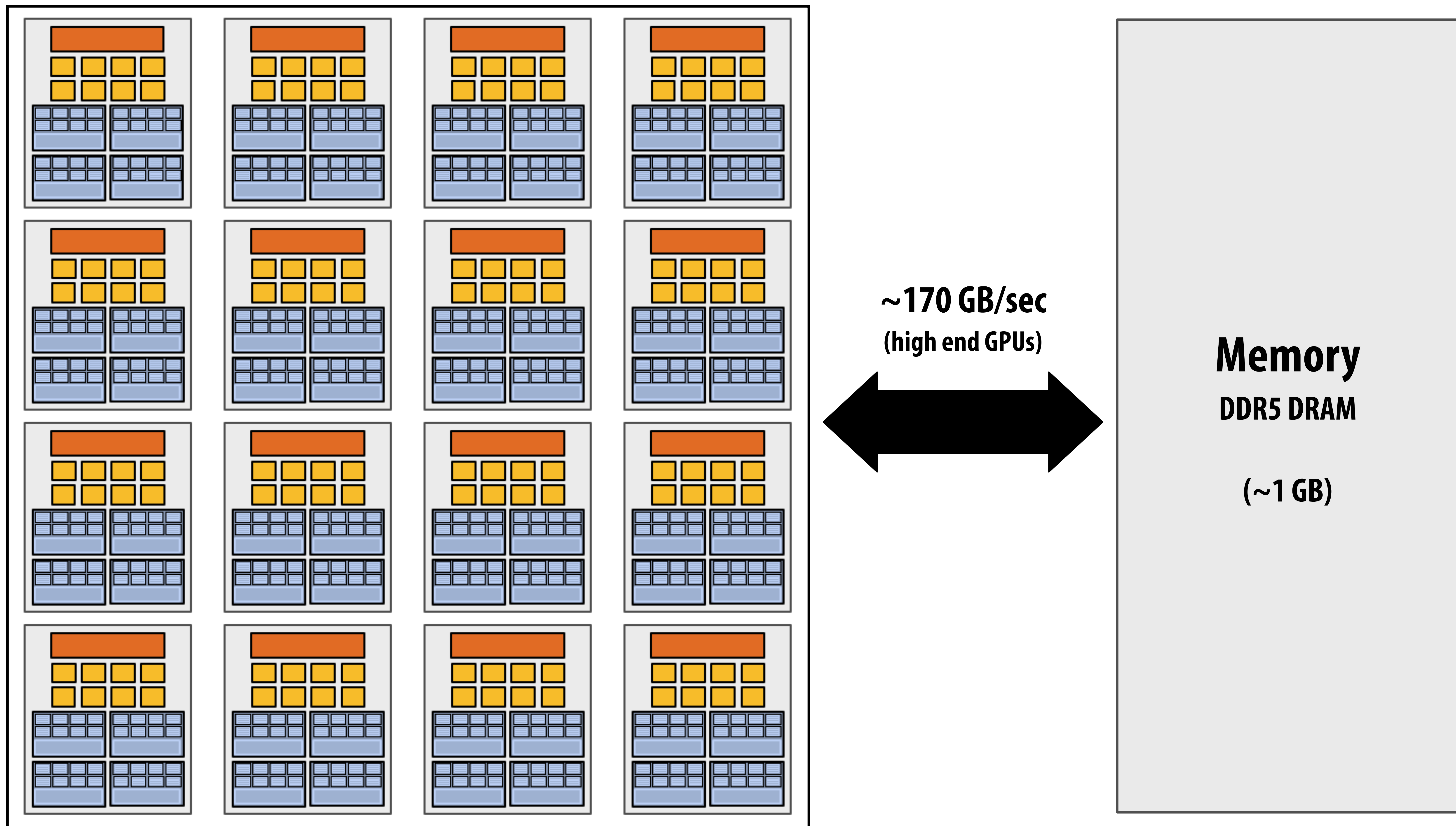
Lecture 6:
GPU Architecture and CUDA Programming

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Today

- History: how graphics chips designed to accelerate Quake evolved into compute engines for a broad class of applications
- GPU programming in CUDA
- A more detailed look at GPU architecture

Recall basic GPU architecture



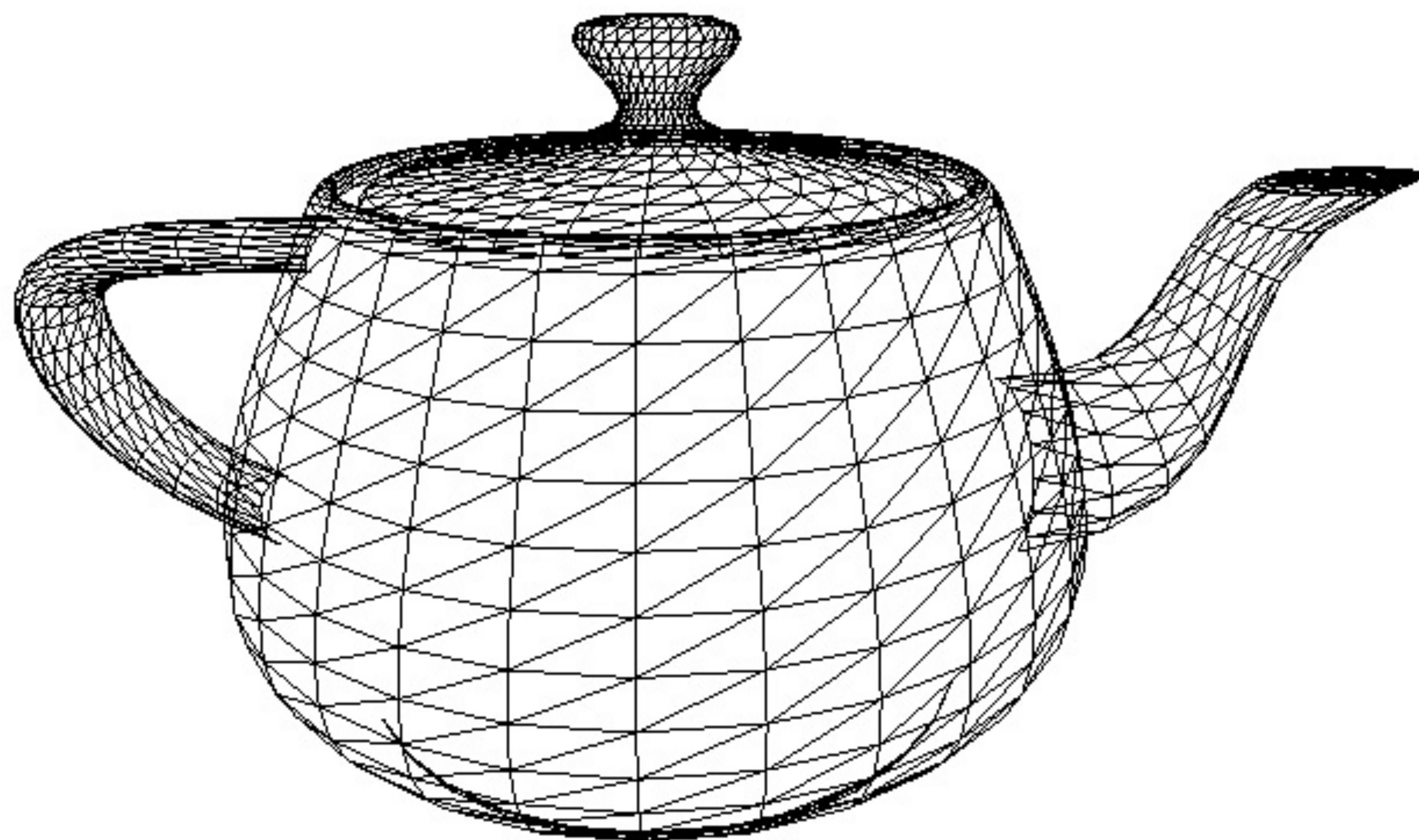
Multi-core chip

SIMD execution within a single core (many ALUs performing the same instruction)

Multi-threaded execution on a single core (multiple threads executed concurrently by a core)

Graphics 101 + GPU history

3D rendering



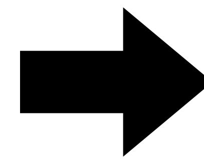


Image credit: Henrik Wann Jensen

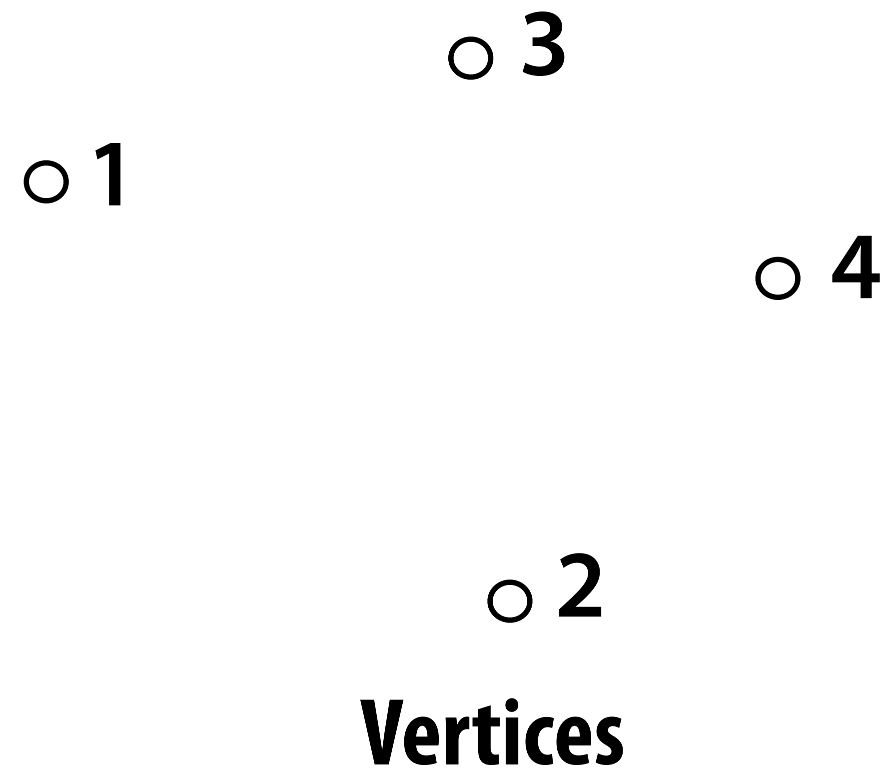
Model of a scene:

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

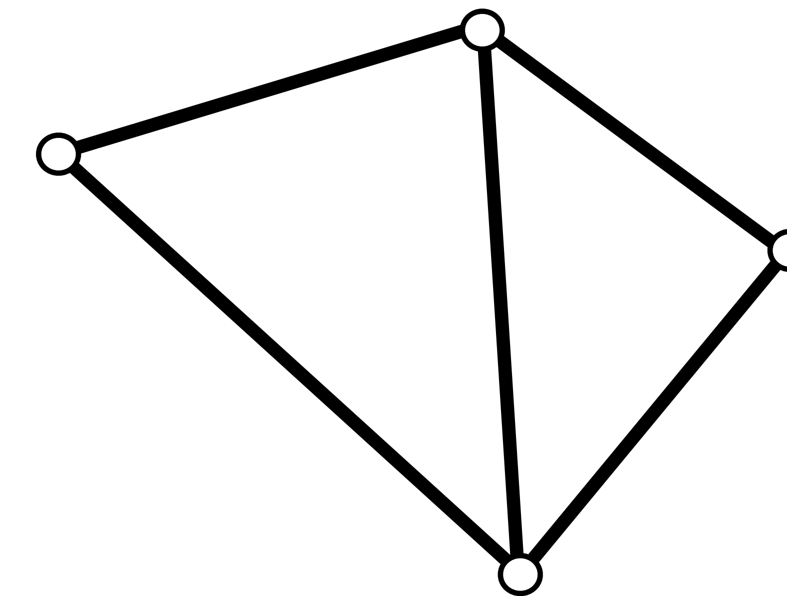
Image

How does each triangle contribute to each pixel in the image?

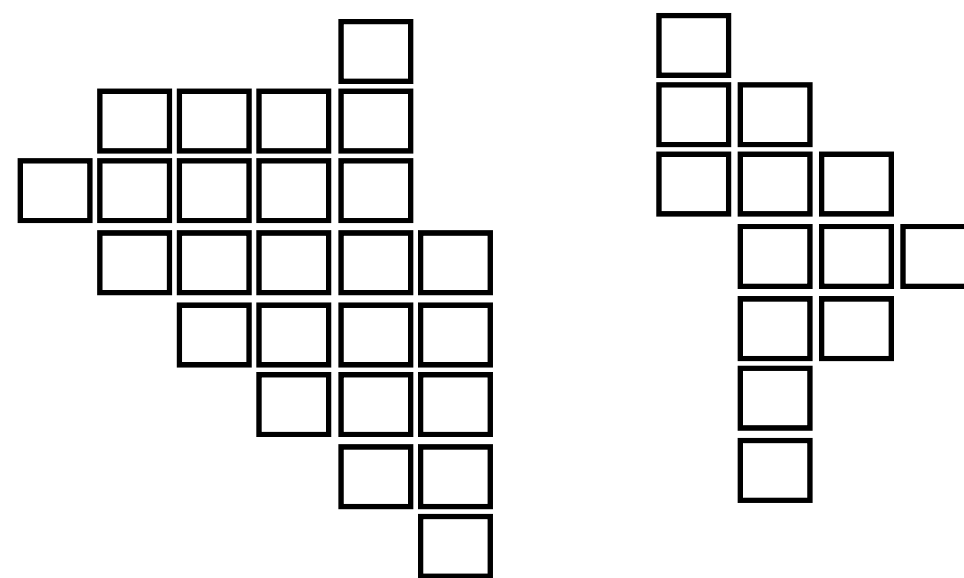
Real-time graphics primitives (entities)



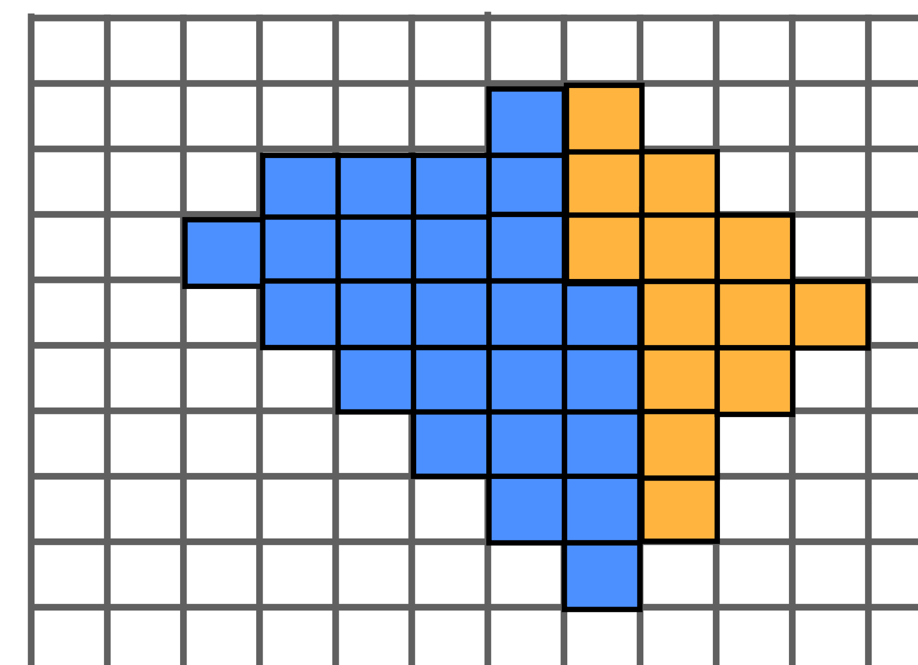
Vertices



Primitives
(triangles, points, lines)

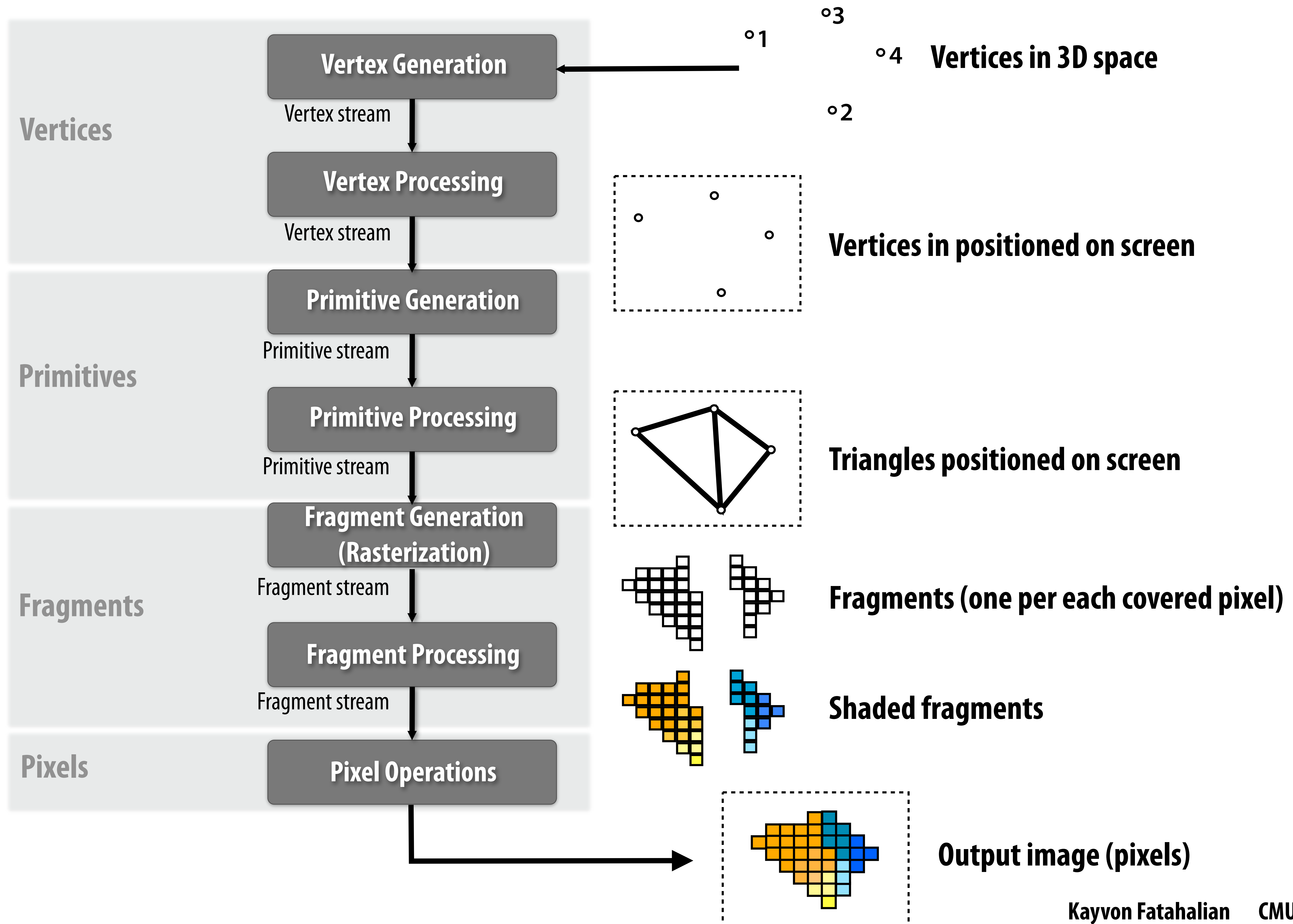


Fragments



Pixels

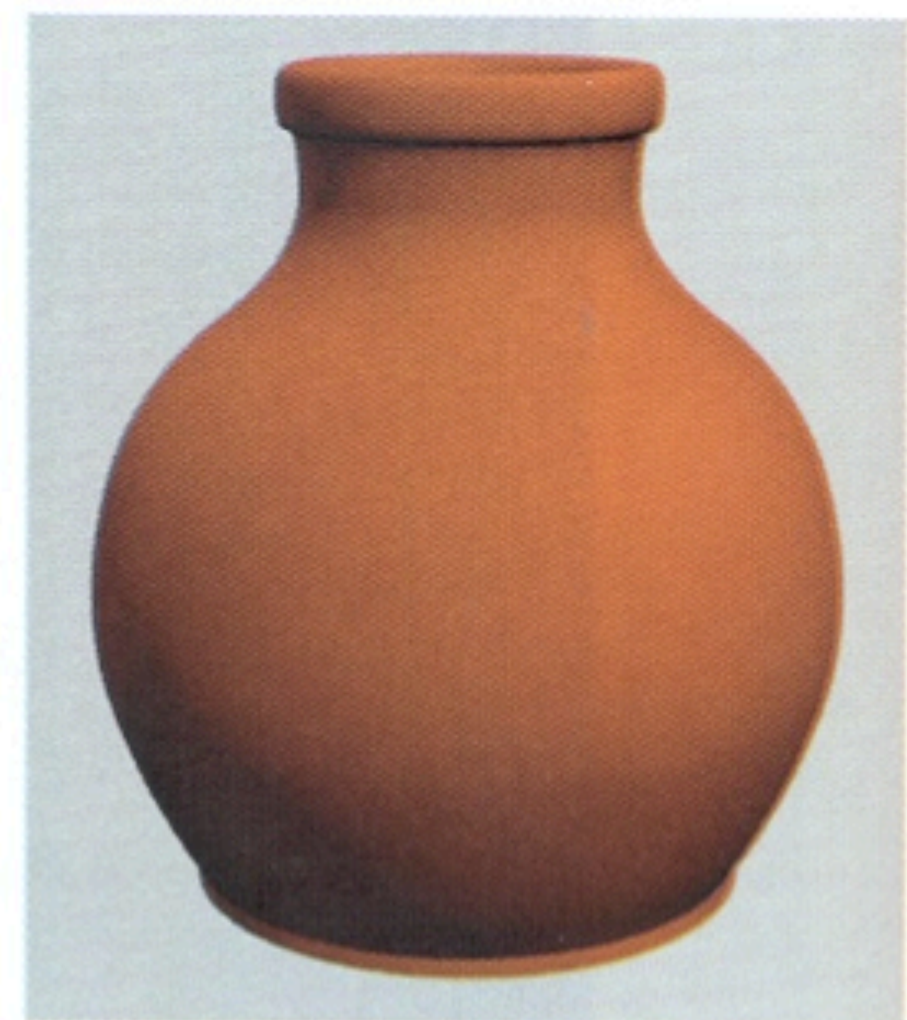
Real-time graphics pipeline (operations)



Example materials

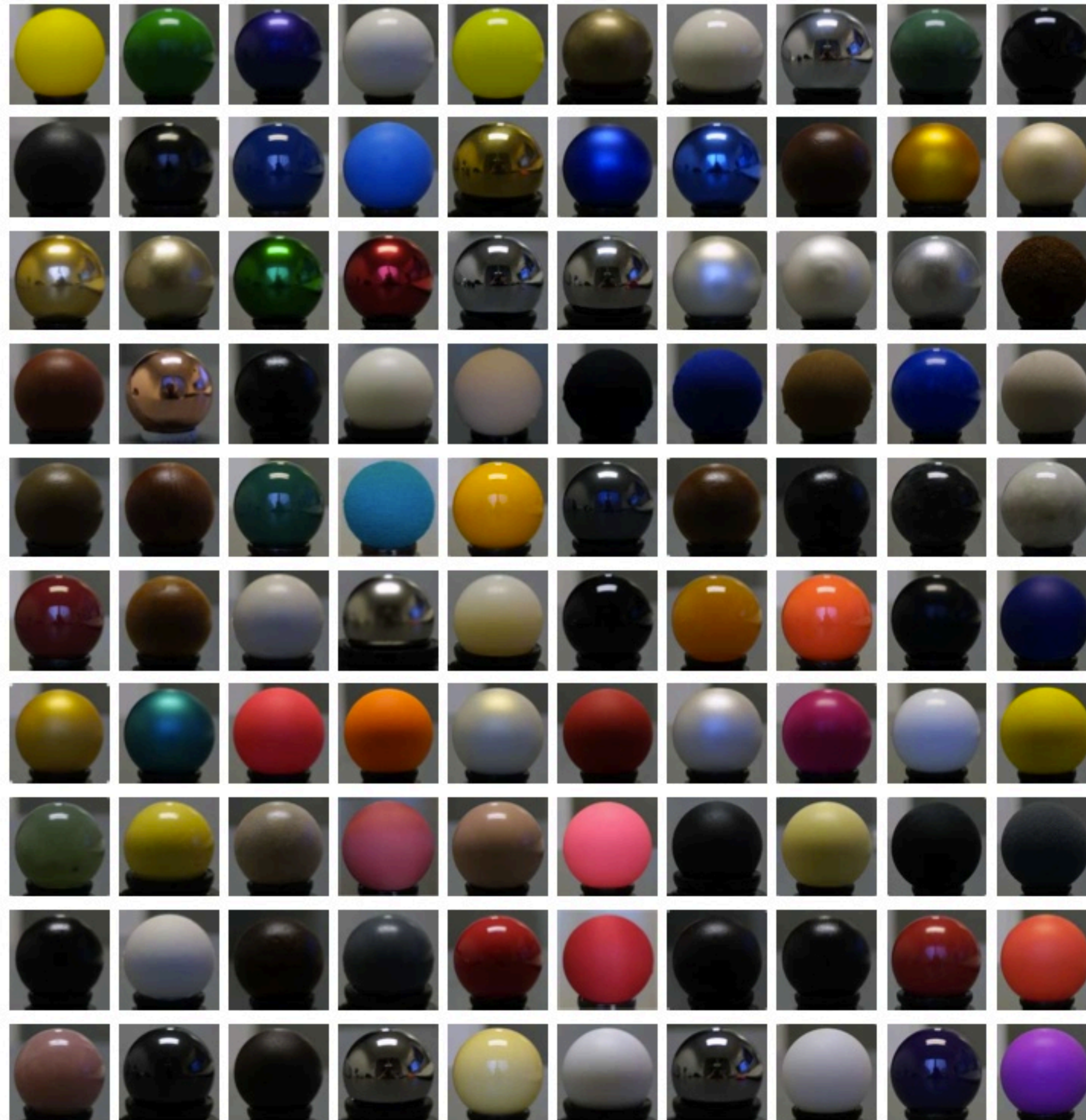
Plastic

Metal


Matte

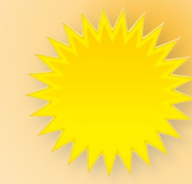
Slide credit Pat Hanrahan
Images from Advanced Renderman [Apodaca and Gritz]

More materials



Slide credit Pat Hanrahan
Images from Matusik et al. SIGGRAPH 2003

Example lights



- **Attenuating light** (intensity falls off with distance: $1/R^2$)
- **Spot light** (does not emit equally in all directions)

- **Environment light** (not a point source: defines light from all directions)
  - Defined by image: pixel(i,j) gives incoming light from direction (ϕ, θ)

- **Non-physical light**
  - Example 1: cast negative light (makes surfaces darker)
  - Example 2: only illuminate blue surfaces

Early graphics programming (OpenGL)

- `glLight(light_id, parameter_id, parameter_value)`
  - 10 parameters (e.g., ambient/diffuse/specular color, position, direction, attenuation coefficient)

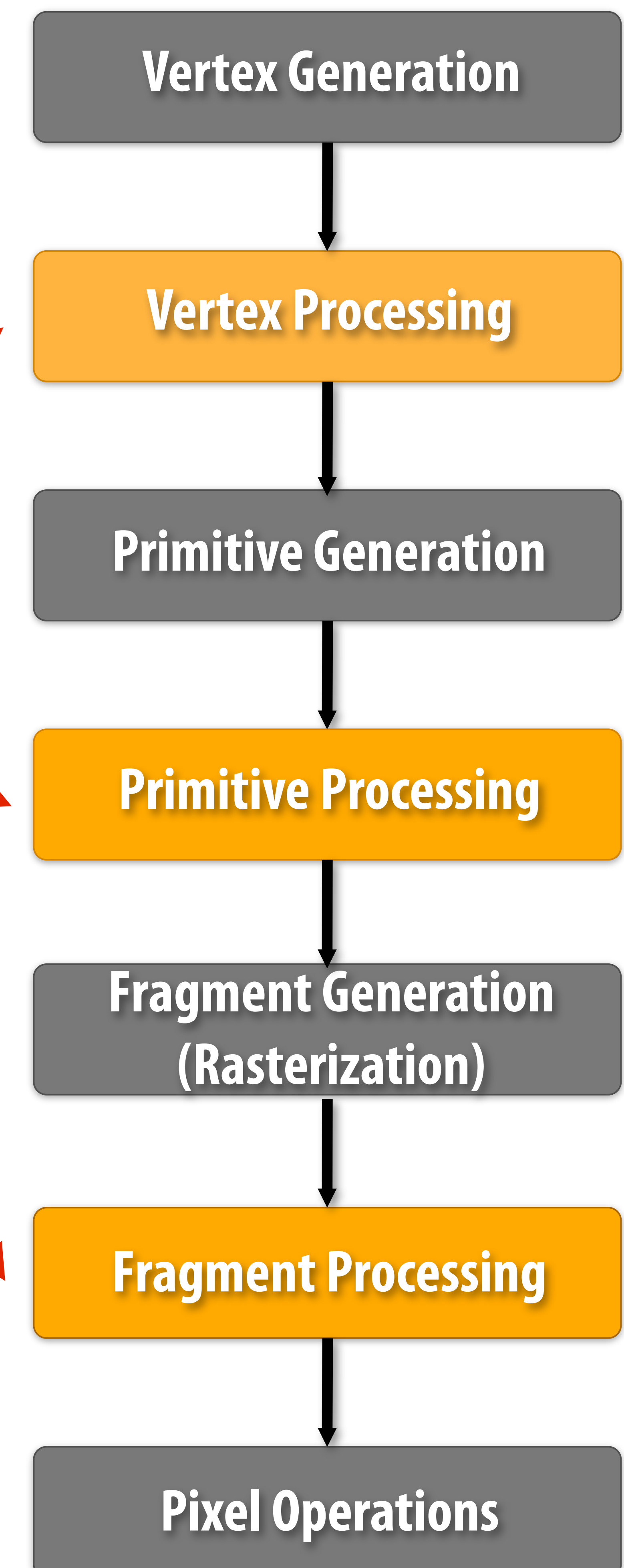
- `glMaterial(face, parameter_id, parameter_value)`
  - Parameter examples (surface color, shininess)

- **Parameterized shading function evaluated at each mesh vertex**
  - Summation over all enabled lights
  - Resulting per-vertex color modulated by result of texturing

Shading languages

- Specify materials and lights programmatically!
  - support large diversity in materials
  - support large diversity in lighting conditions

- Programmer provides mini-programs ("shaders") that defines pipeline logic for certain stages



Example fragment shader

Run once per pixel covered by a triangle

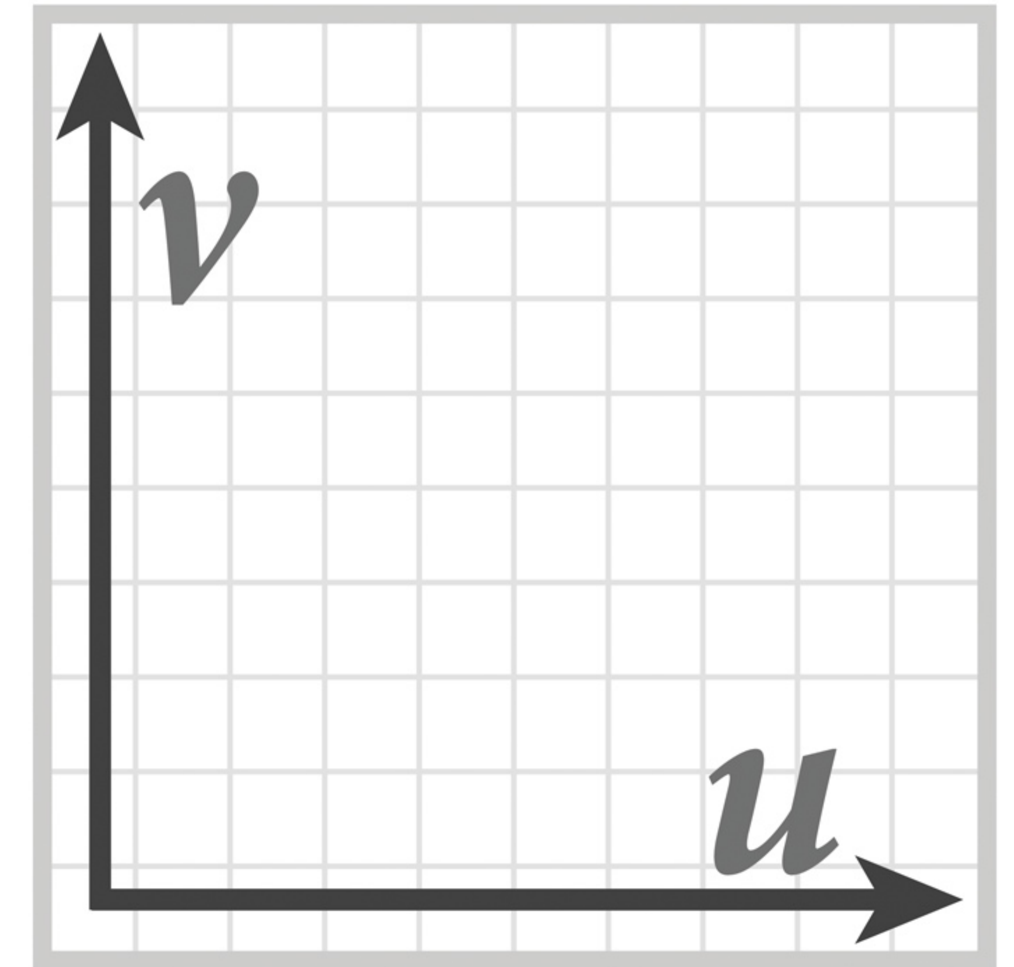
HLSL shader program: defines behavior of fragment processing stage

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

Let:

`lightDir = [-1, -1, 1]`

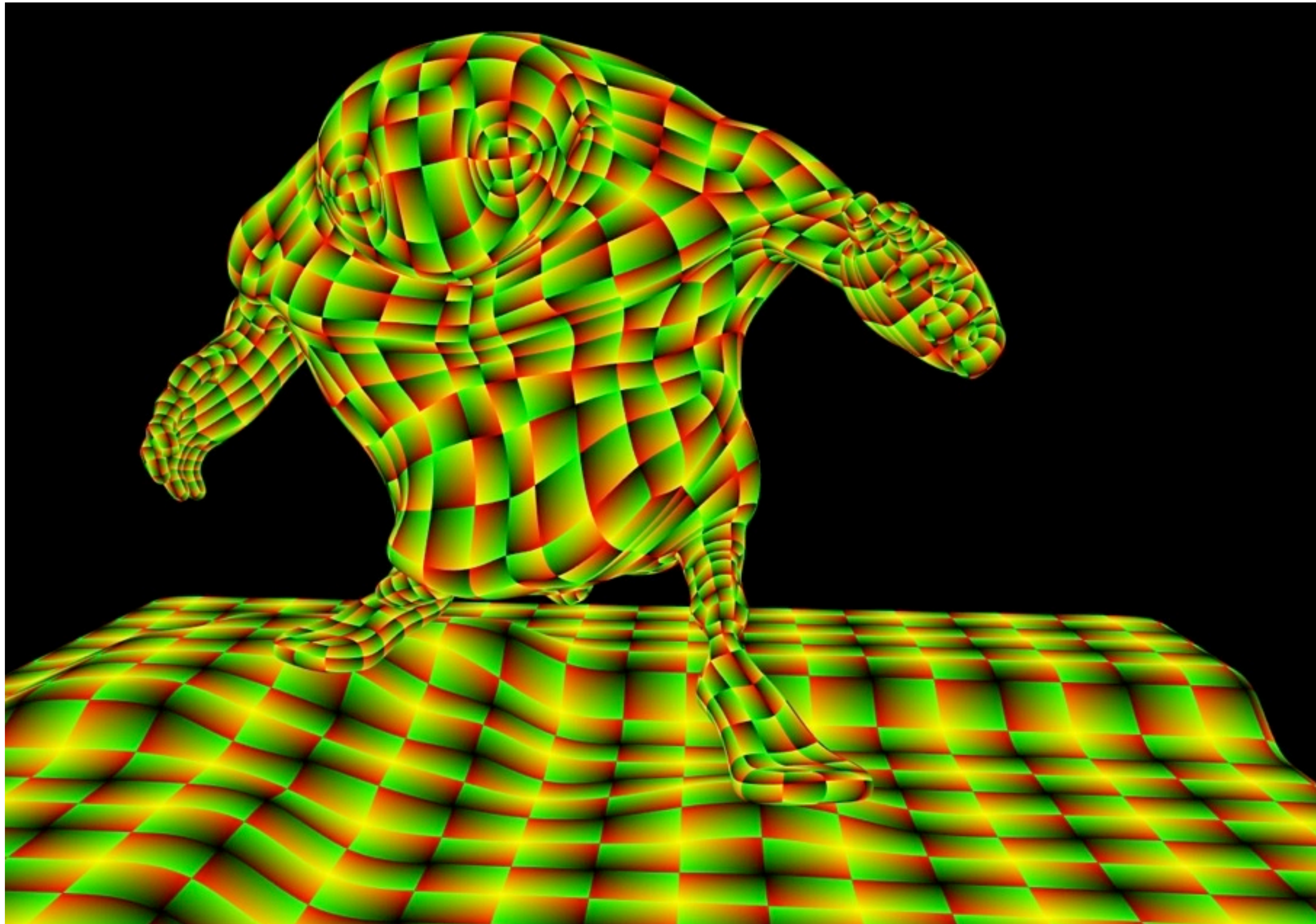
`myTex =`



function defined on $[0,1]^2$ domain:
myTex : $[0,1]^2 \longrightarrow$ color
(represented by an image)

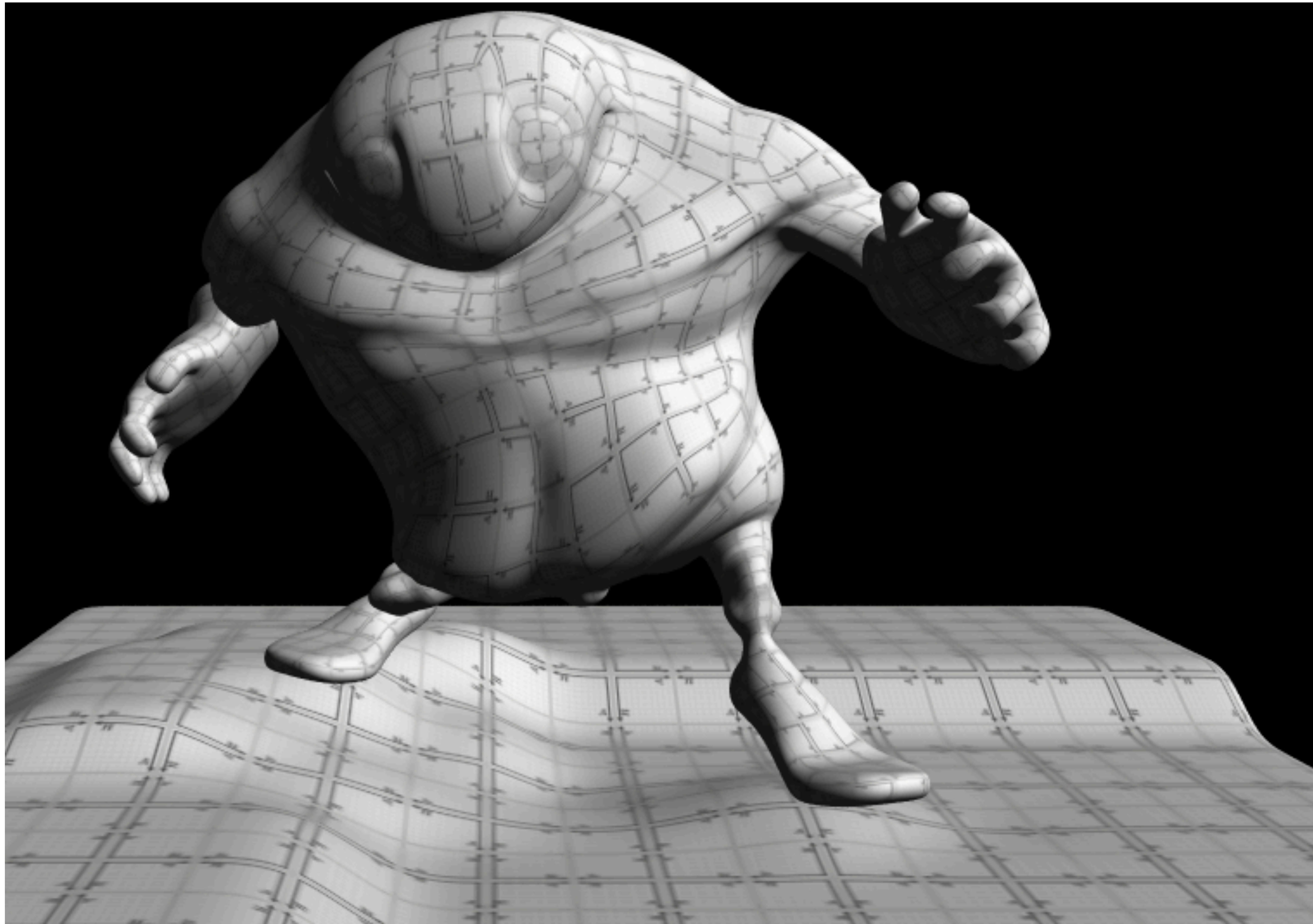
`mySamp` **defines how to sample from function to generate value at (u,v)**

Texture coordinates (UV)



Red channel = u
Green channel = v

Shaded result

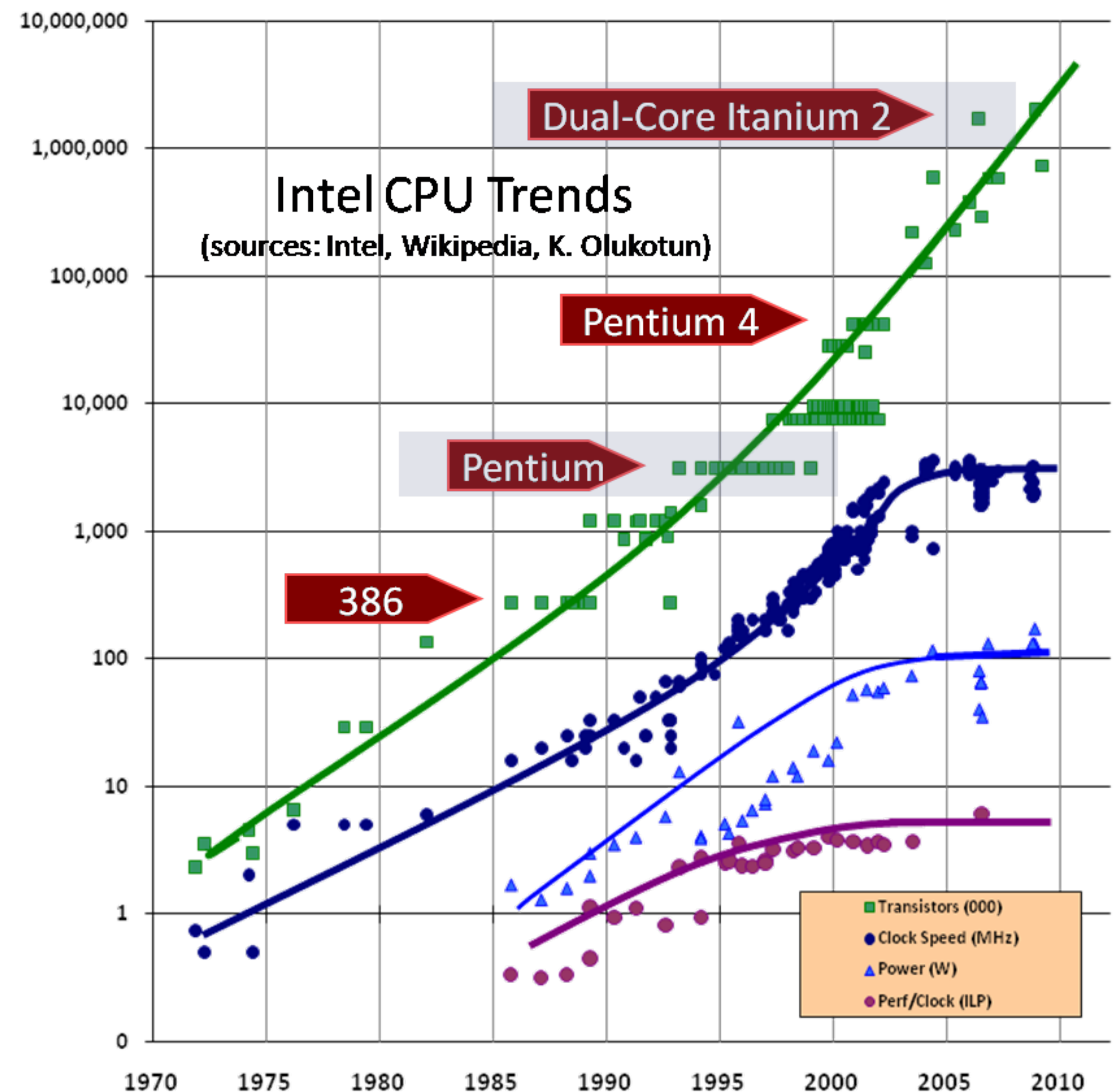


Observation circa 2001-2003

These GPUs are very fast processors for performing the same computation (shaders) on collections of data (streams of vertices, fragments, pixels)

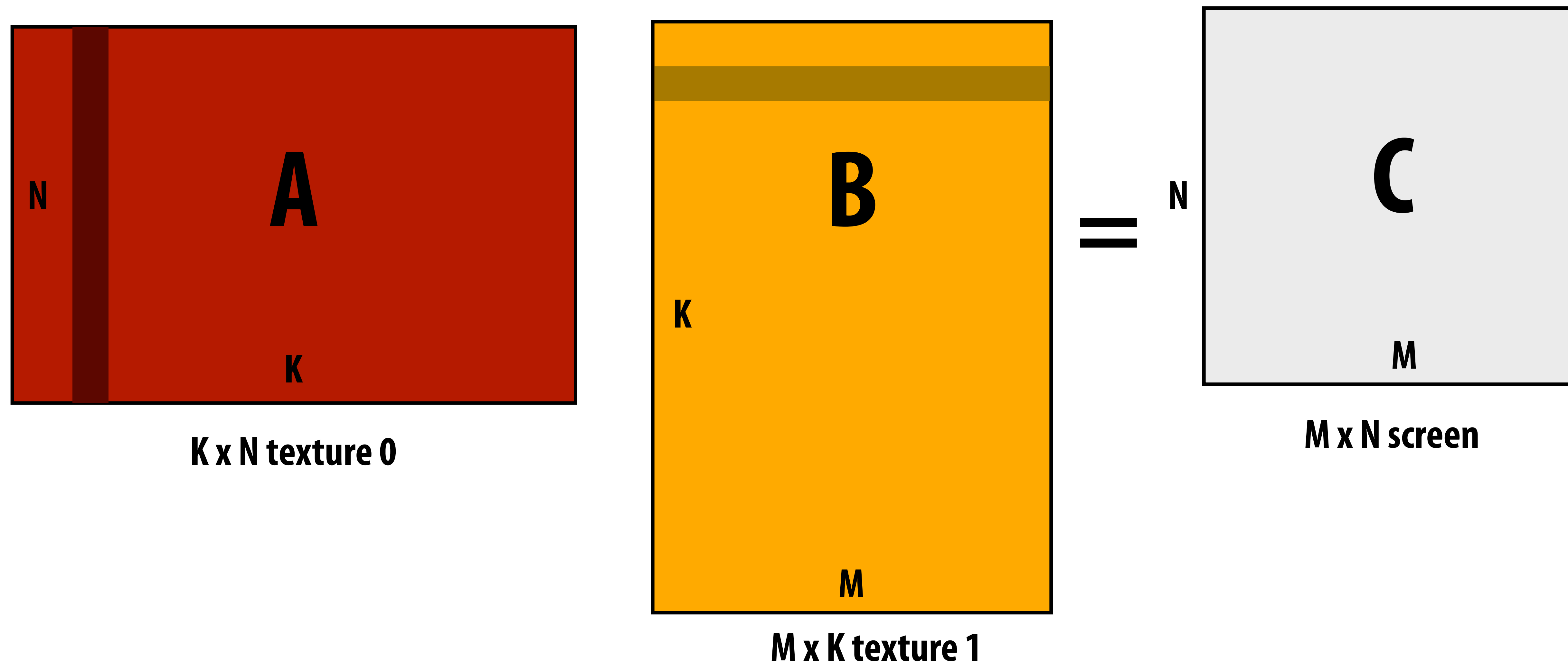
Wait a minute! That sounds a lot like data-parallelism to me!

(I remember data-parallelism from exotic supercomputers in the 90s)



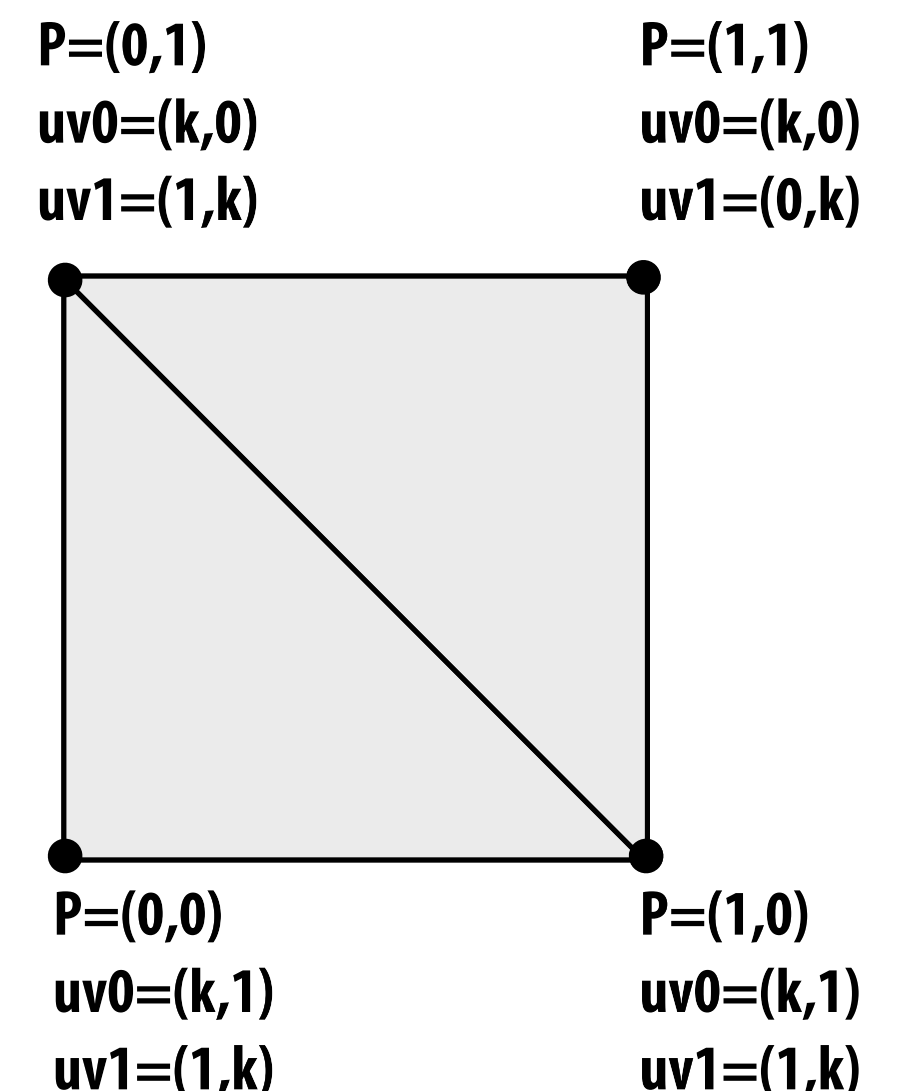
Hack! early GPU-based scientific computation

Dense matrix-matrix multiplication by drawing triangles



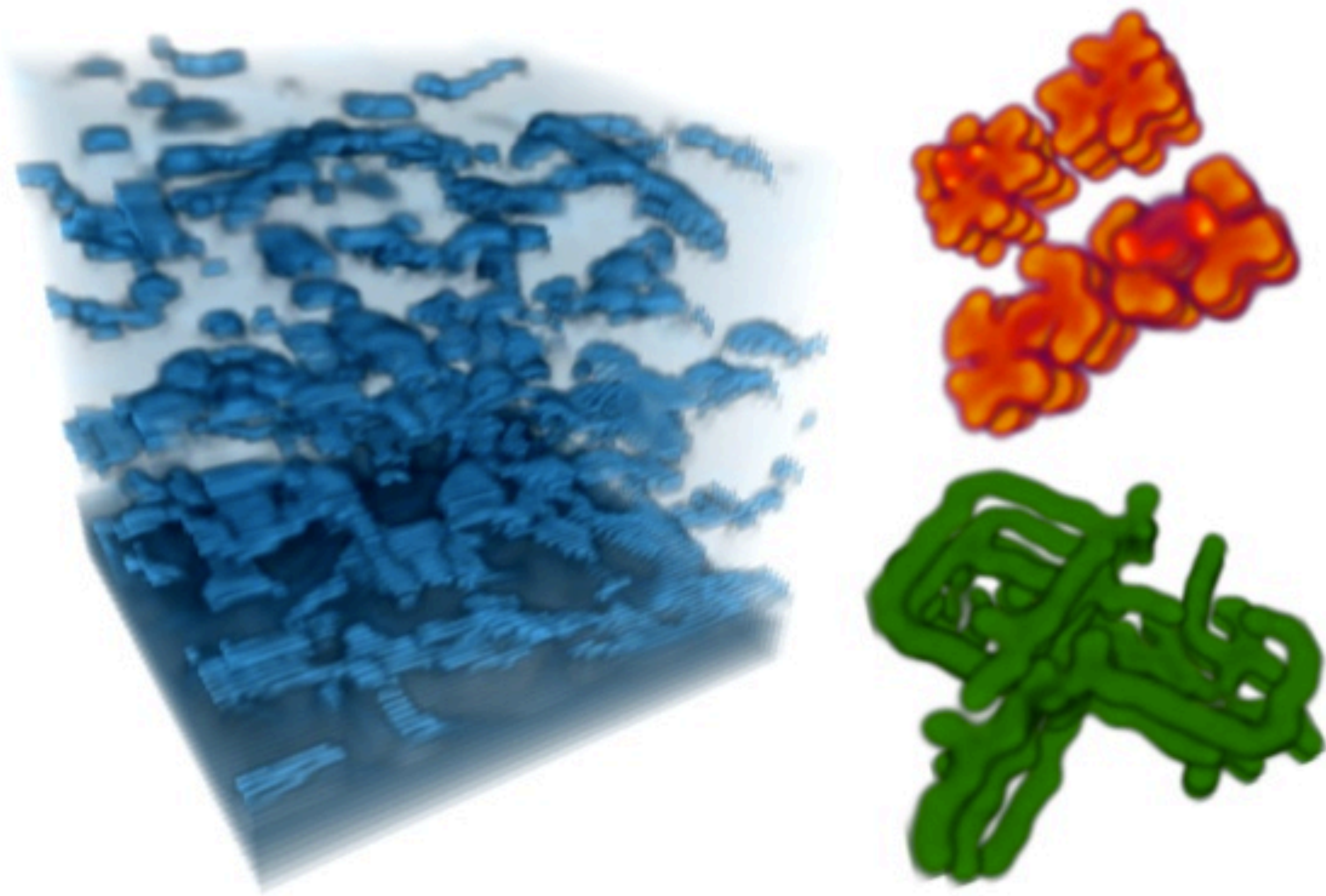
[Larson and McAllister, SC 2001]

Set screen size to be output matrix size
Set graphics blend mode to ADD
for k=0 to K
    Set texture coords as shown at right
    Render 2 triangles that exactly cover screen
    (one shader computation per pixel)

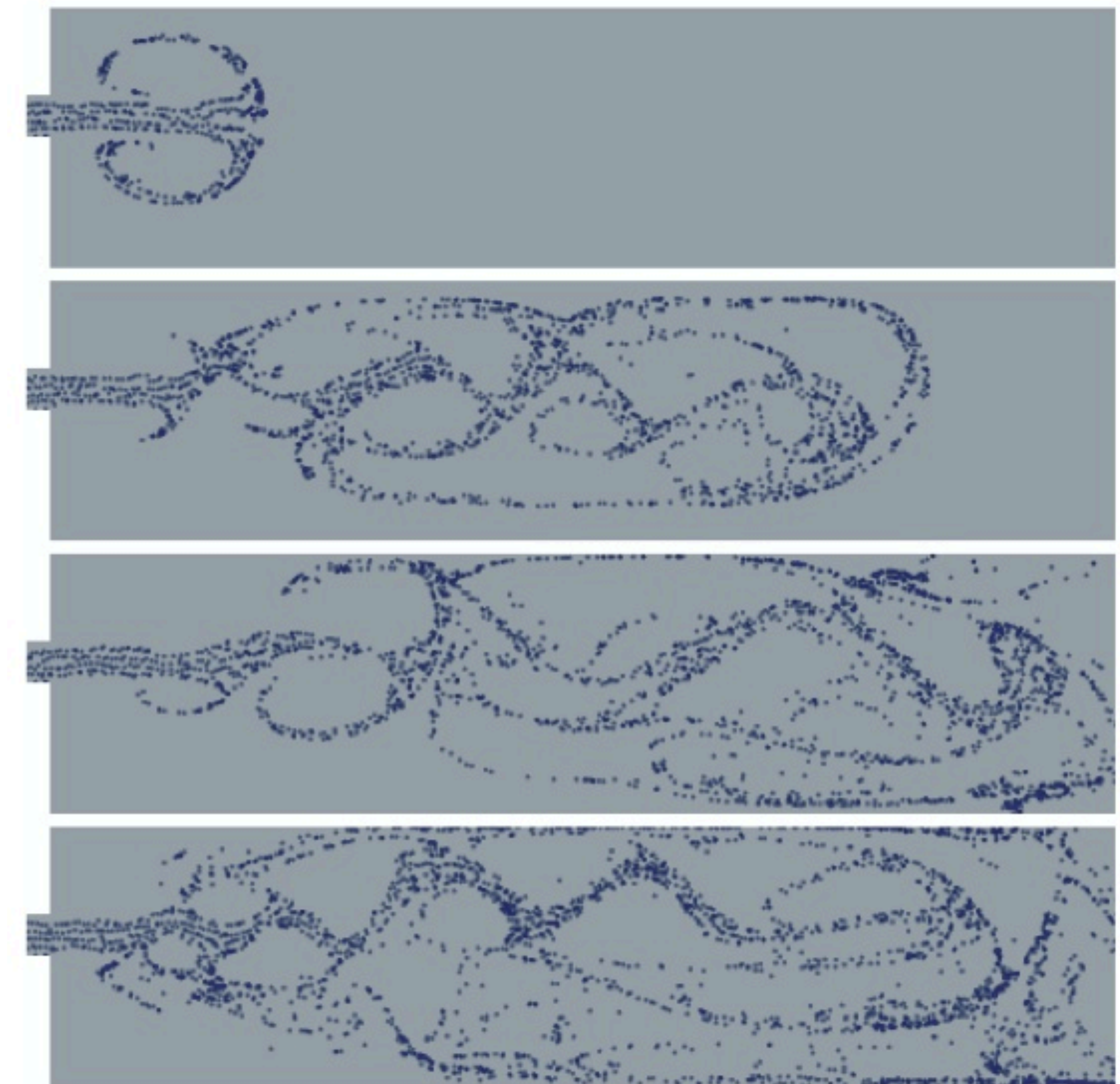


“GPGPU” 2002-2003

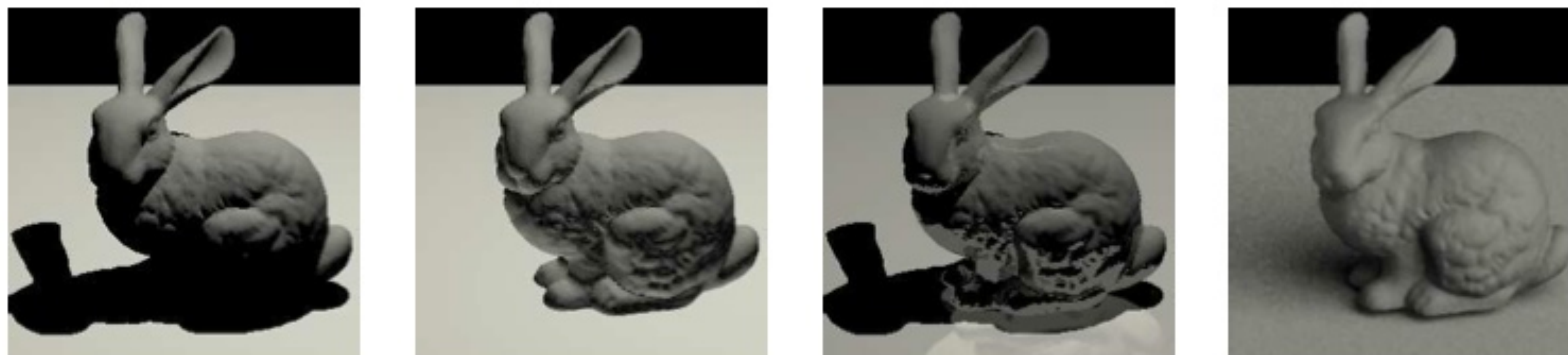
GPGPU = “general purpose” computation on GPUs



Coupled Map Lattice Simulation [Harris 02]



Sparse Matrix Solvers [Bolz 03]



Ray Tracing on Programmable Graphics Hardware [Purcell 02]

Brook language (2004)

- **Research project**
- **Abstract GPU as data-parallel processor**

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

// note: omitting initialization
float scale_amount;
float input_stream<1000>;
float output_stream<1000>;

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

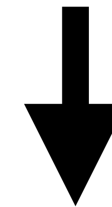
- **Brook compiler turned program into GPU commands such as createTexture, drawTriangles**

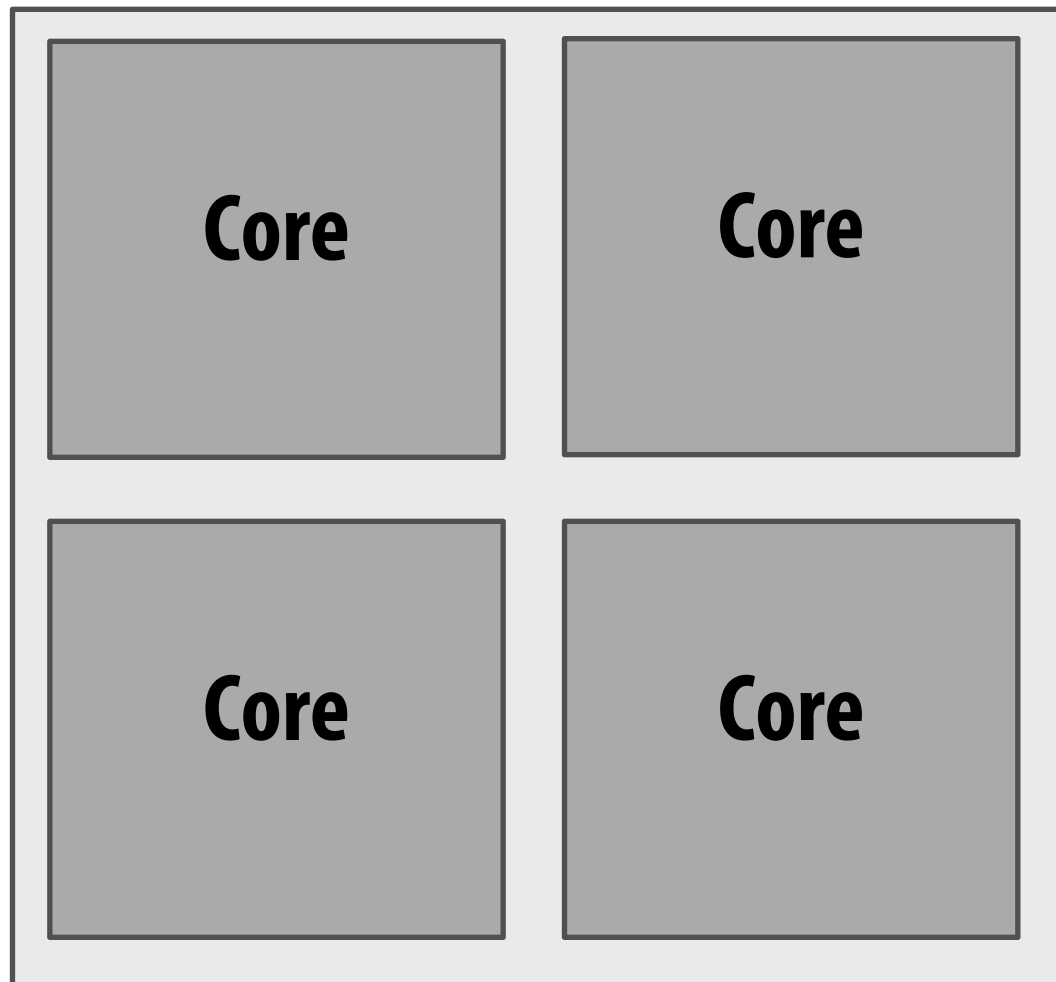
GPU Compute Mode

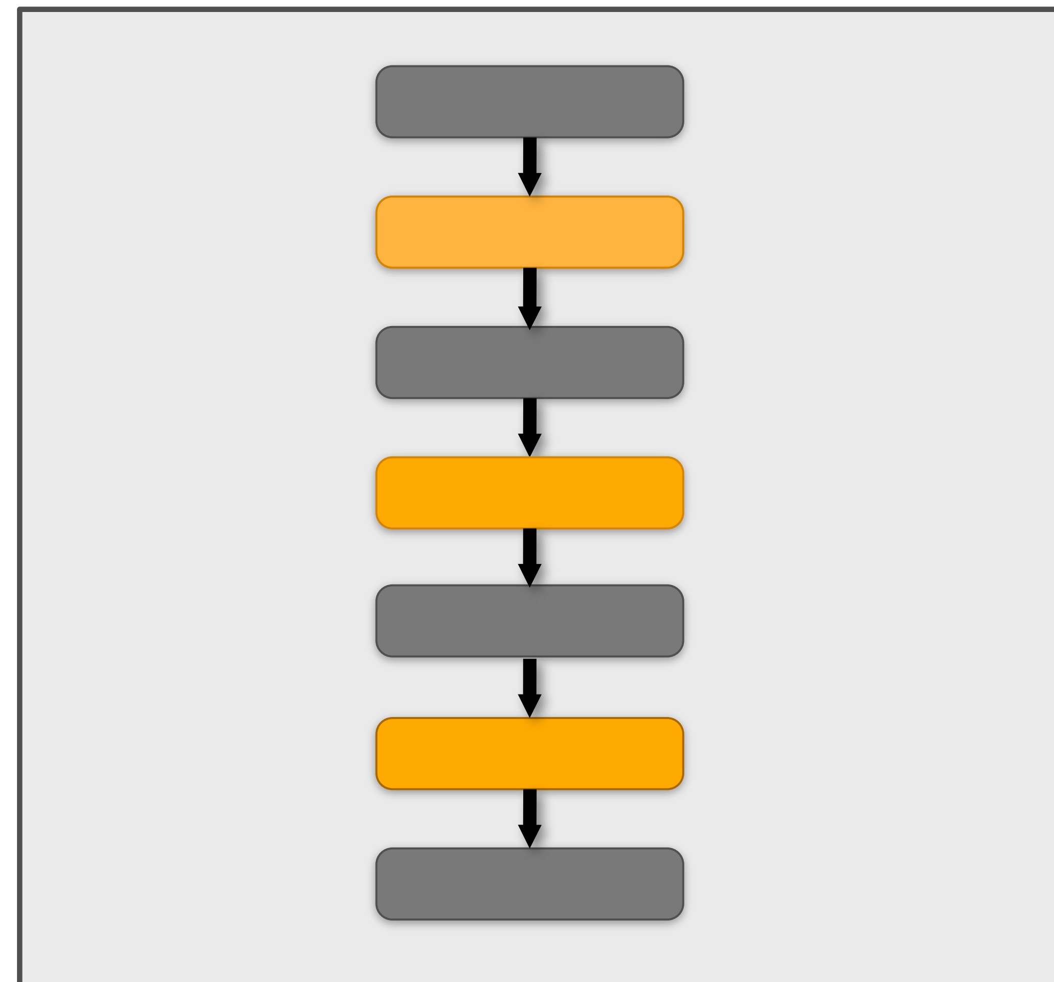
NVIDIA Tesla architecture (2007)

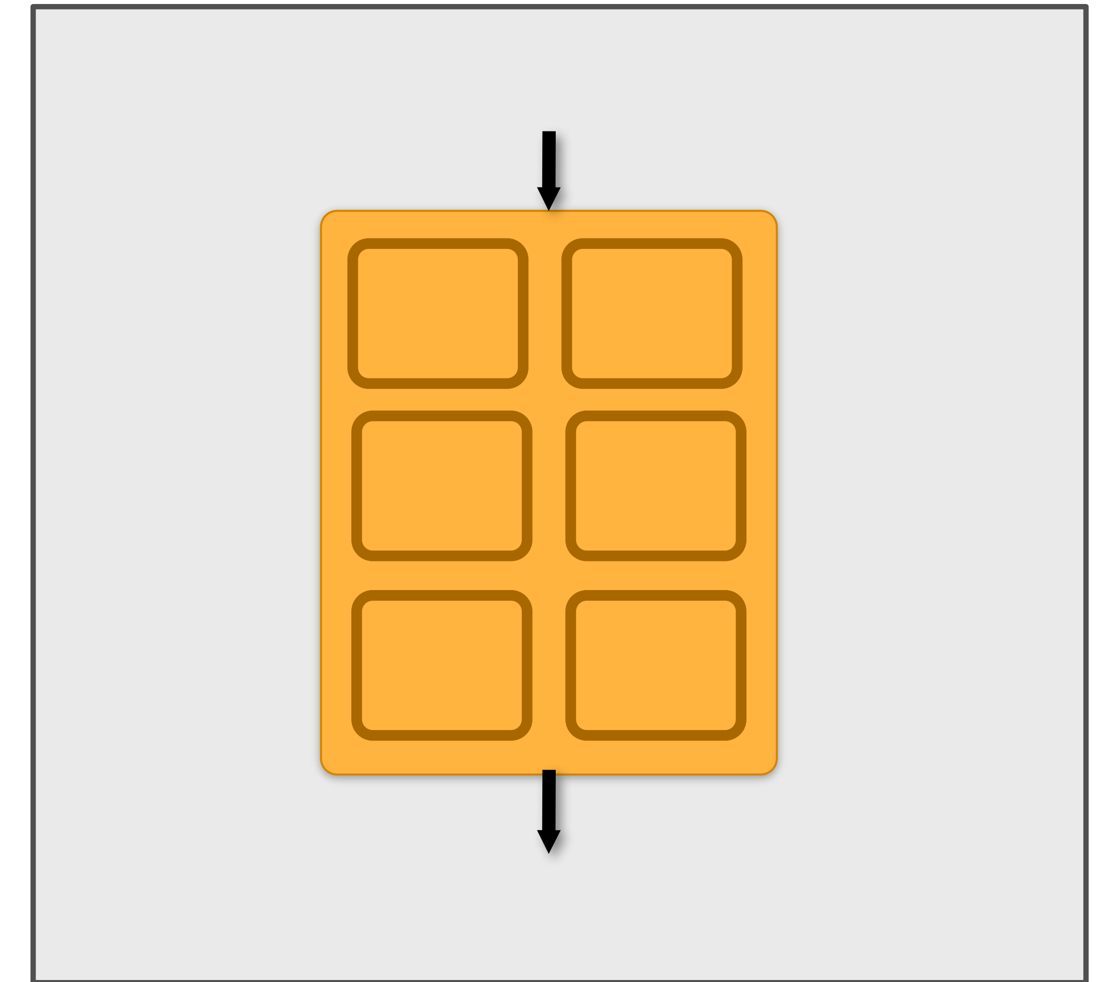
(GeForce 8xxx series)

Provided alternative, non-graphics-specific ("compute mode") software interface to GPU









Multi-core CPU architecture

CPU presents itself to system software (OS) as multi-processor system

ISA provides instructions for managing context (program counter, VM mappings, etc.) on a per core basis

Pre-2007 GPU architecture

GPU presents following interface** to system software (driver):

Set shader program start PC
DrawTriangles

(** interface also included many other commands for configuring graphics pipeline: e.g, set screen output size)

Post-2007 "compute mode" GPU architecture

GPU presents following interface to system software (driver):

Set kernel program start PC
Launch(kernel, N)

CUDA programming language

- Introduced in 2007 with NVIDIA Tesla architecture
- C-like language to express programs that run on GPUs using the compute-mode hardware interface
- Relatively low-level: (low abstraction distance) CUDA's abstractions closely match the capabilities/performance characteristics of modern GPUs
- Note: OpenCL is an open standards version of CUDA
  - CUDA only runs on NVIDIA GPUs
  - OpenCL runs on CPUs/GPUs from many vendors
  - Almost everything I say about CUDA also holds for OpenCL
  - At this time CUDA is better documented, thus I find it preferable to teach with

The plan

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

Things to consider throughout this lecture:

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

Clarification

- I am going to describe CUDA abstractions using CUDA terminology

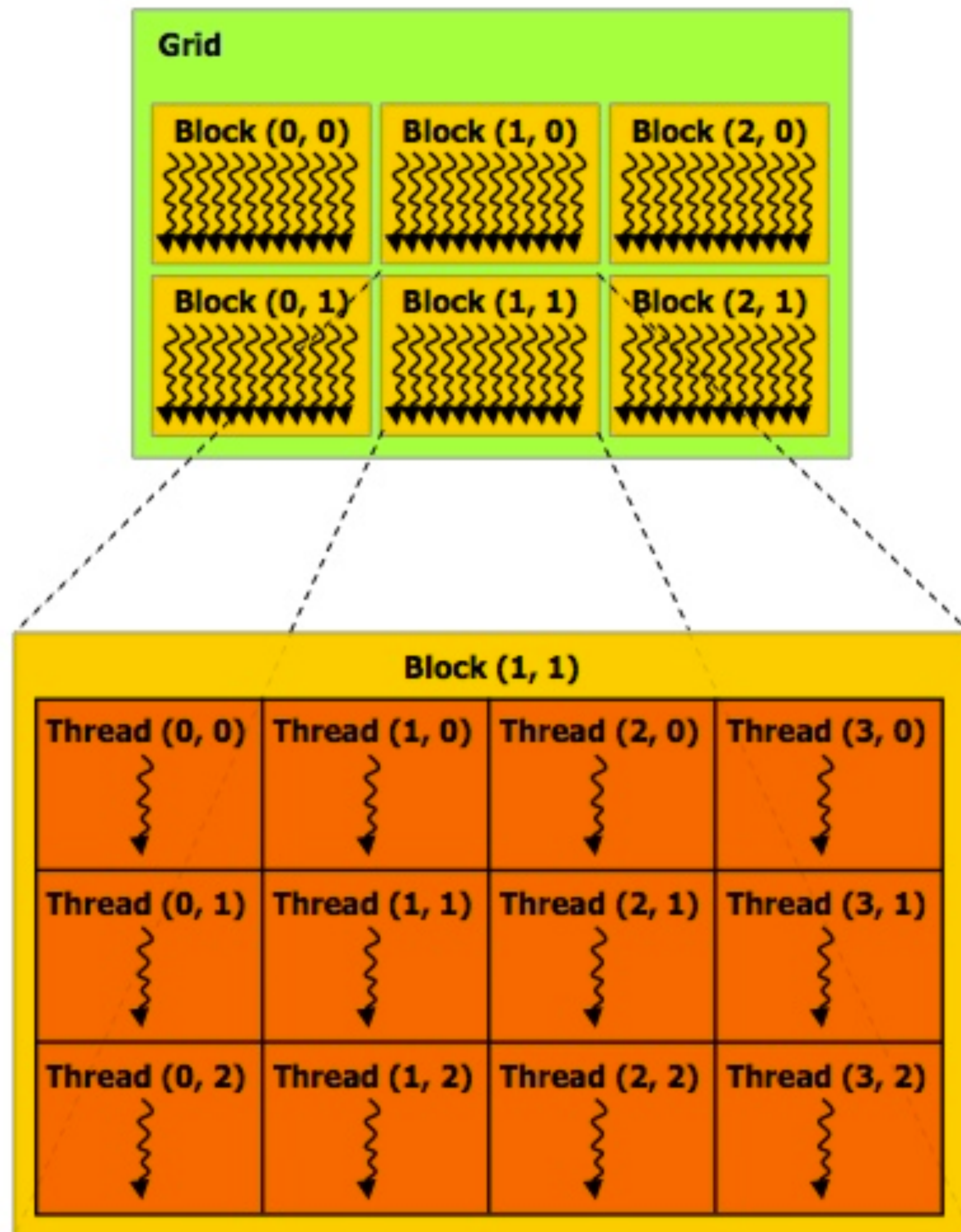
- Specifically, the term thread is going to carry different than how I've used it in class so far.

- 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 nD




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

// 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[i][j] = A[i][j] + B[i][j];
}

//////////////////////////////////////////////

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

CUDA programs consist of a hierarchy of concurrent threads

SPMD execution of device code:

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

"Device" code: SPMD execution
kernel function (denoted by __global__) runs on co-processing device (GPU)

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

Bulk launch of many threads
Precisely: launch a grid of thread blocks
Call returns when all threads have terminated

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

// 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[i][j] = A[i][j] + B[i][j];
}

//////////////////////////////////////////////

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

Clear separation of host and device code

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

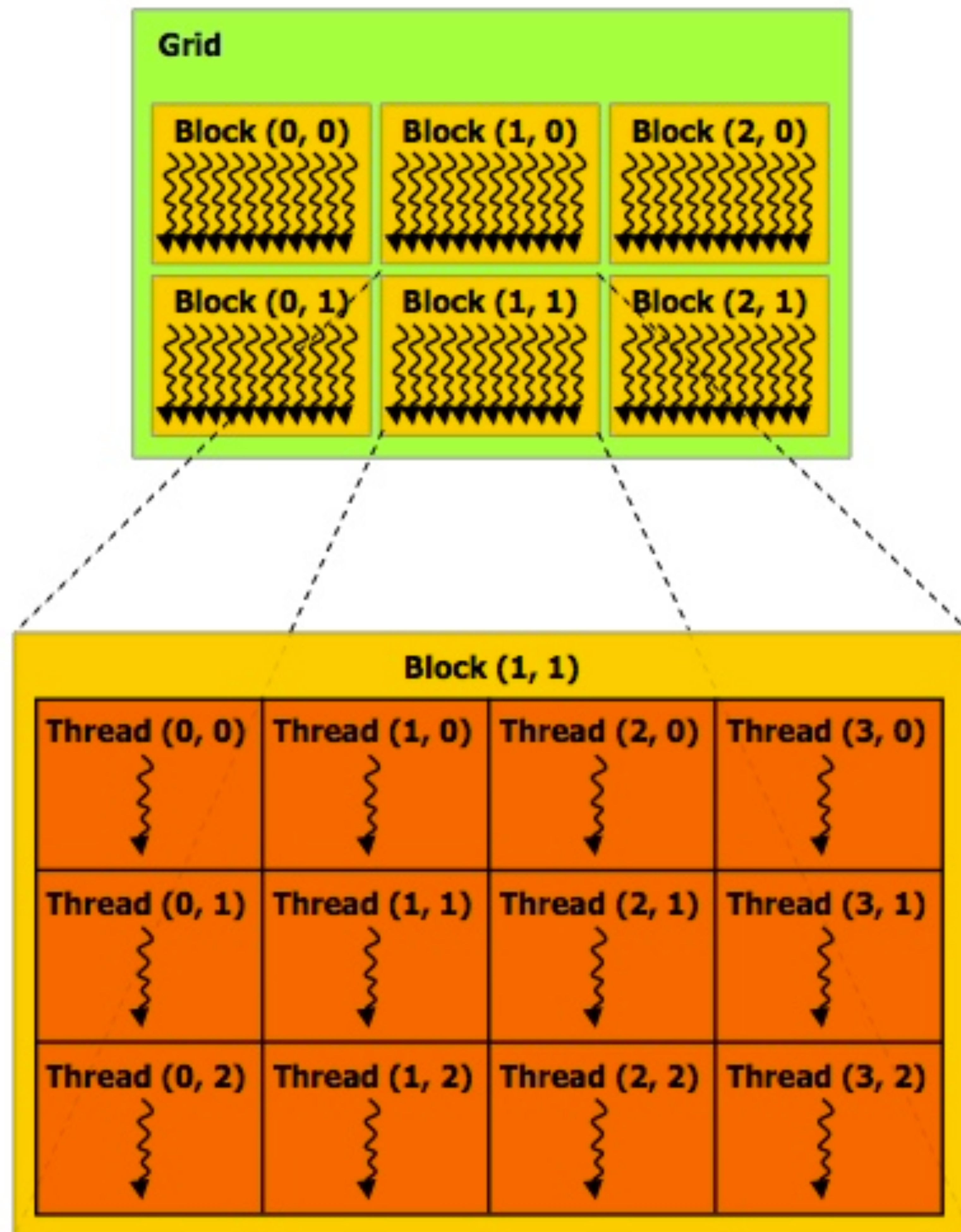
"Device" code
SPMD execution on GPU

"Host" code : serial execution on CPU

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

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

//////////////////////////////////////////////

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);
```

Number of SPMD threads is explicit in program

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




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

__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[i][j] = A[i][j] + B[i][j];
}

//////////////////////////////////////////////

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

Execution model

Host
(serial execution)

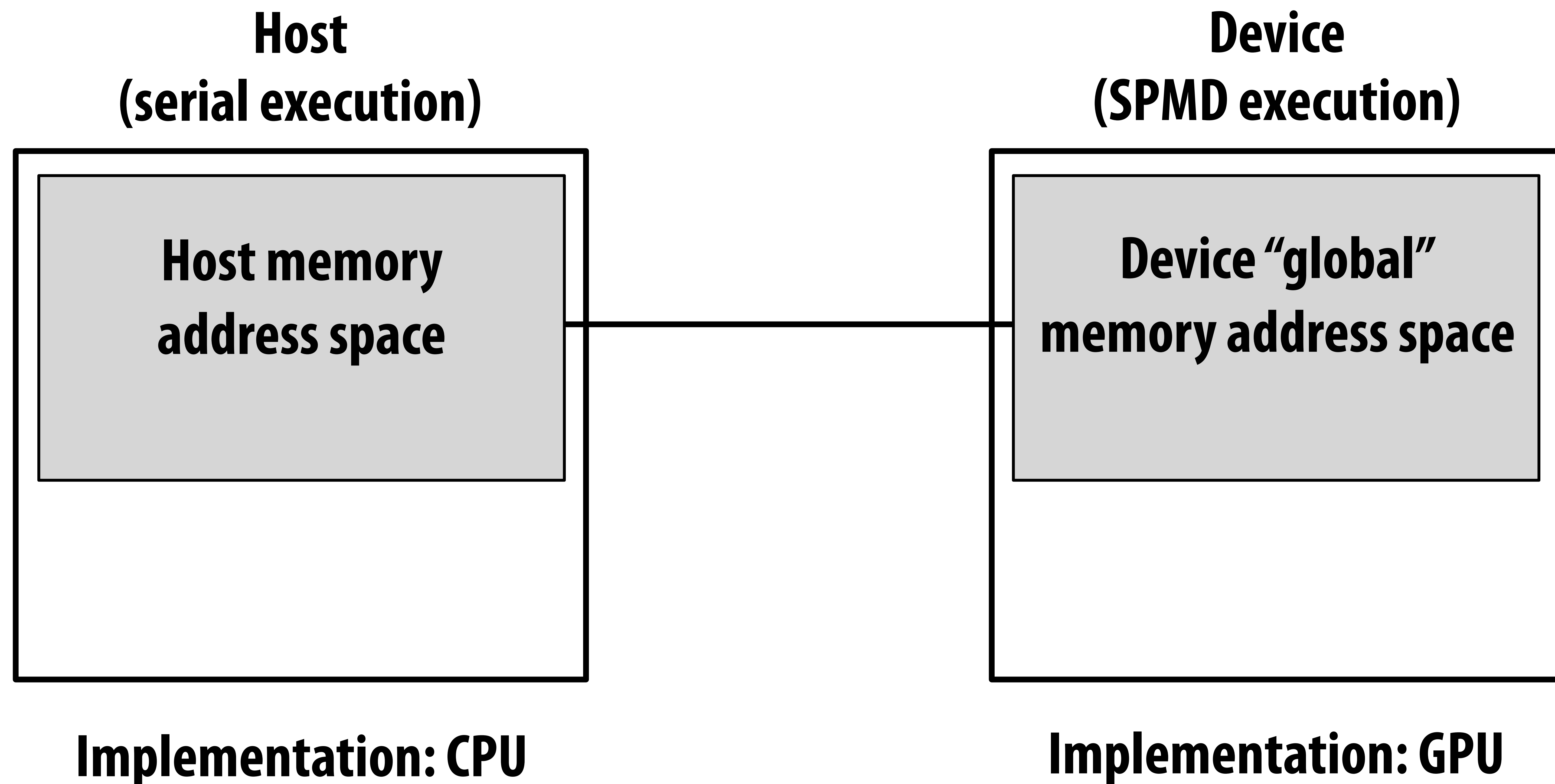
Implementation: CPU

Device
(SPMD execution)

Implementation: GPU

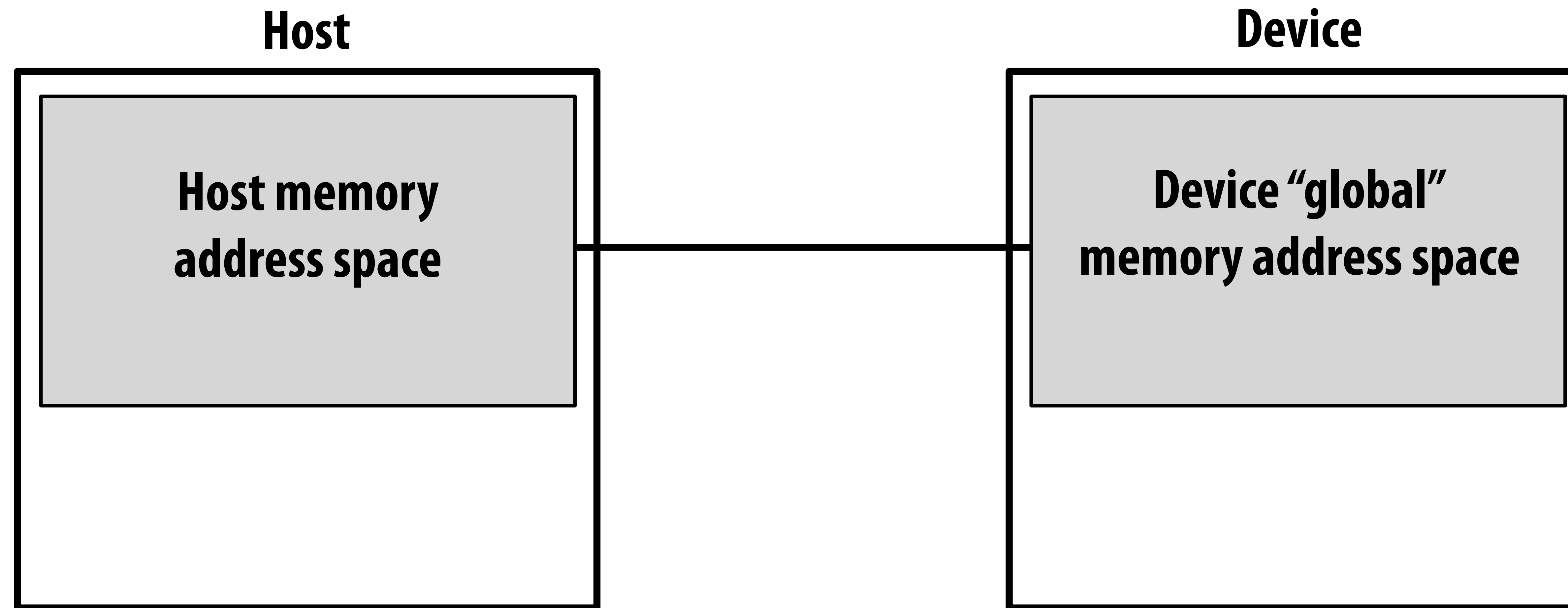
Memory model

Distinct host and device address spaces



memcpy primitive

Move data between address spaces



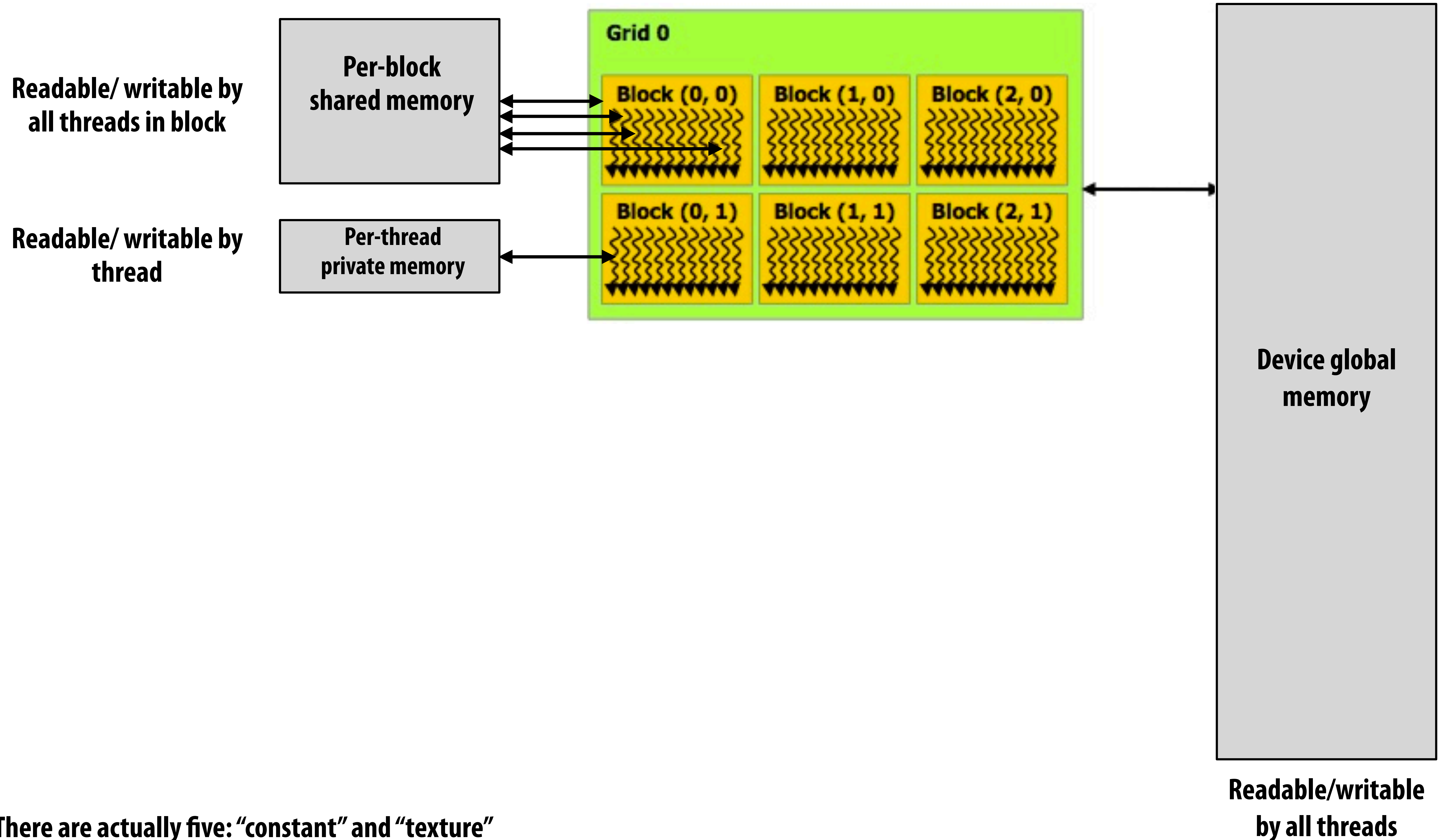

```
float* A = new float[N];        // allocate buffer in host mem

// 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?

(async variants also exist)

CUDA device memory model

Three** distinct types of memory visible to device-side code



** There are actually five: “constant” and “texture” address spaces not discussed here (carried over from graphics shading languages)

CUDA synchronization constructs

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

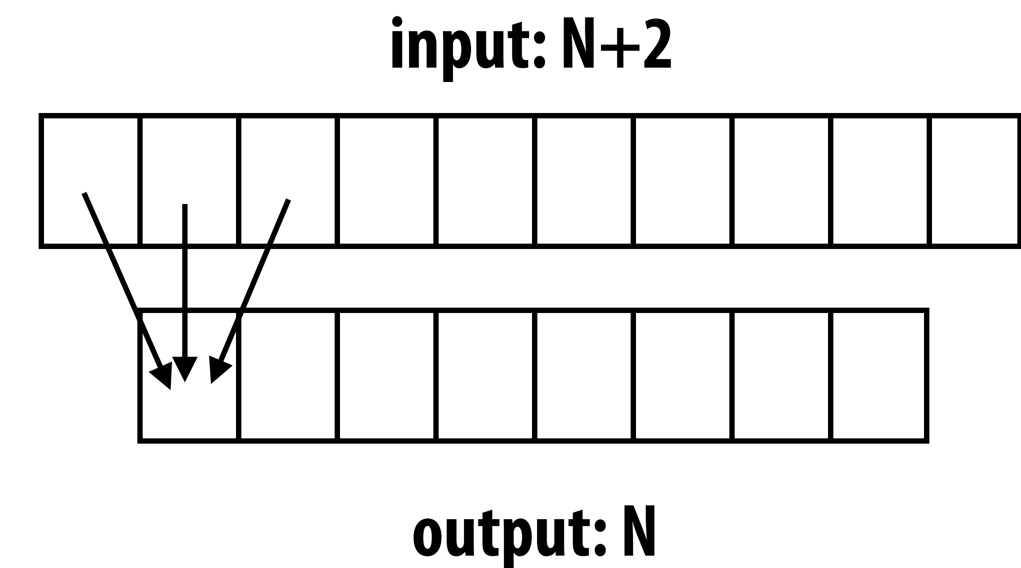
- Atomic operations
  - e.g., float atomicAdd(float* addr, float amount)
  - Atomic operations on both global memory and shared memory variables

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

1D convolution in CUDA

One thread per output element

```
#define THREADS_PER_BLK 128

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

  __shared__ float support[THREADS_PER_BLK+2];  // shared across block
   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;
}

// host code //////////////////////////////////////////////////////////////
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/THREAD_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```



cooperatively load block's support region from global memory into shared memory

barrier (only threads in block)

each thread computes result for one element

write result to global memory

CUDA abstractions

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

- **Distributed address space**
  - Built-in memcpy primitives to copy between host and device address spaces
  - Three types of variables in device space
    - Per thread, per block ("shared"), or per program ("global")
      (can think of types as residing within different address spaces)

- **Barrier synchronization primitive for threads in thread block**

- **Atomic primitives for additional synchronization (shared and global variables)**

CUDA semantics

```
#define THREADS_PER_BLK 128

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

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

   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;
}

// host code //////////////////////////////////////////////////////
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/THREAD_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

Consider pthreads:

Call to pthread_create():
Allocate thread state:
- **Stack space for thread**
- **Allocate control block so OS/ libpthread can schedule thread**

Will CUDA program create 1 million instances of local variables/stack?

8K instances of shared variables? (`support`)

launch over 1 million CUDA threads (over 8K thread blocks)

Assigning work

High-end GPU

(16 cores)

Mid-range

(6 cores)

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

Note: no concept of num_cores in the CUDA programs I have shown: similar in spirit to forall loop in data parallel model examples

CUDA compilation

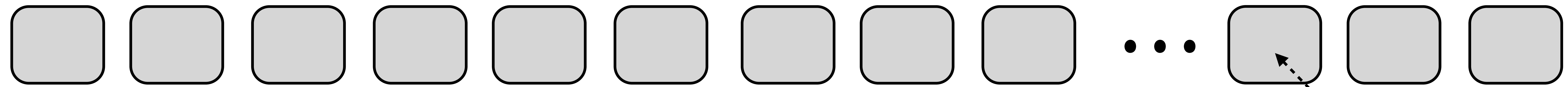
```
#define THREADS_PER_BLK 128

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

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

   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;
}

// host code //////////////////////////////////////////////////////
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 over 8K thread blocks

Key components of compiled CUDA device binary:

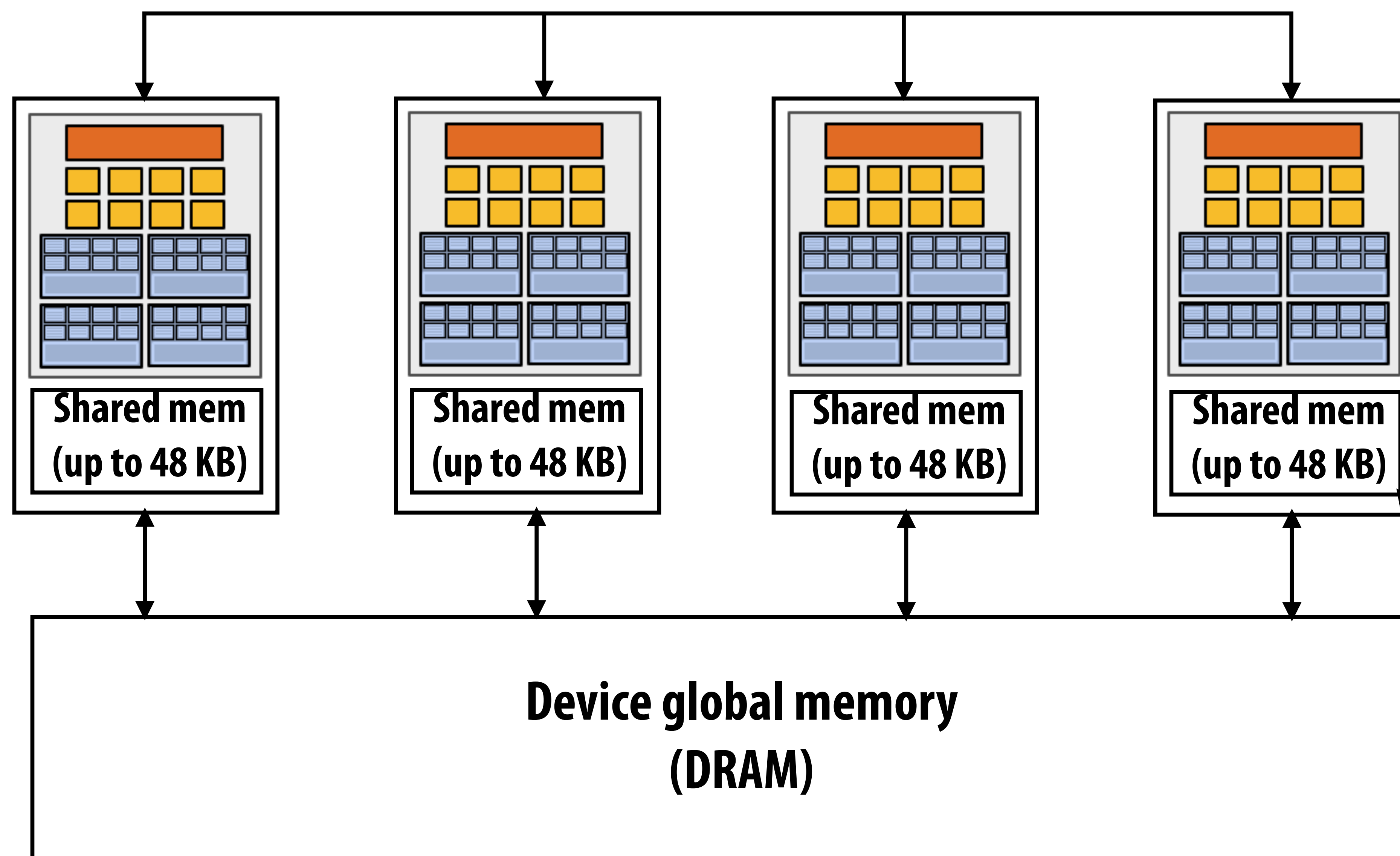
Program text (instructions)
Resource declarations:
- **128 threads per block**
- **X bytes of local data per thread**
- **130 floats (520 bytes) of shared space per block**

CUDA thread-block assignment



Grid of convolve thread blocks (specified by kernel launch)

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

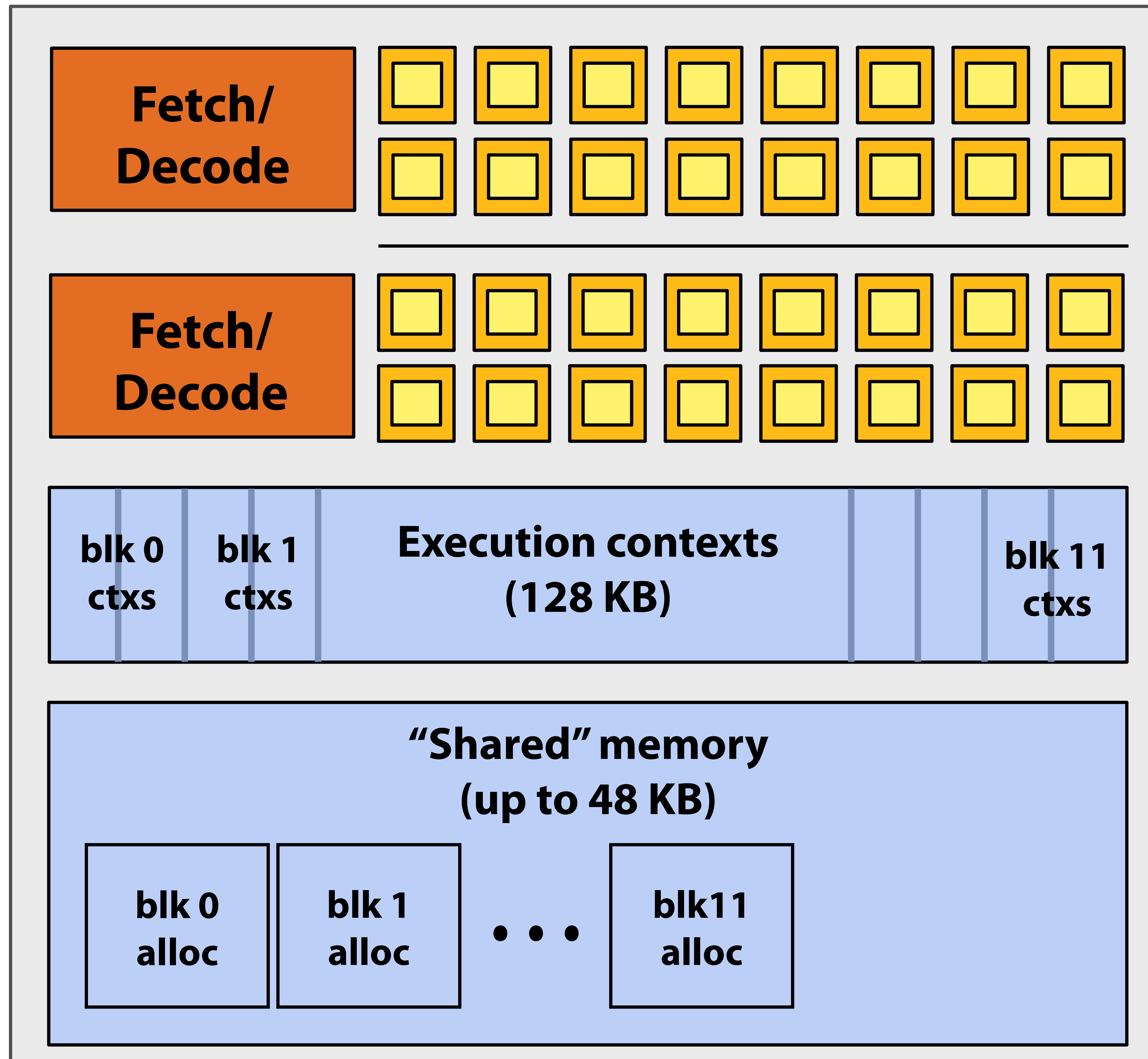


Shared mem is fast on-chip memory

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

Implementation assigns thread blocks ("work") to cores using dynamic scheduling policy that respects resource requirements

NVIDIA Fermi: per core resources



Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

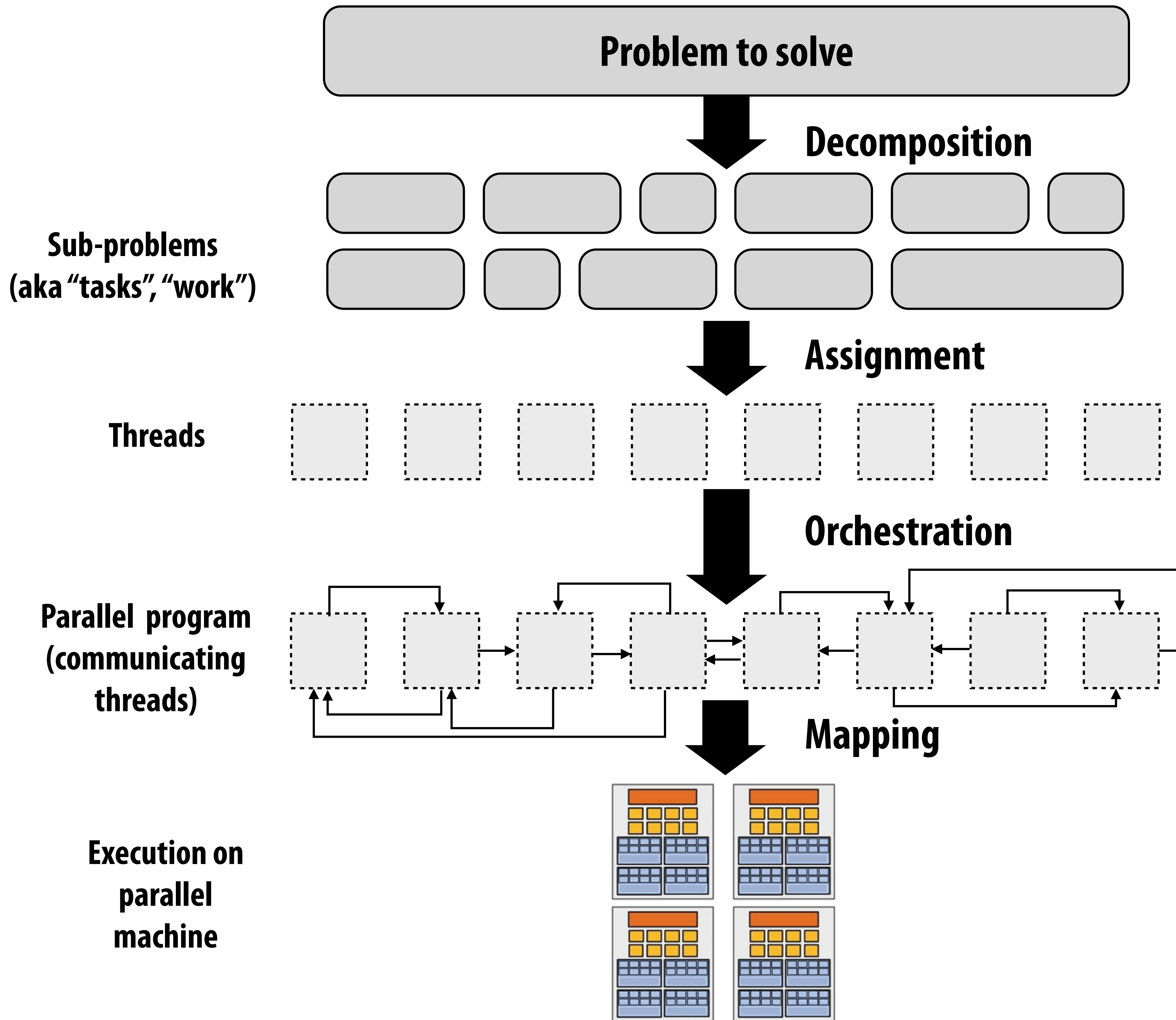
Core Limits:
48 KB of shared memory
Up to 1536 CUDA threads

In the case of convolve:
Thread count limits the number of blocks that can be scheduled on a core at once to twelve

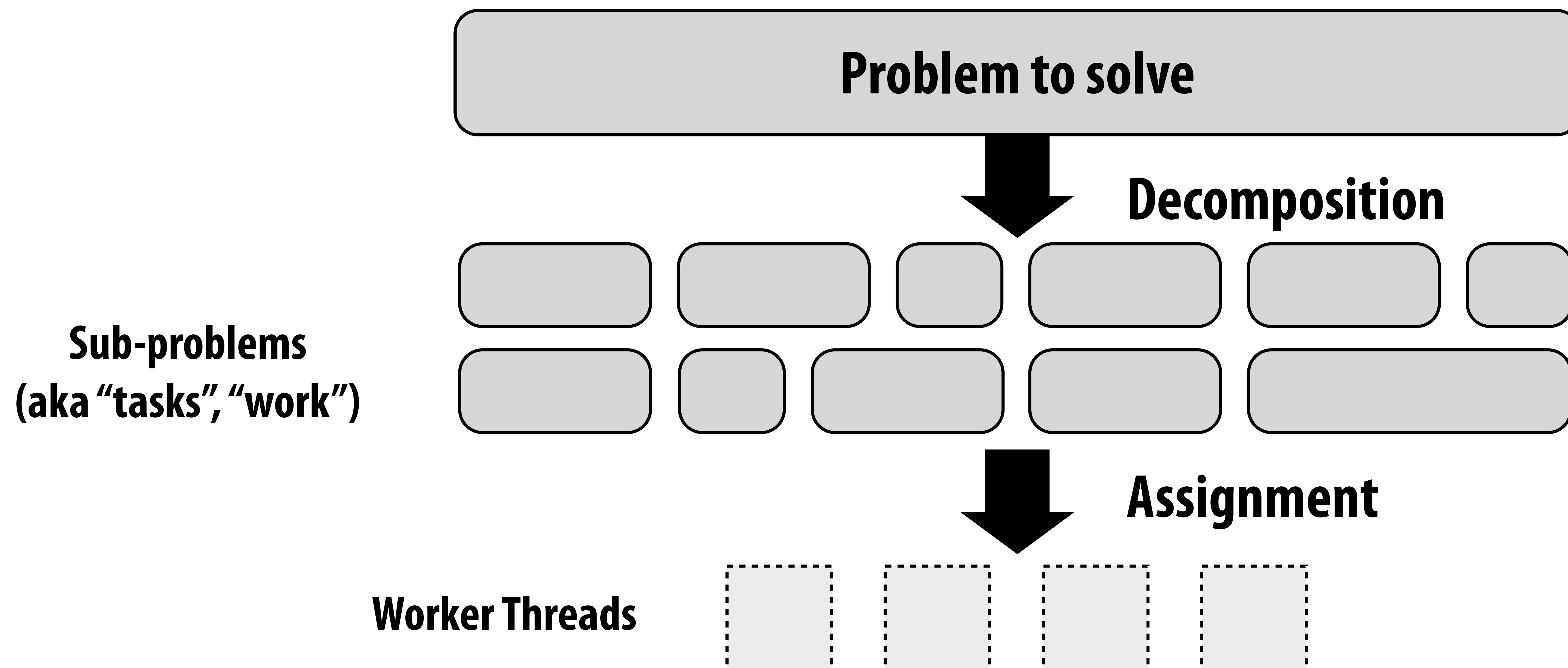
At kernel launch: semi-statically partition core into 12 thread block contexts = 1536 thread contexts (allocate these up front)
Think of these as the "workers"
(note: number of workers is chip resource dependent. Number of logical CUDA threads is NOT)

Reuse contexts for many blocks dynamically assigned to core over duration of computation!

Steps in creating a parallel program



Pools of worker "threads"



Best practice: create enough workers to "fill" parallel machine, and no more:

- One worker per parallel execution resource (e.g., core)
- N workers per core (where N is large enough to hide memory/IO latency
- Pre-allocate resources for each worker
- Dynamically assign tasks to worker threads. (reuse allocation for many tasks)

Examples:

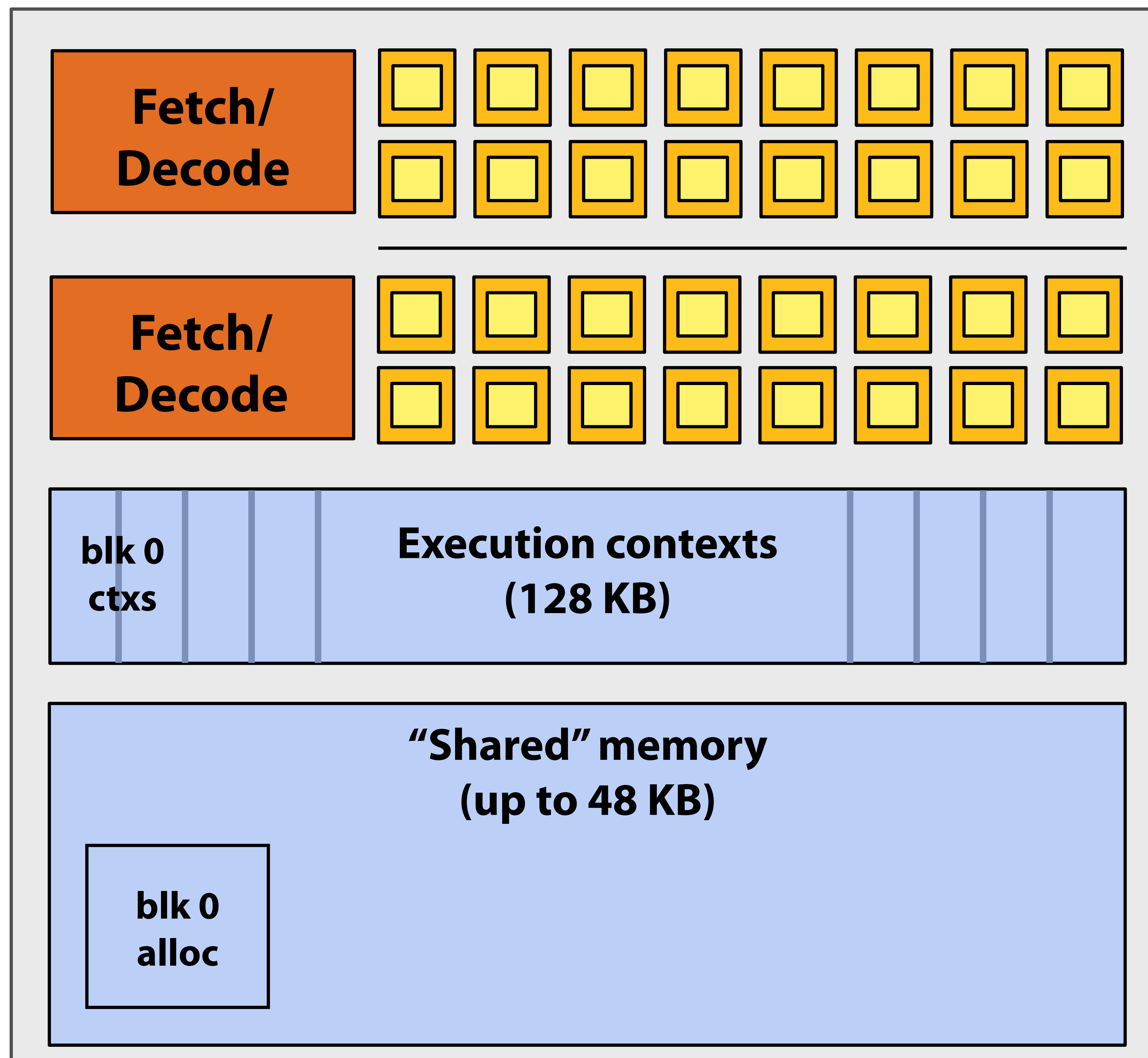
- Thread pool in a web server
  - Number of threads is a function of number of cores, not number of outstanding requests
  - Threads spawned at web server launch, wait for work to arrive
- ISPC's implementation of launch[] tasks
  - Creates one pthread for each hyper-thread on CPU. Threads kept alive for reminder of program

Assigning CUDA threads to core execution resources

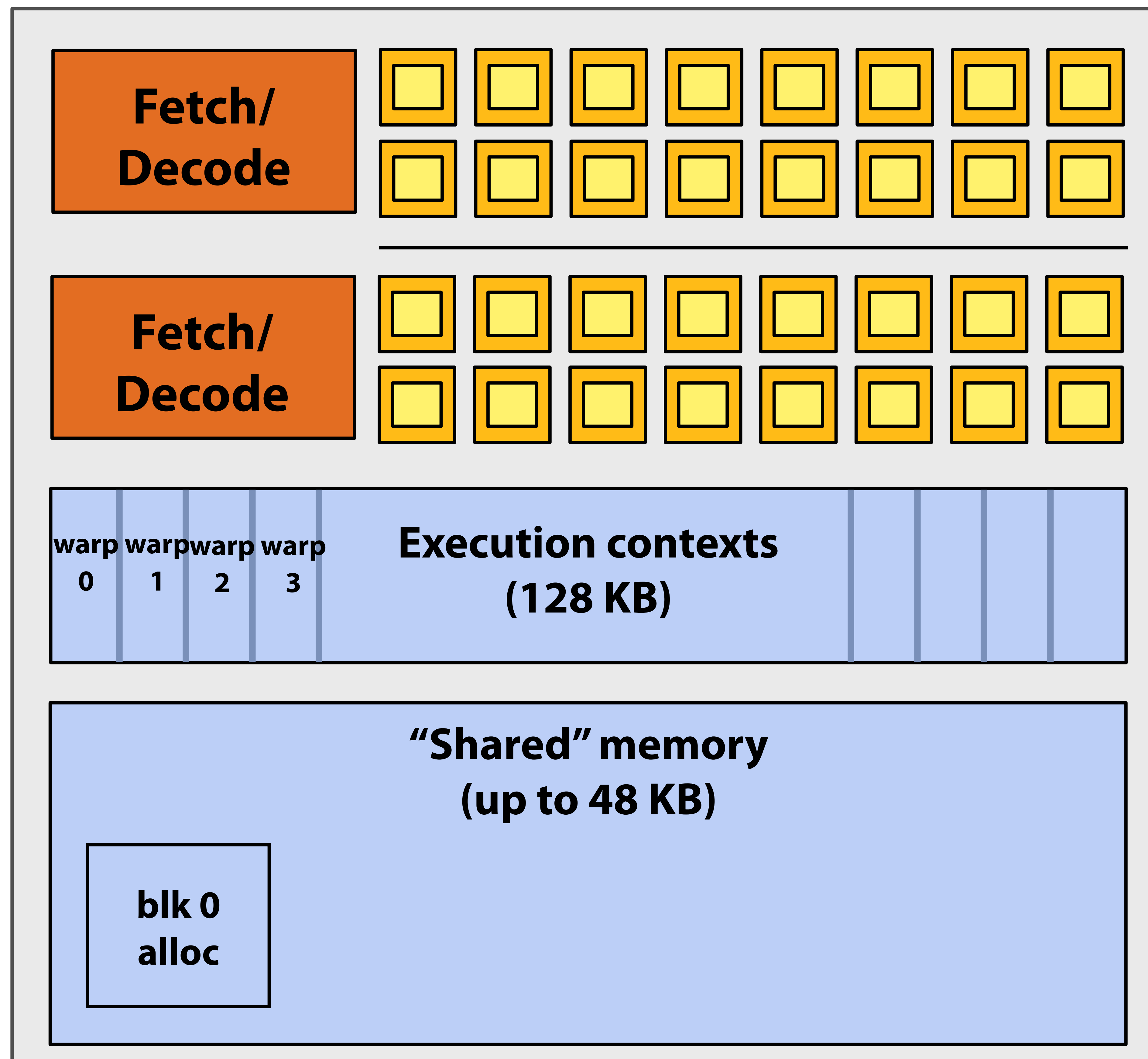
CUDA thread block has been assigned to core

How do we execute logic for the block?

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

   output[index] = result;
}
```



Fermi warps: groups of threads sharing instruction stream




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

   output[index] = result;
}
```

CUDA kernels are SPMD programs

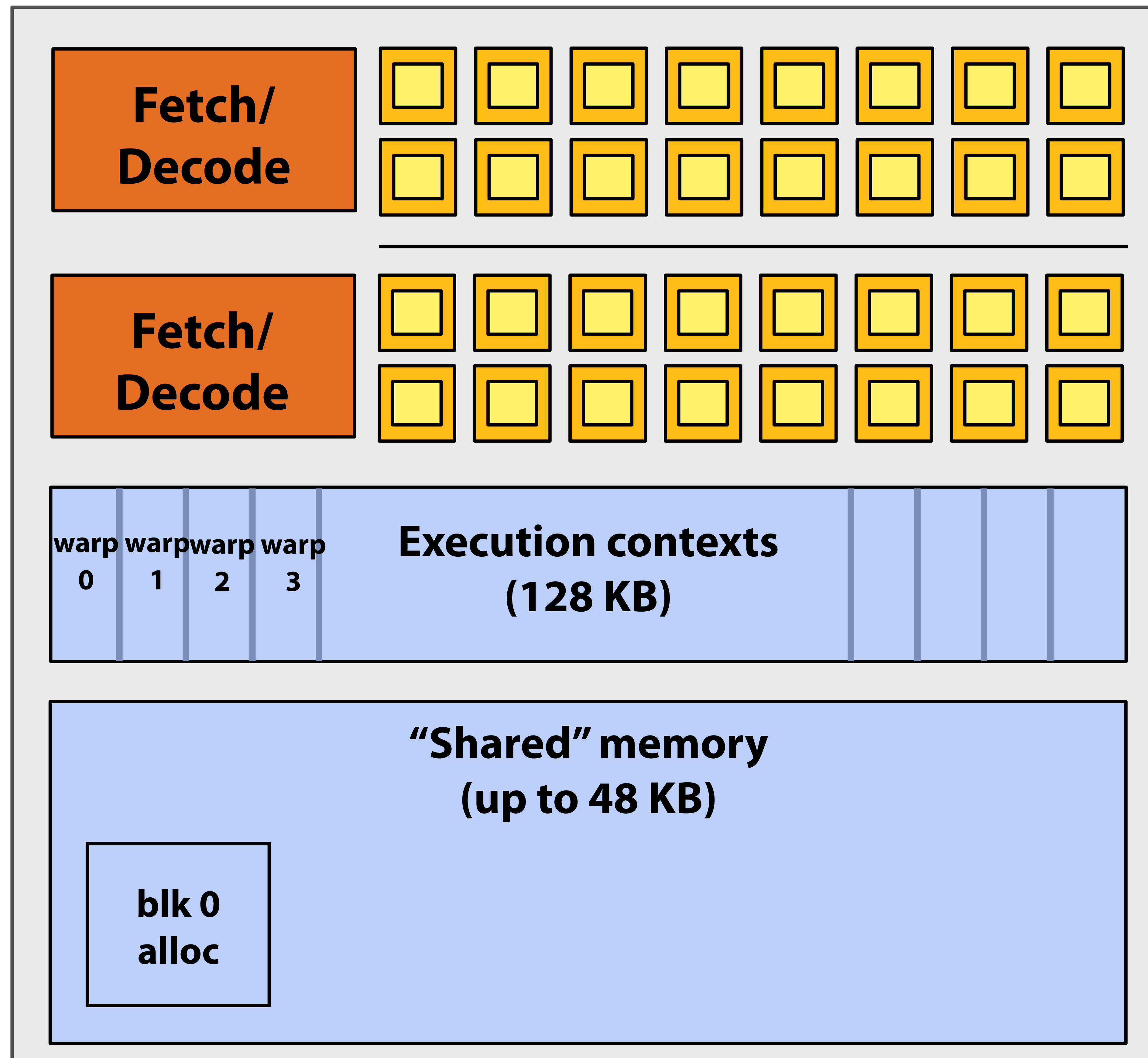
On NVIDIA GPUs groups of 32 CUDA threads share an instruction stream. These groups called "warps".

This thread block consists of 4 warps

(Note: warps are an implementation detail, not a CUDA concept)

Executing warps

(** yes, of course, in reality its more complicated than this)




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

   output[index] = result;
}
```

NVIDIA Fermi core has two independent groups of 16-wide SIMD ALUs (clocked at 2x rate of rest of chip)

Each "slow" clock, core's warp scheduler: **

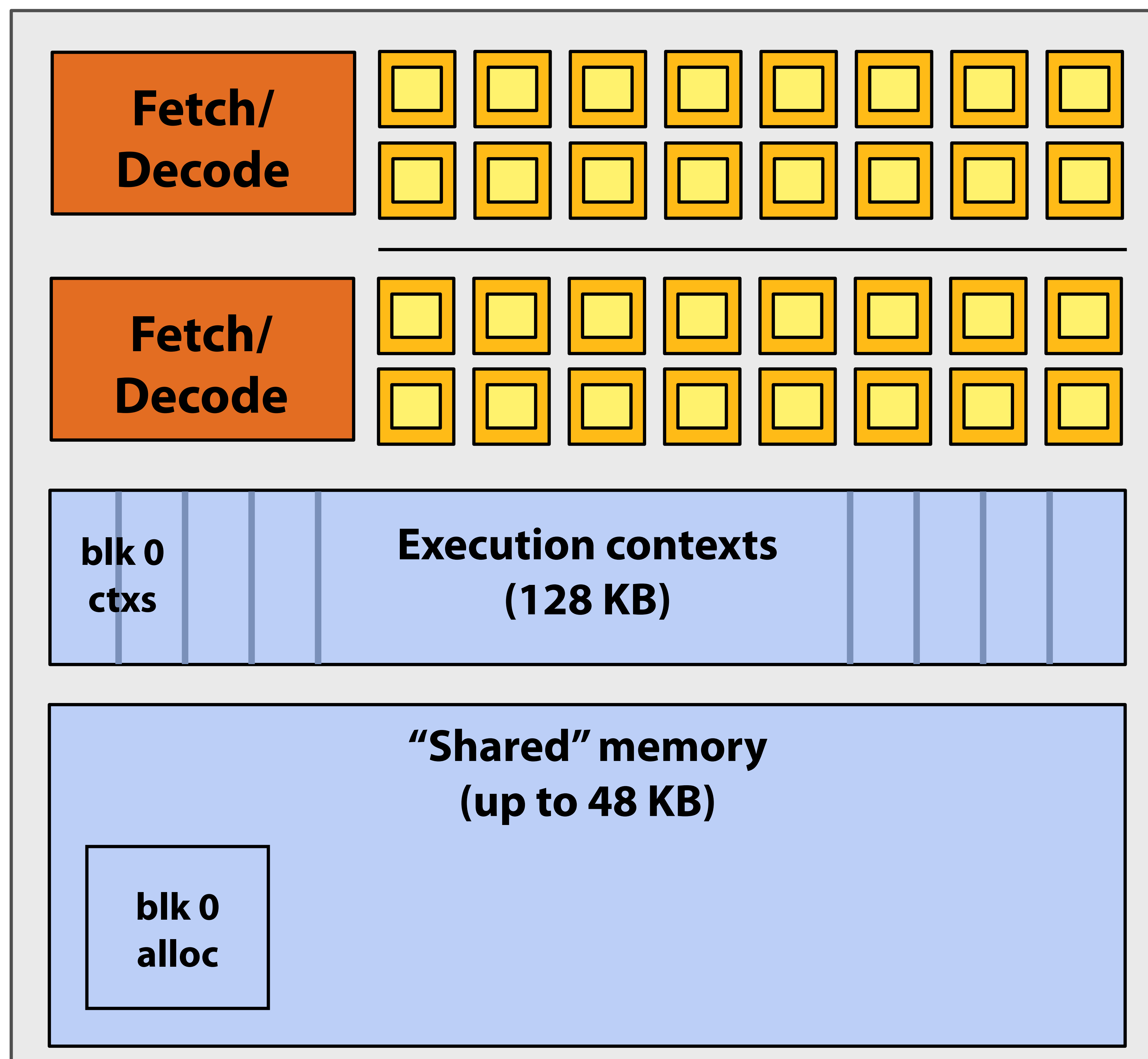
1. Selects 2 runnable warps (warps that are not stalled)
2. Decodes instruction for each warp (can be different instructions)
3. For each warp, executes instruction for all 32 CUDA threads using 16 SIMD ALUs over two fast clocks (has divergence behavior of 32-wide SIMD)

Why allocate execution contexts for all threads in block?

128 logical CUDA threads

Only 2 warps worth of parallel execution in HW

Why not have a pool of two "worker" warps?




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

   output[index] = result;
}
```

CUDA kernels may create dependencies between threads in a block

Simplest example is __syncthreads

Threads in a block cannot be executed by the system in any order when dependencies exist.

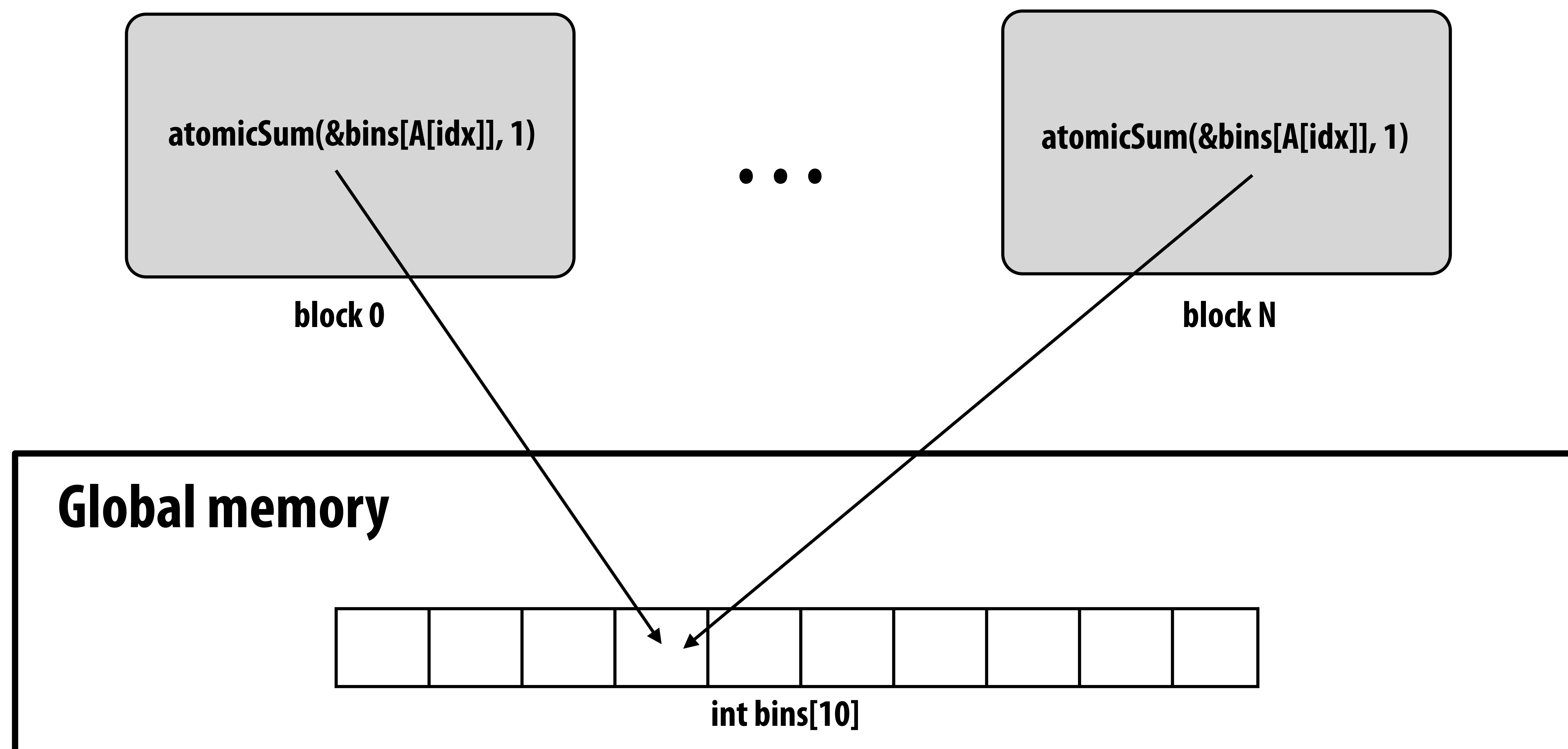
CUDA semantics: threads in a block ARE running concurrently. If a thread in a block is runnable it will eventually be run! (no deadlock)

CUDA execution semantics

- Thread blocks can be scheduled in any order by the system
  - System assumes no dependencies
  - A lot like ISPC tasks, right?
- Threads in a block DO run concurrently
  - When block begins, all threads are running concurrently (these semantics impose a scheduling constraint on the system)
  - A CUDA thread block <u>is itself</u> an SPMD program (like an ISPC gang of program instances)
  - Threads in thread-block are concurrent, cooperating "workers"
- CUDA implementation:
  - A Fermi warp has performance characteristics akin to an ISPC gang of instances (but unlike an ISPC gang, a warp is not manifest in the programming model)
  - All warps in a thread block are scheduled onto the same core, allowing for high-BW/ low latency communication through shared memory variables
  - When all threads in block complete, block resources become available for next block

Implications of CUDA atomics

- Notice that I did not say CUDA thread blocks were independent. I only claimed they can be scheduled in any order.
- CUDA threads can atomically update shared variables in global memory
  - Example: build a histogram of values in an array
- Observe how this use of atomics does not impact implementation's ability to schedule blocks in any order (atomics for mutual exclusion, and nothing more)



Implications of CUDA atomics

- But what about this?
- Consider single core GPU, resources for one block per core
  - What are the possible outcomes of different schedules?

```
// do stuff
atomicAdd(&myFlag, 1);
```

block 0

. . .

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

// do stuff
```

block N

Global memory

int myFlag

(assume initialized to 0)

“Persistent thread” technique

```
#define THREADS_PER_BLK 128
#define BLOCKS_PER_CHIP 15 * 12   // specific to a certain GTX 480 GPU

__device__ int workIndex = 0;  // global mem variable

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

  __shared__ float support[THREADS_PER_BLK+2];  // shared across block
  while (1) {
     int blockWorkIndex = atomicInc(workCounter, THREADS_PER_BLK);
     if (blockWorkIndex >= N)
        break;

     int index = blockWorkIndex + threadIdx.x; // thread local
     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;

      __syncthreads();
   }
}

// host code //////////////////////////////////////////////////////
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<<<BLOCKS_PER_CHIP, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

Programmer launches exactly as many thread-blocks as will fill the GPU

(Exploit knowledge of implementation: that GPU will in fact run all blocks concurrently)

Work assignment to blocks is implemented entirely by the application

(circumvents GPU thread block scheduler, and intended CUDA thread block semantics)

Now programmer’s mental model is that *all* threads are concurrently running on the machine at once.

CUDA summary

- Execution semantics
  - Partitioning problem into thread blocks is in the spirit of the data-parallel model (intended to be machine independent, CUDA schedules blocks onto any number of cores)
  - Threads in a thread block run concurrently (they have to, since they cooperate)
    - 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 host/device memorys
  - Local/block shared/global variables
    - Loads/stores move data between them (as a result can think about this as a NUMA address space, or different address spaces)
- Key implementation detail:
  - Threads in a thread block scheduled onto same GPU core to allow fast communication through shared memory