Alternative Rendering Pipelines Using NVIDIA CUDA

Andrei Tatarinov

Alexander Kharlamov

NVIDIA

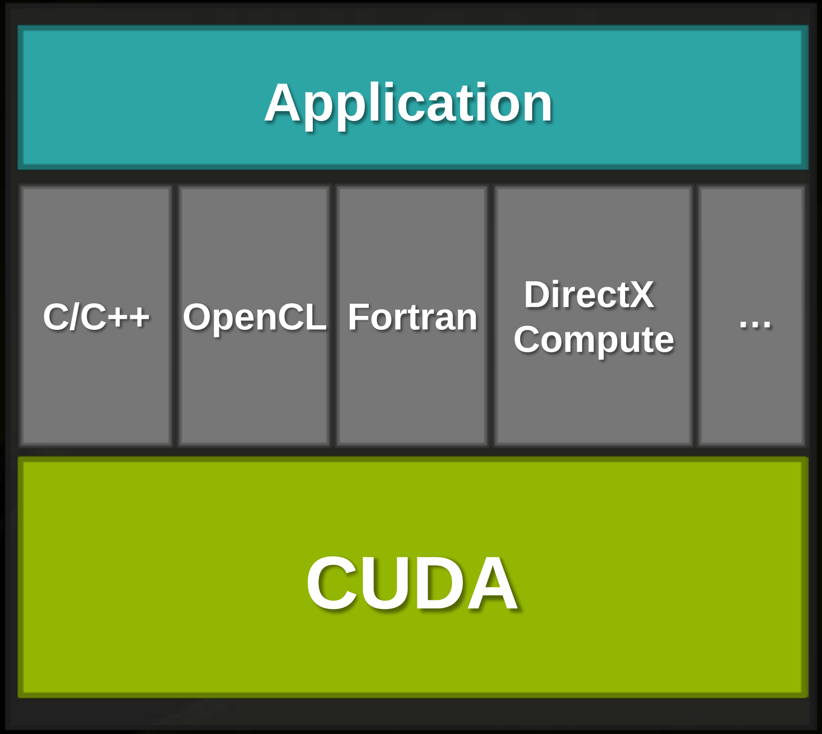
Outline

- CUDA overview
- Ray-tracing
- REYES pipeline
- Future ideas

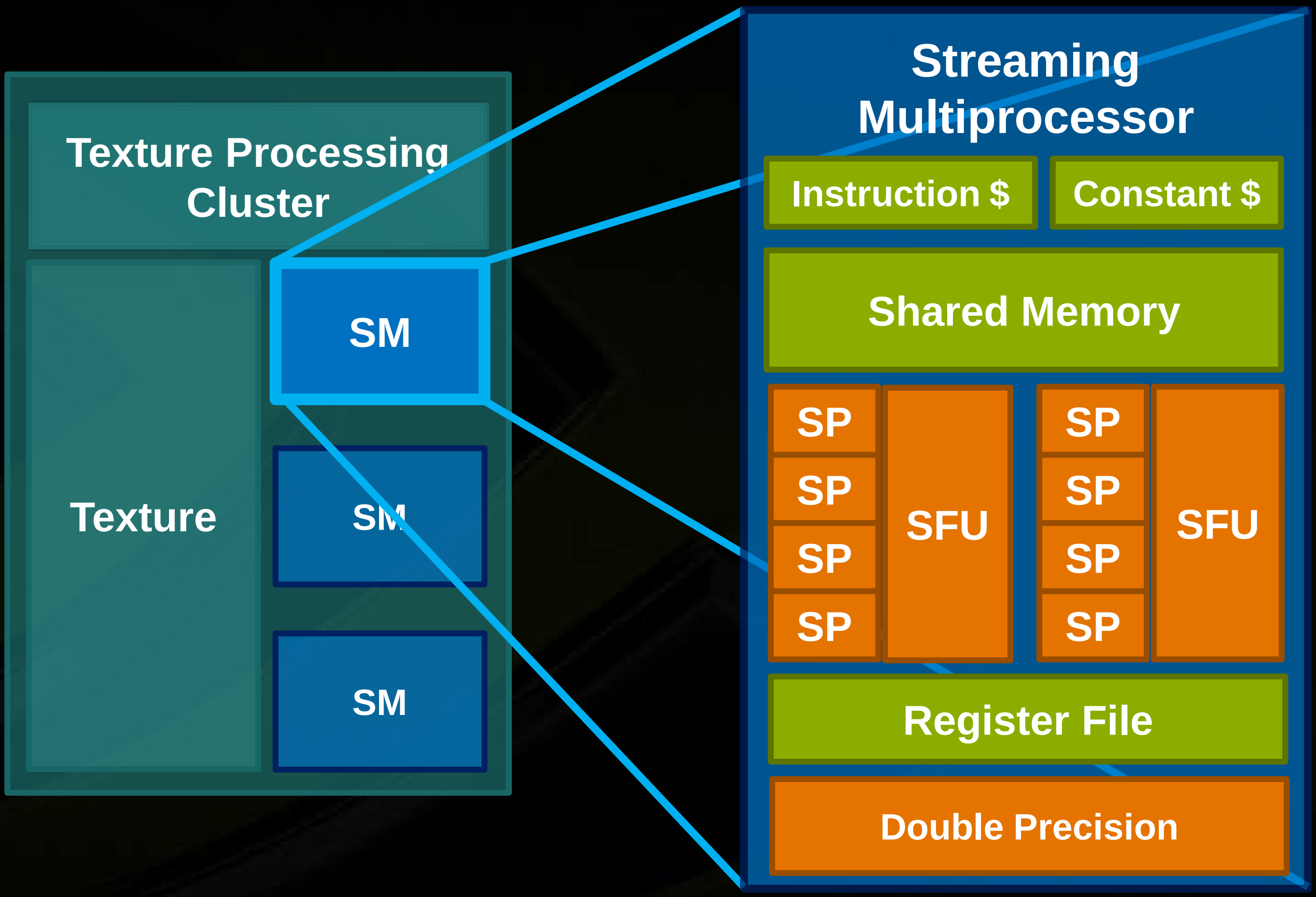
CUDA Overview

Compute Unified Device Architecture (CUDA)

- Parallel computing architecture
- Allows easy access to GPU
- A back-end for different APIs

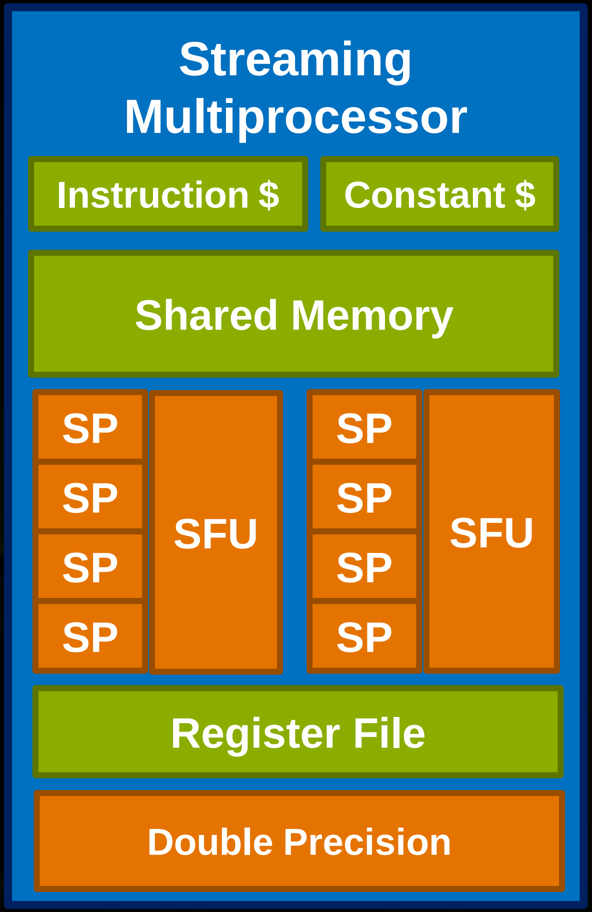


Streaming Multiprocessor



Threads and Blocks

- One block is executed on one SM
- Threads within a block can cooperate
  - Shared memory
  - *__syncthreads()*



Multiprocessor Occupancy

- Registers (r.) & Threads
  - 8192 r. per Streaming Multiprocessor on 8800GTX

128 r. – way too many registers

r. ≤ 40: 6 active warps

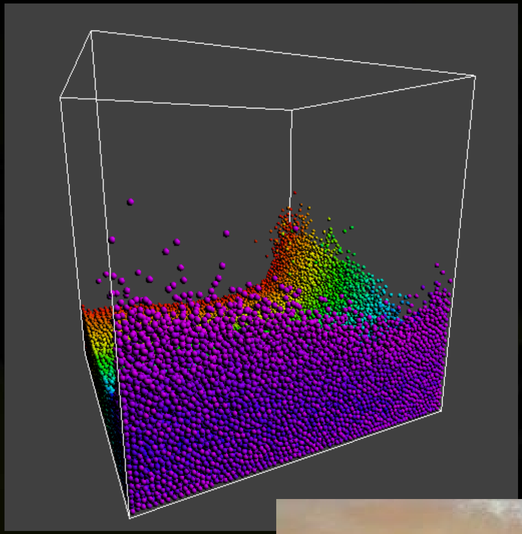
r. ≤ 32: 8 active warps

r. ≤ 24: 10 active warps

r. ≤ 20: 12 active warps

r. ≤ 16: 16 active warps

Usecases



Ray tracing

Ray tracing

- Natural rendering pipeline
- Important tool for determining visibility

Research goals

- Investigate rendering pipelines
- Collaborative research with [Moscow State University]

Path of a ray

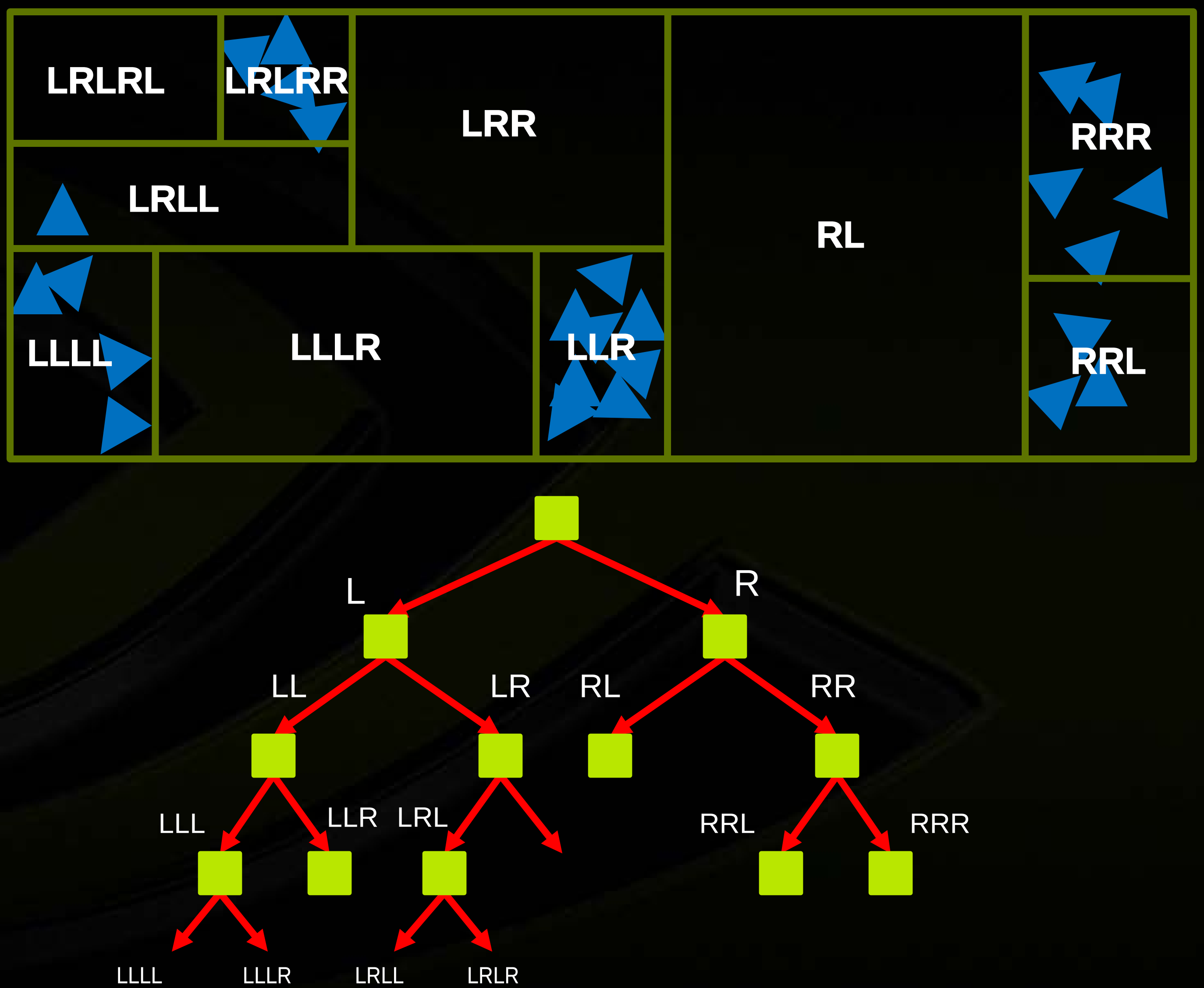
Tree traversal kernel
Primitive intersection kernel
Material & Light Kernel
Shading Kernel
Select K Leaves
Select Next Leaf
Intersection found: Primitive ID
Compute light equation
Ray
Tree traversal
Ray-triangle intersect
Generate Shadow Rays
Shading
Generate Secondary Rays
Select Next Primitive
Shaded cluster is sampled

Path of a ray

- Unknown number of rays
- Ray workload and memory access is highly irregular
- Register & Bandwidth pressure is high

Kd-tree

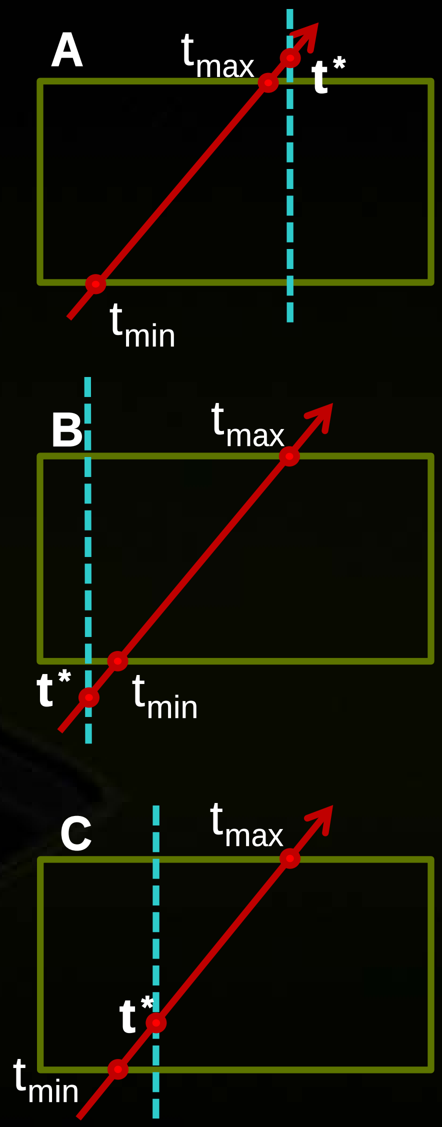
Kd-tree



LRLRL
LRLRR
LRR
RRR
LRLL
RL
LLLL
LLLR
LLR
RRL
L
R
LL
LR
RL
RR
LLL
LLR
LRL
RRL
RRR
LLLL
LLLR
LRLL
LRLR

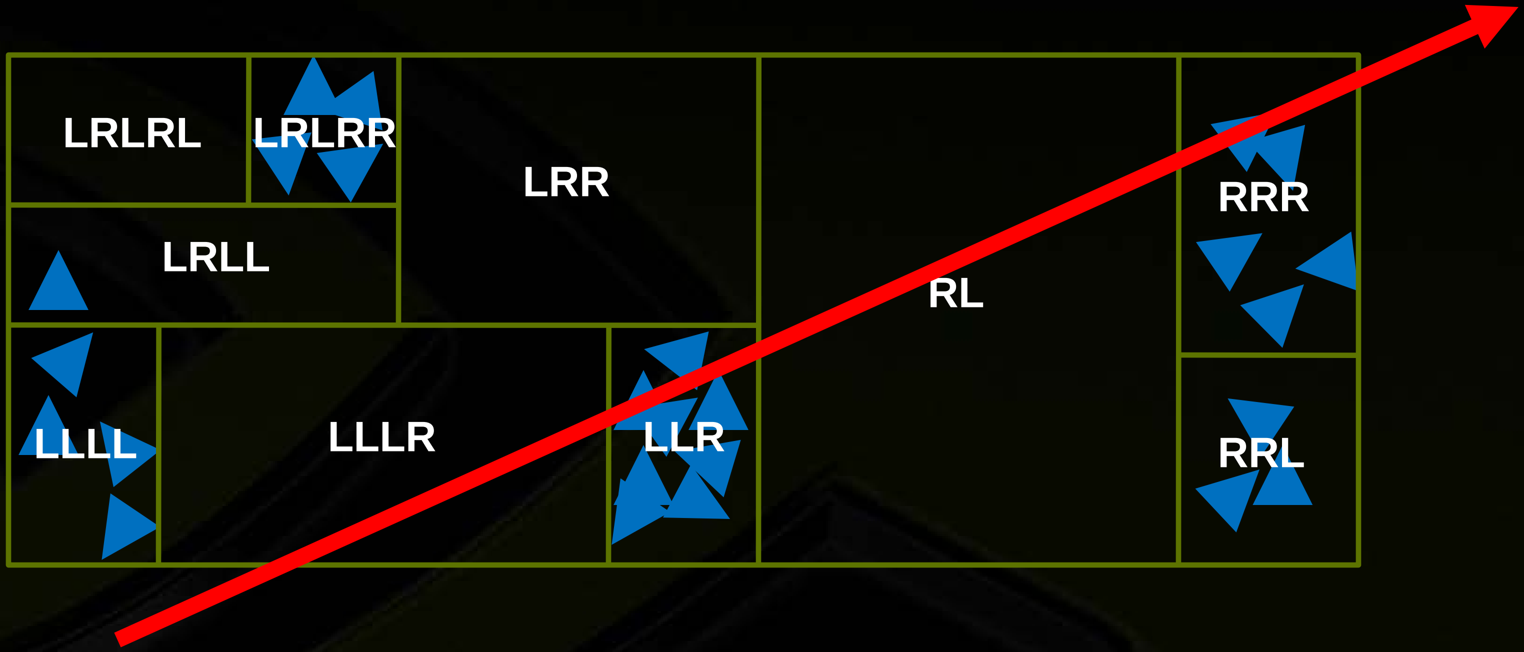
Kd-tree

- Registers – 13 min:
  - Ray – 6
  - t, t_{min}, t_{max} – 3
  - node – 2
  - tid, stack_top – 2
  - 19 registers – is a practical number
  - Stack in local memory



Kd-tree

- **Tree traversing**

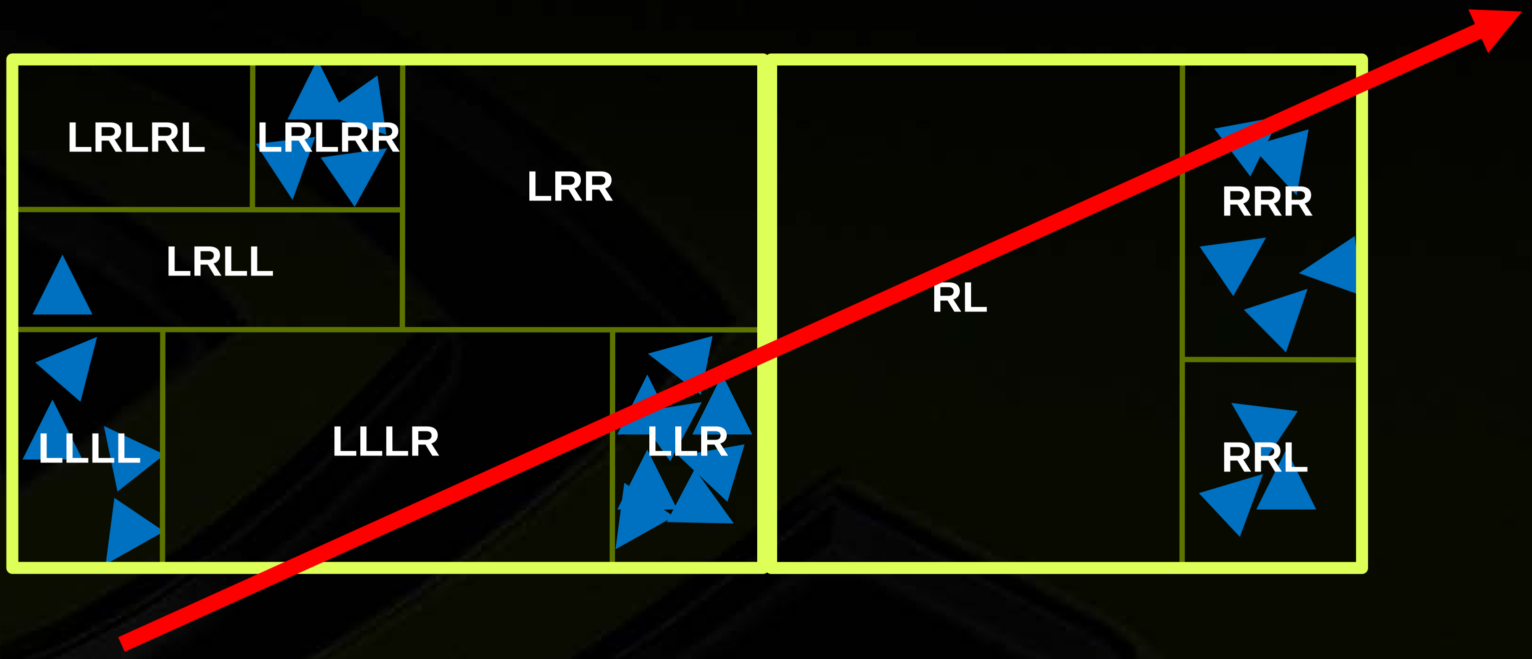


Stack:

Current Node:

Kd-tree

- Tree traversing

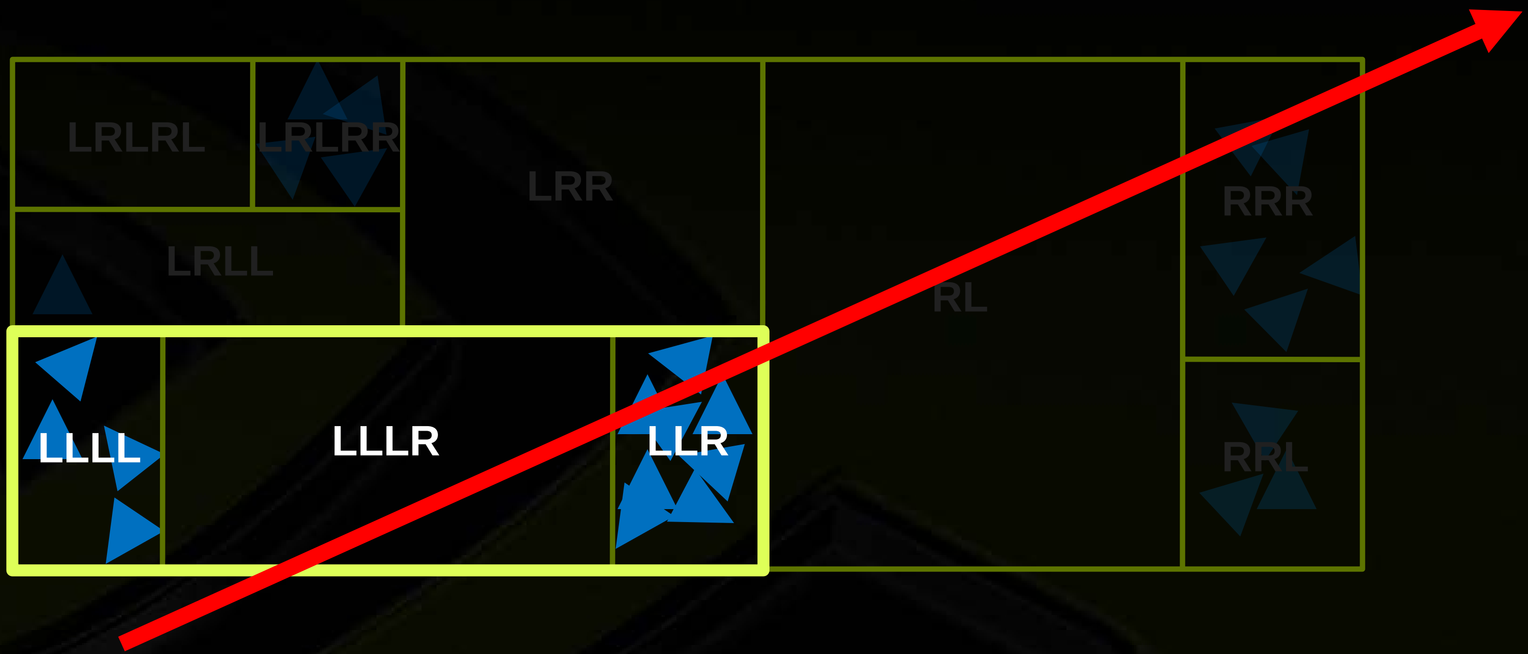


Stack: **R**

Current Node: **L**

Kd-tree

- **Tree traversing**

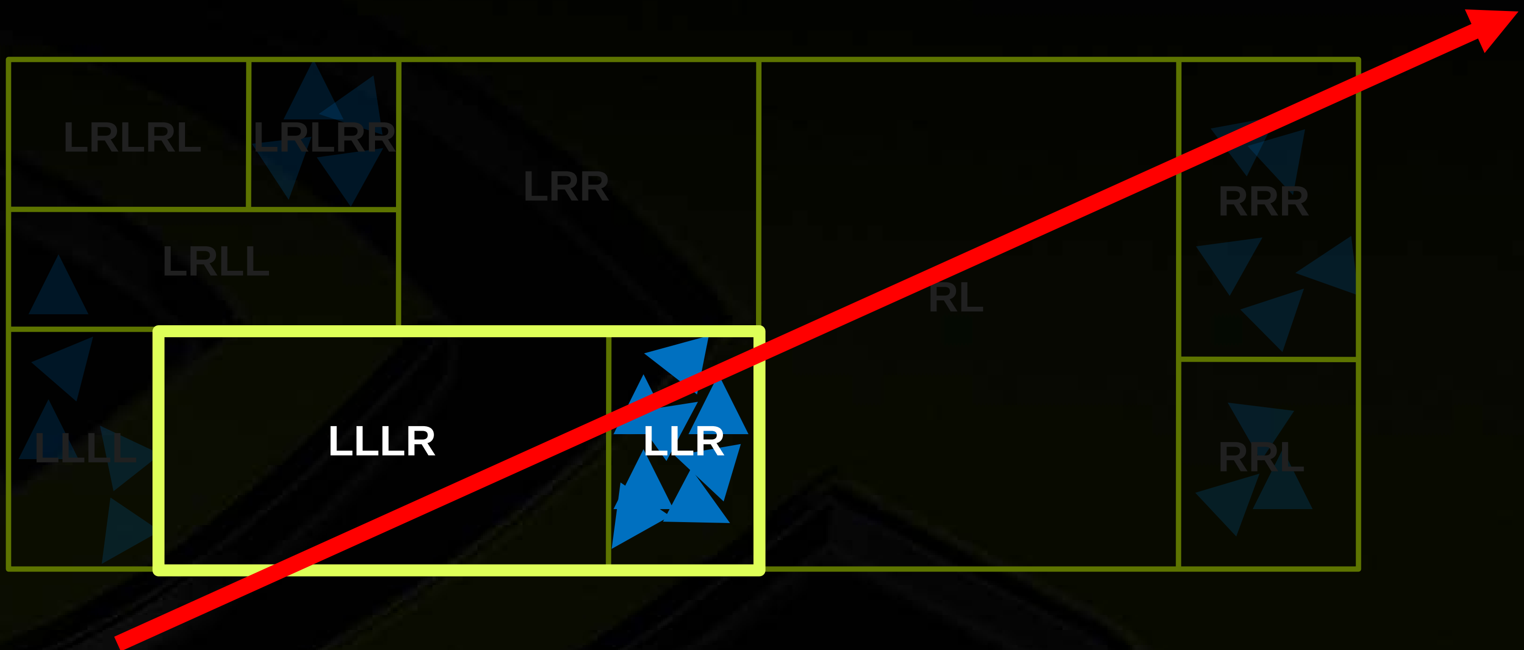


Stack: **R**

Current Node: **LL**

Kd-tree

- **Tree traversing**



Stack: **LLR, R**

Current Node: **LLL**

Kd-tree

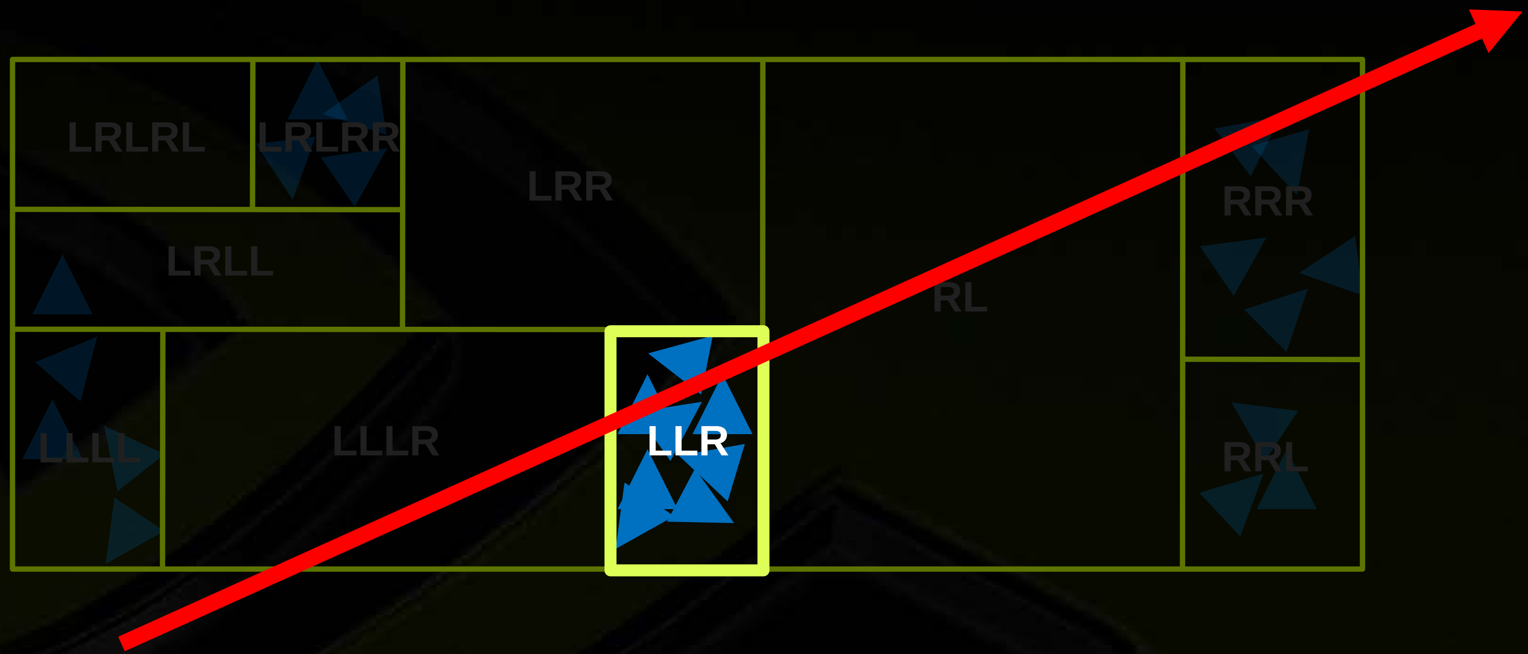
- **Tree traversing**

Stack: **LLR, R**

Current Node: **LLLR**

Kd-tree

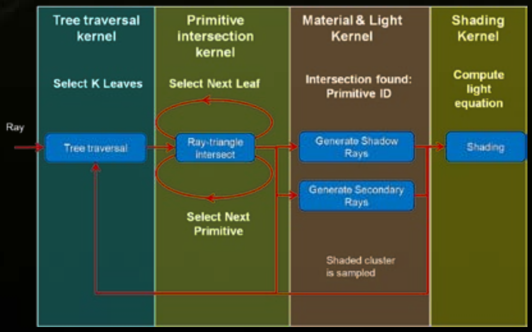
- **Tree traversing**



Stack: **R**

Current Node: **LLR**

We could stop here!



Kd-tree

- **Tree traversing**

Stack:

Current Node: **R**

Kd-tree

- **Tree traversing**

Stack: **RR**

Current Node: **RL**

Kd-tree

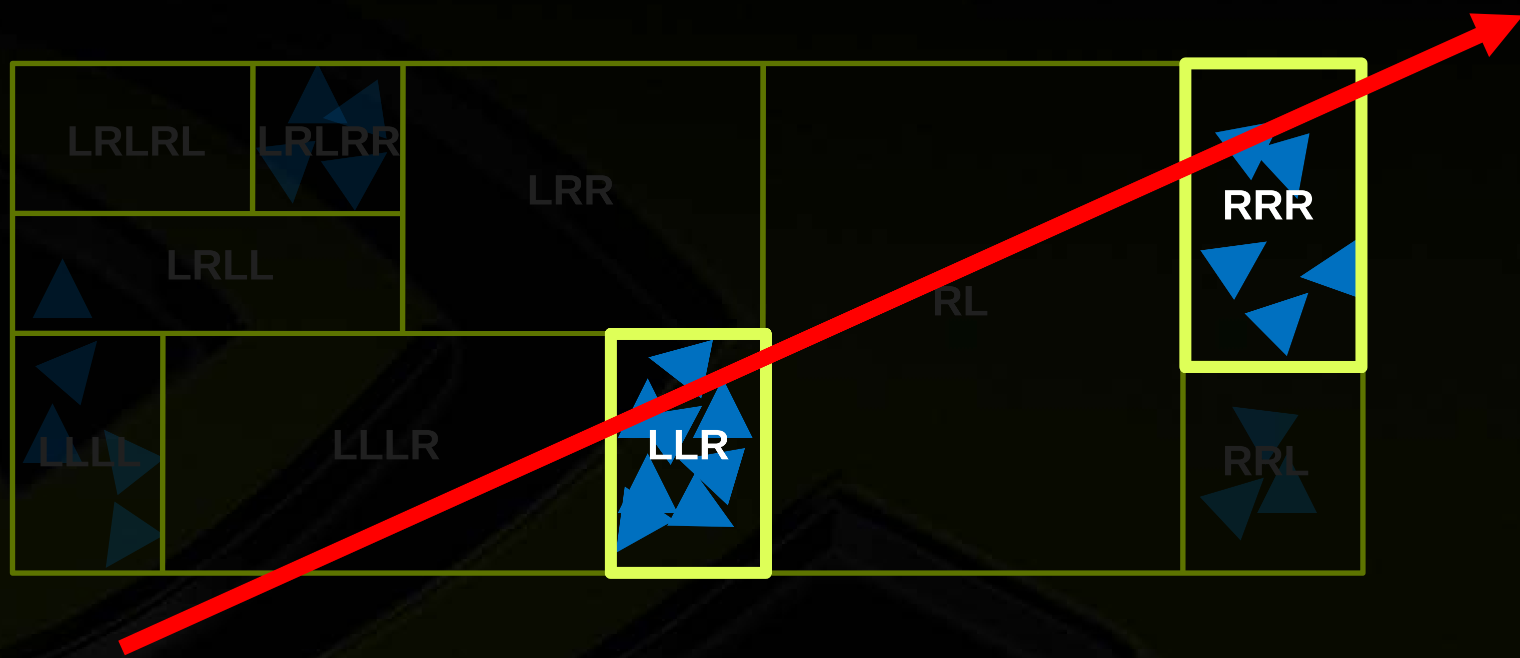
- **Tree traversing**

Stack:

Current Node: **RR**

Kd-tree

- **Tree traversing**

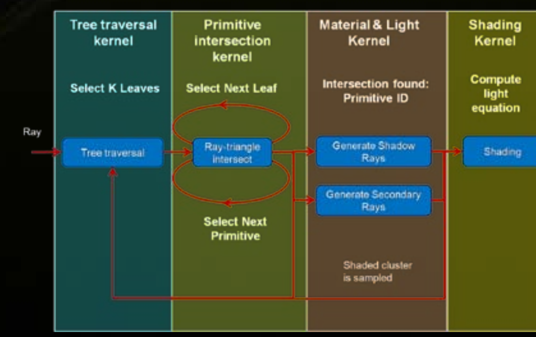


Stack:

Current Node: **RRR**

Result:

LLR, RRR



Tree traversal

- Different rays may run for different time
  - One thread can stall a whole block
- Each thread needs a buffer to store all possible leafs
  - Worst case: a ray intersects all possible leafs of a tree

Tree traversal

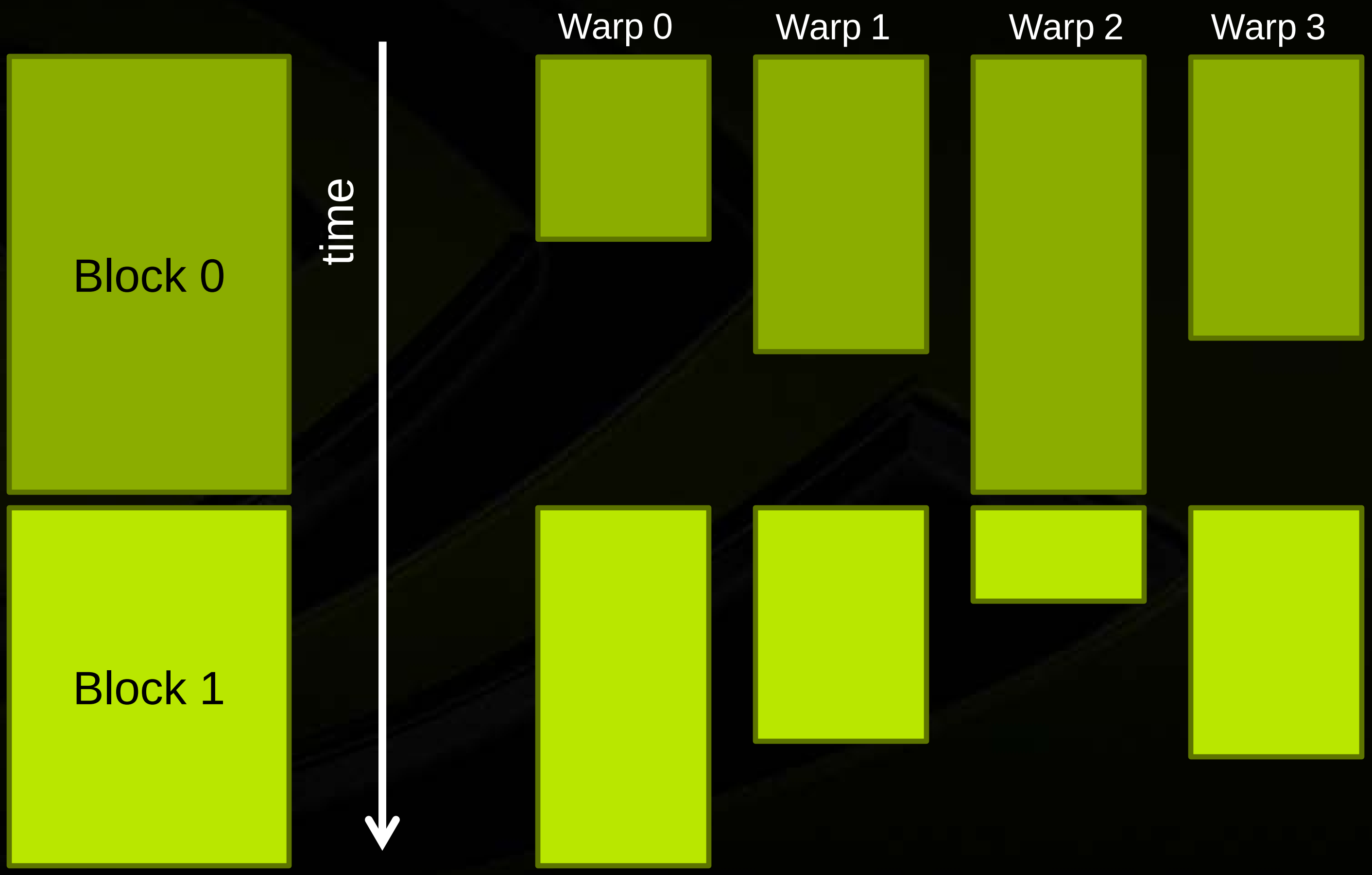
- Different rays may run for different time
  - Solution: Persistent threads
- Each thread needs a buffer to store all possible leafs
  - Solution: Screen tiling

Persistent threads

- Launch as many threads as possible
  - Depends on HW architecture and kernel requisites
- Keep all threads busy
  - Create a pool of rays to traverse a tree

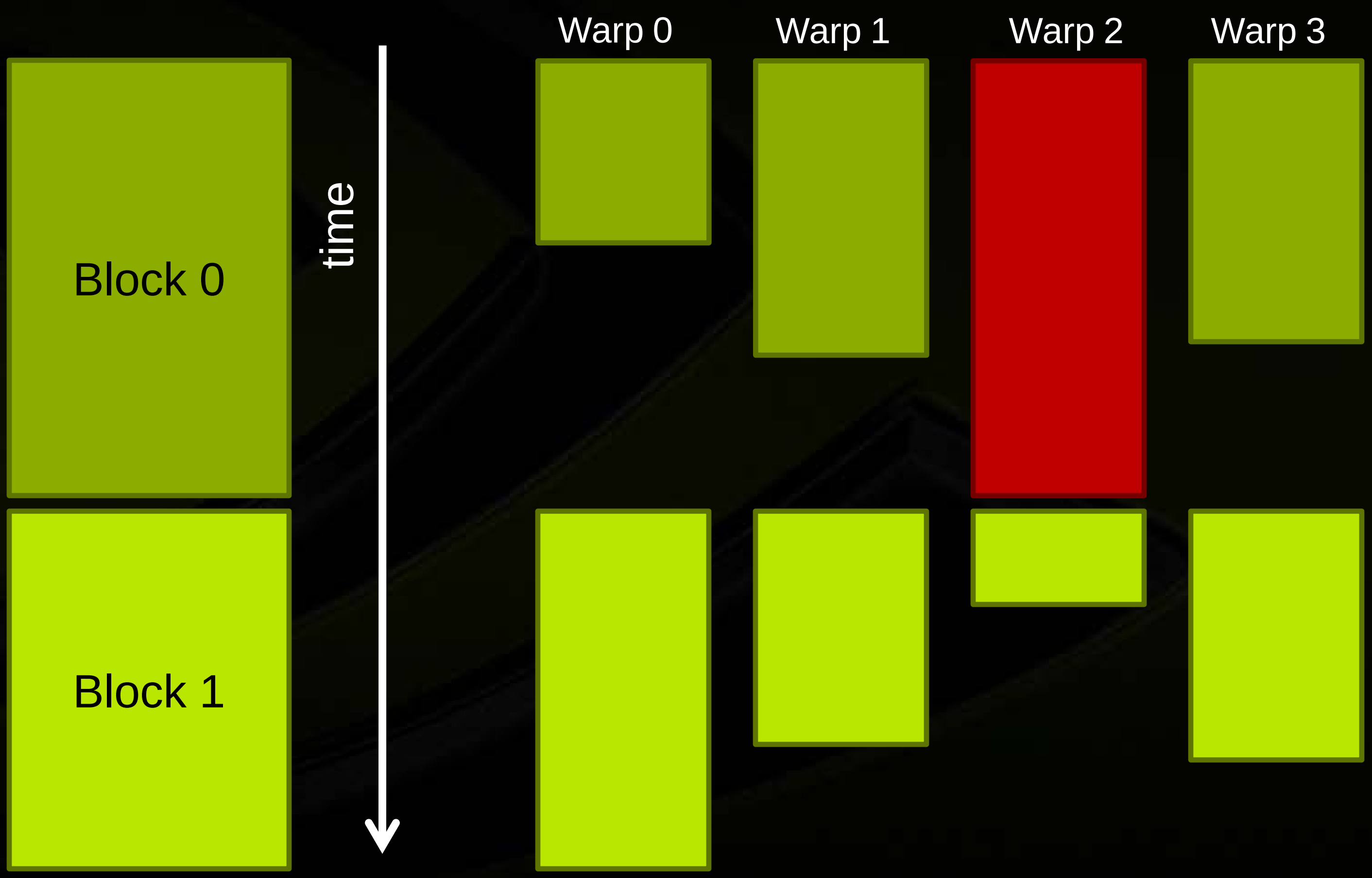
Regular execution

- **Disadvantages**
  - **Waiting until all threads finish execution to launch new block**



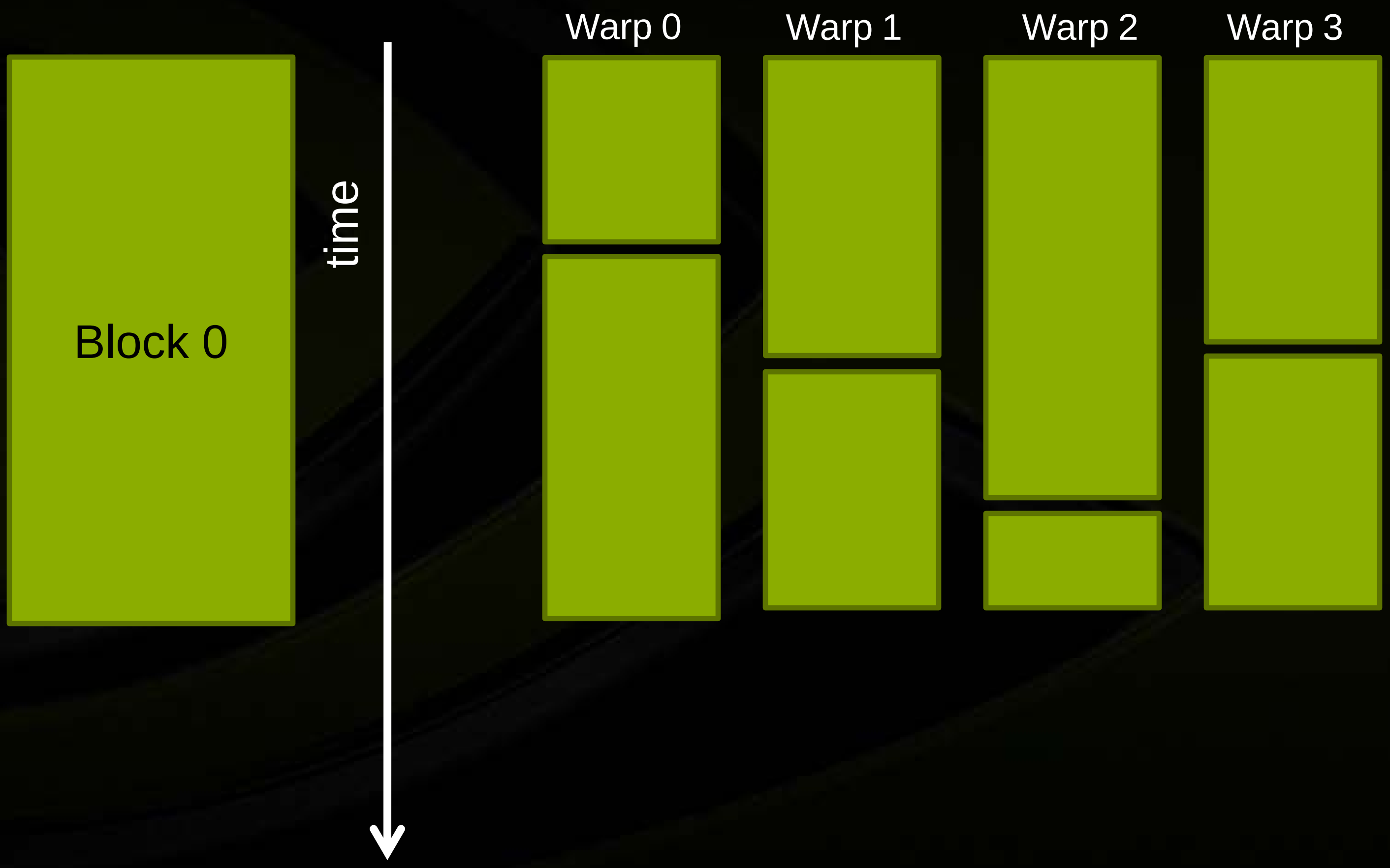
Regular execution

- **Disadvantages**
  - **Waiting until all threads finish execution to launch new block**



Persistent threads execution

- **Advantages**
  - **Workload is balanced between warps**



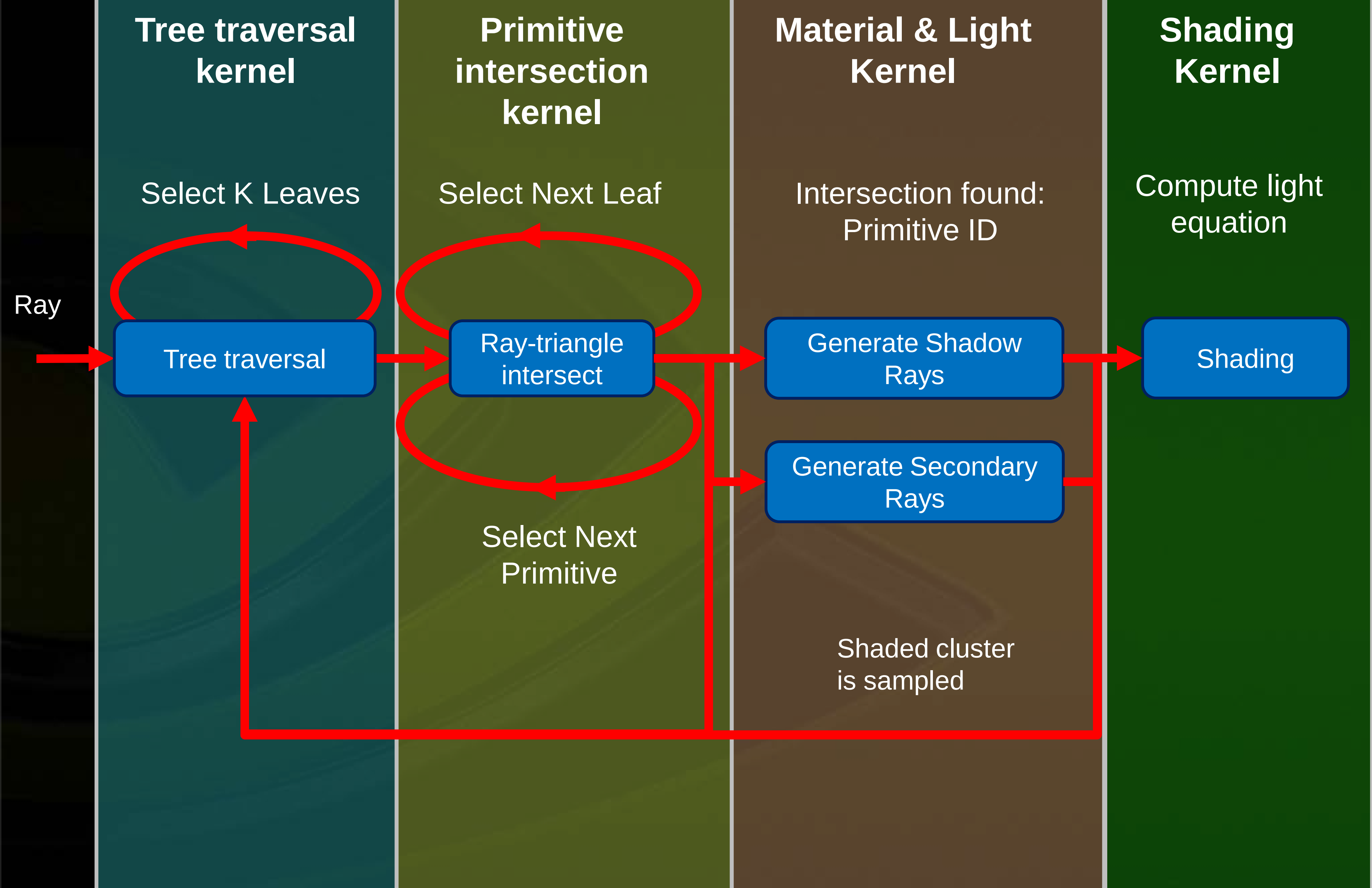
Screen Tiling

- Split the screen into multiple tiles
- Render tiles separately
- Tiles of 128x128 / 256x256 work well
  - 128x128 is still 16K of threads!
- Allows easy multi-GPU performance scaling
- Control over memory

Tree traversal

- Screen is split into tiles (256x256)
  - Reserve place for a number of non-empty leafs
- Launch fixed number of threads

Path of a ray



Tree traversal kernel
Primitive intersection kernel
Material & Light Kernel
Shading Kernel
Select K Leaves
Select Next Leaf
Intersection found: Primitive ID
Compute light equation
Ray
Tree traversal
Ray-triangle intersect
Generate Shadow Rays
Shading
Generate Secondary Rays
Select Next Primitive
Shaded cluster is sampled

Ray-triangle intersection

- **Minimum storage ray-triangle intersection**

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{dot(P, E_1)} \begin{bmatrix} dot(Q, E_2) \\ dot(P, T) \\ dot(Q, D) \end{bmatrix}$$

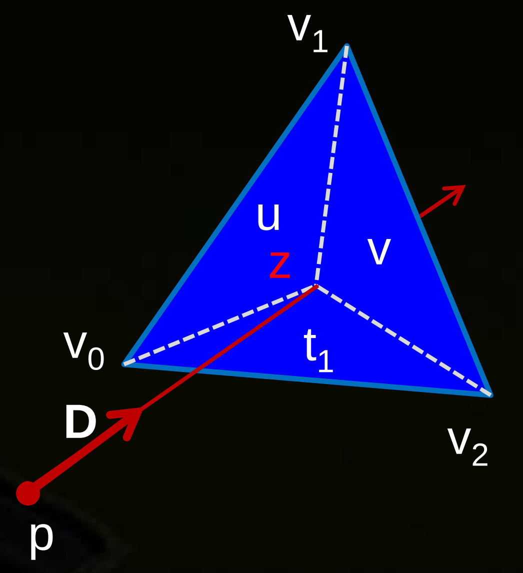
$$E_1 = v_1 - v_0$$

$$E_2 = v_2 - v_0$$

$$T = p - v_0$$

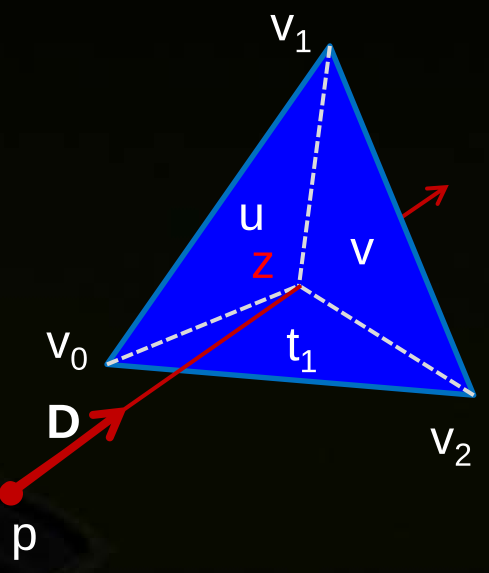
$$P = cross(D, E_2)$$

$$Q = cross(T, E_1)$$



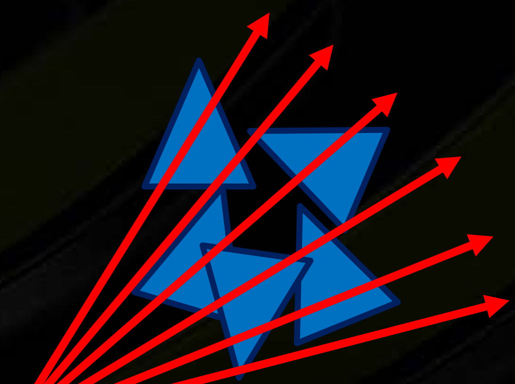
Ray-triangle intersection

- Computational complexity (>30 MADs)
- Register Pressure (>23)
  - 6 r. per ray
  - 9 r. per triangle
  - 3 r. for intersection result (t, u, v)
  - 1 r. for Triangle Count
  - 1 r. for loop index
  - 1 r. for thread ID (tid)
  - 2 r. min_t и min_id

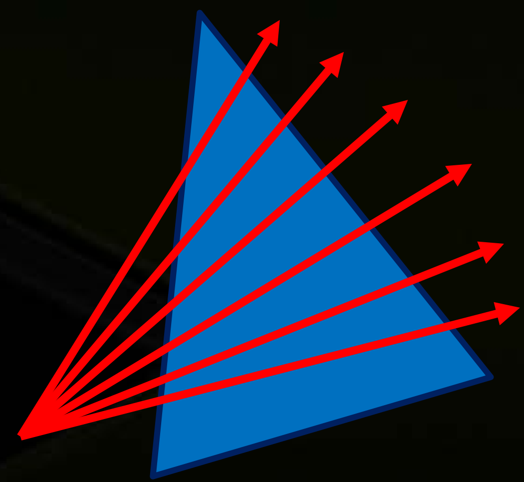


Ray-triangle kernel

- Each thread is mapped to a ray
  - Each ray operates on its triangle

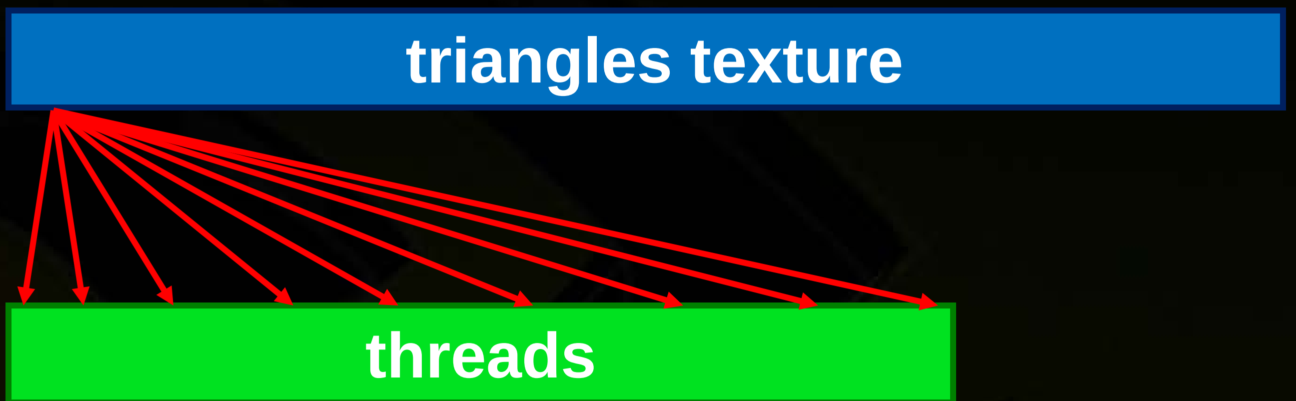


  - Block of threads shares triangles (packet)



Ray-triangle intersection

- **Each thread is mapped to a ray**




```
for (int i=0;i<triNum;i++)
{
    (A,B,C) = tex1Dfetch(tex,i);
    // intersection code
}
```

Kernel takes **32** registers

Ray-triangle intersection

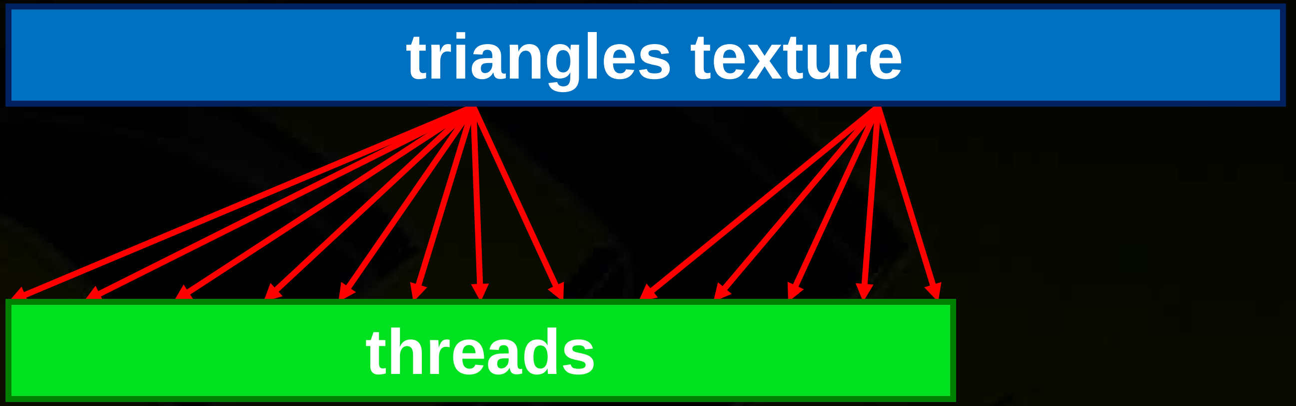
- **Each thread is mapped to a ray**


```
for (int i=0;i<triNum;i++)
{
    (A,B,C) = tex1Dfetch(tex,i);
    // intersection code
}
```

Kernel takes **32** registers

Ray-triangle intersection

- Each thread is mapped to a ray

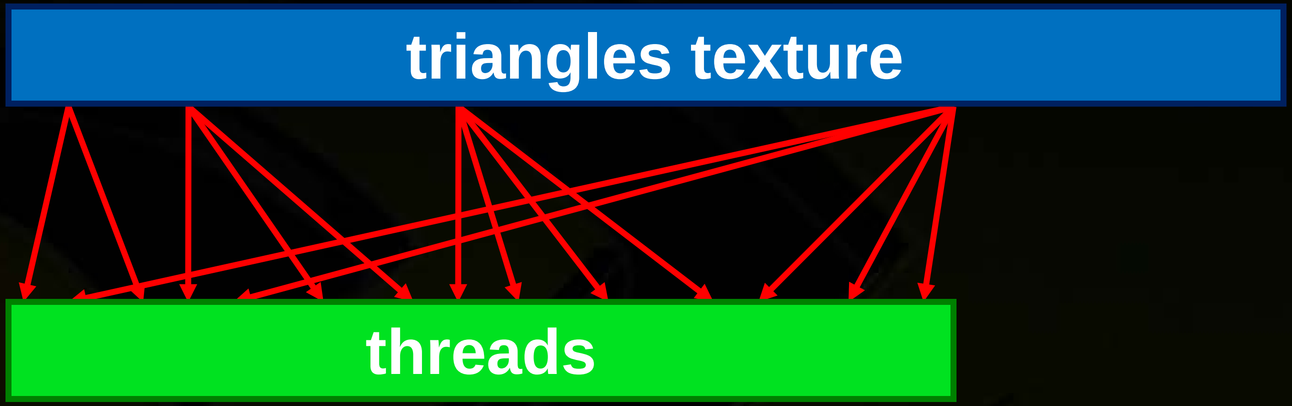



```
for (int i=0;i<triNum;i++)
{
    (A,B,C) = tex1Dfetch(tex,i);
    // intersection code
}
```

Kernel takes **32** registers

Ray-triangle intersection

- **Each thread is mapped to a ray**

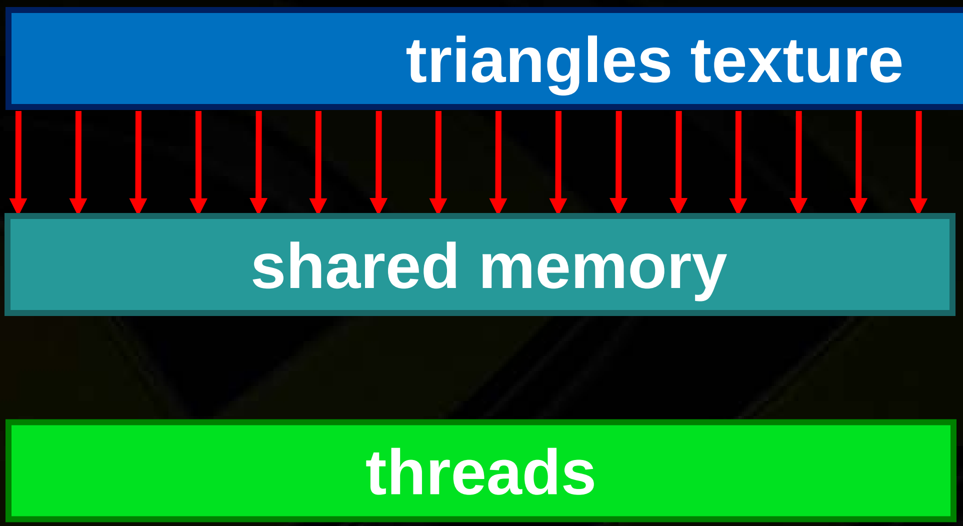


Kernel takes **32** registers

```
for (int i=0;i<triNum;i++)
{
    (A,B,C) = tex1Dfetch(tex,i);
    // intersection code
}
```

Ray-triangle intersection

- **Packet tracing**

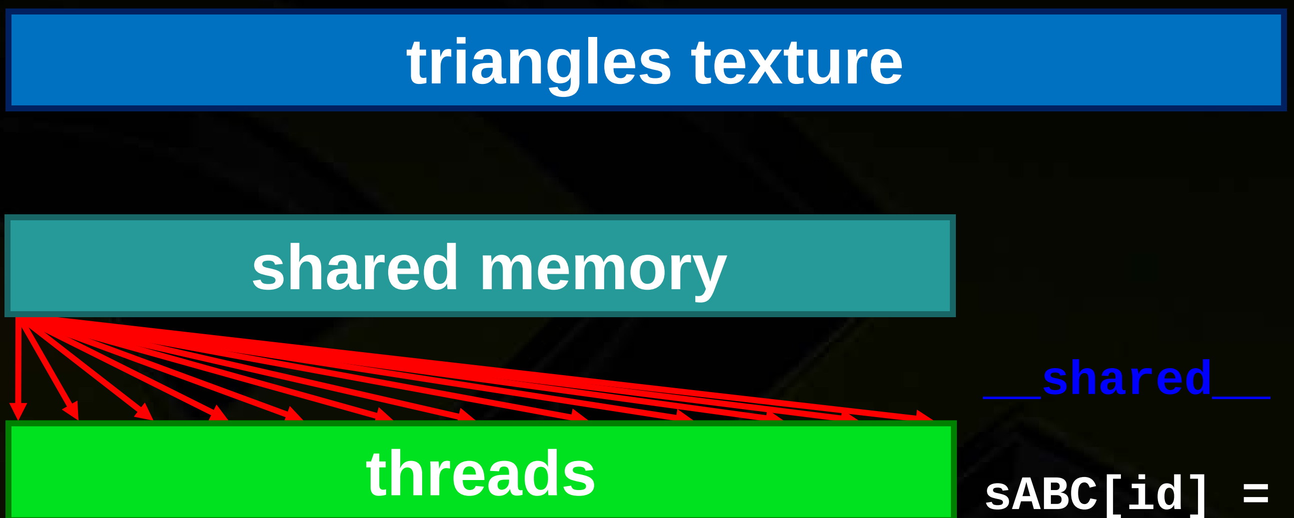



```
__shared__  sABC[N];

sABC[id] = tex1Dfetch(tex,i);
__syncthreads();

for (int i=0;i<N;i++)
{
    (A,B,C) = sABC[i];
    // intersection code
}
```

Kernel takes **20** registers

Ray-triangle intersection

- **Packet tracing**

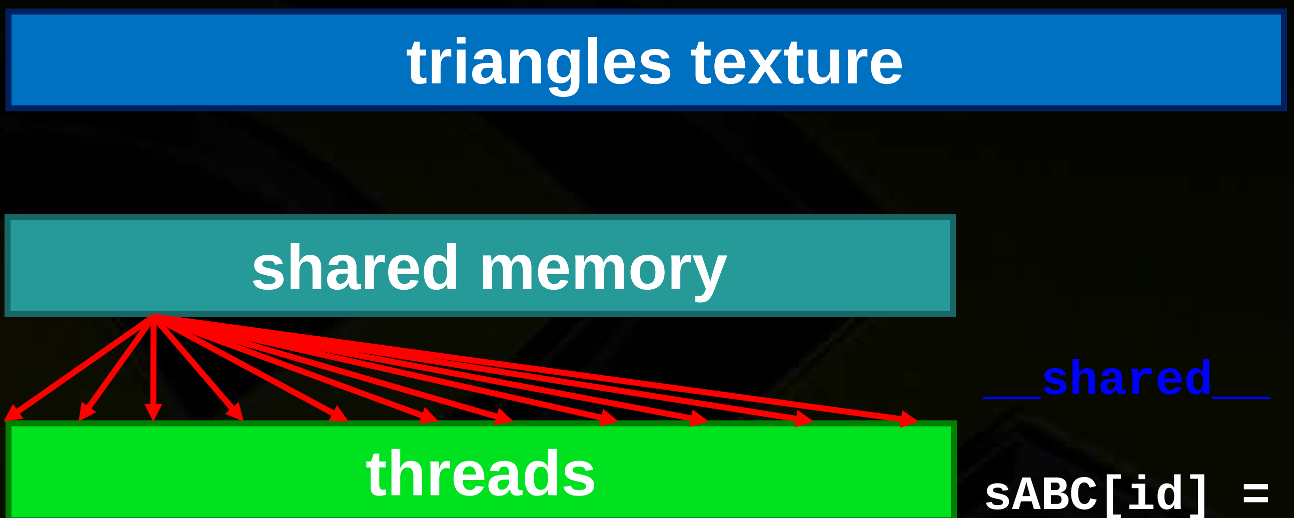



```
__shared__  sABC[N];

sABC[id] = tex1Dfetch(tex,i);
__syncthreads();

for (int i=0;i<N;i++)
{
   (A,B,C) = sABC[i];
   // intersection code
}
```

Kernel takes **20** registers

Ray-triangle intersection

- **Packet tracing**




```
__shared__  sABC[N];

sABC[id] = tex1Dfetch(tex,i);
__syncthreads();

for (int i=0;i<N;i++)
{
   (A,B,C) = sABC[i];
   // intersection code
}
```

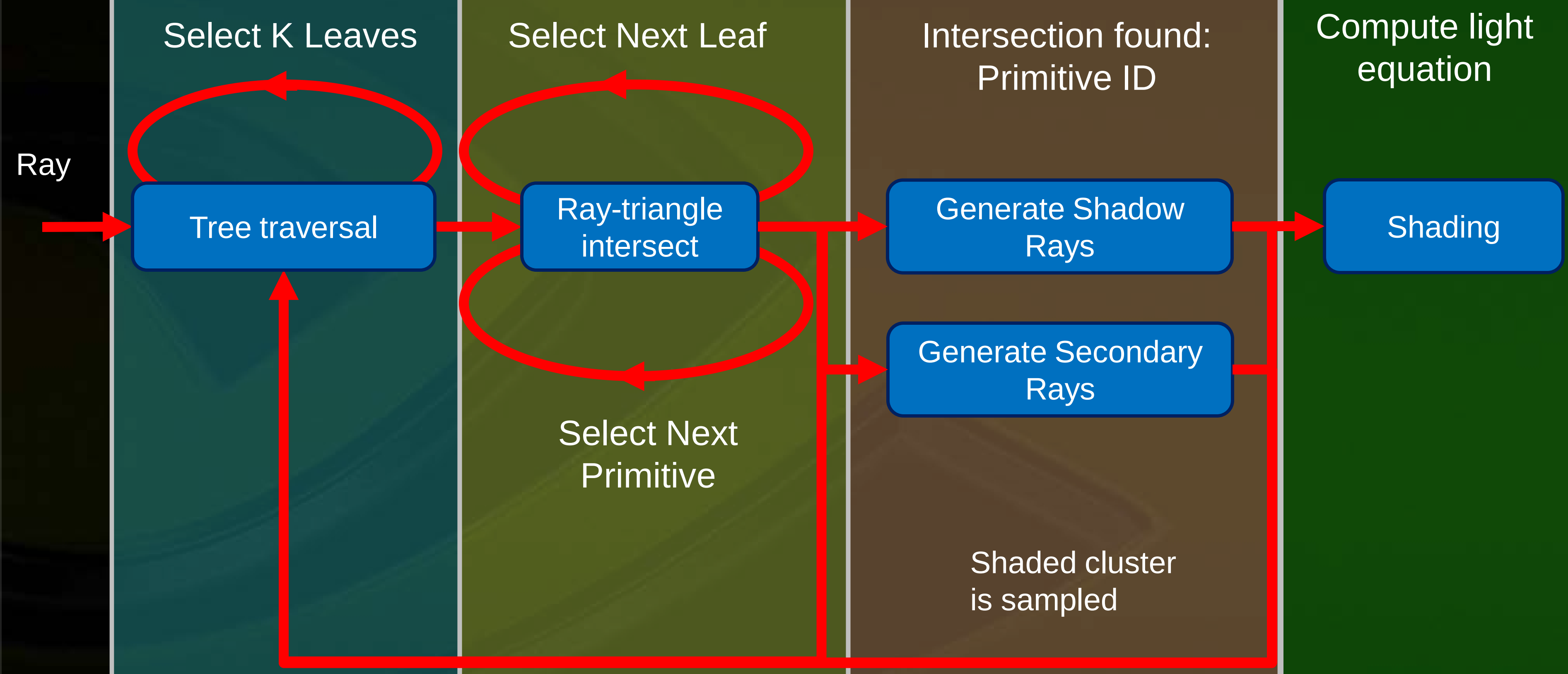
Kernel takes **20** registers

Ray-triangle intersection

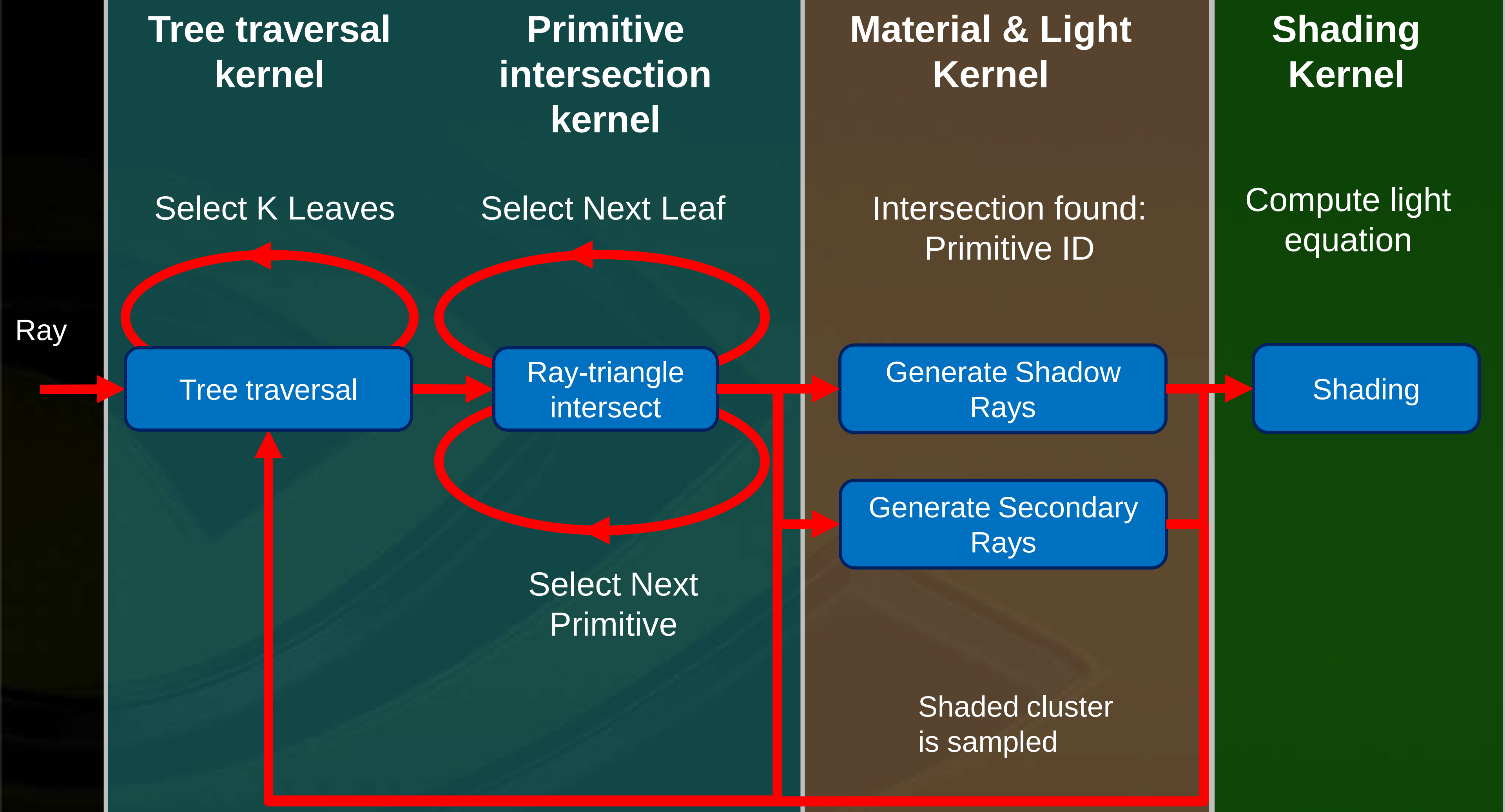
- Performance comparison
  - One thread per ray: 1x
  - Packet tracing: 1.3x

Path of a ray

Tree traversal kernel	Primitive intersection kernel	Material & Light Kernel	Shading Kernel



Path of a ray


Tree traversal kernel
Primitive intersection kernel
Material & Light Kernel
Shading Kernel
Select K Leaves
Select Next Leaf
Intersection found: Primitive ID
Compute light equation
Ray
Tree traversal
Ray-triangle intersect
Generate Shadow Rays
Generate Secondary Rays
Shading
Select Next Primitive
Shaded cluster is sampled

Uber-kernel

- **Uber kernel – a kernel of the following structure:**

```
if ( condition_A )
{
    Do_Work_A();
}
else if ( condition_B )
{
    Do_Work_B();
}
…
```

- **Ideally *condition_A* / *condition_B* … are constant per block**

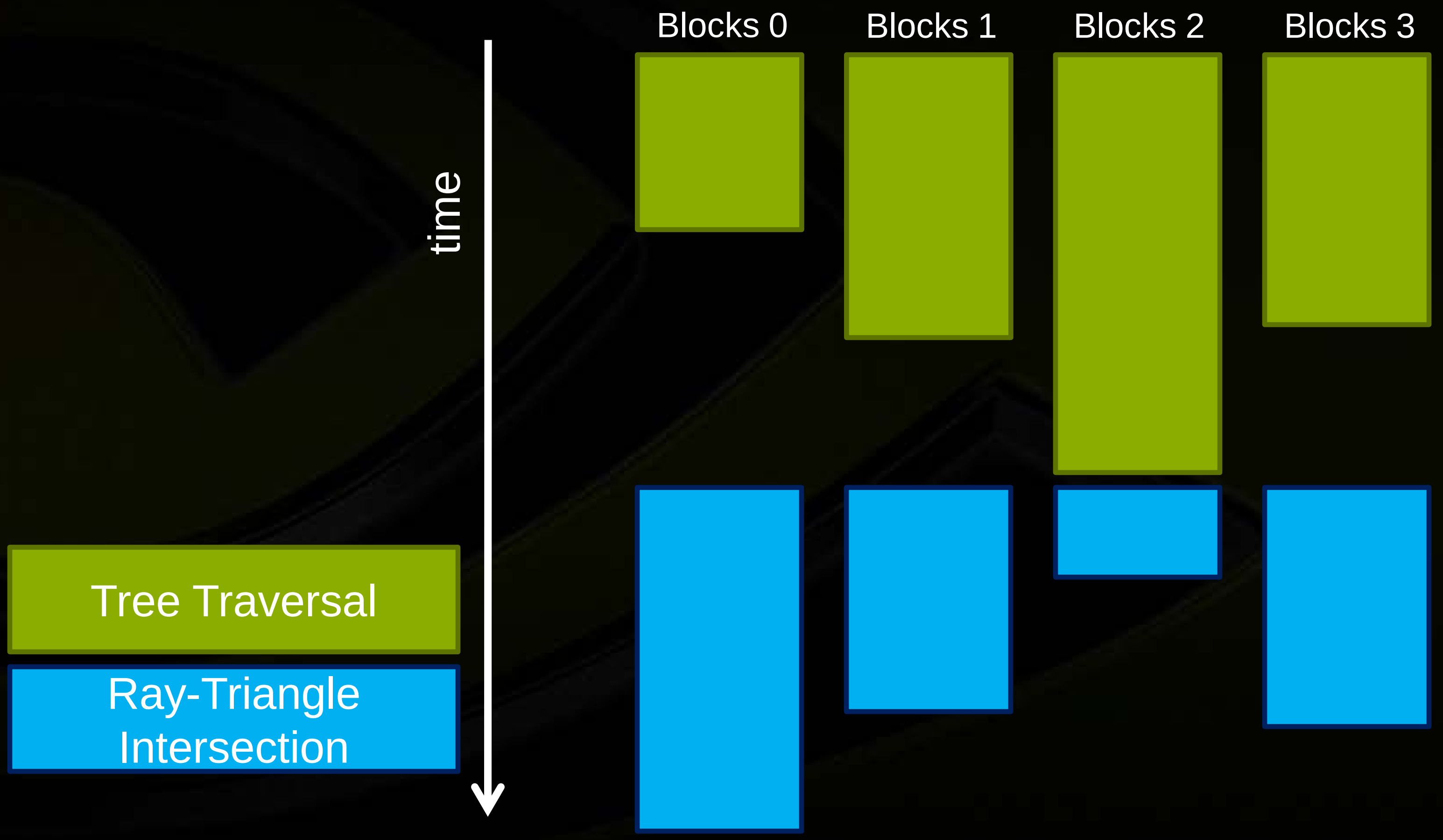
Ray tracing Uber-kernel

- **Uber kernel – a kernel of the following structure:**

```
if ( Leaf_List_is_Empty )
{
    Select_K_Leafs();
}
else
{
    Intersect_Trianles();
}
…
```

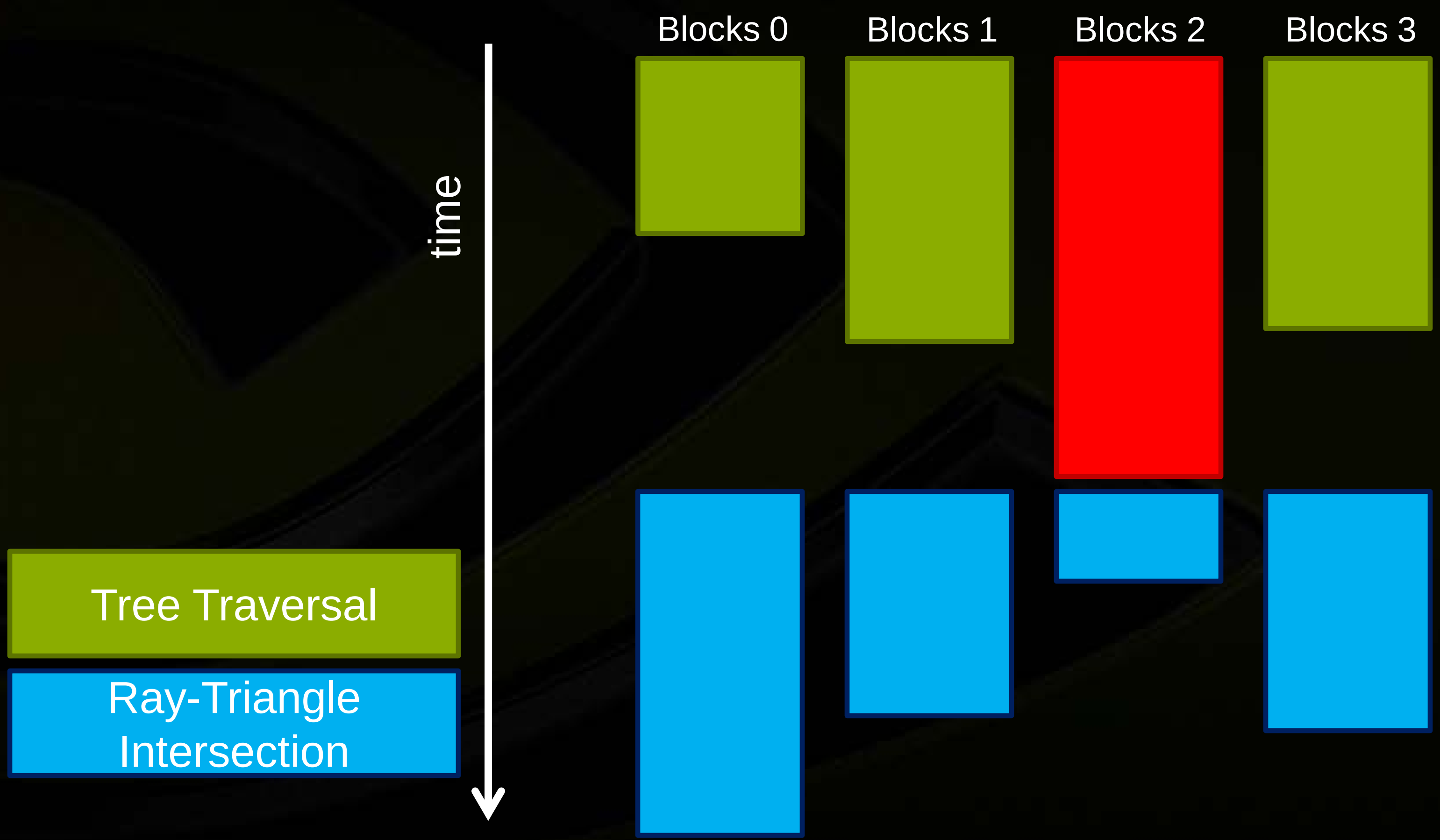
Ray-tracing: separate kernels

- **Disadvantages**
  - **Waiting until traversal kernel finishes execution**



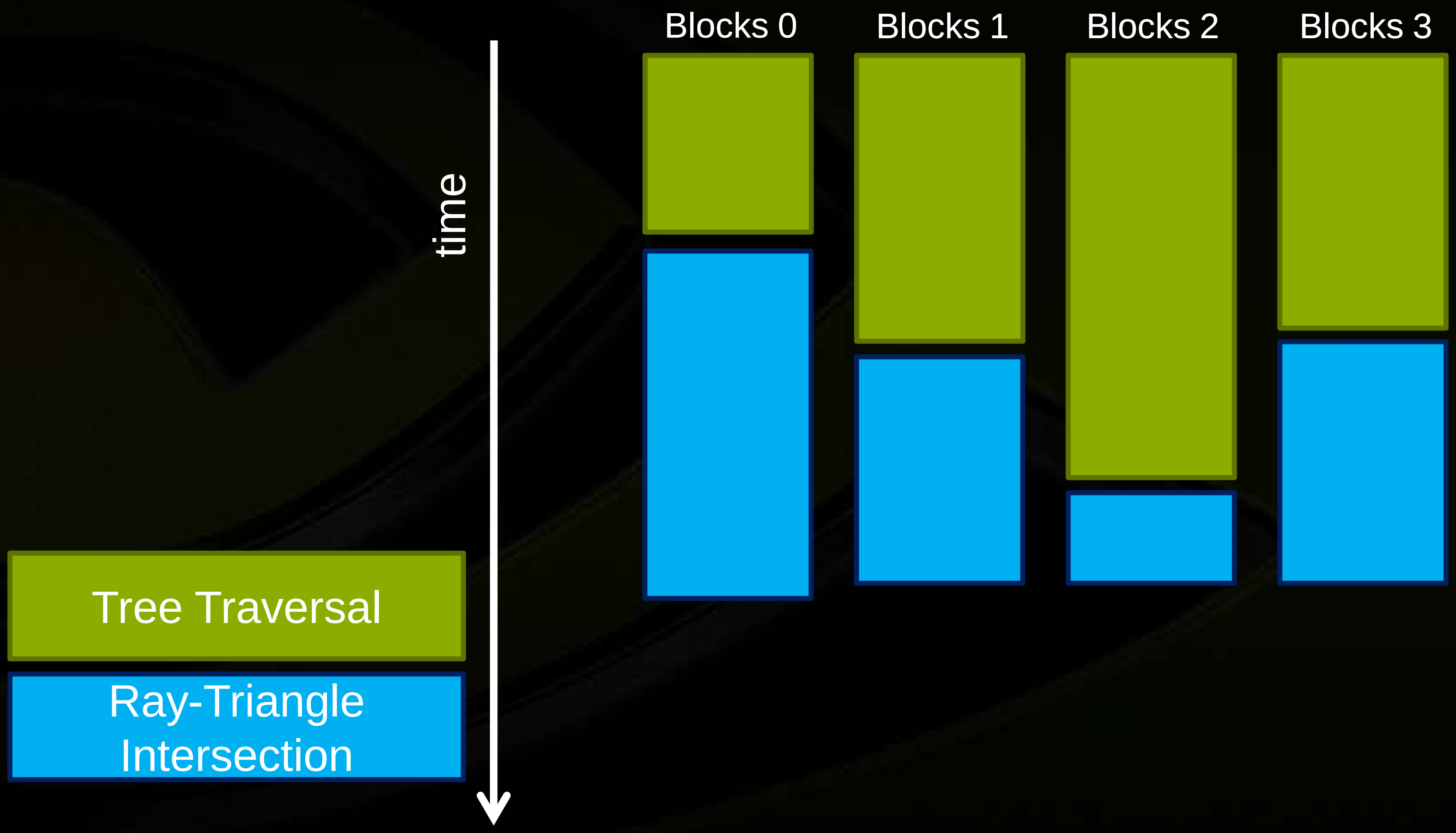
Ray-tracing: separate kernels

- **Disadvantages**
  - **Waiting until traversal kernel finishes execution**



Ray-tracing uber-kernel

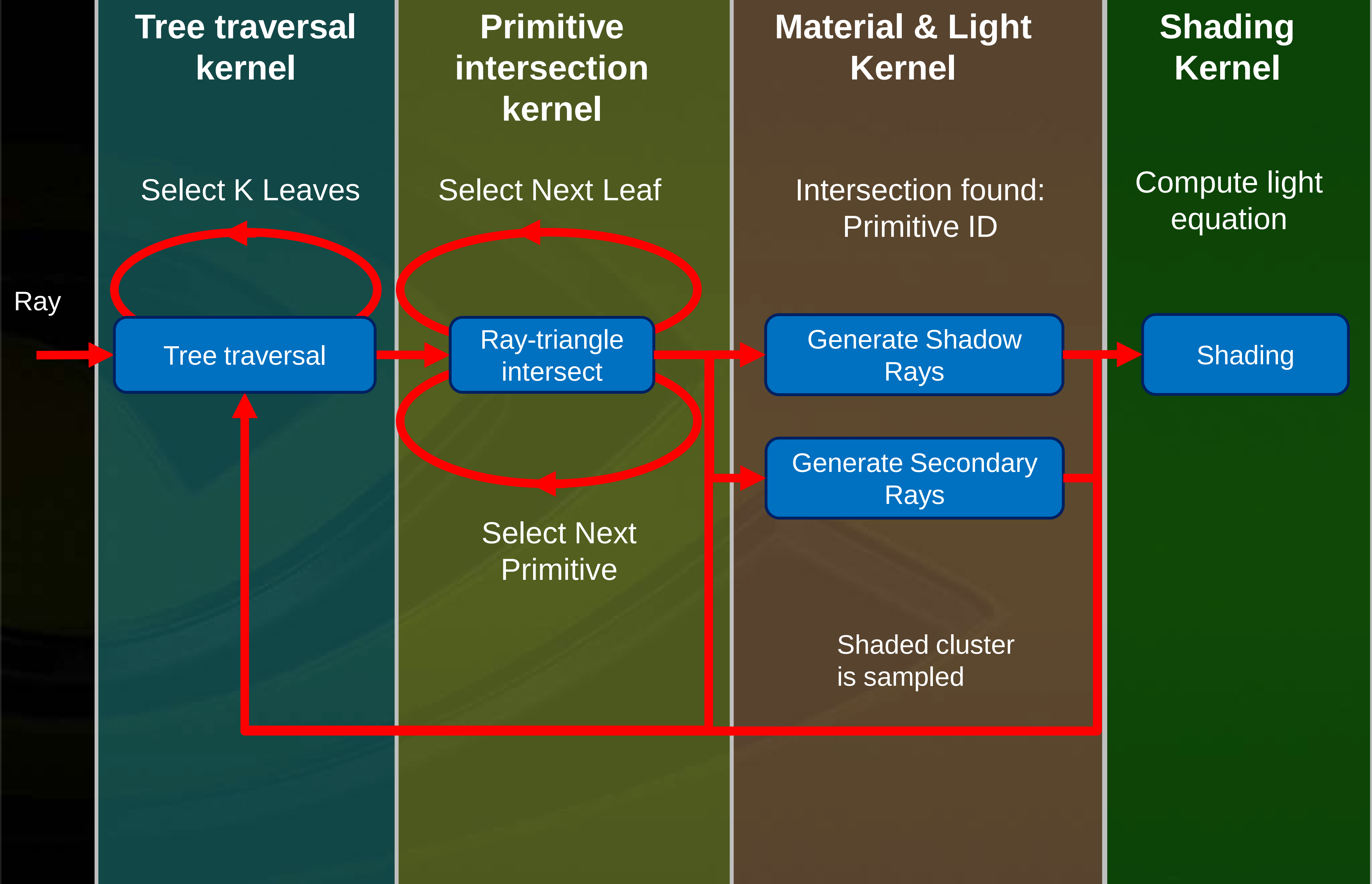
- **Advantages**
  - **Waiting until a block reaches the barrier**



Uber-kernel vs. Separate kernels

- Pros
  - Work switching
  - Memory savings
- Caution!
  - Hard to profile
  - Poor resources utilization

Path of a ray

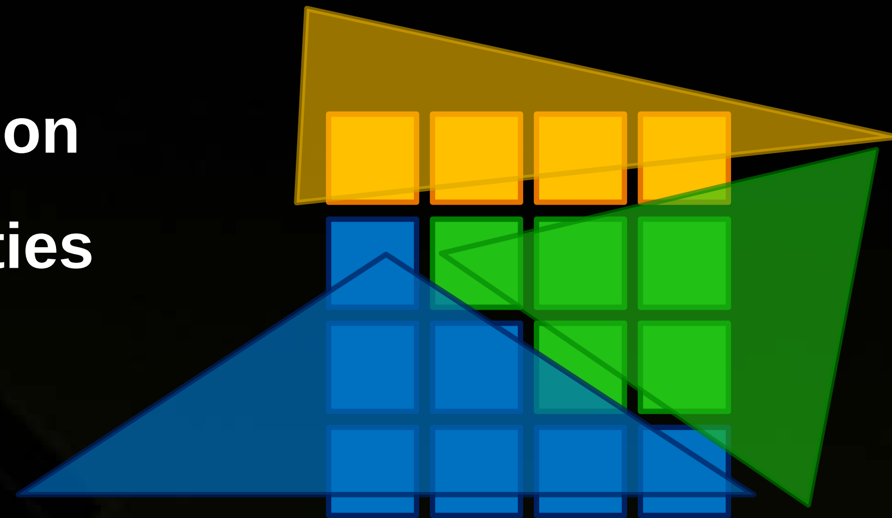

Tree traversal kernel
Primitive intersection kernel
Material & Light Kernel
Shading Kernel
Select K Leaves
Select Next Leaf
Intersection found: Primitive ID
Compute light equation
Ray
Tree traversal
Ray-triangle intersect
Generate Shadow Rays
Generate Secondary Rays
Shading
Select Next Primitive
Shaded cluster is sampled

Material & Light Kernel

- Ray-tracing result: primitive ID and depth buffer
  - Points to primitive matrix, material, etc
- For each pixel inside the tile evaluate:
  - Number of secondary rays
  - Number of shadow rays
- Get total number of rays

Material & Light Kernel

- **Number of rays depends on material and light properties**



Primitive ID

0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	1

Refraction ray count

5	5	5	5
0	1	1	1
0	0	1	1
0	0	0	0

Reflection ray count

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

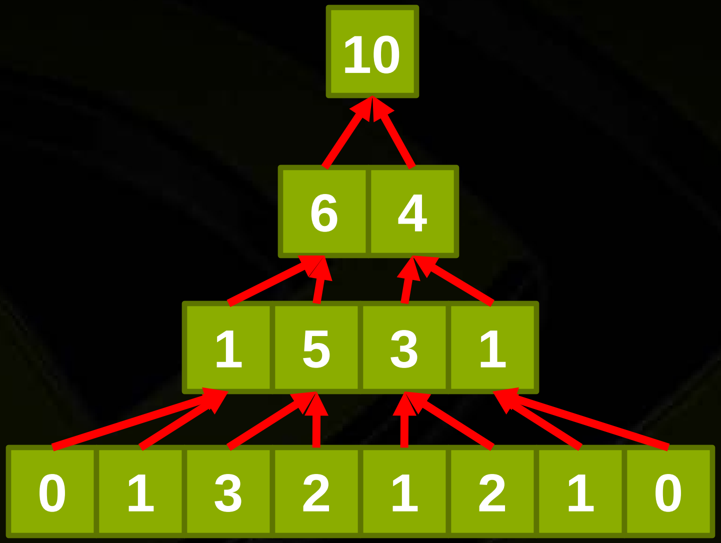
Shadow ray count

0	0	0	0
0	9	9	9
0	0	9	9
0	0	0	0

Diffuse ray count

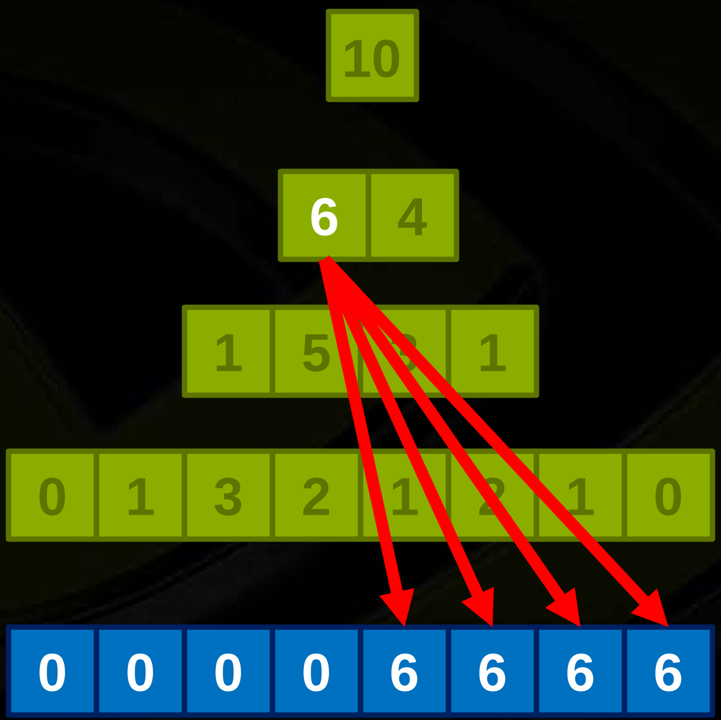
Histogram Pyramid

- Use histogram pyramid for stream compaction



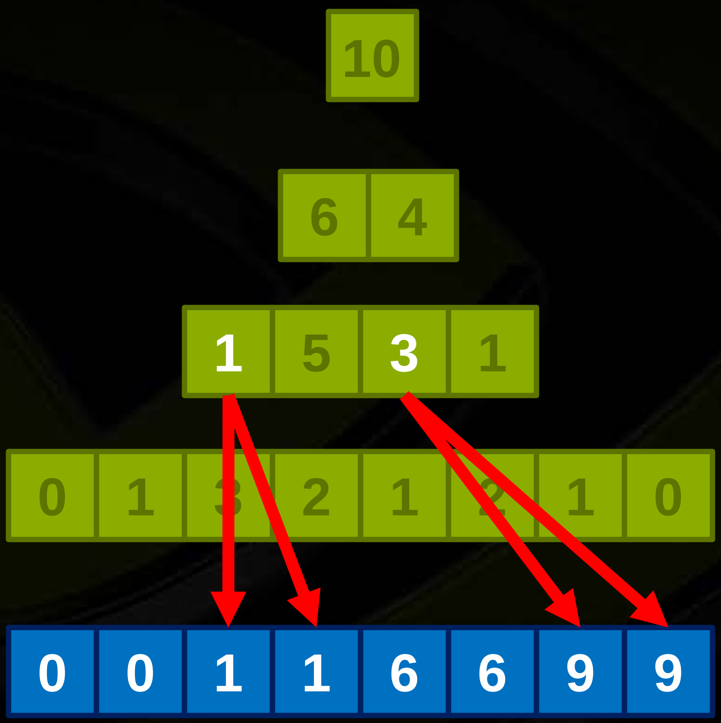
Histogram Pyramid

- Use histogram pyramid for stream compaction



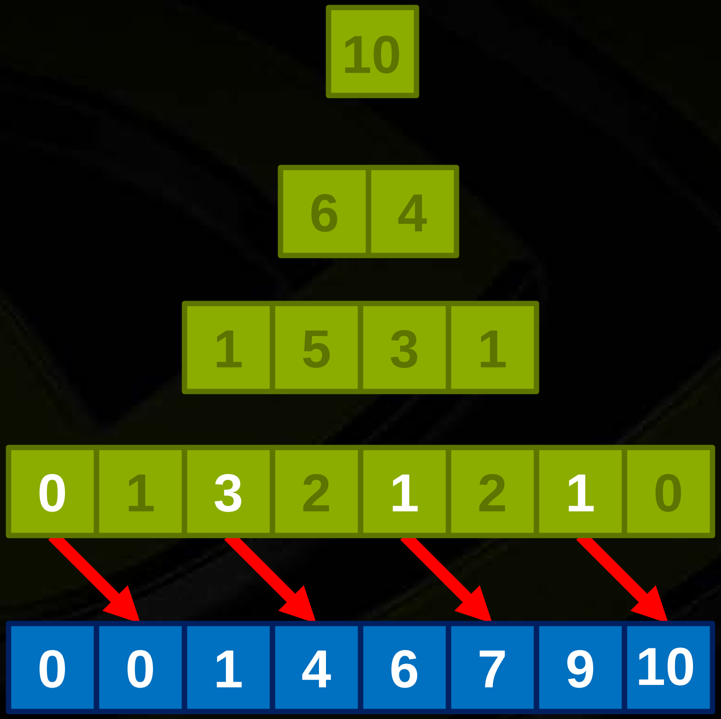
Histogram Pyramid

- Use histogram pyramid for stream compaction



Histogram Pyramid

- **Use histogram pyramid for stream compaction**



Material & Light Kernel

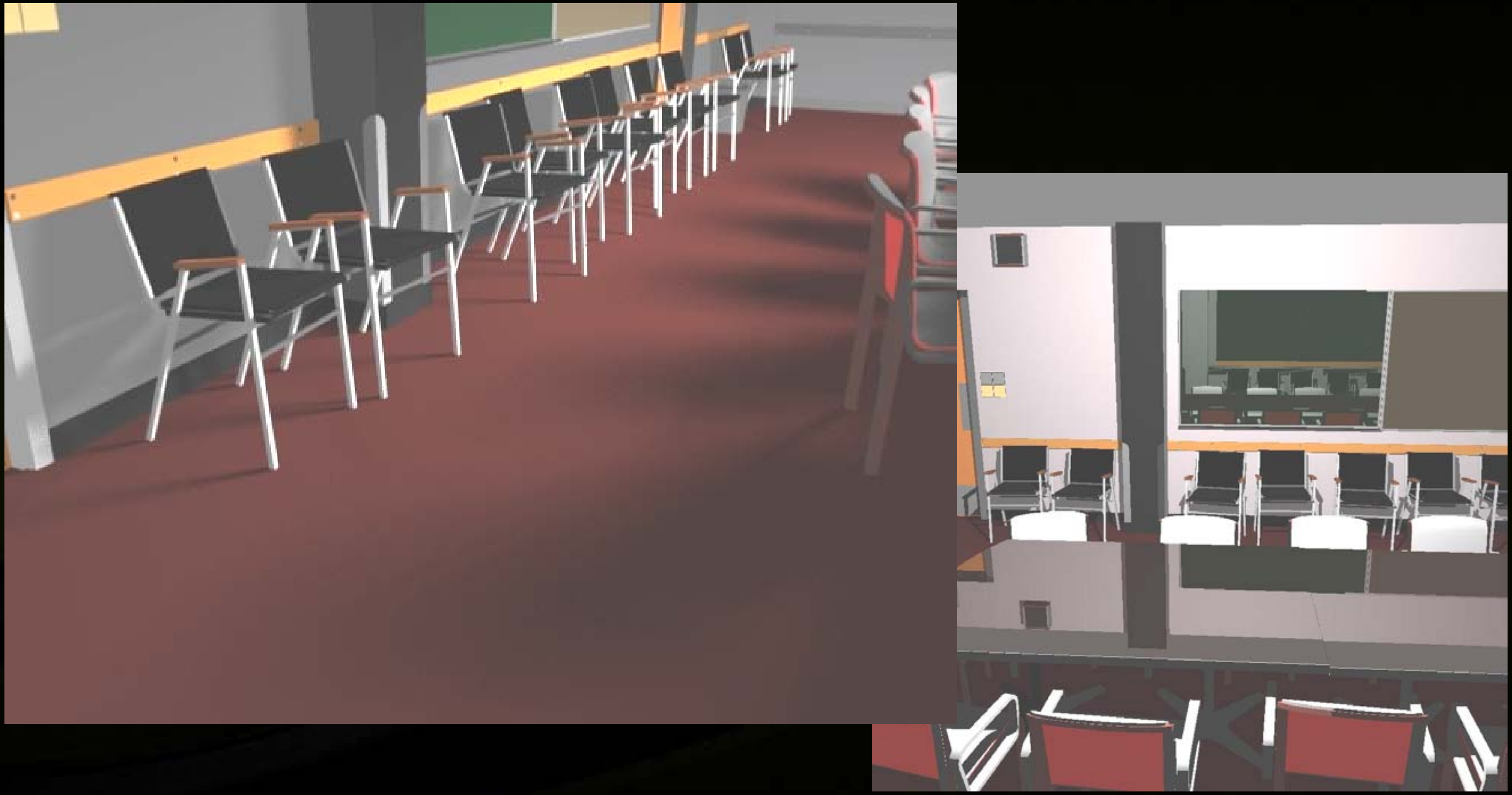
- Read back the total number of rays required
- Check if resources are available
- Send generated rays to tree traversal & primitive intersection stage

Path of a ray

Tree traversal kernel
Primitive intersection kernel
Material & Light Kernel
Shading Kernel
Select K Leaves
Select Next Leaf
Intersection found: Primitive ID
Compute light equation
Ray
Tree traversal
Ray-triangle intersect
Generate Shadow Rays
Shading
Generate Secondary Rays
Select Next Primitive
Shaded cluster is sampled

Shading

- **Deferred rendering style**

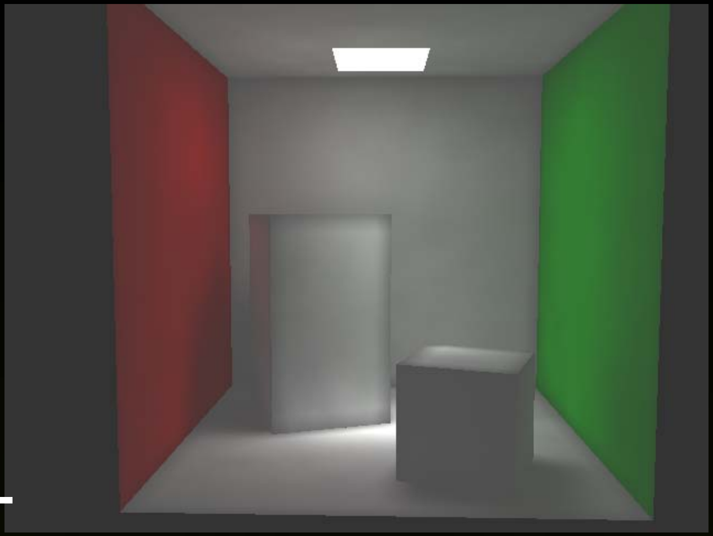


Ray-tracing & Global Illumination

- To simulate global illumination we can:
  - Path-tracing: trace multiple rays through each pixel
  - Photon mapping: trace photons from light source

Photon mapping

- Brute force solution:
  - Trace N photons from the light source
  - Create lists of photons for each Kd-tree node
  - For each visible pixel:
    - Collect photons that are within radius R



REYES Pipeline

REYES

- Micropolygonal rendering pipeline
- Industrial standard for high-quality rendering
- Widely used in film production

Research

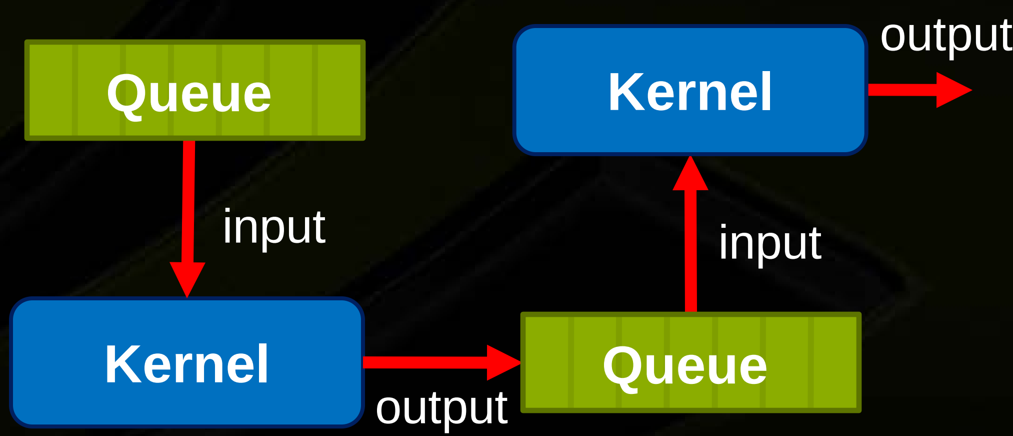
- [Zhou09]
- [Patney08]
- Problems of implementing REYES on CUDA are common to many tasks in parallel computing

Peculiarities

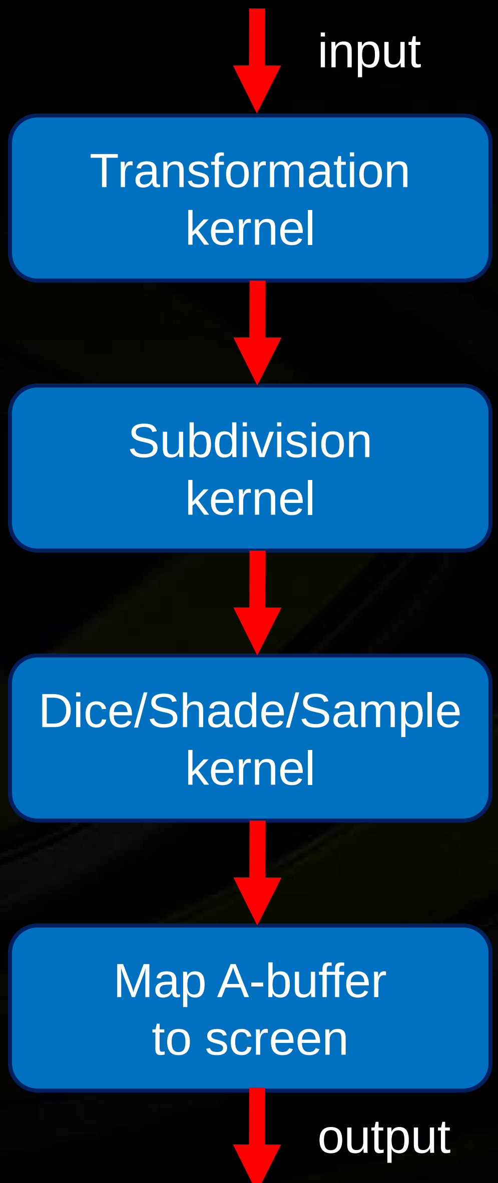
- A lot of uniform, non-divergent computations
- Natural parallelism exposed through bucketing
- Amount of work per input data element may vary significantly
- Some stages of pipeline can generate enormous amounts of data

Implementation Design

- **The idea is to implement REYES as a set of kernels which communicate through queues**
  - **This allows implementing advanced scheduling schemes**



Pipeline overview

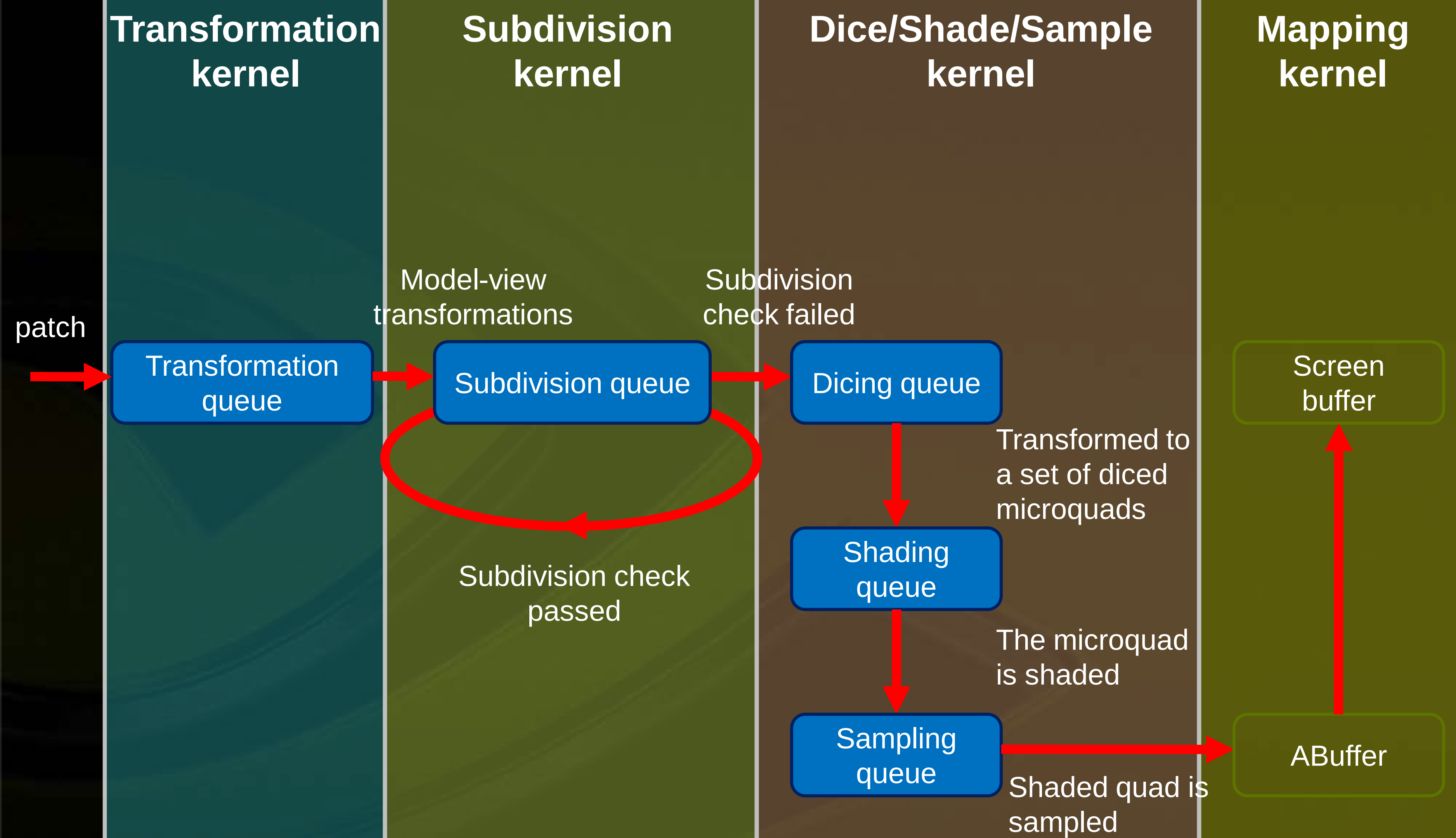


- **Transforms control points**
- **Bounds and splits patches**
- **Uberkernel which can do dicing, shading and sampling**
- **Map A-buffer to screen buffer**

Why queues?

- High-level memory access interface
  - *Push()/Pop()*-style access
  - Encapsulates complex memory management schemes
- Queue-specific tricks
  - Recursive processing
  - Workload management

Path of the Patch



Queues

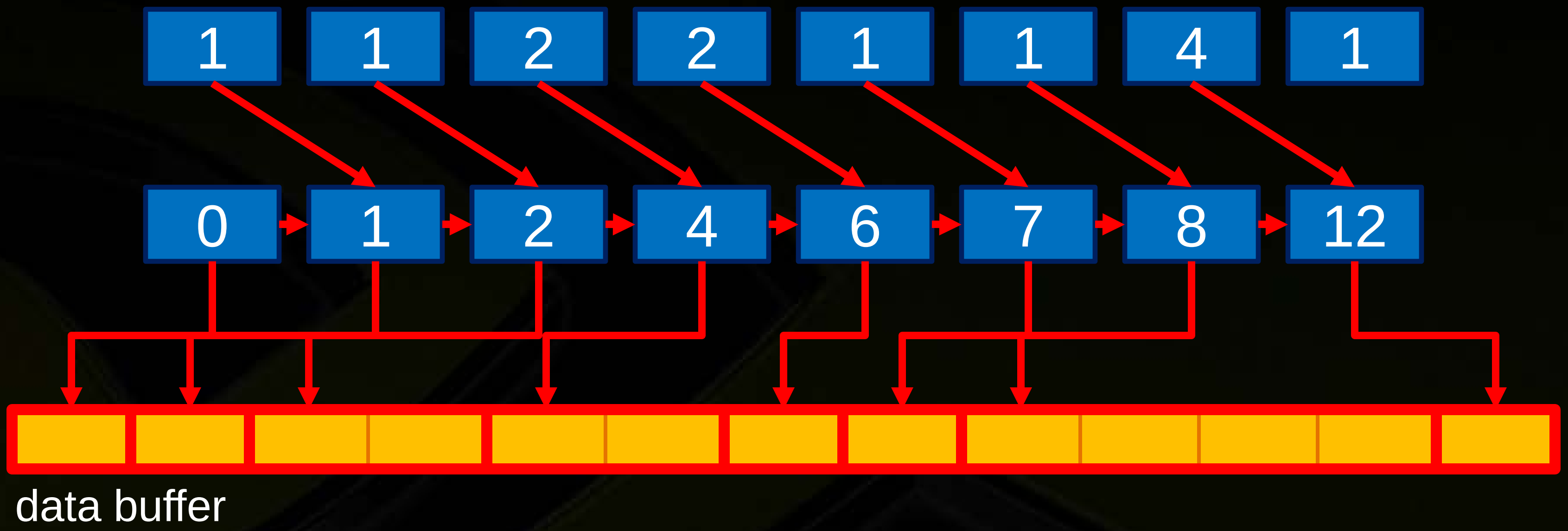
- **High-level memory access interface**
  - ***Map()/Unmap()* routines**
- **Implicitly perform scan inside *Map()* calls**
  - **All threads get access to memory synchronously**

Scan operation

- Prefix sum
- Fundamental operation used in GPU computing
- Allows managing the memory depending on the needs of each thread

Scan operation

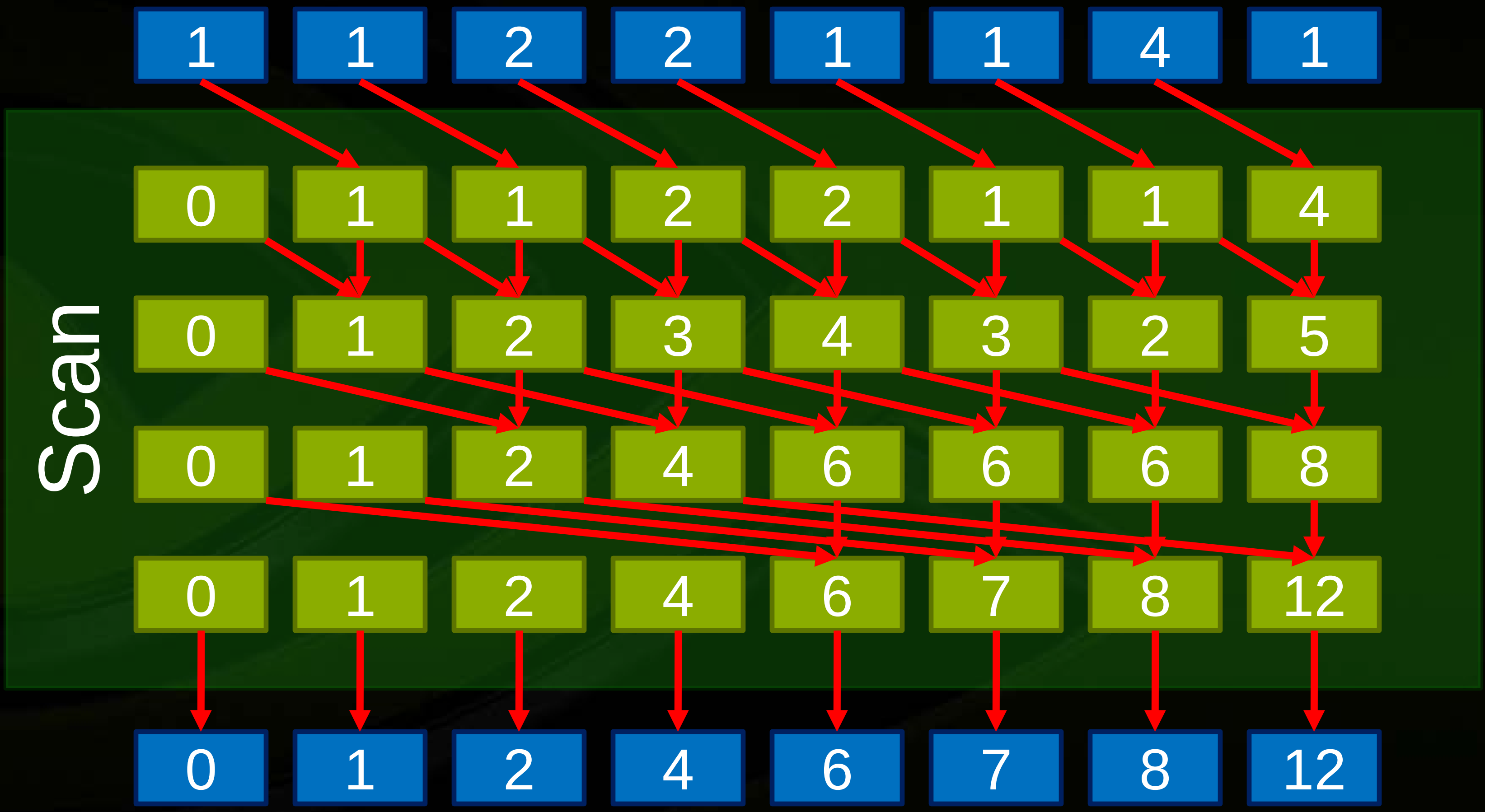
- Can be performed by sequentially summing elements



- This implementation is not suited for parallel execution

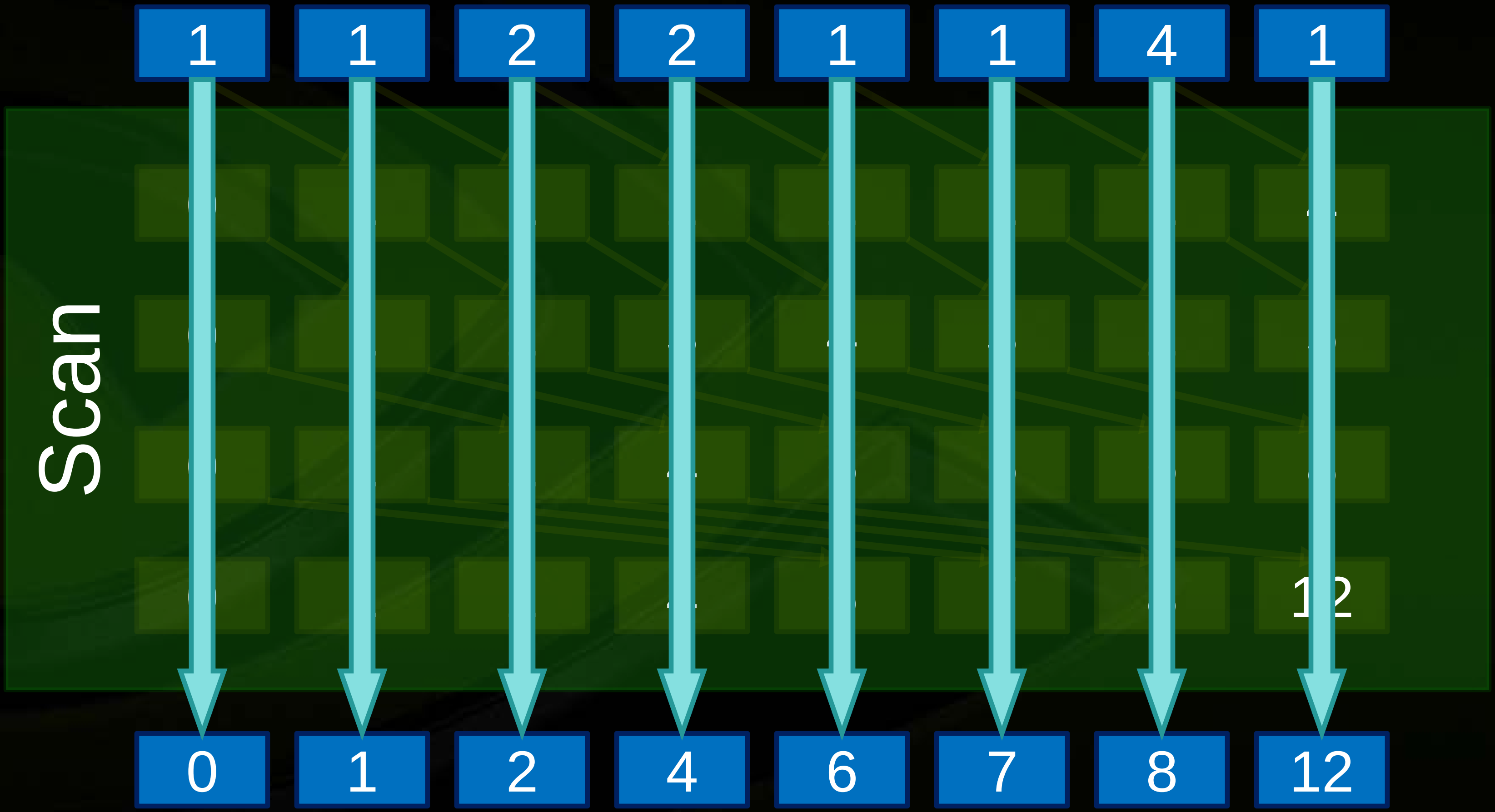
Scan operation

- The naïve (*n log(n)* complexity) scan approach is used



Scan operation

- The naïve (*n log(n)* complexity) scan approach is used



Scan operation

For more work efficient Scan approaches, see **[Harris07]**

Queues

	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7
Calling *Map()*	Map(**1**)	Map(**1**)	Map(**2**)	Map(**2**)	Map(**1**)	Map(**1**)	Map(**4**)	Map(**1**)

Queues

	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7
Calling *Map()*	Map(**1**)	Map(**1**)	Map(**2**)	Map(**2**)	Map(**1**)	Map(**1**)	Map(**4**)	Map(**1**)
Performing scan	**1**	**1**	**2**	**2**	**1**	**1**	**4**	**1**
	↓	↓	↓	↓	↓	↓	↓	↓
	0	**1**	**2**	**4**	**6**	**7**	**8**	**12**

Queues

	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7
Calling *Map()*	Map(**1**)	Map(**1**)	Map(**2**)	Map(**2**)	Map(**1**)	Map(**1**)	Map(**4**)	Map(**1**)
Performing scan	**1**	**1**	**2**	**2**	**1**	**1**	**4**	**1**
	0	**1**	**2**	**4**	**6**	**7**	**8**	**12**
Accessing data	Read(**0**)	Read(**1**)	Read(**2**)	Read(**4**)	Read(**6**)	Read(**7**)	Read(**8**)	Read(**12**)

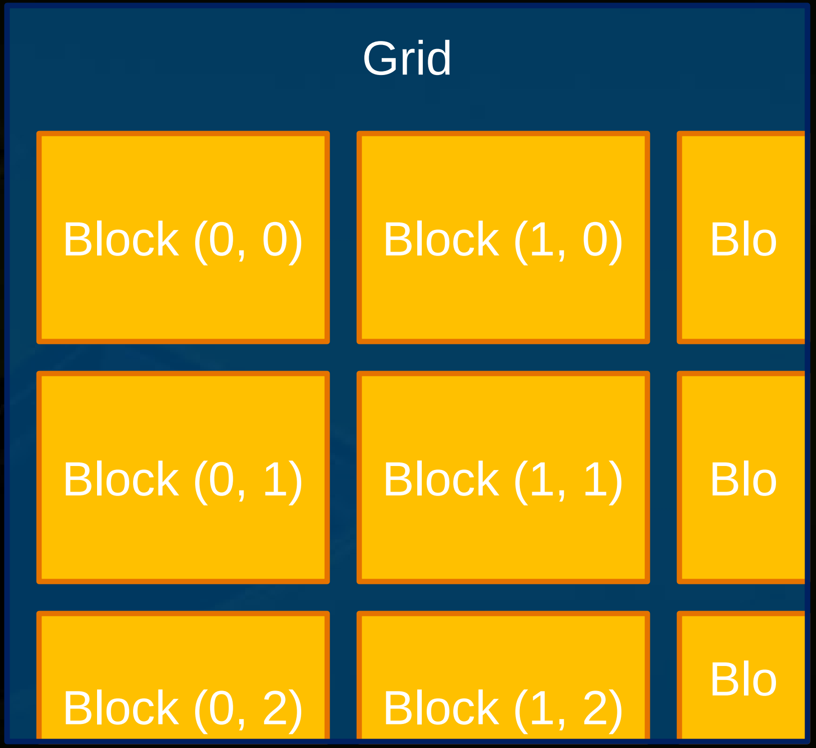
Advantages of using scan

- Used when a lot of threads need to access one shared resource
- Advantages:
  - All threads get access to data synchronously
  - No racing conditions and deadlocks

Kernels
in detail

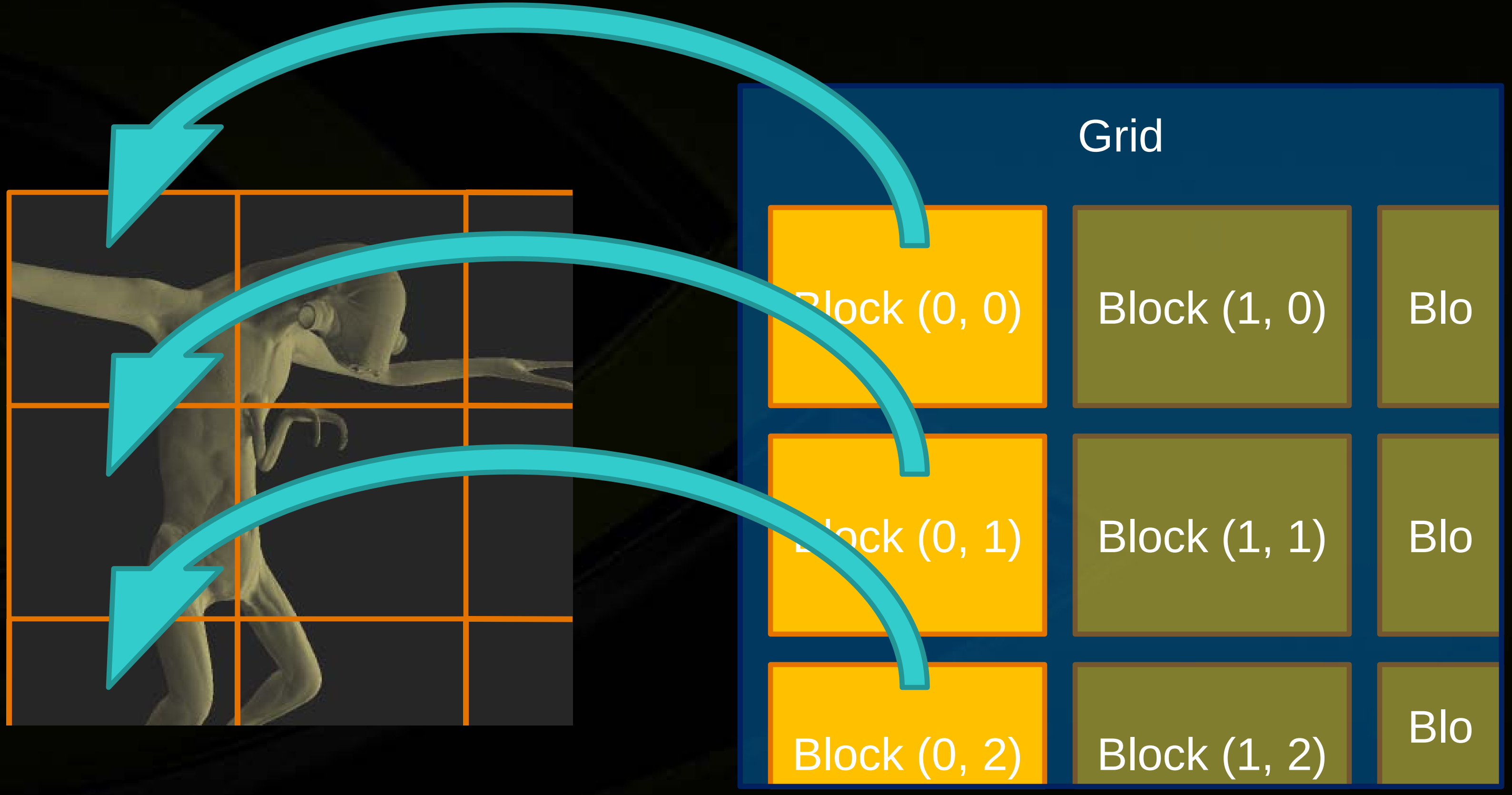
Bucketed rendering

- **Each block in a kernel grid processes one screen tile**



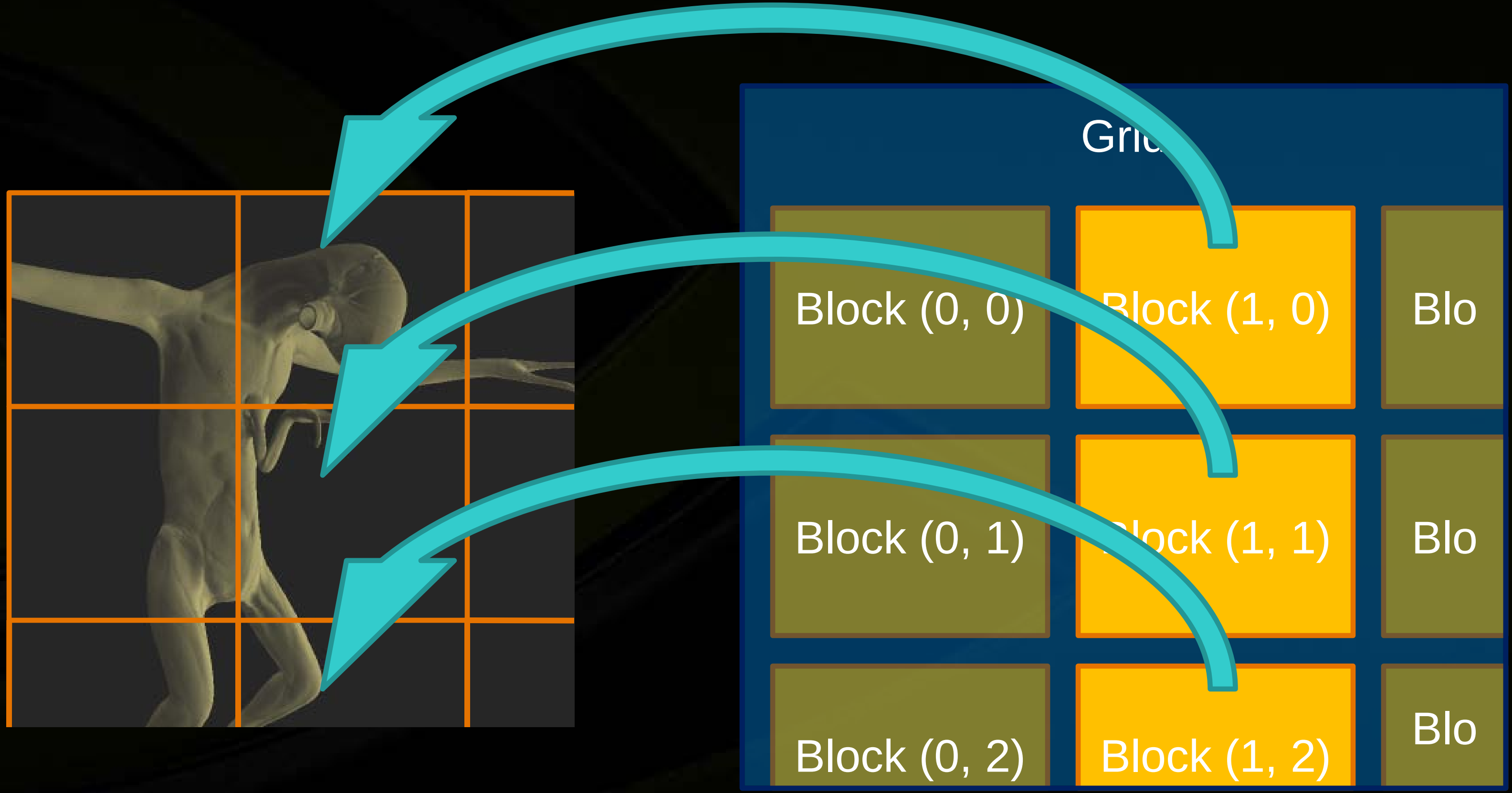
Bucketed rendering

- Each block in a kernel grid processes one screen tile



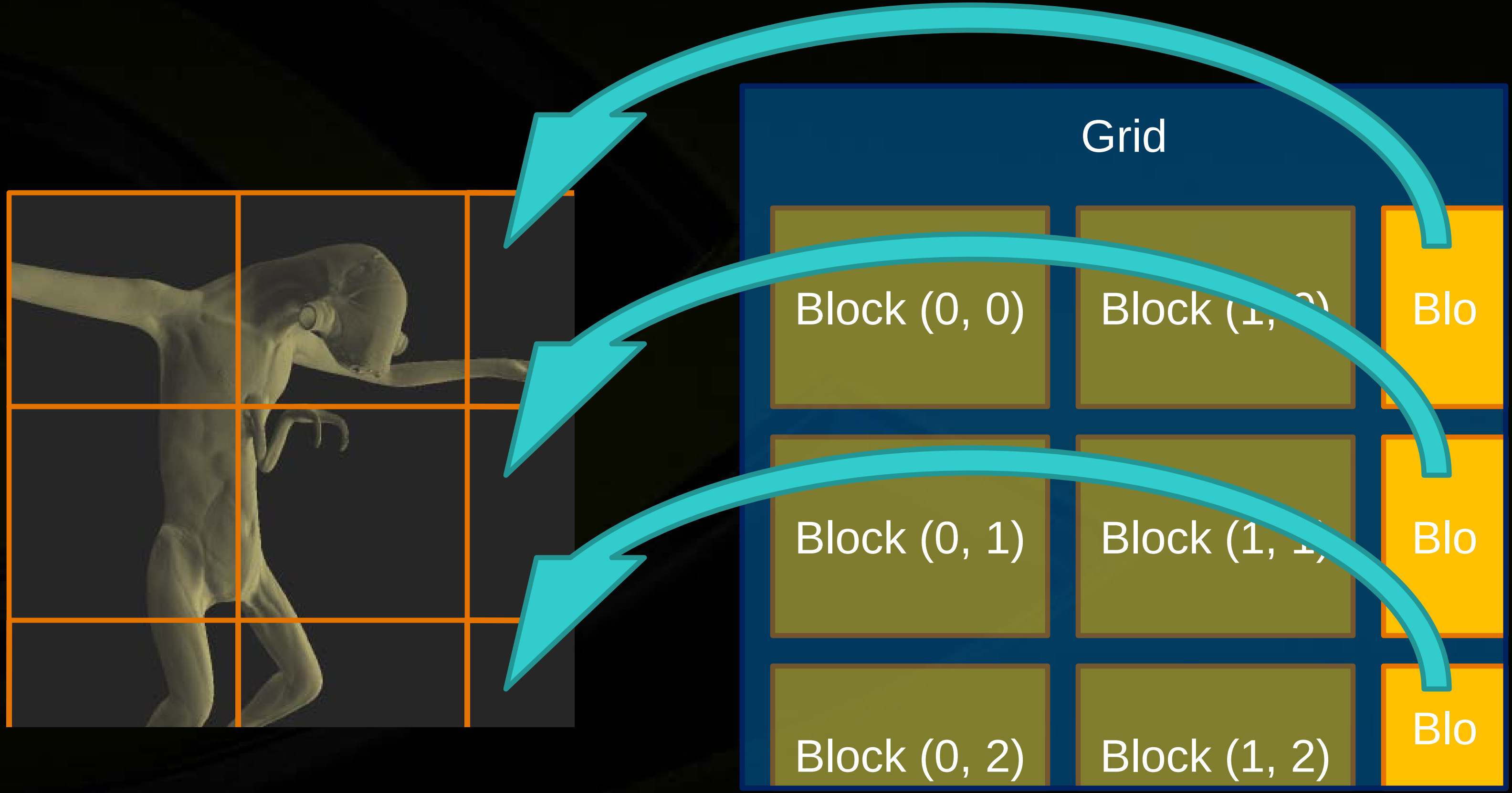
Bucketed rendering

- **Each block in a kernel grid processes one screen tile**

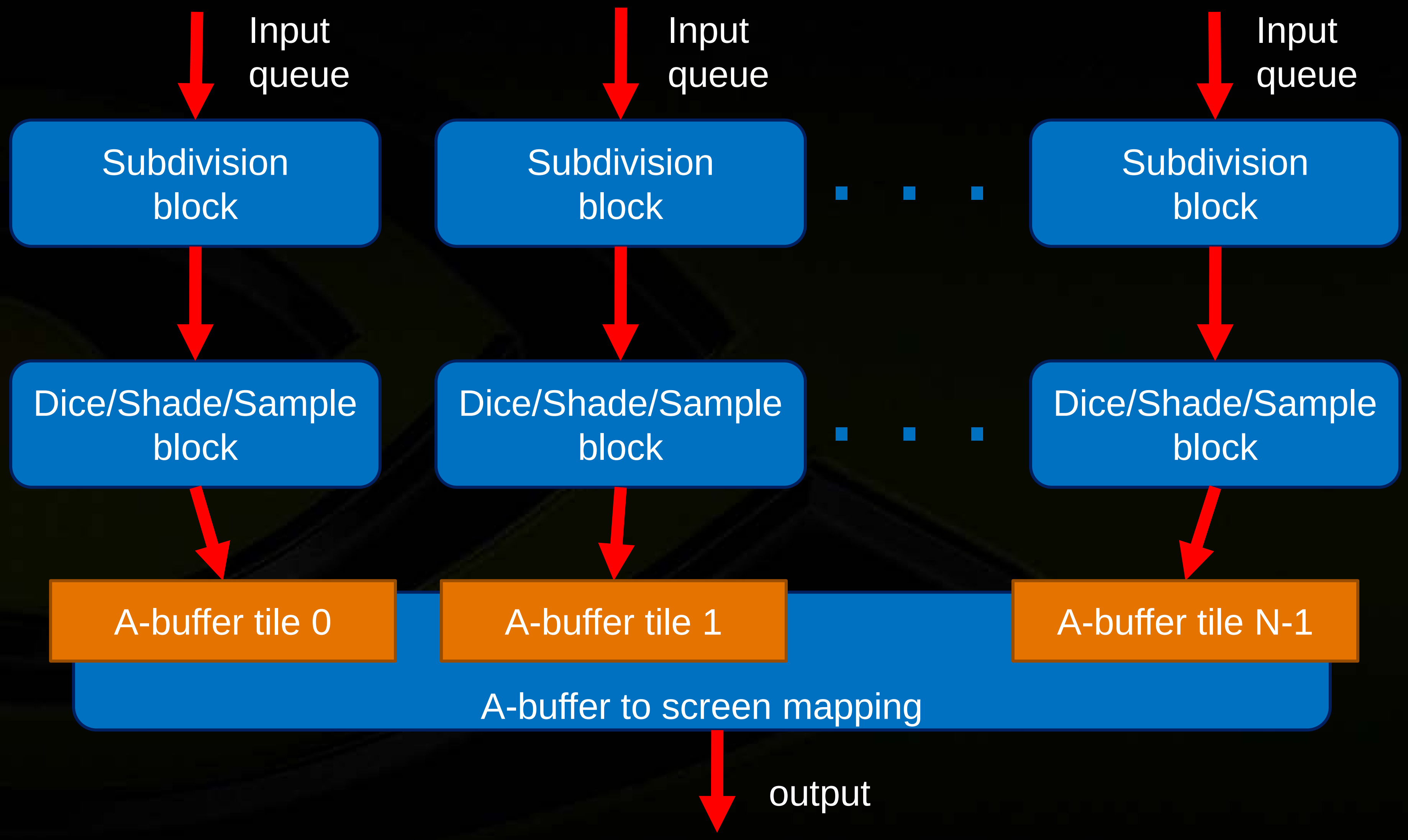


Bucketed rendering

- **Each block in a kernel grid processes one screen tile**



Bucketed rendering



Undetermined workload

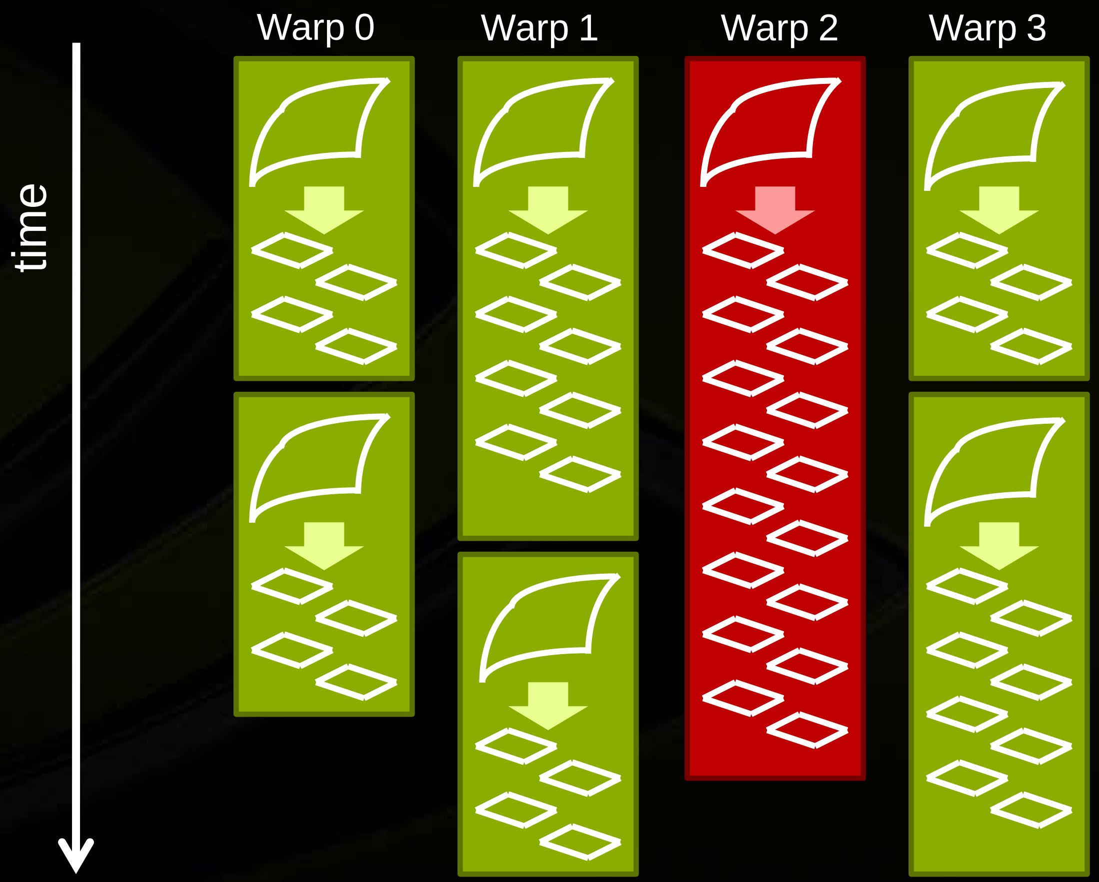
- Data parallelism is not well-suited when amount of input data is undetermined
- Need to schedule kernel execution on CPU

Persistent threads

- Launch as many threads as possible on GPU
- Use work stealing scheme to balance workload between threads
- Kernel works until the job is done

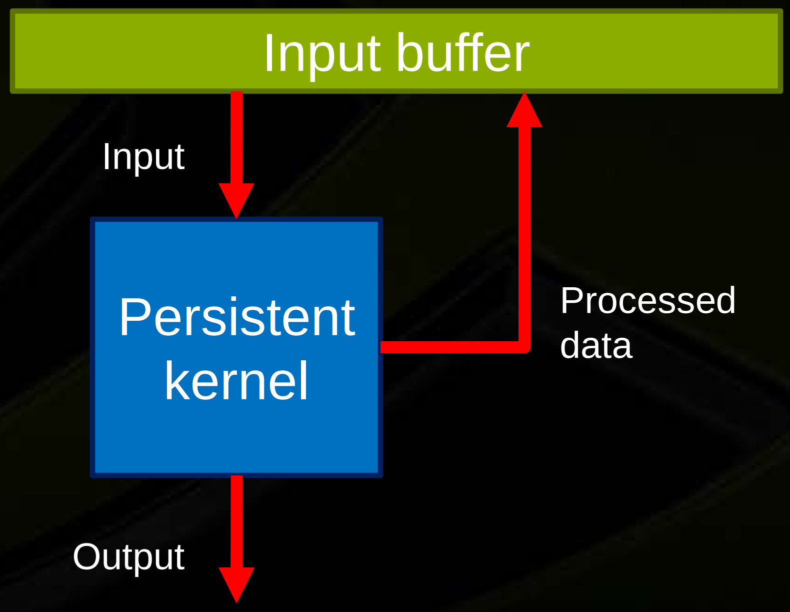
Persistent threads

- **Advantages**
  - **Workload is balanced between warps automatically**



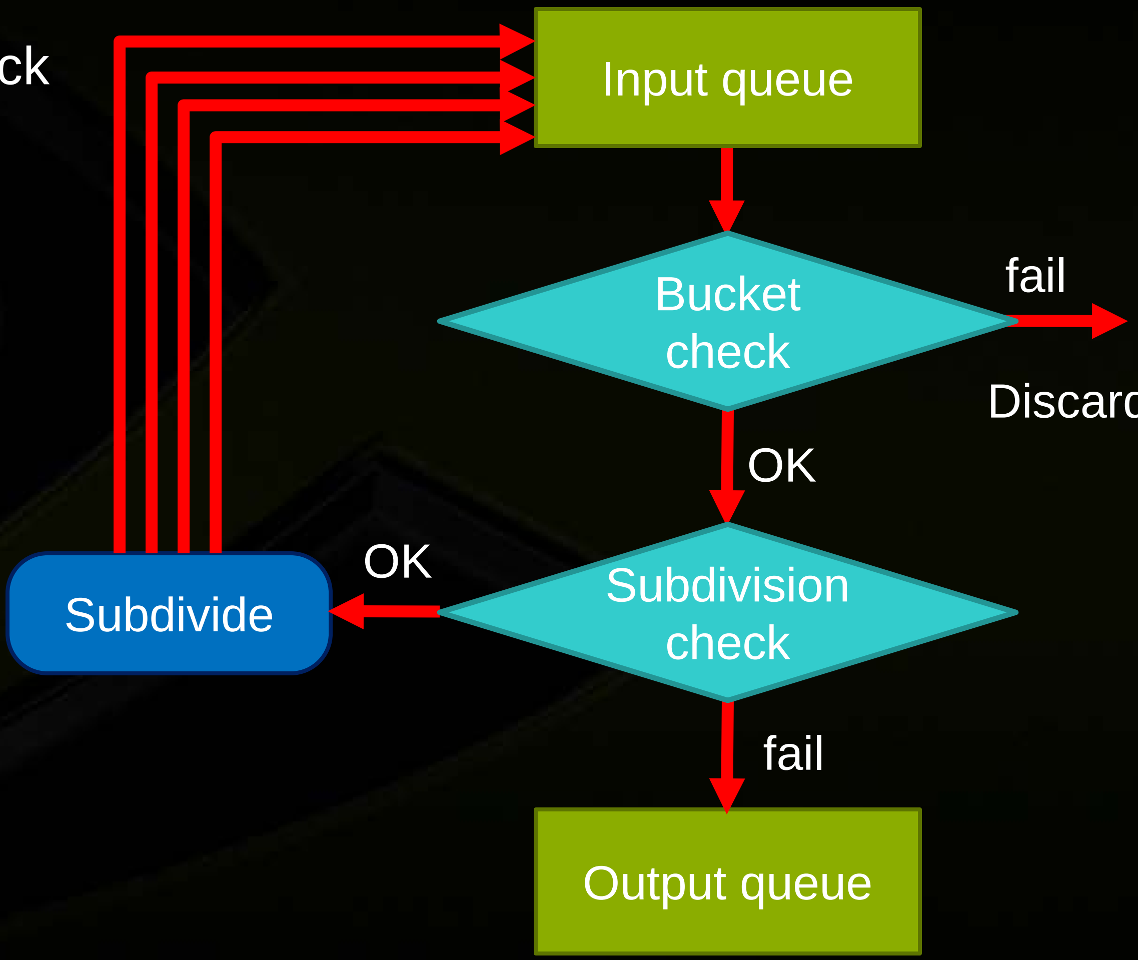
Persistent threads

- **Trick**
  - **Can write data back to the input buffer**
  - **Allows performing recursion**



Subdivision kernel

- Take a patch from input FIFO
- Perform bucket check
- Perform subdivision check
- Subdivide and store the results

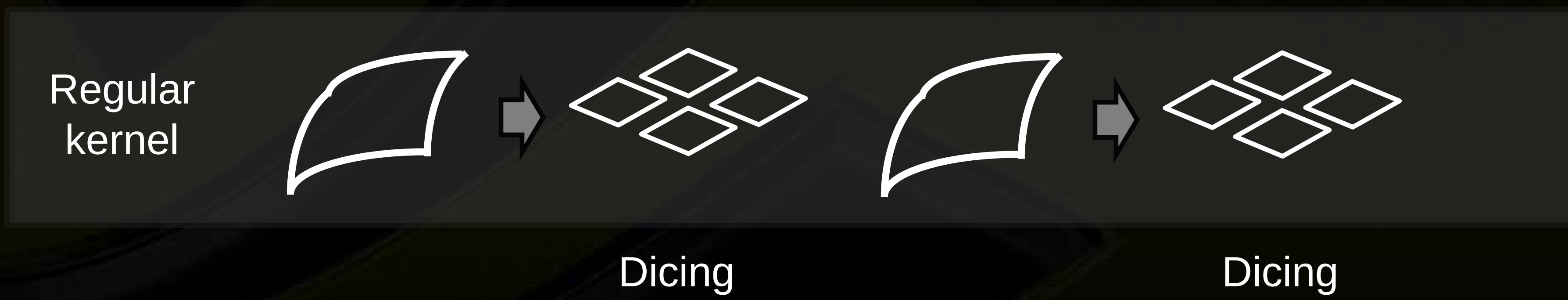


Working within limited memory

- Dicing kernel can generate enormous amounts of data
- Need to pipeline the computations to fit the limited amount of memory

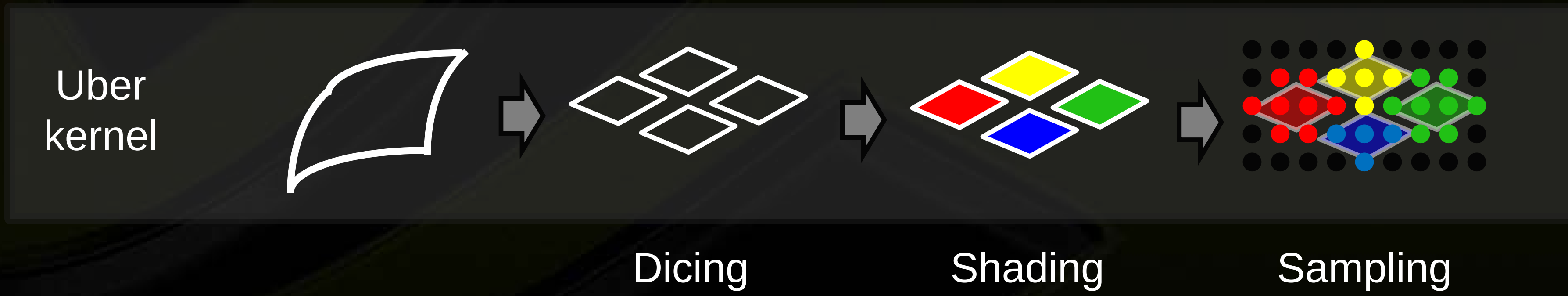
Uber-kernel

- Kernel capable of doing multiple types of job
- Can pipeline the computations



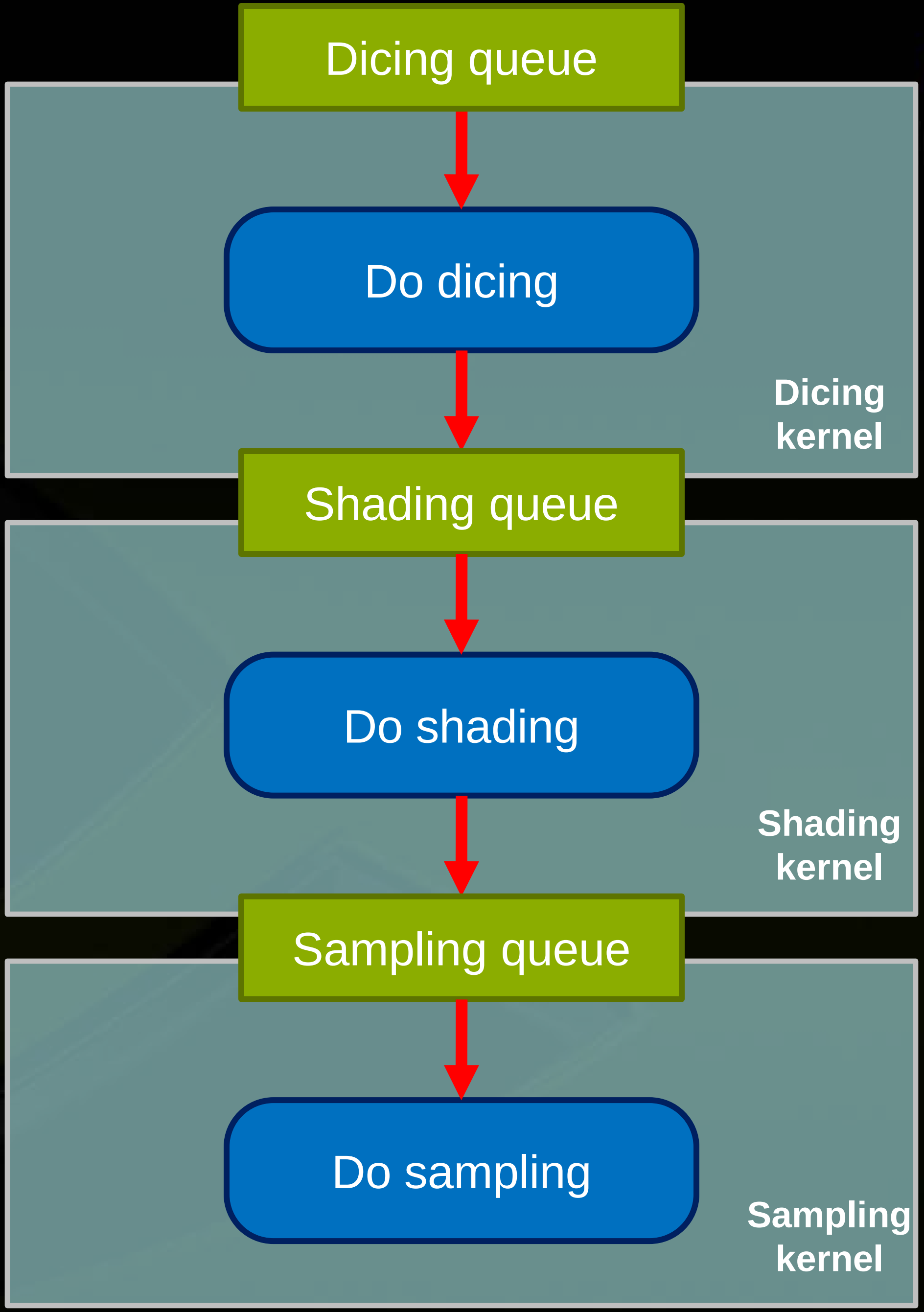
Dice/Shade/Sample kernel

- Uber-kernel capable of dicing, shading and sampling
- Switch between jobs is based on the state of input queues



Regular kernels

- Requires feedback from GPU to CPU and some scheduling logic on the host side

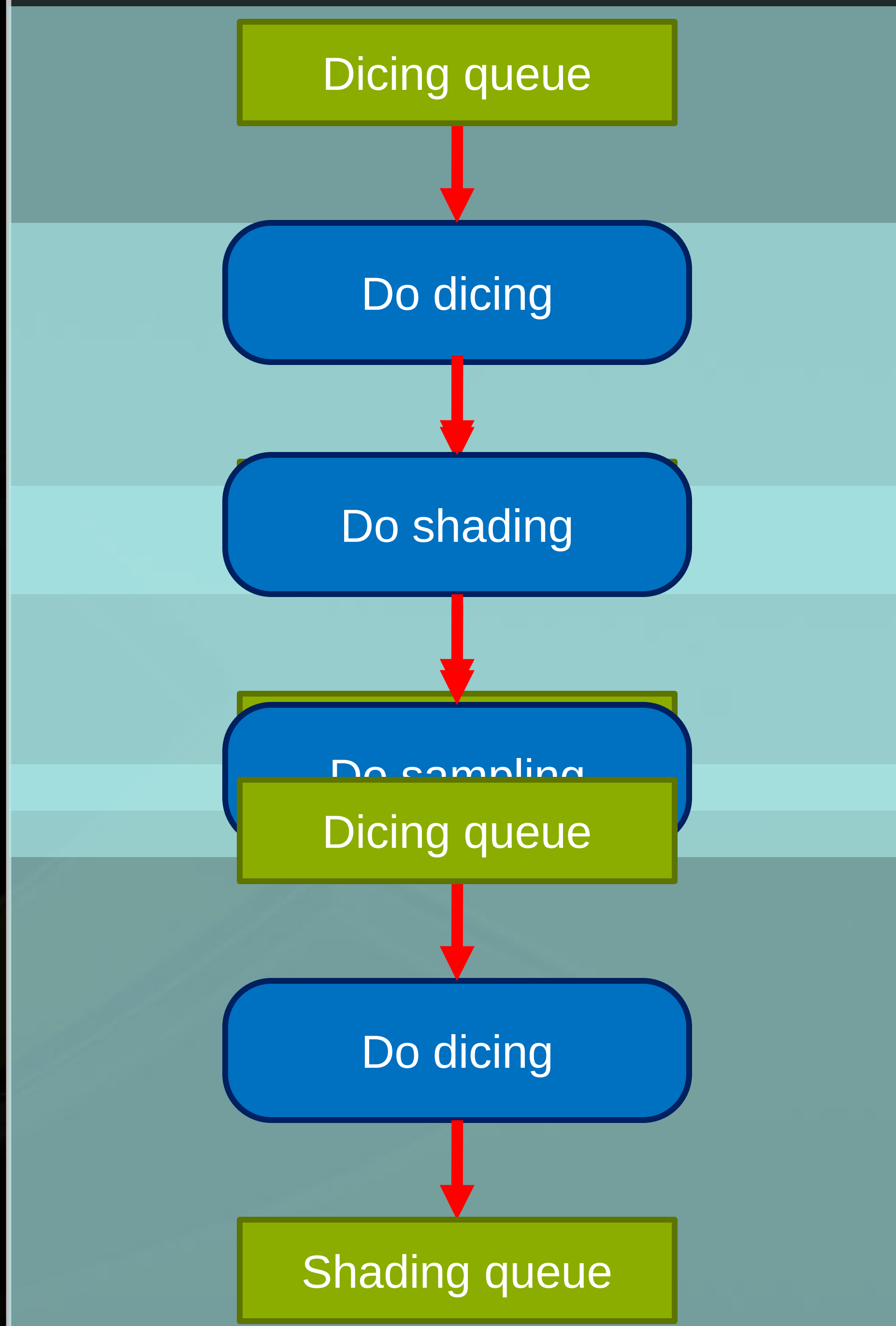


Uber-kernel

When shading queue is full, switch to shading

When sampling queue is full, proceed with sampling

If there is something left in dicing queue, return to dicing

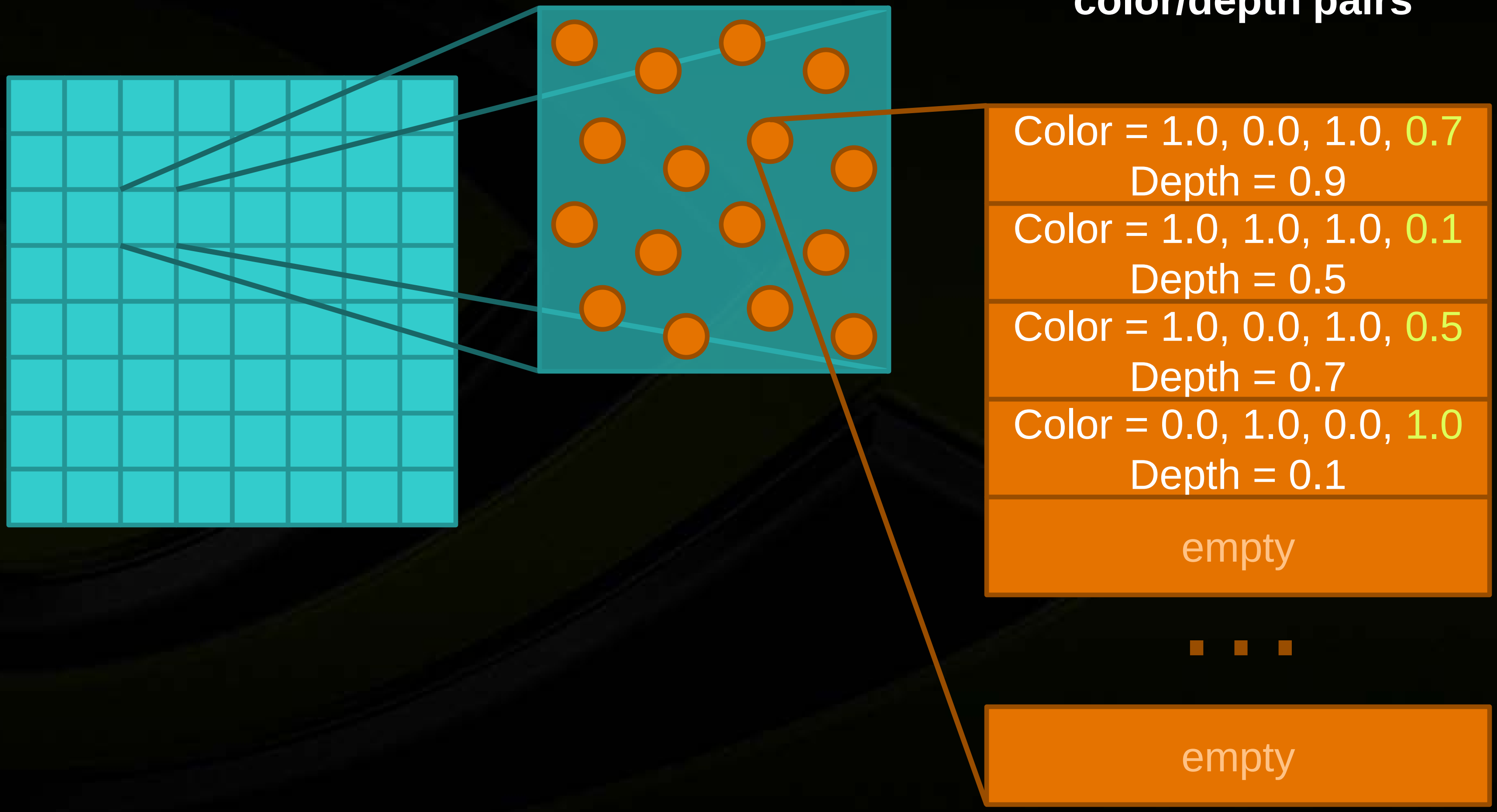


Dice/Shade/Sample kernel

- **Allows to pipeline the computation process within the limited amount of memory**
- **Doesn’t require CPU readbacks and additional host-side scheduling**

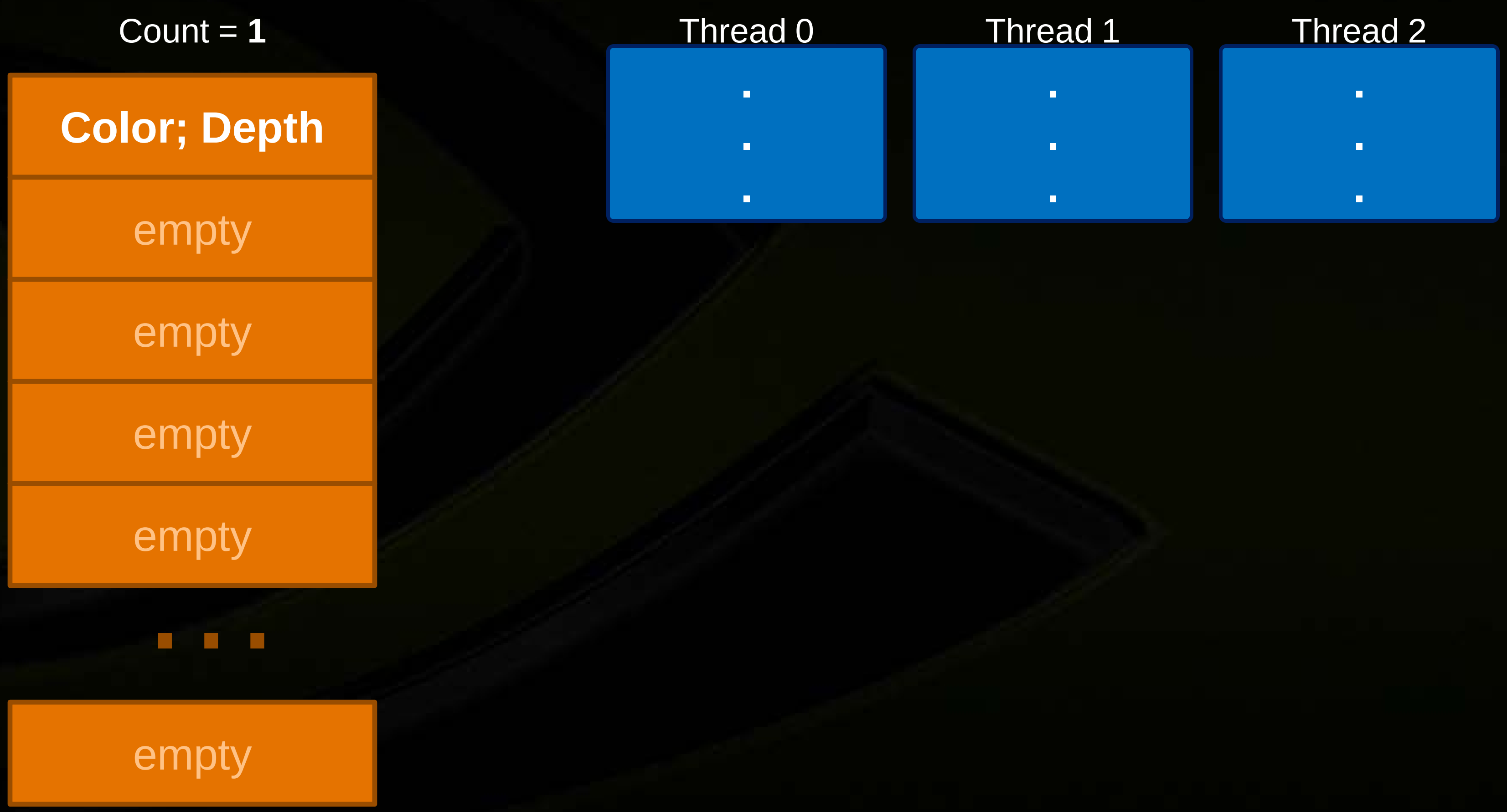
A-buffer

- A-buffer is a set of pixels
- Pixel is a set of samples
- Each sample can contain up to N color/depth pairs



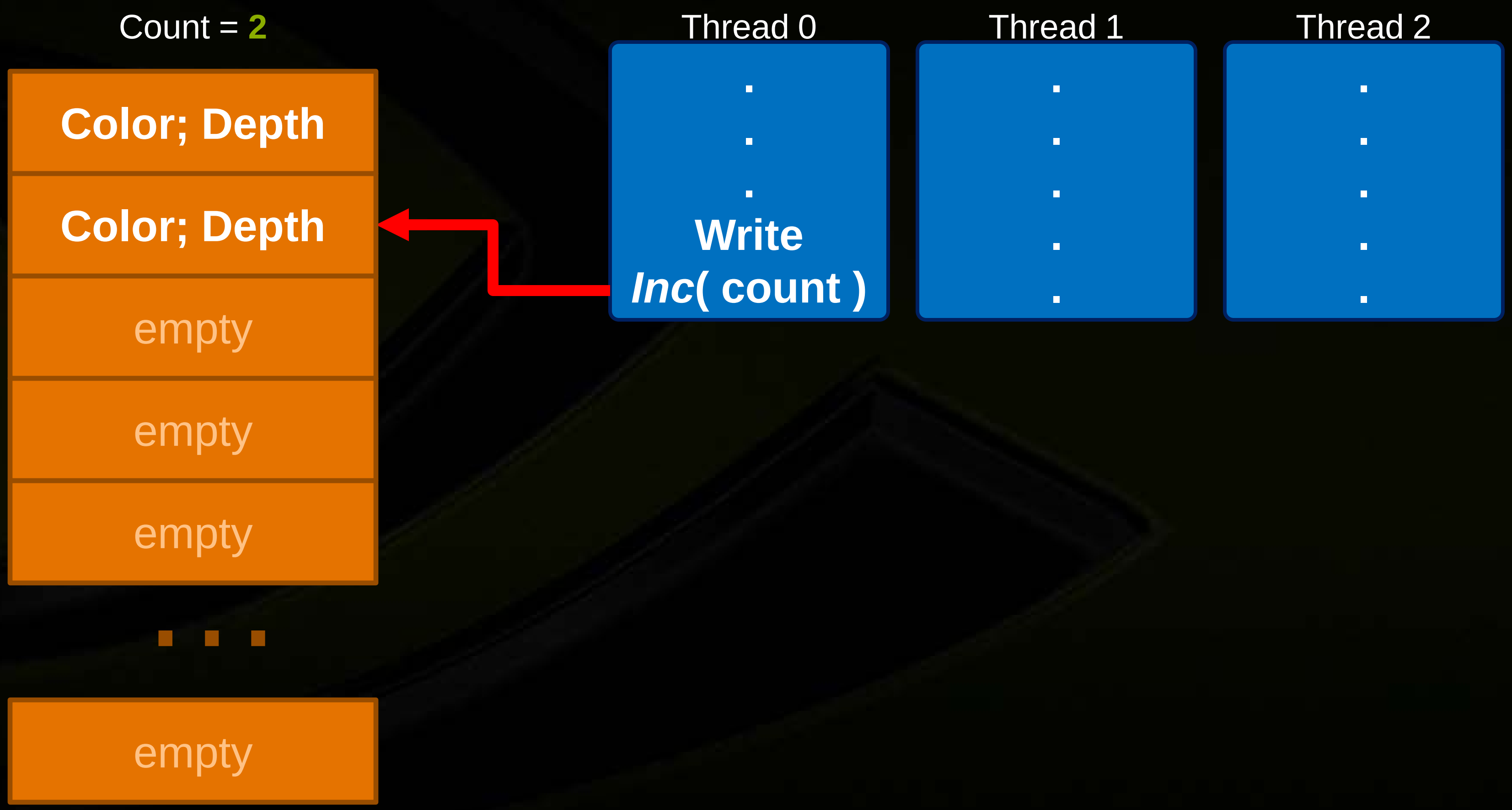
Synchronizing access to samples

- A-buffer is a shared resource
- Use atomics to secure the slot



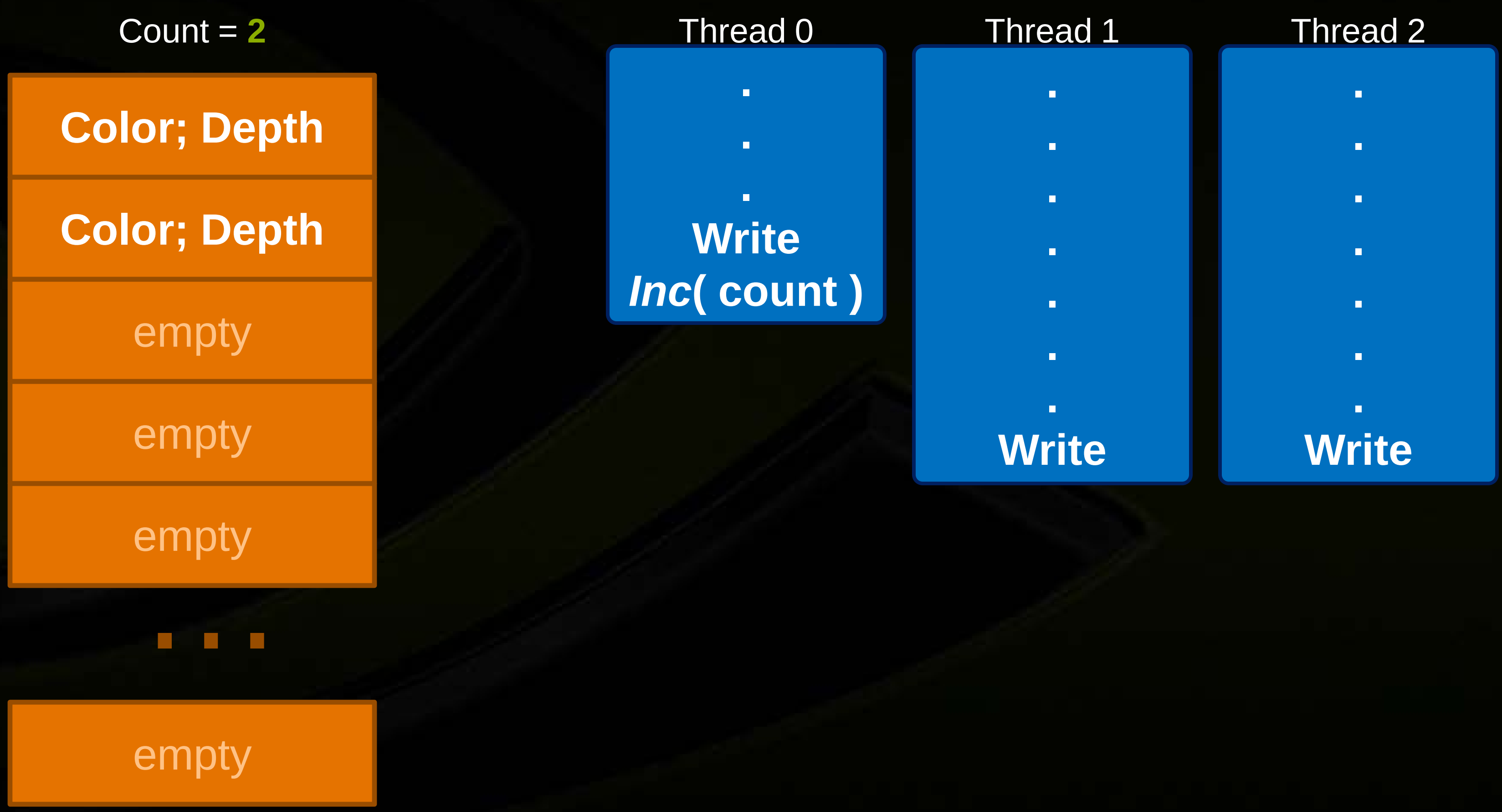
Synchronizing access to samples

- A-buffer is a shared resource
- Use atomics to secure the slot



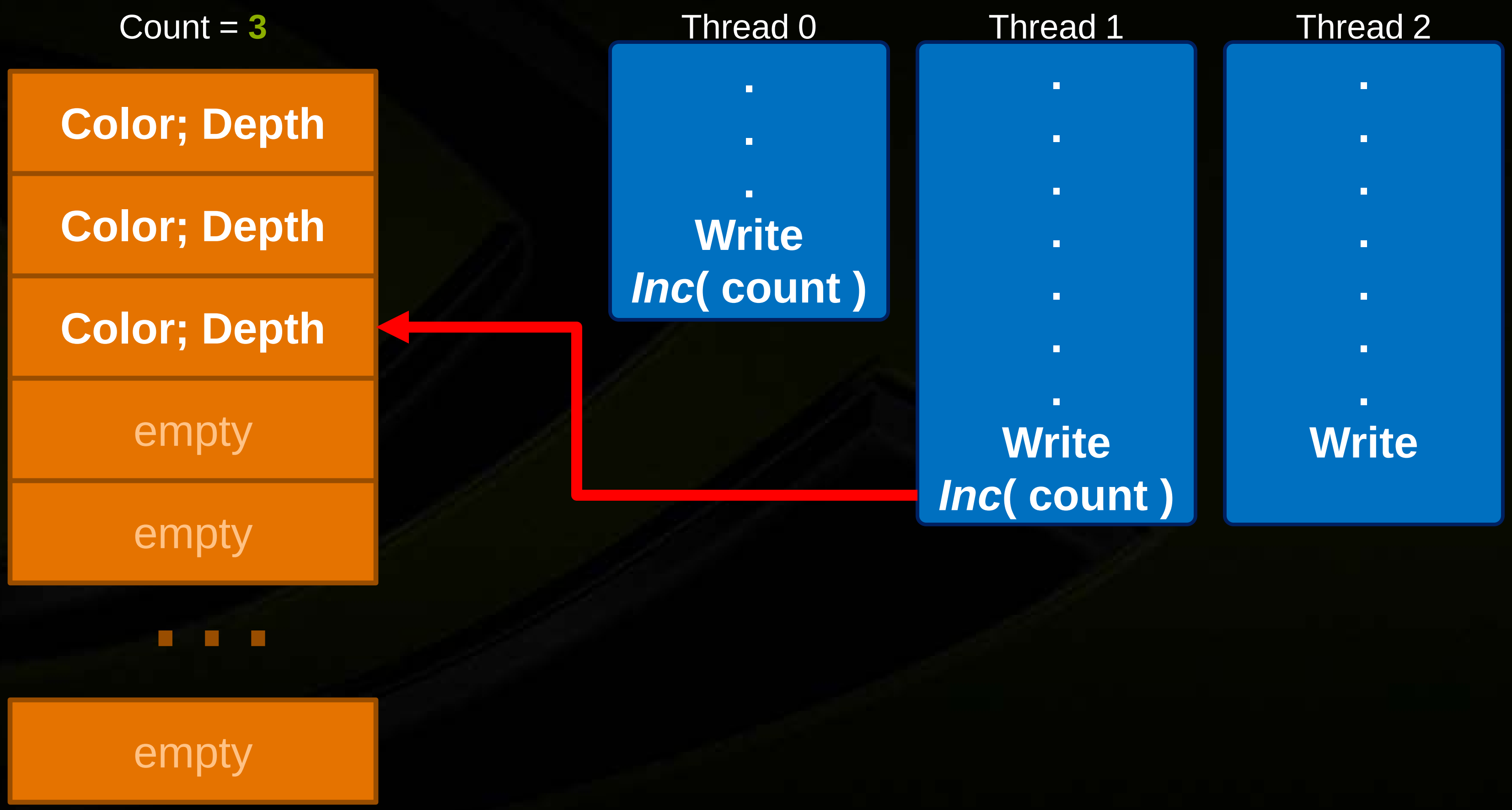
Synchronizing access to samples

- A-buffer is a shared resource
- Use atomics to secure the slot



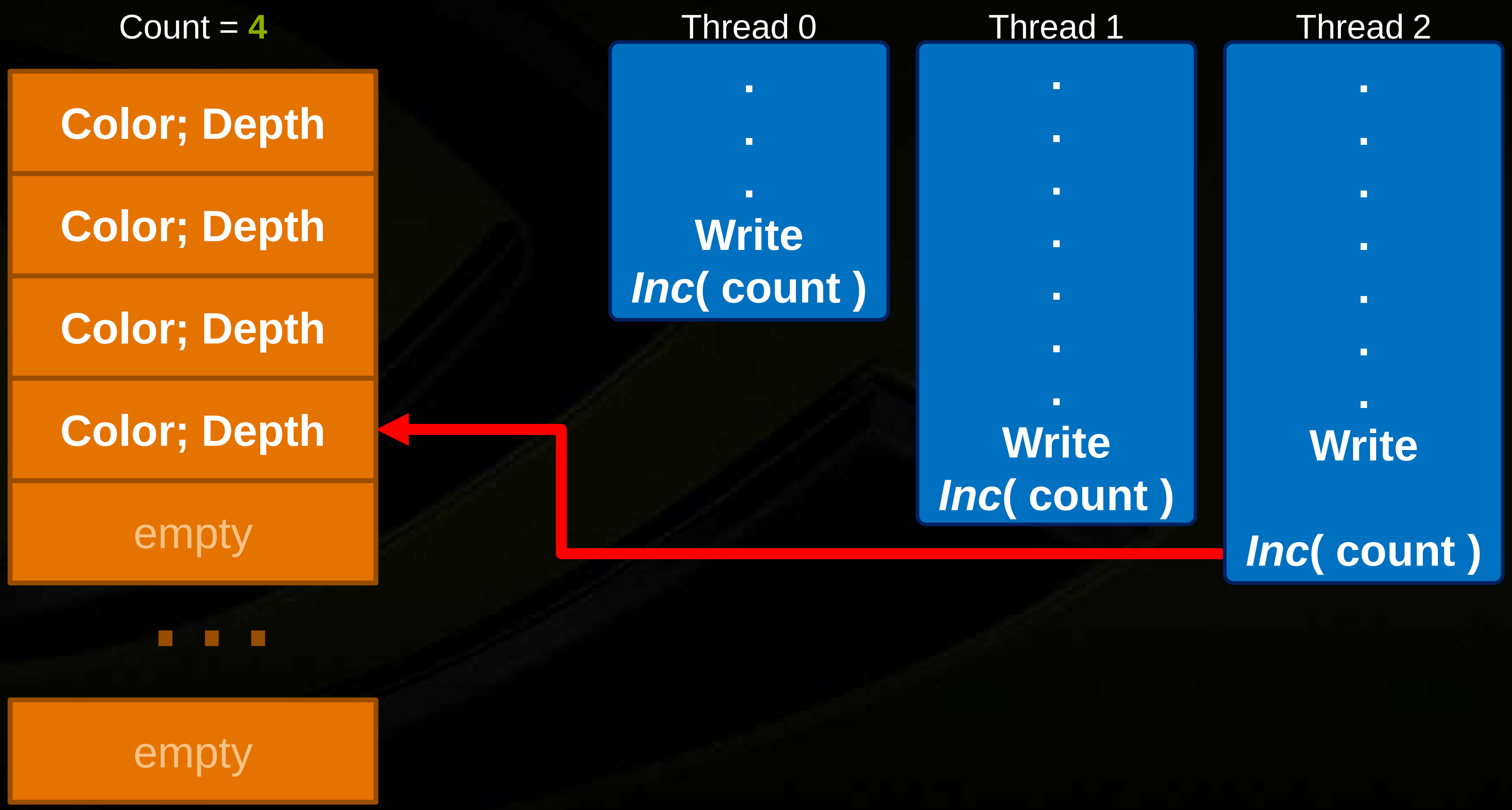
Synchronizing access to samples

- A-buffer is a shared resource
- Use atomics to secure the slot



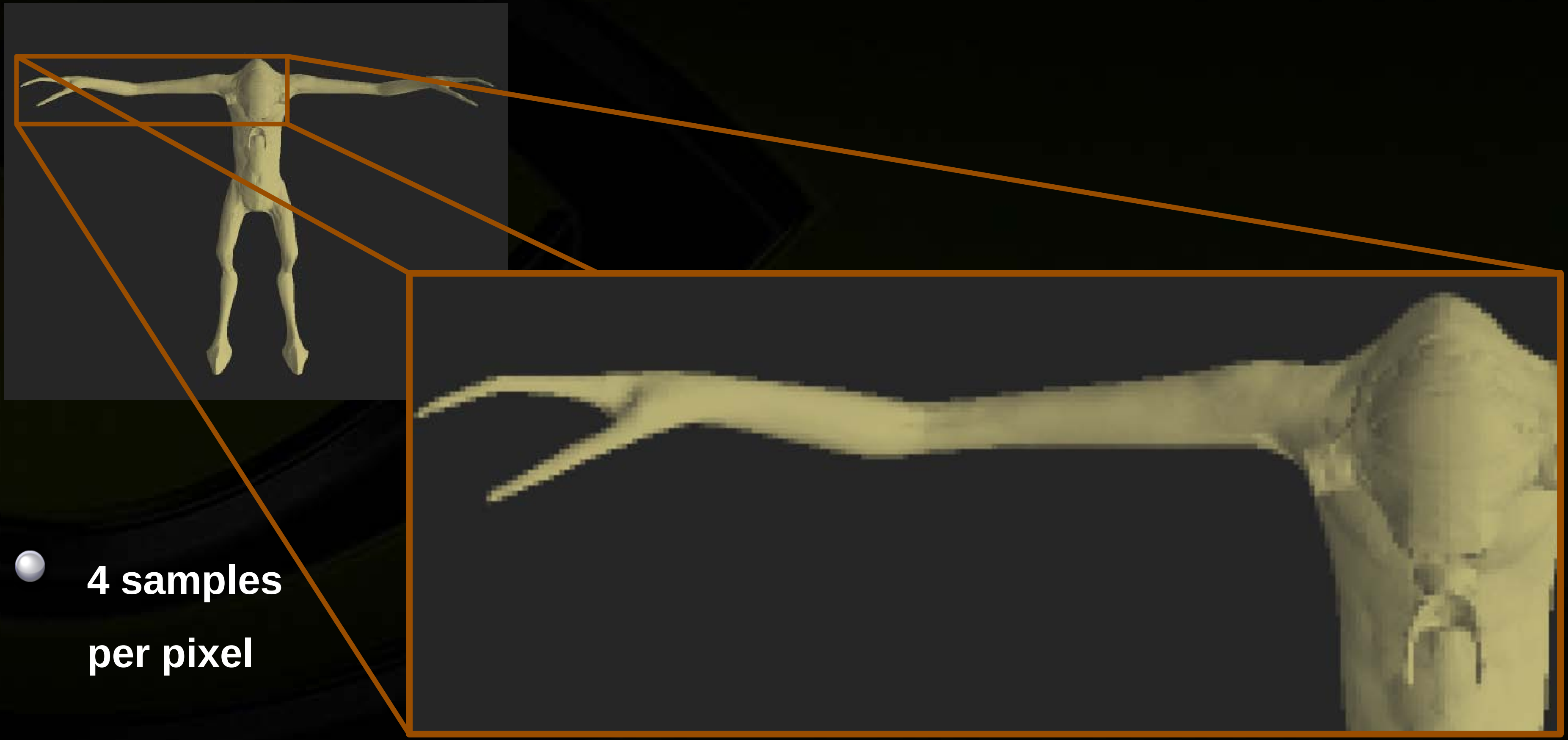
Synchronizing access to samples

- A-buffer is a shared resource
- Use atomics to secure the slot



Mapping A-buffer to screen

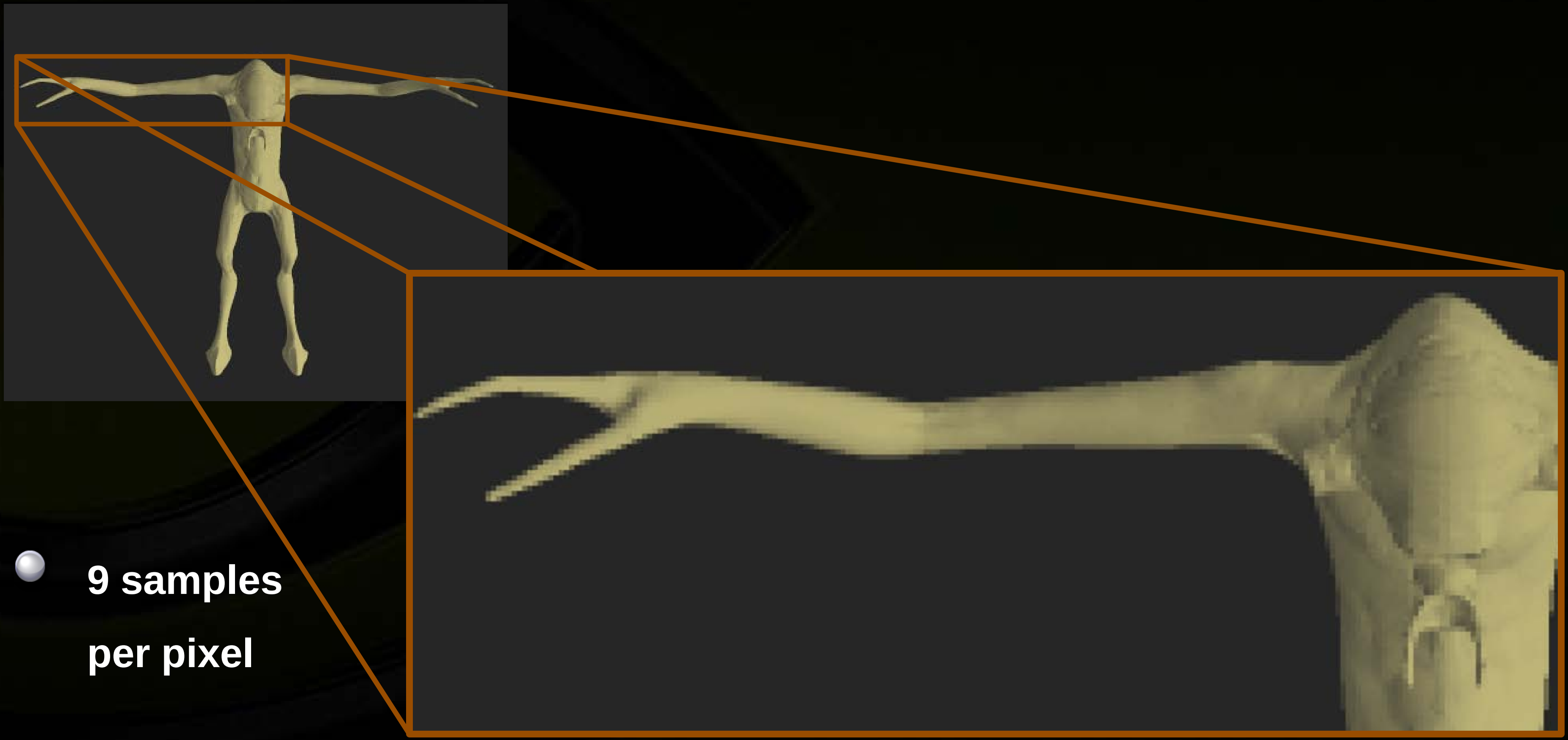
- Sort and blend all color/depth pairs per sample
- Sum all samples in pixel to get final pixel color



- 4 samples per pixel

Mapping A-buffer to screen

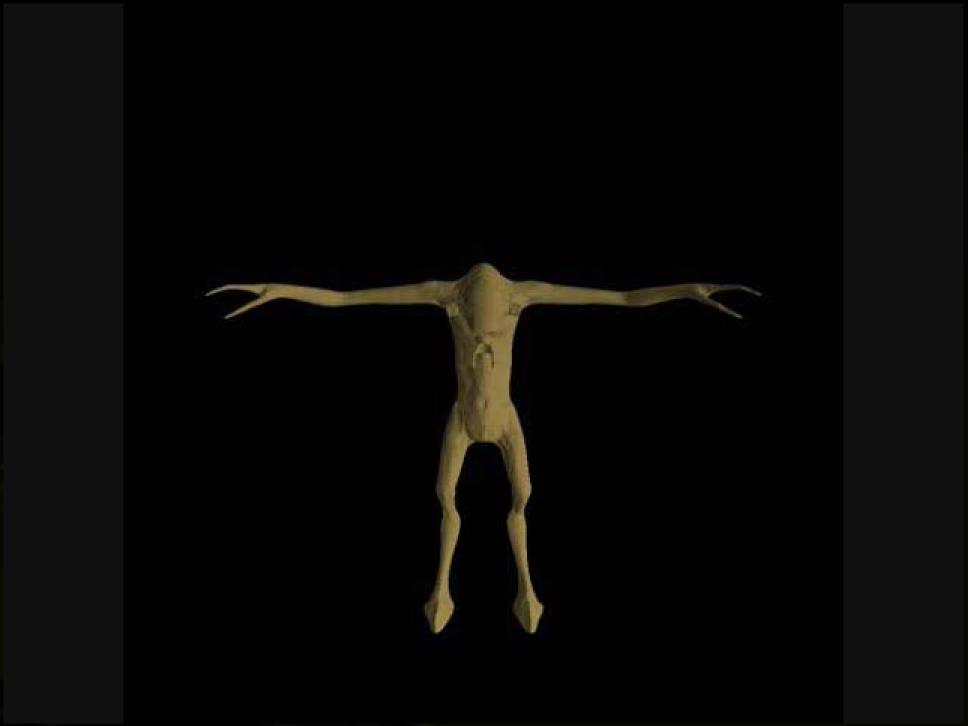
- Sort and blend all color/depth pairs per sample
- Sum all samples in pixel to get final pixel color



- 9 samples per pixel

Demo

- **Vortigaunt model**
- Resolution: 512x512 pixels
- 9 samples per pixel
- Shading rate: 0.25
  - ~4 microquads per pixel
- **Rendering time: ~150ms**



Vortigaunt model © Valve Software

Performance: persistent threads

- **Subdivision kernel via persistent threads**

vs

- **Regular threads with CPU readbacks**

	Regular threads	Persistent threads	Perf improvement
Bucket time	110ms	40ms	**3x**

Performance: uber-kernel

- **Dice/Shade/Sample kernel**

  vs

- **Separate kernels and CPU scheduling**

	Separate kernels	Uber-kernel	Perf improvement
Bucket time	500ms	110ms	**5x**

Future work

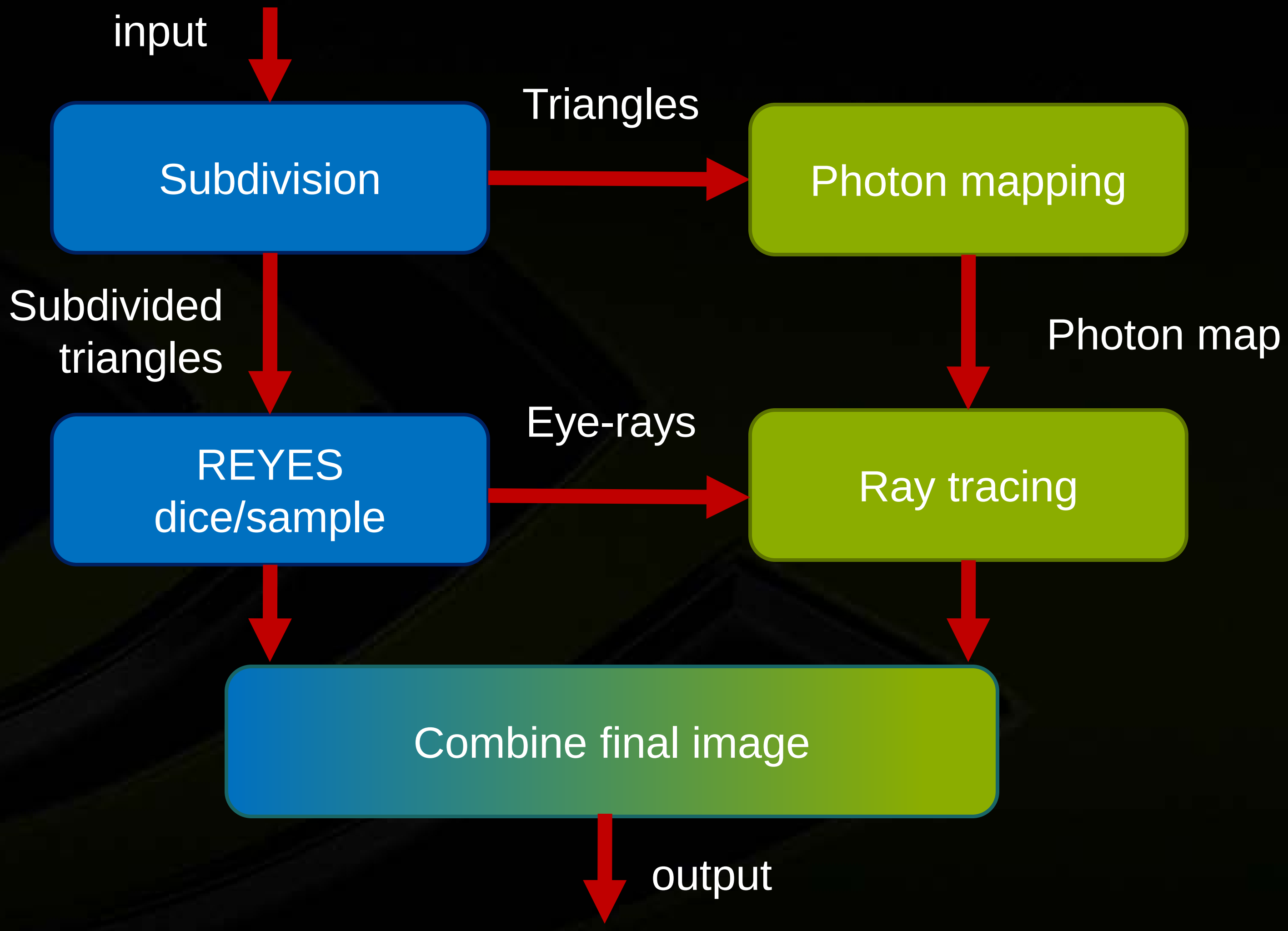
Future work

- Implement completely GPU-accelerated photorealistic renderer
- Use REYES pipeline to generate fine picture
- Use photon mapping to compute global illumination

Idea

- Use REYES to rasterize a picture
  - Compute **eye-rays** during rasterization
- During subdivision stage REYES pipeline would generate **a set of triangles**
- Use these triangles and eye-rays to compute GI

Idea


input
Subdivision
Triangles
Photon mapping
Subdivided triangles
Photon map
REYES dice/sample
Eye-rays
Ray tracing
Combine final image
output

Conclusion

Conclusion

- CUDA has proven to be a powerful instrument for computation-heavy tasks in computer graphics
- CUDA can also be used in tasks which generate non-uniform work and which are not easy to parallelize
- Scheduling mechanisms can be implemented in CUDA to allow better workload and memory balancing

Keep in mind

- Persistent threads
- Uber-kernels
- Work stealing
- Scan (Prefix sum)
- Histogram pyramids
- Tiling / Bucketing

Thanks!

Ray-tracing references

- [ZTTS06] Gernot Ziegller et al. “GPU Point List Generation through Histogram Pyramids”
- [HSHH07] Daniel Reiter Horn et al. “Interactive k-D Tree GPU Raytracing”
- [PH04] Matt Pharr, Greg Humphreys “Physically Based Rendering”
- [AL09] Timo Aila Samuli Laine. “Understanding the Efficiency of Ray Traversal on GPUs”
- [MT97] Tomas Moller, Ben Trumbore. “Fast Minimum Storage Ray / Triangle Intersection”

REYES references

- [Zhou09] Kun Zhou et al, “RenderAnts: Interactive REYES Rendering on GPUs”
- [Patney08] Anjul Patney, John D. Owens, “Real-Time Reyes-Style Adaptive Surface Subdivision”
- [Harris07] Mark Harris, “Parallel Prefix Sum (Scan) with CUDA” available at developer.nvidia.com

GPU Technology Conference

Sept 30 – Oct 2, 2009 – The Fairmont San Jose, California

Learn about the latest breakthroughs developers, engineers and researchers are achieving on the GPU

- Learn about the seismic shifts happening in computing
- Preview disruptive technologies and emerging applications
- Get tools/techniques to impact mission critical projects now
- Network with experts and peers from across several industries

Special for SIGGRAPH Attendees!

Full Conference Pass for only $400!

(offer expires 8/21)

Register and Learn More: www.nvidia.com/gtc

Use registration code FC300SIG90

NVIDIA.

GPU TECHNOLOGY CONFERENCE

Confirmed Sessions @ GPU Tech Conf

Sept 30 – Oct 2, 2009 – The Fairmont San Jose, California

KEYNOTES:

Opening

Jen-Hsun Huang
Co-Founder / CEO
NVIDIA

Day 2

Hanspeter Pfister
Professor
Harvard

Day 3

Richard Kerris
CTO
Lucasfilm

GENERAL SESSIONS:

- **Hot Trends in Visual Computing: Computer Vision, Augmented Reality, Visual Analytics, Interactive Ray Tracing**
- **Breakthroughs in High Performance Computing: Energy, Medical Science, Supercomputing, Research**

Conference Sessions Covered Topics:

- **3D**
- **Algorithms & Numerical Techniques**
- **Astronomy & Astrophysics**
- **Computational Finance**
- **Computational Fluid Dynamics**
- **Computational Imaging**
- **Computer Vision**
- **Databases & Data Mining**
- **Embedded & Mobile**
- **Energy Exploration**
- **Film**
- **GPU Clusters**
- **High Performance Computing**
- **Life Sciences**
- **Machine Learning & Artificial Intelligence**
- **Medical Imaging & Visualization**
- **Molecular Dynamics**
- **Physical Simulation**
- **Programming Languages & APIs**
- **Advanced Visualization**

Learn more and register: www.nvidia.com/gtc