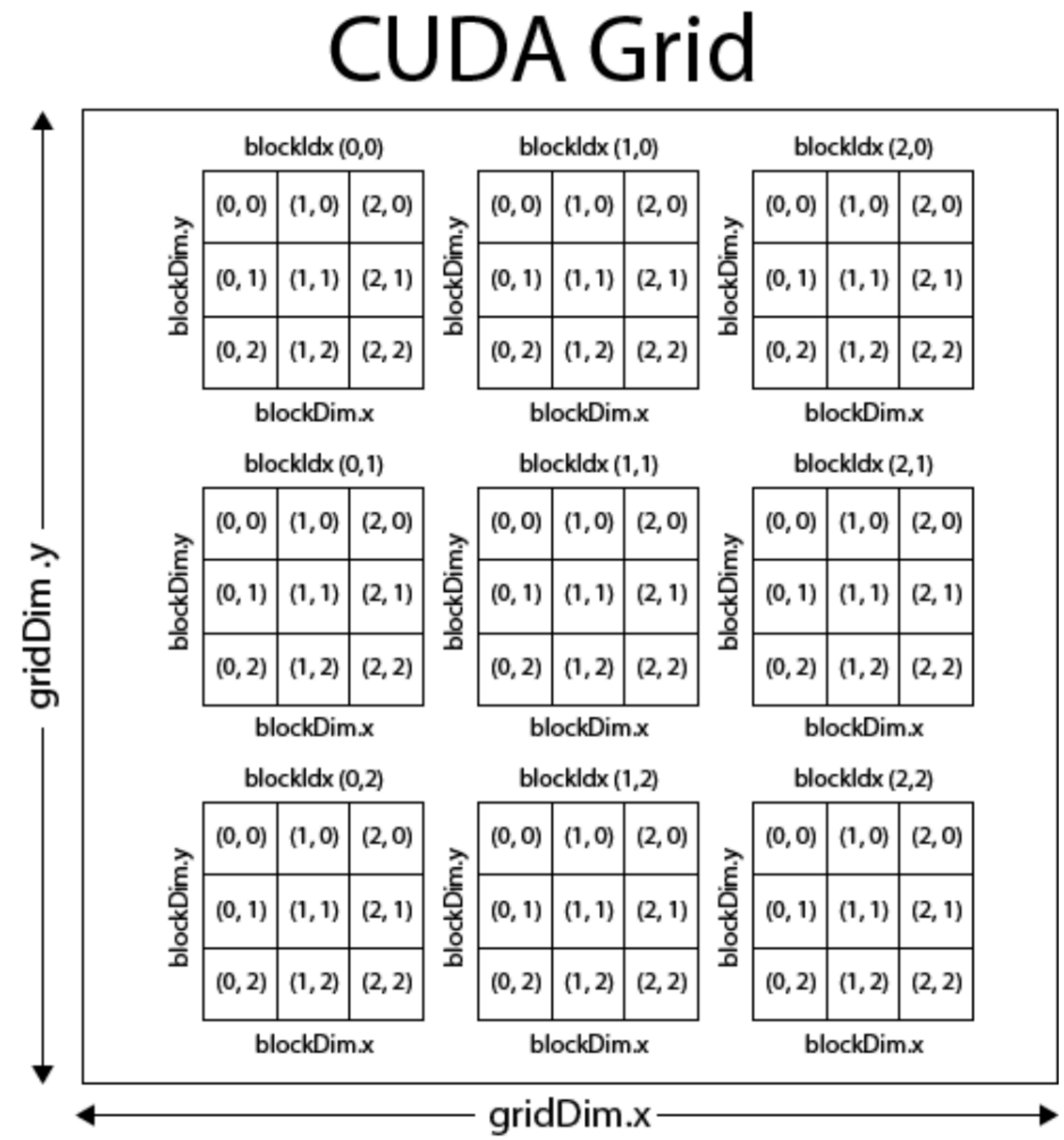




GPU

Grid, Block, and Thread

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



Why not have gridIdx and threadDim?

Basic CUDA syntax

Regular application thread running on CPU (the "host")

"Host" code : serial execution
Running as part of normal C/C++ application on CPU

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

```
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);
```

SPMD execution of device kernel function:

CUDA kernel definition

"CUDA device" code: kernel function (`__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`)

```
// 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];
}
```

Clear separation of host and device code

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

"Host" code : serial execution on CPU

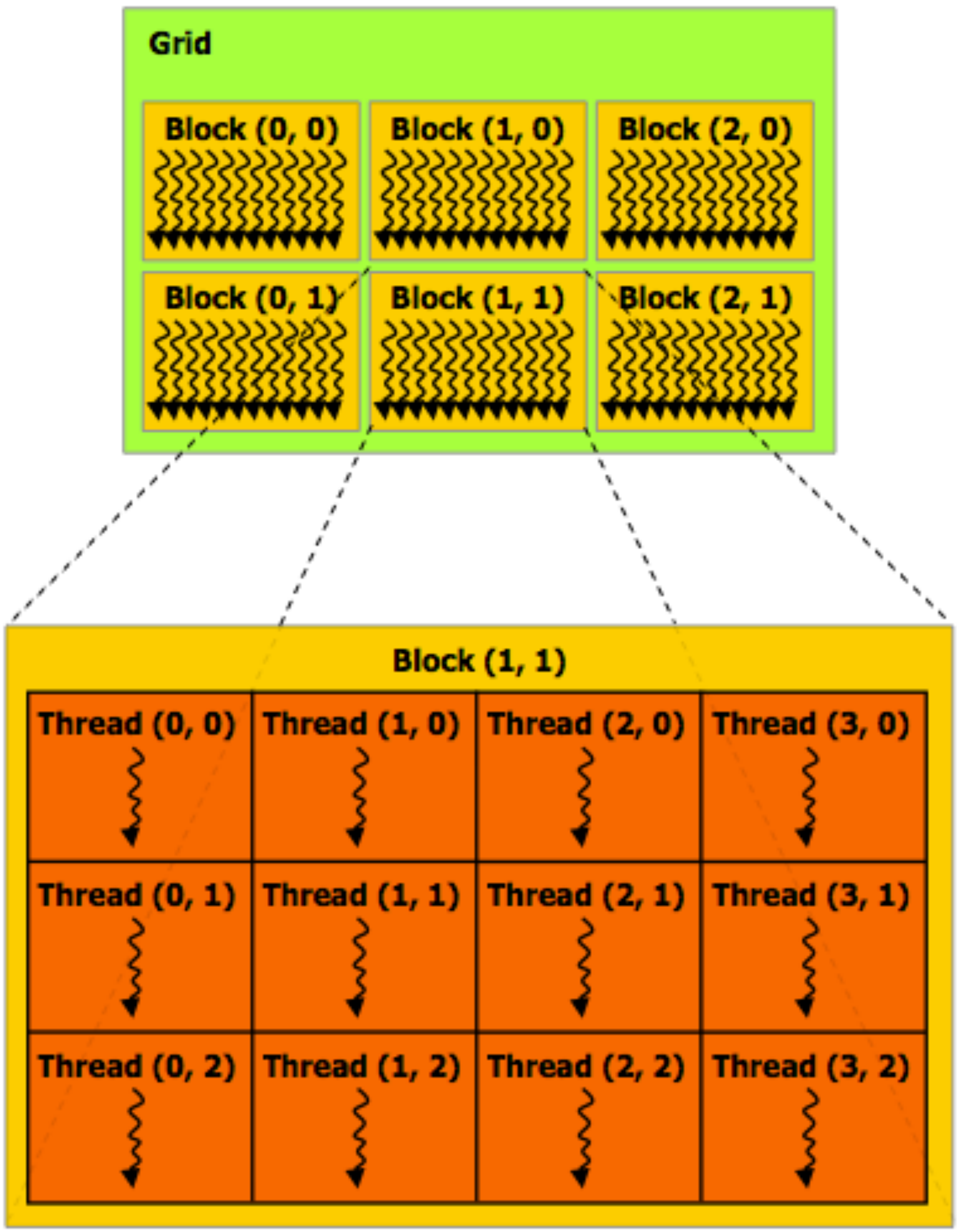
```
const int Nx = 12;
const int Ny = 6;

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" code (SPMD execution on GPU)

```
__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]);
}
```

Number of SPMD threads is explicit in program

Number of kernel invocations is not determined by size of data collection
(a kernel launch is not map(kernel, collection) as was the case with graphics shader programming)



Regular application thread running on CPU (the "host")

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

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks((Nx+threadsPerBlock.x-1)/threadsPerBlock.x,
               (Ny+threadsPerBlock.y-1)/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
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

CUDA 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];
}
```

CUDA execution model

Host
(serial execution)

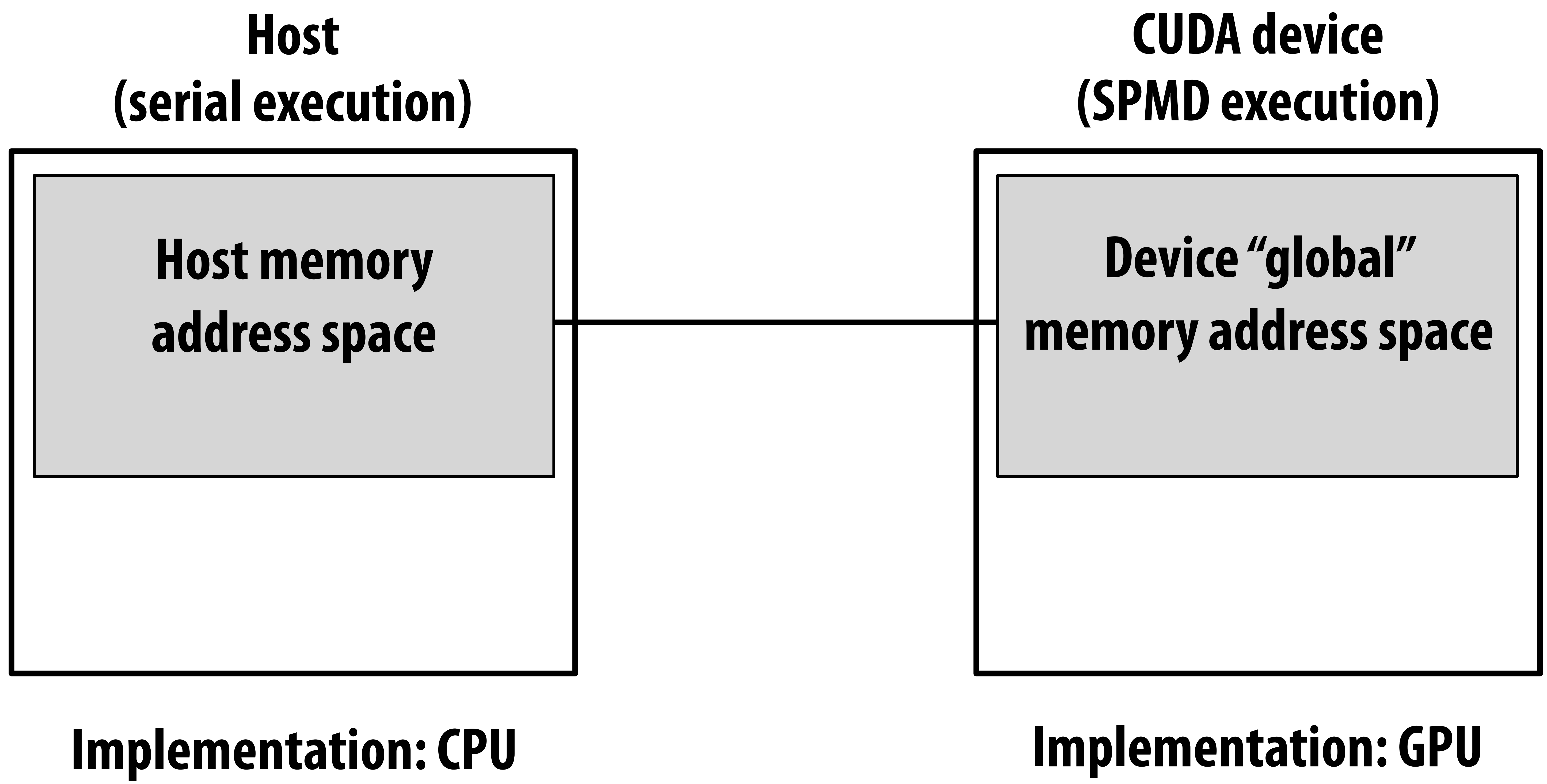
Implementation: CPU

CUDA device
(SPMD execution)

Implementation: GPU

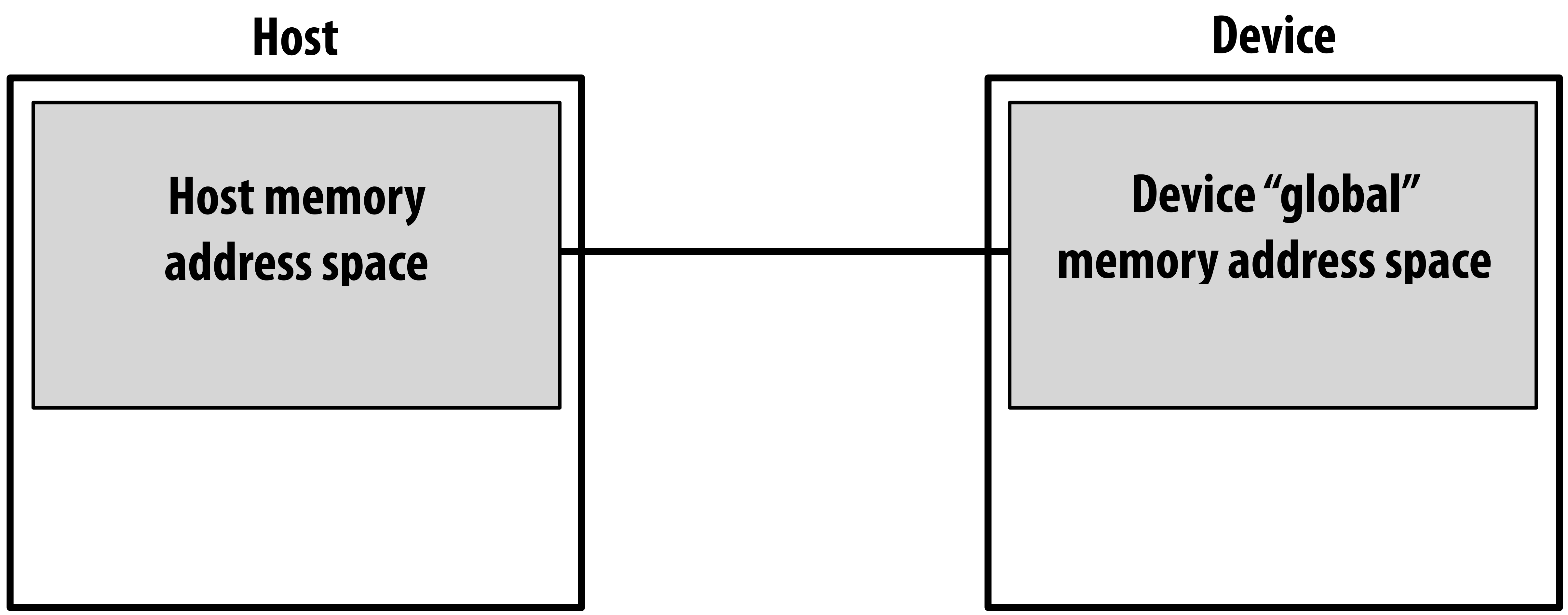
CUDA memory model

Distinct host and device address spaces



memcpy primitive

Move data between address spaces

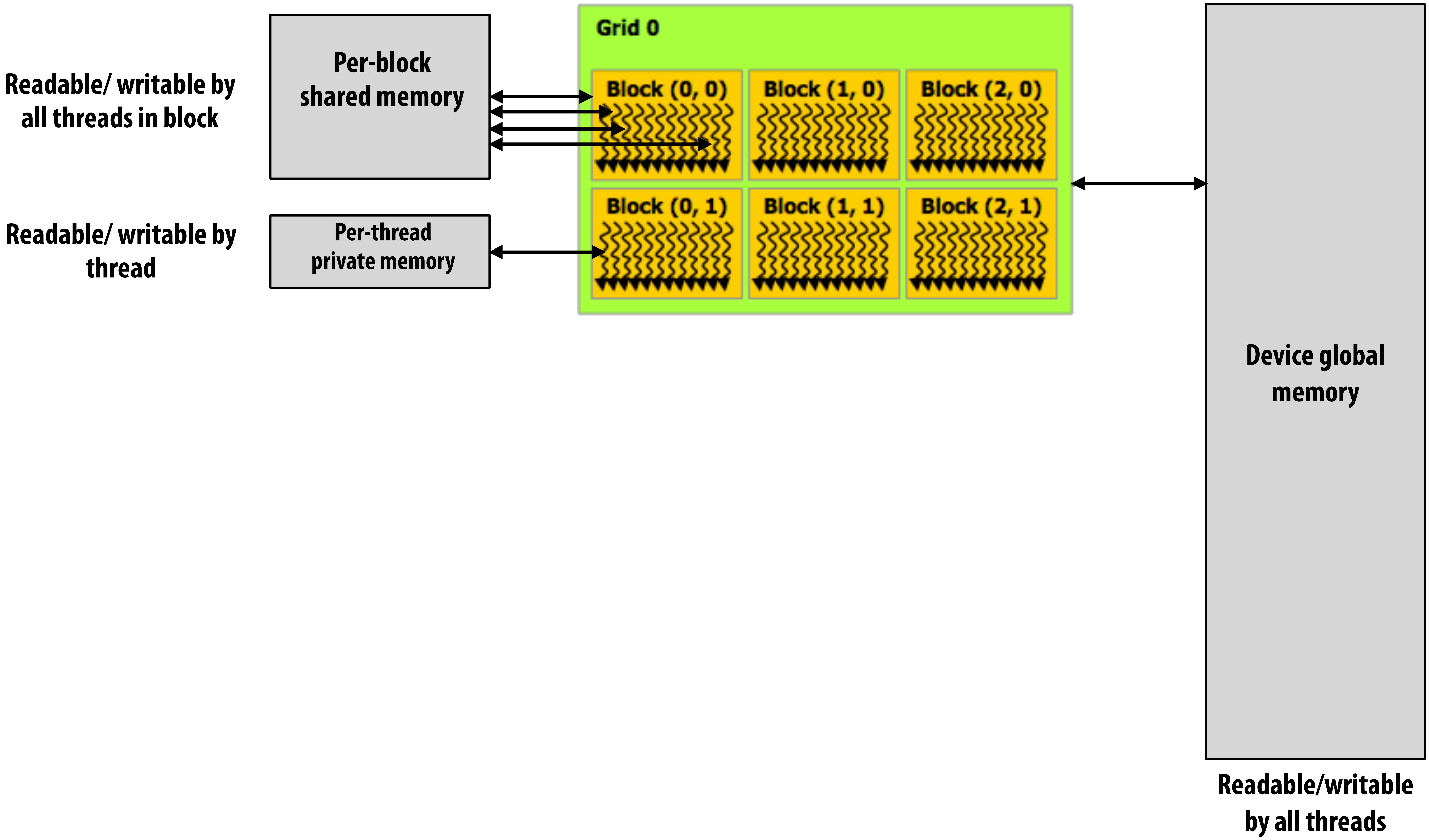



```
float* A = new float[N];       // allocate buffer in host mem

// 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.)
```

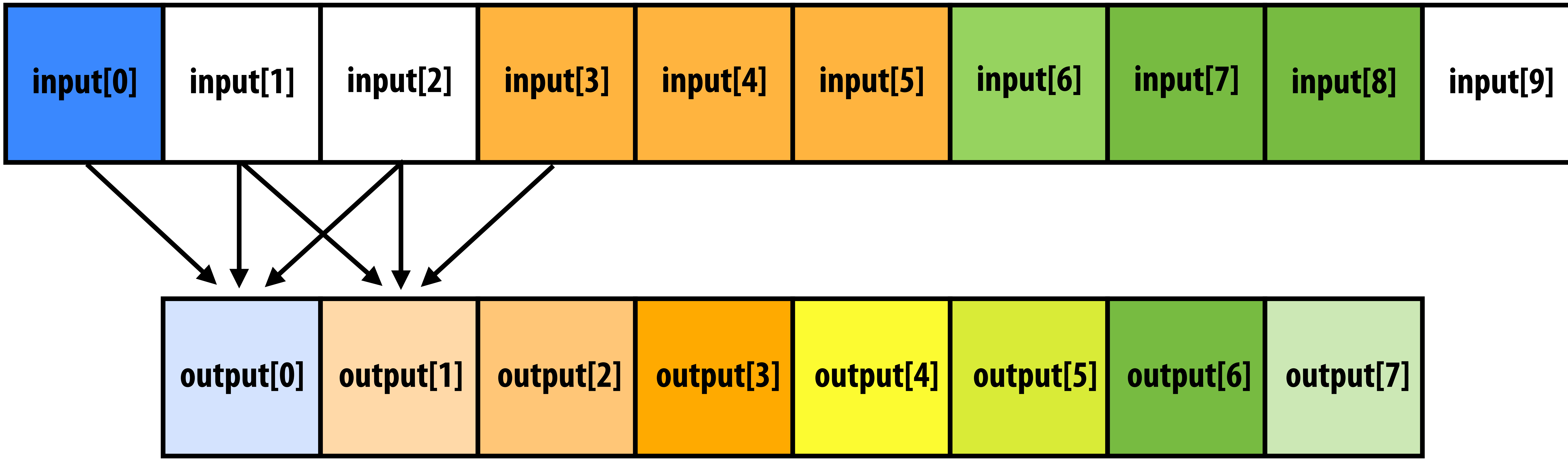
What does `cudaMemcpy` remind you of?

CUDA device memory model

Three distinct types of memory visible to kernels



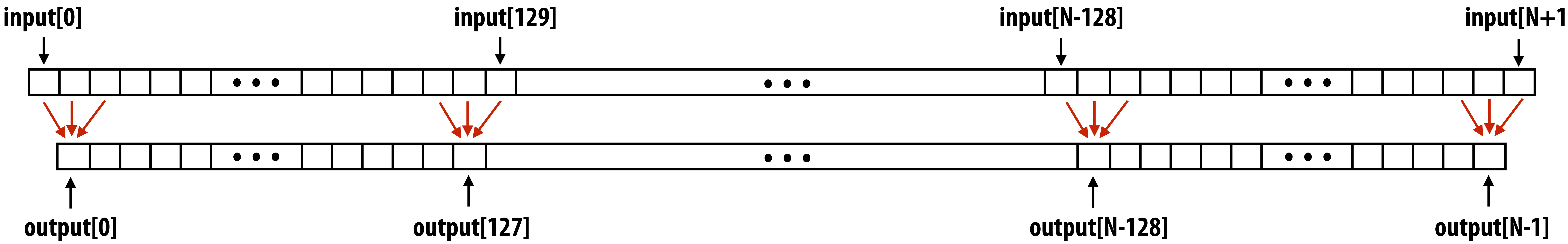
CUDA example: 1D convolution




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

1D convolution in CUDA (version 1)

One thread per output element



CUDA Kernel

```
#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

write result to global memory

Host code

```
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);
```

1D convolution in CUDA (version 2)

One thread per output element: stage input data in per-block shared memory

CUDA Kernel

```
#define THREADS_PER_BLK 128

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

   __shared__ float support[THREADS_PER_BLK+2];        // per-block allocation
   int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

   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 memory into shared memory
(total of 130 load instructions instead of 3 * 128 load instructions)

barrier (all threads in block)

each thread computes result for one element

write result to global memory

Host code

```
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);
```

CUDA synchronization constructs

- **__syncthreads()**
  - Barrier: wait for all threads in the block to arrive at this point

- **Atomic operations**
  - e.g., `float atomicAdd(float* addr, float amount)`
  - Atomic operations on both global memory and shared memory variables

- **Host/device synchronization**
  - Implicit barrier across all threads at return of kernel

Recap: CUDA abstractions

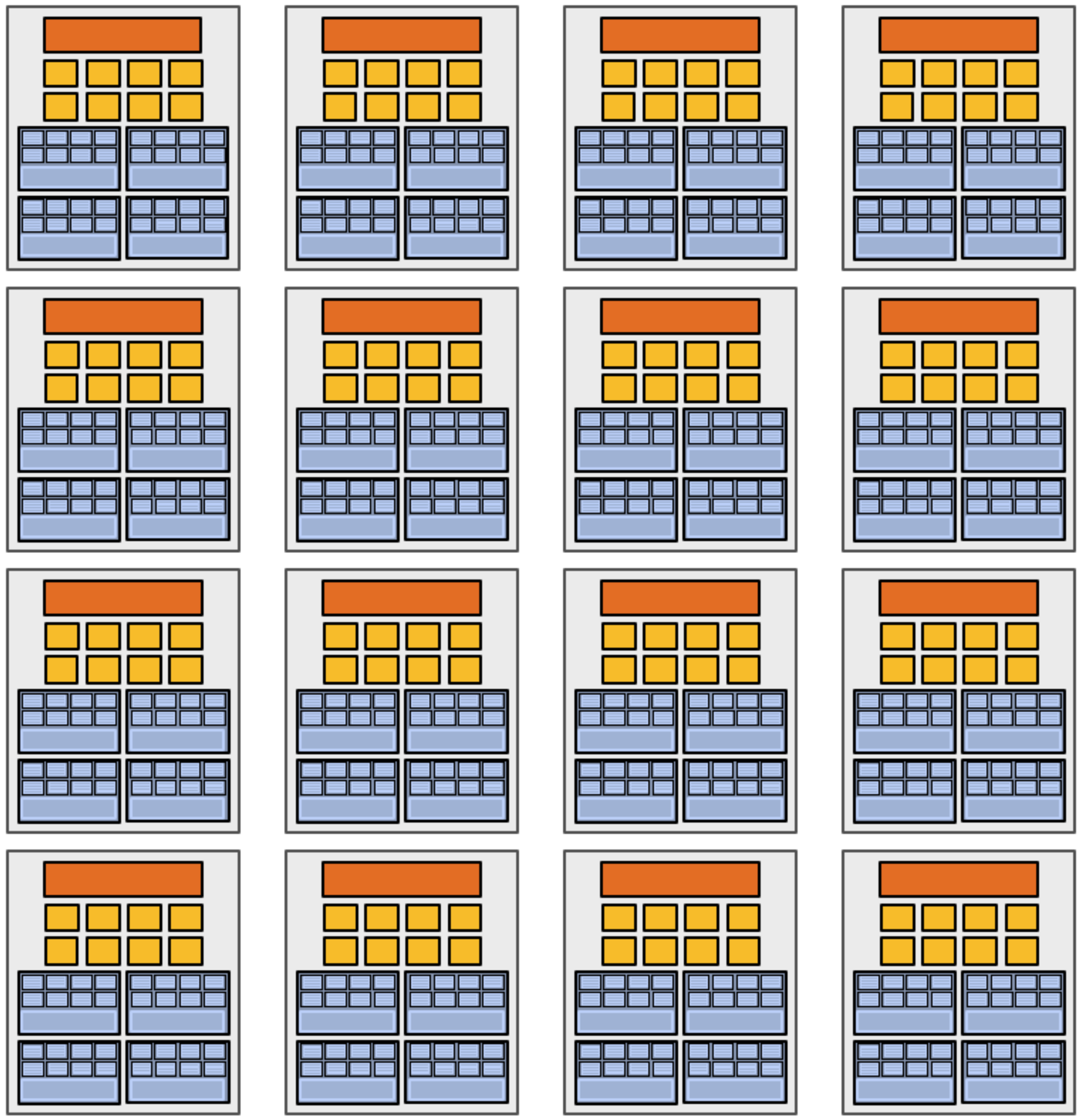
- Execution: thread hierarchy
  - Bulk launch of many threads (this is imprecise... I'll clarify later)
  - Two-level hierarchy: threads are grouped into thread blocks

- Distributed address space
  - Built-in 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)

Assigning work



High-end GPU
(16 cores)

Mid-range GPU
(6 cores)

Want CUDA program to run on all of these GPUs without modification

Note: there is no concept of `num_cores` in the CUDA programs I have shown you. (CUDA thread launch is similar in spirit to a forall loop in data parallel model examples)

CUDA compilation

```
#define THREADS_PER_BLK 128

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

   __shared__ float support[THREADS_PER_BLK+2];  // per block allocation
   int index = blockIdx.x * blockDim.x + threadIdx.x; // thread local var

   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;
}
```

A compiled CUDA device binary includes:

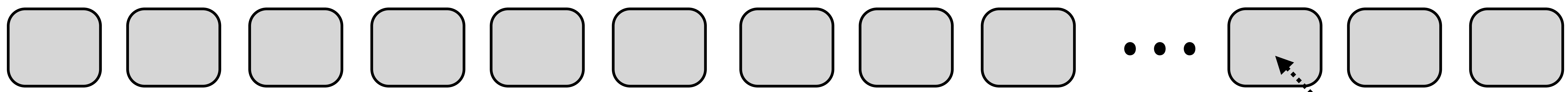
Program text (instructions)

Information about required resources:
- **128 threads per block**
- **B bytes of local data per thread**
- **130 floats (520 bytes) of shared space per thread block**

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

// property initialize contents of devInput here ...

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

launch 8K thread blocks

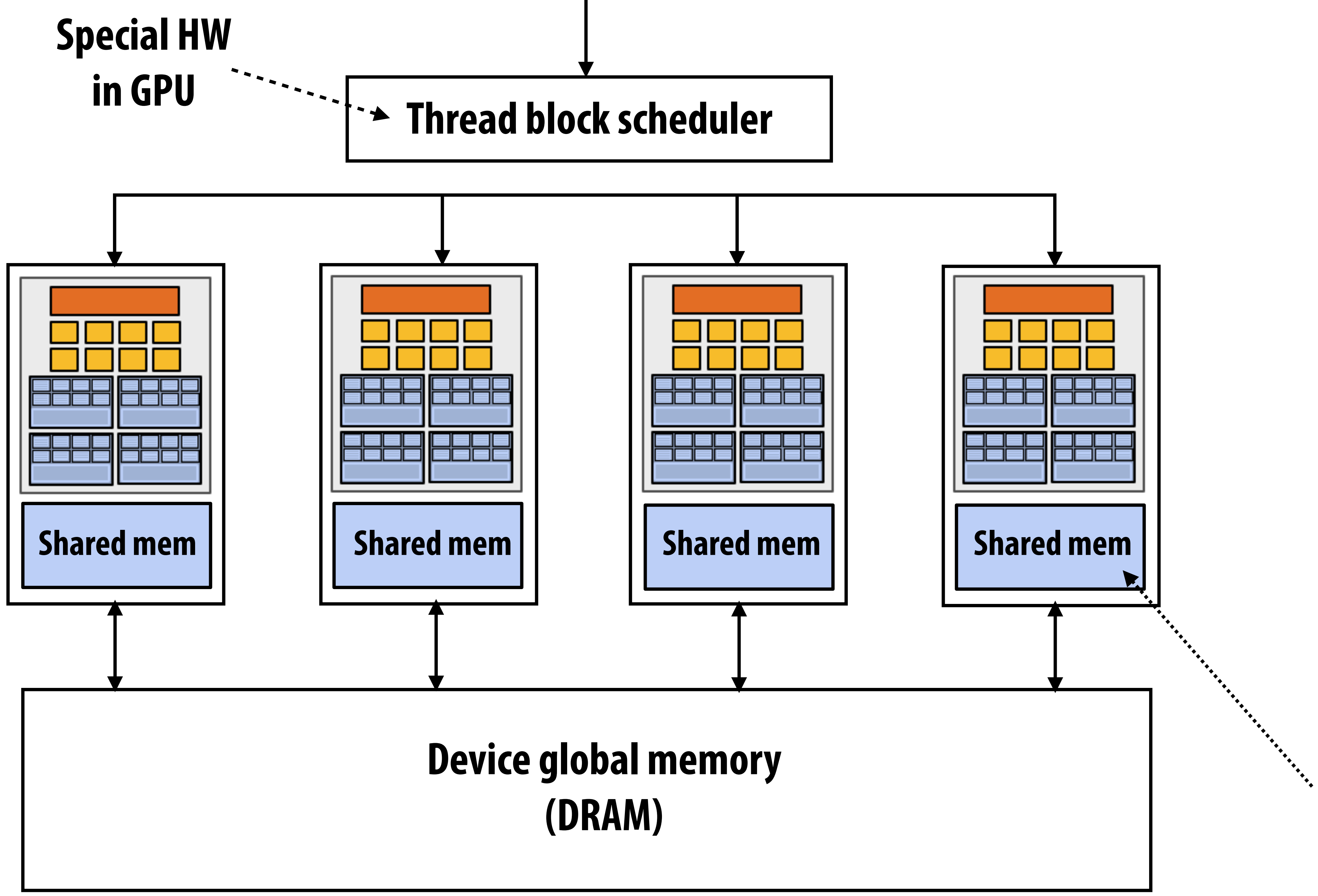
CUDA thread-block assignment



Grid of 8K `convolve` thread blocks (specified by kernel launch)

Block resource requirements:
(contained in compiled kernel binary)
128 threads
520 bytes of shared mem
(128 x B) bytes of local mem

Kernel launch command from host
`launch(blockDim, convolve)`



Major CUDA assumption: thread block execution can be carried out in any order (no dependencies between blocks)

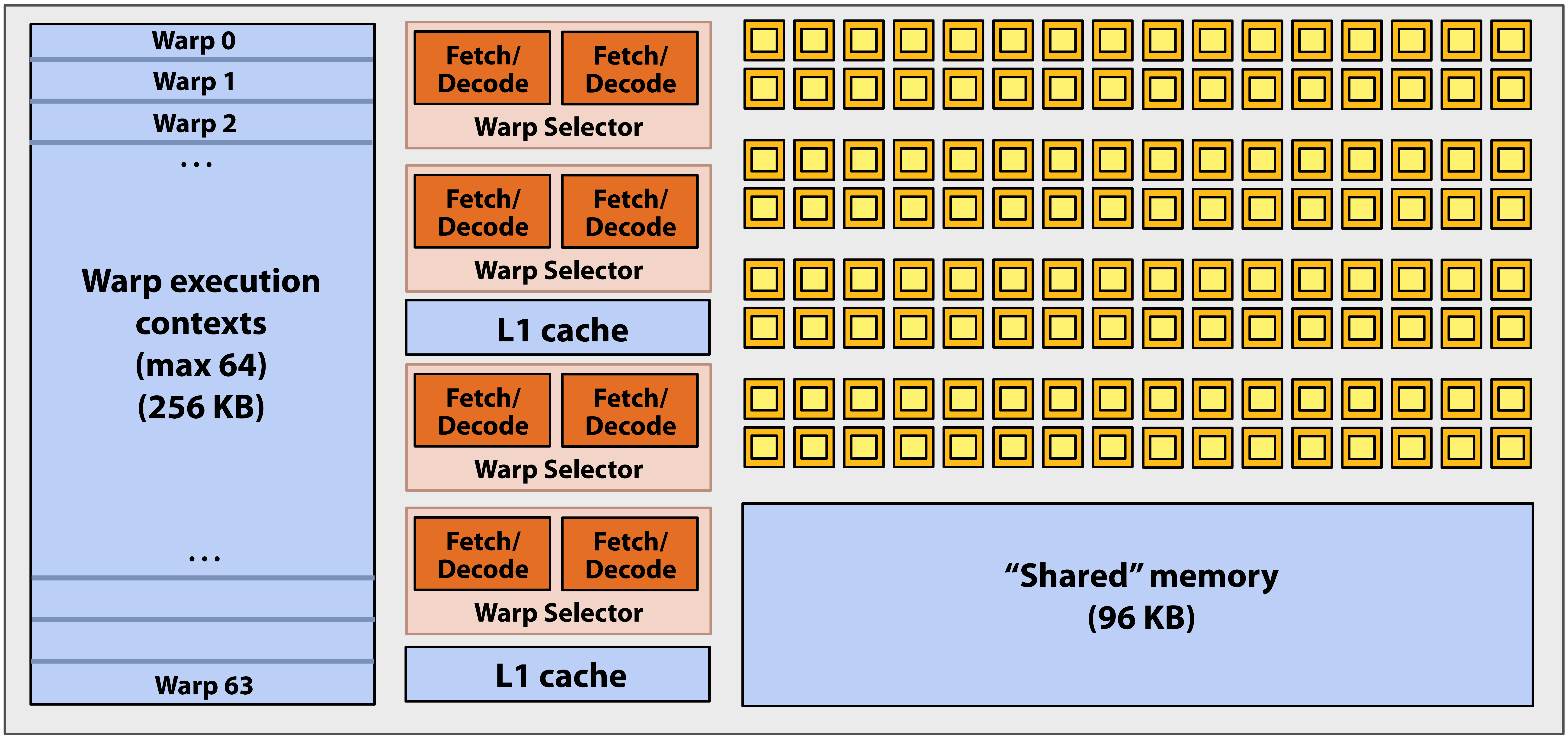
GPU implementation maps thread blocks ("work") to cores using a dynamic scheduling policy that respects resource requirements

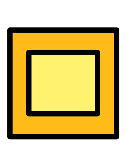
Quiz

NVIDIA GTX 980 (2014)

This is one NVIDIA Maxwell GM204 architecture SMM unit (one “core”)

A warp is a set of 32 threads
executing the same instruction

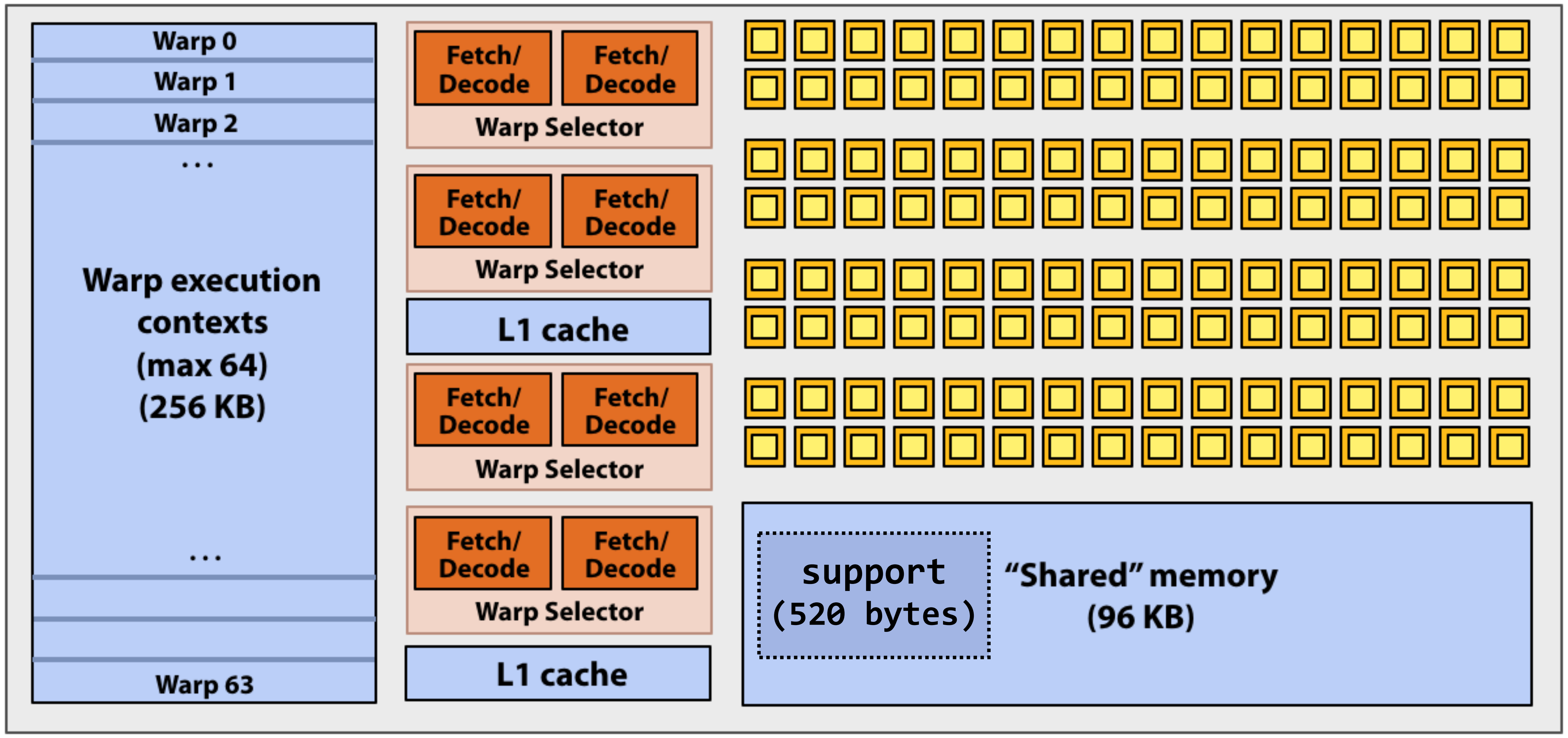


 = SIMD functional unit,
control shared across 32 units
(1 MUL-ADD per clock)

SMM resource limits:

- Max warp execution contexts: 64
  (2,048 total CUDA threads)
- 96 KB of shared memory

Running a single thread block on a SMM "core"




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   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
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result;
}
```

Recall, CUDA kernels execute as SPMD programs

On NVIDIA GPUs groups of 32 CUDA threads share an instruction stream. These groups called "warps".

A `convolve` thread block is executed by 4 warps (4 warps x 32 threads/warp = 128 CUDA threads per block)

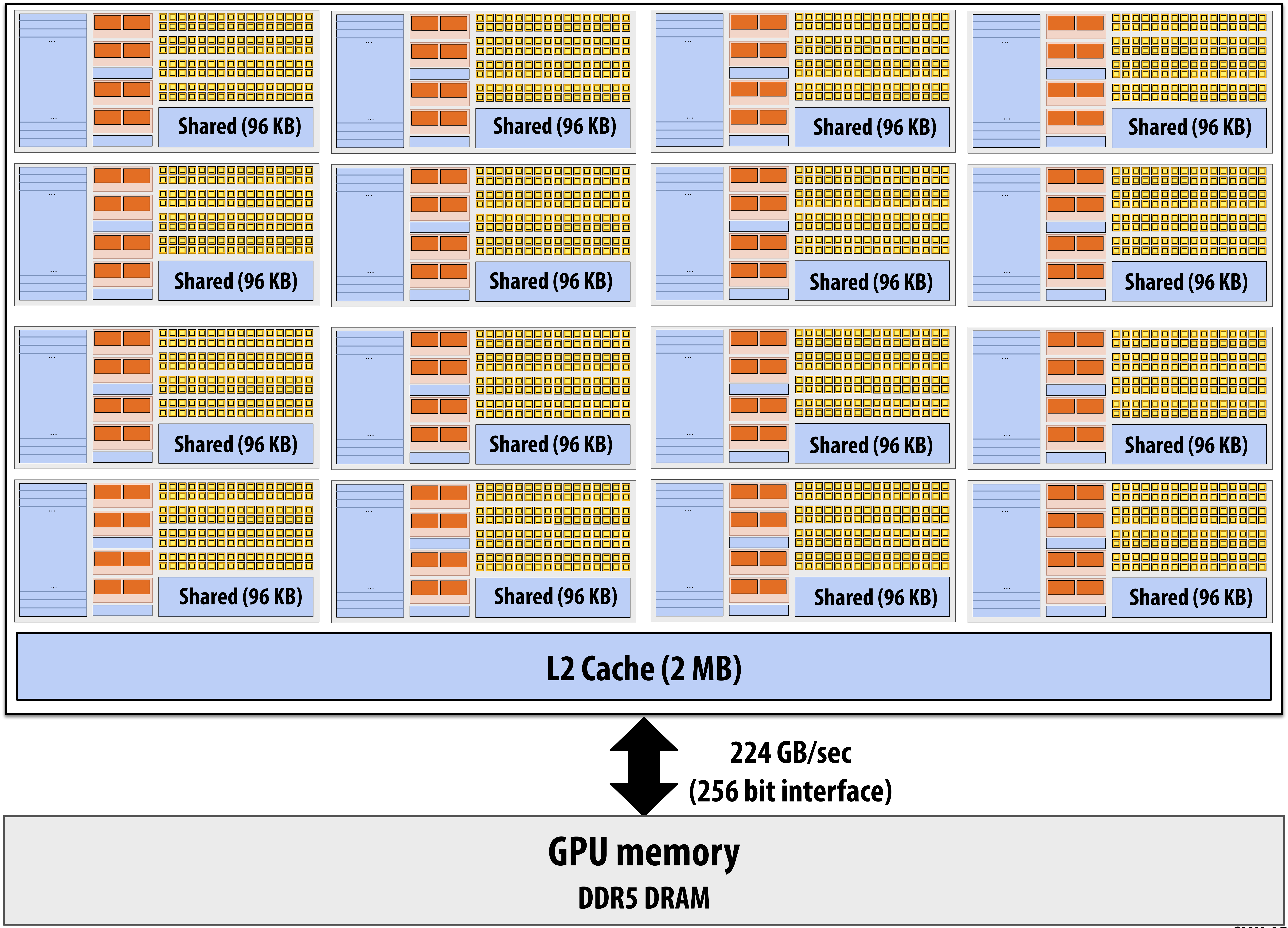
(Warps are an important GPU implementation detail, but not a CUDA abstraction!)

SMX core operation each clock:

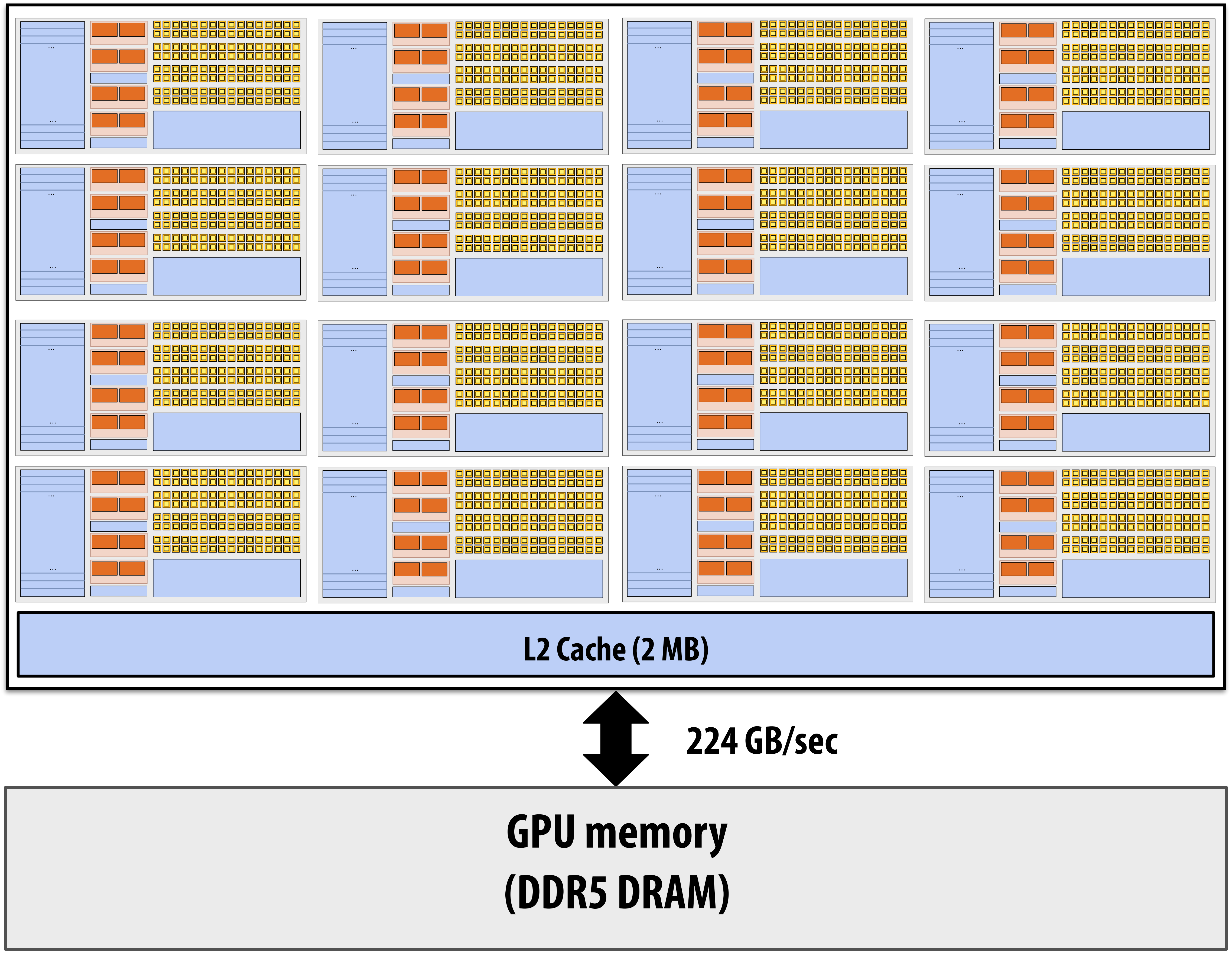
- **Select up to four runnable warps from 64 resident on SMM core (thread-level parallelism)**
- **Select up to two runnable instructions per warp (instruction-level parallelism) ***

* This diagram doesn't show additional units used to execute load/store instructions or "special math" (like pow, sin/cos, etc.)

NVIDIA GTX 980 (16 SMMs)



NVIDIA GTX 980 (2014)



1.1 GHz clock

16 SMM cores per chip

16 x 4 warps x 32 threads/warp
= 2,048 SIMD mul-add ALUs
= 4.6 TFLOPs

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

TDP: 165 watts

GTX 980 (2014) -> A100 (2020)

- SMMs remain the same
  - Clock speed: 1064 MHz -> 1110 MHz
  - Max warps per SMM: 64 -> 64
  - Threads per warp: 32 -> 32
  - Shared memory per SMM: 96KB -> 96KB (V100) -> 192 KB(A100)

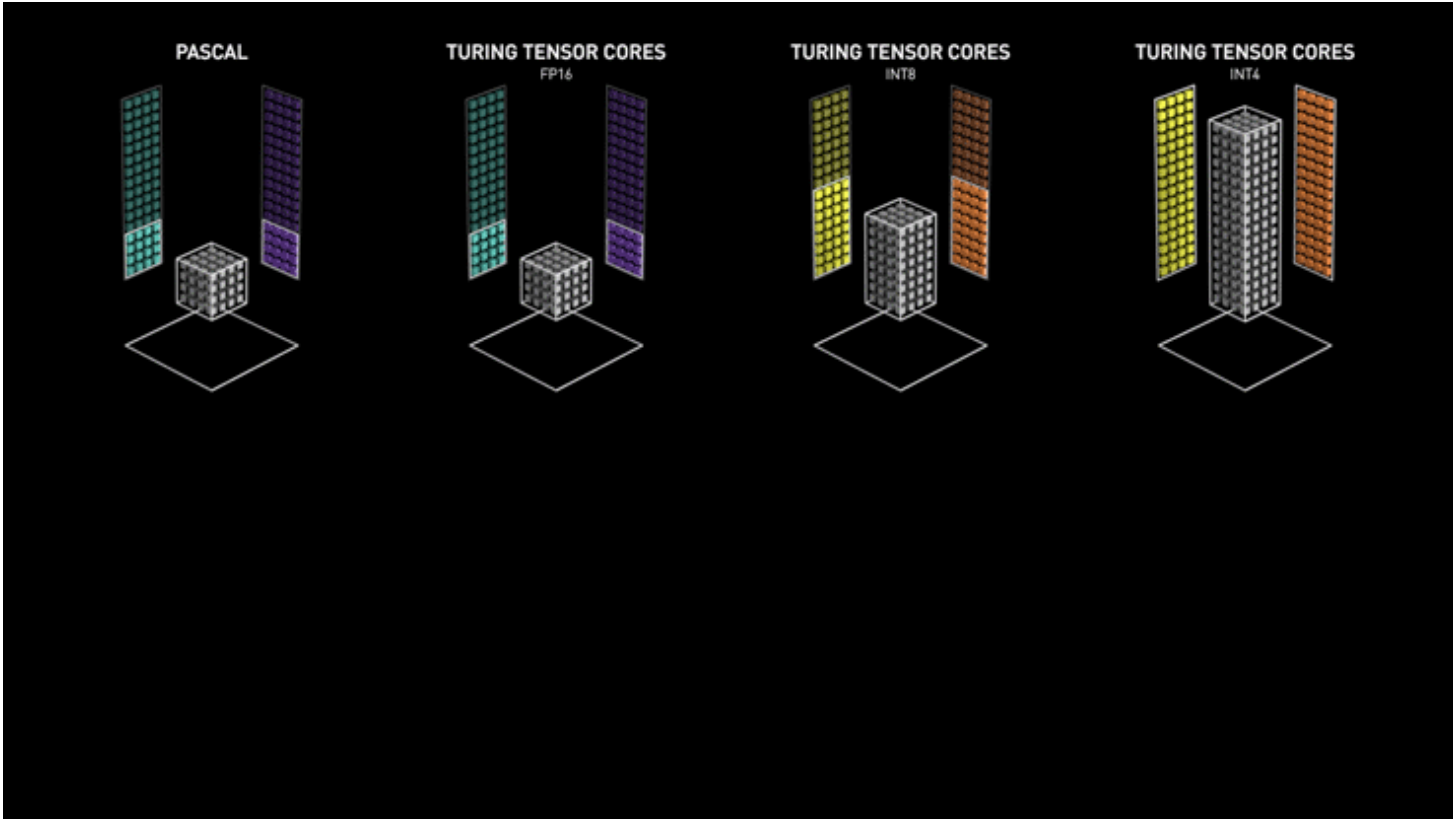
- Streaming multiprocessors: 16 SMMs -> 128 SMMs
- Peak performance: 4.6 TFLOPs -> 312 TFLOPs (largely because of tensor cores)

Tensor Cores

- **Matrix multiplication unit in SMM**

$$\underset{\text{FP16 or FP32}}{D} = \underset{\text{FP16}}{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}} \underset{\text{FP16}}{\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}} + \underset{\text{FP16 or FP32}}{\begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}}$$

Tensor Cores



Review

(If you understand this example you understand how CUDA programs run on a GPU)

Running the kernel

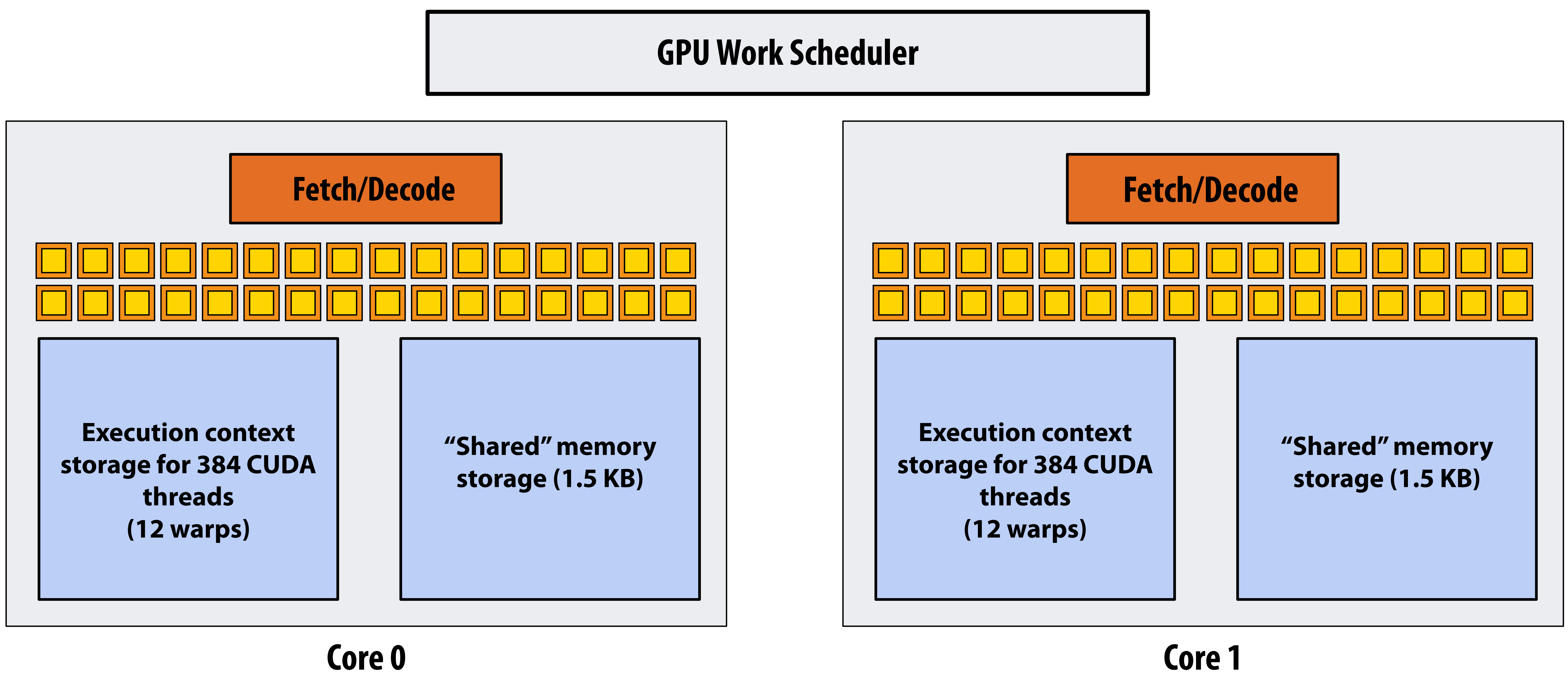
`convolve` hernel's execution requirements:

- Each thread block must execute 128 CUDA threads
- Each thread block must allocate 130 x sizeof(float) = 520 bytes of shared memory

Let's assume array size N is very large, so the host-side kernel launch generates thousands of thread blocks.

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

Let's run this program on the fictitious two-core GPU below.
(Note: my fictitious cores are much "smaller" than the GTX 980 cores discussed in lecture: fewer execution units, support for fewer active threads, less shared memory, etc.)

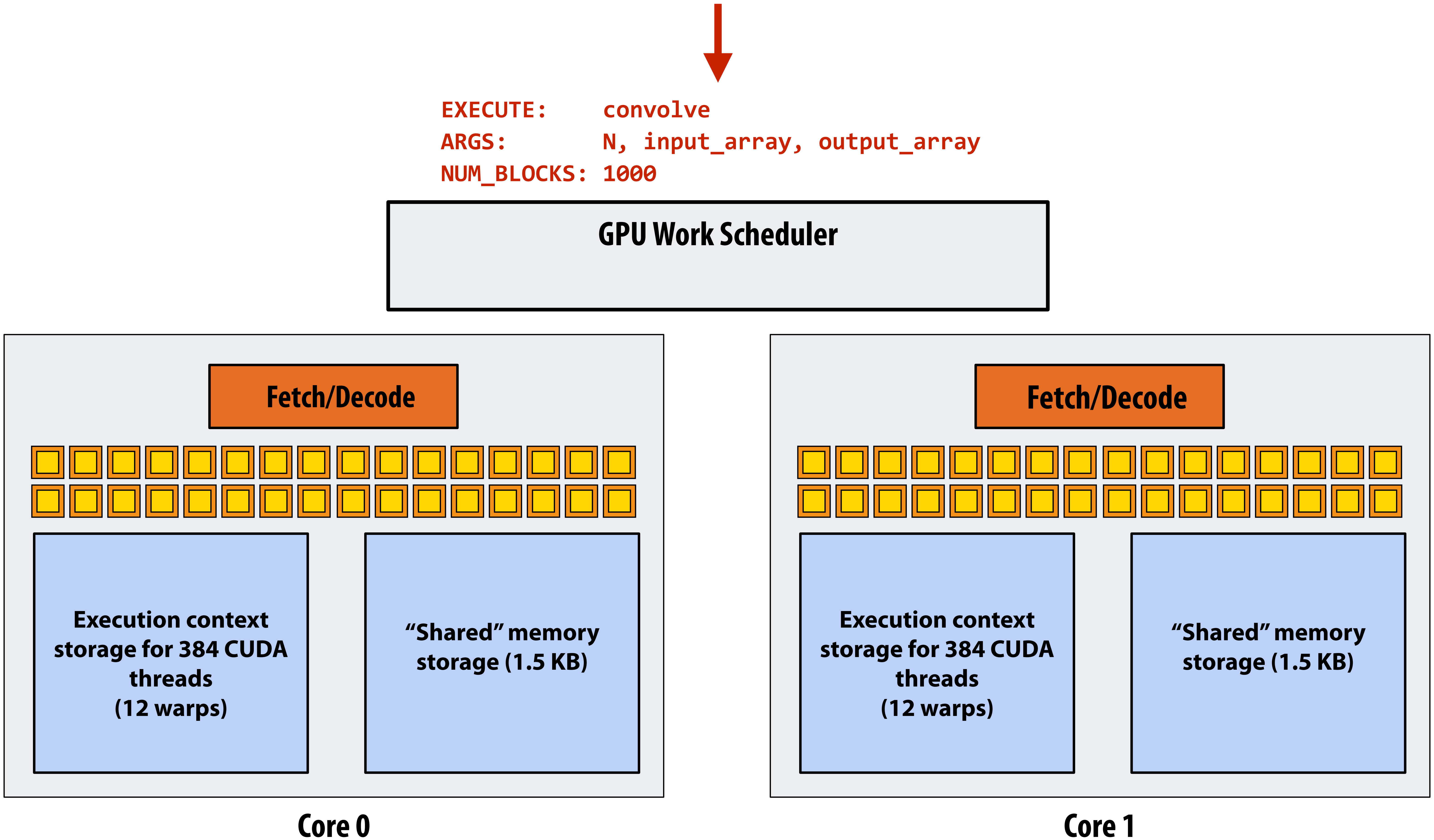


Running the CUDA kernel

Kernel's execution requirements:

- Each thread block must execute 128 CUDA threads
- Each thread block must allocate 130 x sizeof(float) = 520 bytes of shared memory

Step 1: host sends CUDA device (GPU) a command ("execute this kernel")



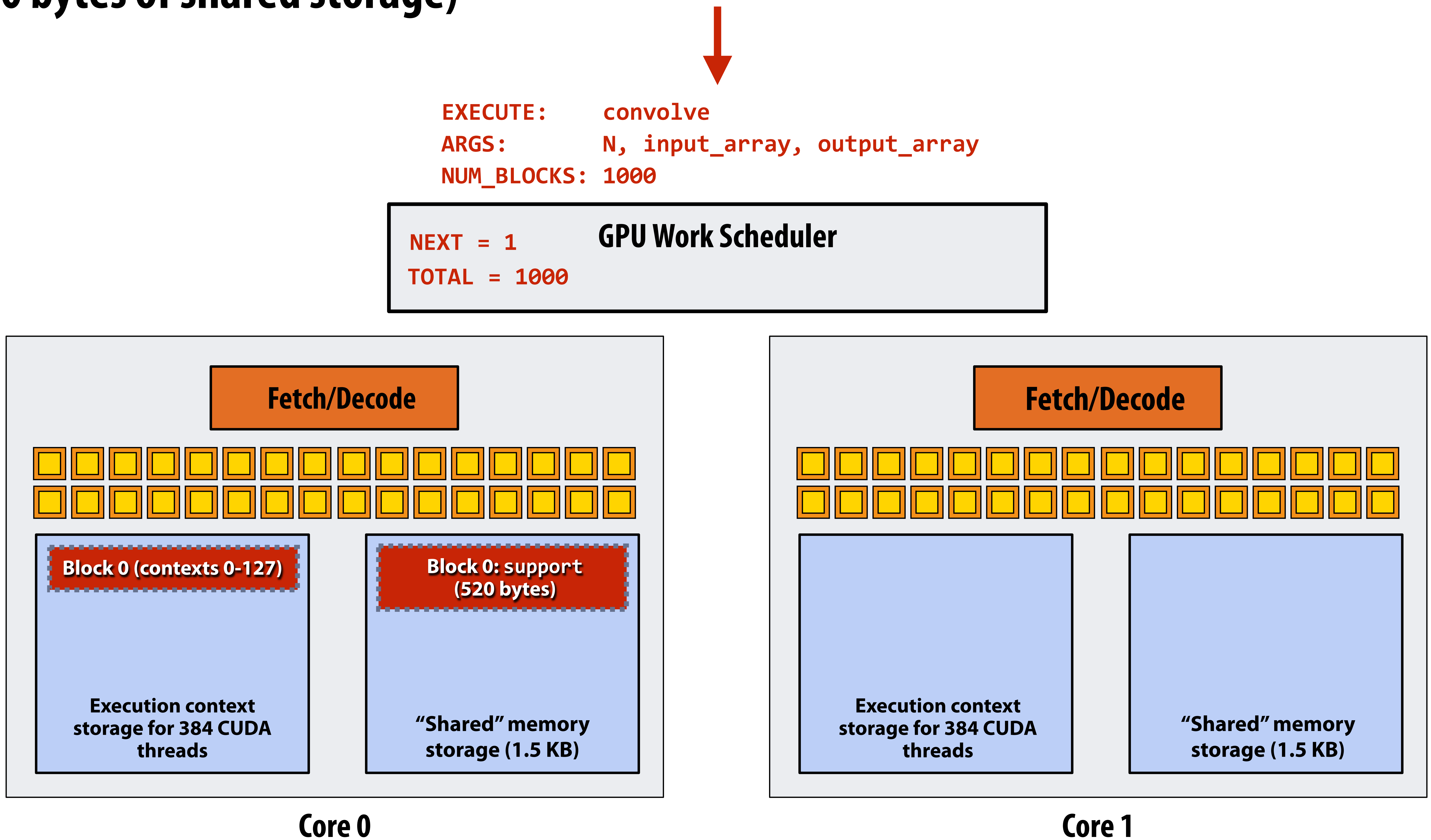
Running the CUDA kernel

Kernel's execution requirements:

Each thread block must execute 128 CUDA threads

Each thread block must allocate 130 x sizeof(float) = 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 storage)



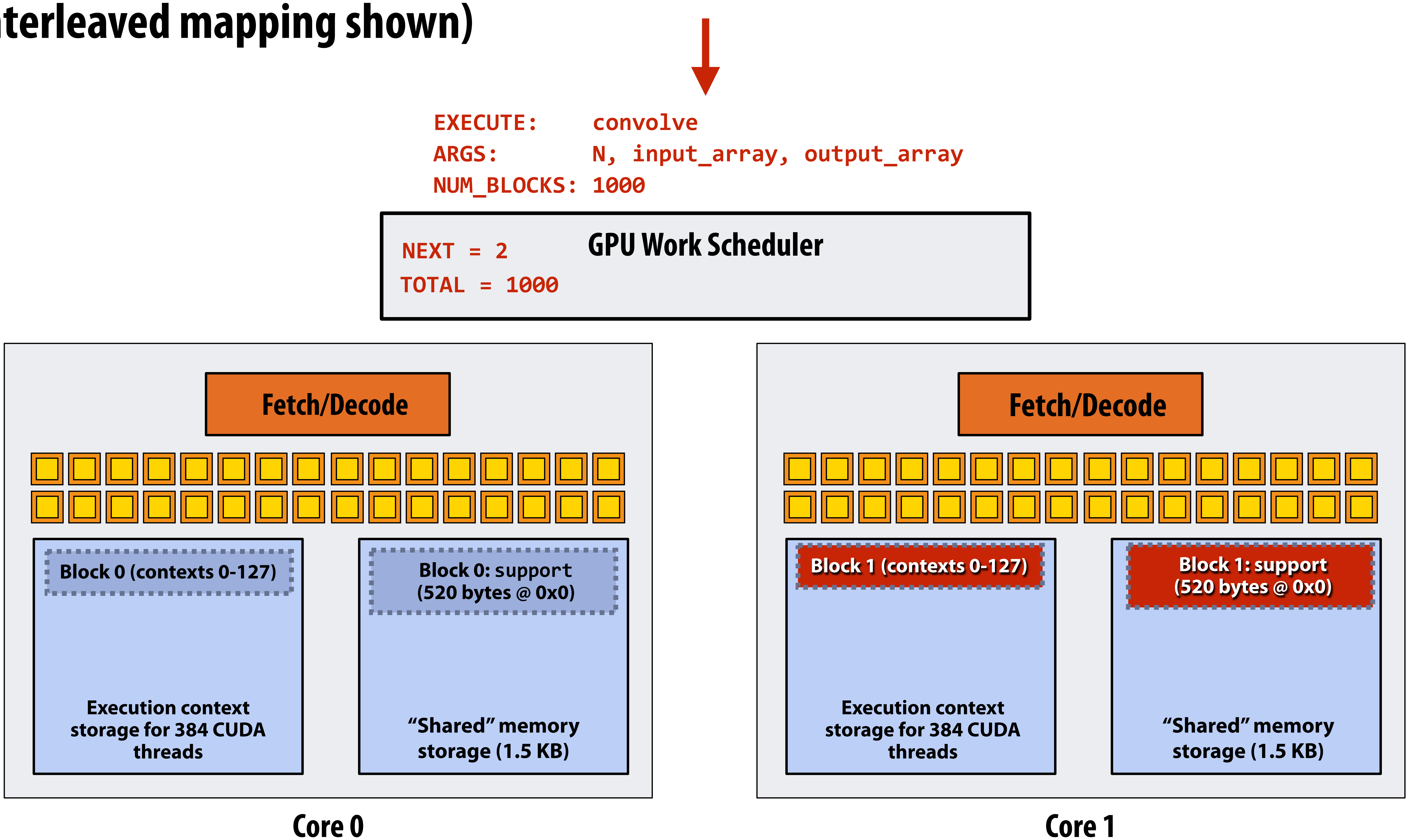
Running the CUDA kernel

Kernel's execution requirements:

Each thread block must execute 128 CUDA threads

Each thread block must allocate 130 x sizeof(float) = 520 bytes of shared memory

Step 3: scheduler continues to map blocks to available execution contexts (interleaved mapping shown)

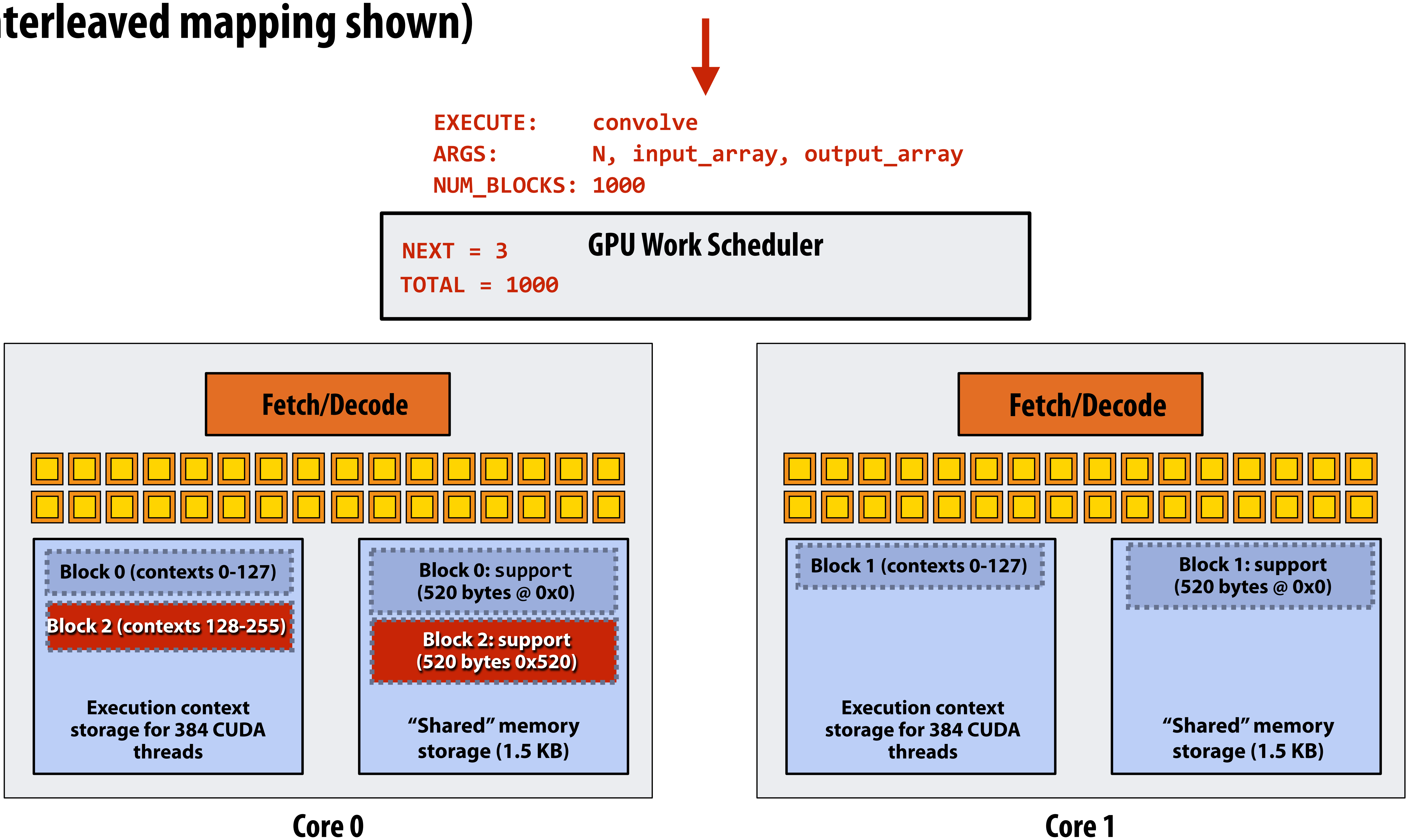


Running the CUDA kernel

Kernel's execution requirements:

- Each thread block must execute 128 CUDA threads
- Each thread block must allocate 130 x sizeof(float) = 520 bytes of shared memory

Step 3: scheduler continues to map blocks to available execution contexts (interleaved mapping shown)



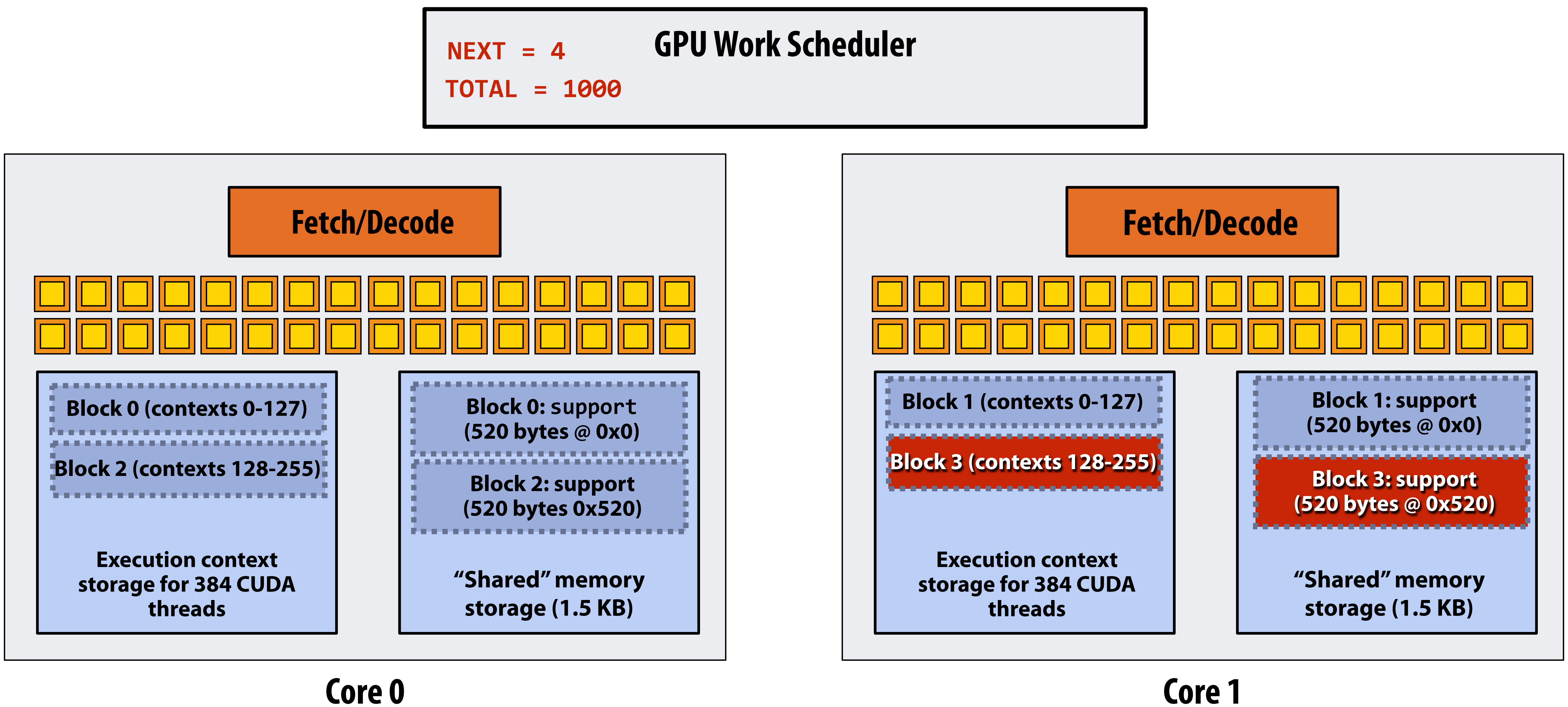
Running the CUDA kernel

Kernel's execution requirements:

- Each thread block must execute 128 CUDA threads
- Each thread block must allocate 130 x sizeof(float) = 520 bytes of shared memory

Step 3: scheduler continues to map blocks to available execution contexts (interleaved mapping shown).
Only two thread blocks fit on a core
(third block won't fit due to insufficient shared storage 3 x 520 bytes > 1.5 KB)

```
EXECUTE:    convolve
ARGS:       N, input_array, output_array
NUM_BLOCKS: 1000
```



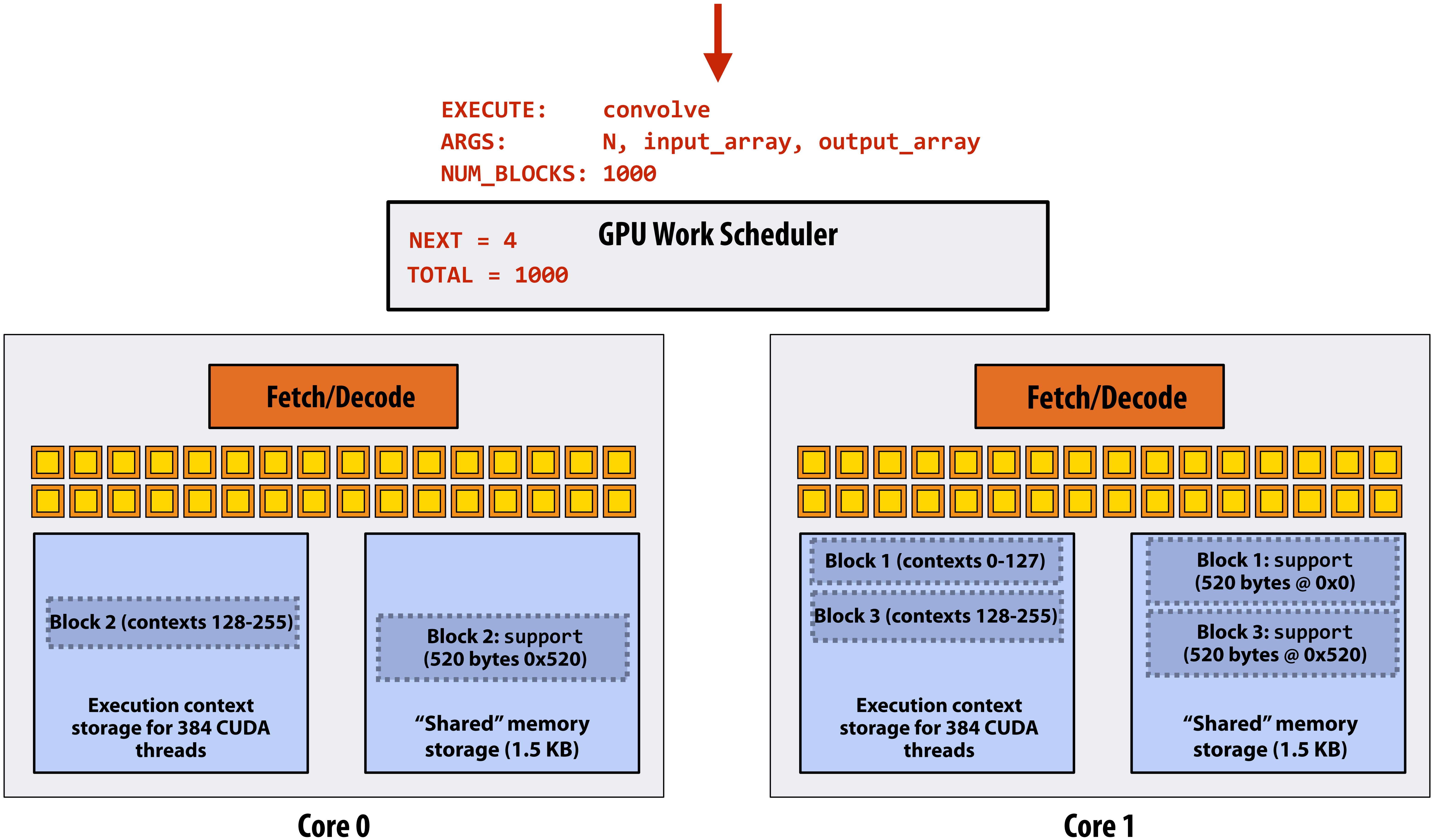
Running the CUDA kernel

Kernel's execution requirements:

Each thread block must execute 128 CUDA threads

Each thread block must allocate 130 x sizeof(float) = 520 bytes of shared memory

Step 4: thread block 0 completes on core 0



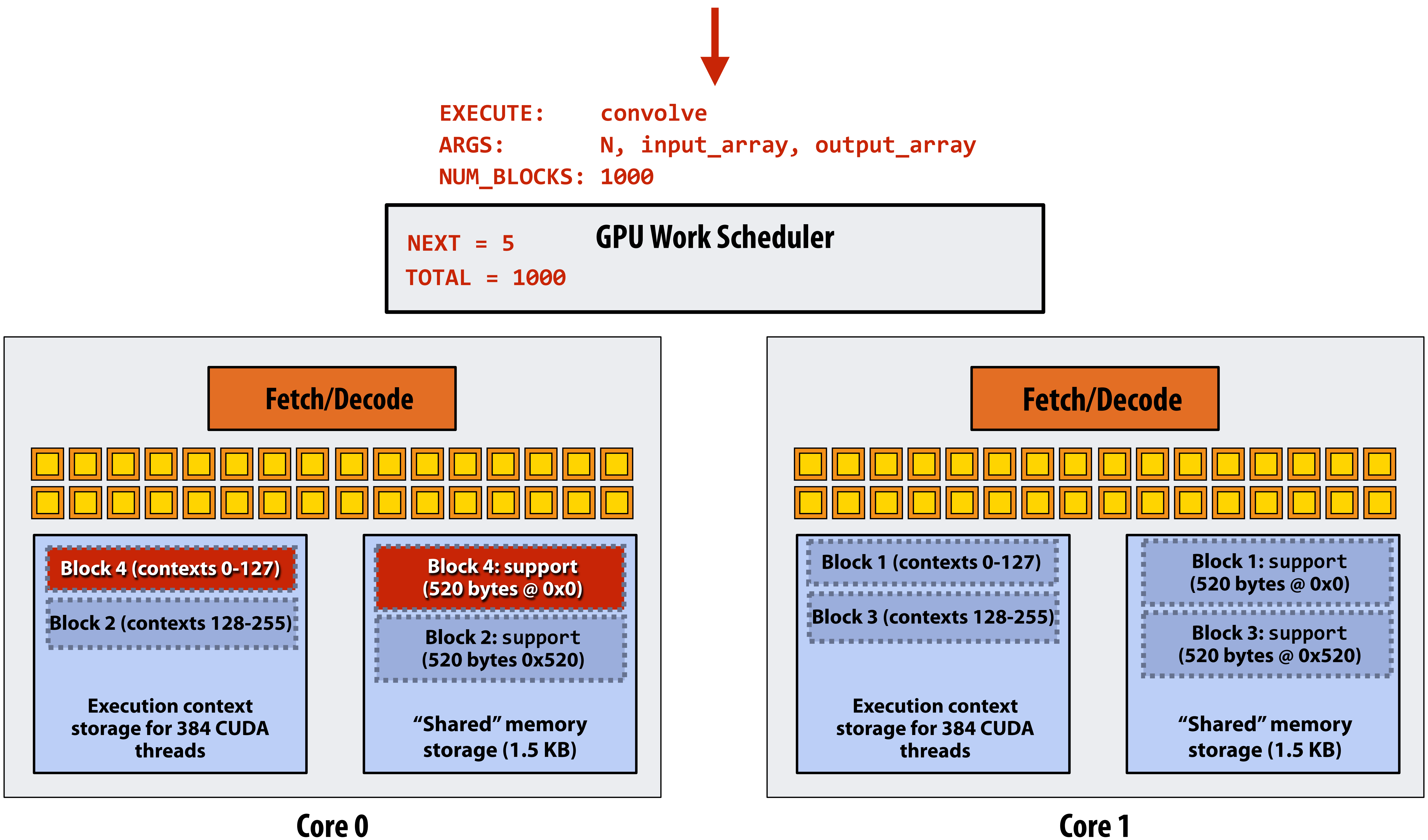
Running the CUDA kernel

Kernel's execution requirements:

Each thread block must execute 128 CUDA threads

Each thread block must allocate 130 x sizeof(float) = 520 bytes of shared memory

Step 5: block 4 is scheduled on core 0 (mapped to execution contexts 0-127)



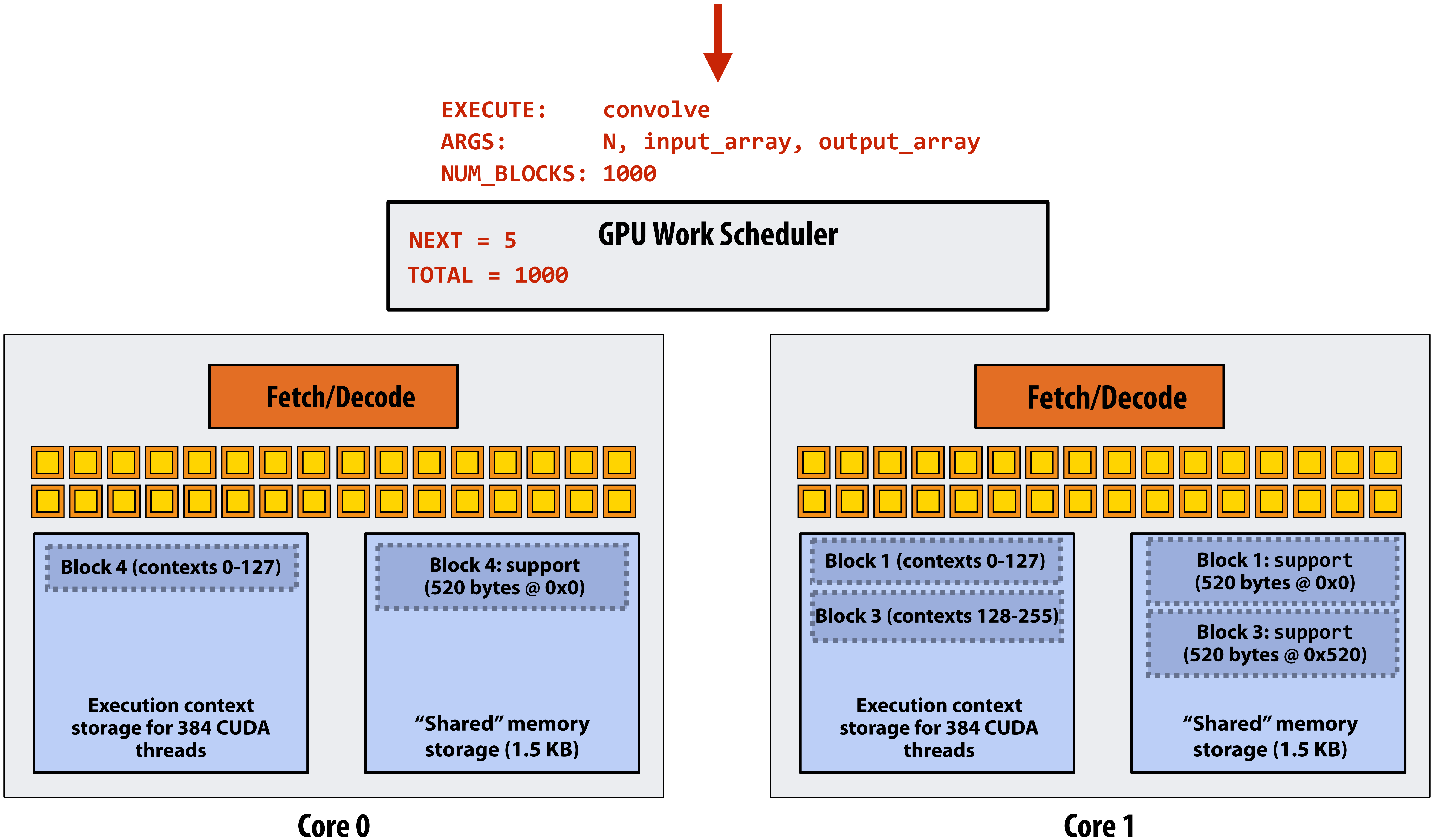
Running the CUDA kernel

Kernel's execution requirements:

Each thread block must execute 128 CUDA threads

Each thread block must allocate 130 x sizeof(float) = 520 bytes of shared memory

Step 6: thread block 2 completes on core 0

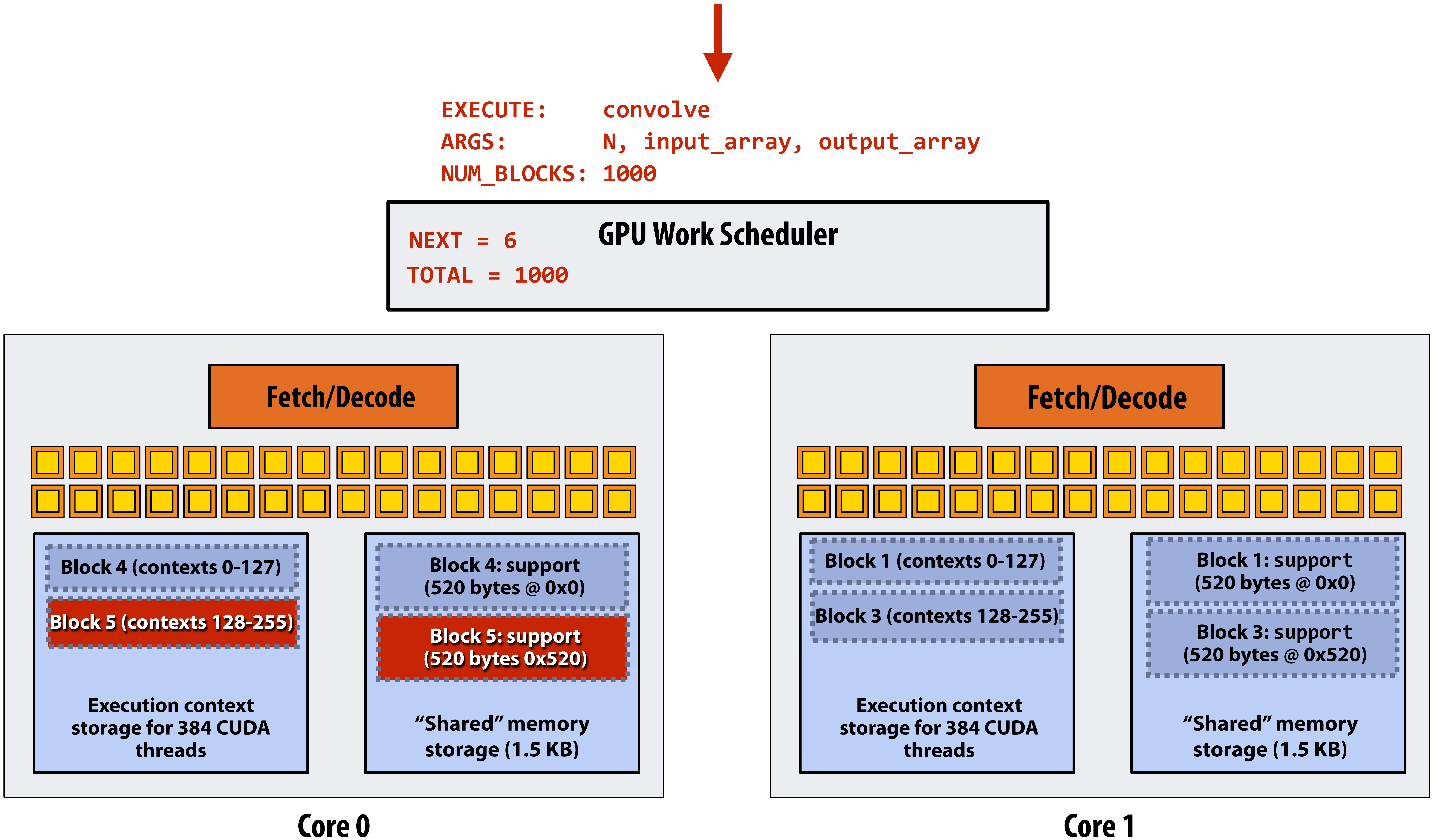


Running the CUDA kernel

Kernel's execution requirements:

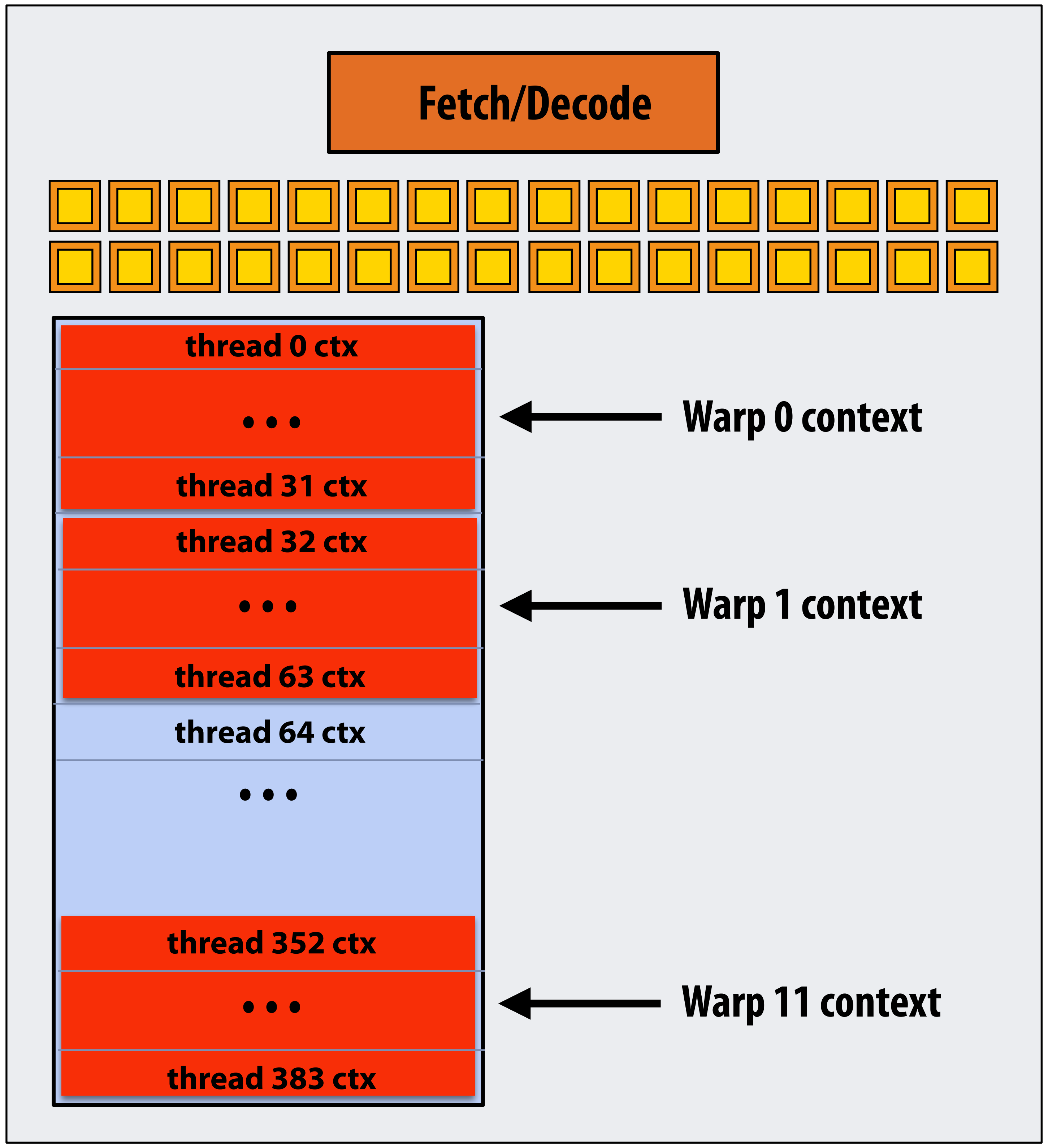
- Each thread block must execute 128 CUDA threads
- Each thread block must allocate 130 x sizeof(float) = 520 bytes of shared memory

Step 7: thread block 5 is scheduled on core 0 (mapped to execution contexts 128-255)



Review: what is a "warp"?

- A warp is a CUDA implementation detail on NVIDIA GPUs
- On modern NVIDIA hardware, groups of 32 CUDA threads in a thread block are executed simultaneously using 32-wide SIMD execution.



In this fictitious NVIDIA GPU example:
Core maintains contexts for 12 warps
Selects one warp to run each clock

Review: what is a "warp"?

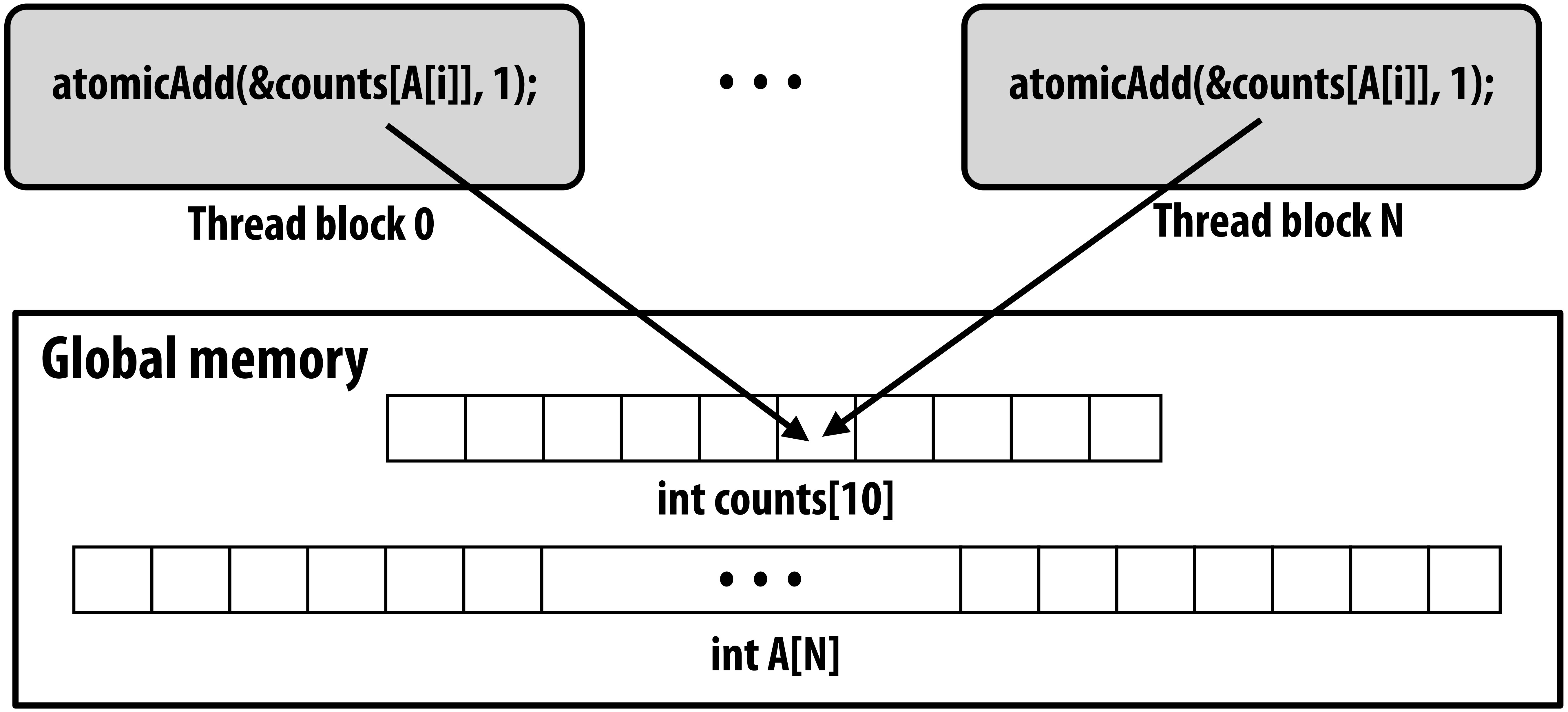
- A warp is a CUDA implementation detail on NVIDIA GPUs
- On modern NVIDIA hardware, groups of 32 CUDA threads in a thread block are executed simultaneously using 32-wide SIMD execution.
  - These 32 logical CUDA threads share an instruction stream and therefore performance can suffer due to divergent execution.
  - This mapping is similar to how ISPC runs program instances in a gang.
- The group of 32 threads sharing an instruction stream is called a warp.
  - In a thread block, threads 0-31 fall into the same warp (so do threads 32-63, etc.)
  - Therefore, a thread block with 256 CUDA threads is mapped to 8 warps.
  - Each "SMM" core in the GTX 980 we discussed last time is capable of scheduling and interleaving execution of up to 64 warps.
  - So a "SMM" core is capable of concurrently executing multiple CUDA thread blocks.

A more advanced review

(If you understand the following examples you really understand how CUDA programs run on a GPU, and also have a good handle on the work scheduling issues we've discussed in class to this point.)

Consider a program that creates a histogram:

- This example: build a histogram of values in an array
  - All CUDA threads atomically update shared variables in global memory
- Notice I have never claimed CUDA thread blocks were guaranteed to be independent. I only stated CUDA reserves the right to schedule them in any order.
- This is valid code! This use of atomics does not impact implementation's ability to schedule blocks in any order (atomics used for mutual exclusion, and nothing more)



int* A = {0, 3, 4, 1, 9 , 2, . . . , 8, 4 , 1 }; // array of integers between 0-9

But is this reasonable CUDA code?

- Consider implementation on a single core GPU with resources for one CUDA thread block per core
  - What happens if the CUDA implementation runs block 0 first?
  - What happens if the CUDA implementation runs block 1 first?

```
// do stuff here

atomicAdd(&myFlag, 1);
```

Thread block 0

. . .

```
while(atomicAdd(&myFlag, 0) == 0)
   { }

// do stuff here
```

Thread block 1

Global memory

int myFlag

(assume myFlag is initialized to 0)

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