Lecture 11:
“GPGPU” computing and the CUDA/OpenCL Programming Model

Kayvon Fatahalian
CMU 15-869: Graphics and Imaging Architectures (Fall 2011)

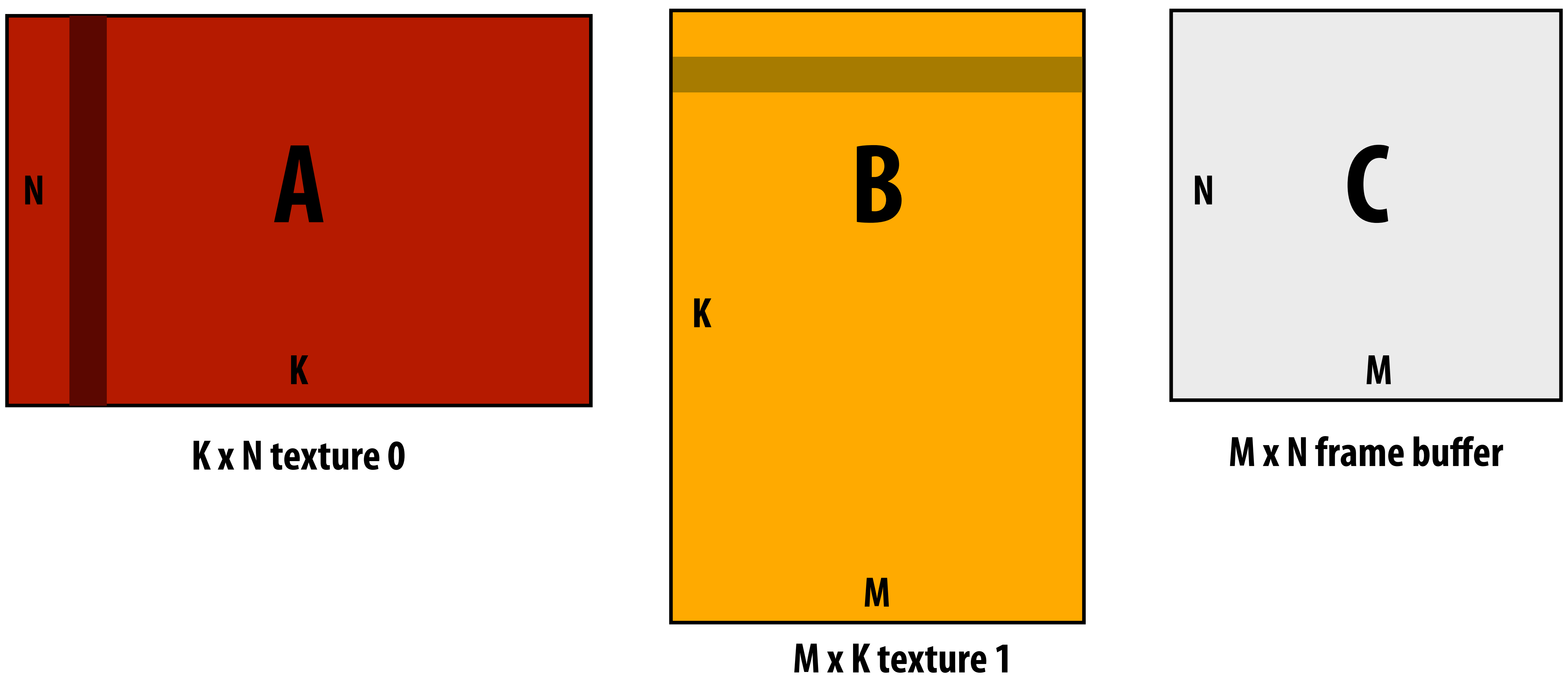
Today

- Some GPGPU history
- The CUDA (or OpenCL) programming model
- (if time) GRAMPS: An attempt to create programmable graphics pipelines

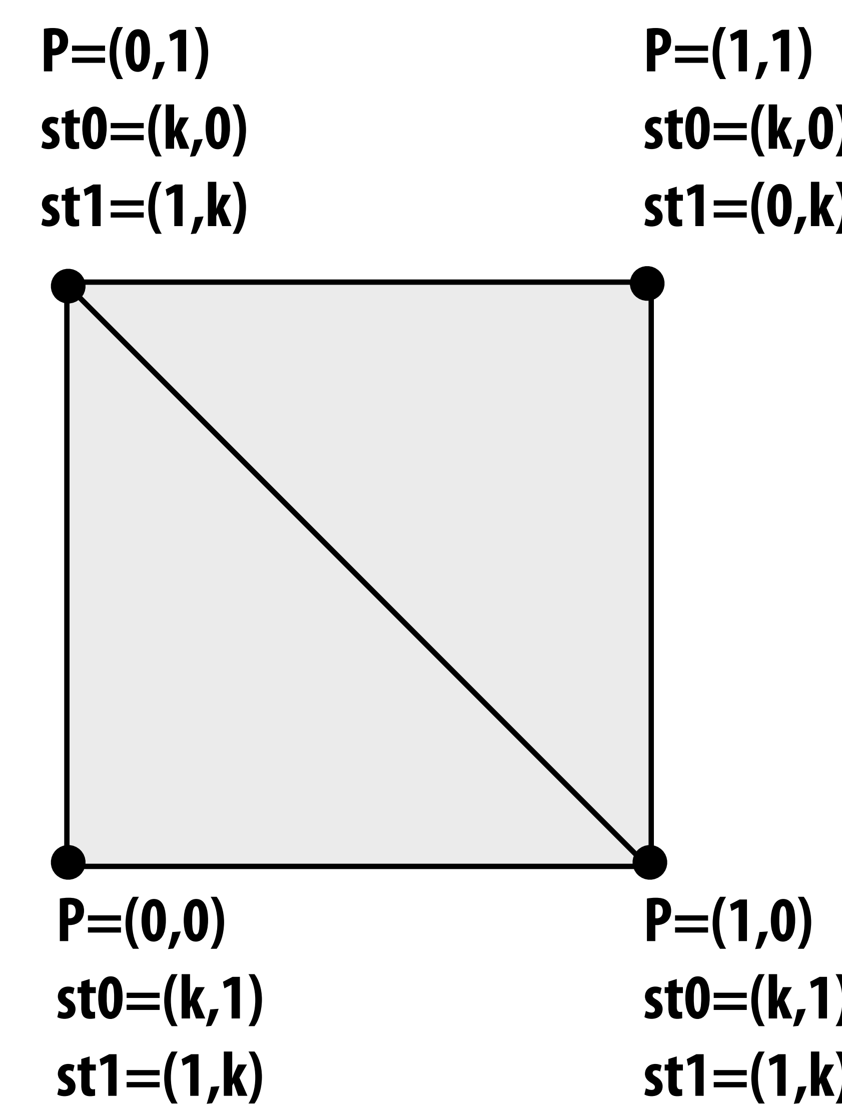
Early GPU-based scientific computation

Dense matrix-matrix multiplication

[Larson and McAllister, SC 2001]

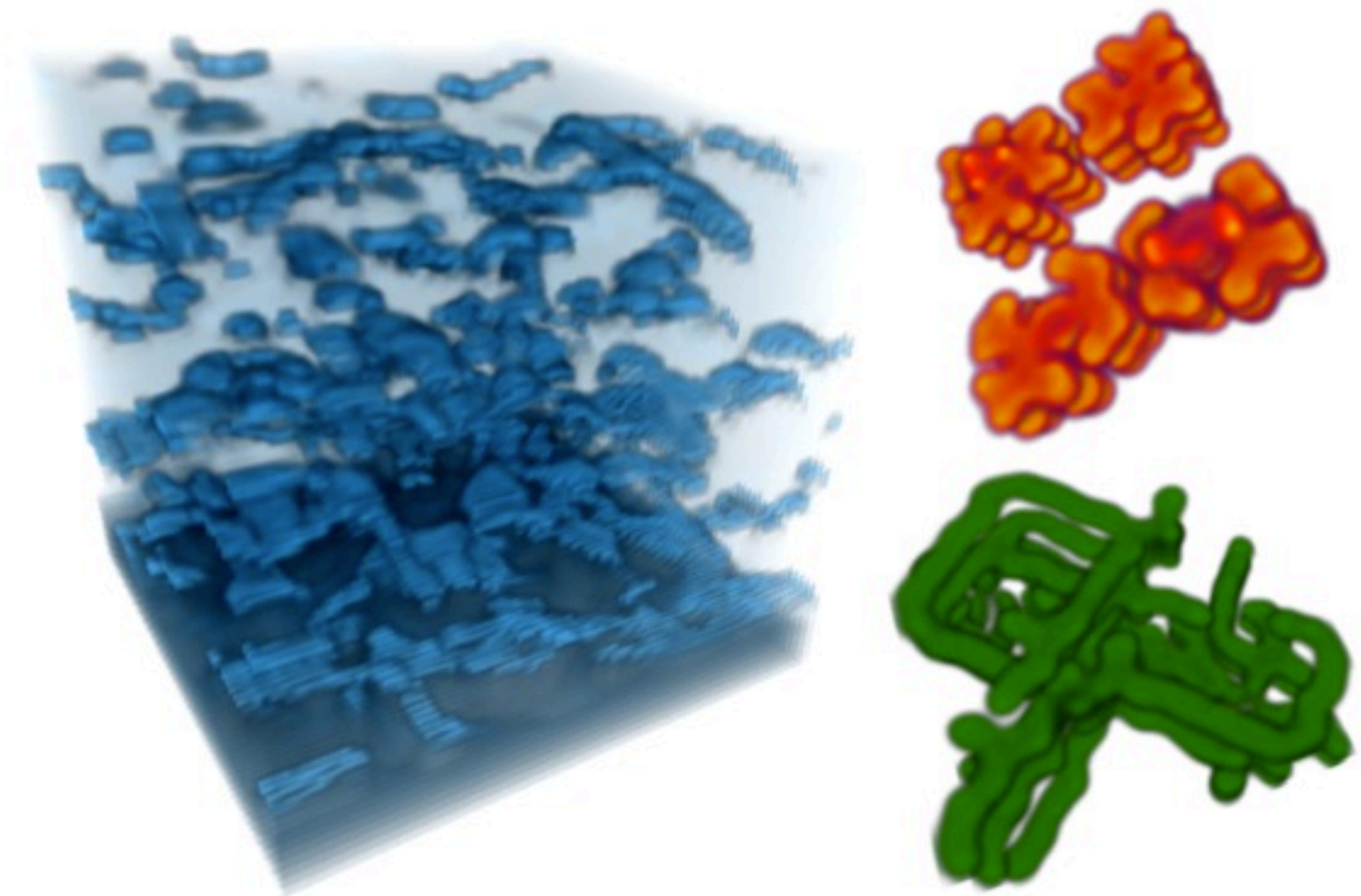


Set frame buffer blend mode to ADD
for k=0 to K
    Set texture coords
    Render 1 full-screen quadrilateral

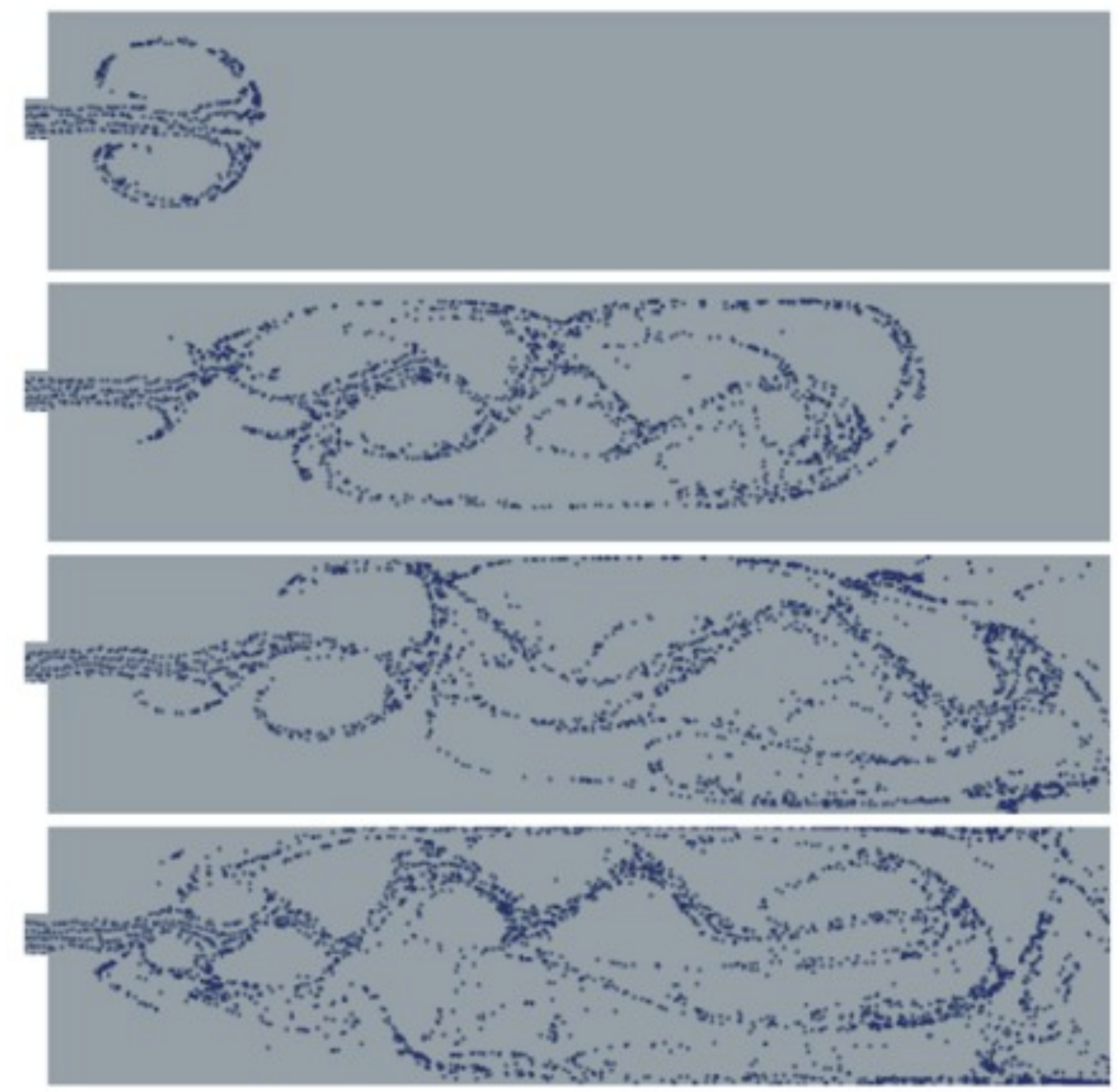


Note: this work followed [Percy 00], which modeled OpenGL with multi-texturing as a SIMD processor for multi-pass rendering (we discussed this last time in the shade-tree example)

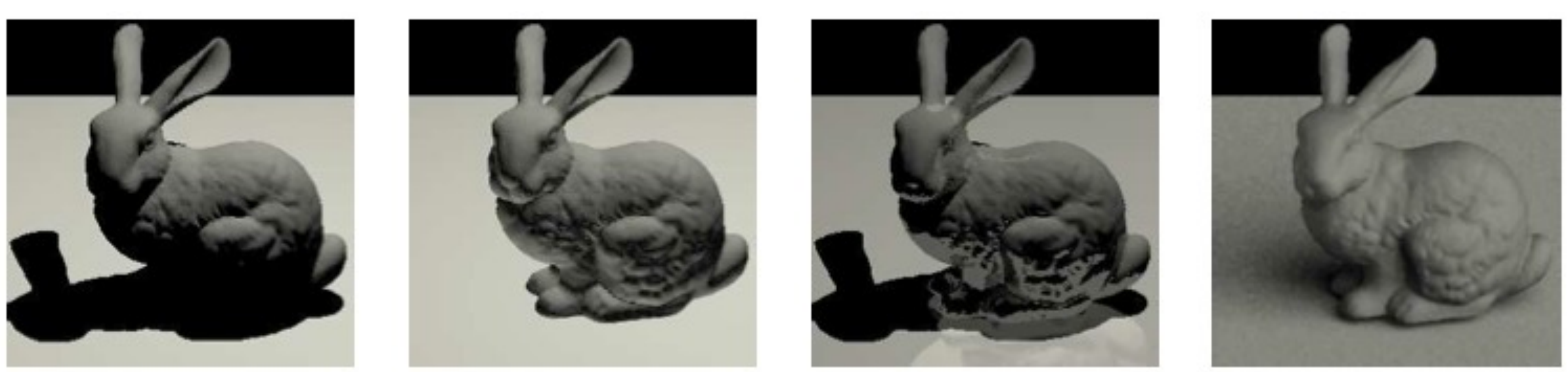
"GPGPU" 2002-2003



Coupled Map Lattice Simulation [Harris 02]



Sparse Matrix Solvers [Bolz 03]



Ray Tracing on Programmable Graphics Hardware [Purcell 02]

Brook for GPUs [Buck 04]

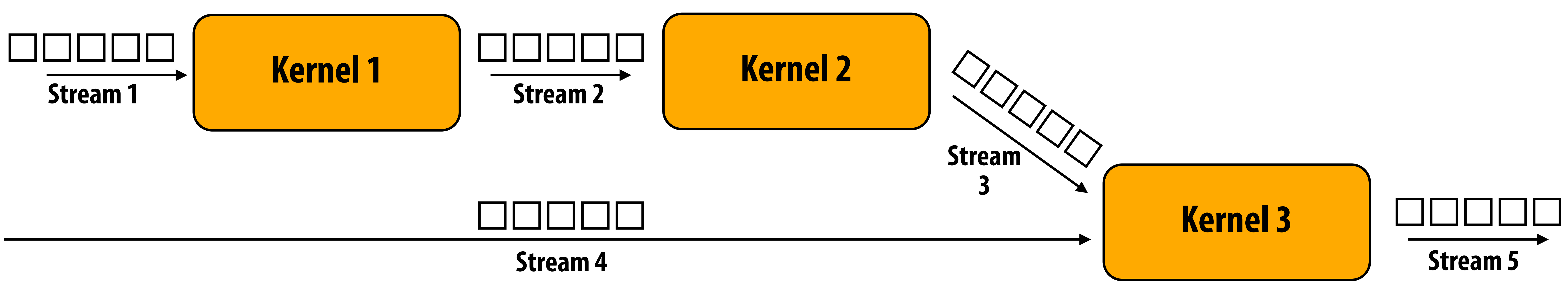
- Abstract GPU as a generic stream processor (C extension)
  - Streams: 1D, 2D arrays of data
  - Kernels: per-element processing of stream data **
  - Reductions: stream --> scalar
- Influences
  - Data-parallel programing: ZPL, Nesl
  - Stream programming: StreaMIT, StreamC/Kernel
- Brook runtime generates appropriate OpenGL calls

```
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);
```

** Broke traditional stream processing model with in-kernel gather (more on this later)

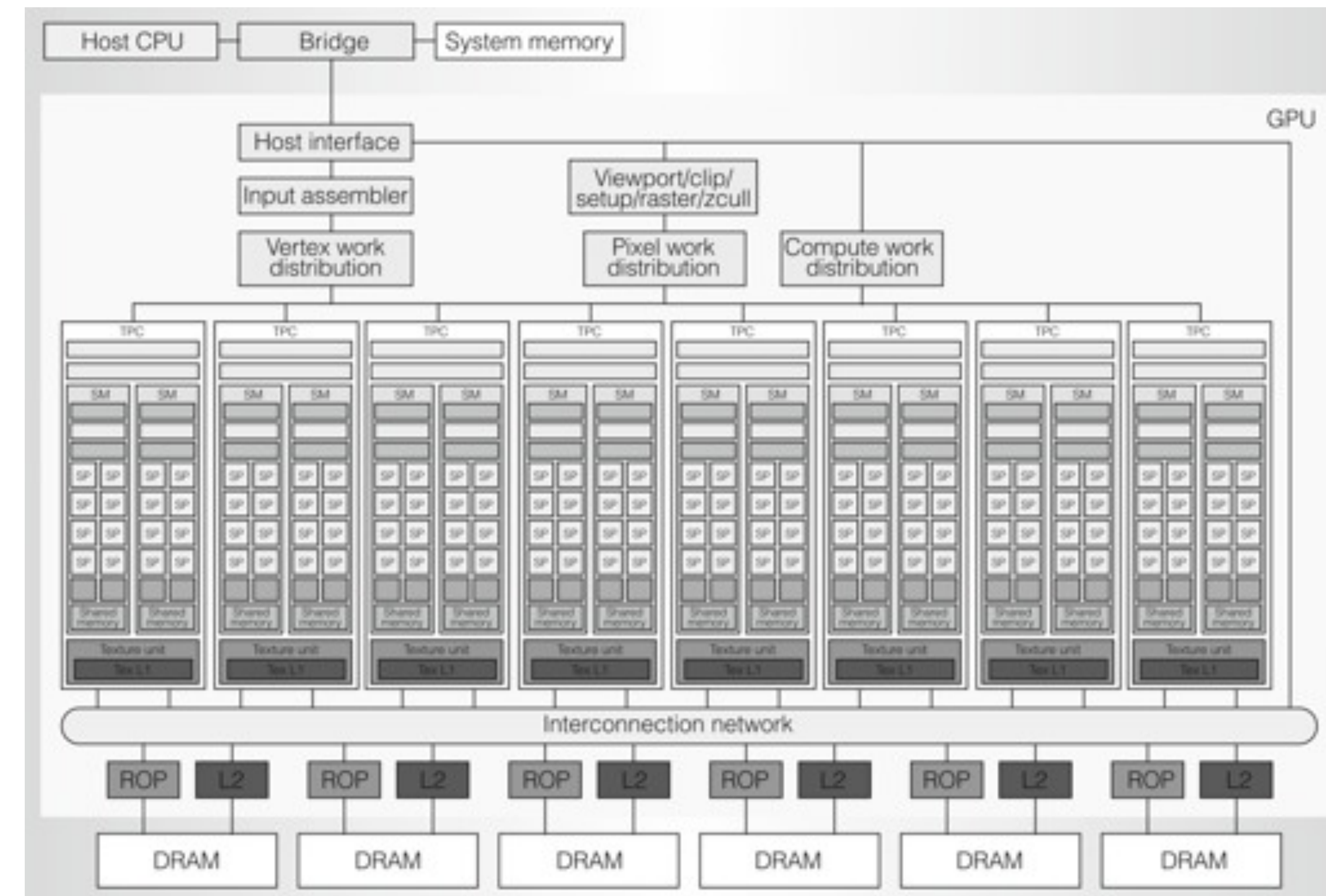
Stream programming ("pure")

- Streams
  - Encapsulate per-element parallelism
  - Encapsulate producer-consumer locality
- Kernels
  - Functions (side-effect-free)
  - 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**
  - Recall: Tesla was first “unified shading” GPU



- **Low level, reflects capabilities of hardware**
  - Recall arguments in Cg paper
  - Combines some elements of streaming, some of threading (like HW does)

- **Today: 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

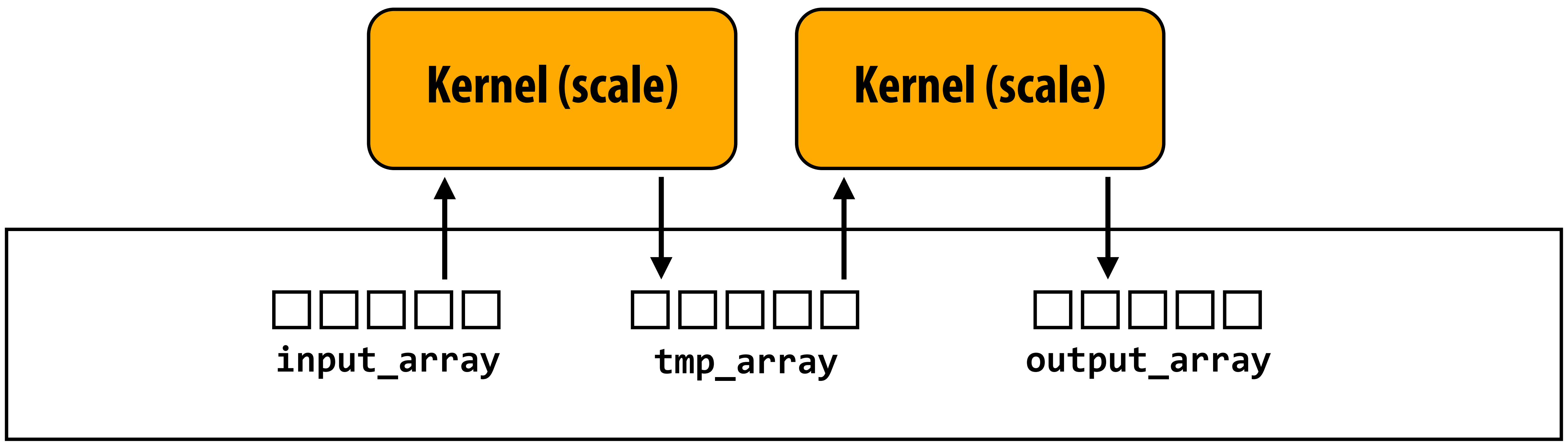
What is the behavior of this kernel?

```
// 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 find producer-consumer?

```
// 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.

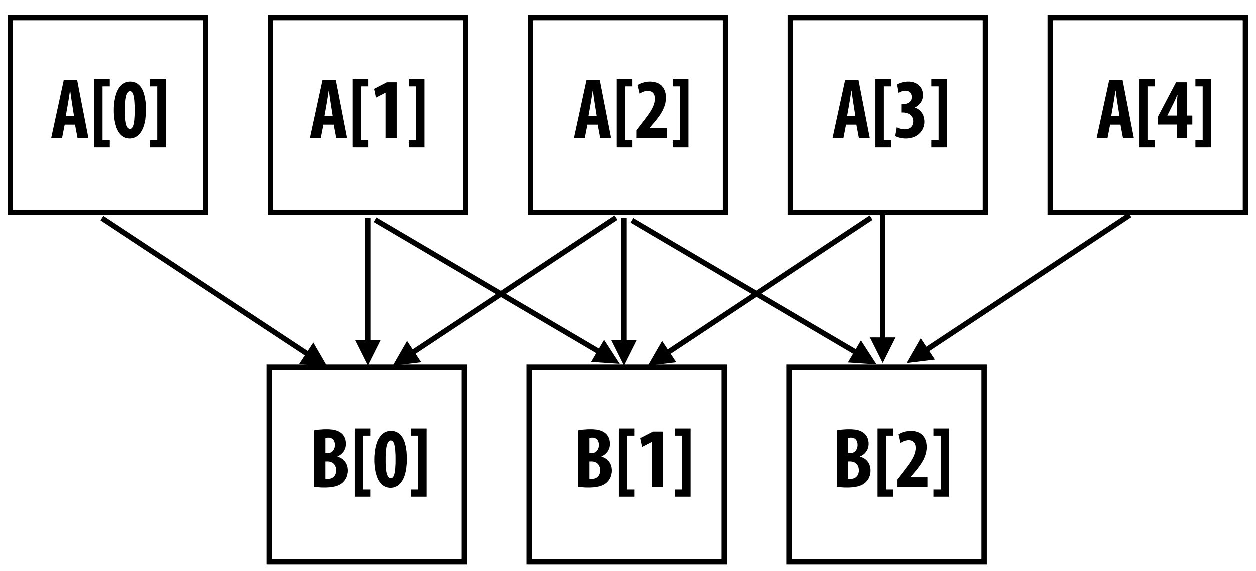
(programmer just has to do a little indexing in the kernel to get a reference to stream inputs/outputs)

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)



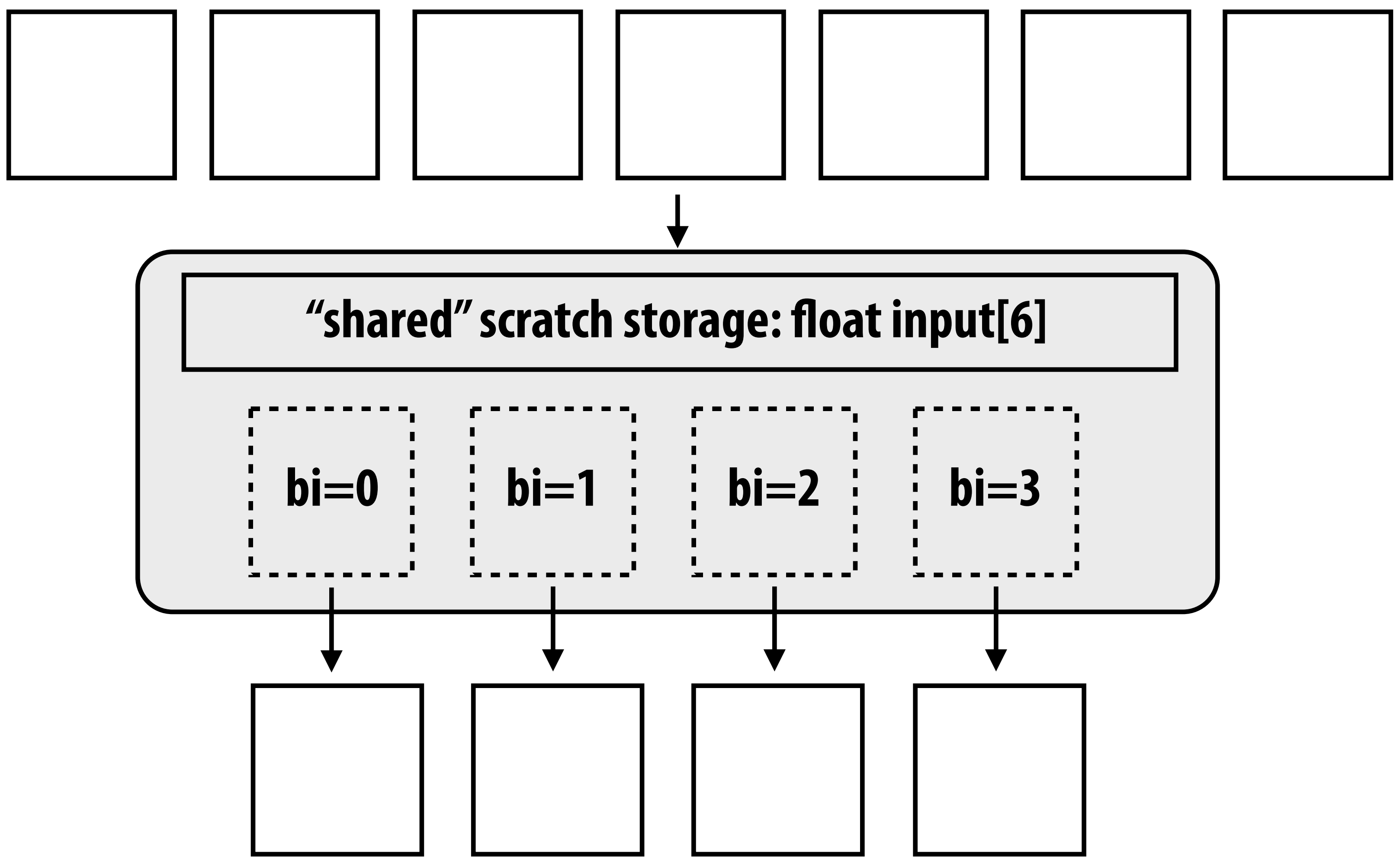
CUDA thread hierarchy

```
#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 <u>not</u> independent.
They can cooperate to process shared data.

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



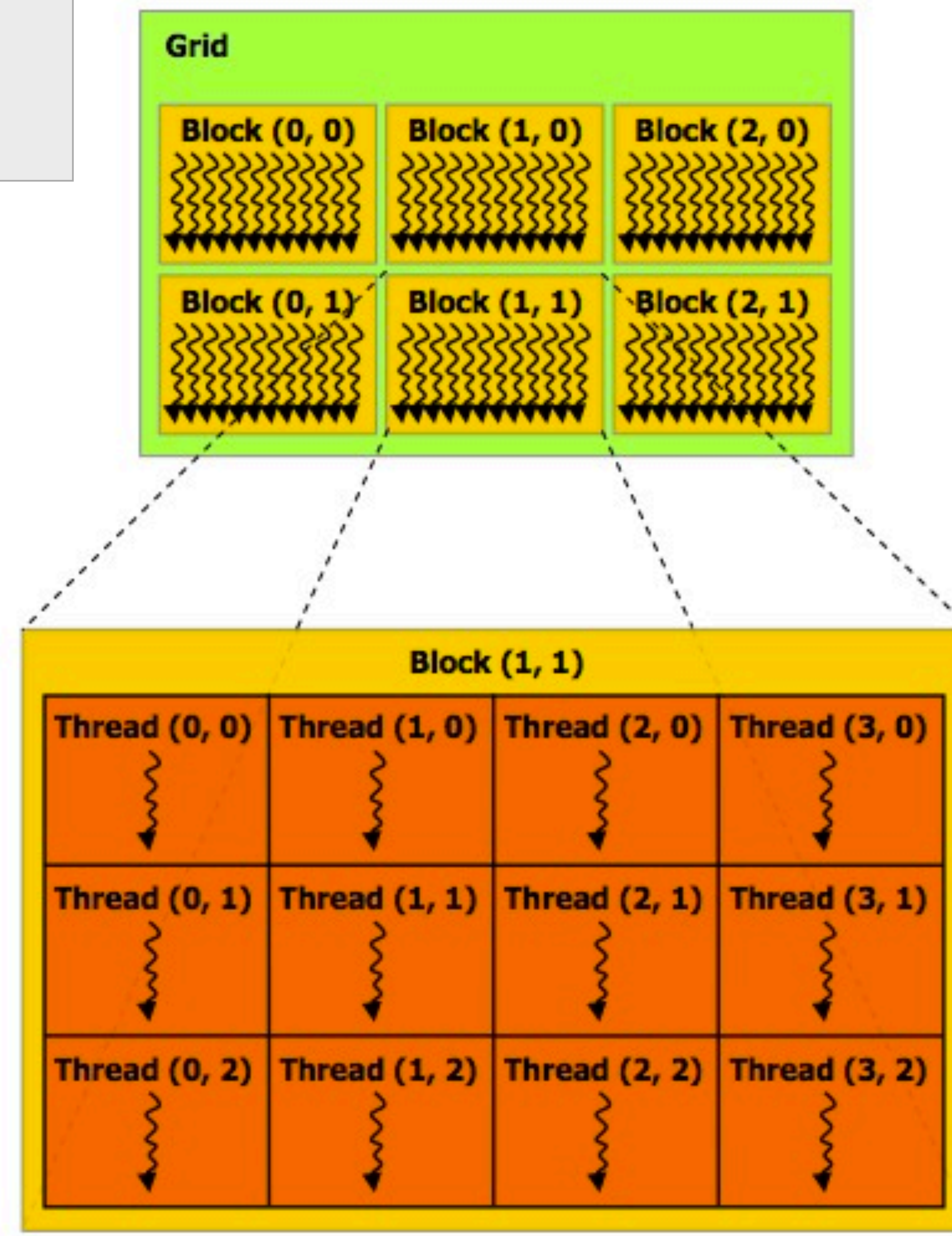
CUDA thread hierarchy

```
// 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 1D, 2D, 3D
(Convenience: many CUDA programs operate on n-D grids)

Thread blocks 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)

- **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)****

** Using 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
  - Achieves 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?

“Persistent” threads

- **No semblance of streaming at all any more**
- **Programmer is always thinking explicitly parallel**
- **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)

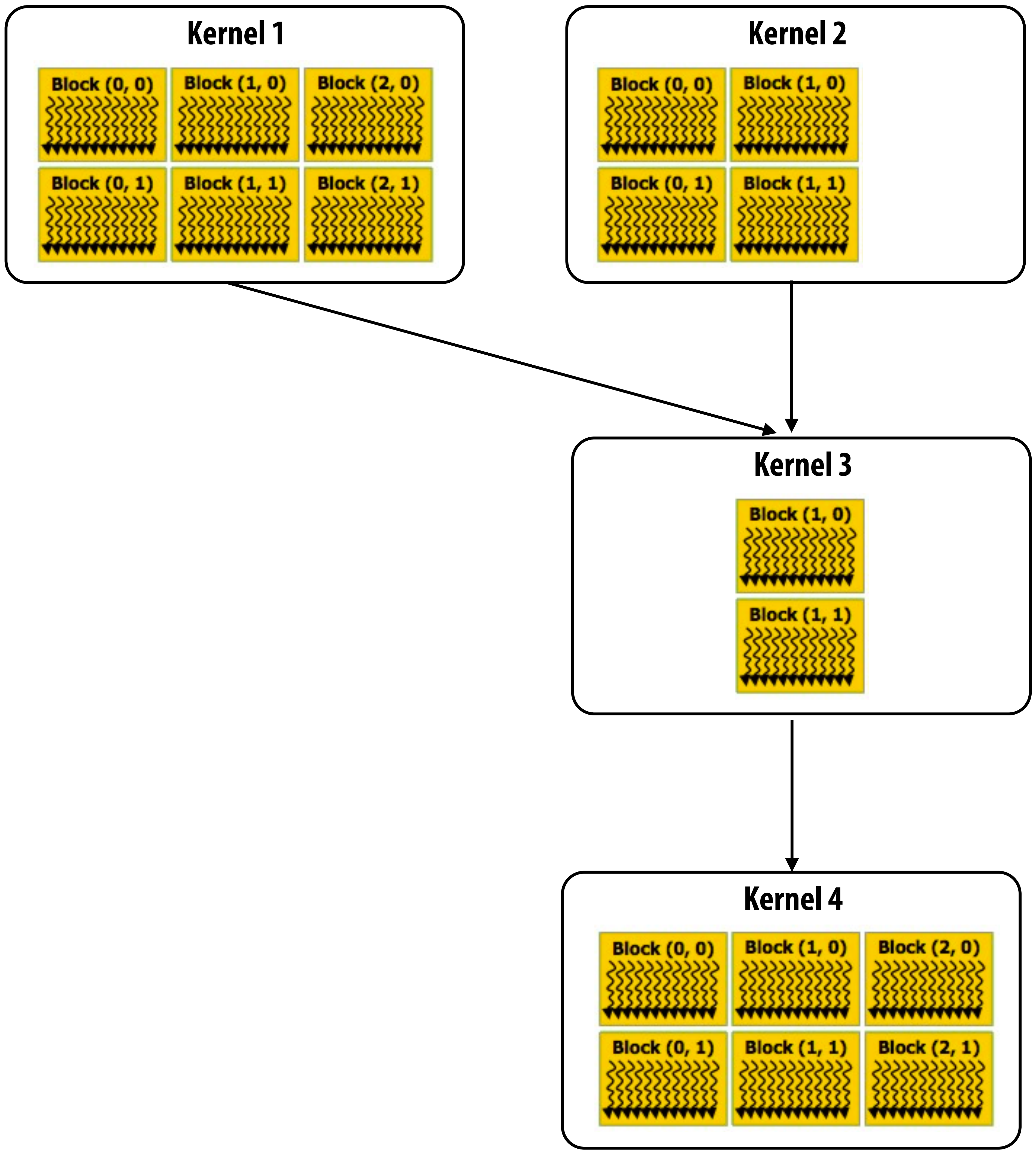
1. Provides access to a cheap, very fast machine

2. SPMD abstraction allows programmer to write scalar code, have it (almost trivially) mapped to vector hardware

Note: Five years later... one Intel employee (with LLVM and a graphics background)

3. More like thread programming than streaming: arbitrary in-kernel gather (+ 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



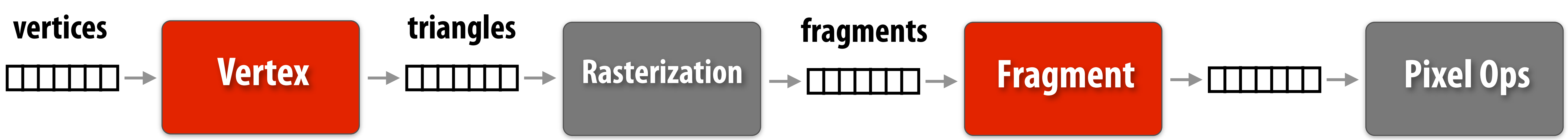
Note: arrows are specified dependencies between batch thread launches

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

Part 2: Programmable Pipeline

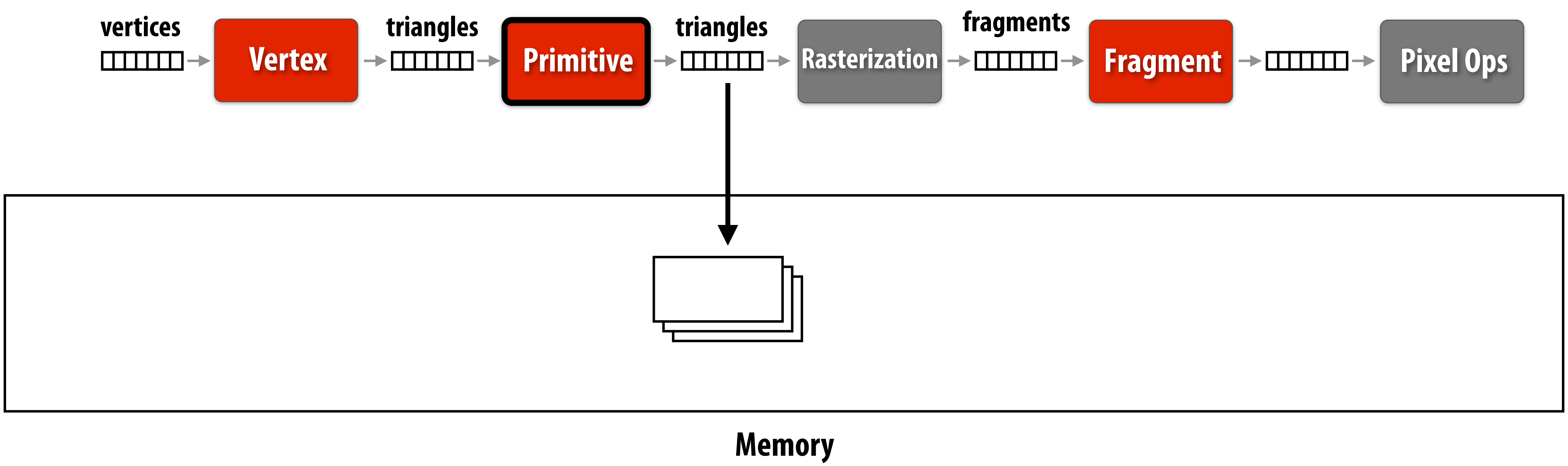
(Programmable Pipeline Structure, Not Programmable Stages)

Graphics pipeline pre Direct3D 10



Graphics pipeline circa 2007

[Blythe, Direct3D 10]

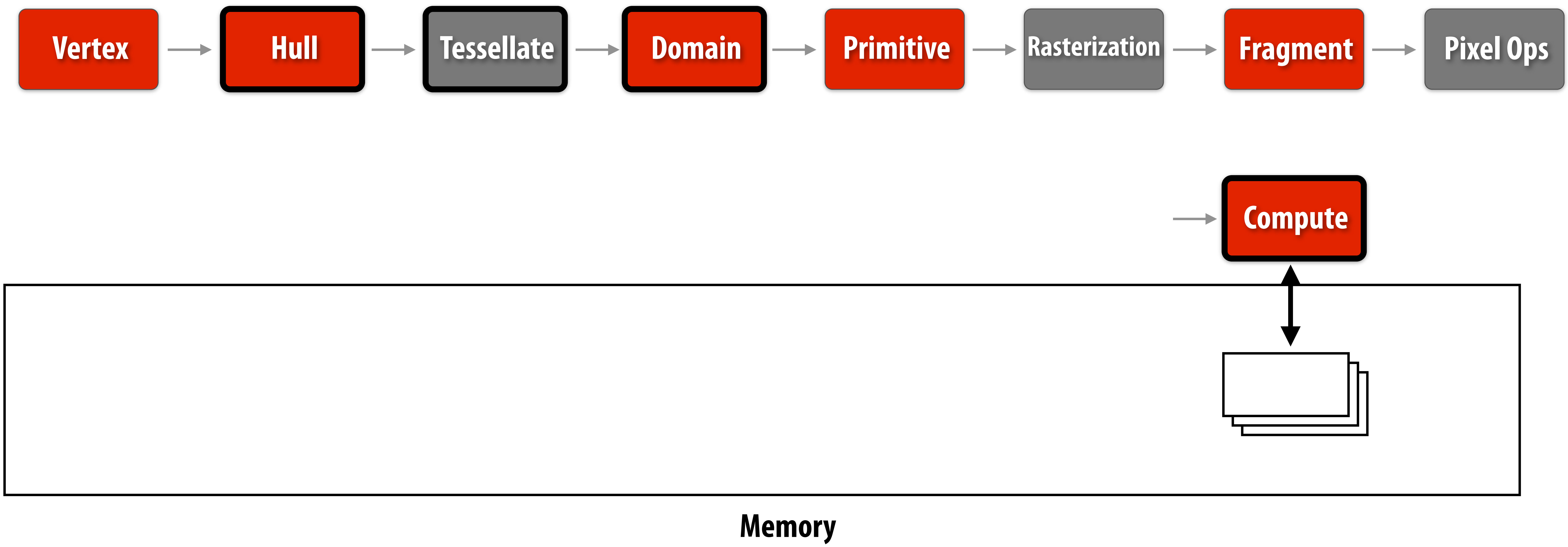


Added new stage

Added ability to dump intermediate results out to memory for reuse

Pipeline circa 2010

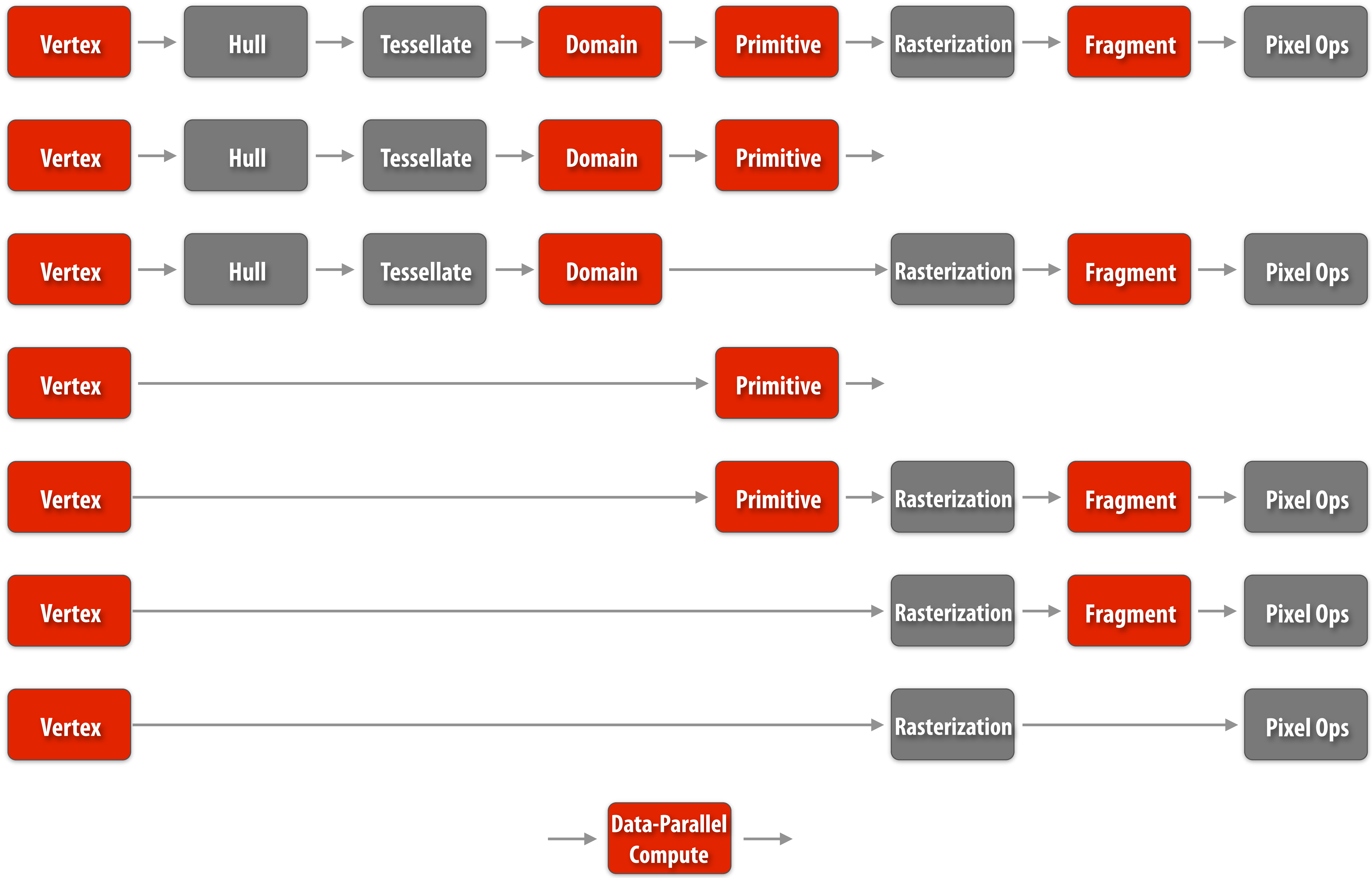
[Direct3D 11, OpenGL 4]



Added <u>three</u> new stages (new data flows needed to support high-quality surfaces)

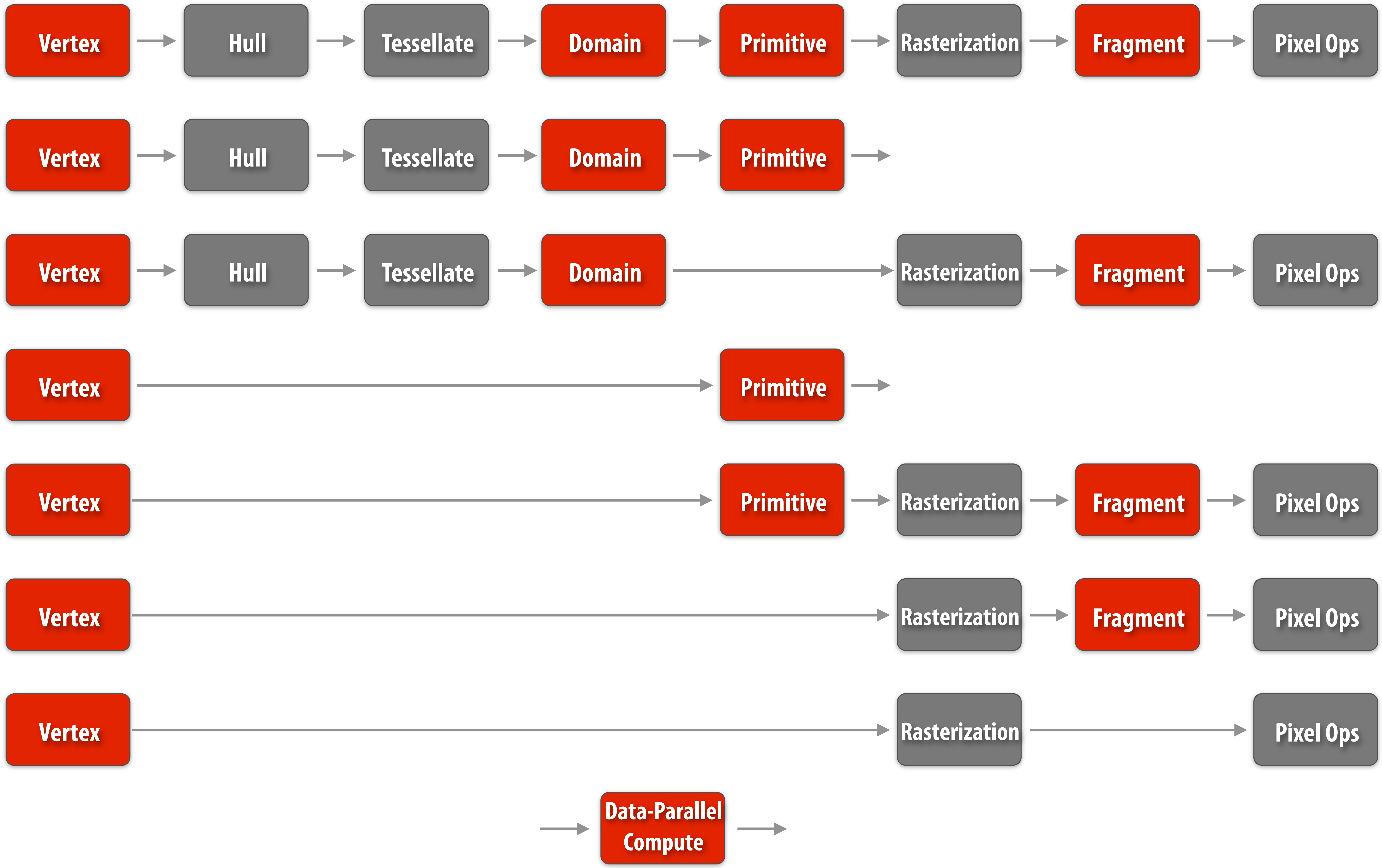
Forked off a separate 1-stage pipeline (a.k.a. "OpenCL/CUDA)
(with relaxed data-access and communication/sync rules)

Modern graphics pipeline: highly configurable structure



Direct3D 11, OpenGL 4 pipeline configurations

Modern graphics pipeline: highly configurable structure



Direct3D 11, OpenGL 4 pipeline configurations

Kayvon's Micropolygon Rendering Pipeline

[Fatahalian 09, Fisher 09, Fatahalian 10, Boulos 10, Brunhaver 10]

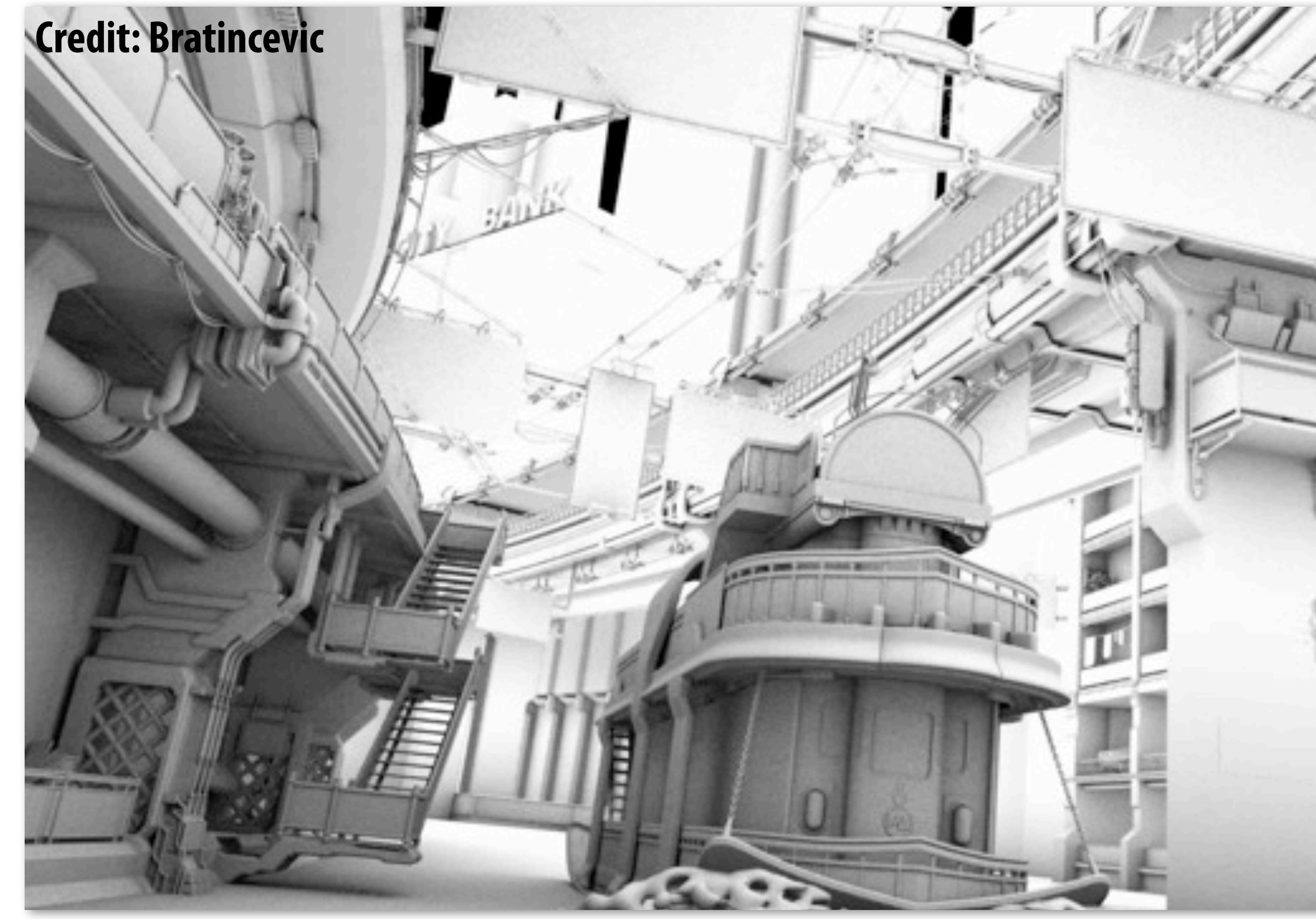
Current realities / trends in interactive graphics

- 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

Global illumination algorithms

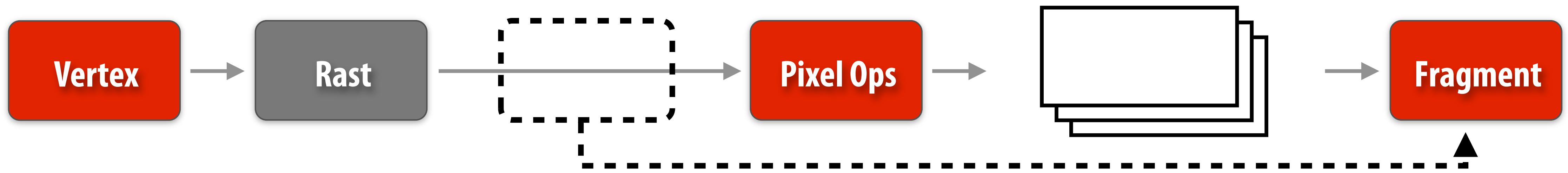

Credit: NVIDIA


Credit: Bratincevic

Credit: Ingo Wald

Ray tracing:
for accurate reflections, shadows

Alternative shading structures ("deferred shading")



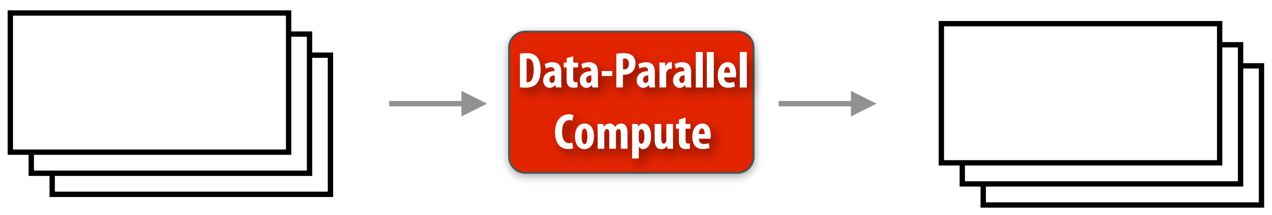
For more efficient scaling to many lights (1000 lights, [Andersson 09])

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

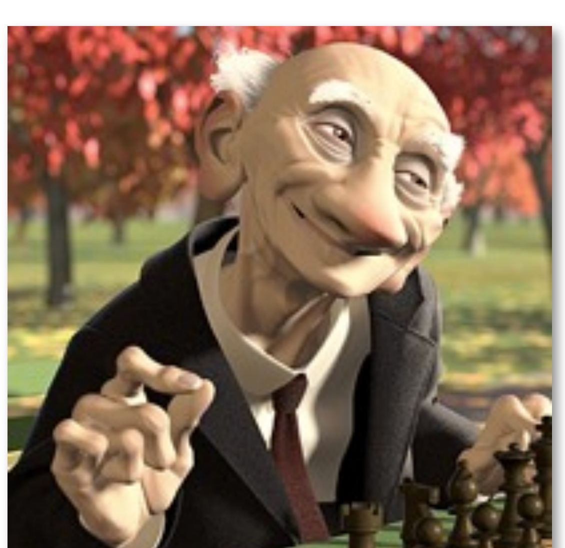


CUDA / OpenCL / ComputeShader

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]

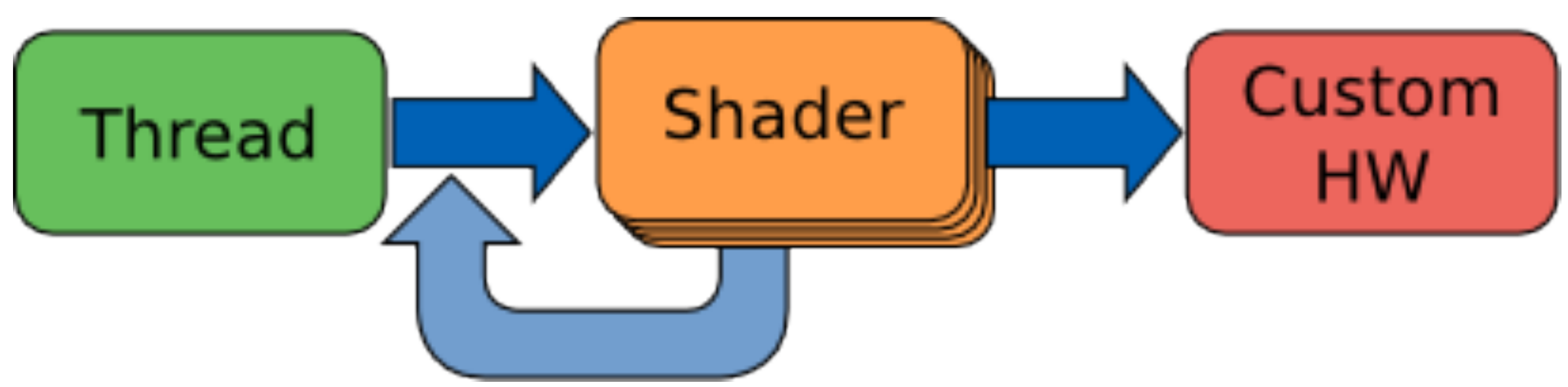
GRAMPS programming system: goals

- **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)

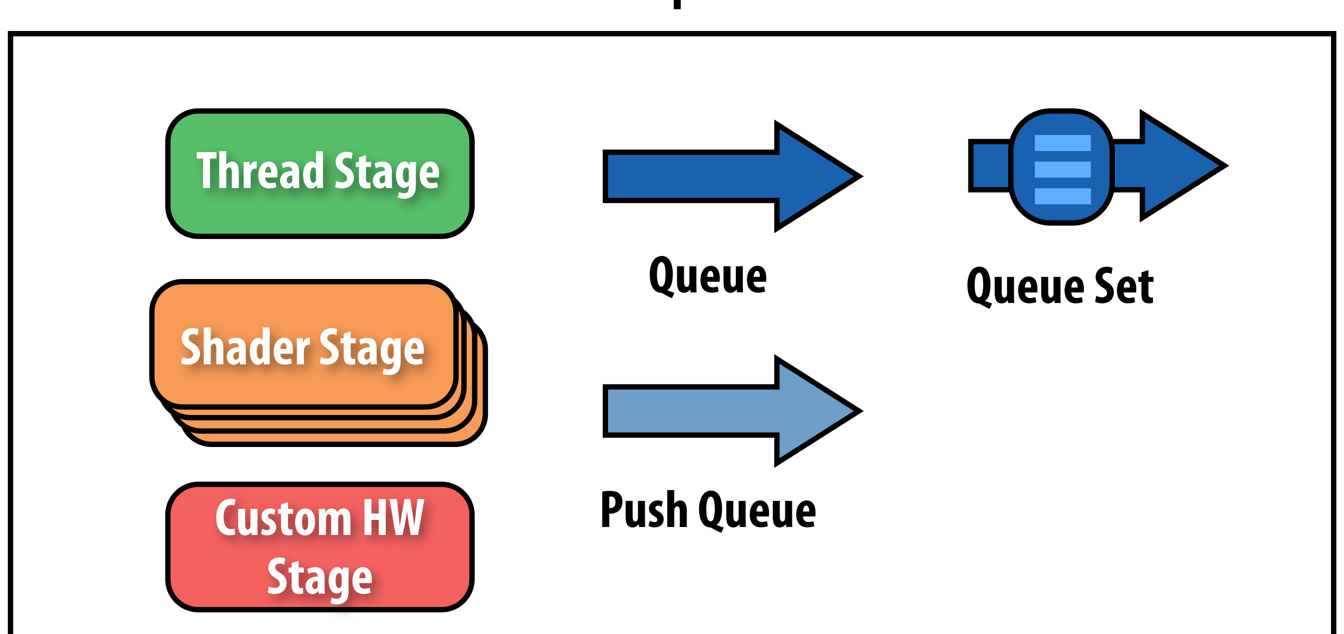
- **High performance: target future GPUs (embrace heterogeneity)**
  - Throughput ("accelerator") processing cores
  - Traditional CPU-like processing cores
  - Fixed-function units

GRAMPS overview

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

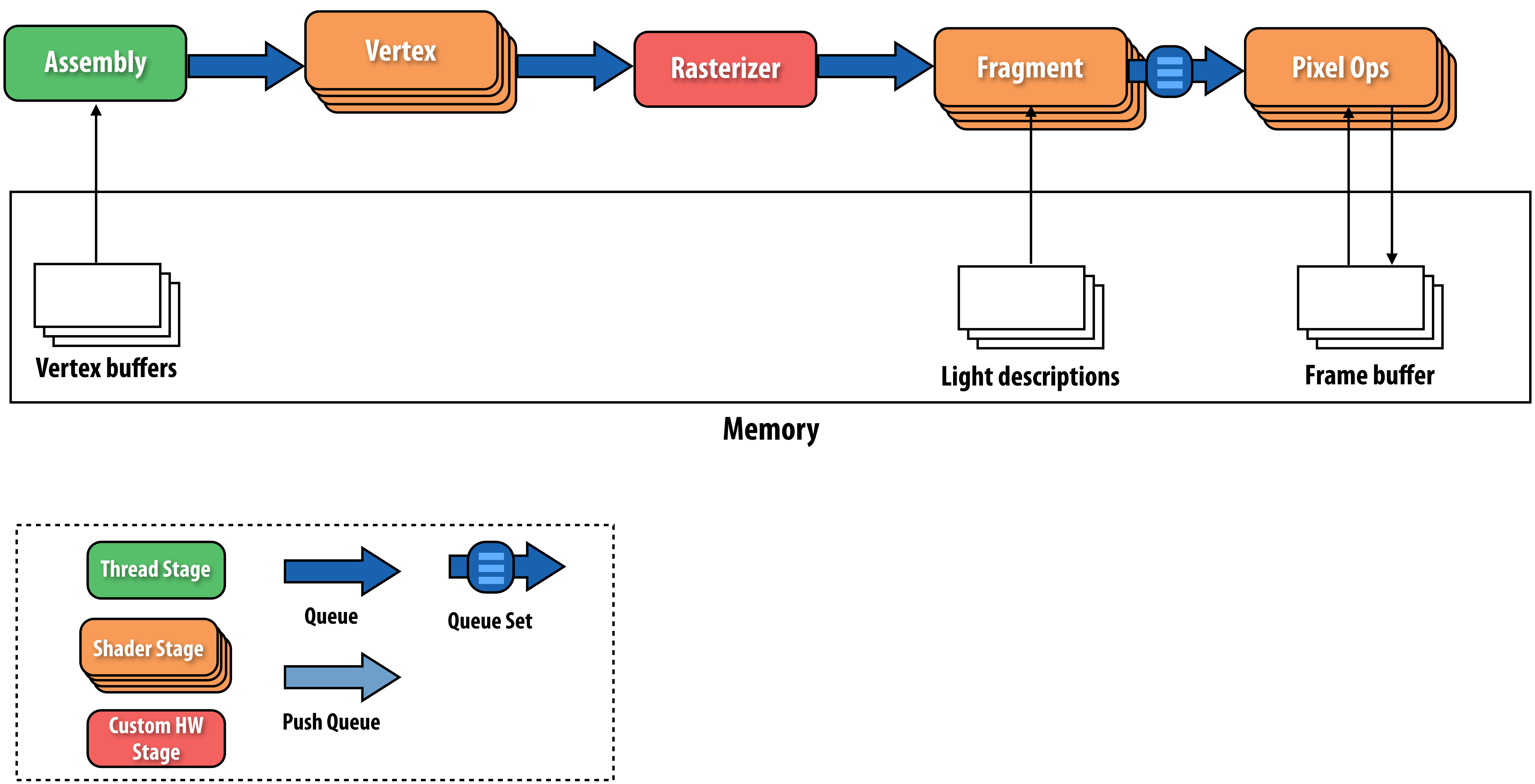


GRAMPS primitives



Writing a GRAMPS program

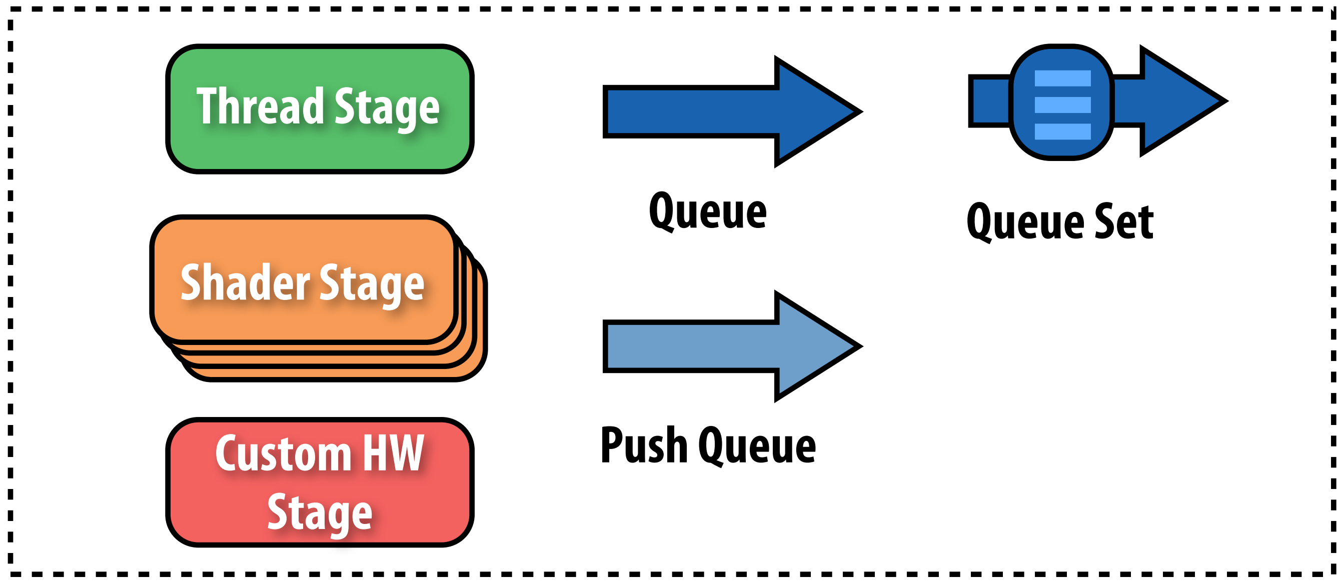
1. Design application graph and queues
2. Implement the stages
3. Instantiate graph and launch



Queues

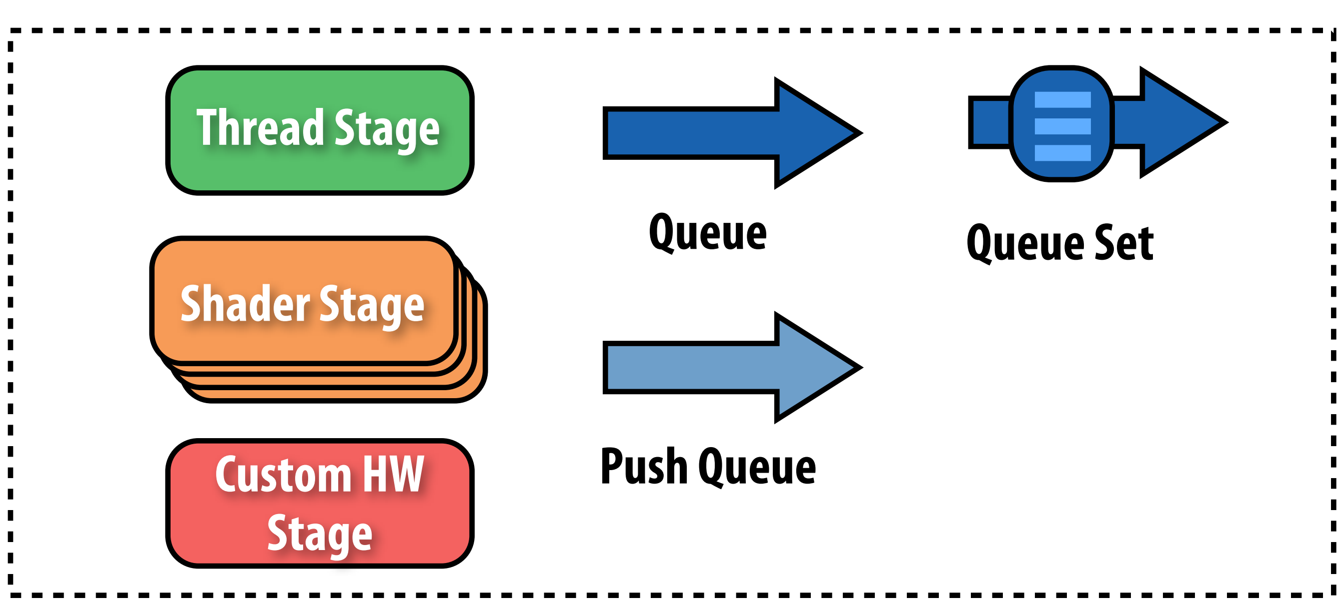
- Bounded size, operate at granularity of "packets" (structs)
  - Packets are either:
    1. Completely opaque to system
    2. Header + array of opaque elements

- Queues are optionally FIFOs (to preserve ordering)



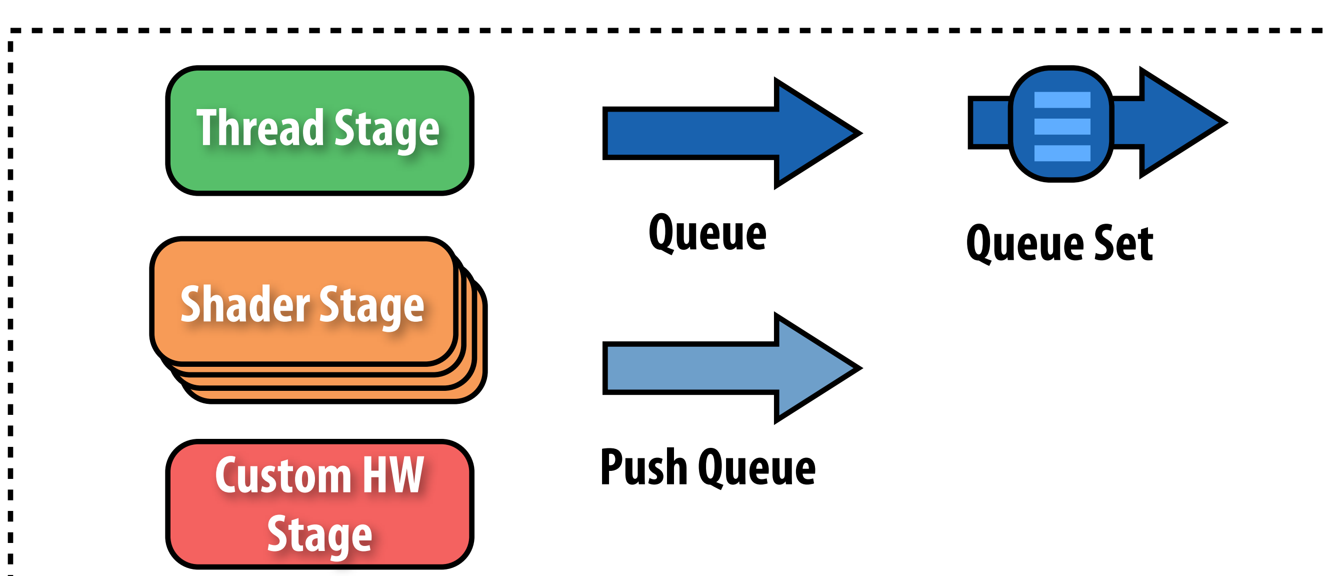
"Thread" and custom HW stages

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



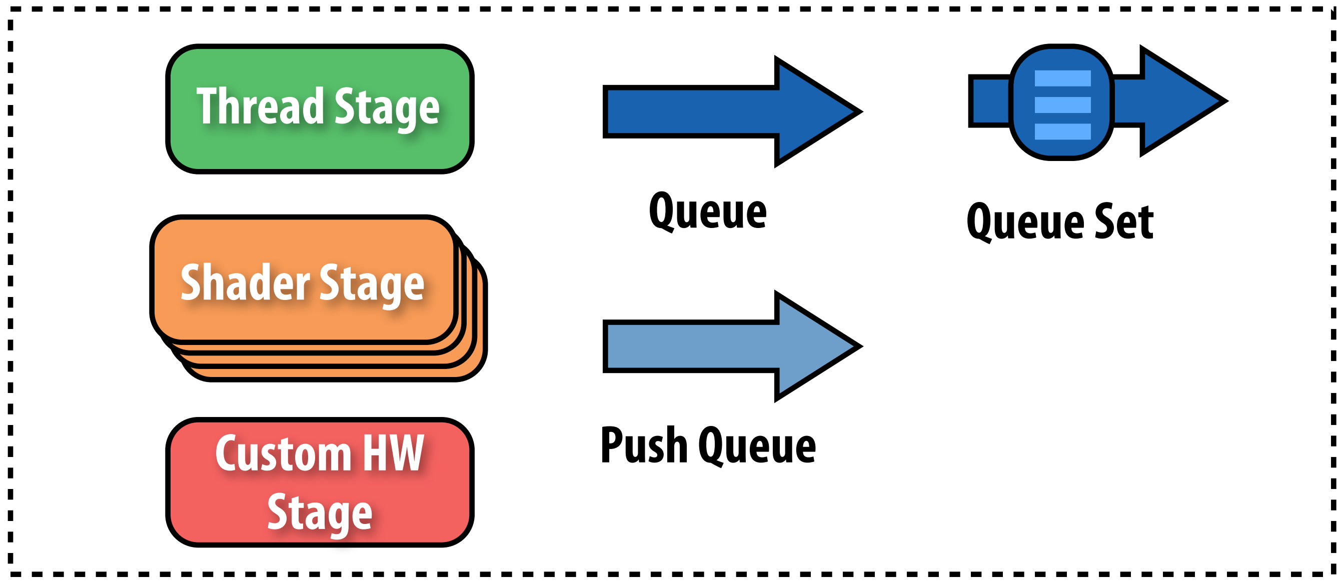
“Shader” stages

- 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)

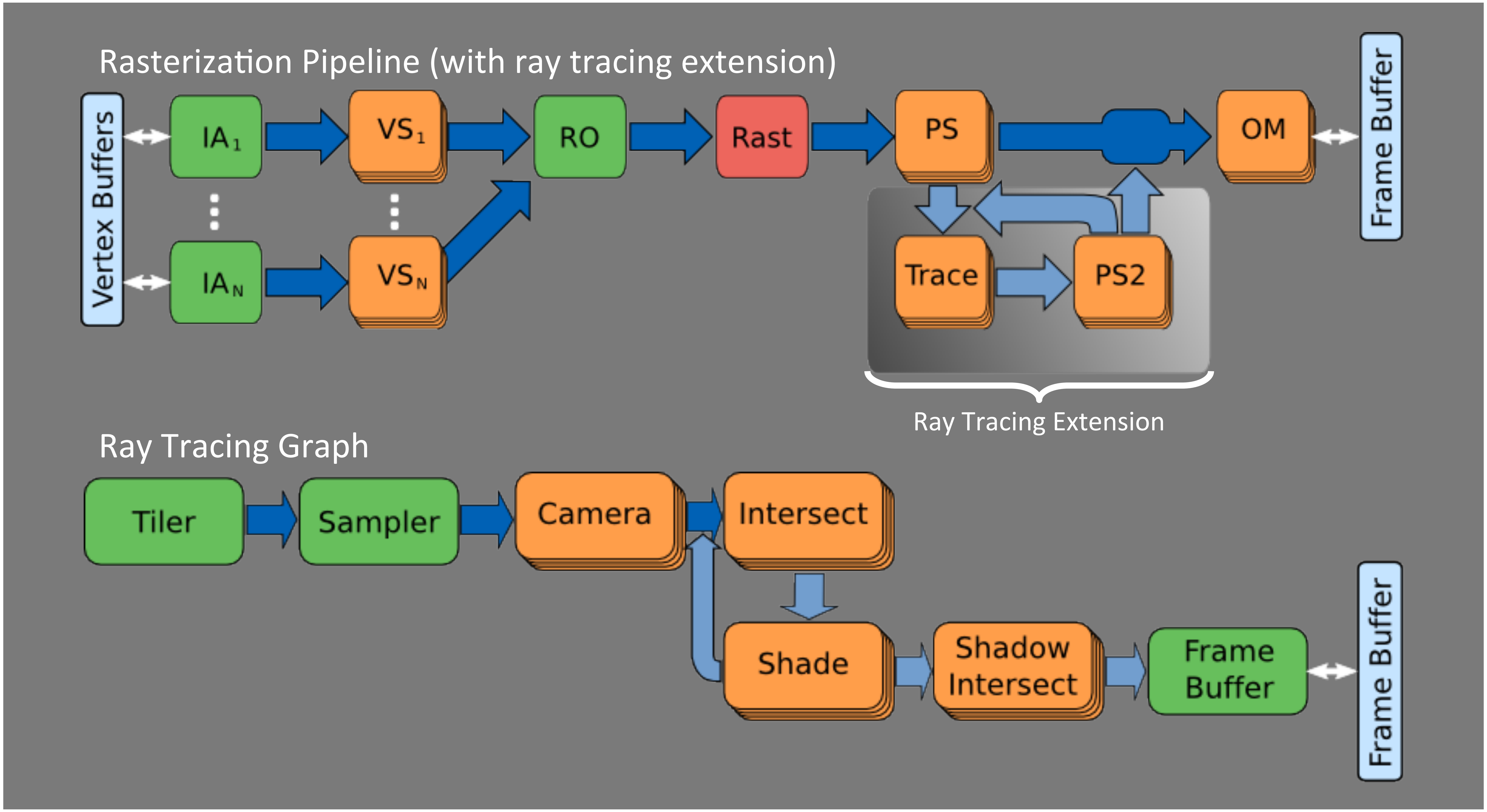


Queue sets (for mutual exclusion)

- 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)

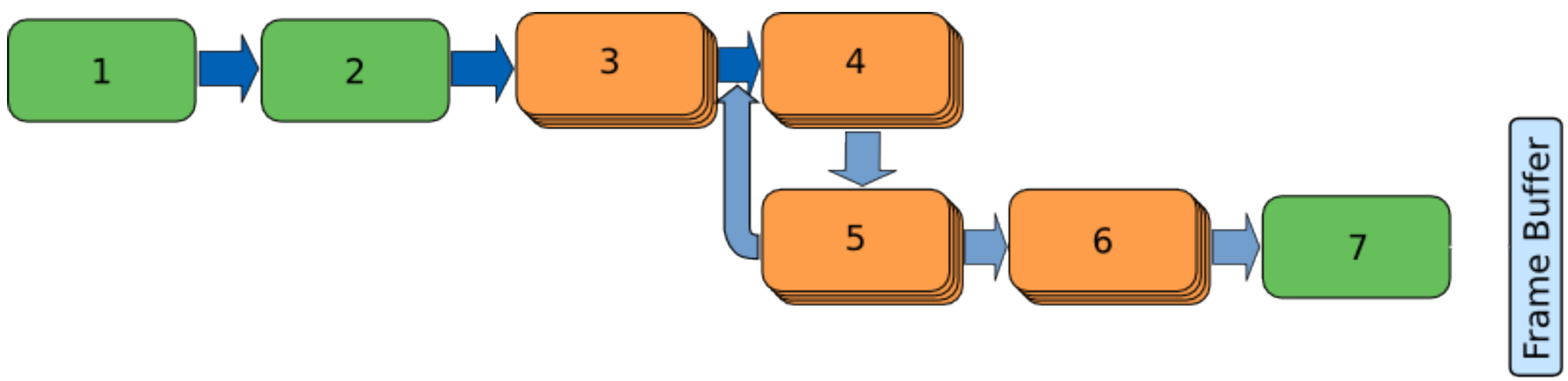


Graphics pipelines in GRAMPS



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:
  - Structure is good
  - Embrace heterogeneity in application and in target architecture
    - Interesting graphics apps have tightly coupled irregular parallelism and regular data parallelism (should be encoded in structure)

- Alternative to current design of CUDA/OpenCL
  - They are giving up structure, not providing it

GRAMPS from a graphics perspective

- Set out to make graphics pipeline structure programmable

- Result: Lower level abstraction than today's pipeline: 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)

- Experience: mapping key graphics abstractions to GRAMPS abstractions efficiently requires a knowledgable 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 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 will likely trigger emergence of alternative high-level frameworks for niches in interactive graphics
  - Example: NVIDIA Optix: new framework for ray tracing
    - Application provides key kernels, Optix compiler/runtimes schedules
    - Built on CUDA