

Lecture 8:

Compute-mode GPU Programming Interfaces

**Visual Computing Systems
CMU 15-869, Fall 2013**

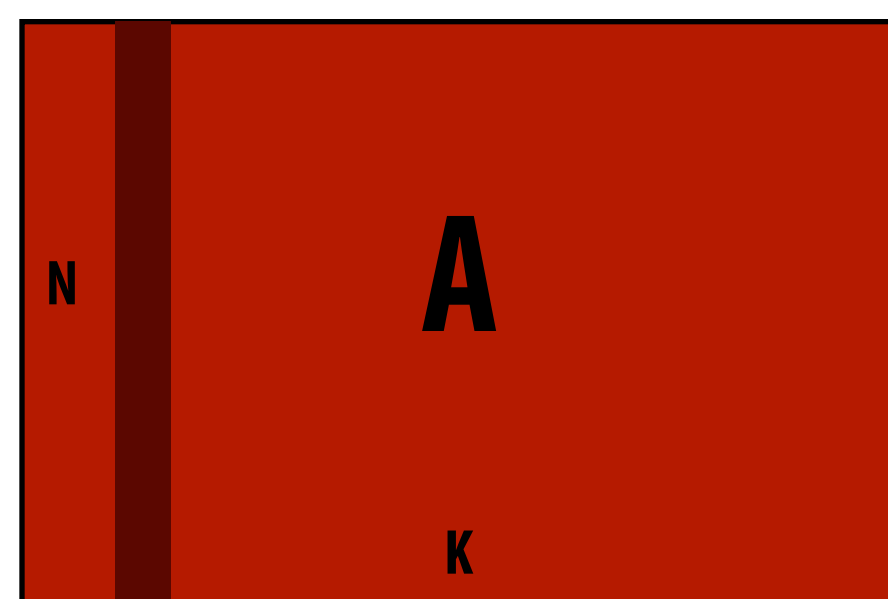
Today

- **Some “GPGPU” history**
- **The CUDA (or OpenCL) programming model**
- **GRAMPS: an attempt to create a programmable graphics pipeline**

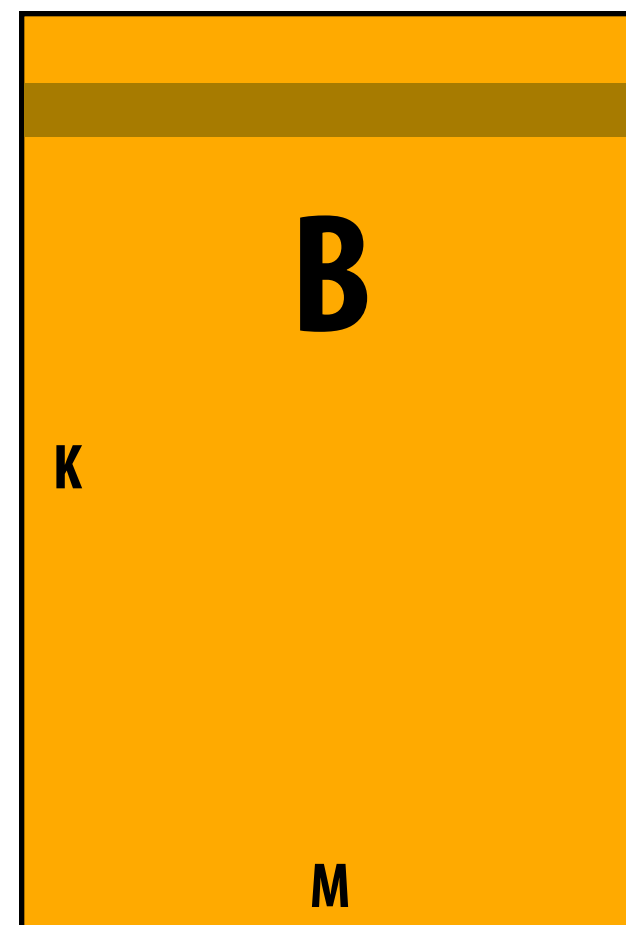
Early GPU-based scientific computation

Dense matrix-matrix multiplication

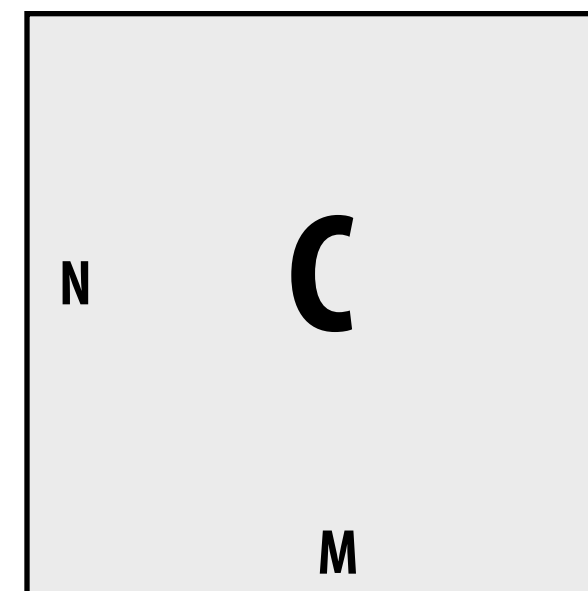
[Larson and McAllister, SC 2001]



K x N texture 0



M x K texture 1



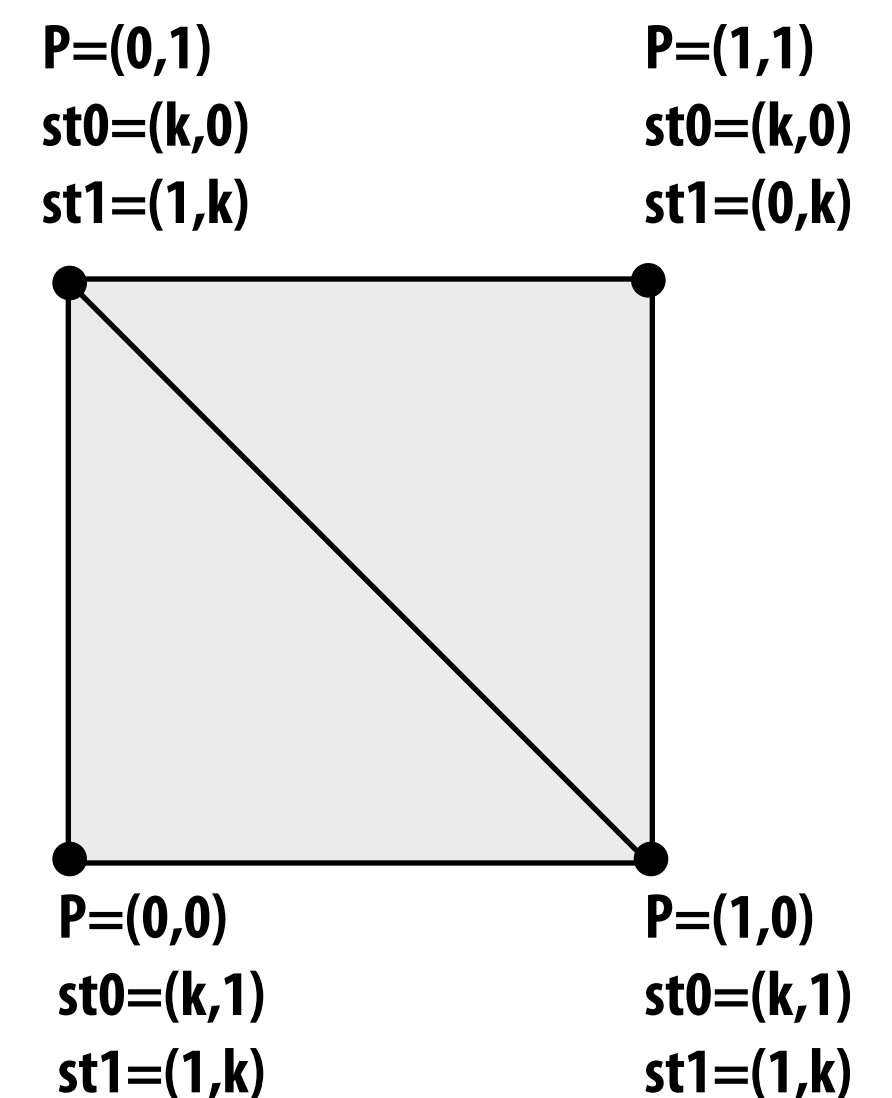
M x N frame buffer

Set frame buffer blend mode to ADD

for k=0 to K

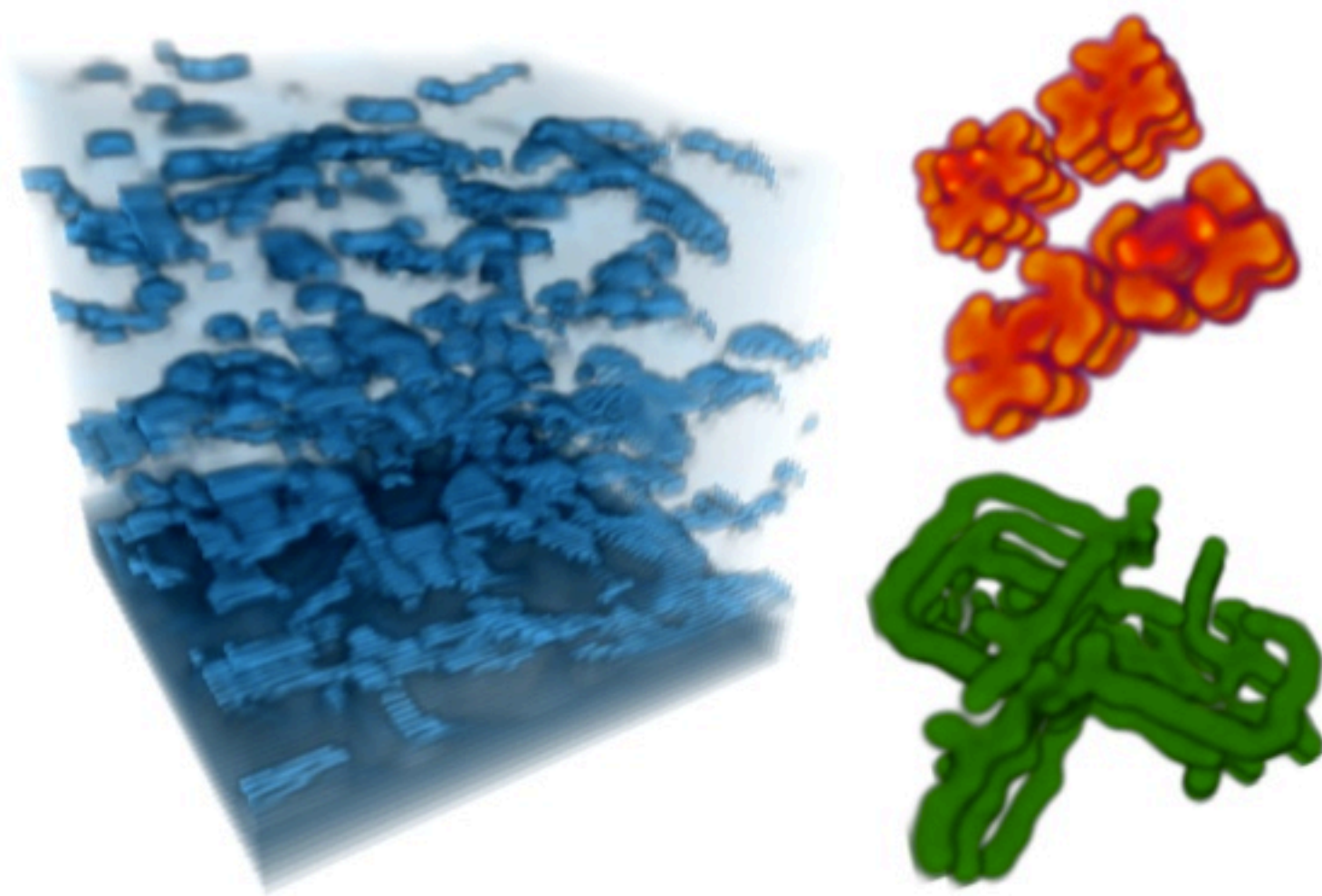
Set texture coords

Render 1 full-screen quadrilateral

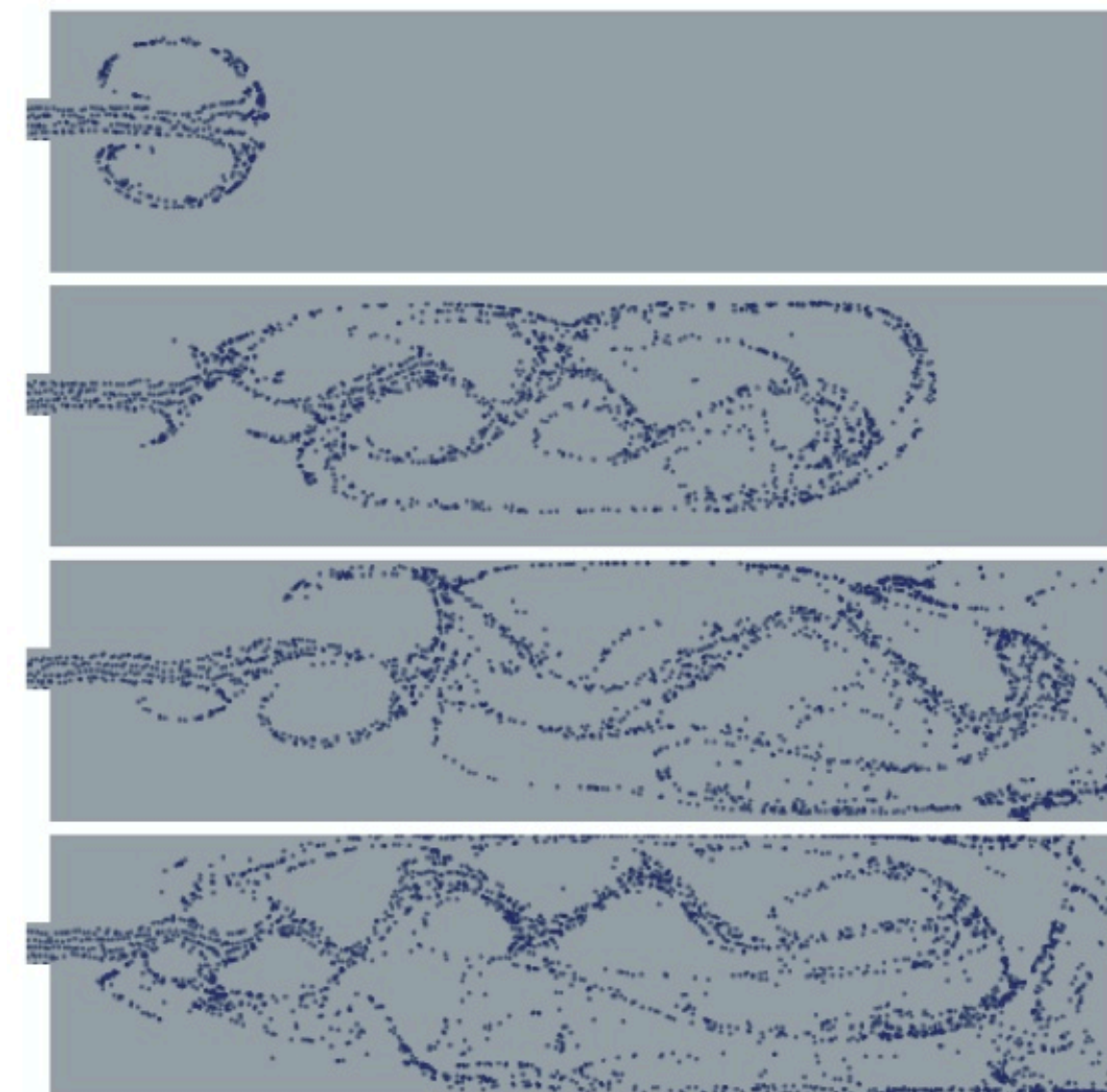


Note: this work followed [Percy 00], which modeled OpenGL with multi-texturing as a SIMD processor for multi-pass rendering (we discussed this last time in the shade-tree example)

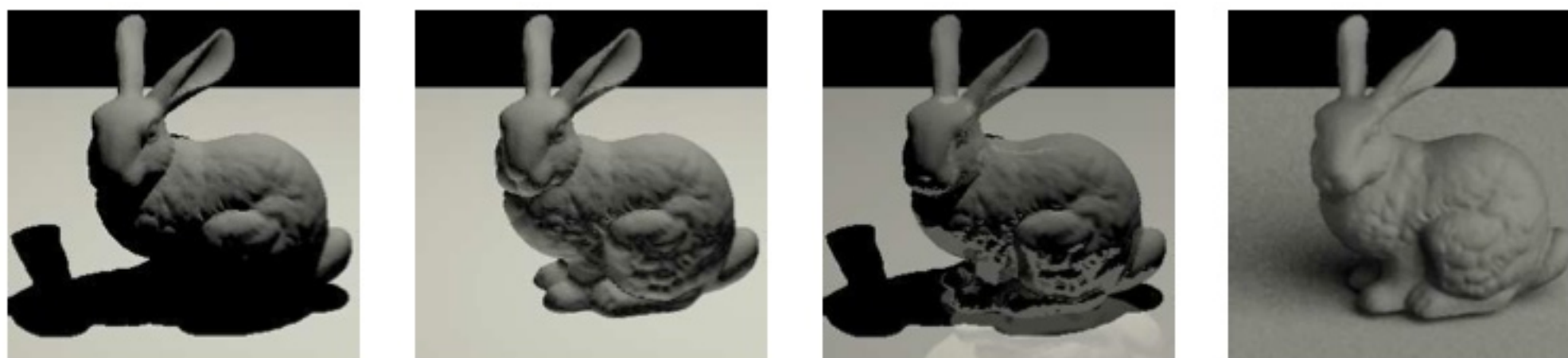
“GPGPU” 2002-2003



Coupled Map Lattice Simulation [Harris 02]



Sparse Matrix Solvers [Bolz 03]



Ray Tracing on Programmable Graphics Hardware [Purcell 02]

Brook for GPUs [Buck 04]

- **New language (C extensions) that abstracting GPU as a stream processor**
 - Streams: 1D, 2D arrays of data
 - Kernels: per-element processing of stream data **
 - Reductions: stream --> scalar
- **Influences**
 - Data-parallel programming from scientific computing: ZPL, Nesl
 - Stream programming: StreamIT, StreamC/Kernel
- **Brook language runtime generated appropriate OpenGL calls**

Note:
No loops, array indexing, or
explicit parallelism in code.

**Semantics of kernel call are
to invoke kernel once per
output stream element**

```
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);
```


Stream programming (“pure”)

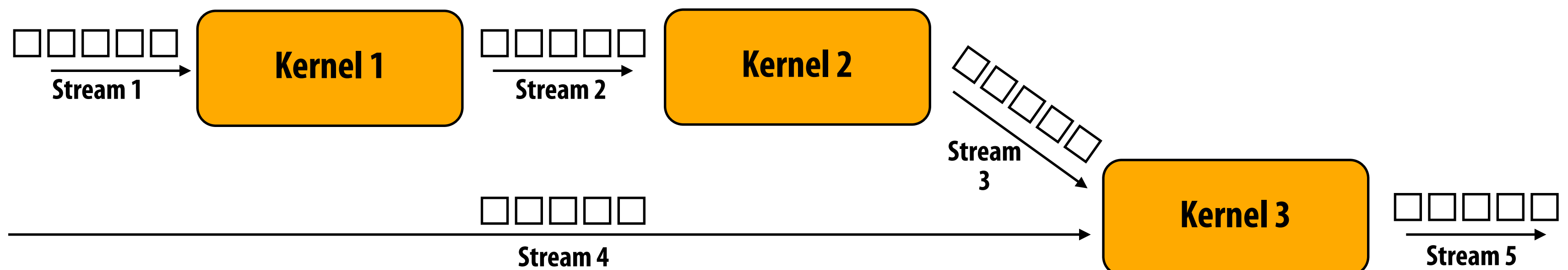
■ Streams

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

■ Kernels

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

- Modern implementations (e.g., StreamIT, StreamC/KernelC) relied on static scheduling by compiler to achieve high performance

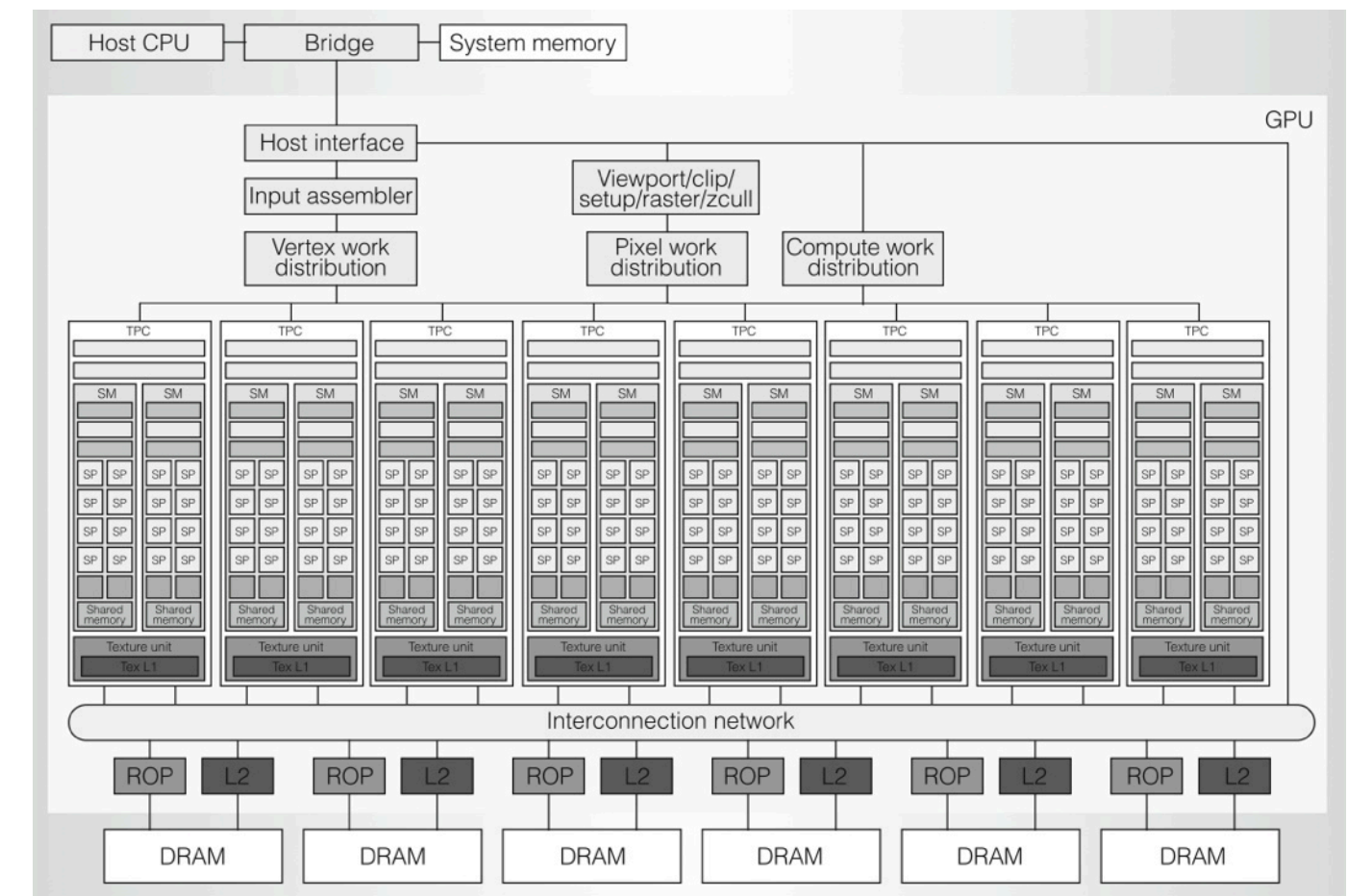


NVIDIA CUDA

[Ian Buck at NVIDIA, 2007]

■ Alternative programming interface to Tesla-class GPUs

- “Compute mode”
- Recall: Tesla was first “unified shading” GPU



■ Low level abstraction, reflects capabilities of hardware

- Recall arguments in Cg paper
- Combines some elements of streaming and multi-threading (like HW does)

■ Open standards embodiment of this programming model is OpenCL (Microsoft embodiment is 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 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 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);
```

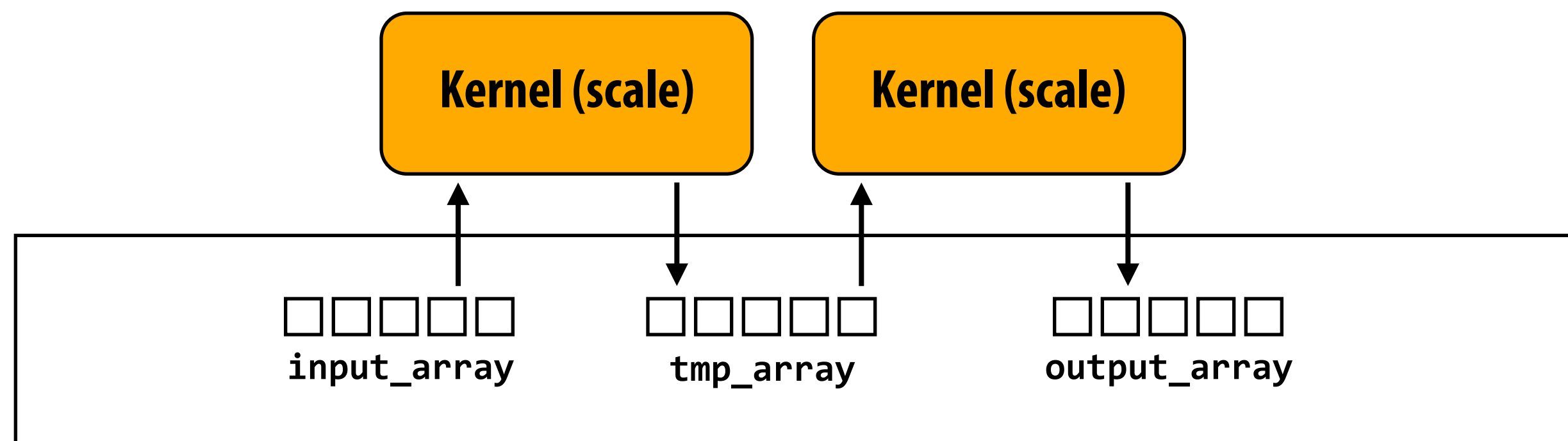
← Bulk thread launch: logically spawns N threads

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 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 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 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, thinking about a CUDA kernel as a stream processing kernel, and CUDA arrays as streams is perfectly reasonable.

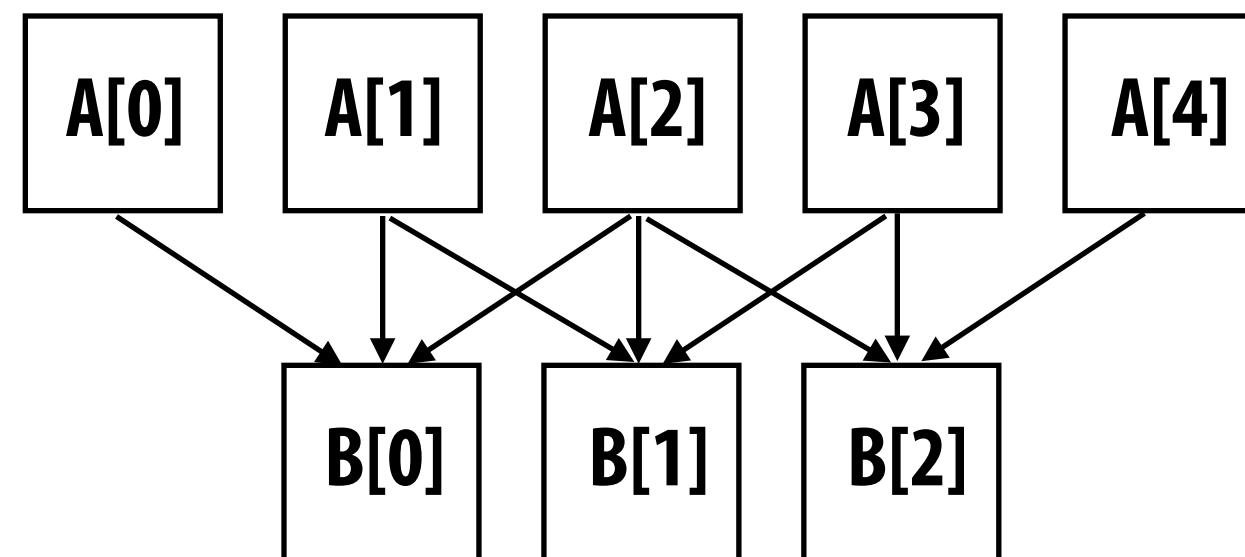
(programmer just has to do a little indexing in the kernel to get a reference to stream inputs/outputs)

Convolution example

```
// assume len(A) = len(B) + 2
__global__ void convolve(float* a, float* b)
{
    // ignore
    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 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];
}

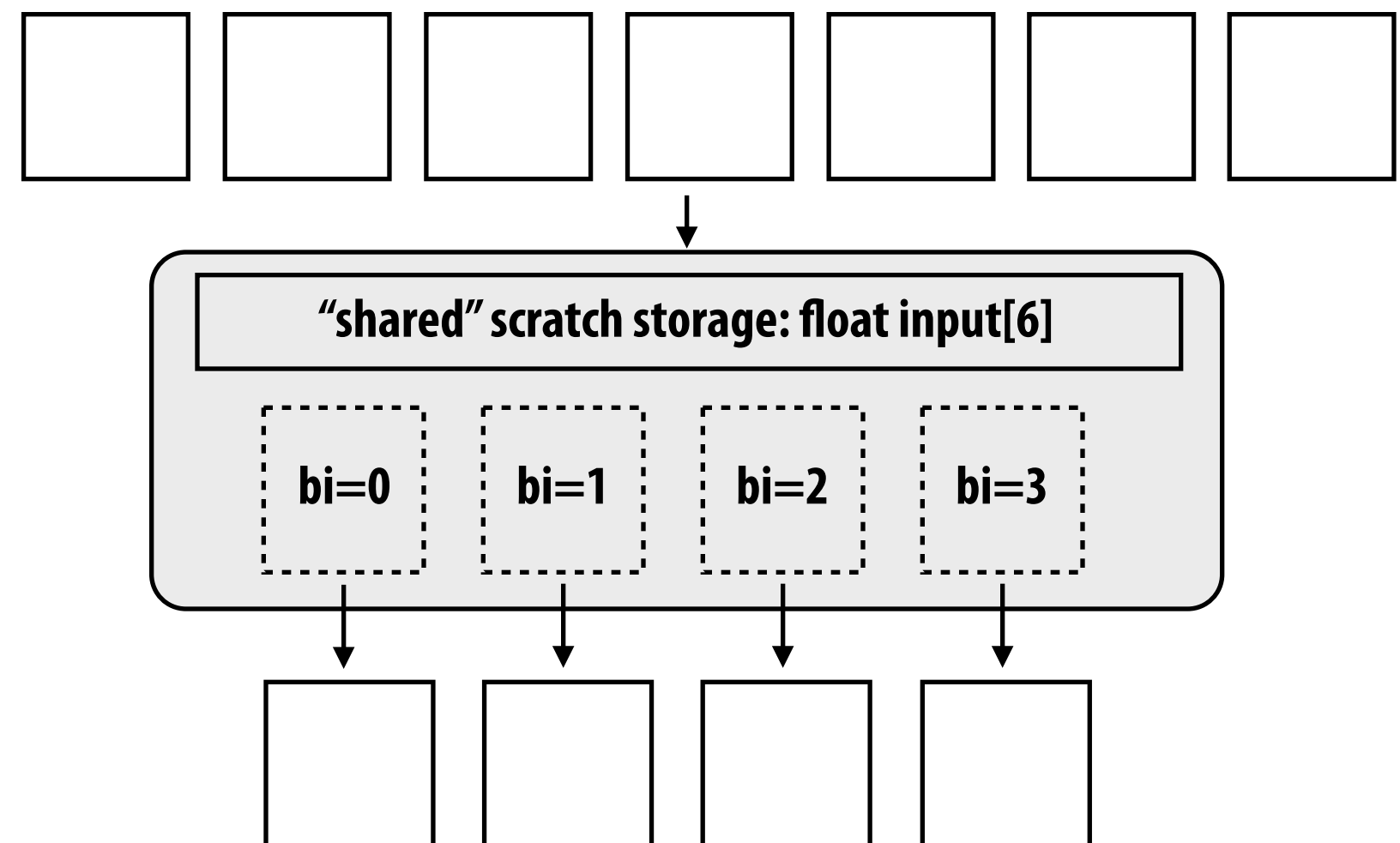
// allocation omitted
// assume len(A) = N+2, len(B)=N
float* A, *B;

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

CUDA threads are grouped into thread blocks

Threads in a block are not 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>>();
```

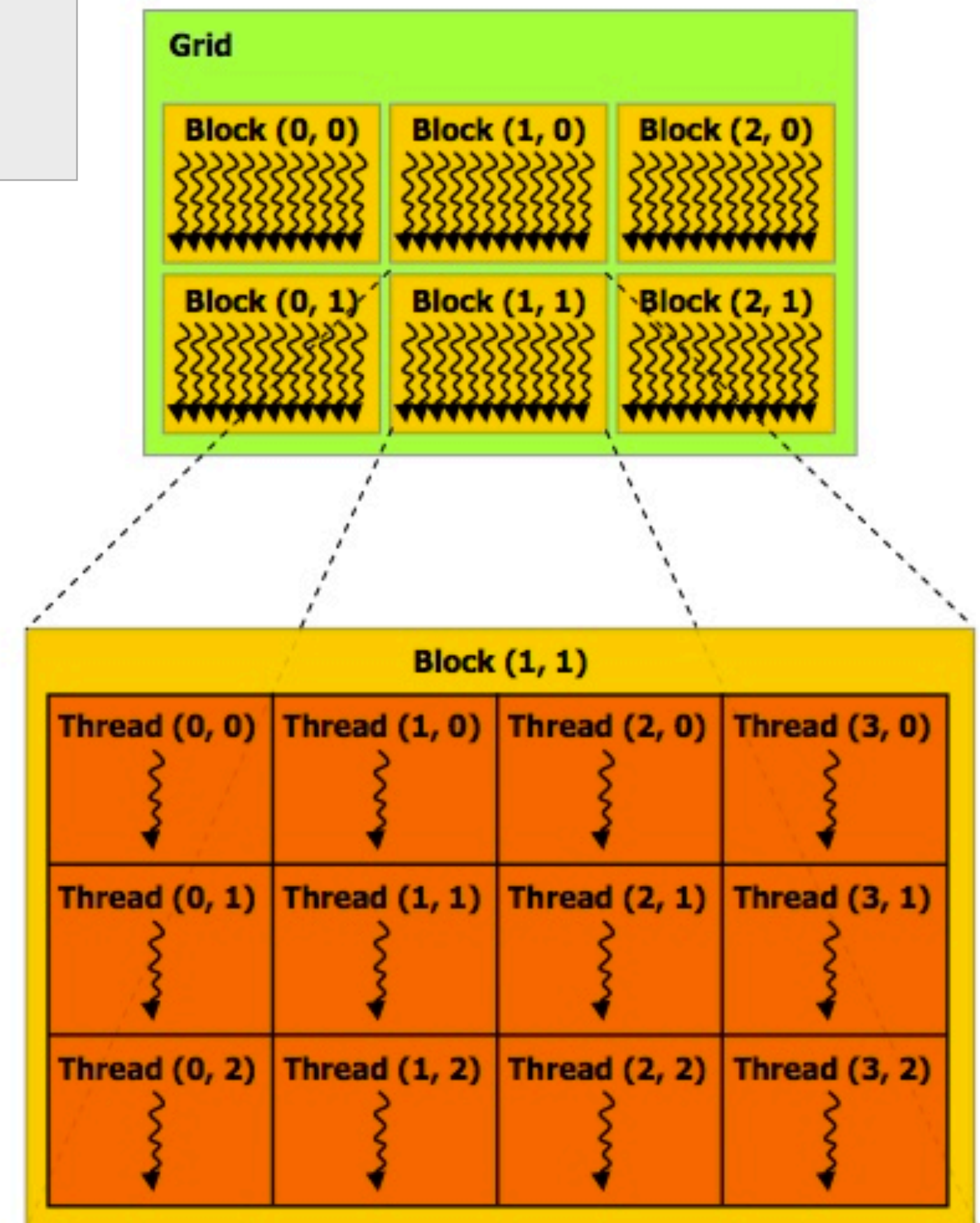
Thread blocks (and the overall “grid” of blocks) can be 1D, 2D, 3D
(Convenience: many CUDA programs operate on n-D grids)

Thread blocks represent independent execution

Threads in a thread block executed simultaneously on same GPU core

Why on the same core?

Why simultaneously?



Source: CUDA Programming Manual

The common way to think about CUDA

(thread centric)

- **CUDA is a multi-threaded programming model**
- **Threads are logically grouped together into blocks and gang scheduled onto cores**
- **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: thread block centric)

- **CUDA is a stream programming model (recall Brook)**
 - **Stream elements are now blocks of data**
 - **Kernels are thread blocks (larger working sets)**
- **Kernel invocations independent, but are multi-threaded**
 - **Multi-threading exposed additional fine-grained parallelism**
- **Think: Implicitly parallel across thread blocks (kernels)**
- **Think: Explicitly parallel within a block**

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 threads should be in a thread block?

Recall from GPU core lecture:

How many threads per core?

How much shared local memory per core?

CUDA programming style: “persistent” threads

- No semblance of streaming at all any more
- Programmer is always thinking explicitly parallel, and writing code that is specific to the number of processors in the machine
- Threads use atomic global memory operations to cooperate

```
// Persistent thread: Run until work is done, processing multiple work
// elements, rather than just one. Terminates when no more work is available
__global__ void persistent(int* ahead, int* bhead, int count, float* a, float* b)
{
    int in_index;
    while ( (in_index = read_and_increment(ahead)) < count)
    {
        // load a[in_index];

        // do work

        int out_index = read_and_increment(bhead);

        // write result to b[out_index]
    }
}

// launch exactly enough threads to fill up machine
// (to achieve sufficient parallelism and latency hiding)
persistent<<numBlocks,blockSize>>(ahead_addr, bhead_addr, total_count, A, B);
```

Questions:

What does CUDA system do for the programmer?

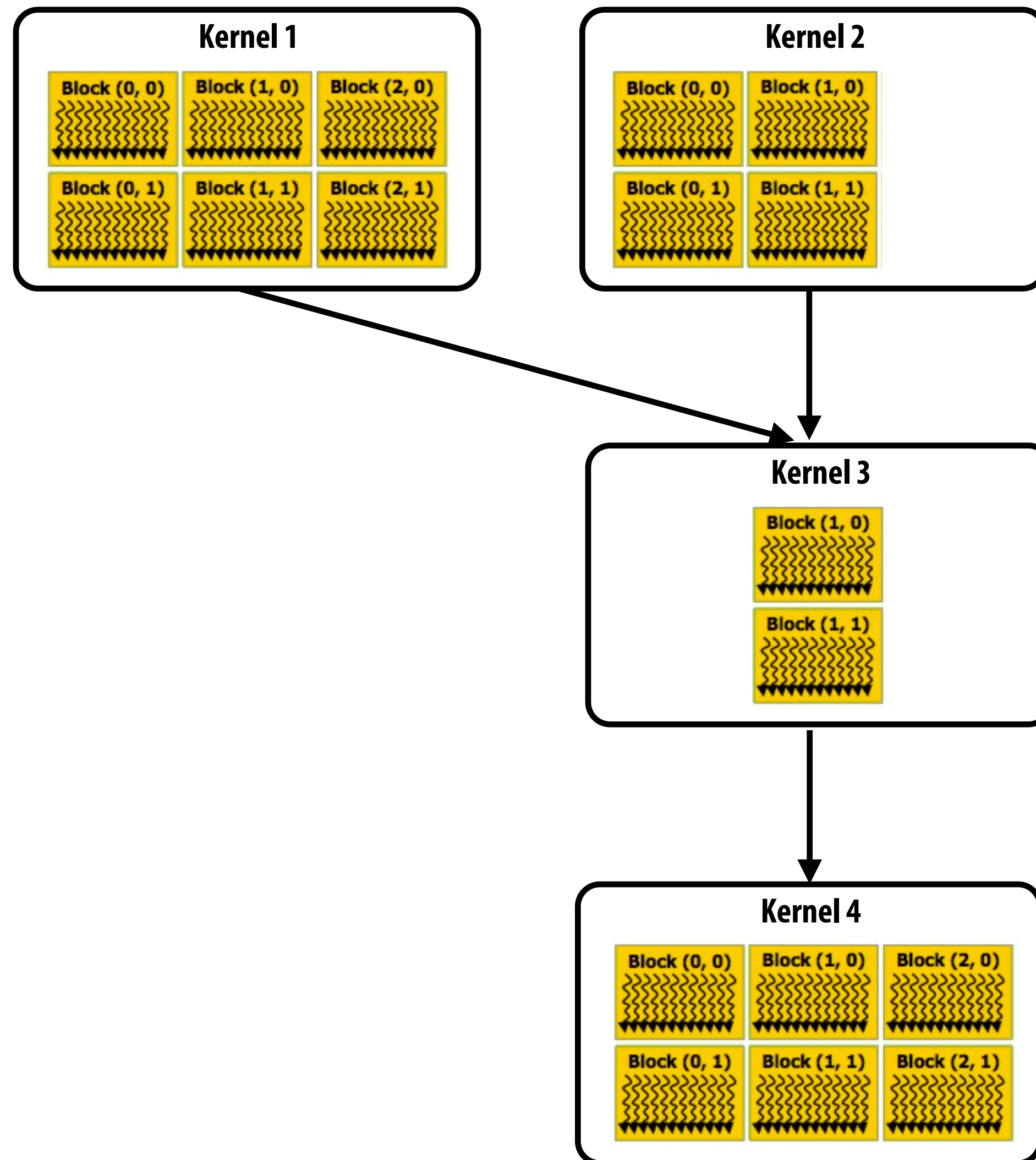
How does it compare to OpenGL?

Quick aside: why was CUDA successful?

(Kayvon's personal opinion)

1. **Provides access to a cheap, very fast machine (modern GPU)**
2. **SPMD abstraction allows programmer to write scalar code, have it (almost trivially) mapped to wide SIMD hardware**
 - **Compilers for generating explicit vector code easily for CPUs remain shockingly immature, even in 2013 (see Intel's ISPC for a useful tool)**
3. **More like thread programming than streaming: arbitrary in-kernel array indexing (+ GPU hardware multi-threading to hide memory latency)**
 - **More familiar, convenient, and flexible in comparison to more principled data-parallel or streaming systems**
[StreamC/KernelC, StreamMIT, ZPL, Nesl, synchronous data-flow, and many others]
 - **The first program written is often pretty good**
 - **1-to-1 with hardware behavior**

Modern CUDA/OpenCL: DAGs of kernel launches



Note: arrows are specified dependencies between batch thread launches

Think of each launch like a draw() command in OpenGL (but application can turn off order, removing dependency on previous launch)

Part II:

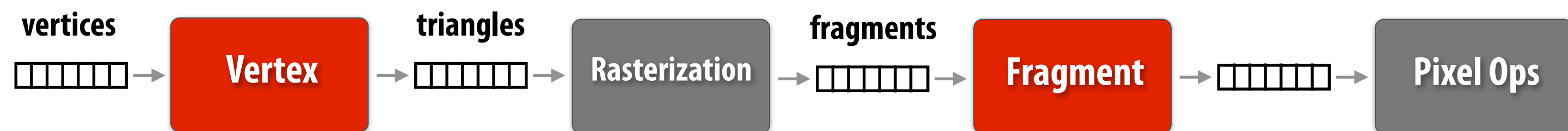
Programmable Graphics Pipelines

(Programmable Pipeline Structure, Not Programmable Stages)

A unique (undesirable?) property of GPU design

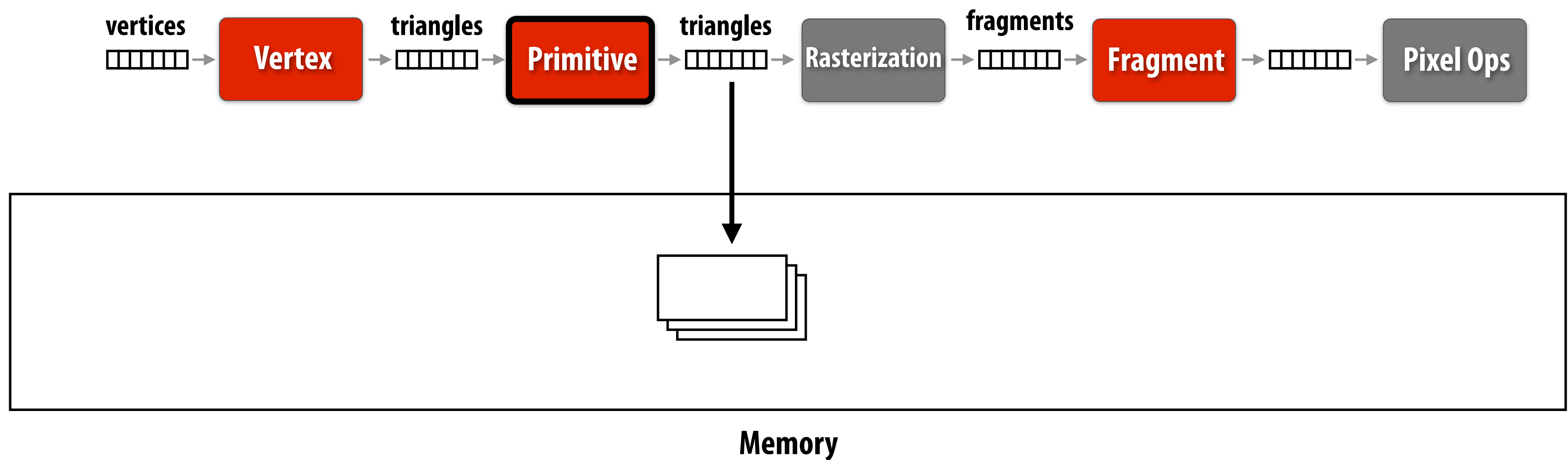
- **The fixed-function components on a GPU control the operation of the programmable components**
 - Fixed function logic generates work (e.g., input assembler, tessellator, rasterizer all generate elements for processing by programmable cores)
 - Programmable logic processes elements
- **In other words... application-programmable logic forms the inner loops of the rendering computation, not the outer loops!**
- **Ongoing research question: can we flip this design around?**
 - Maintain efficiency of heterogeneous hardware implementation, but give programmers control of how hardware is used and managed.

Graphics pipeline pre Direct3D 10



Graphics pipeline circa 2007

[Blythe, Direct3D 10]

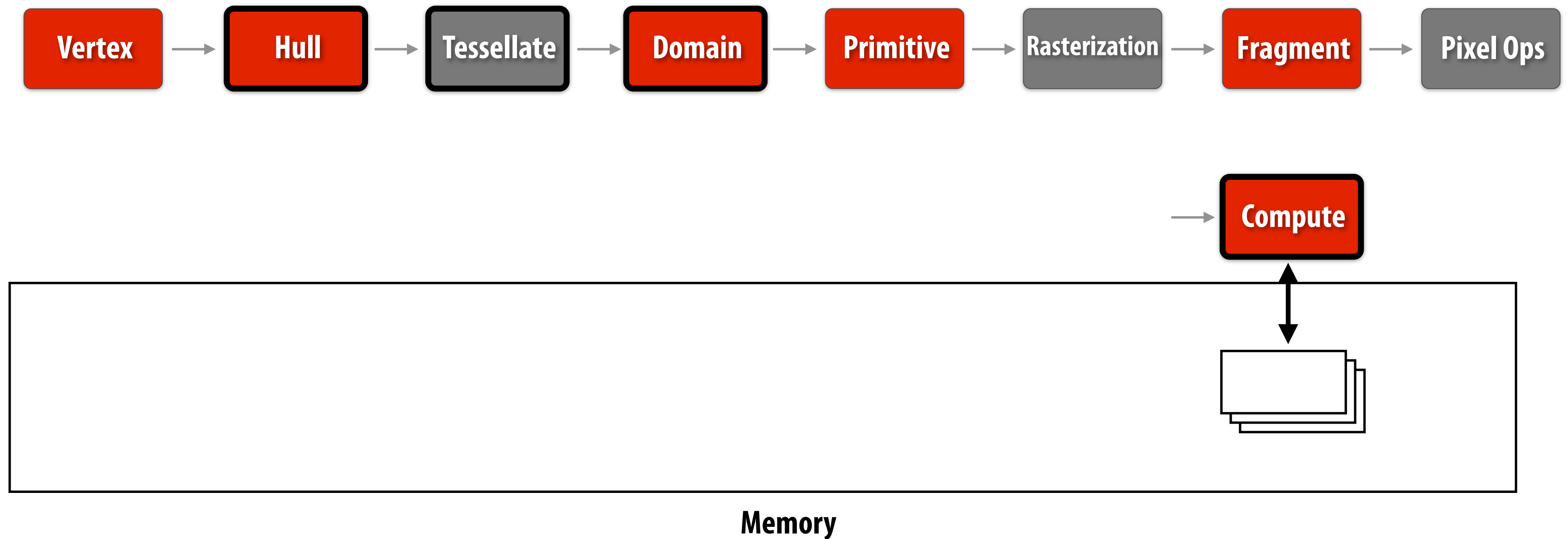


Added new stage

Added ability to dump intermediate results out to memory for reuse

Pipeline circa 2010

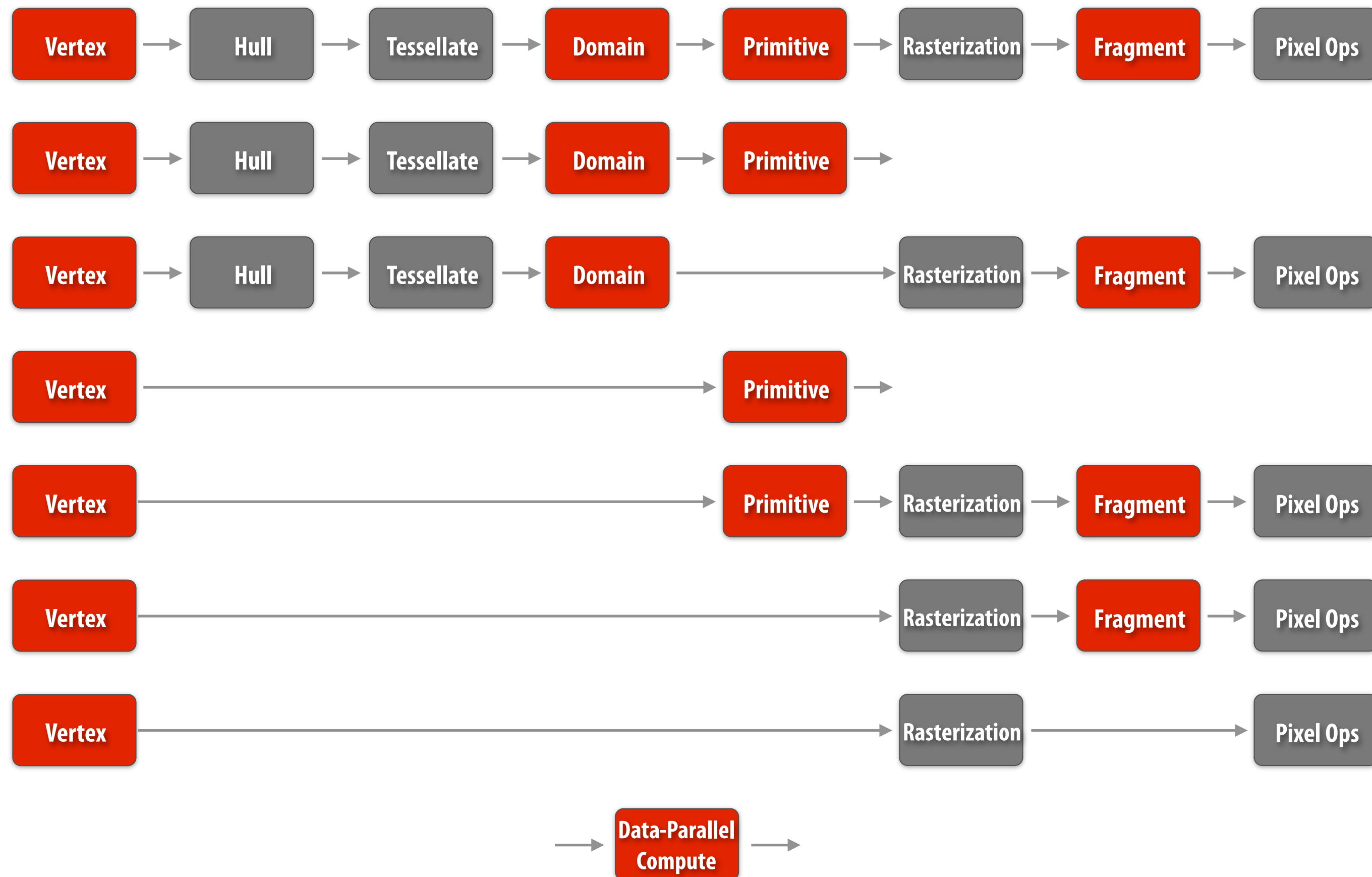
[Direct3D 11, OpenGL 4]



Added three new stages (new data flows needed to support high-quality surfaces)

Forked off a separate 1-stage pipeline (a.k.a. "OpenCL/CUDA")
(with relaxed data-access and communication/sync rules)

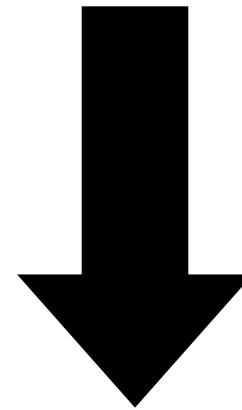
Modern graphics pipeline: highly configurable structure



Direct3D 11, OpenGL 4 pipeline configurations

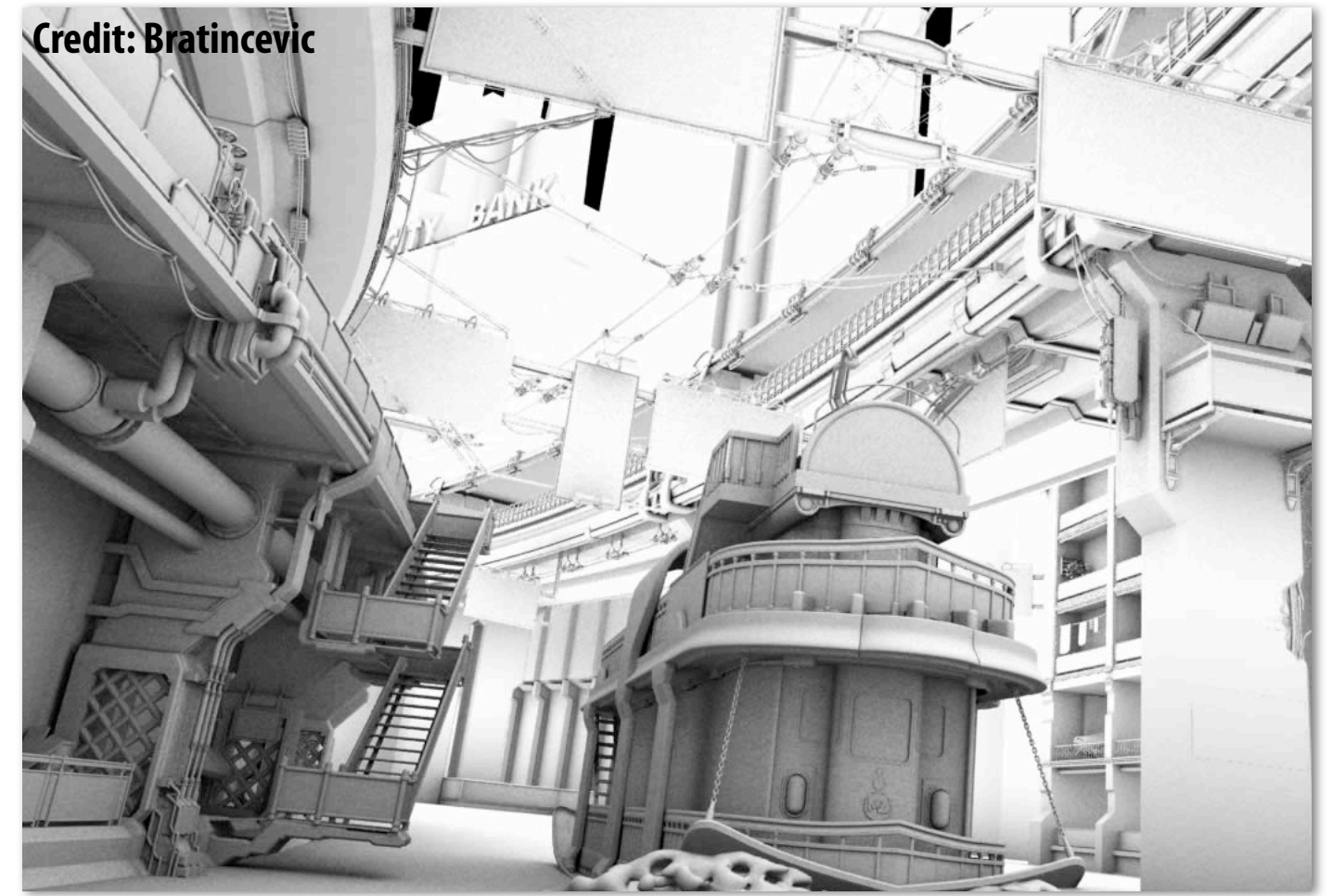
Current realities / trends in interactive graphics

- **Rapid parallel algorithm development in community**
- **Increasing machine performance**
 - **“Traditional” discrete GPU designs**
 - **Emerging hybrid CPU + GPU platforms** (“accelerated” many-core CPUs)



Space of candidate algorithms for future real-time use is growing rapidly

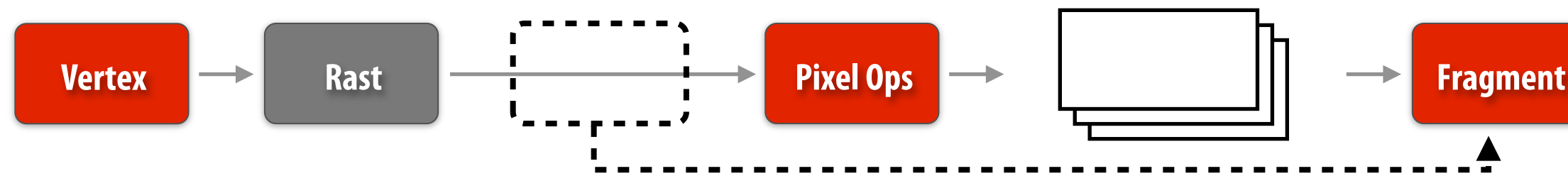
Example: global illumination algorithms



**Ray tracing:
for accurate reflections, shadows**



Alternative shading structures (“deferred shading”)



For more efficient scaling to many lights (1000 lights, [Andersson 09])

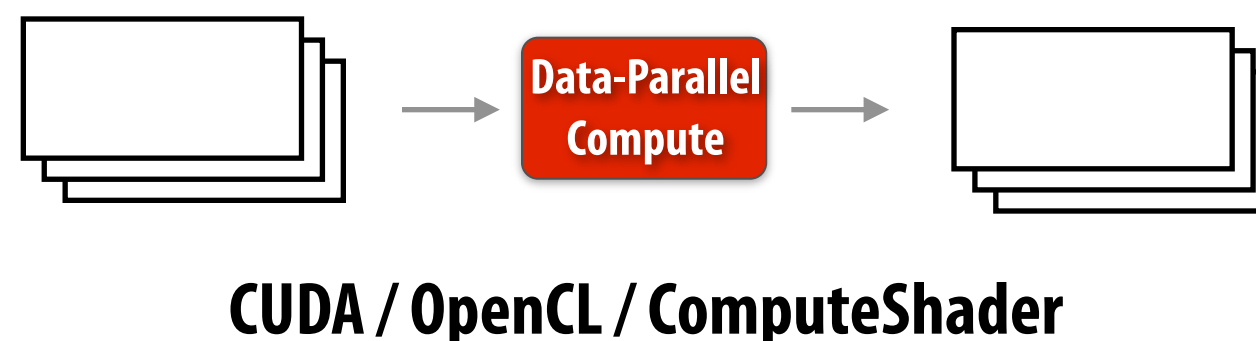
Game physics / simulation



Challenge

- **Future interactive systems → broad application scope**
 - Not a great fit for current pipeline structure
 - Pipeline structure could be extended further, but complexity is growing unmanageable
- **Must retain high efficiency of current systems**
 - Future hardware platforms (especially CPU+accelerator hybrids) will be designed to run these workloads well
 - Continue to leverage fixed-function processing when appropriate

Option 1: discard pipeline structure, drop to lower-level frameworks



Challenge

- **Future interactive systems → broad application scope**
 - Not a great fit for current pipeline structure
 - Pipeline structure could be extended further, but complexity is growing unmanageable
- **Must retain high efficiency of current systems**
 - Future hardware platforms (especially CPU+accelerator hybrids) will be designed to run these workloads well
 - Continue to leverage fixed-function processing when appropriate

Strategy: make the structure of the pipeline programmable



GRAMPS: A Programming Model for Graphics Pipelines

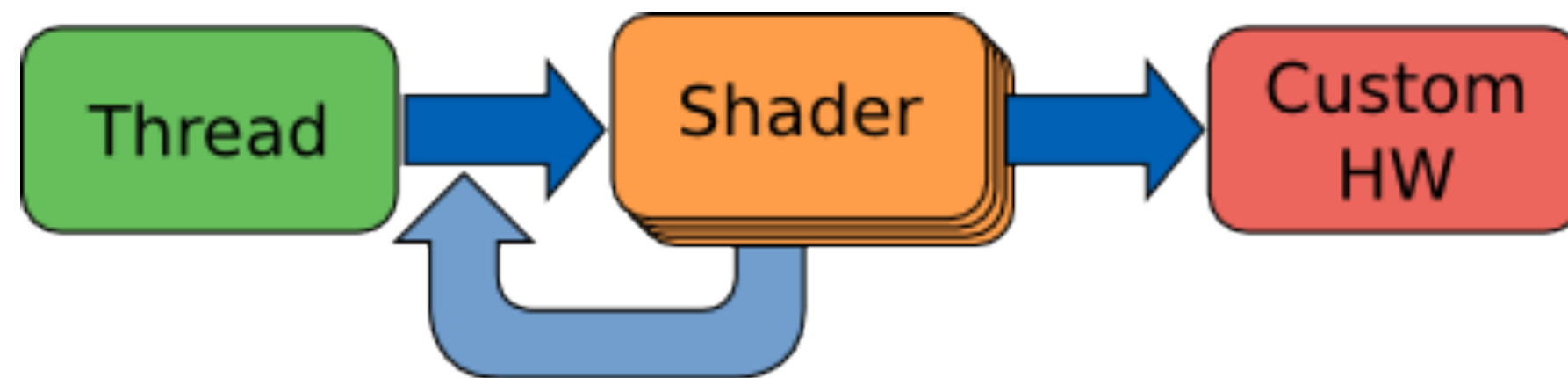
[Sugerman, Fatahalian, Boulos, Akeley, Hanrahan 2009]

GRAMPS programming system: goals

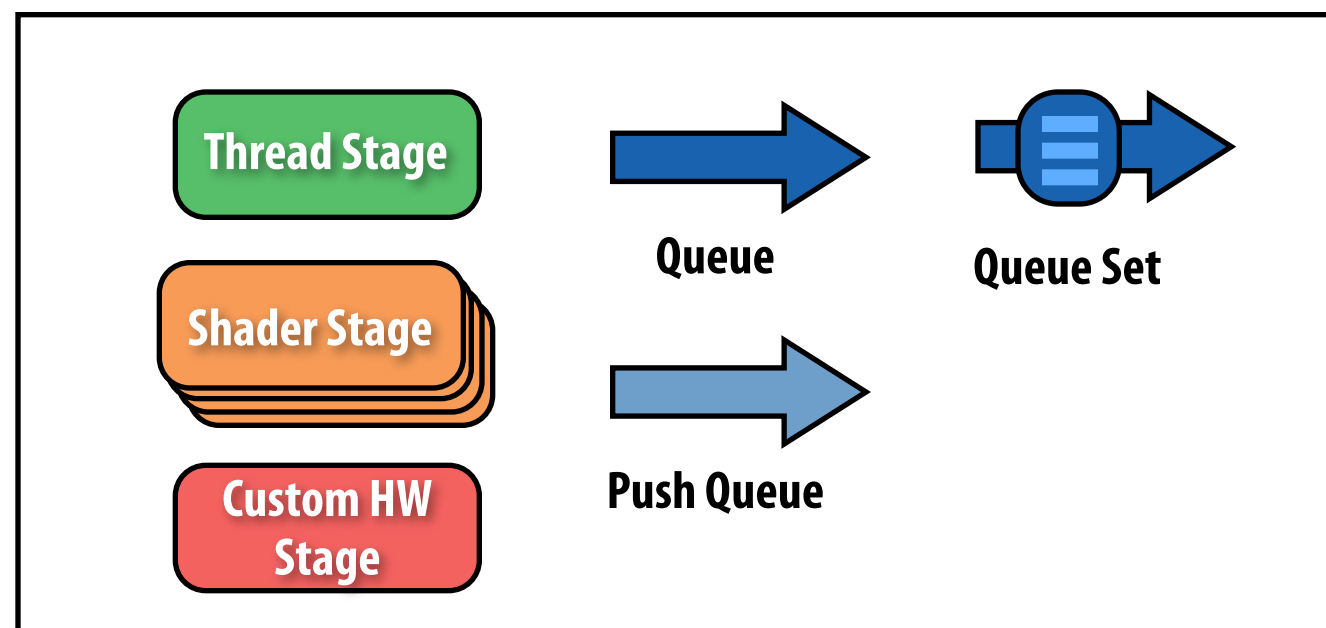
- **Enable development of application-defined graphics pipelines**
 - **Producer-consumer locality is important**
 - **Accommodate heterogeneity in workload**
 - **Many algorithms feature both regular data parallelism and irregular parallelism (recall: current graphics pipelines encapsulate irregularity in non-programmable parts of pipeline)**
- **High performance: target future CPU+GPUs (embrace heterogeneity)**
 - **Throughput (“accelerator”) processing cores**
 - **Traditional CPU-like processing cores**
 - **Fixed-function units**

GRAMPS overview

- **Programs are graphs of stages and queues**
 - **Expose program structure**
 - **Leave stage internals largely unconstrained**

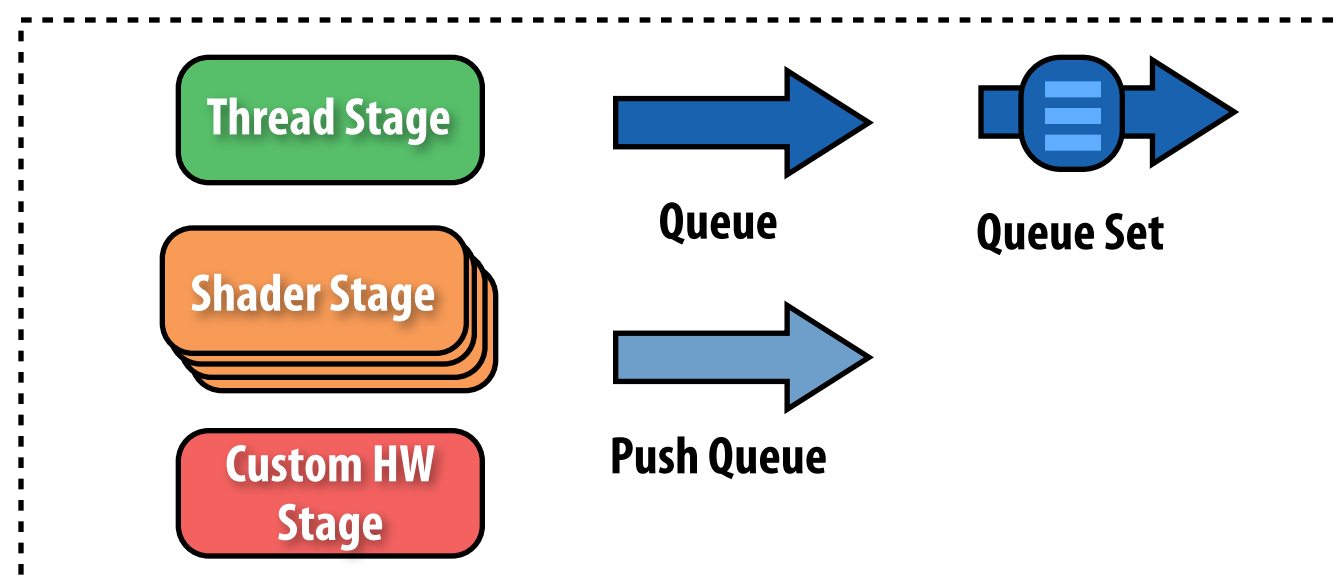
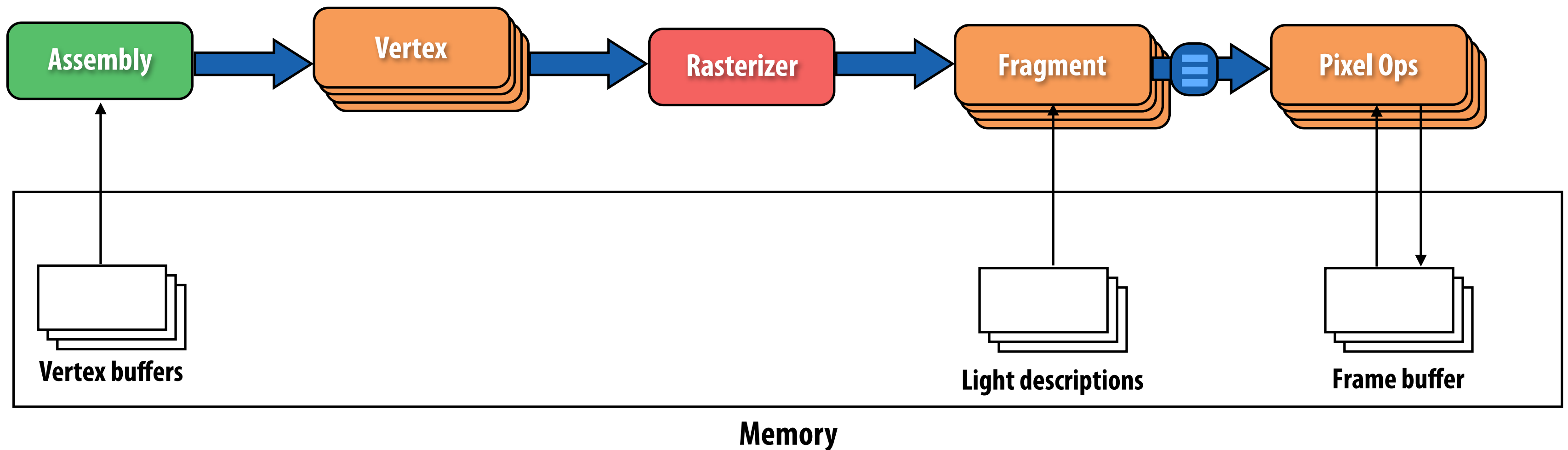


GRAMPS primitives



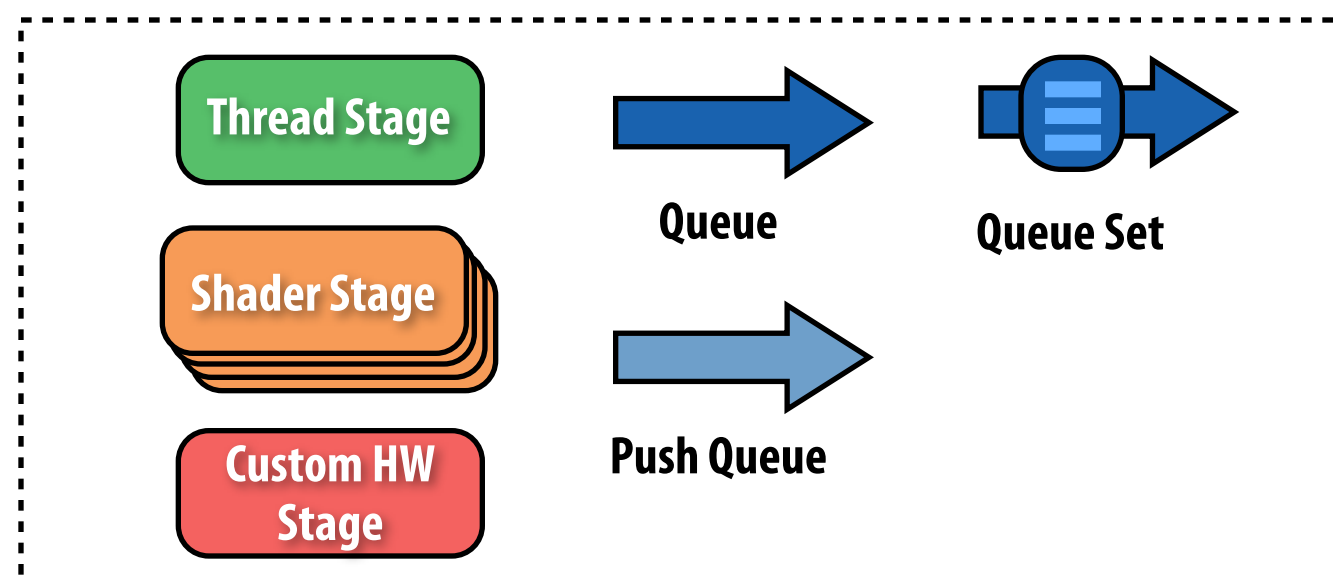
Writing a GRAMPS program

1. Design application graph and queues
2. Implement the stages
3. Instantiate graph and launch



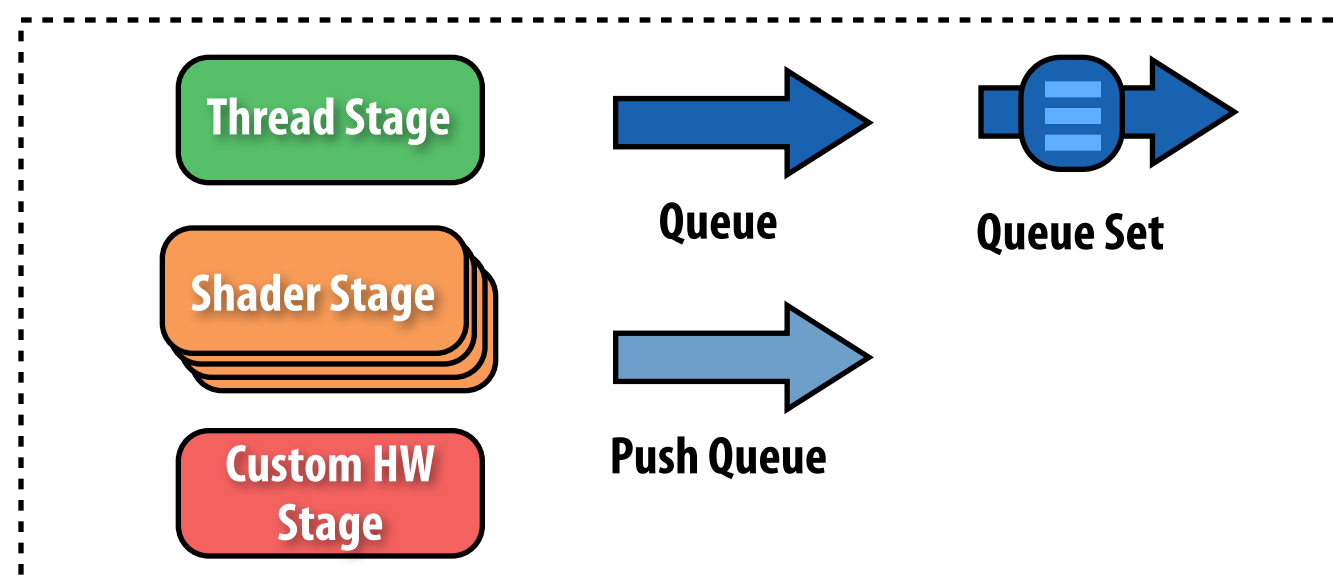
Queues

- **Bounded size, operate at granularity of “packets” (structs)**
 - Packets have one of two formats:
 1. Blob of data: completely opaque to system
 2. Header + array of opaque elements
- **Queues are optionally FIFOs (to preserve ordering)**



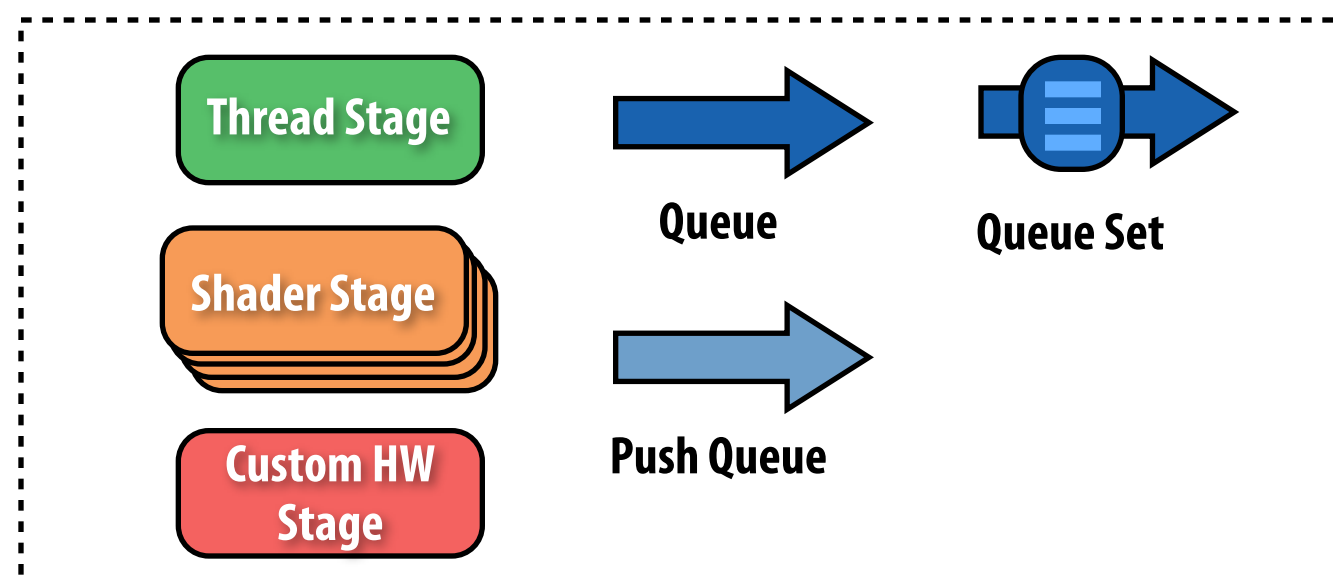
“Thread” and custom HW stages

- **Preemptible, long-lived and stateful (think pthreads)**
 - Threads orchestrate computation: merge, compare repack inputs
- **Manipulate queues via in-place `reserve/commit`**
- **Custom HW stages are logically just threads, but implemented by HW**



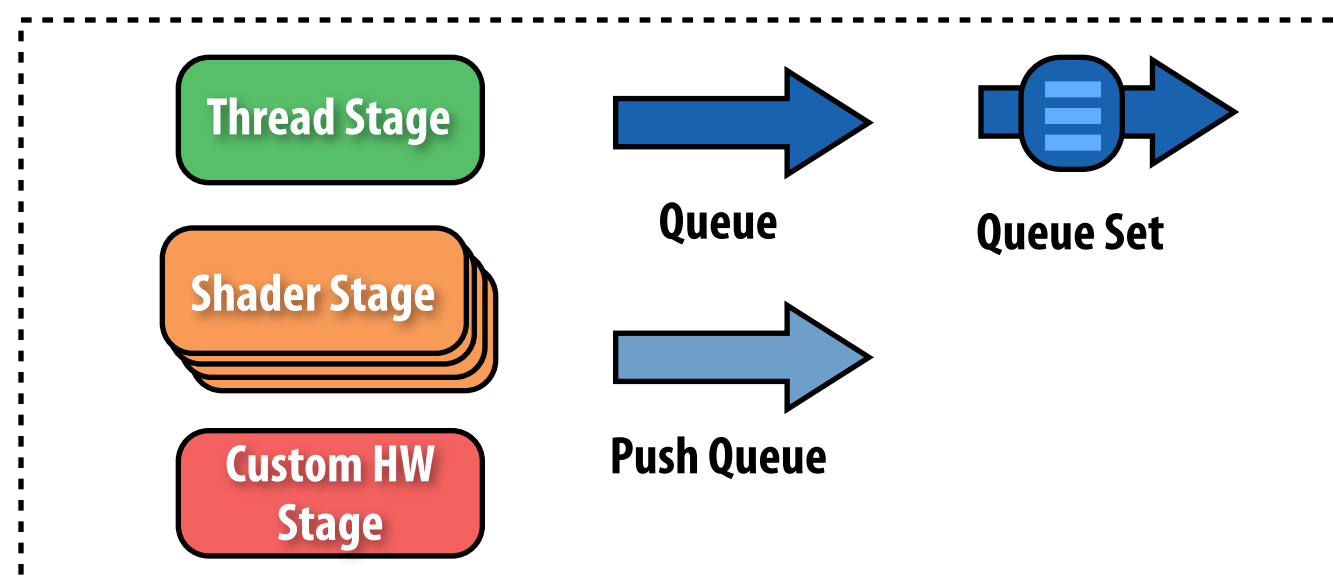
“Shader” stages

- **Anticipate data-parallel execution**
 - Defined per element (like graphics shaders today)
 - Automatically instanced and parallelized by GRAMPS
- **Non-preemptible, stateless**
 - System has preserved queue storage for inputs/outputs
- **Push: can output variable number of elements to output queue**
 - GRAMPS coalesces output into full packets (of header + array type)

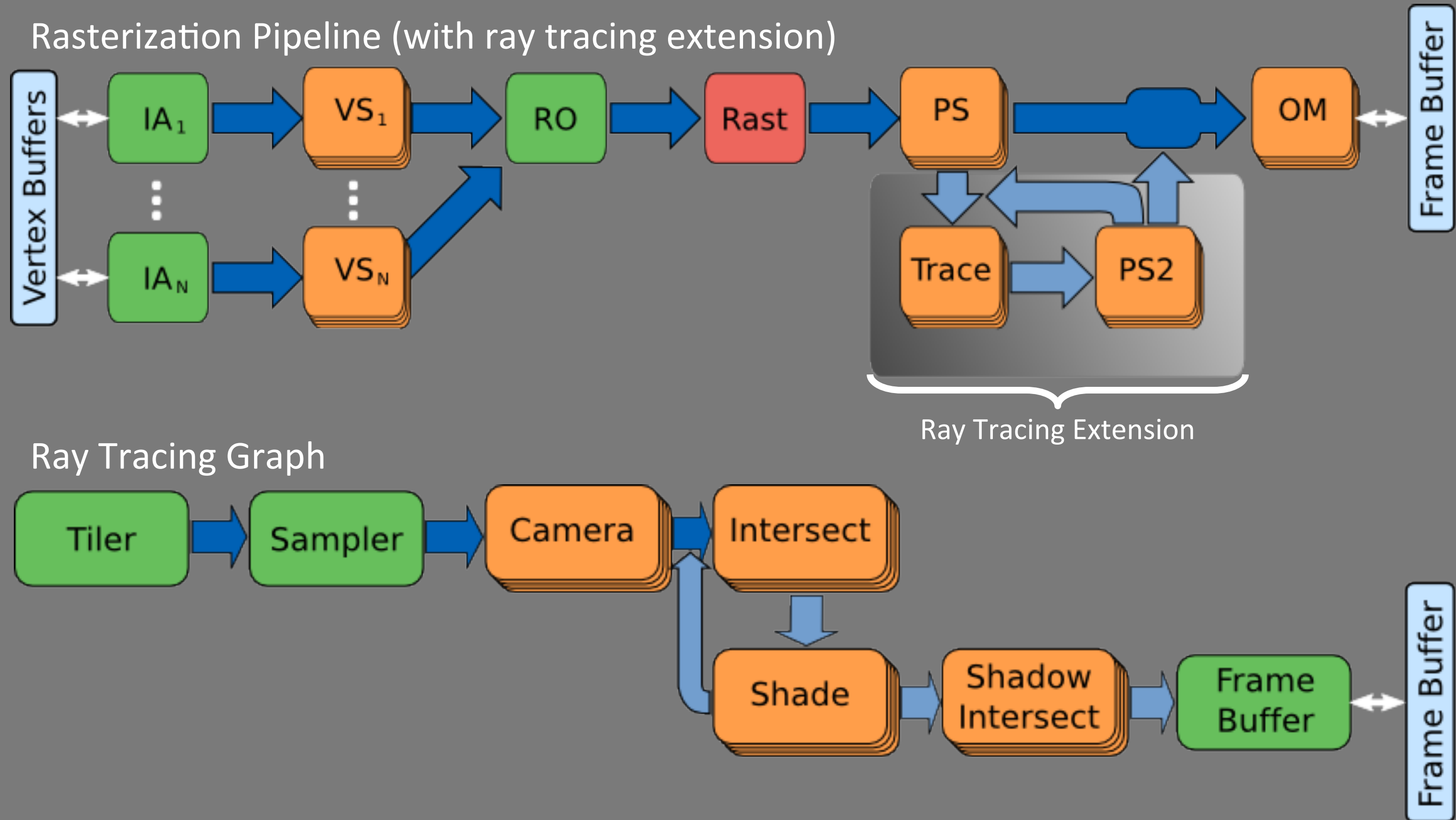


Queue sets (for mutual exclusion)

- Like N independent serial subqueues (but attached to a single instanced stage)
 - Subqueues created statically or on first output
 - Can be sparsely indexed (can think of subqueue index as a key)

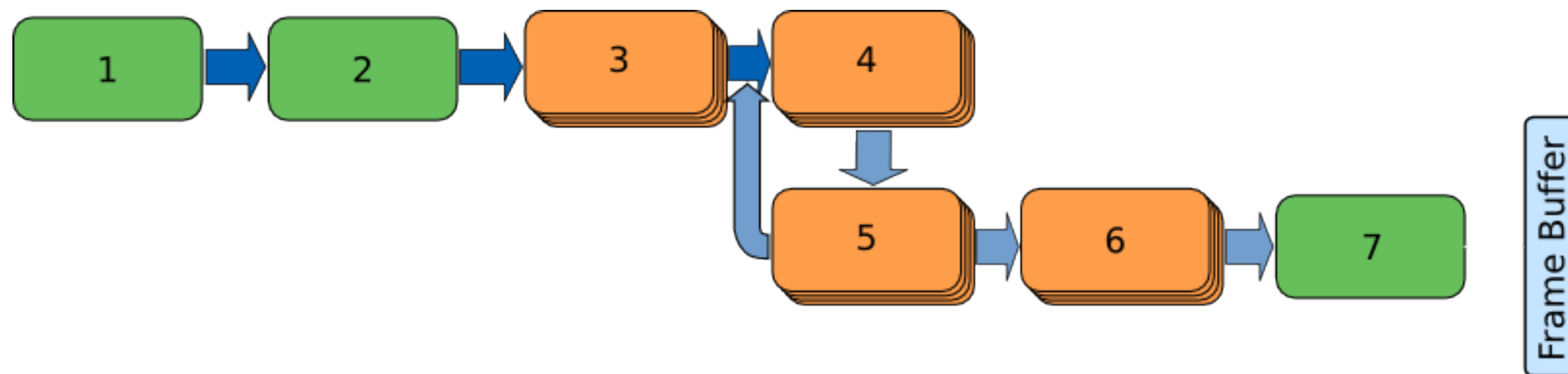


Graphics pipelines in GRAMPS



Simple scheduler

- **Use graph structure to set simple stage priorities**
 - Could do some dynamic re-prioritization based on queue lengths
- **Only preempt Thread Stages on `reserve/commit` operations**



GRAMPS recap

- **Key abstraction is the computation graph: typed stages and queues**
 - **Thread, custom HW, and “shader” stages**
 - **A few types of queues**
- **Key underlying ideas:**
 - **Enforcing structure on computations is useful for system optimization**
 - **Embrace heterogeneity in application and machine architecture**
 - **Interesting graphics applications have tightly coupled irregular parallelism and regular data parallelism (should be encoded in structure)**
- **Alternative to current design of CUDA/OpenCL**
 - **Gives up almost all structure, it does not providing**

GRAMPS from a graphics perspective

- **We set out to make the graphics pipeline structure programmable**
- **We ended up with a lower level abstraction than today's pipeline: GRAMPS lost domain knowledge of graphics (graphics pipelines are implemented on top of GRAMPS)**
 - **Good: now programmable logic controls the fixed-function logic (in the current graphics pipeline it is the other way around)**
 - **Good: system remains aware of program's overall structure (GRAMPS graph)**
- **Reality: mapping graphics abstractions to GRAMPS abstractions efficiently requires a near expert graphics programmer**
 - **Coming up with the right graph is hard (setting packet sizes, queue sizes has some machine dependence, some key optimizations are global)**

Graphics abstractions today

- **Real-time graphics pipeline is still hanging in there (Direct3D 11 / OpenGL 4)**
- **But lots of algorithm development in OpenCL/Direct3D compute shader/CUDA**
 - **Good: makes GPU compute power accessible. Triggering re-evaluation of best practices in field**
 - **Bad: community shifting too-far toward only thinking about current GPU-style data-parallelism**
- **CPU+GPU fusion is begging for alternative high-level frameworks for interactive graphics**
 - **Example: NVIDIA Optix: new framework for ray tracing**
 - **Application provides key kernels, Optix compiler/runtimes schedules**
 - **Built on top of CUDA**