Lecture 8:

Compute-mode GPU Programming Interfaces

Visual Computing Systems
CMU 15-869, Fall 2014

What is a programming model?

Programming models impose structure

- A programming model provides a set of primitives/abstractions that impose structure on programs written using it
  - The structure captures the salient features of this class of programs
  - "Here's how a programmer should think of problems of this type…"
  - Powerful/efficient: write less code, system performs key optimizations

- Analogy: in data analysis, what does choosing the right model for a data set entail?
  - Fits the structure of the data (the phenomenon) being described well
  - Powerful/efficient: few parameters left up to user
  - Generalizes to describe new data/phenomenon of similar types

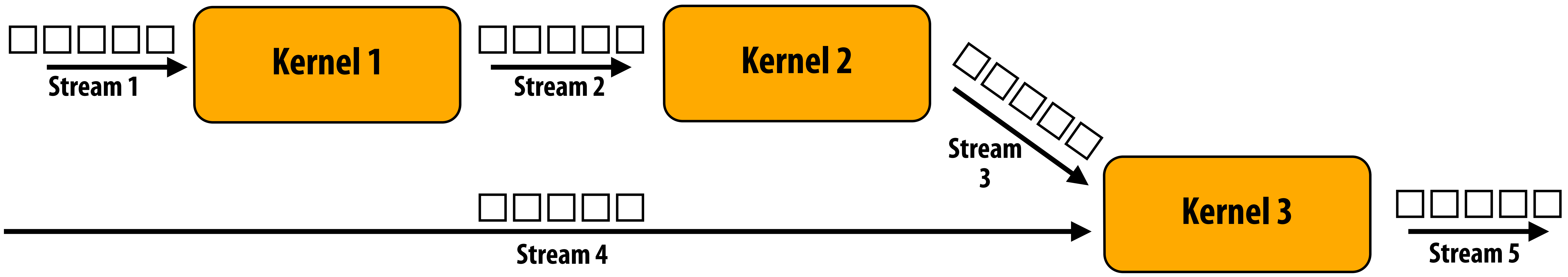
THE QUESTION to ask yourself when trying to assess the value of a programming model

- **What does imposing this particular structure do for the programmer?**

- **In other words: what services does the system provide as a result of the structure?**
  - **Is certain boilerplate code (or difficult to implement algorithms) provided in a convenient library or primitive?**
  - **Is a certain part of the code parallelized automatically?**
  - **Is the code mapped to a certain type of specialized hardware?**

Stream programming model

Emits programs structured as a series of kernels operating on elements of data streams



- Streams
  - Encapsulate per-element independence (every element can be processed in parallel)
  - Encapsulate producer-consumer locality

- Kernels
  - Side-effect-free functions operating on stream elements
  - Encapsulate locality (kernel's working set defined by inputs, outputs, and temporaries)
  - Encapsulate instruction-stream coherence (same kernel applied to each stream element)

- Many implementations (e.g., StreaMIT, StreamC/KernelC, SDF) rely on static scheduling by the compiler to achieve very high performance

Stream programming model: kernels

No loops, array indexing, or <u>explicit parallelism</u> in code.

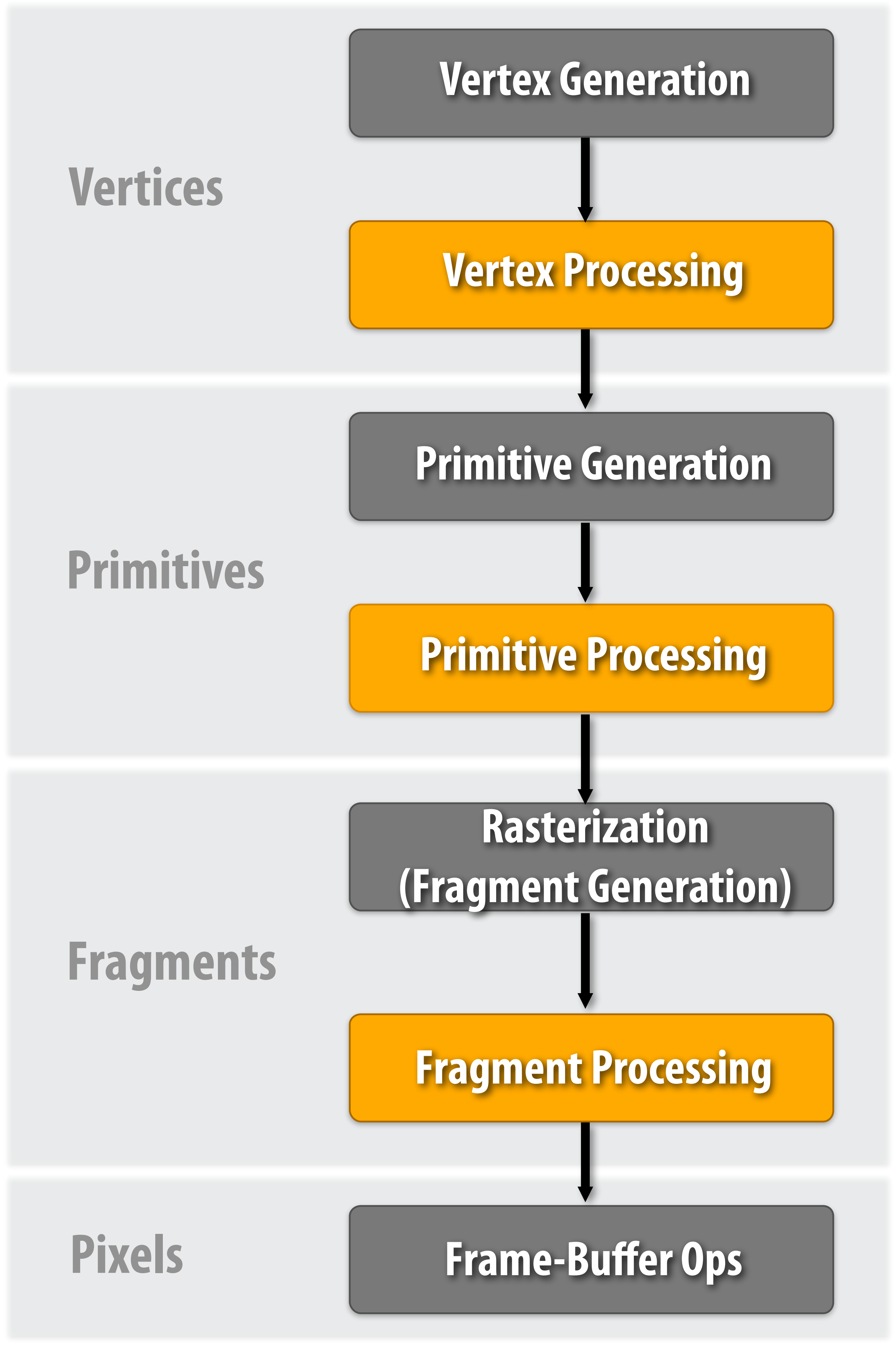
```
kernel void scale(float amount, float a, out float b)
{
   b = amount * a;
}

// note: omitting initialization
float scale_amount;
stream<float> input_stream(1000);
stream<float> intermediate_stream;
stream<float> output_stream(1000);

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

Semantics of kernel calls are to invoke kernel once per output stream element

Graphics pipeline has many streaming aspects

- Streams: elements between pipeline stages (e.g., implemented using statically allocated on-chip buffers)
- Kernels: pipeline stages (implemented by fixed-function logic or shader code)
- What are aspects of the graphics pipeline that are NOT streaming in nature?
  - Texture fetch (unpredictable array access)
  - Frame-buffer update (not independent)

Basic Graphics Pipeline

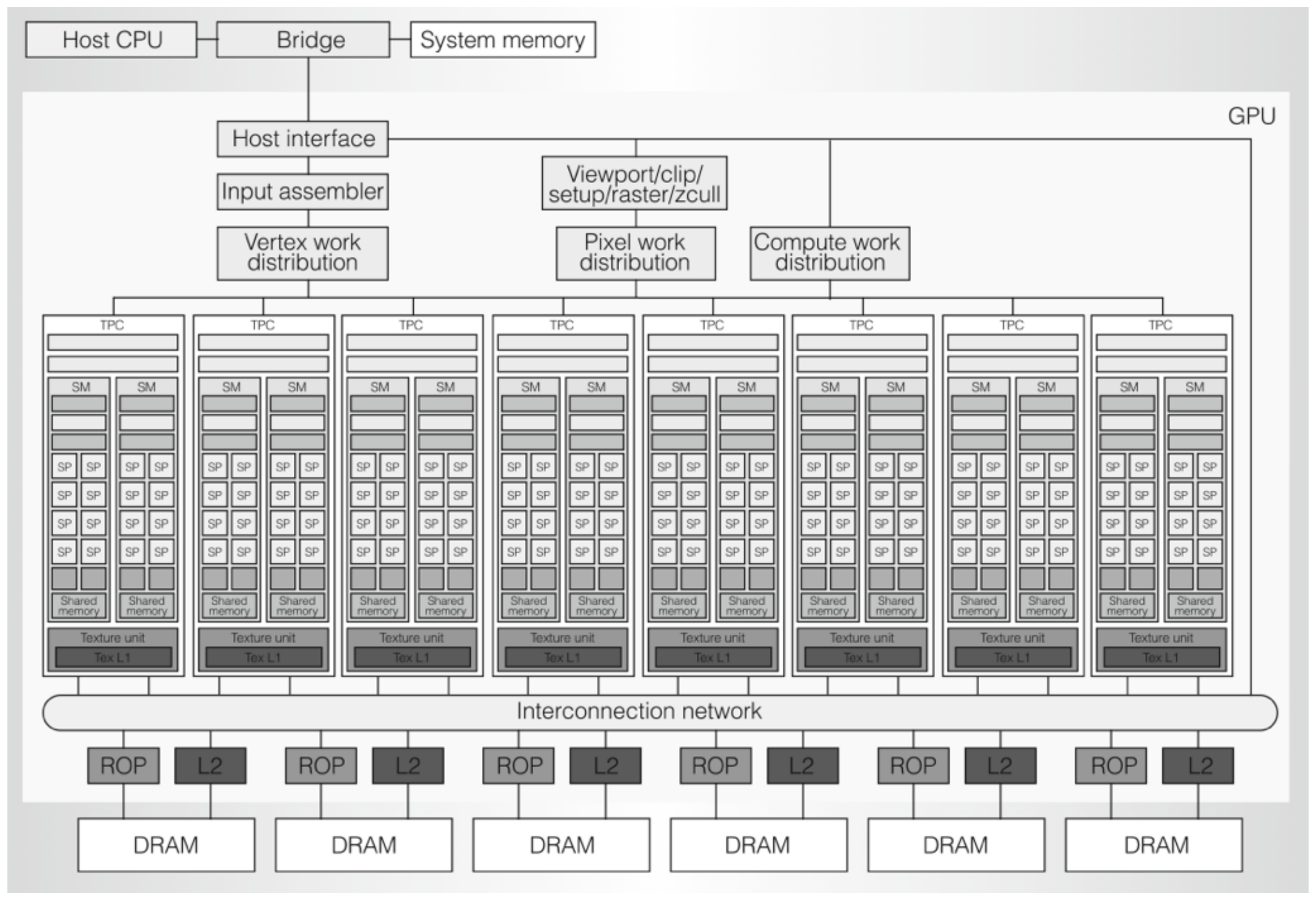


Research question in early 2000's

- What programming models can make is easy to write applications that efficiently use emerging high-throughput parallel processors.
  - Topic of computer architecture, supercomputing, programming language communities (stream programming was a major focus)
  - Note: still a Ph.D. level research question in 2014
- Real-time graphics folks:
  - Even if you ignore all the fixed-function stuff in a GPU, the programmable cores in a GPU are a very high performance parallel processor (that's cheap and widely available today!)
  - It's unfortunate that the only way to use these processors is to rasterize triangles (recall: fixed-function units control programmable units of GPU)

NVIDIA's CUDA [Designed by Ian Buck at NVIDIA, circa 2007]

- **Alternative architecture definition for Tesla-class GPU hardware)**
  - "Compute mode" interface
  - Tesla was first "unified shading" GPU



- **Low-level abstraction that reflects capabilities of hardware**
  - Recall arguments in Cg paper: do not have abstraction get in the way of using hardware, even if it makes it more tedious to write code)
  - Combines some elements of streaming and multi-threading (like HW does)

- **Open standards embodiment of this programming model is OpenCL (Microsoft's embodiment is D3D Compute Shader)**

CUDA constructs (the “kernel”)

```
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
   int i = threadIdx.x;   // CUDA builtin: get thread id
   b[i] = amount * a[i];
}

// note: omitting array initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;

// launch N threads, each thread executes kernel ‘scale’
scale<<1,N>>(scale_amount, input_array, output_array);
```

Bulk thread launch: logically spawns N threads

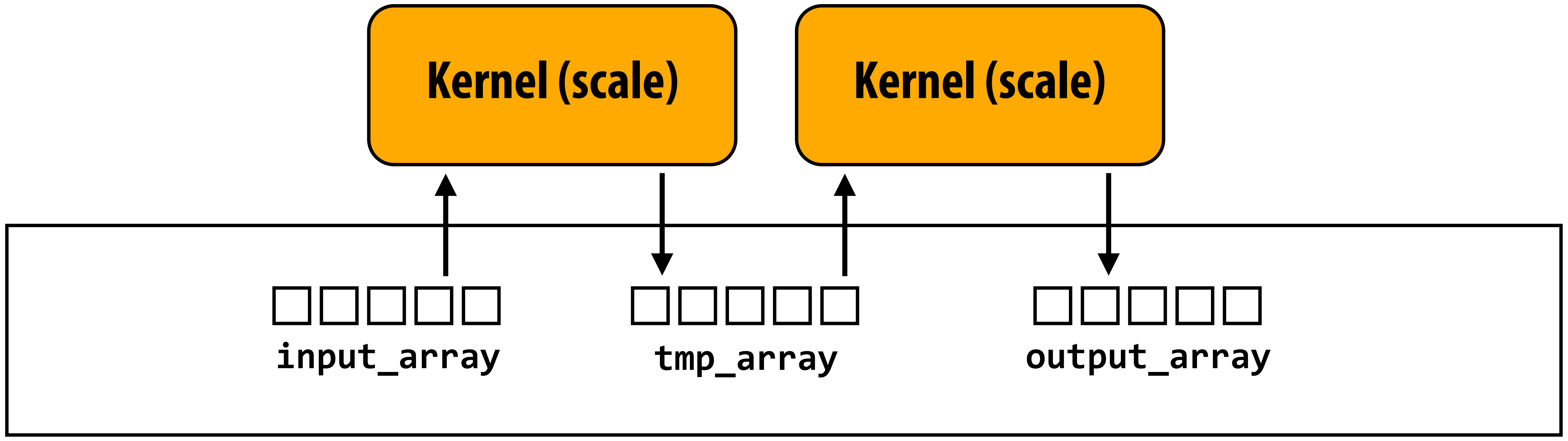
What is the behavior of this kernel?

```
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
   int i = threadIdx.x;   // CUDA builtin: get thread id
   b[0] = amount * a[i];
}

// note: omitting array initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;

// launch N threads, each thread executes kernel 'scale'
scale<<1,N>>(scale_amount, input_array, output_array);
```

Bulk thread launch: logically spawns N threads

Can system discover producer-consumer locality?

```
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
   int i = threadIdx.x;   // CUDA builtin: get thread id
   b[i] = amount * a[i];
}

// note: omitting array initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;
float* tmp_array;

scale<<1,N>>(scale_amount, input_array, tmp_array);
scale<<1,N>>(scale_amount, tmp_array, output_array);
```



CUDA constructs (the kernel)

```
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
   int i = threadIdx.x;   // CUDA builtin: get thread id
   b[i] = amount * a[i];
}

// note: omitting array initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;

// launch N threads, each thread executes kernel ‘scale’
scale<<1,N>>(scale_amount, input_array, output_array);
```

Bulk thread launch: logically spawns N threads

Question: What should N be?

Question: Do you normally think of “threads” this way?

CUDA constructs (the kernel)

```
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
   int i = threadIdx.x;   // CUDA builtin: get thread id
   b[i] = amount * a[i];
}

// note: omitting array initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;

// launch N threads, each thread executes kernel 'scale'
scale<<1,N>>(scale_amount, input_array, output_array);
```

Given this implementation: each invocation of `scale` kernel is independent.

(bulk thread launch semantics no different than sequential semantics)

CUDA system has flexibility to parallelize any way it pleases.

In many cases, such as in the example above, thinking about a CUDA kernel as a stream processing kernel, and CUDA arrays as streams is perfectly reasonable, ALTHOUGH THIS STRUCTURE IS NOT IMPOSED BY THE CUDA PROGRAMMING MODEL.

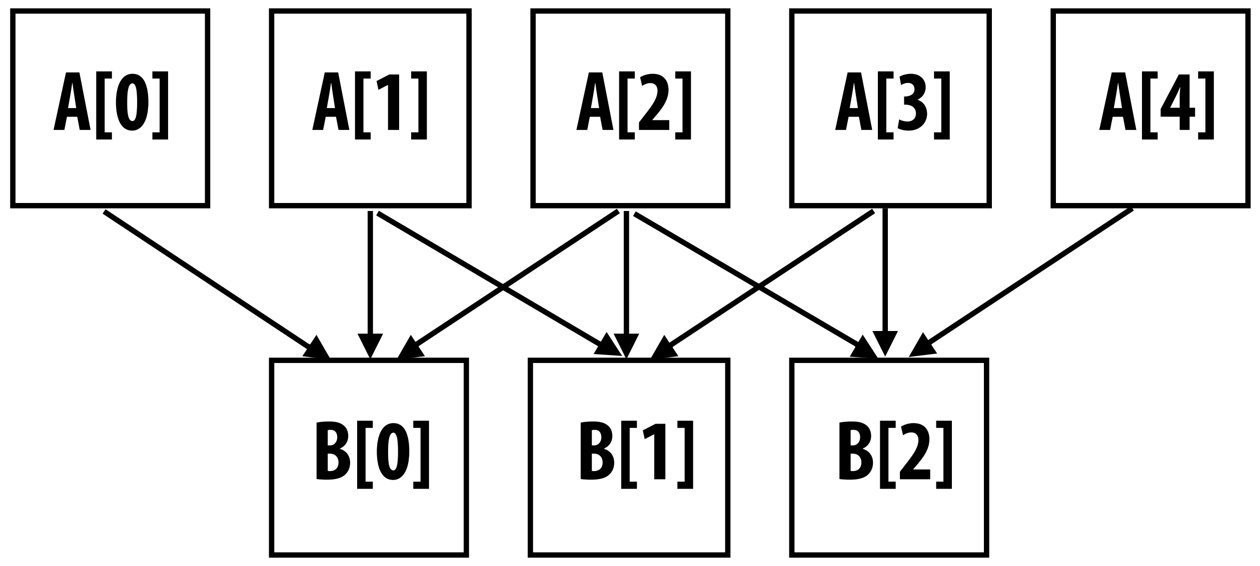
(Programmer just has to do a little indexing in the kernel to get a reference to stream inputs/outputs)

Convolution example

```
// assume len(A) = len(B) + 2
__global__ void convolve(float* a, float* b)
{
   int i = threadIdx.x;
   b[i] = a[i] + a[i+1] + a[i+2];
}
```

Note "adjacent" threads load same data.

Here: 3x input reuse **(reuse increases with increasing width of convolution filter)**



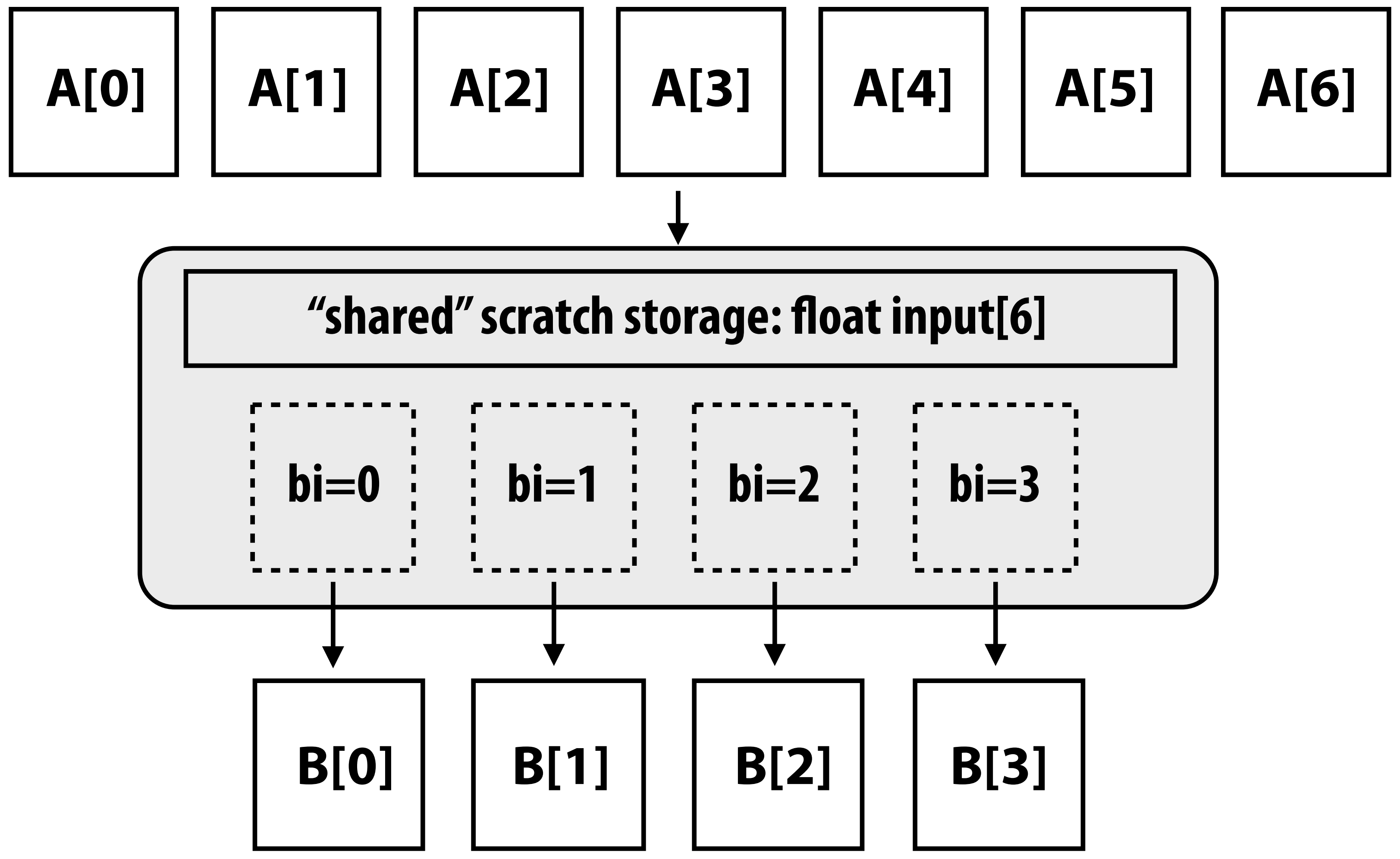
CUDA thread hierarchy

```
#define BLOCK_SIZE 4

__global__ void convolve(float* a, float* b)
{
   __shared__ float input[BLOCK_SIZE + 2];

   int bi = blockIdx.x;
   int ti = threadIdx.x;

   input[bi] = A[ti];
   if (bi < 2)
   {
      input[BLOCK_SIZE+bi] = A[ti+BLOCK_SIZE];
   }

   __syncthreads();   // barrier

   b[ti] = input[bi] + input[bi+1] + input[bi+2];
}

// allocation omitted
// assume len(A) = N+2, len(B)=N
float* A, *B;

convolve<<BLOCK_SIZE, N/BLOCK_SIZE>>(A, B);
```

CUDA threads are grouped into thread blocks

Threads in a block are <u>not</u> independent.
They can cooperate to process shared data.

1. **Threads communicate through `__shared__` variables**
2. **Threads barrier via `__syncthreads()`**



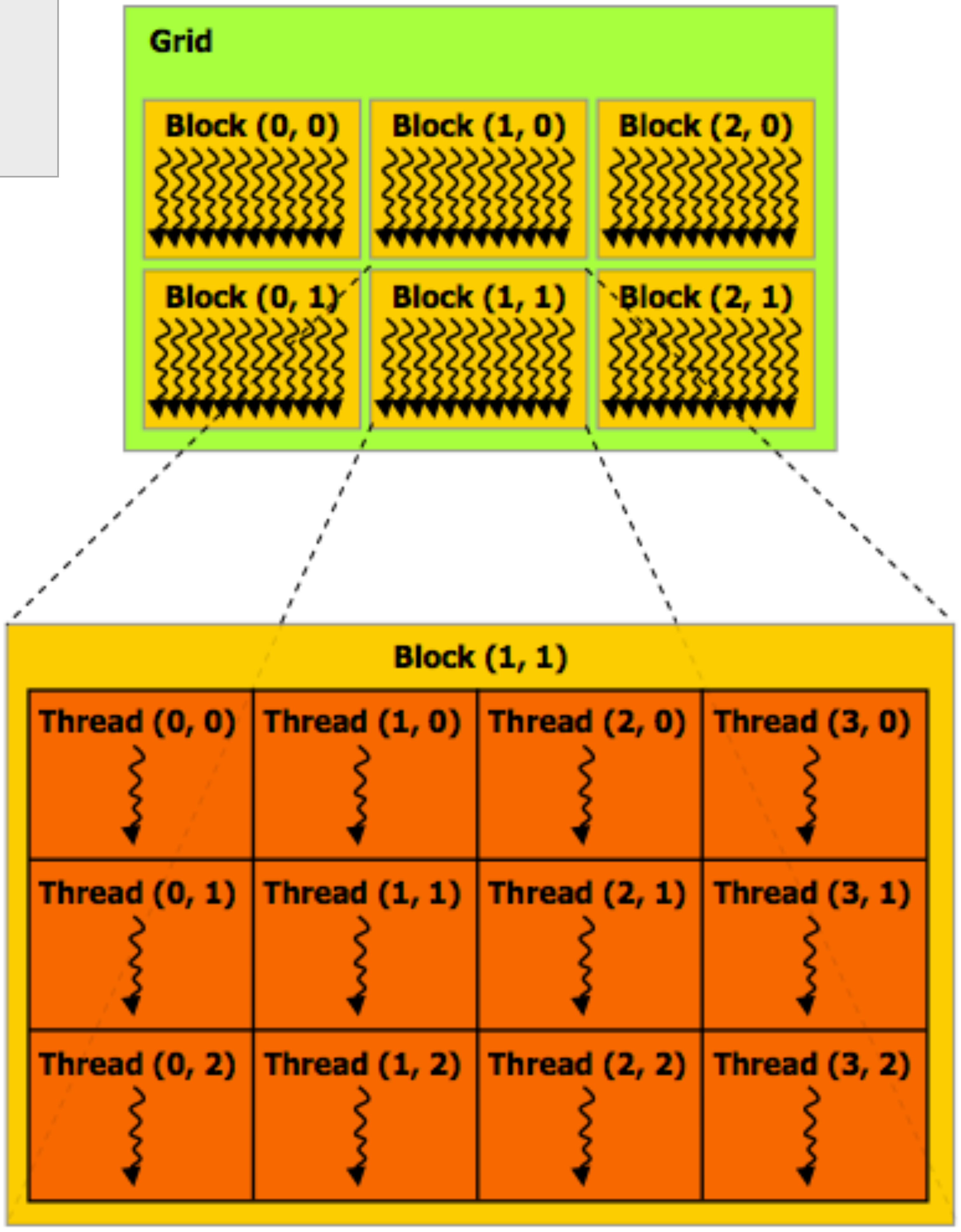
CUDA thread hierarchy

```
// this code will launch 96 threads
// 6 blocks of 16 threads each

dim2 threadsPerBlock(4,4);
dim2 blocks(3,2);
myKernel<<blocks, threadsPerBlock>>();
```

Thread blocks (and the overall "grid" of blocks) can be organized in 1D, 2D, or 3D arrangements of threads
(Convenience: many CUDA programs operate on n-D grids)

Thread blocks (often) represent independent execution.

Threads in a thread block executed <u>simultaneously</u> on same GPU core
- **Why on the same core?**
- **Why simultaneously?**



Source: CUDA Programming Manual

The common way to think about CUDA

(thread centric view of the programming model)

- **CUDA is a multi-threaded programming model**
- **Threads are logically grouped together into blocks and gang scheduled onto cores. This grouping is a locality hint.**
- **Threads in a block are allowed to synchronize and communicate through barriers and shared local memory**
- **Note: Lack of communication between threads in different blocks gives scheduler some flexibility (can "stream" blocks through the system)****

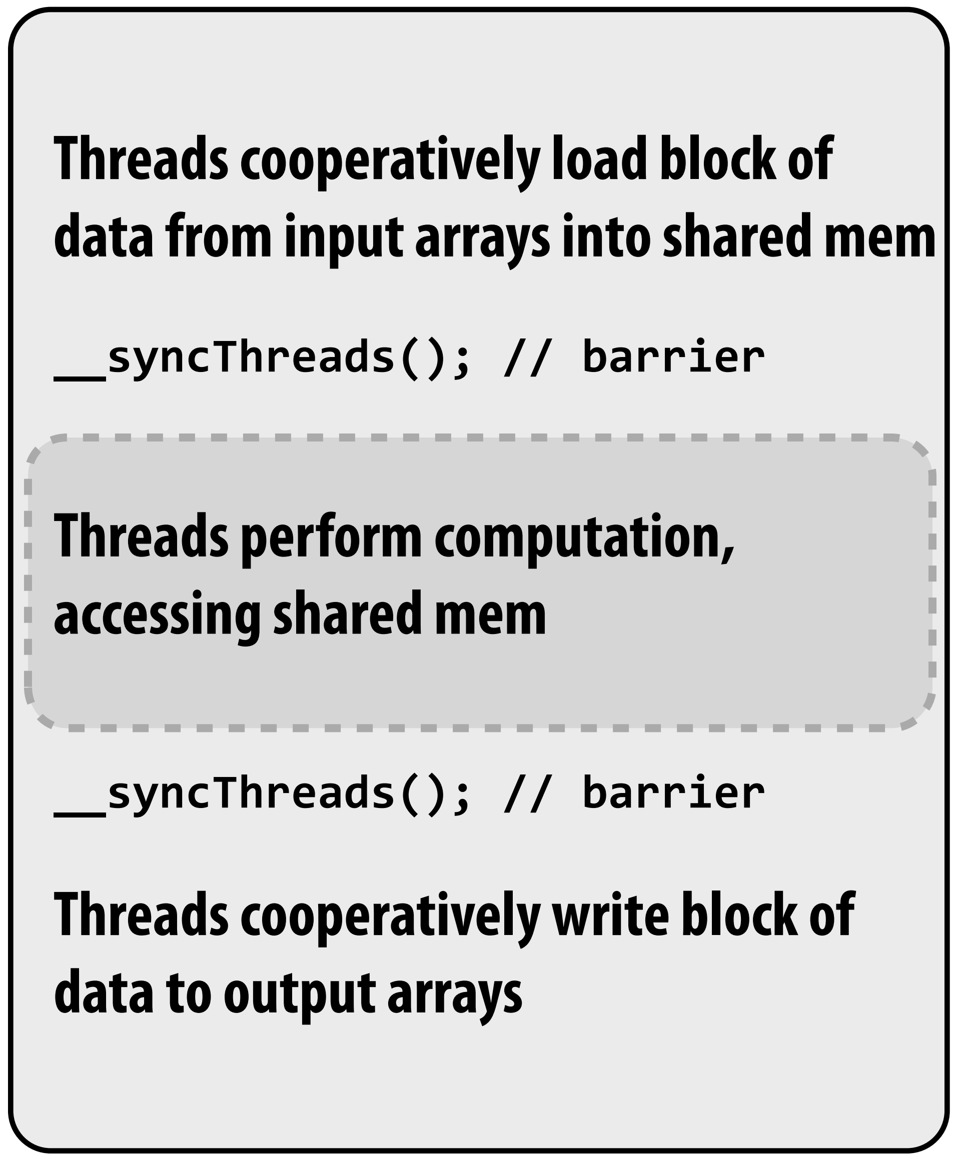
**** Global memory atomic operations provide a form of inter-thread block communication (more on this in a second)**

Another way to think about CUDA

(like a streaming system: stream of thread blocks view)

- CUDA is a stream programming model (recall Brook)
  - Kernels are CUDA thread blocks
  - Stream elements are now blocks of data accessed by kernel (larger working sets than just individual elements)
- Kernel invocations independent, but logic is multi-threaded
  - Multi-threading exposed additional fine-grained parallelism
- Think: implicitly parallel across thread blocks
- Think: explicitly parallel within a thread block (explicit synchronization of individual threads via barriers)

Canonical CUDA thread block program:



Threads cooperatively load block of data from input arrays into shared mem

```
__syncThreads(); // barrier
```

Threads perform computation, accessing shared mem

```
__syncThreads(); // barrier
```

Threads cooperatively write block of data to output arrays

Choosing thread-block sizes

Question: how many CUDA threads should be in a CUDA thread block?

Recall from GPU core lecture:

- **How many threads per core?**
- **How much shared local memory per core?**

Another CUDA programming style: “persistent” threads

- No attempt to maintain streaming structure at all any more
- Programmer is always “thinking” about explicitly parallel code, and writing code that is aware of the number of processors in the machine (very much like a pthread programmer)
- Threads use atomic global memory operations to cooperatively implement work assignment to thread blocks

```
// “Persistent thread” implementation: Run thread block until all work is done,
// processing multiple work elements, rather than just one per block. Thread block
// terminates when no more work is available

__global__ void persistent(int* head, int count, float* a, float* b)
{
    int index;
    while ( (index = read_and_increment(head, 1)) < count)
    {
         // load a[index];

         // do work

         // write result to b[index]
    }
}

// launch exactly enough thread blocks to fill up machine
// (to achieve sufficient parallelism and latency hiding)

int head = 0;
persistent<<numBlocks, blockSize>>(&head, total_count, A, B);
```

Questions:

What does CUDA system do for the programmer?

How does it compare to OpenGL?

Quick aside: why did CUDA become popular?

(Kayvon's personal opinion: ignoring marketing reasons)

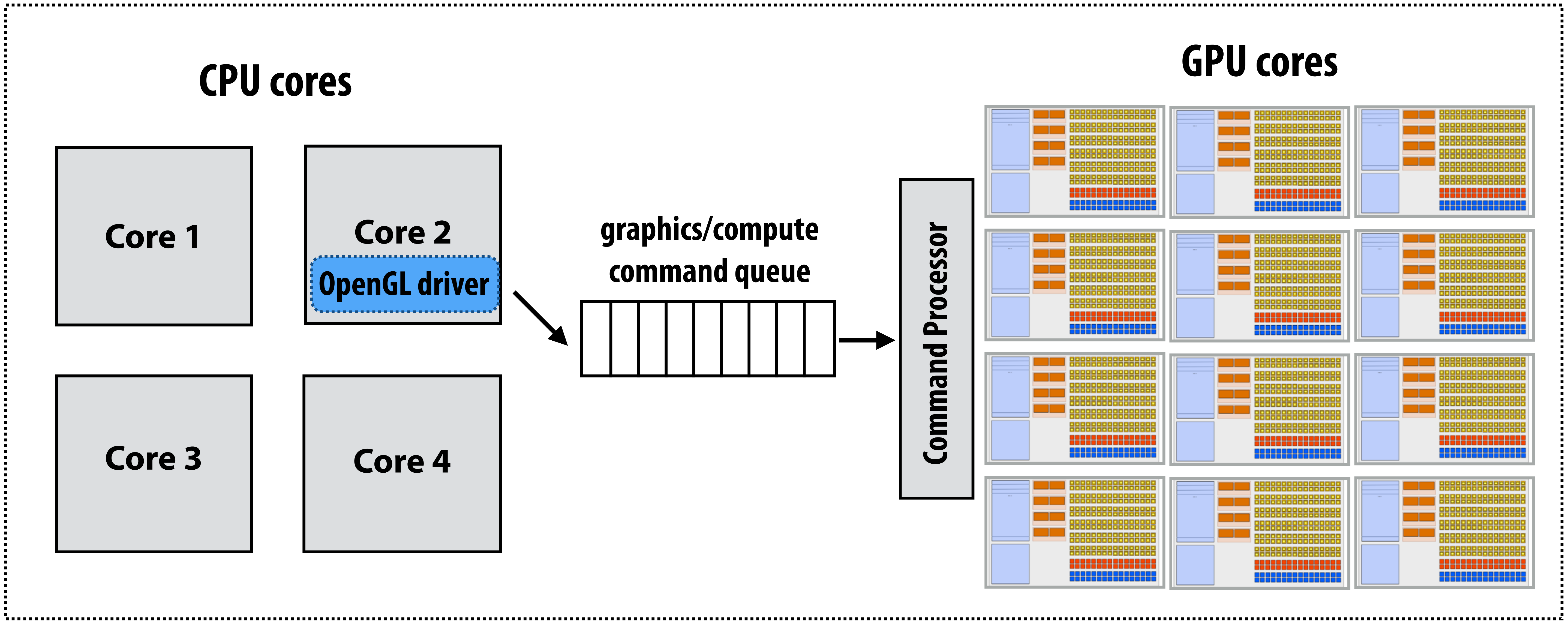
1. **Provides access to a cheap, fast machine (GPU's programmable cores and high-bandwidth memory system)**

2. **SPMD programming abstraction allows programmer to write scalar code, have it mapped to wide SIMD hardware**
   - CPU vendors were adamant about mapping arbitrary C/C++ code to vector instructions (not willing to constrain program structure to make problem easier)
   - Even today, generating explicit vector code for CPUs remain shockingly hard to do (OpenCL is exceptionally heavyweight, see Intel's ISPC for a useful tool)

3. **Programming model was familiar: much more like thread programming than stream programming — it allowed arbitrary in-kernel array indexing (because GPU's had hardware multi-threading to hide memory latency)**
   - More familiar, convenient, and flexible in comparison to previous data-parallel or streaming systems [StreamC/KernelC, StreamMIT, ZPL, Nesl, synchronous data-flow, and many others]
   - 1-to-1 with hardware behavior (HW had latency-hiding support already, no reliance on "sophisticated" compiler technology to achieve high performance)

Current and future graphics/GPU compute programming model trends

Problem: CPU bottleneck



- **Problem: graphics application is bottlenecked by CPU thread performance**
  - Consider: 5,000 objects, 60Hz, 5 draws per frame (different materials per object, objects drawn multiple times (shadow maps, reflection maps), etc.) —> ~1.5 million draws per frame
    - State of the art engines are in the range of 10's of thousands.
  - Graphics application must iterate through scene database and make appropriate graphics calls (for each scene object: set graphics state based on material, etc. "if object features wood material is lit by three lights, and is close to the camera, bind these textures and shaders")
  - Graphics driver (running on CPU) can only process a fixed number of draw calls per frame: limited by performance of building GPU command buffer from sequence of graphics API calls

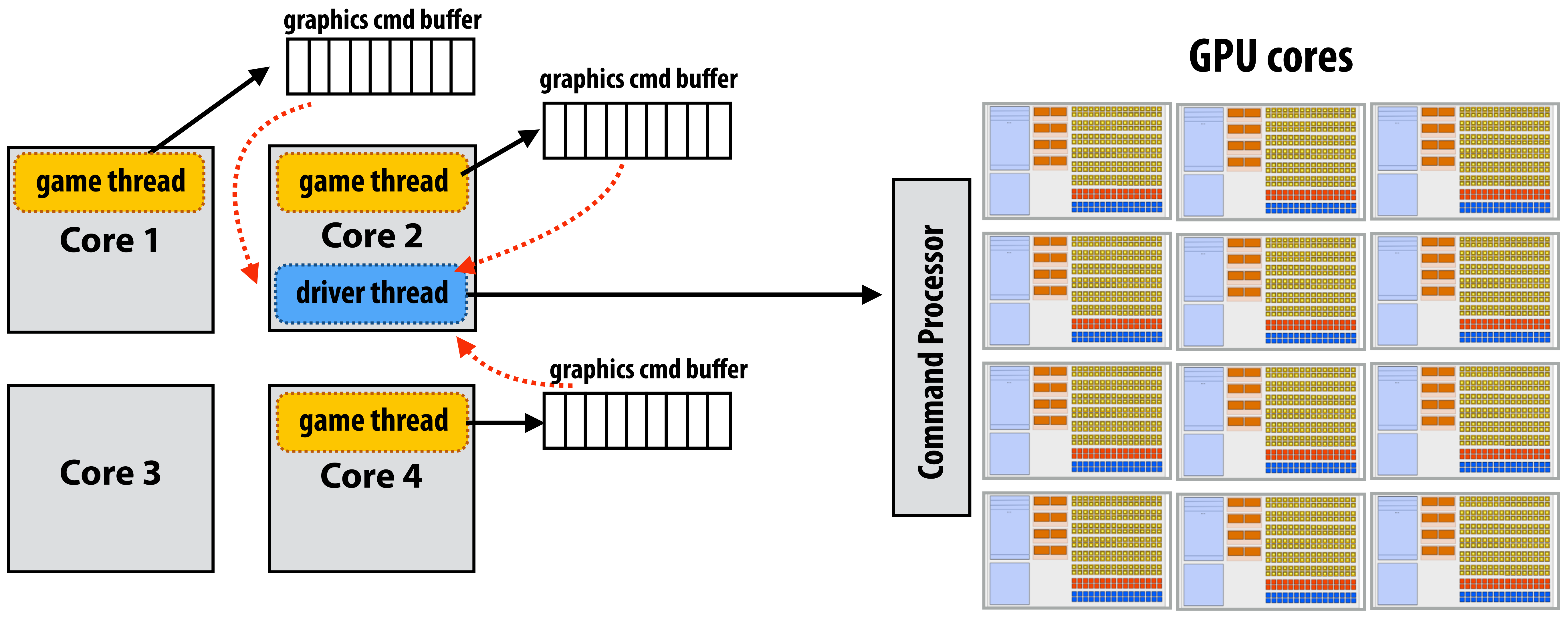
Trend: "to-the-metal" APIs

- AMD Mantle, Apple Metal, Direct3D 12 (announced)
- Idea: pull responsibility for resource management and command buffer generation from monolithic graphics driver back into graphics application
    - Application-facing graphics API changes from issuing draw calls to building GPU command buffers
    - Note 1: the graphics pipeline abstraction is largely unchanged (but now applications have access to a lower-level <u>interface</u> for generating work for the same pipeline abstraction)
    - Note 2: this interface was already the status-quo for game consoles, so PC graphics is just catching up (consoles have low-performance CPUs, and console games are written to a specific GPU architecture, so a lower-level of abstraction has made sense for some time)

MANTLE | POWERED BY AMD

Metal

Example: parallel command buffer generation by multiple CPU threads

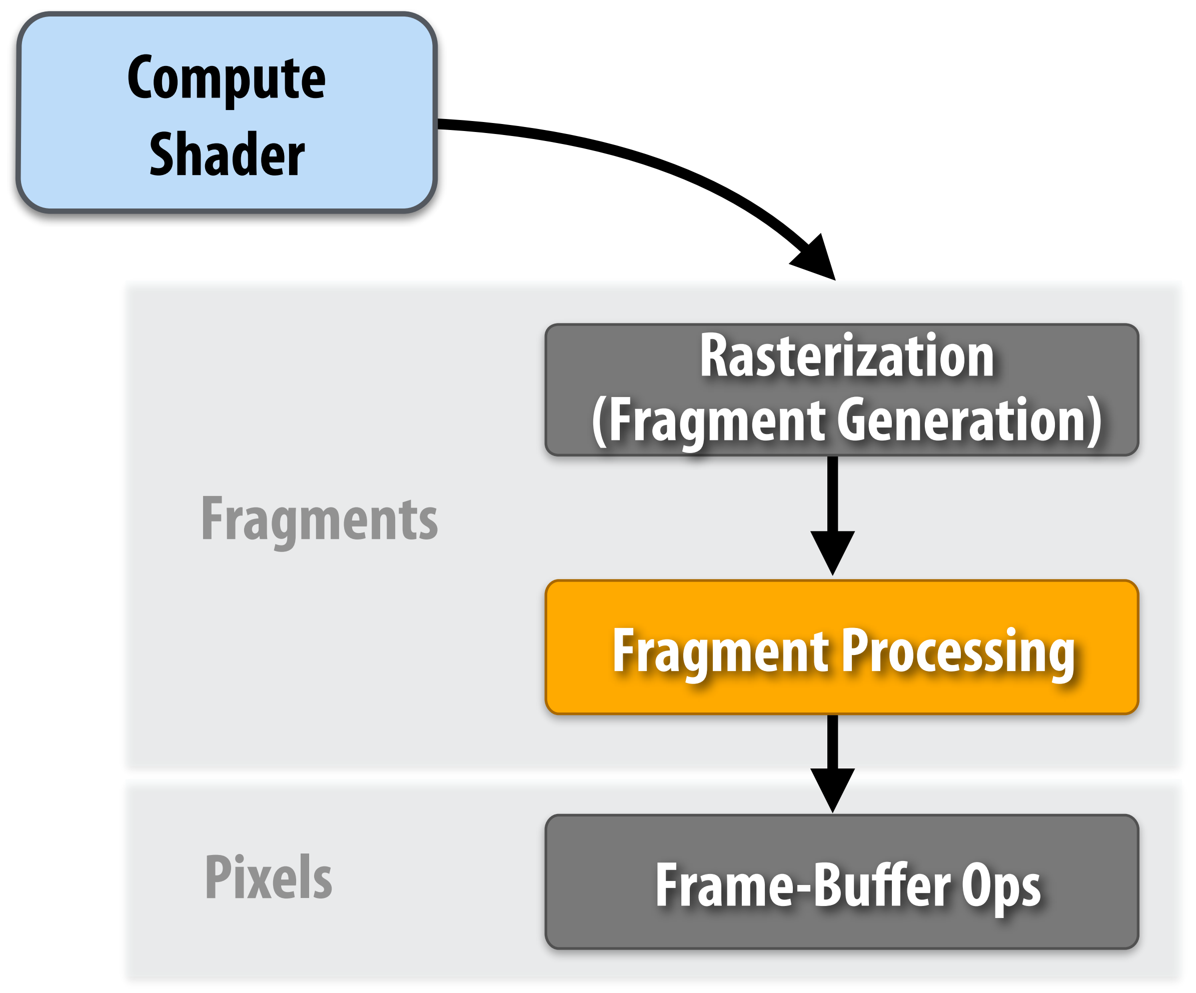


- **Efficiency improvements:**
  - Parallel command buffer generation (API handed command buffers that are constructed in parallel)
  - Management of command buffer resources can be specialized to application

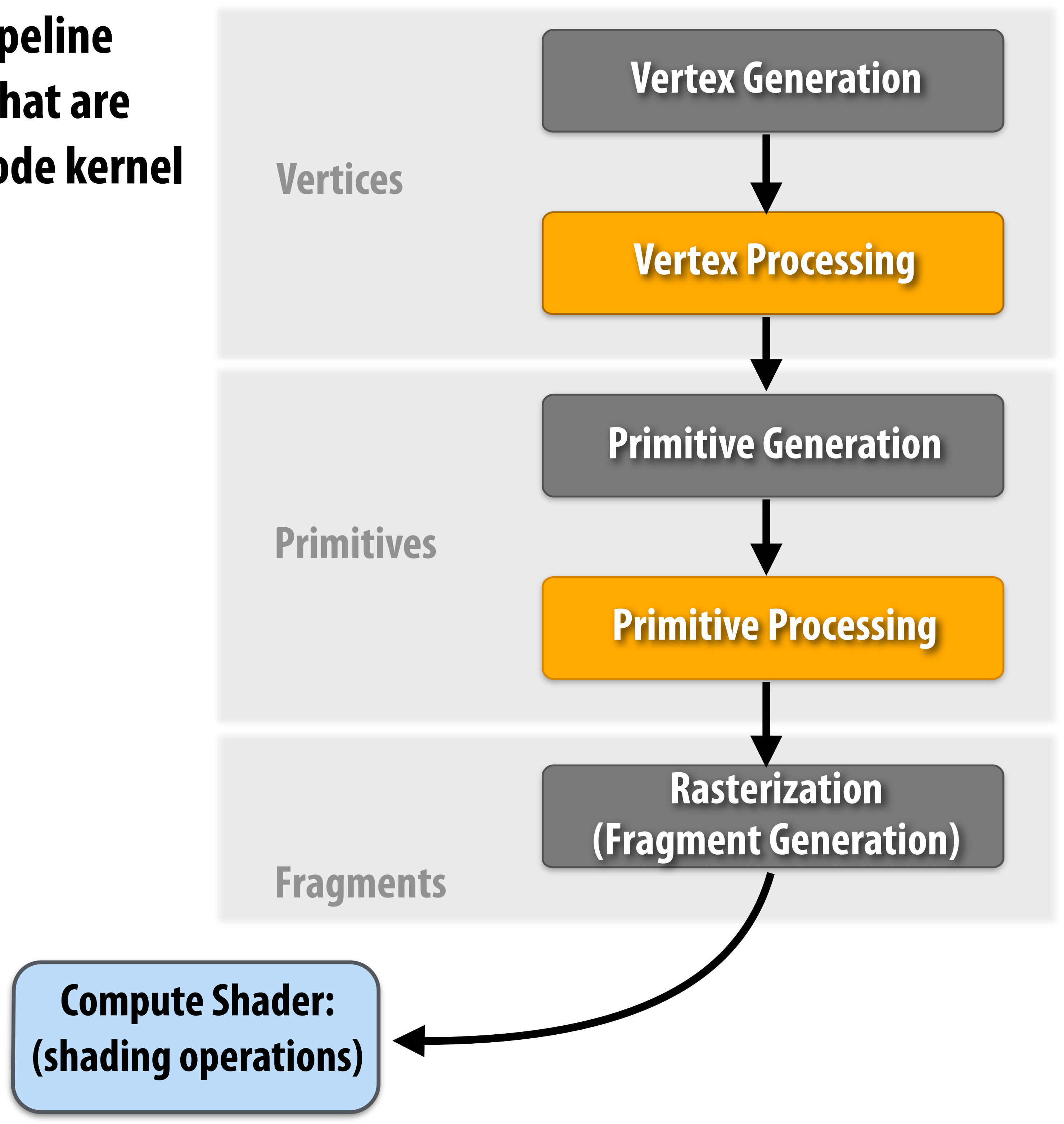
Future trend: compute-mode, graphics-mode integration

- Currently there are two distinct "worlds" for GPU programming
  - Graphics mode: send commands to GPU pipeline, update graphics buffers
  - Compute mode: send commands for CUDA/compute shader processing, update compute buffers
  - Buffer transfer/copy routines make compute buffers visible to graphics (and vice-versa)
- Better fusion of these two programming models is desirable

Example 1: compute mode computations generate geometry directly for graphics pipeline



Example 2: graphics pipeline generates fragments that are shaded by compute mode kernel



Reading

- T. Foley et al. *Spark: Modular, Composable Shaders for Graphics Hardware*. SIGGRAPH 2011
- J. Nickolls et al. *Scalable Parallel Programming with CUDA*. ACM Queue 2008
- See website for a number of interesting blog posts…