Lecture 5:

GPU Architecture & CUDA Programming

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

Alt-J

Nara

(This Is All Yours)

“Have you seen the amount of compute capability packed into modern GPUs? It’s all yours for the taking, but only if you’ve got the right type of workload.”

-Joe Newman

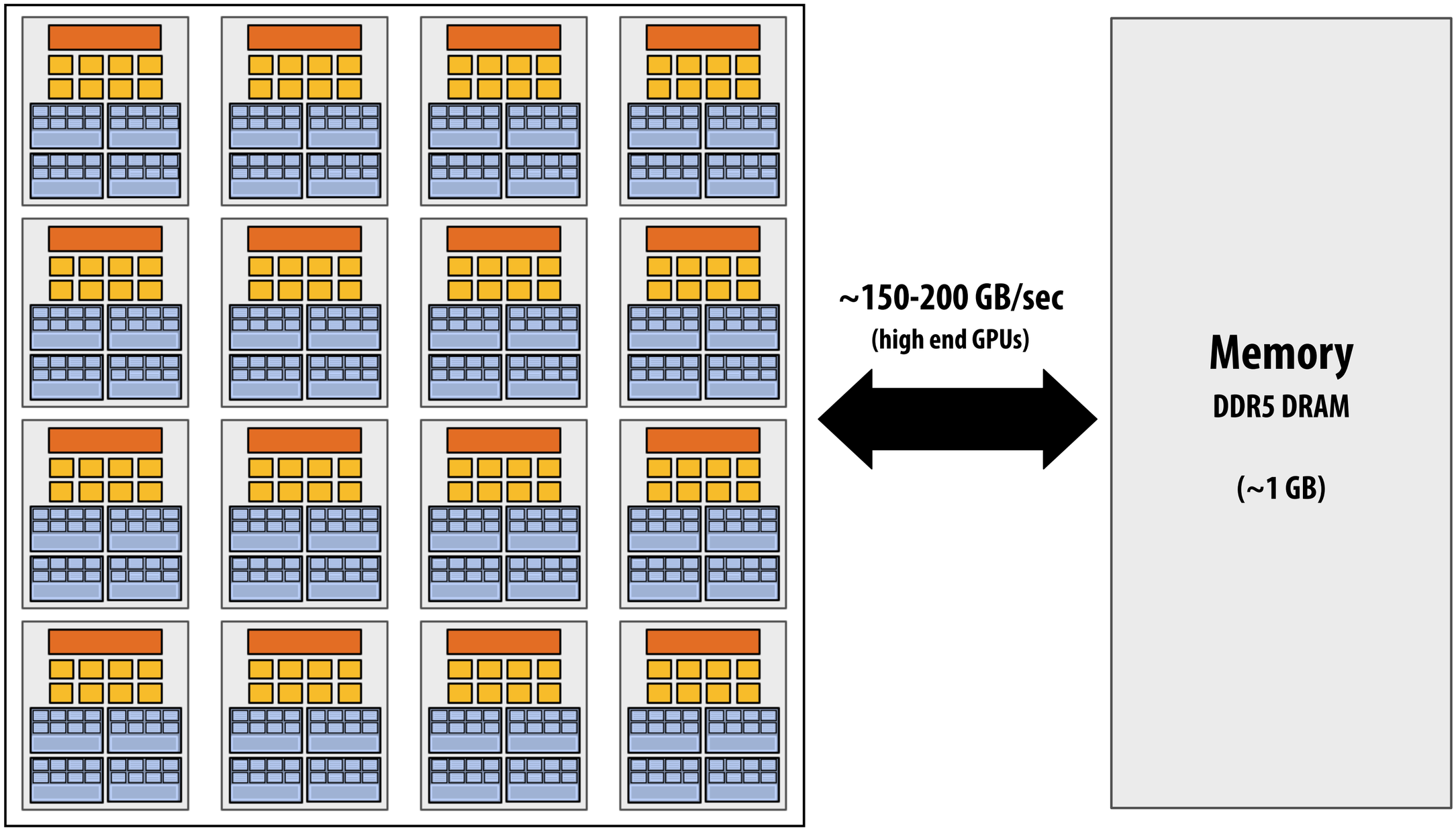
Today

- History: how graphics processors, originally designed to accelerate 3D games like Quake, evolved into 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



GPU

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 (for fun)

What GPUs were originally designed to do: 3D rendering

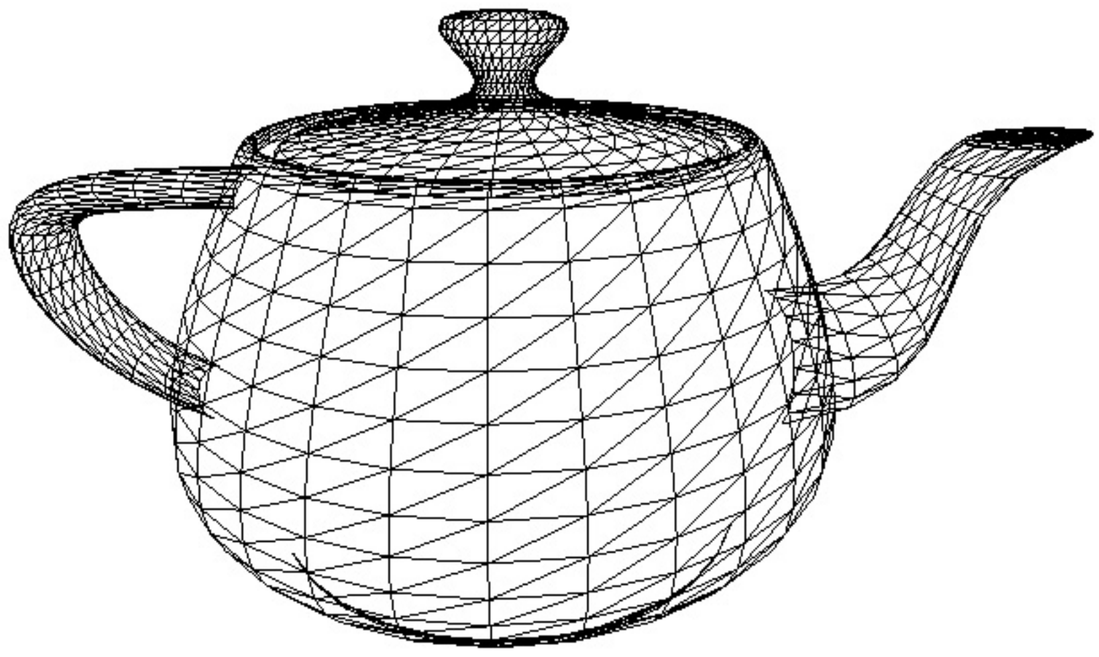


Image credit: Henrik Wann Jensen

Input: description of a scene:

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

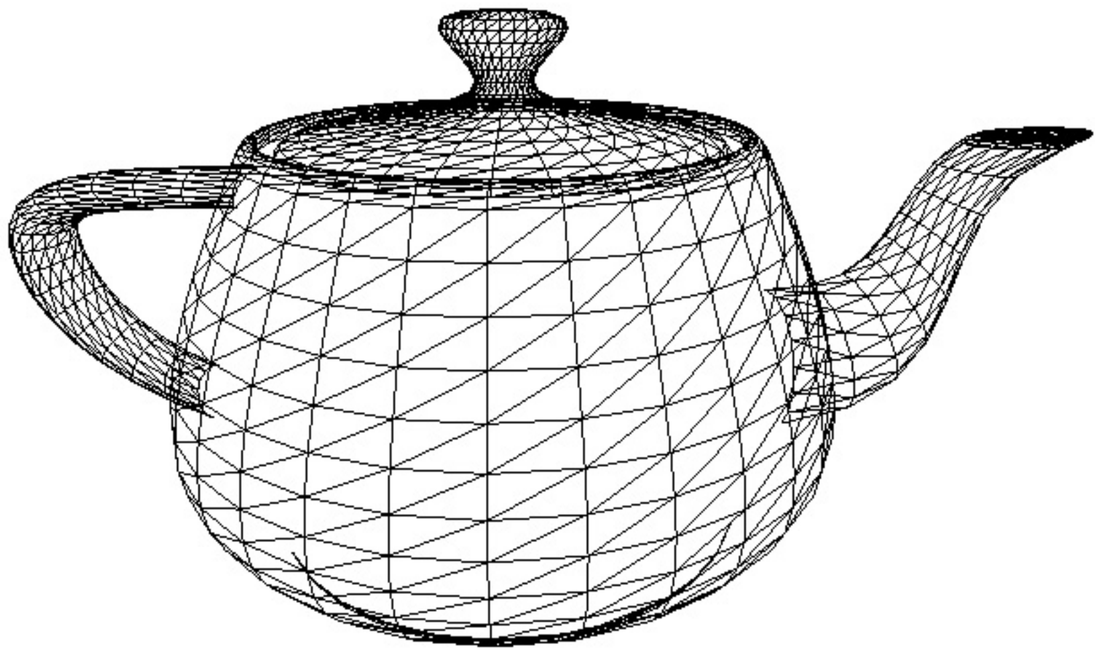
Output: image of the scene

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

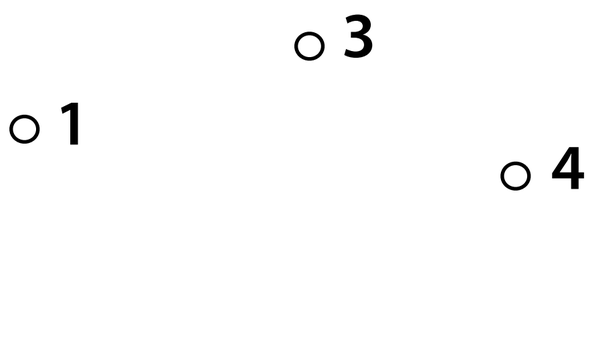
Tip: how to explain a system

- Step 1: describe the things (key entities) that are manipulated
  - The nouns

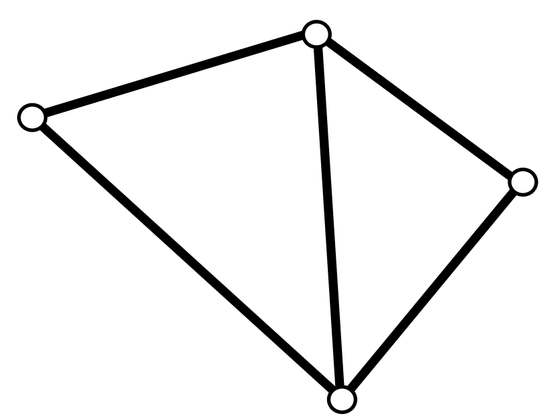
Real-time graphics primitives (entities)



Represent surface as a 3D triangle mesh

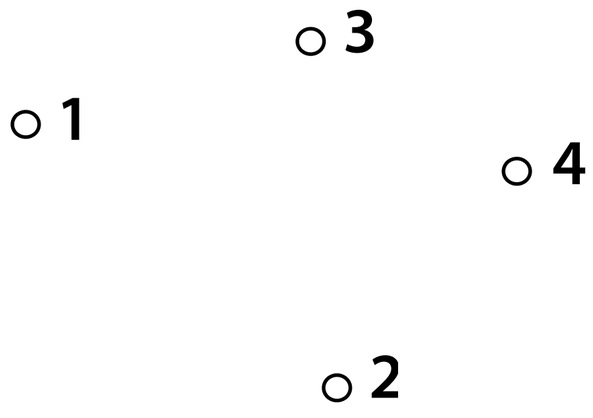


Vertices

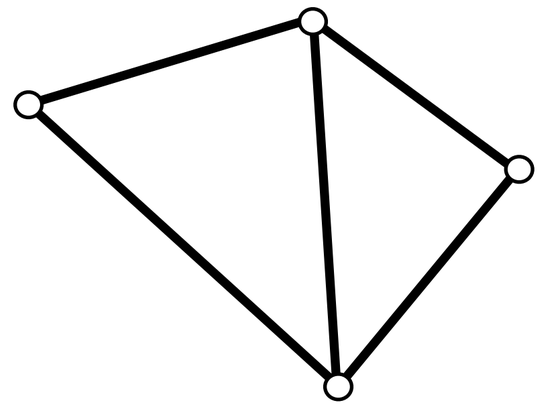


Primitives
(e.g., triangles, points, lines)

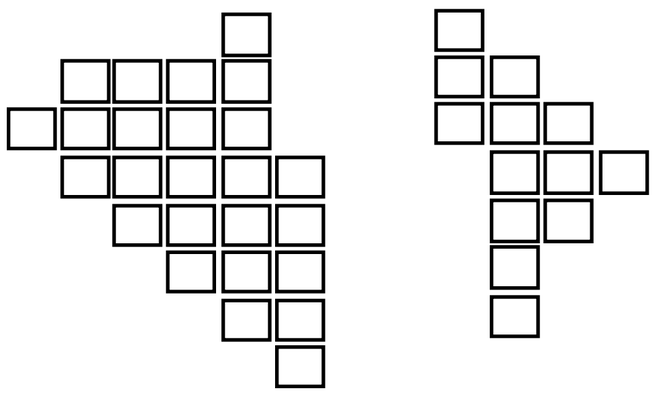
Real-time graphics primitives (entities)



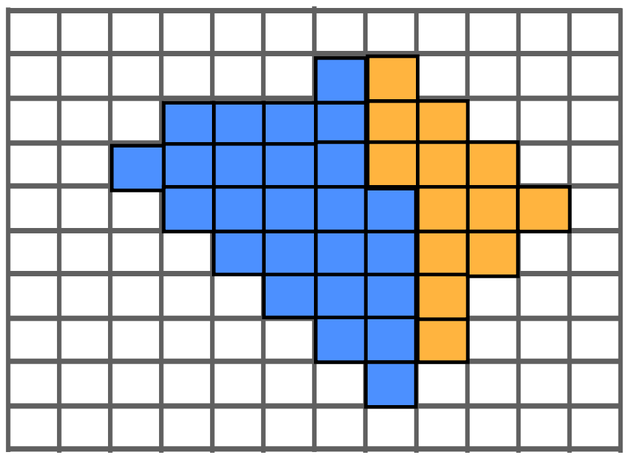
Vertices



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



Fragments



Pixels (in an image)

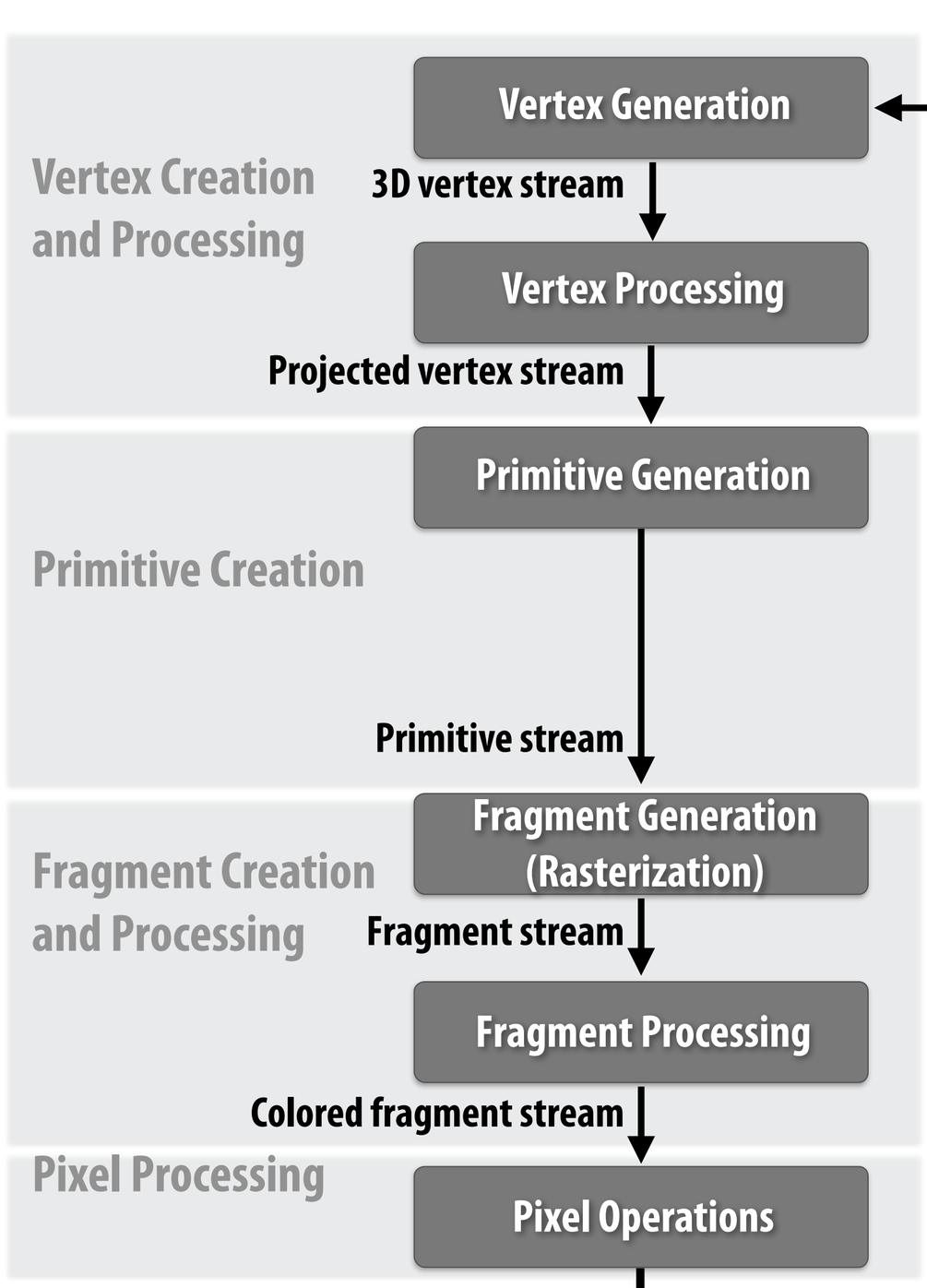
How to explain a system

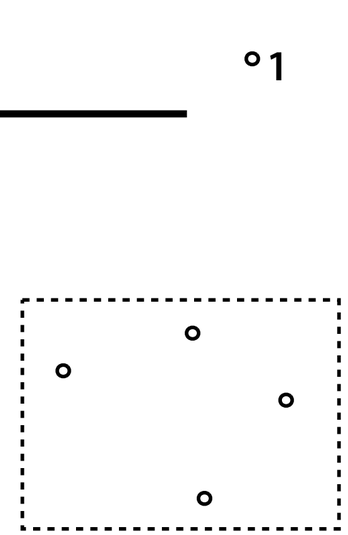
- **Step 1: describe the <u>things</u> (key entities) that are manipulated**
  - **The nouns**

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

Graphics pipeline architecture

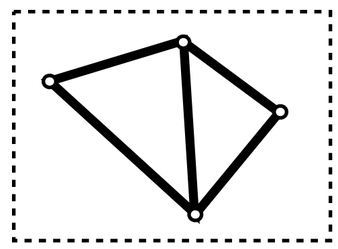
Performs operations on vertices, triangles, fragments, and pixels



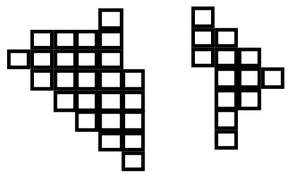


Input: vertices in 3D space + connectivity

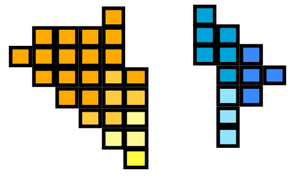
Vertex processing stage computes were vertices appear on screen given a camera position



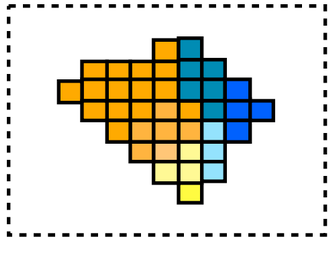
Group vertices into triangles positioned on screen



Fragment generation creates one fragment for each pixel covered by the triangle



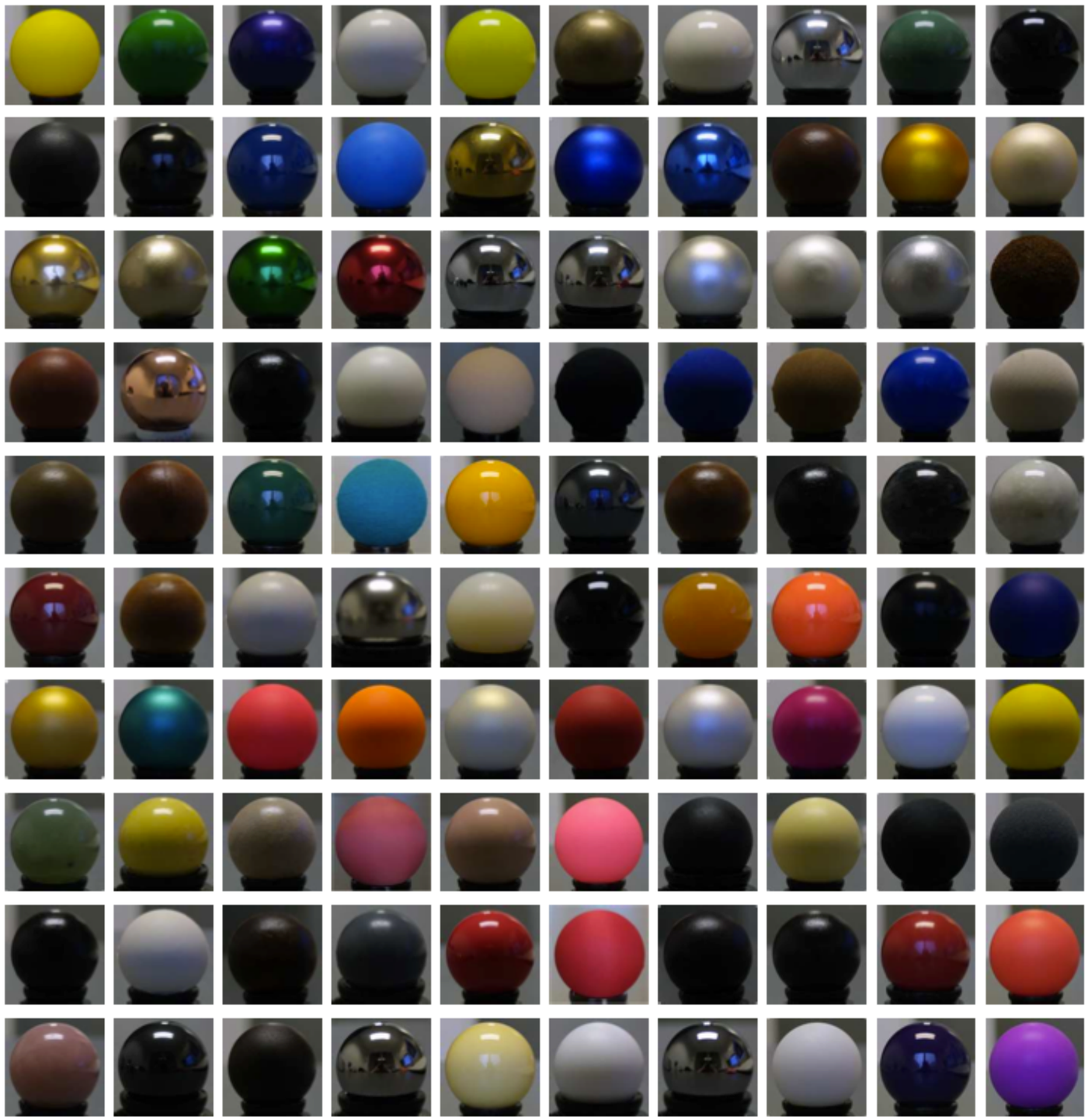
Fragment processing colors the fragments based on the surface characteristics at this pixel



Output image pixels contain color of the closest fragment at each pixel

Fragment processing simulates reflection of light off of real-world materials

Example materials:



Images from Matusik et al. SIGGRAPH 2003

Early graphics programming (via OpenGL API)

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

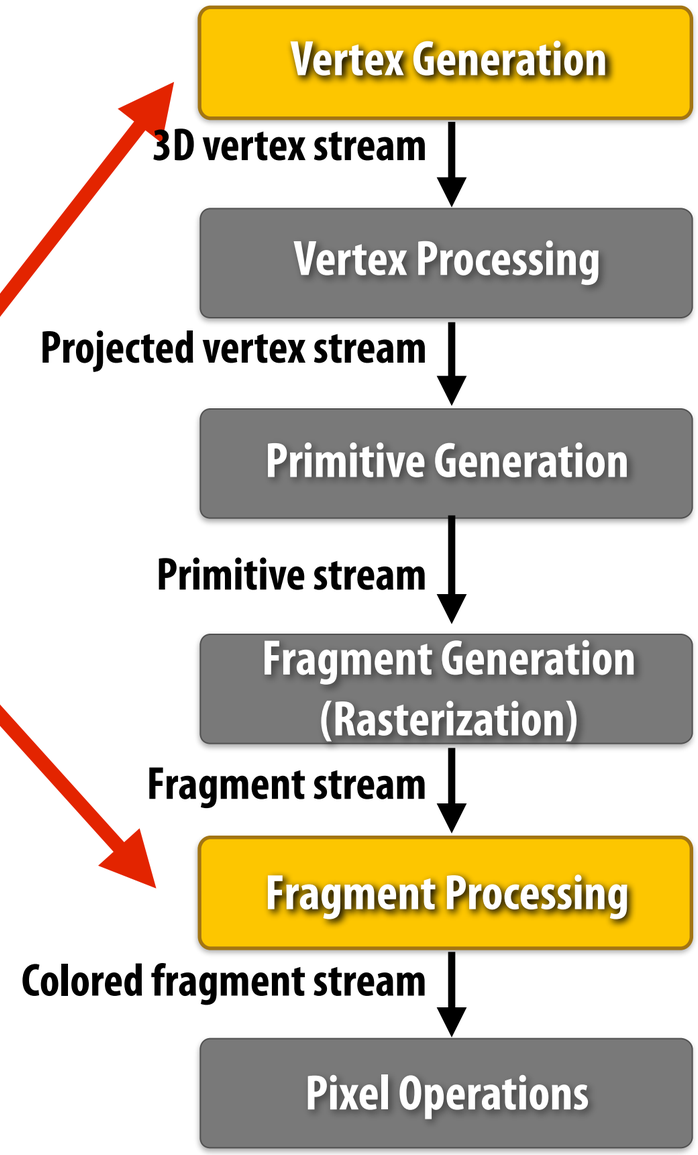
  - `glLight(light_id, parameter_id, parameter_value)`
    - Examples of light parameters: ambient/diffuse/specular color, position, direction

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

Graphics shading languages

- Allow application to 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
  - Pipeline maps shader function onto all elements of input stream



Example fragment shader *

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

HLSL language 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);
}
```

read-only global variables

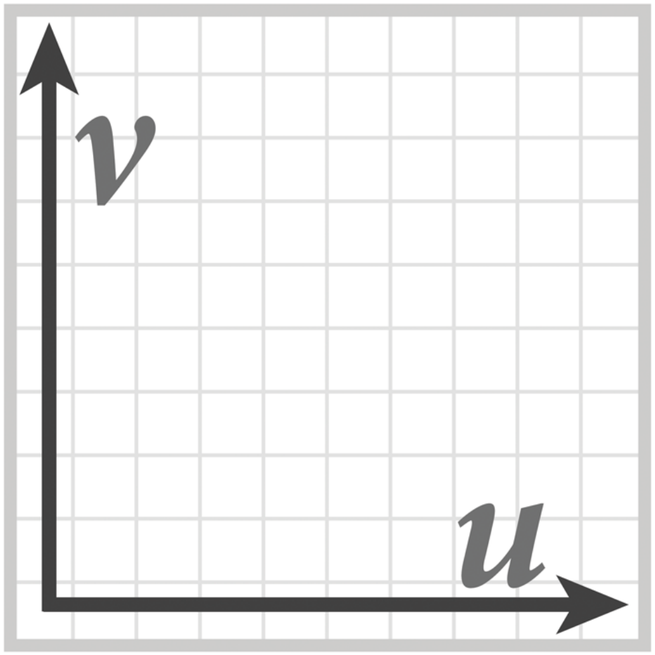
per-fragment inputs

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

per-fragment output: surface color at pixel

myTex is a texture map

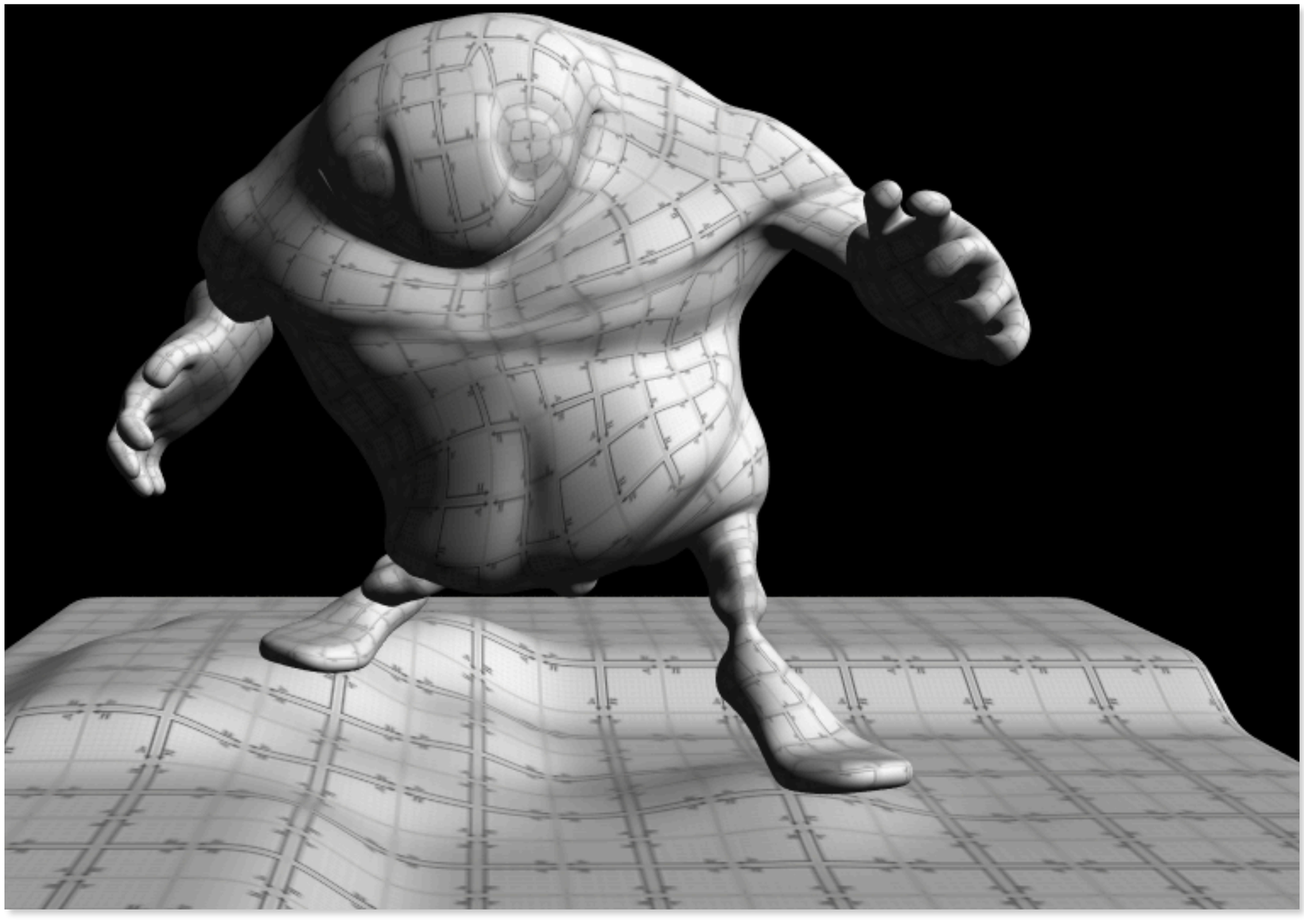
myTex =



* 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

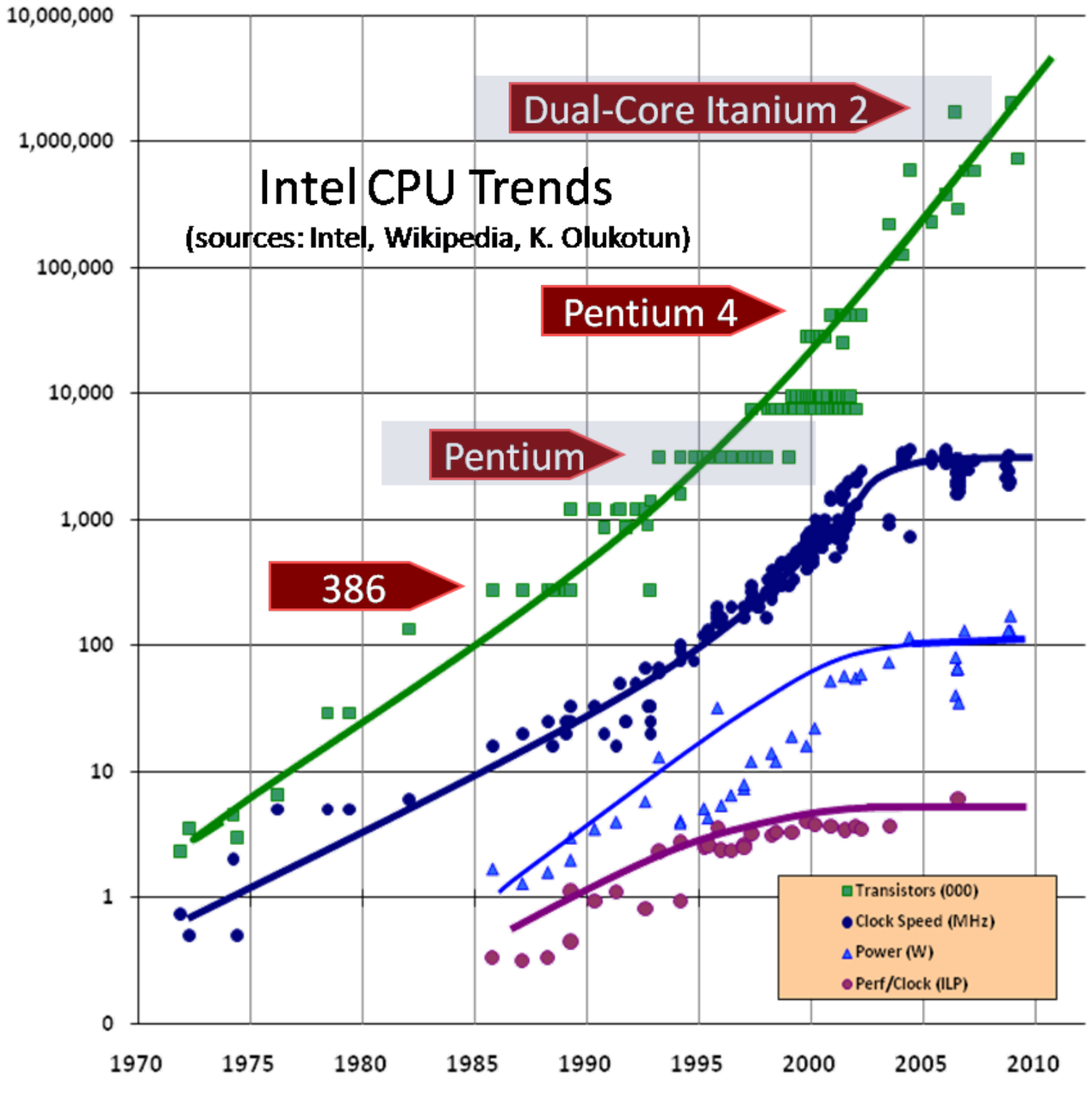
Image contains output of `diffuseShader` for each pixel covered by surface
(Pixels covered by multiple surfaces contain output from surface closest to camera)



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

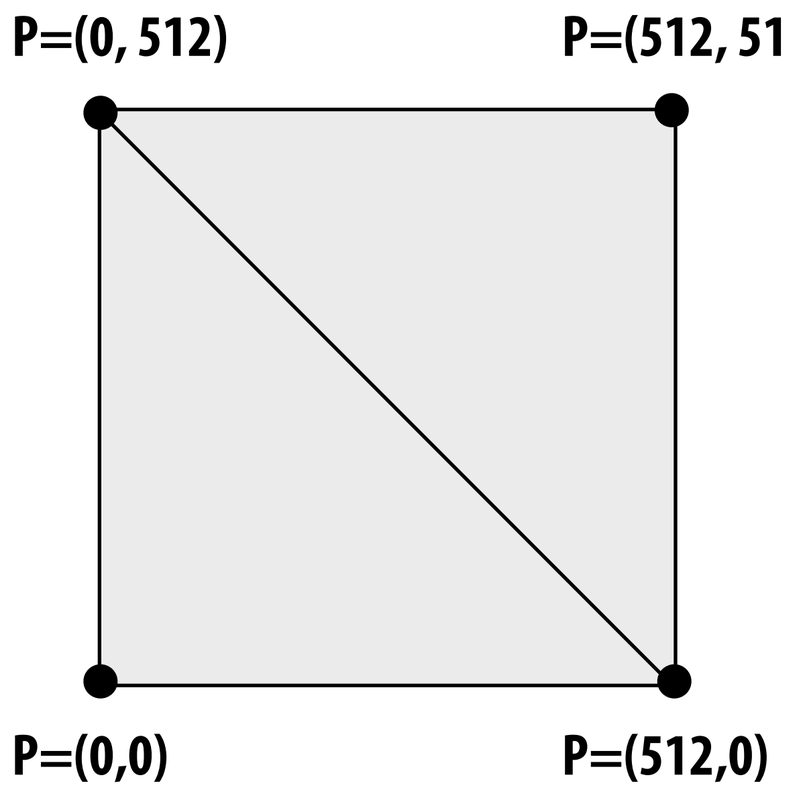
Set OpenGL output image size to be output array size (e.g., 512 x 512)

Render 2 triangles that exactly cover screen

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

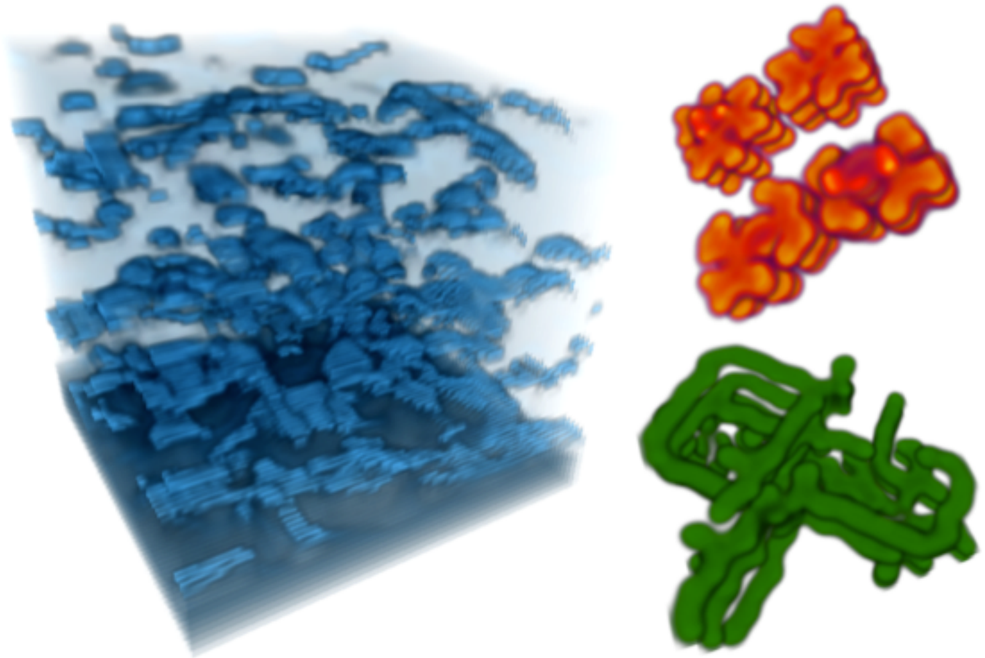
We now can use the GPU like a data-parallel programming system.

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

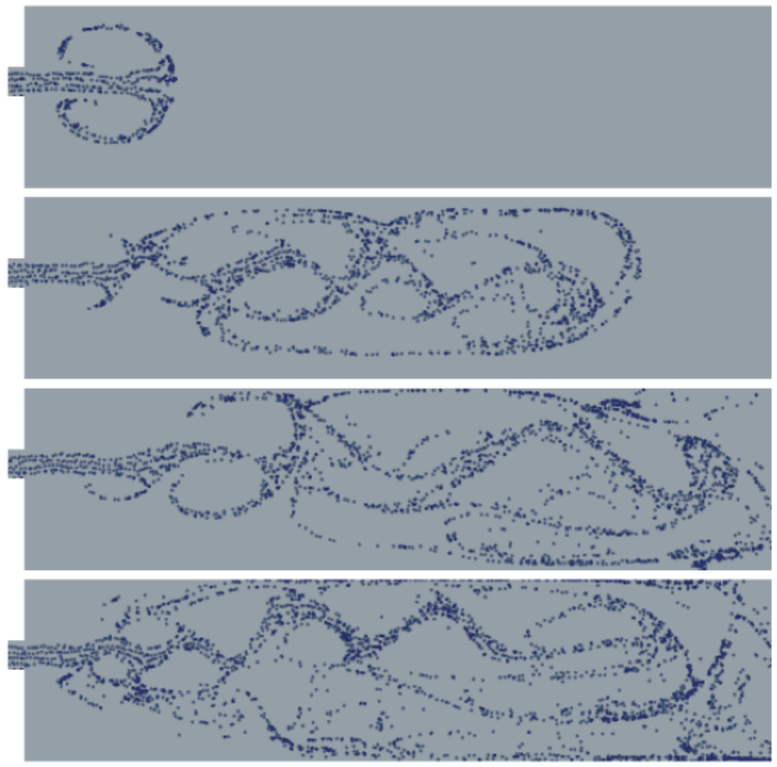


“GPGPU” 2002-2003

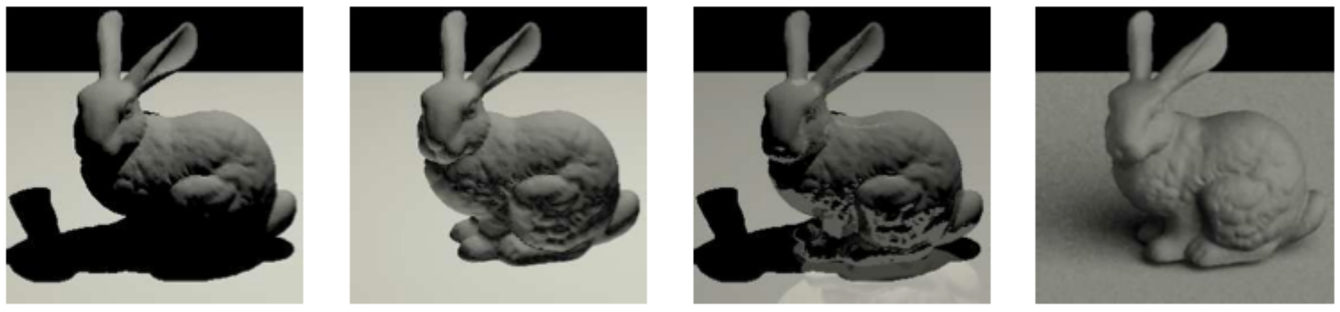
GPGPU = “general purpose” computation on GPUs



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 research project
- Abstract GPU as data-parallel processor

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

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

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

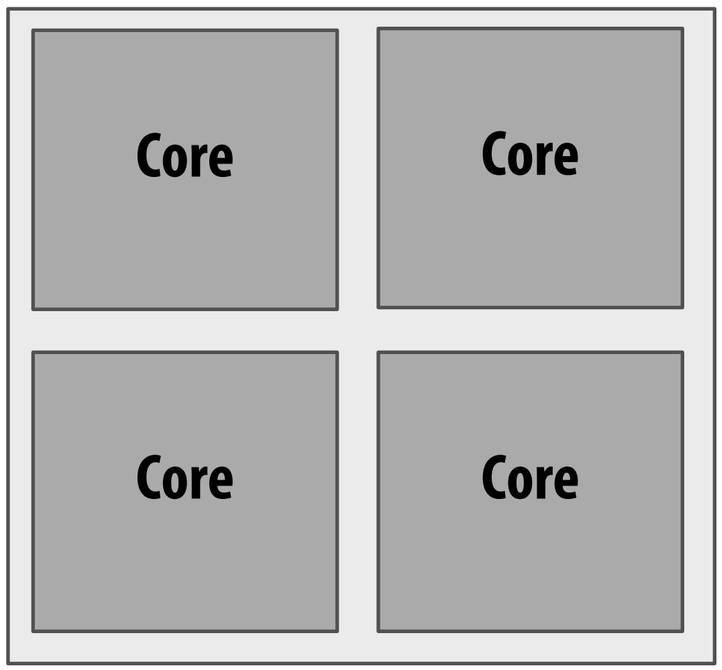
- Brook compiler turned generic stream program into OpenGL commands such as drawTriangles()

GPU Compute Mode

NVIDIA Tesla architecture (2007)

(GeForce 8xxx series)

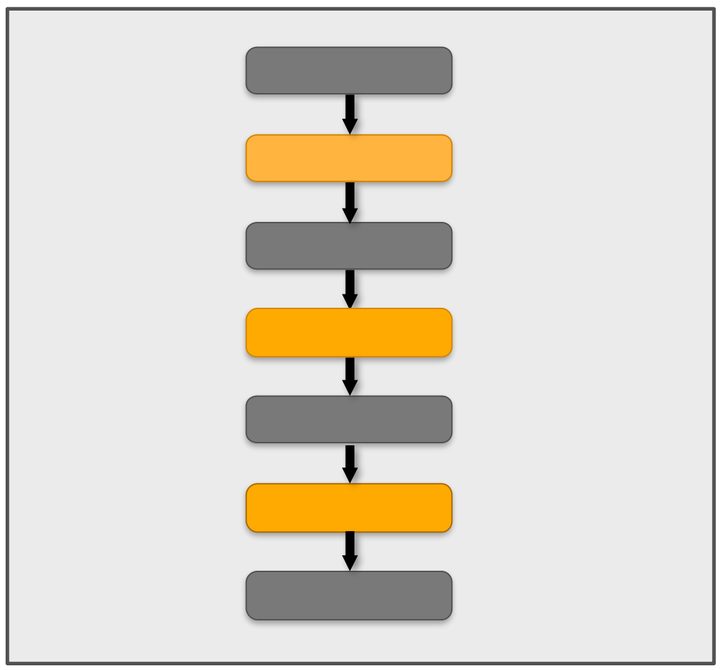
First alternative, non-graphics-specific ("compute mode") software interface to GPU



Multi-core CPU architecture

CPU presents itself to OS as a multi-processor system

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

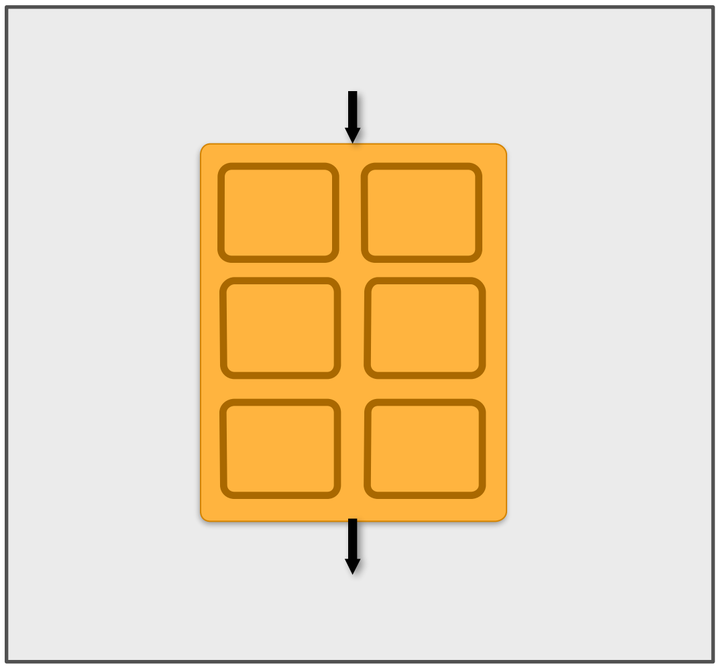


Pre-2007 GPU architecture

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

Set screen size
Set shader program for pipeline
DrawTriangles

(** interface also included many other commands for configuring graphics pipeline)



Post-2007 "compute mode" GPU architecture

GPU presents a new data-parallel interface to system software (driver):

Set kernel program
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: CUDA's abstractions closely match the capabilities/performance characteristics of modern GPUs (low abstraction distance)
- Note: OpenCL is an open standards version of CUDA
  - CUDA only runs on NVIDIA GPUs
  - OpenCL runs on CPUs and GPUs from many vendors
  - Almost everything I say about CUDA also holds for OpenCL
  - CUDA is better documented, thus I find it preferable to teach with

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

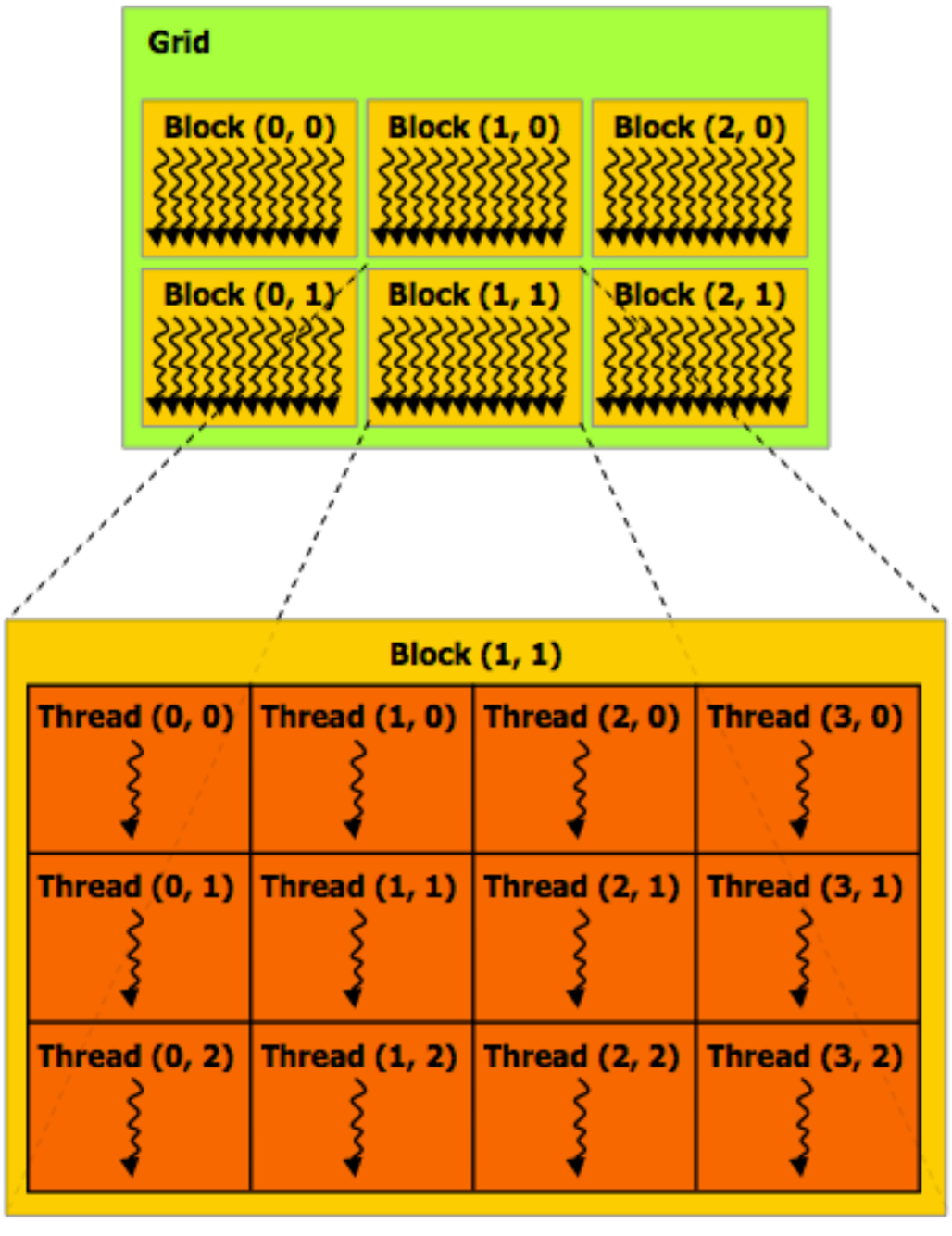
- Specifically, be careful with the use of the term CUDA thread. A CUDA thread presents a similar abstraction as a pthread in that both correspond to logic threads of control, but it's implementation is very different

- 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




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

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

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

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

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

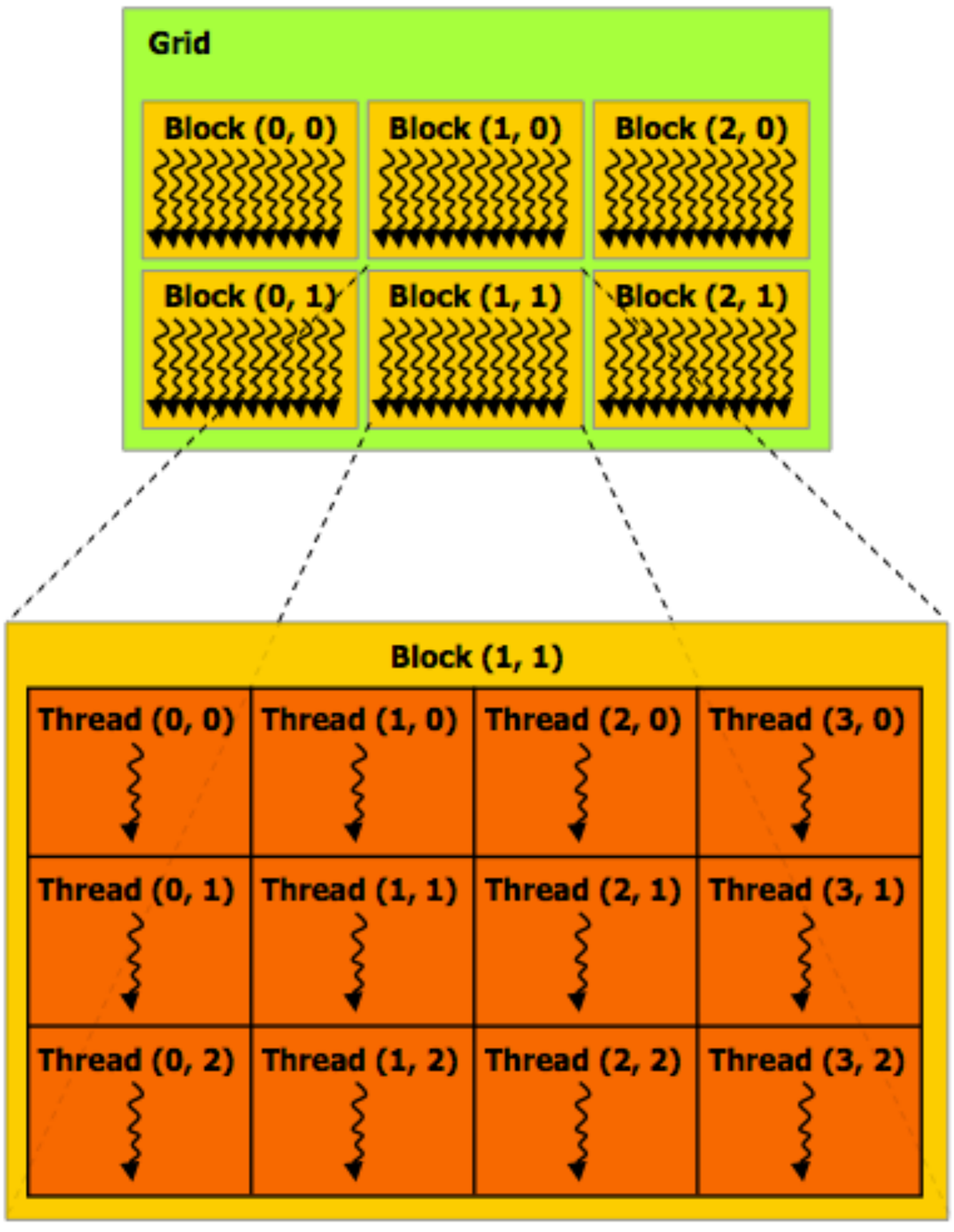
"Device" code
SPMD execution on GPU

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

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

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

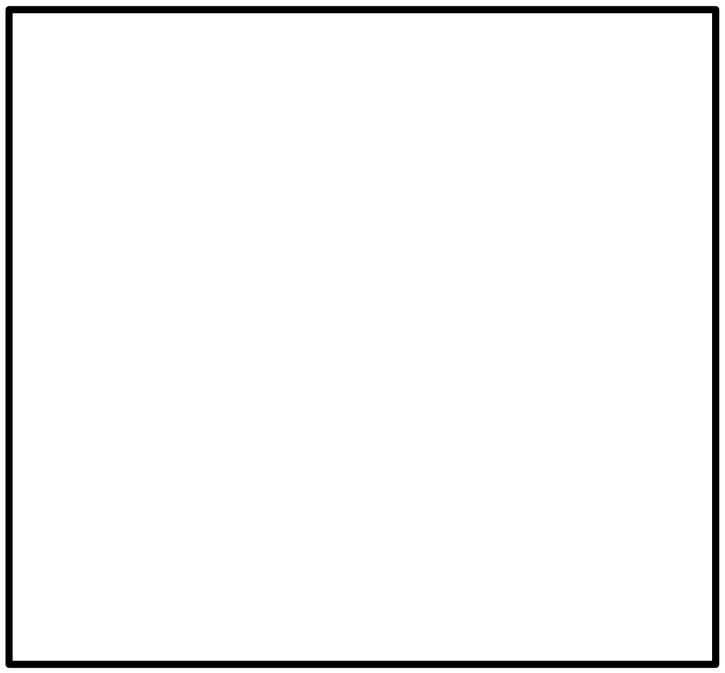
Number of kernel invocations is not determined by size of data collection
(Kernel launch is not map(kernel, collection) as was the case with graphics shader programming)




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

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

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);
```

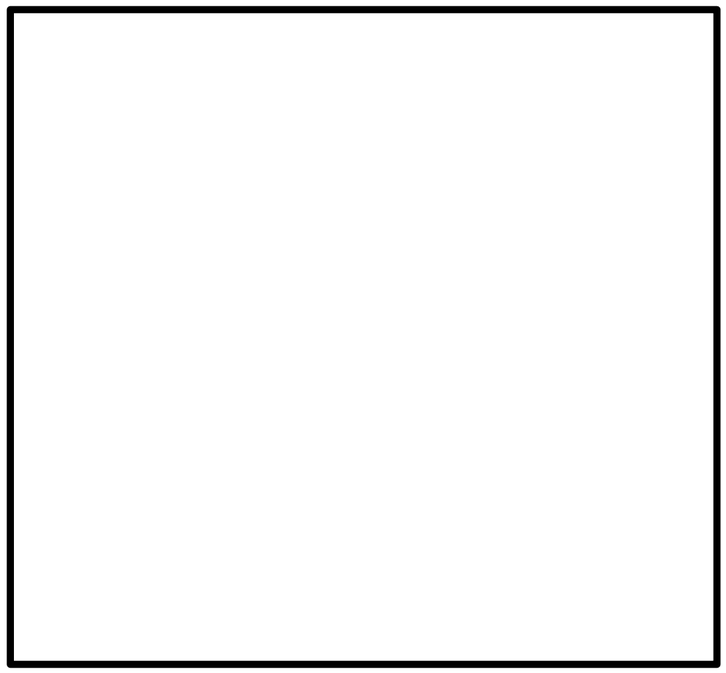
CUDA execution model

Host
(serial execution)



Implementation: CPU

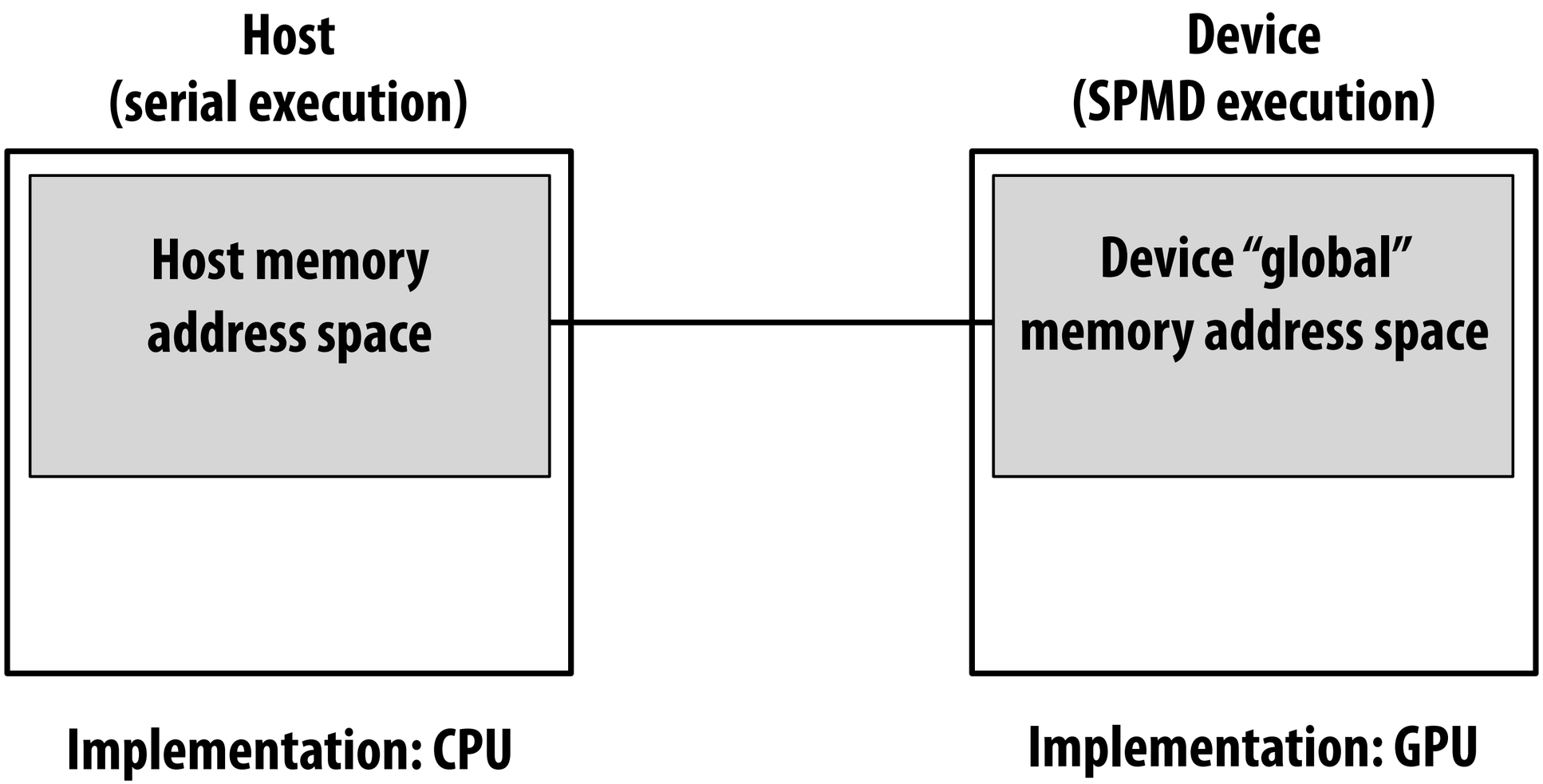
Device
(SPMD execution)



Implementation: GPU

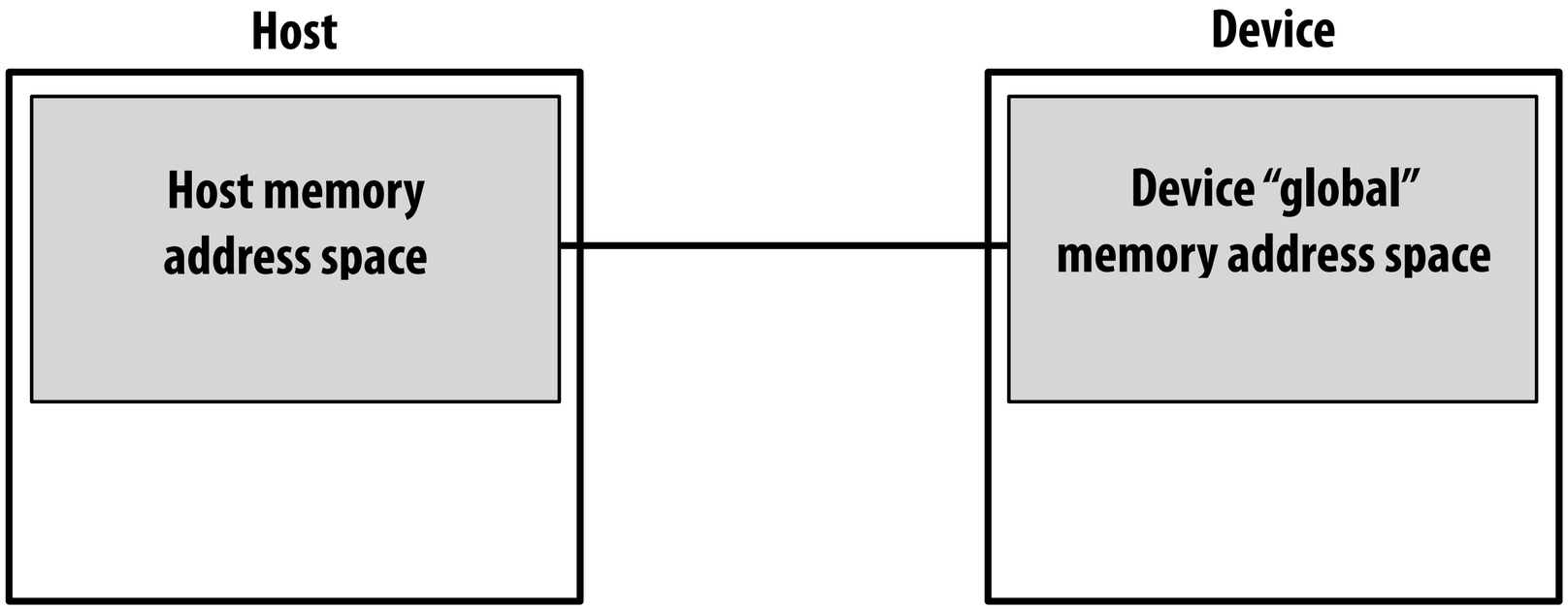
CUDA memory model

Distinct host and device address spaces



memcpy primitive

Move data between address spaces

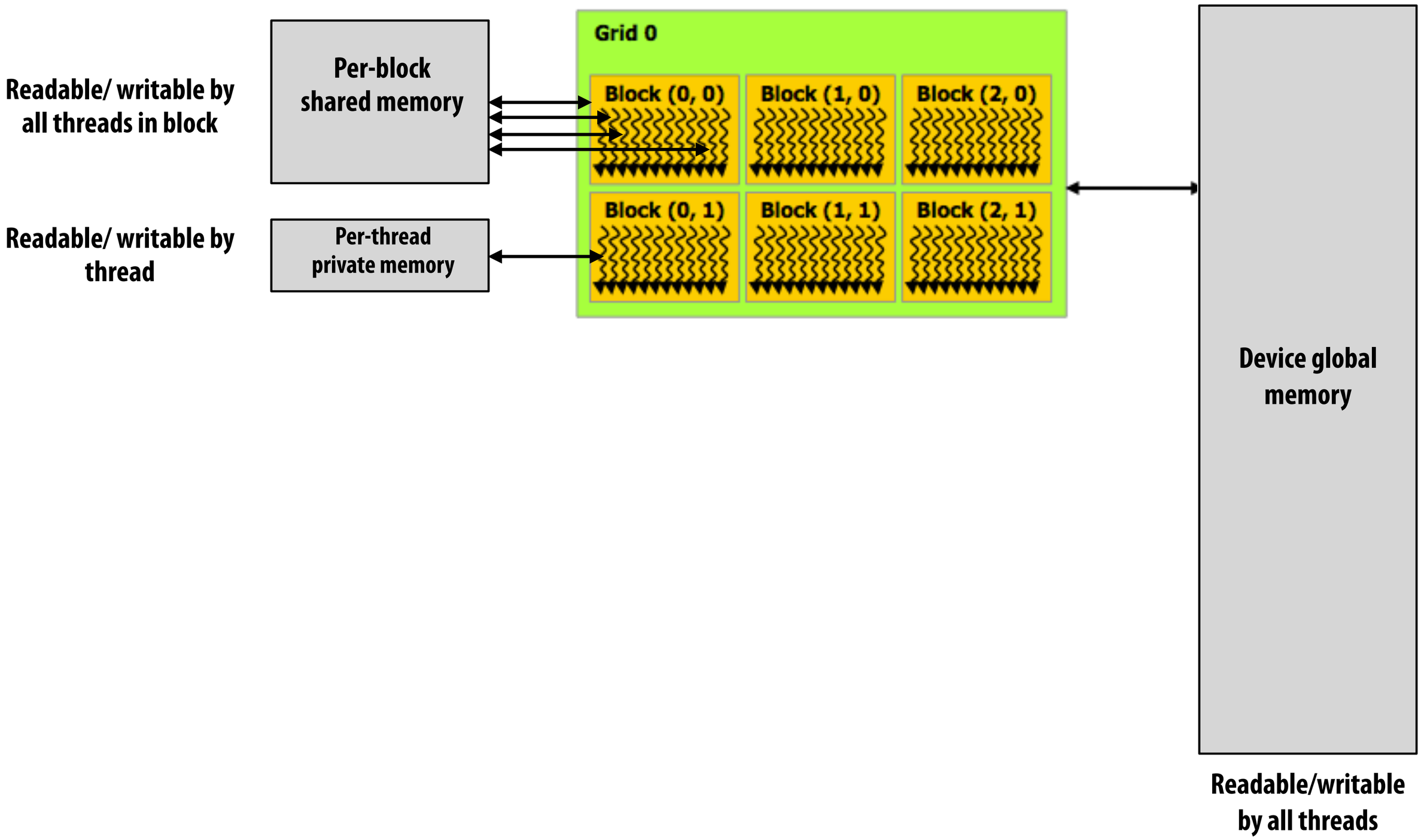



```
float* A = new float[N];       // allocate buffer in host mem

// 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.)
```

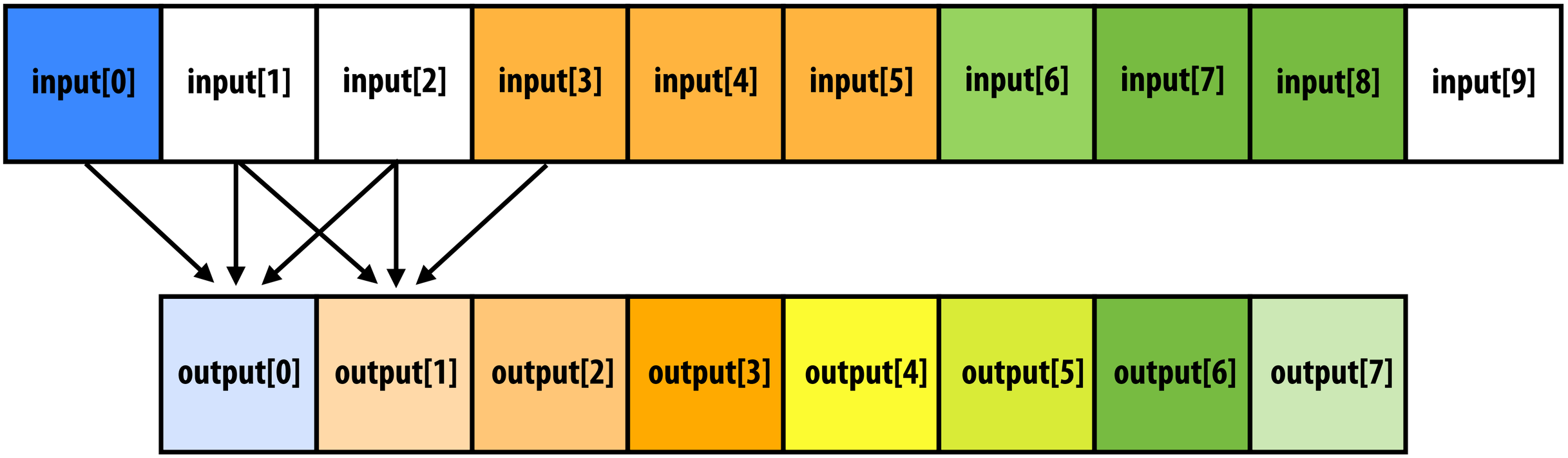
What does `cudaMemcpy` remind you of?

CUDA device memory model

Three distinct types of memory visible to device-side code



Example: 1D convolution

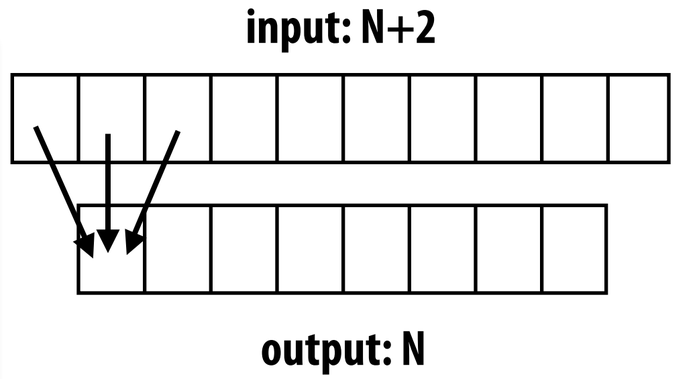



```
output[i] = (input[i] + input[i+1] + input[i+2]) / 3.f;
```

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];        // 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;
}

// 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);
```



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 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

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 each type 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];  // 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 / 3.f;
}

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

Consider implementation of call to pthread_create():

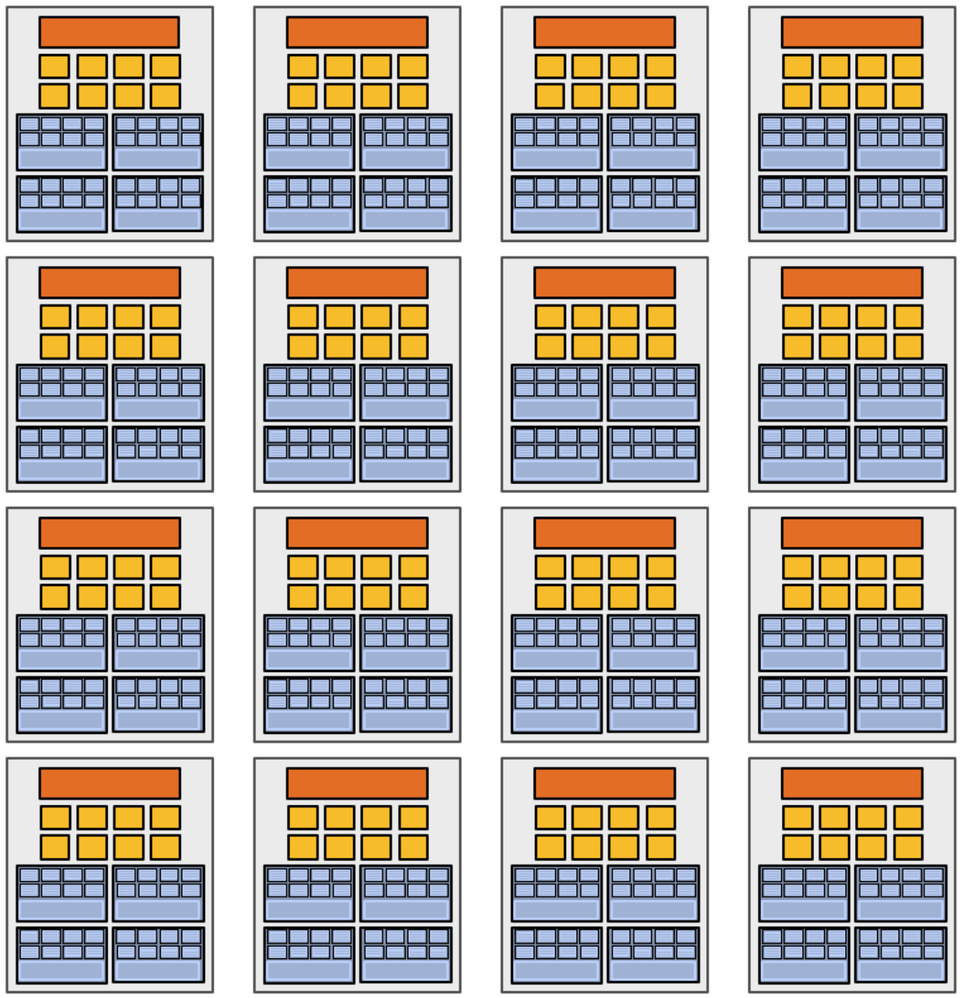
Allocate thread state:

- **Stack space for thread**
- **Allocate control block so OS can schedule thread**

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

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

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: no concept of `num_cores` in the CUDA programs I have shown: CUDA thread launch is 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];  // 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;
}

// 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);
```

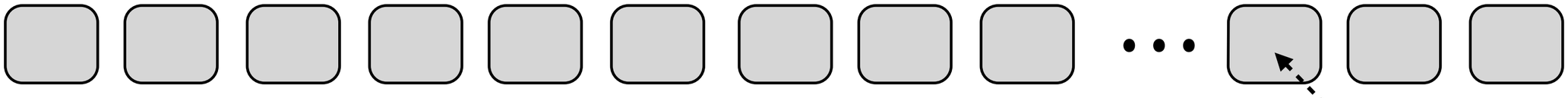
A compiled CUDA device binary includes:

Program text (instructions)
Information about required resources:

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

launch over 8K thread blocks

CUDA thread-block assignment

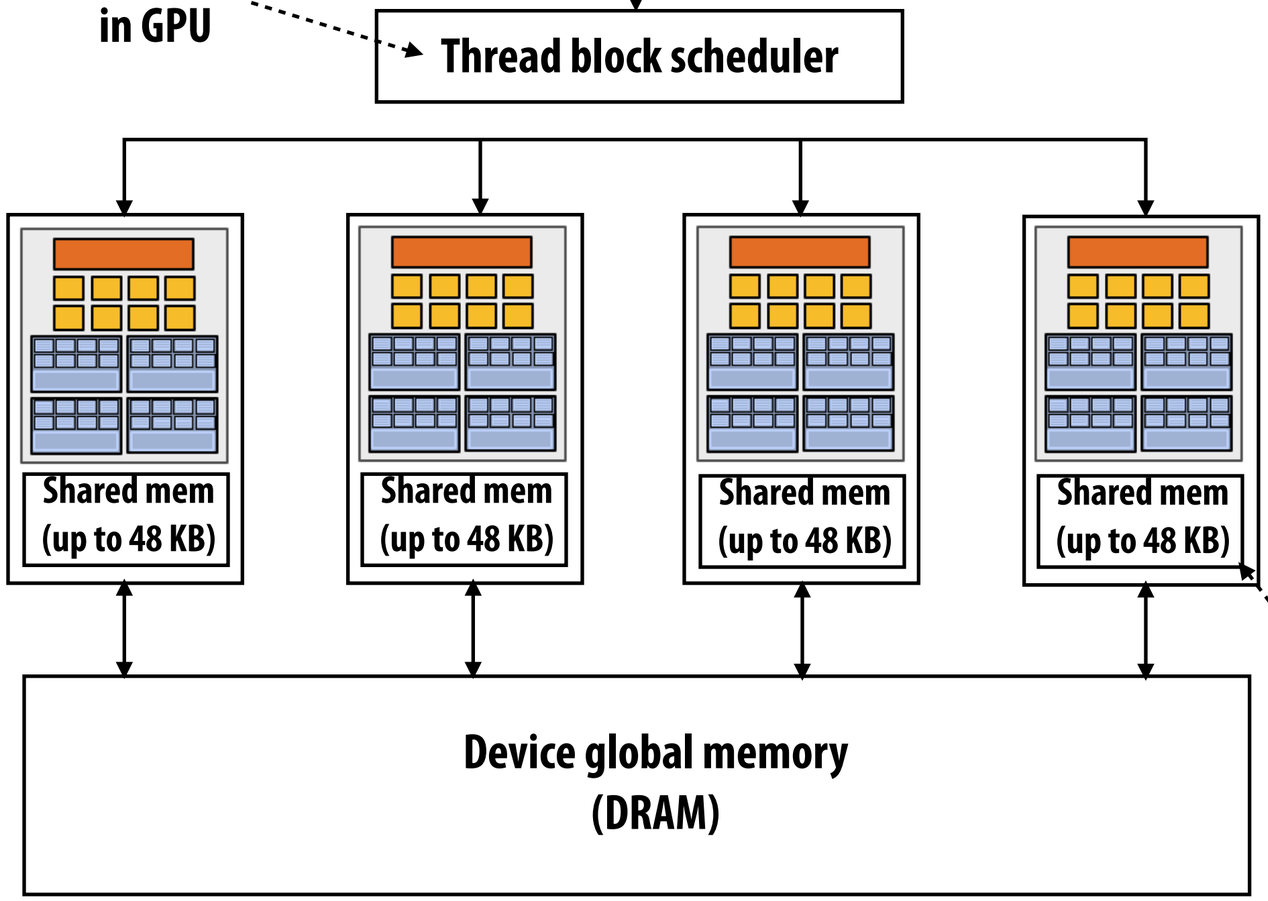


Grid of 8K `convolve` thread blocks (specified by kernel launch)

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

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

Special HW in GPU

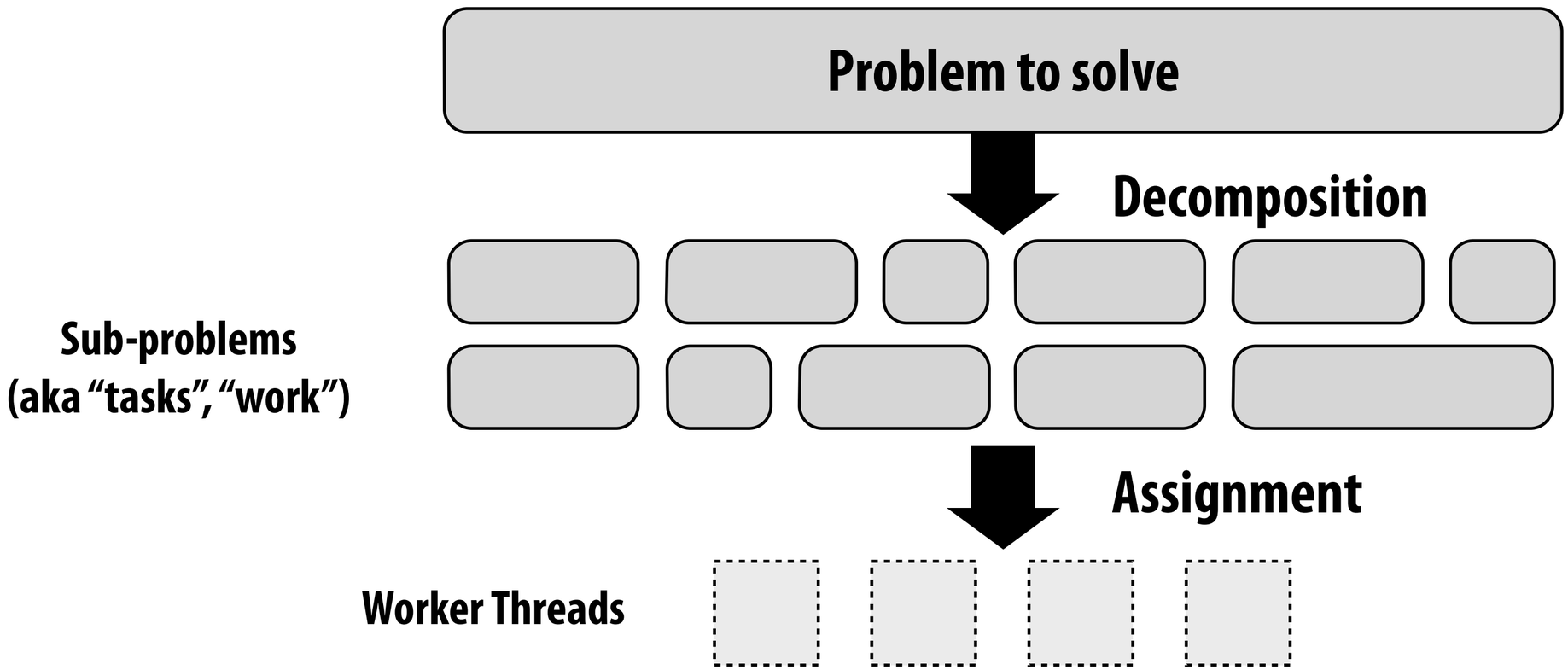


Shared mem is fast on-chip memory

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

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

Common design pattern: pool of worker "threads"



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

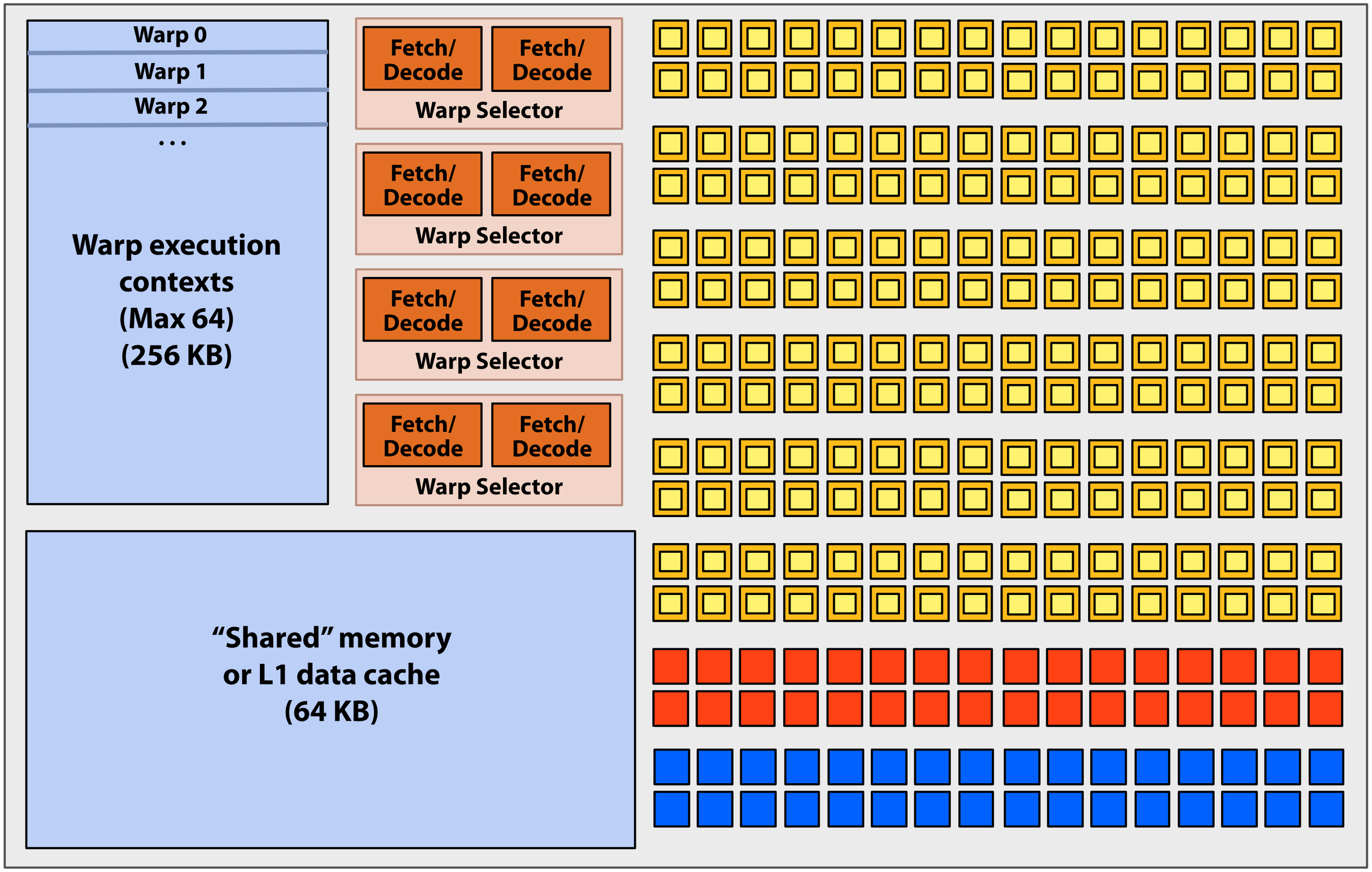
- One worker per parallel execution resource (e.g., CPU core, core execution context)
- May want N workers per core (where N is large enough to hide memory/IO latency)
- Pre-allocate resources for each worker
- Dynamically assign tasks to worker threads. (reuse allocation for many tasks)

Examples:

- ISPC's implementation of launching tasks
  - Creates one pthread for each hyper-thread on CPU. Threads kept alive for remainder of program
- 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

NVIDIA GTX 680 (2012)

This is one NVIDIA Kepler GK104 architecture SMX unit (one "core")



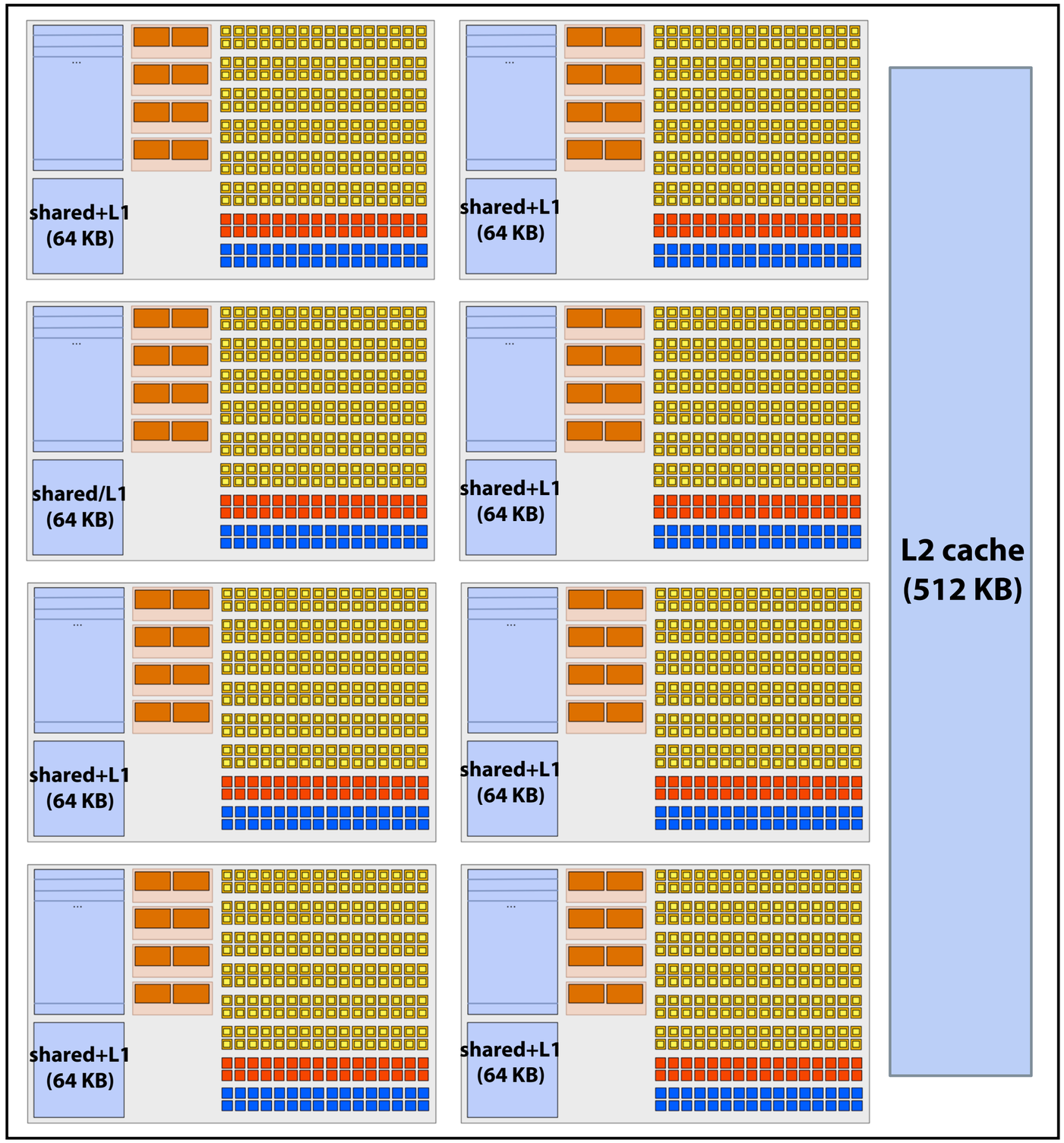
= SIMD functional unit, control shared across 32 units (1 MUL-ADD per clock)

= "special" SIMD functional unit, control shared across 32 units (operations like sin/cos)

= SIMD load/store unit (handles warp loads/stores, gathers/scatters)

NVIDIA GTX 680 (2012)

NVIDIA Kepler GK104 architecture



- 1 GHz clock
- Eight SMX cores per chip
- 8 x 192 = 1,536 SIMD mul-add ALUs
  = 3 TFLOPs
- Up to 512 interleaved warps per chip
  (16,384 CUDA threads/chip)
- TDP: 195 watts

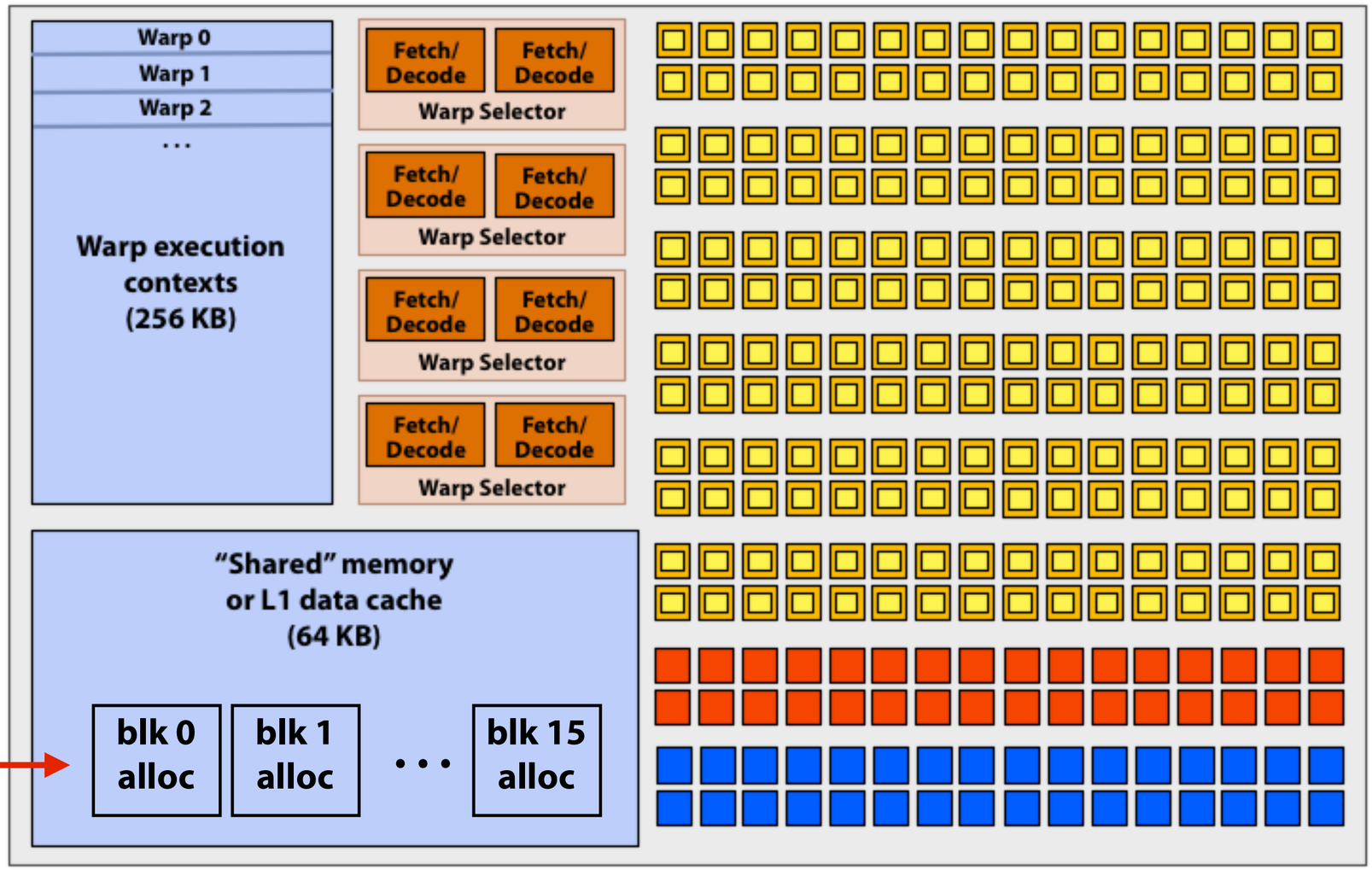
192 GB/sec

Memory

256 bit interface
DDR5 DRAM

Assigning CUDA thread blocks to GPU cores

NVIDIA Kepler GK104 architecture SMX unit (one "core")

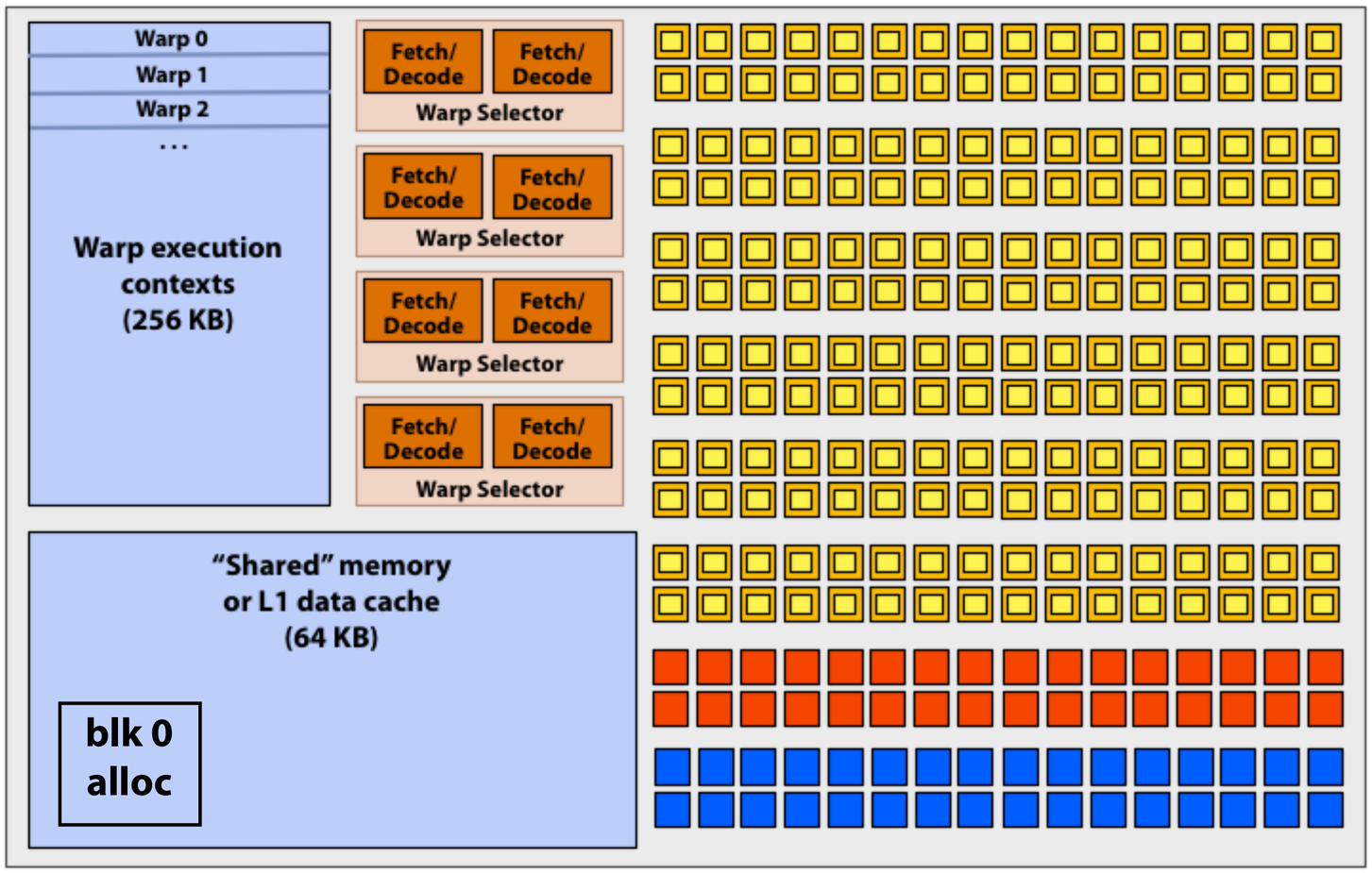


- SMX core resource limits:
  - Max warp exec contexts: 64 (2,048 total CUDA threads)
  - Up to 48 KB of shared memory available in L1

In the case of `convolve()`:

- Thread count constrains the number of blocks that can be scheduled on a core at once: 2048 / 128 = 16 blocks (64 warps)
- At CUDA kernel launch: semi-statically partition core into 16 thread block contexts = 2048 CUDA thread contexts (allocate these up front, at launch)
- Think of these contexts as the "workers"
  - Note: number of workers is chip resource dependent. Number of logical CUDA threads is NOT)
- Reuse context resources for many blocks. GPU scheduler assigns blocks dynamically assigned to core over duration of kernel computation!

Assigning CUDA threads to core execution resources

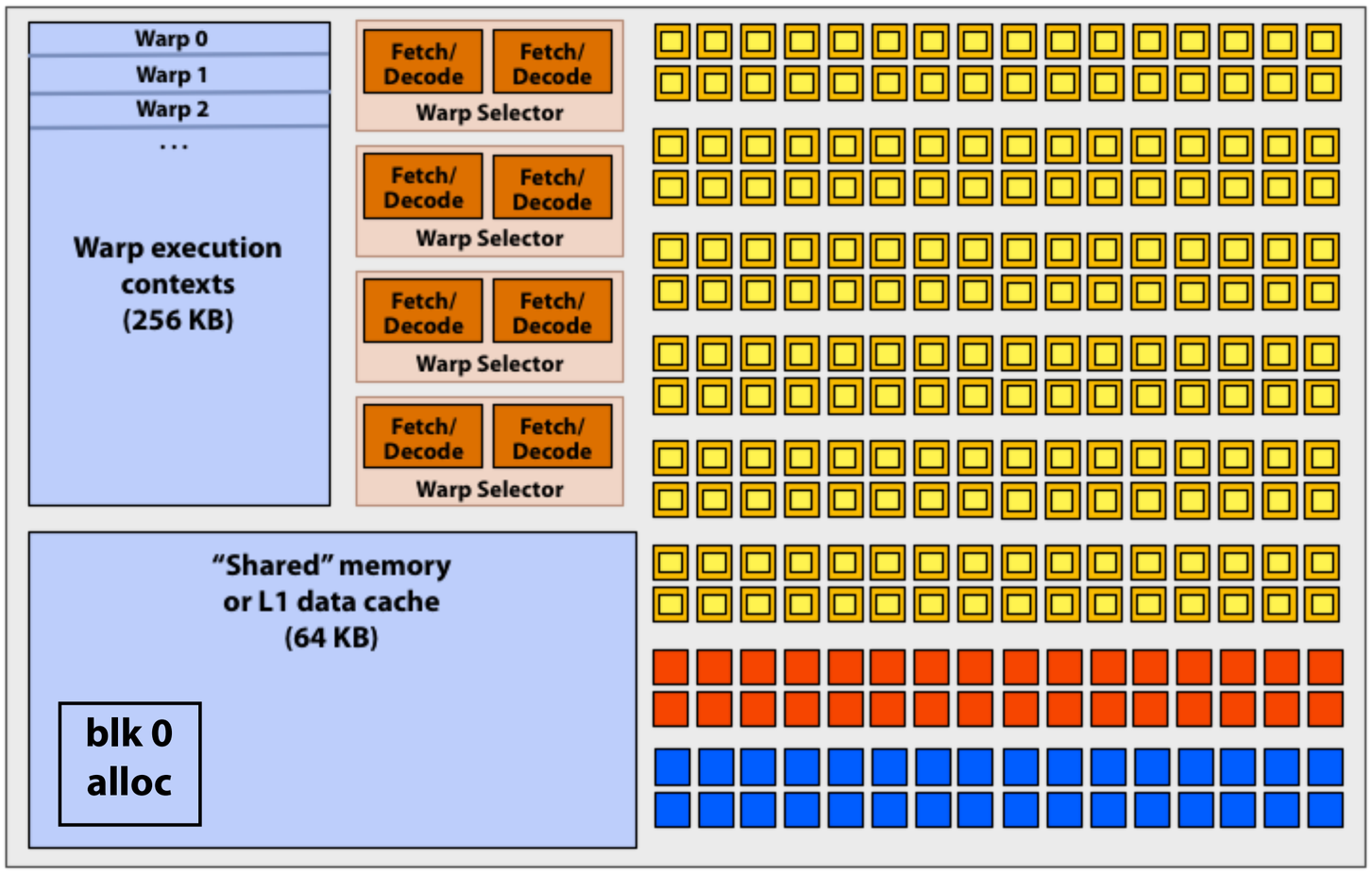



```
#define THREADS_PER_BLK 128

__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 thread block has been assigned to core

How do we execute the logic for the block?

Warps: groups of threads sharing an 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;
}
```

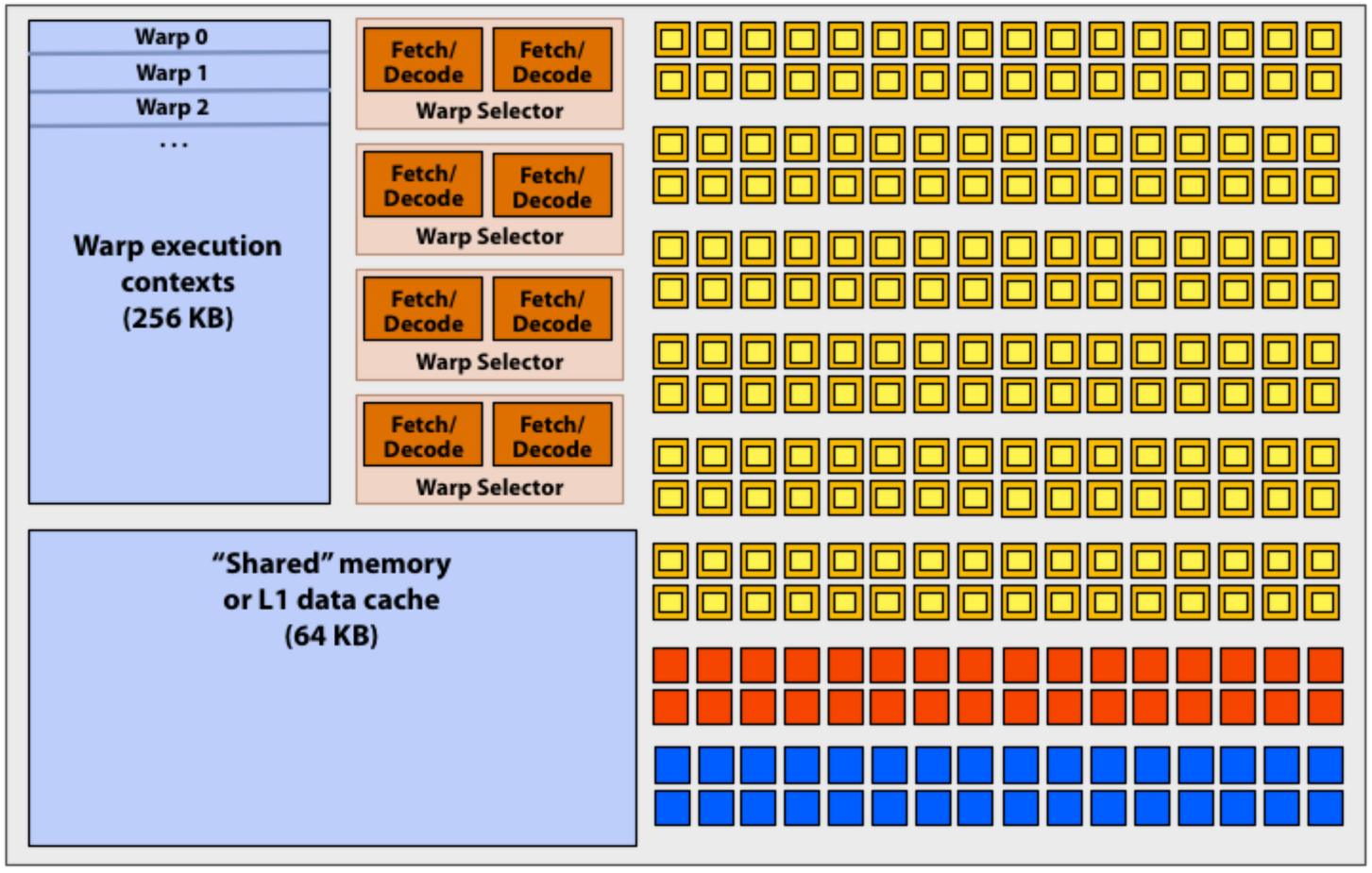
CUDA kernels execute as SPMD programs

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 * 32 threads/warp = 128 CUDA threads per block)

(Note: warps are an important implementation detail, but not a CUDA abstraction)

Executing warps on GTX 680



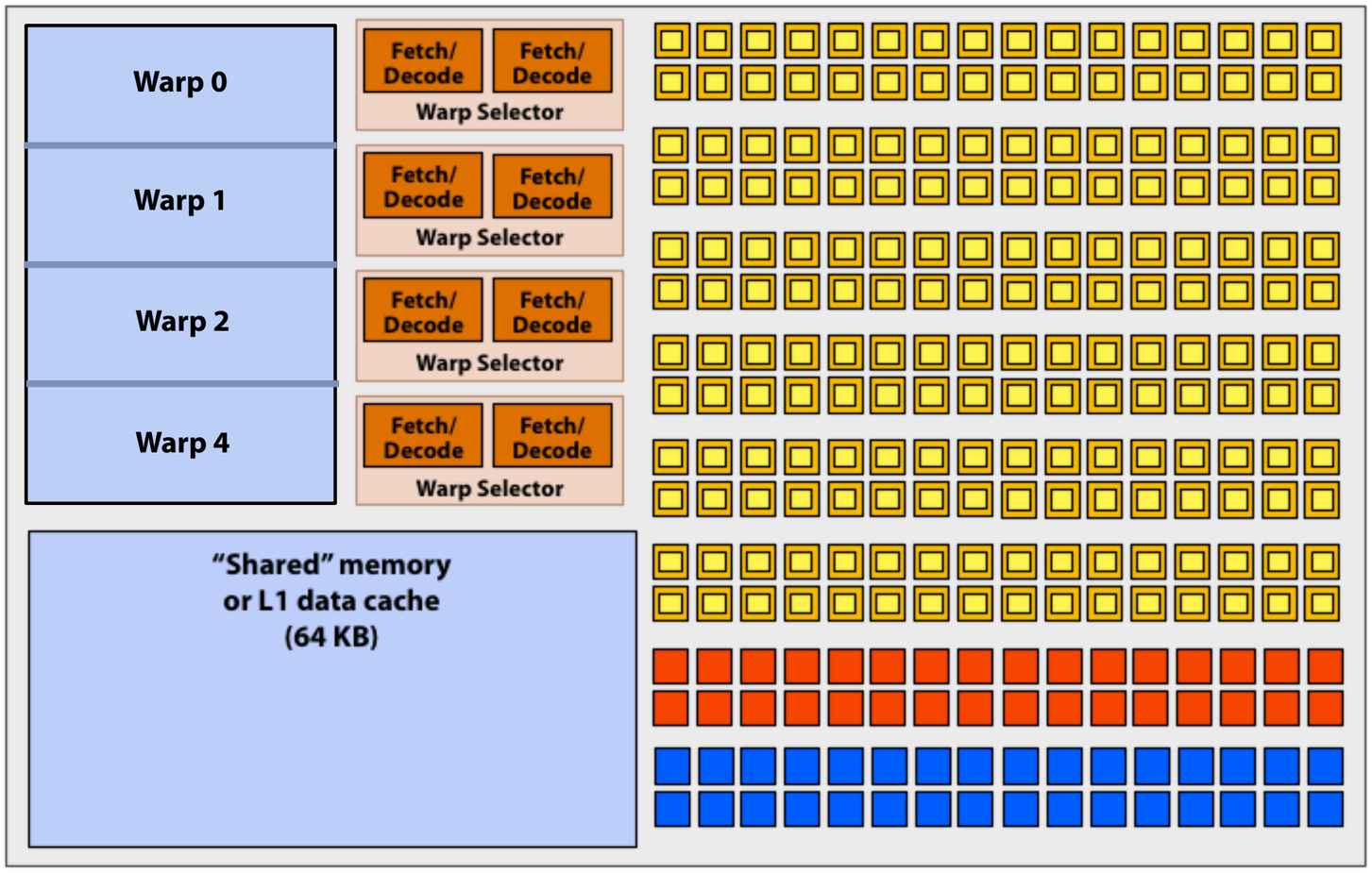
= SIMD functional unit,
control shared across 32 units
(1 MUL-ADD per clock)

= "special" SIMD functional unit,
control shared across 32 units
(operations like sin/cos)

= SIMD load/store unit
(handles warp loads/stores, gathers/scatters)

- SMX core operation each clock:
  - Select up to four runnable warps from up to 64 resident on core (thread-level parallelism)
  - Select up to two runnable instructions per warp (instruction-level parallelism)
  - Execute instructions on available groups of SIMD ALUs, special-function ALUs, or LD/ST units

Why allocate execution contexts for all threads in block?




```
#define THREADS_PER_BLK 256

__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;
}
```

Imagine a thread block with 256 CUDA threads (see code top-right)

Assume a fictitious SMX core with only 4 warps worth of parallel execution in HW (illustrated above)

Why not just run four warps (threads 0-127) to completion then run next four warps (threads 128-255) to completion in order to execute the entire thread block?

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 between blocks
  - Logically concurrent
  - A lot like ISPC tasks, right?

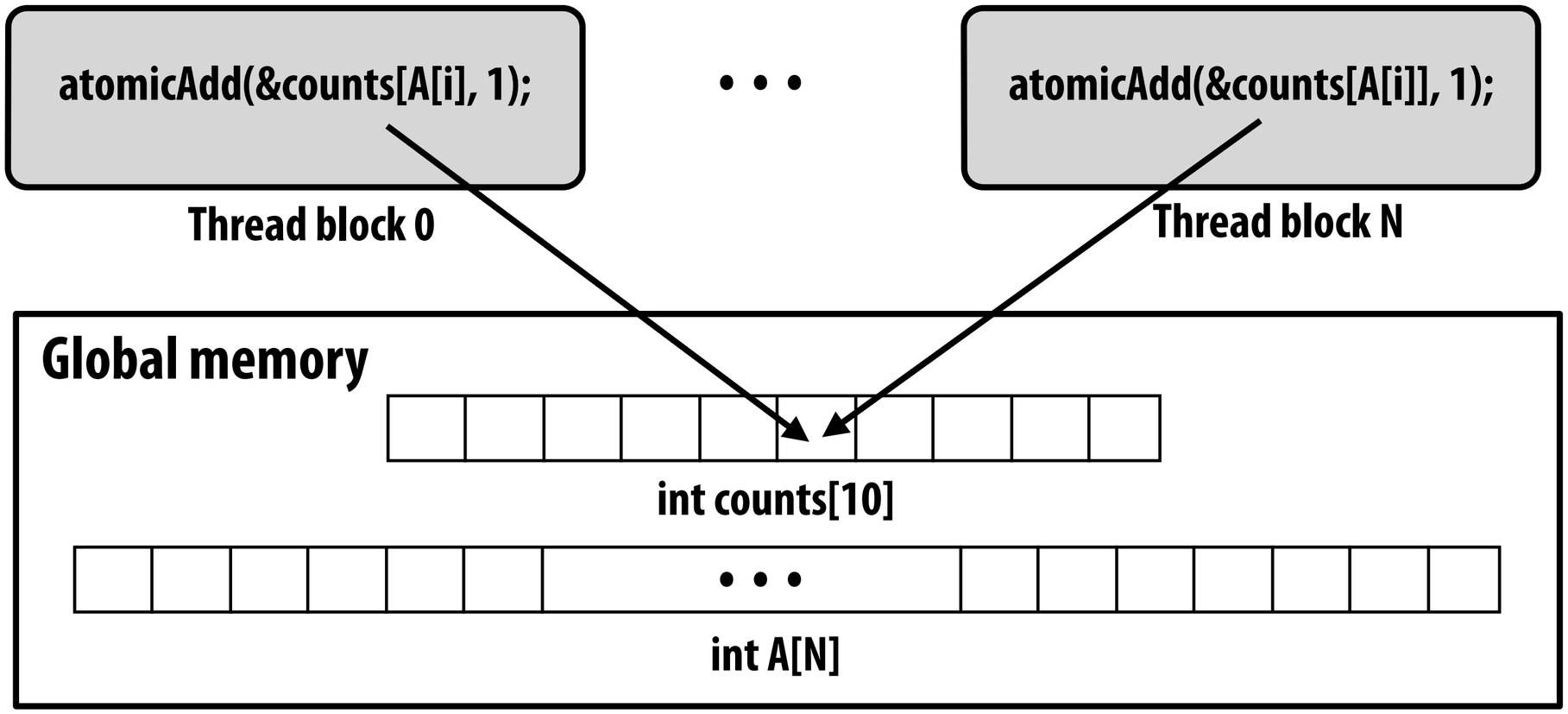
- **CUDA threads in same block DO run at the same time**
  - When block begins executing, all threads are running
    (these semantics impose a scheduling constraint on the system)
  - A CUDA thread block is itself an SPMD program (like an ISPC gang of program instances)
  - Threads in thread-block are concurrent, cooperating "workers"

- **CUDA implementation:**
  - A Kepler GPU warp has performance characteristics akin to an ISPC gang of instances (but unlike an ISPC gang, the warp concept does not exist 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 (shared memory allocations, warp execution contexts) become available for next block

* Exceptions to this statement include intra-warp builtin operations like swizzle and vote

This program creates a histogram. It is reasonable CUDA code?

- This example: build a histogram of values in an array
  - 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 use of atomics does not impact implementation's ability to schedule blocks in any order (atomics used for mutual exclusion, and nothing more)



But is this reasonable CUDA code?

- Consider implementation of 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)

"Persistent thread" CUDA programming style

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

__device__ int workCounter = 0;  // global mem variable

__global__ void convolve(int N, float* input, float* output) {
  __shared__ int startingIndex;
  __shared__ float support[THREADS_PER_BLK+2];  // shared across block
  while (1) {

     if (threadIdx.x == 0)
        startingIndex = atomicInc(workCounter, THREADS_PER_BLK);
     __syncthreads();
     if (startingIndex >= N)
        break;

     int index = startingIndex + 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
// properly initialize contents of devInput here ...

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

Idea: write CUDA code that requires knowledge of the number of cores and blocks per core that are supported by underlying GPU implementation.

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

(Program makes assumptions about GPU implementation: that GPU will in fact run all blocks concurrently. Ug!)

Now, work assignment to blocks is implemented entirely by the application

(circumvents GPU thread block scheduler)

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

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