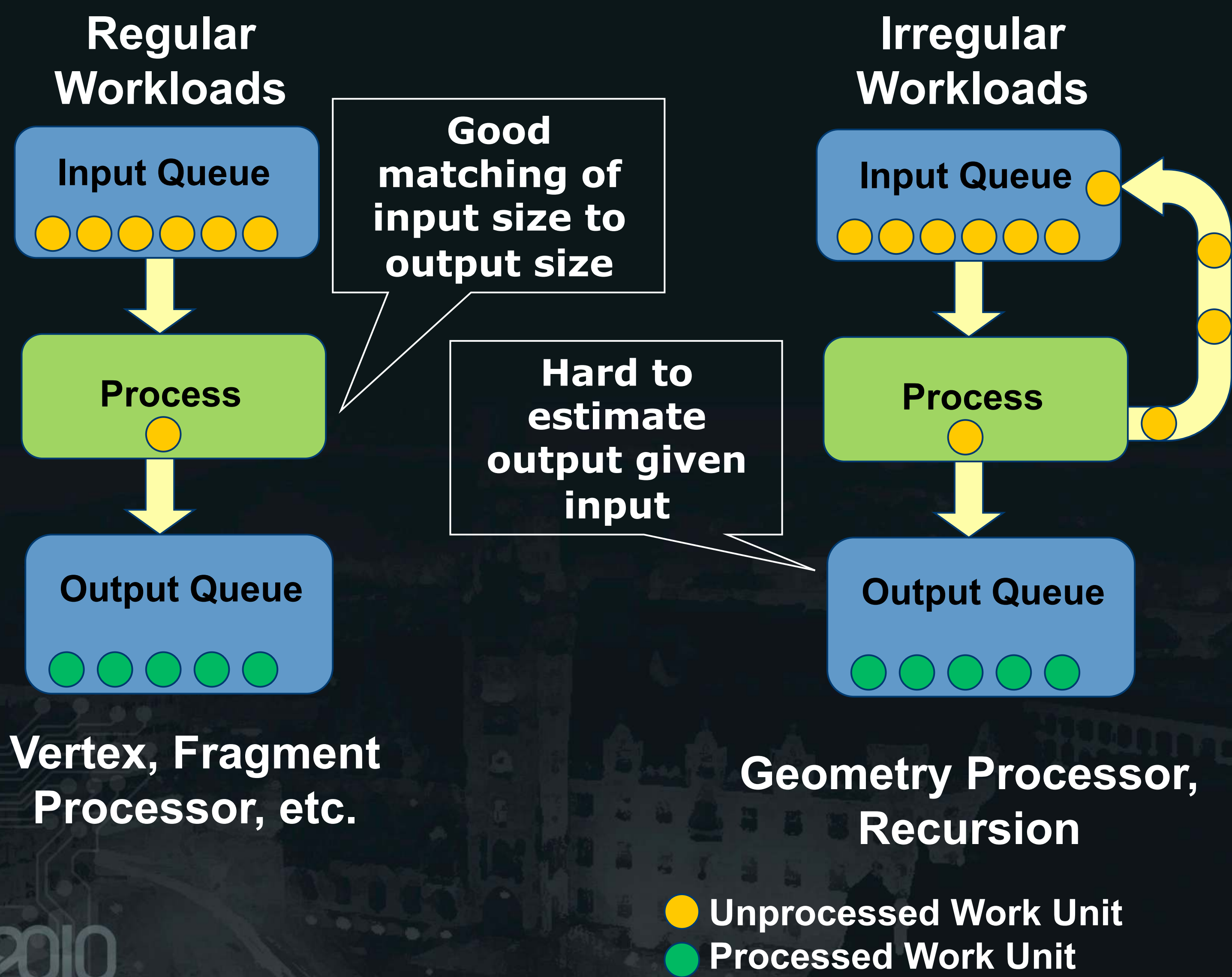
Task Management for Irregular-Parallel Workloads on the GPU

Stanley Tzeng, Anjul Patney, and John D. Owens
University of California, Davis

Introduction – Parallelism in Graphics Hardware



Motivation – Programmable Pipelines

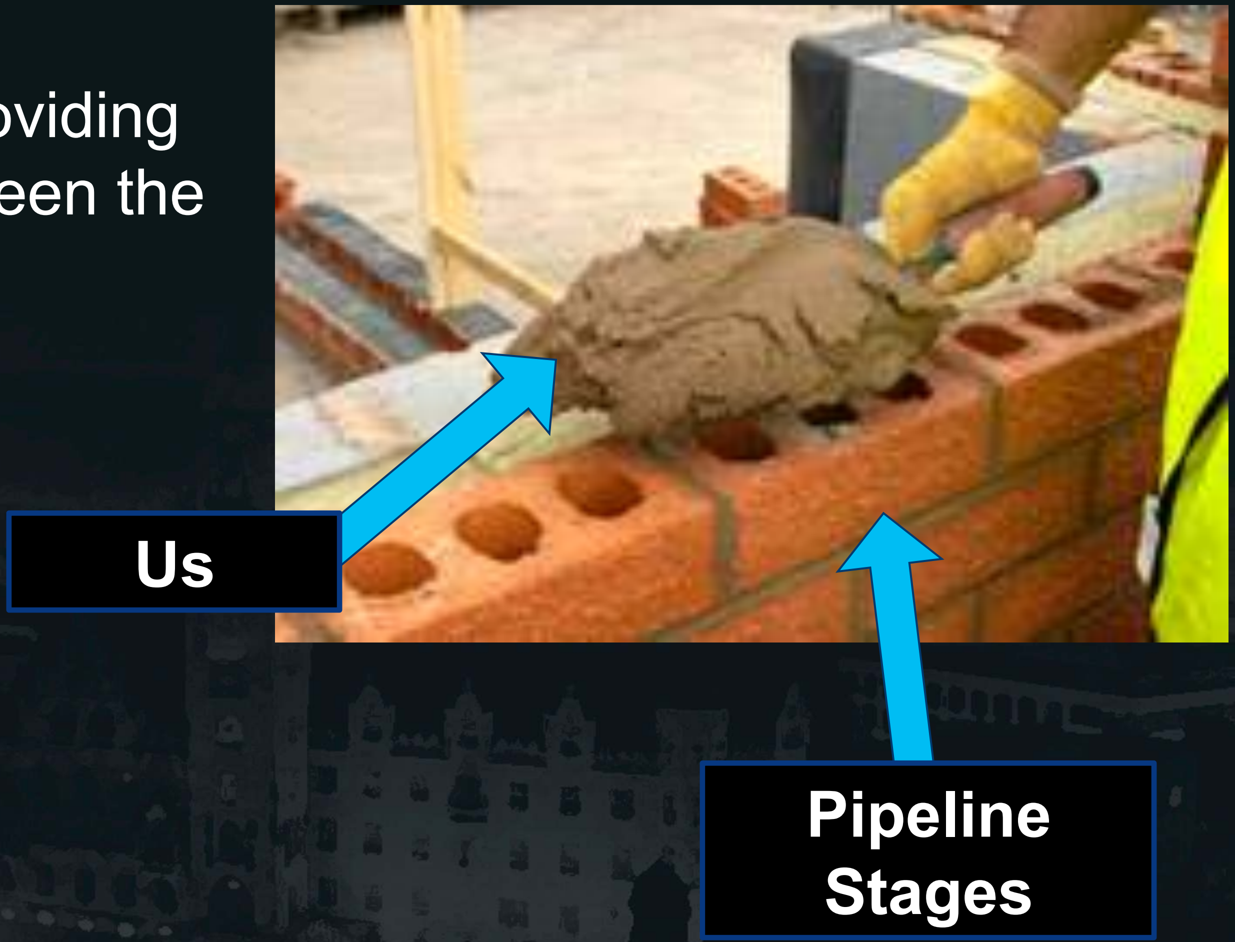
- Increased programmability on GPUs allows different programmable pipelines on the GPU.
- We want to explore how pipelines can be efficiently mapped onto the GPU.
  - What if your pipeline has irregular stages ?
  - How should data between pipeline stages be stored ?
  - What about load balancing across all parallel units ?
  - What if your pipeline is more geared towards **task parallelism** rather than **data parallelism**?

Our paper addresses these Issues!

hpg 2010

In Other Words…

- Imagine that these pipeline stages were actually bricks.
- Then we are providing the mortar between the bricks.



Related Work

- Alternative pipelines on the GPU:
  - Renderants [Zhou et al. 2009]
  - Freepipe [Liu et al. 2009]
  - Optix [NVIDIA 2010]
- Distributed Queuing on the GPU:
  - GPU Dynamic Load Balancing [Cederman et al. 2008]
  - Multi-CPU work
- Reyes on the GPU:
  - Subdivision [Patney et al. 2008]
  - Diagsplit [Fisher et al. 2009]
  - Micropolygon Rasterization [Fatahalian et al. 2009]

Ingredients for Mortar

Questions that we need to address:

What is the proper granularity for tasks?

How many threads to launch?

How to avoid global synchronizations?

How to distribute tasks evenly?

Warp Size Work Granularity

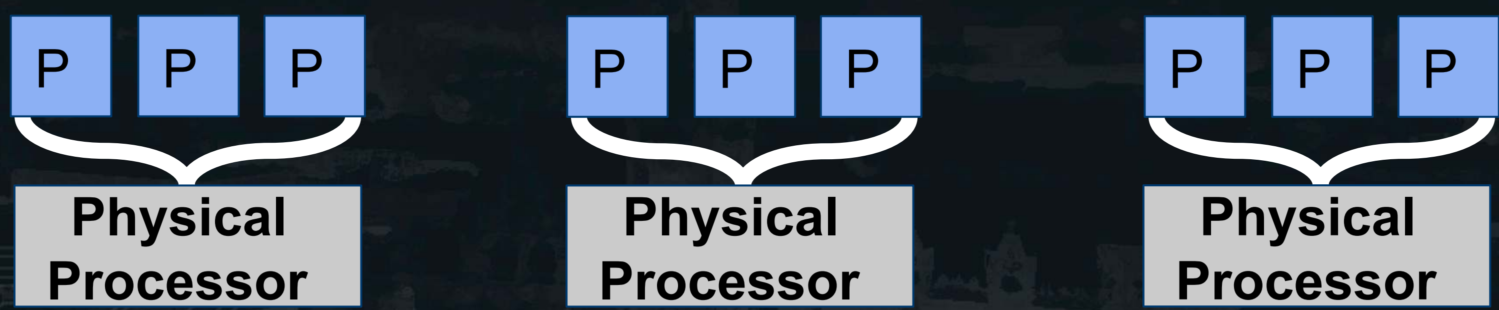
Persistent Threads

Uberkernels

Task Donation

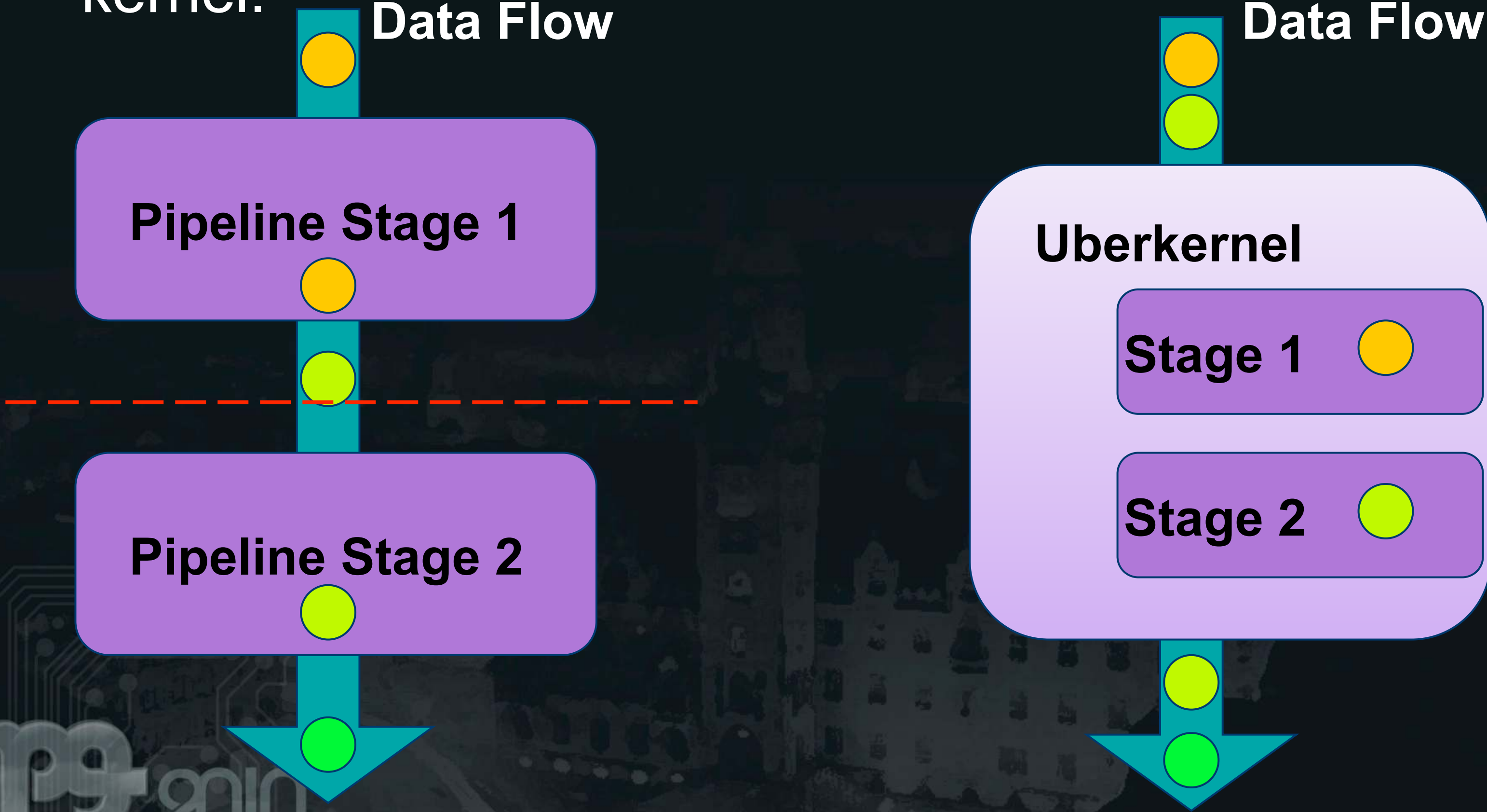
Warp Size Work Granularity

- Problem: We want to emulate task level parallelism on the GPU without loss in efficiency.
- Solution: we choose block sizes of 32 threads / block.
  - Removes messy synchronization barriers.
  - Can view each block as a MIMD thread. We call these blocks *processors*



Uberkernel Processor Utilization

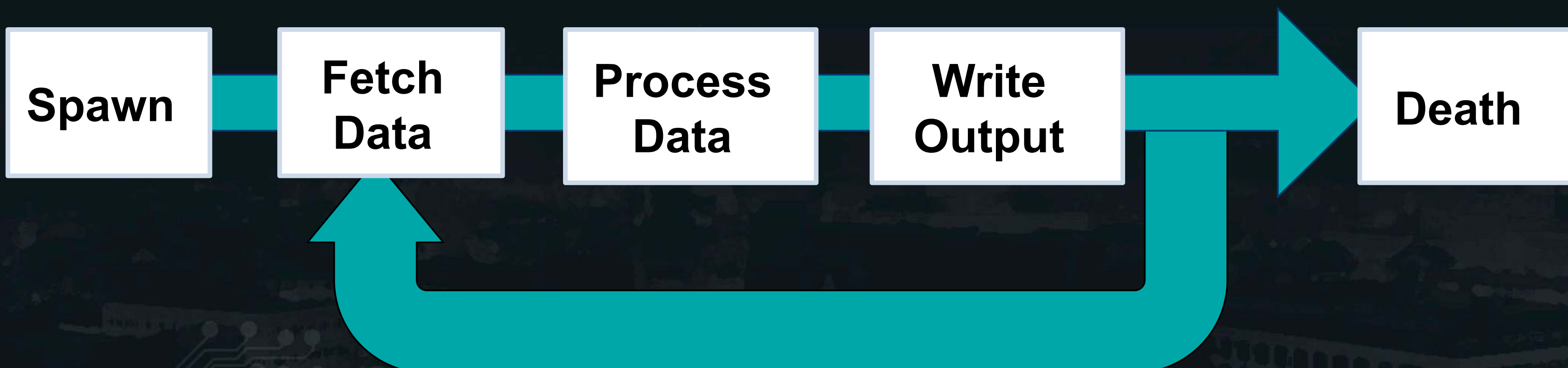
- Problem: Want to eliminate global kernel barriers for better processor utilization
- Uberkernels pack multiple execution routes into one kernel.



Persistent Thread Scheduler Emulation

- Problem: If input is irregular? How many threads do we launch?
- Launch enough to fill the GPU, and keep them alive so they keep fetching work.

Life of a Thread:
Life of a Persistent Thread:



How do we know when to stop?

When there is no more work left

Memory Management System

- Problem: We need to ensure that our processors are constantly working and not idle.
- Solution: Design a software memory management system.
- How each processor fetches work is based on our queuing strategy.
- We look at 4 strategies:
  - Block Queues
  - Distributed Queues
  - Task Stealing
  - Task Donation

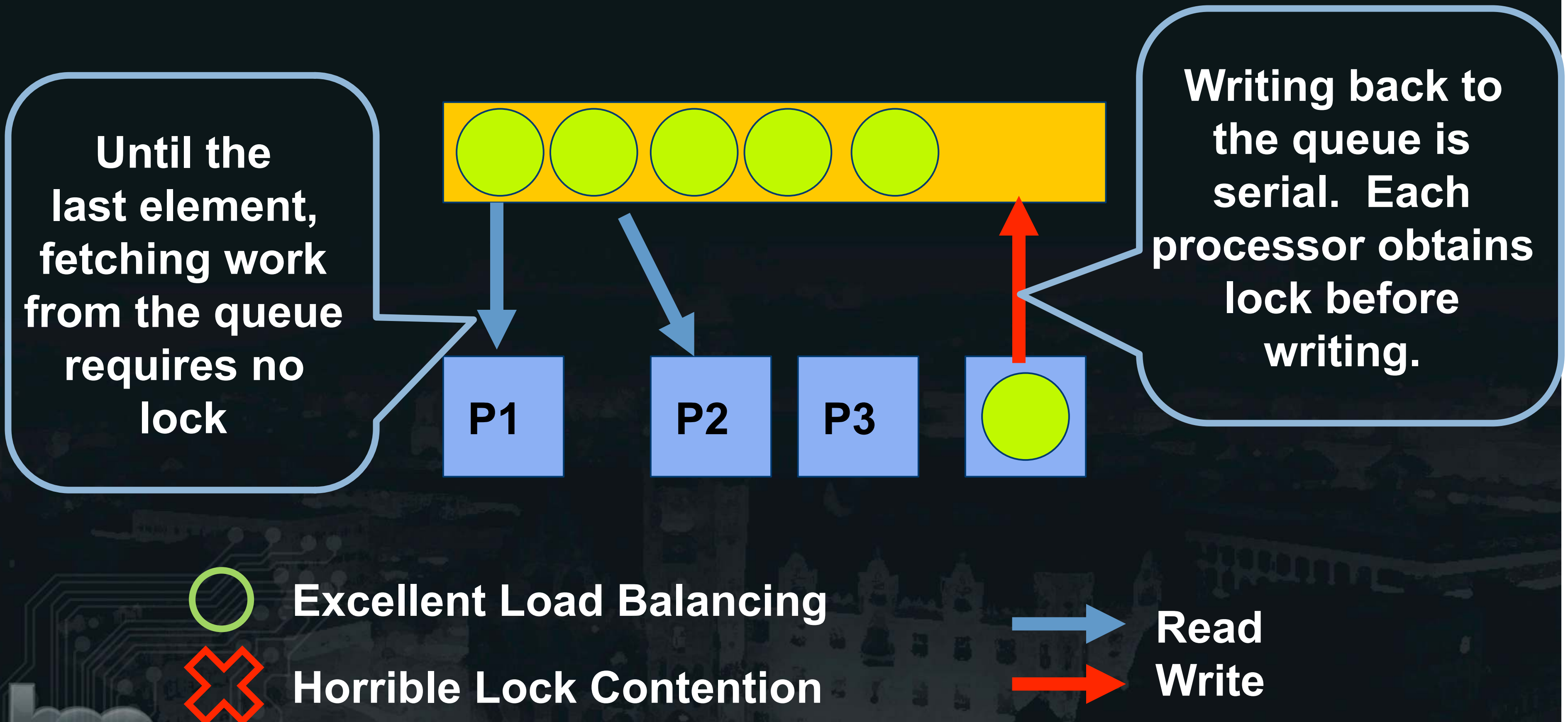
A Word About Locks

- To obtain exclusive access to a queue each queue has a lock.
- Current implementation uses spin locks and are very slow on GPUs.
- We want to use as few locks as possible.

```
while (atomicCAS(lock, 0,1) ==1);
```

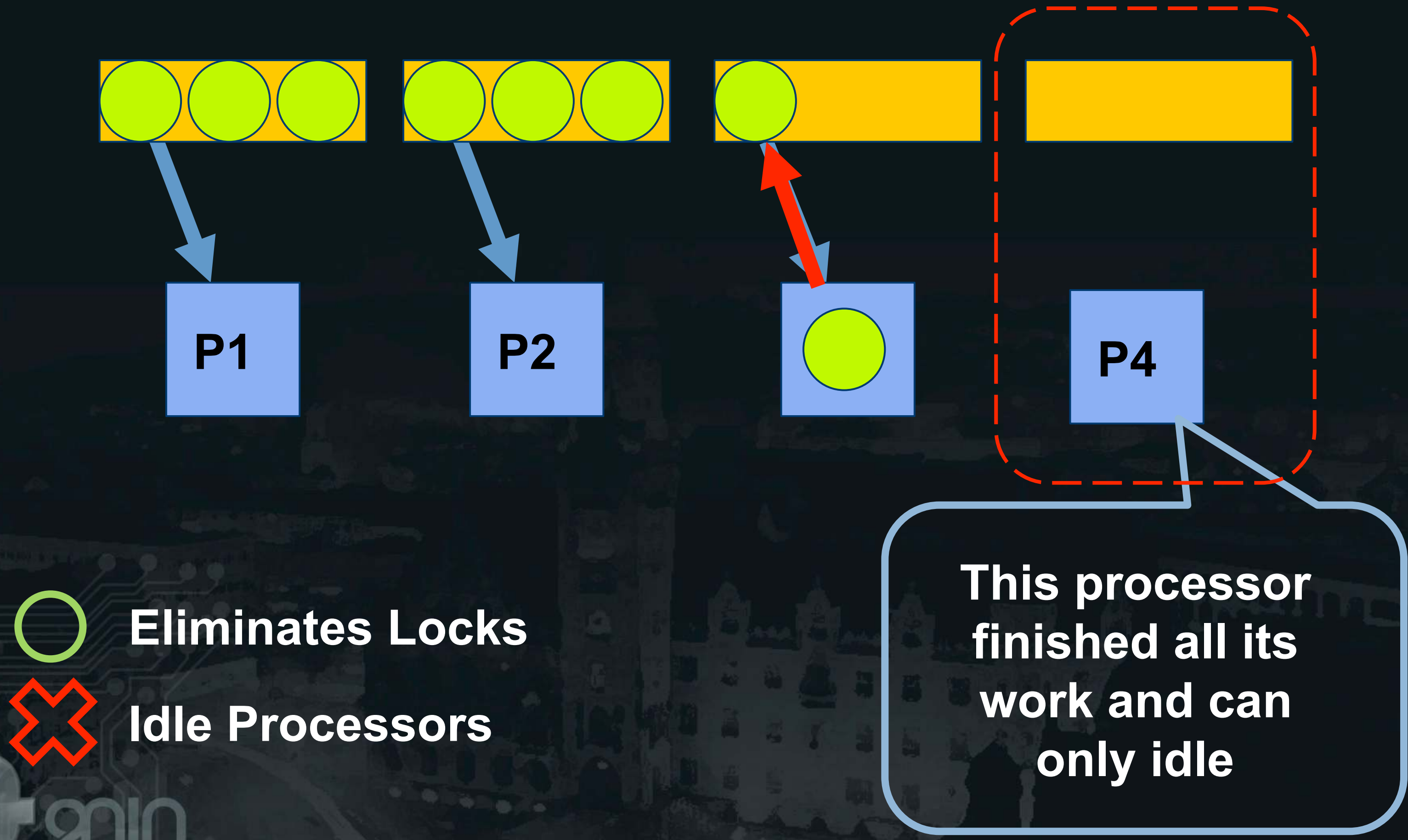
Block Queuing

- 1 dequeue for all processors. Read from one end write back to the other.



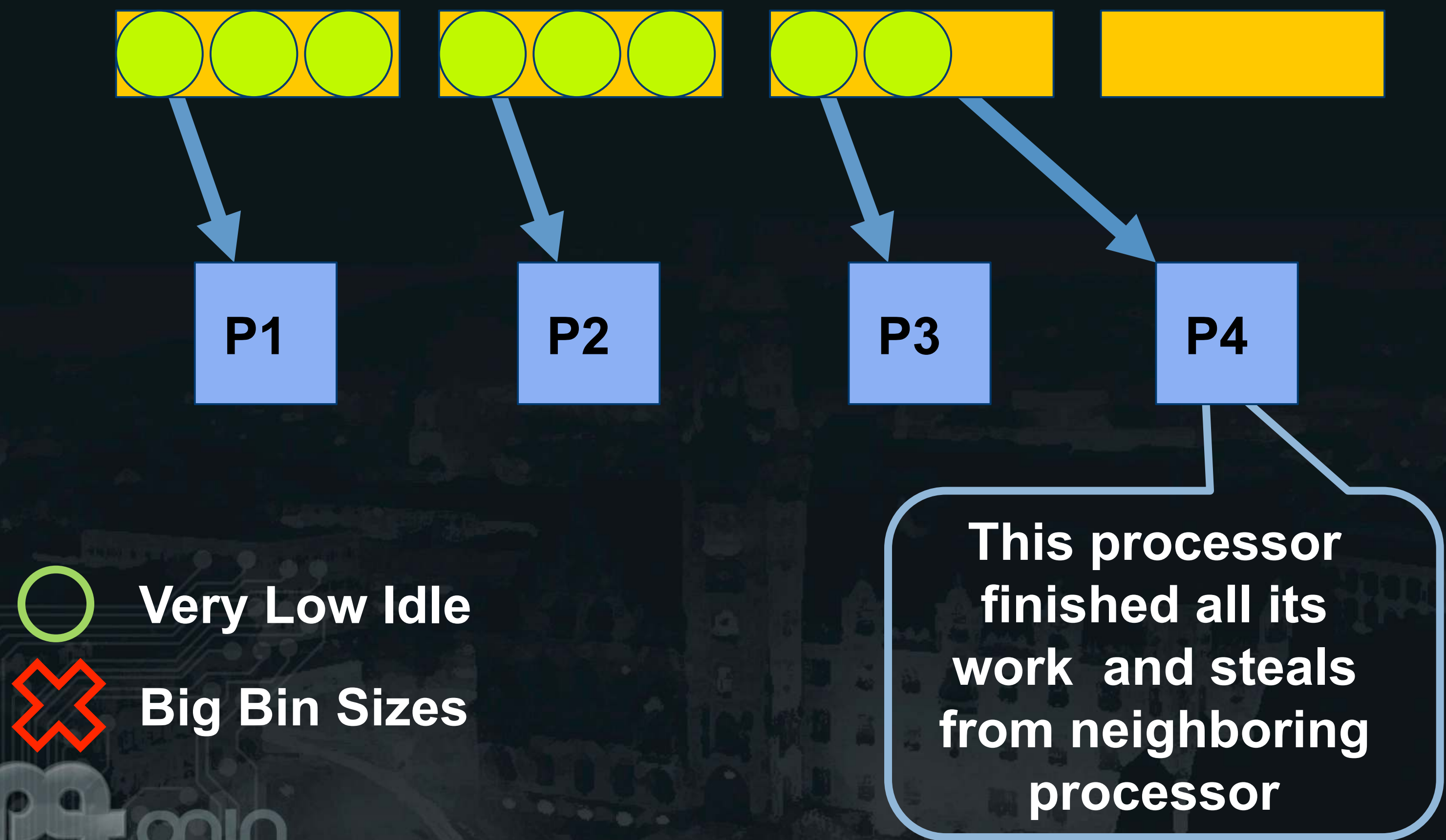
Distributed Queuing

- Each processor has its own dequeue (called a bin) and it reads and writes to it.



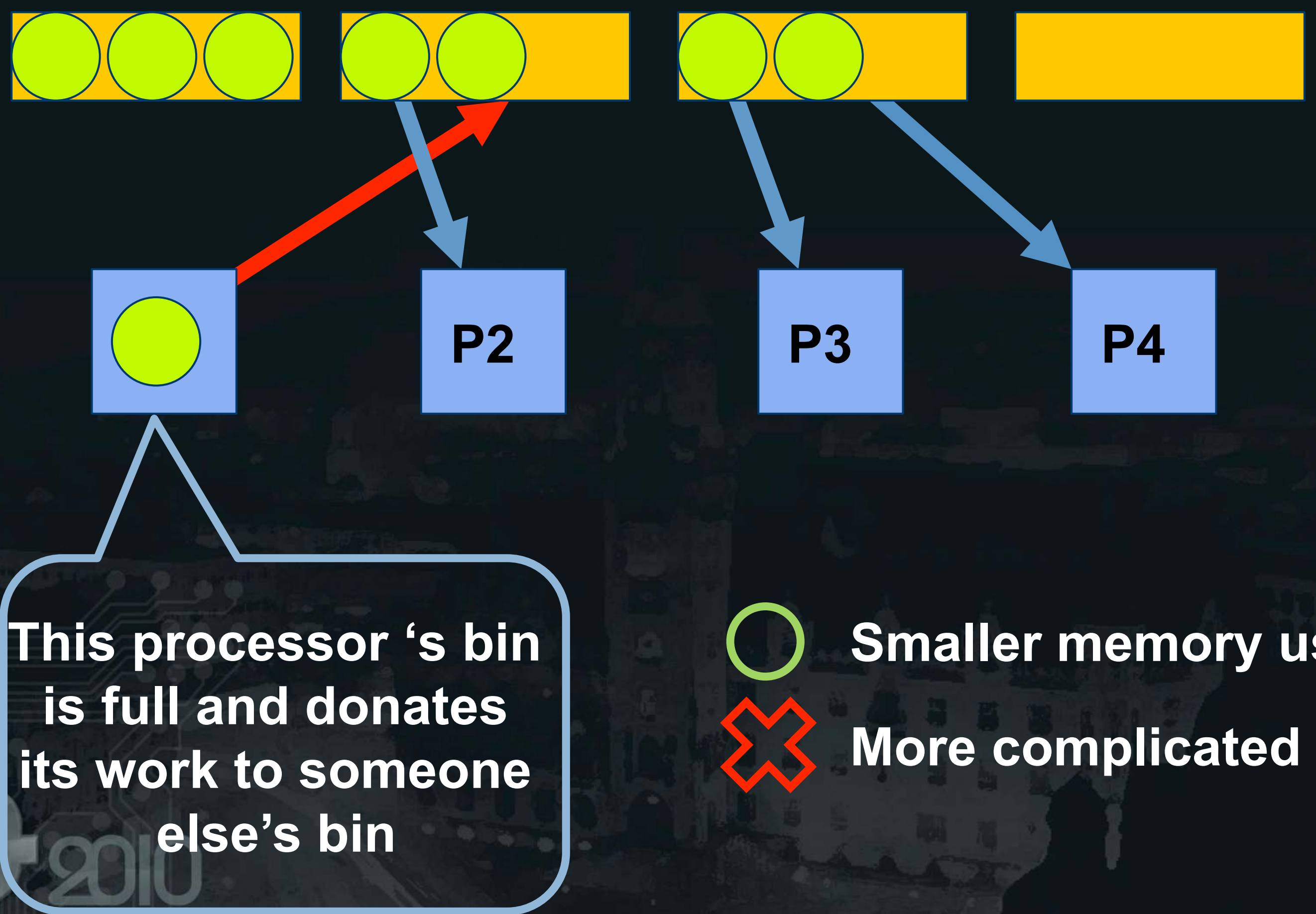
Task Stealing

- Using the distributed queuing scheme, but now processors can steal work from another bin.



Task Donation

- When a bin is full, processor can give work to someone else.



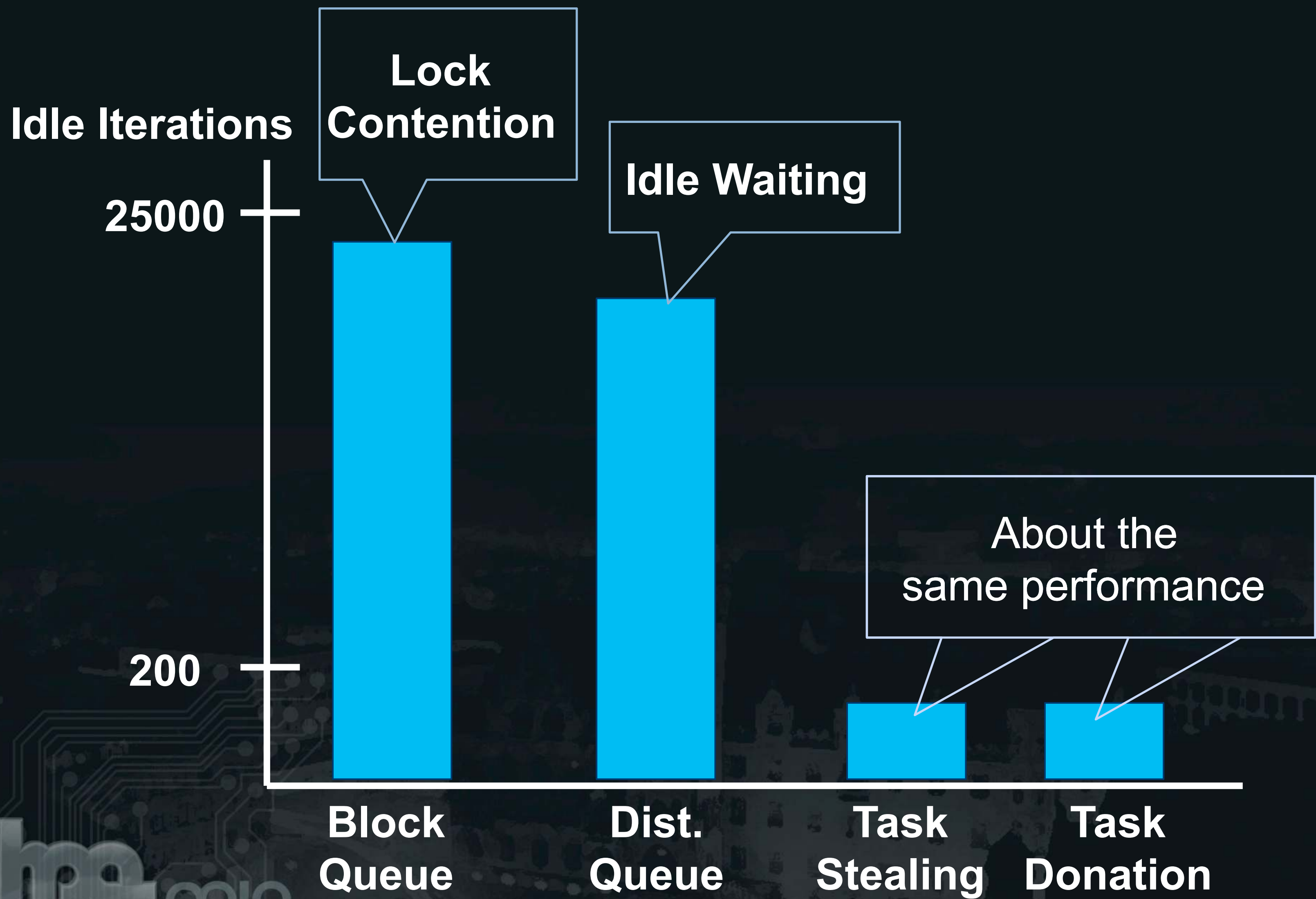
Smaller memory usage

More complicated

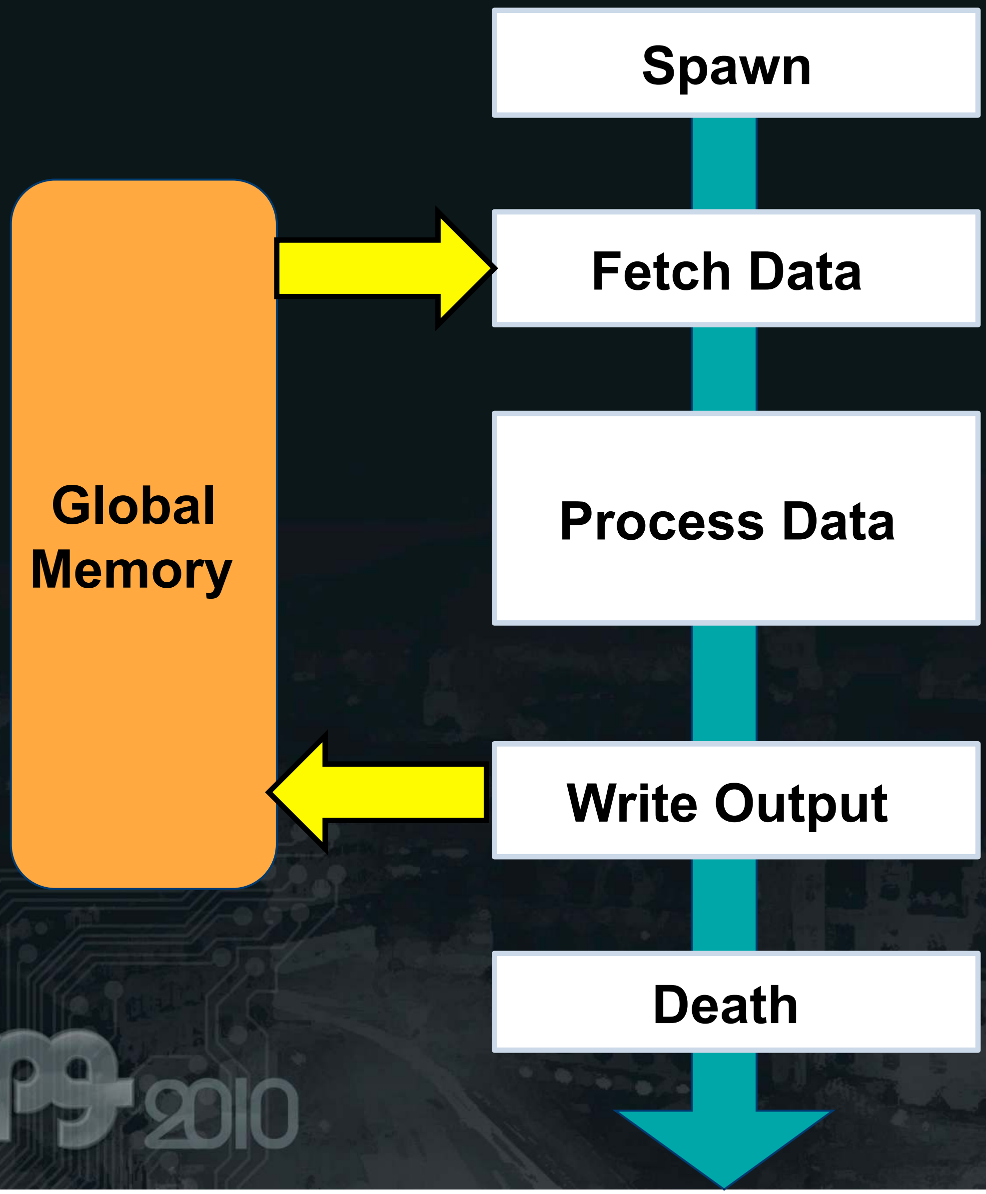
Evaluating the Queues

- Main measure to compare:
  - How many iterations the processor is idle due to lock contention or waiting for other processors to finish.
- We use a synthetic work generator to precisely control the conditions.

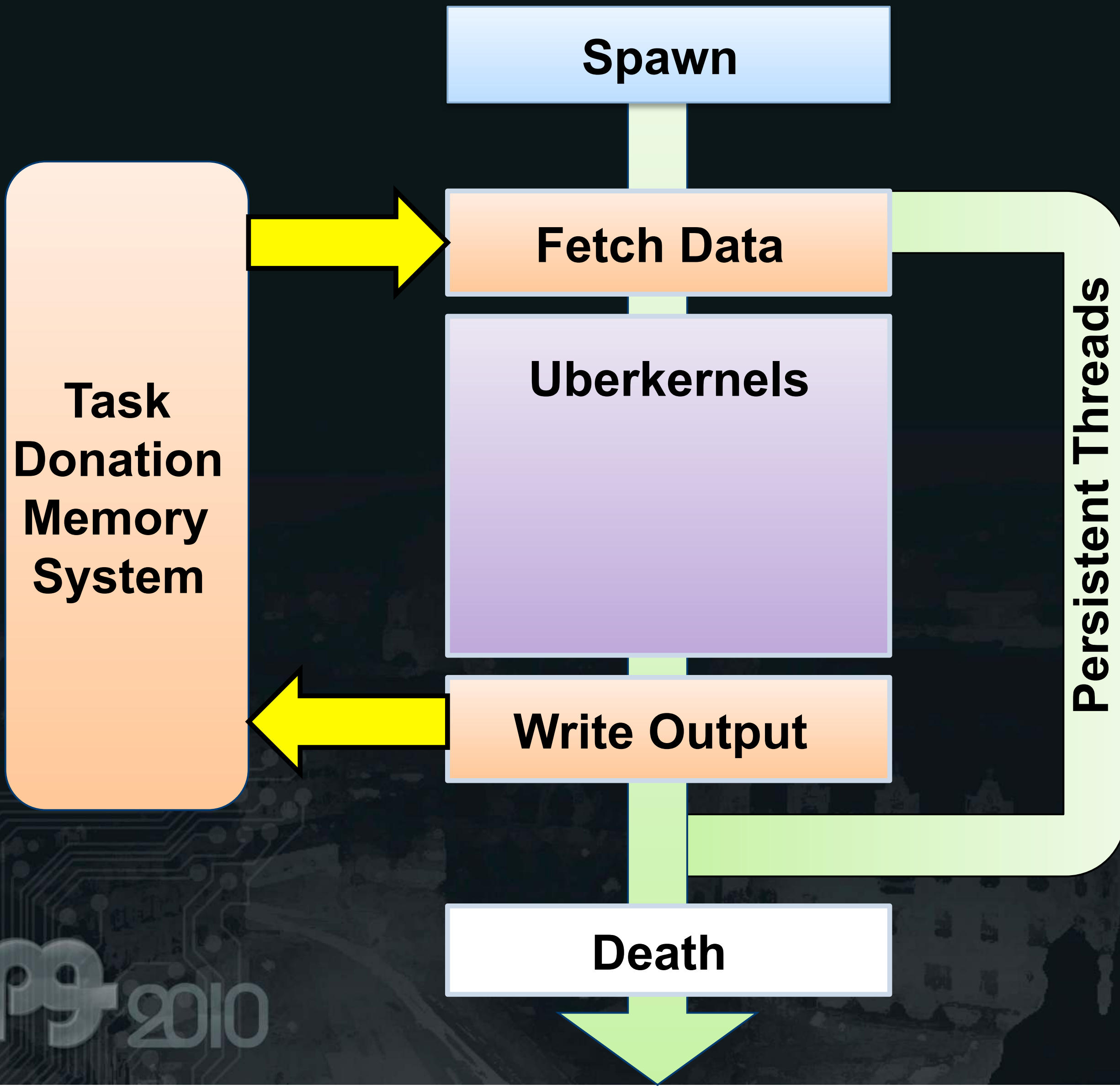
Average Idle Iterations Per Processor



How it All Fits Together

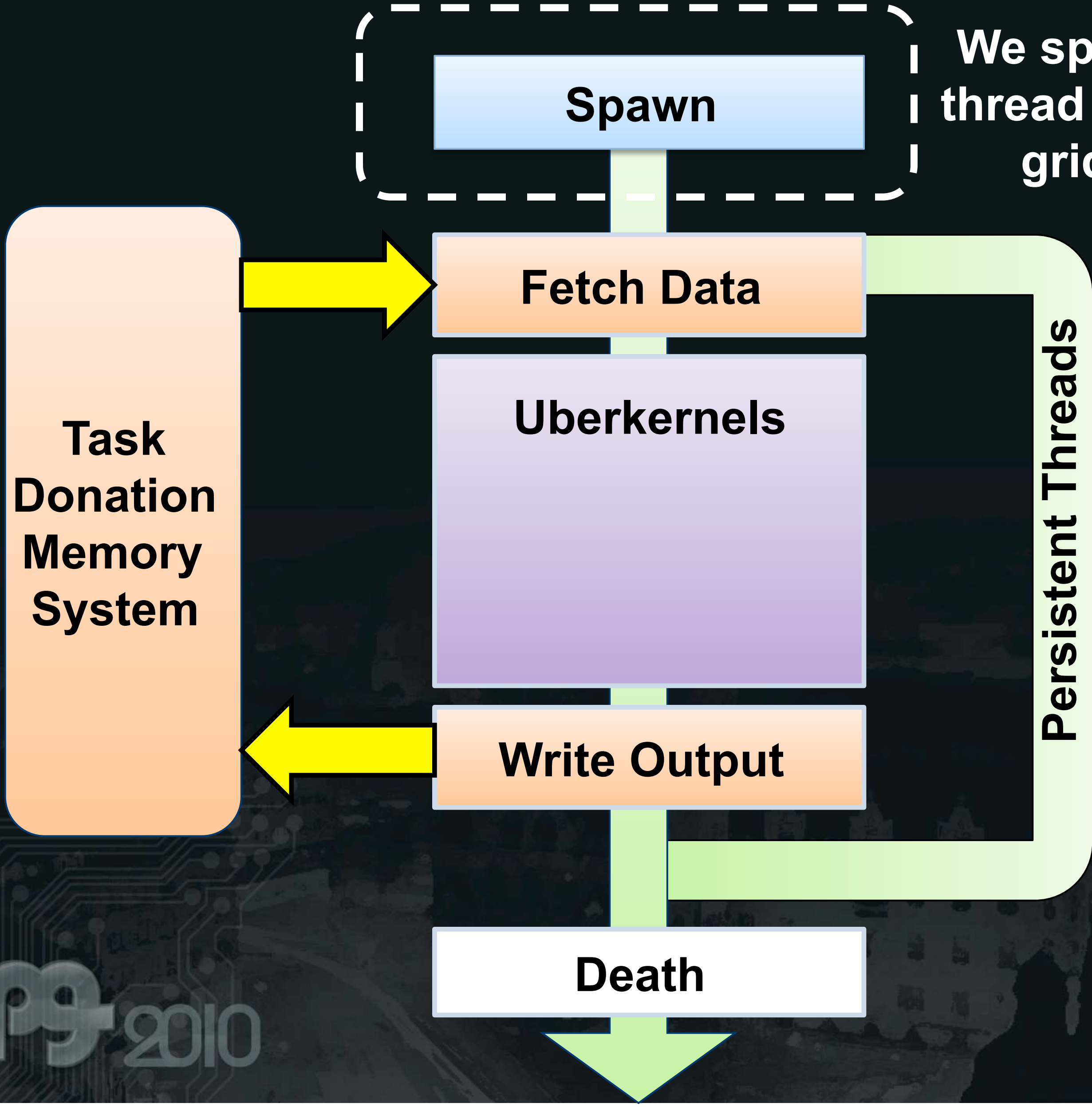


Our Version



Spawn
Task
Donation
Memory
System
Fetch Data
Uberkernels
Write Output
Persistent Threads
Death

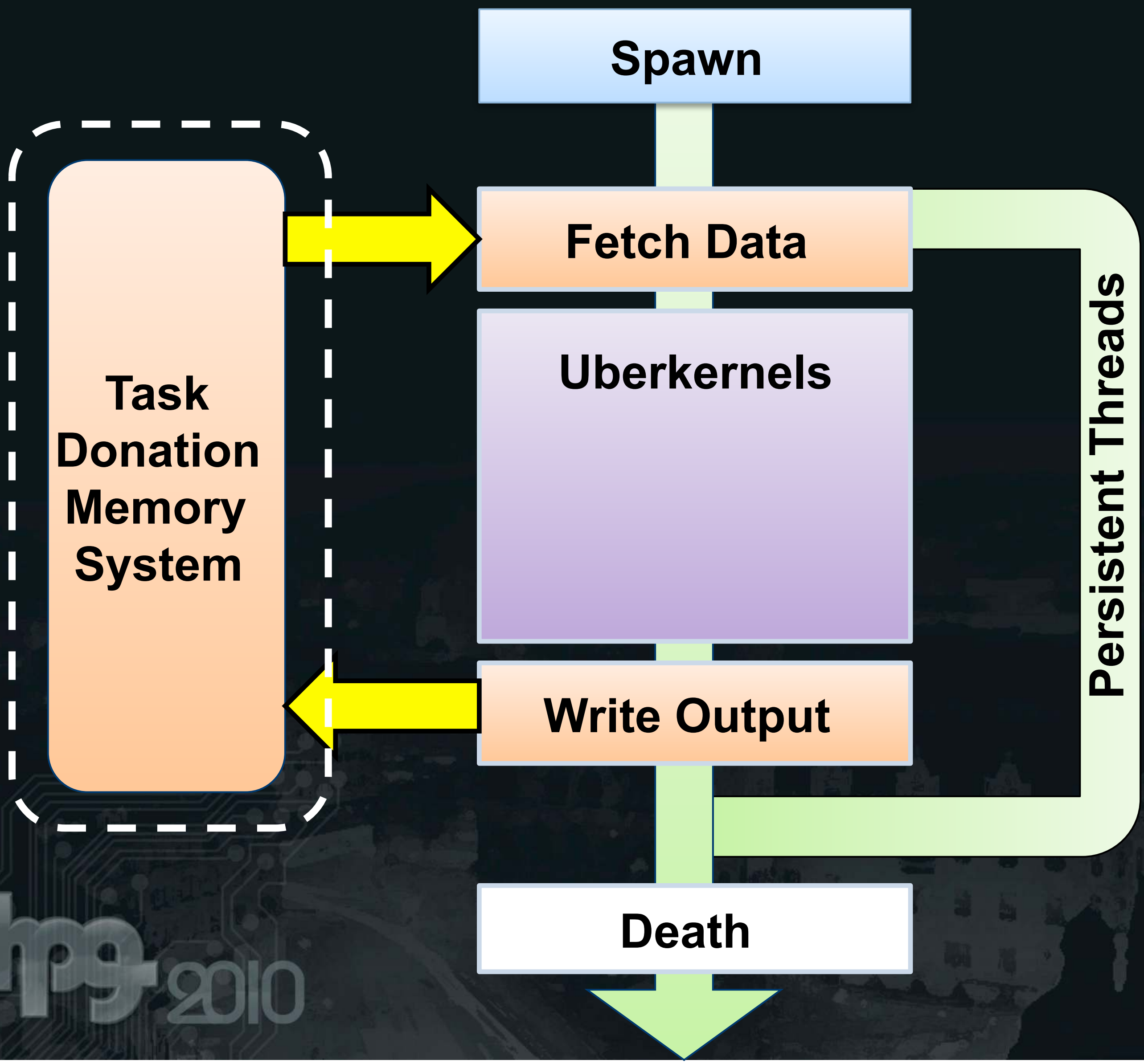
Our Version



We spawn 32 threads per thread group / block in our grid. These are our processors.

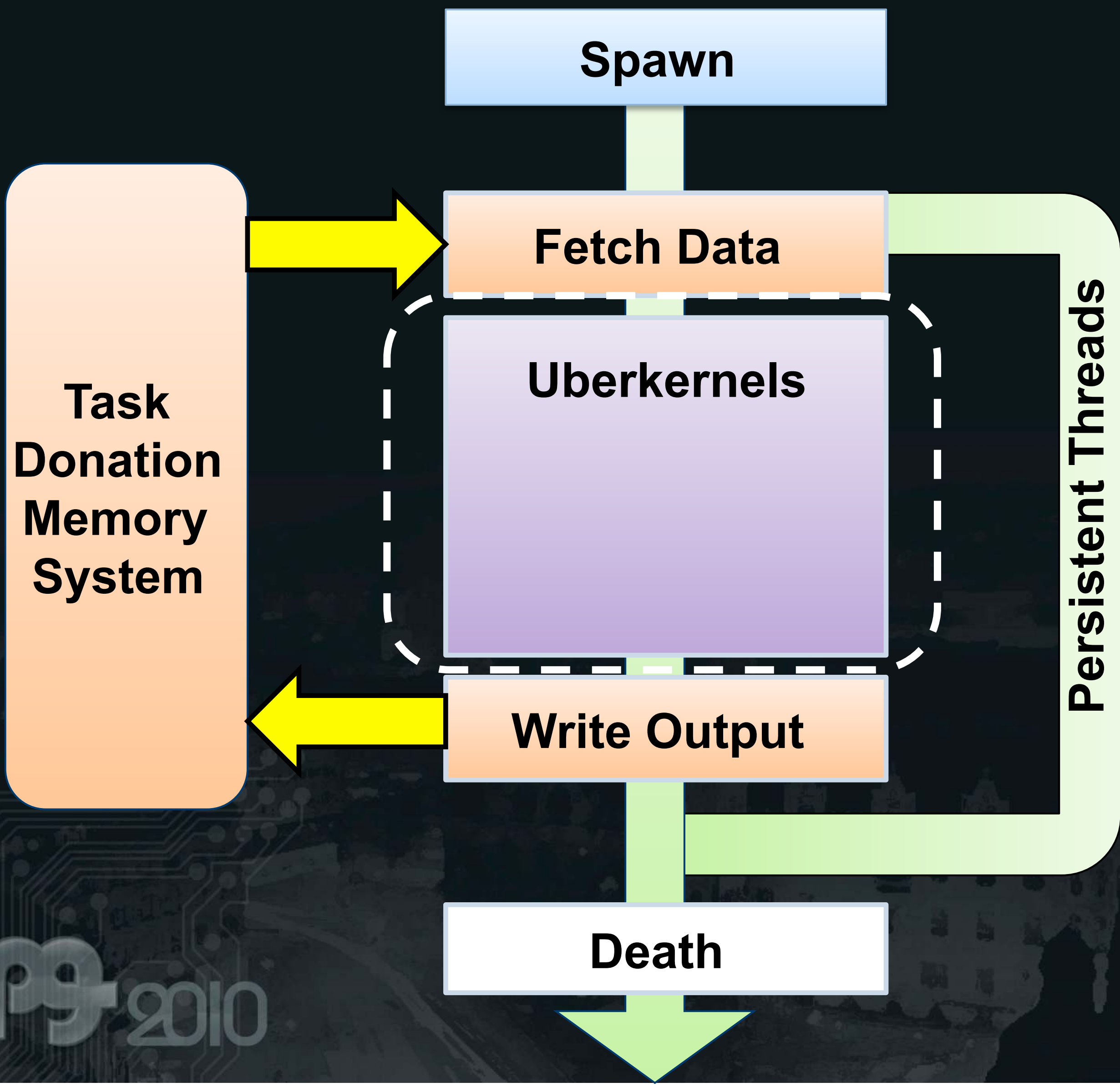
Our Version

Each processor grabs work to process.

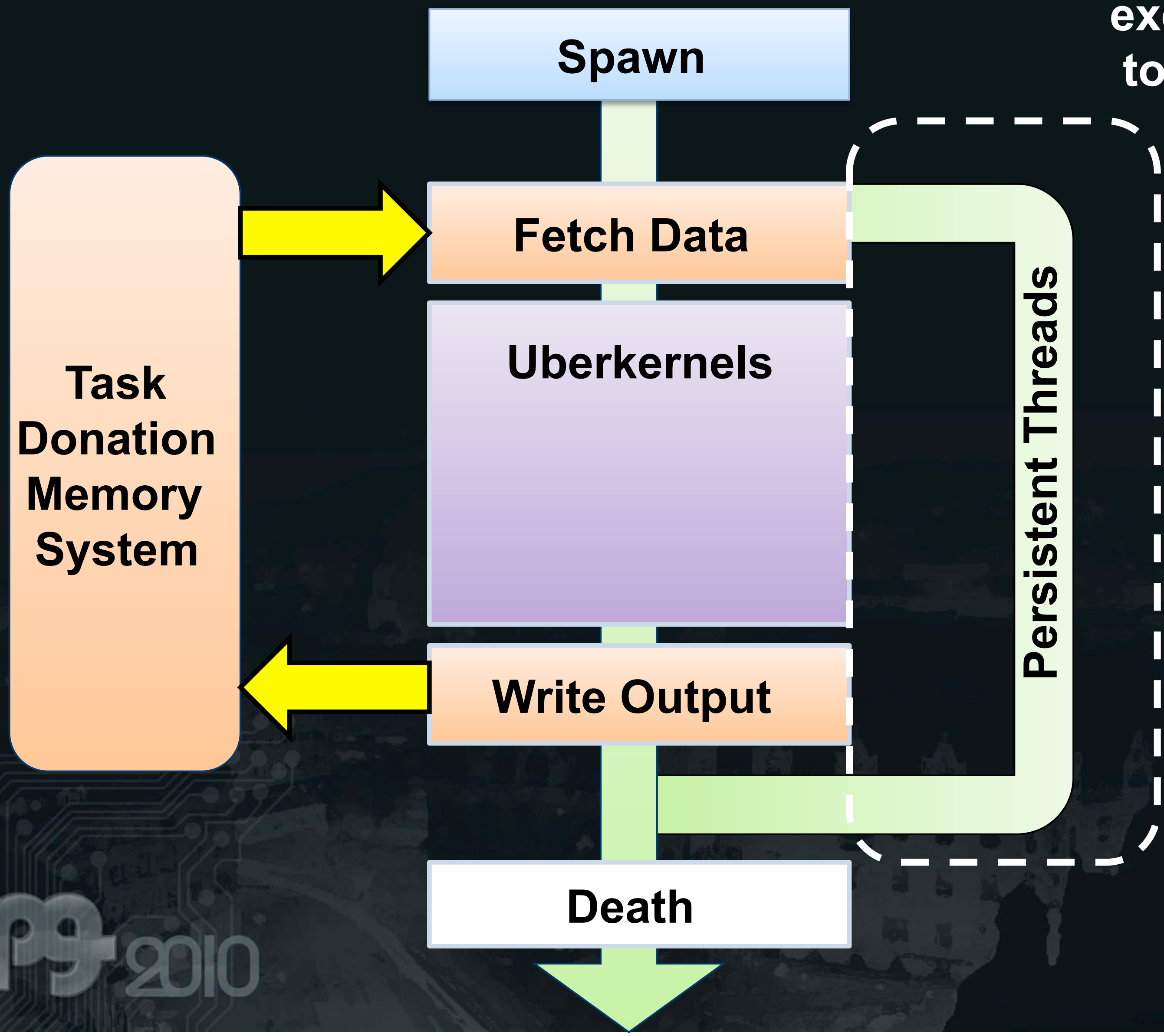


Our Version

Uberkernel decies how to process the current work unit



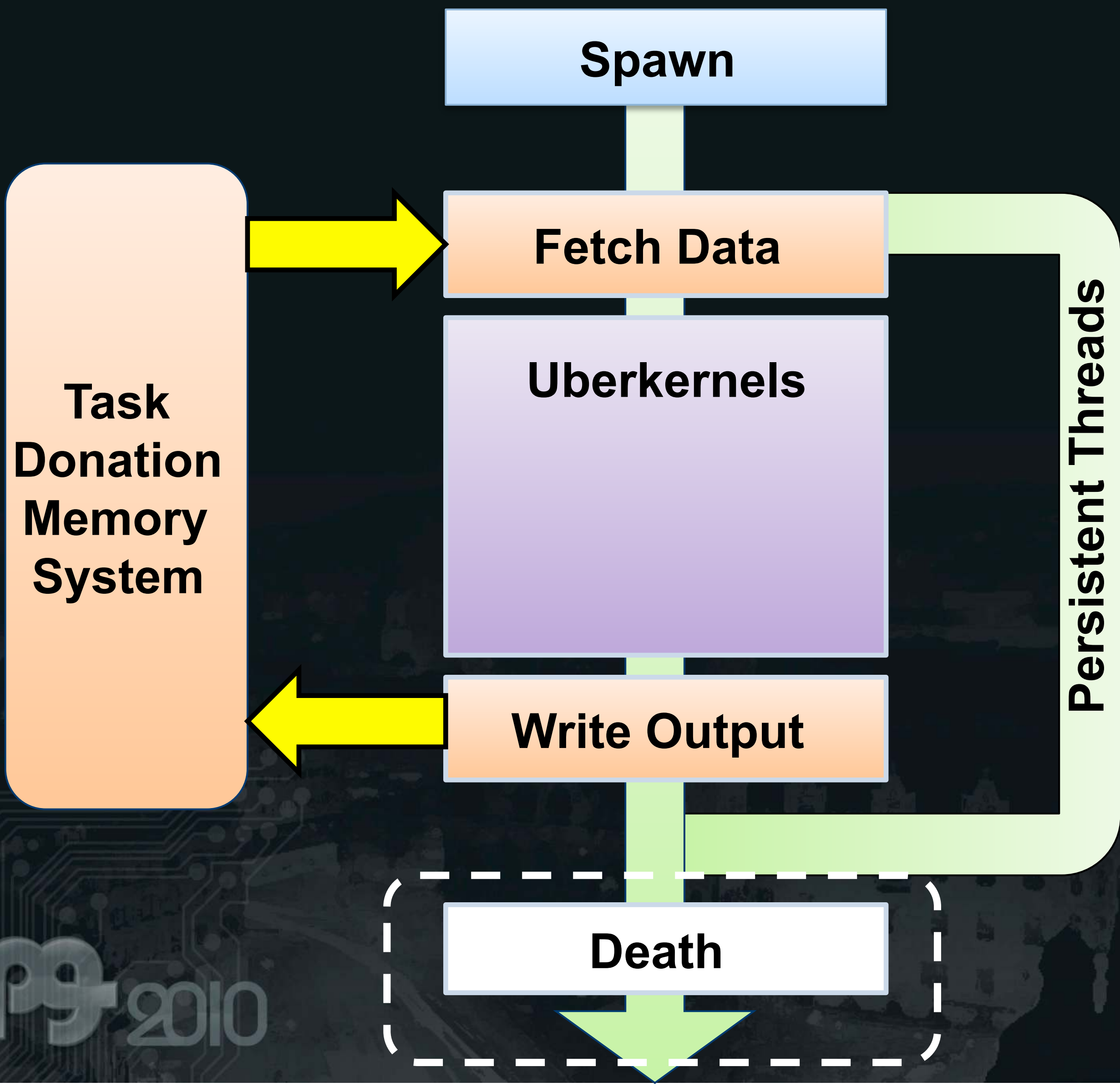
Our Version



Once work is processed, thread execution returns to fetching more work

Our Version

When there is no work left in the queue, the threads retire

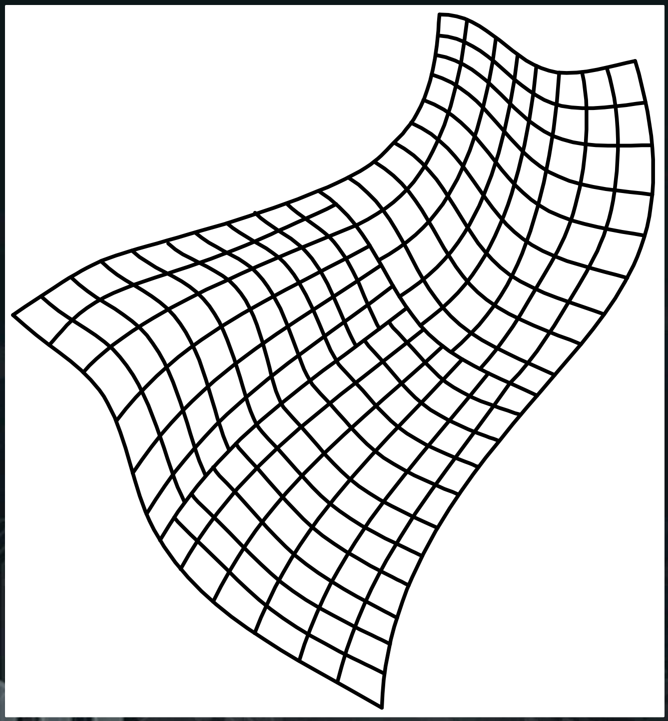


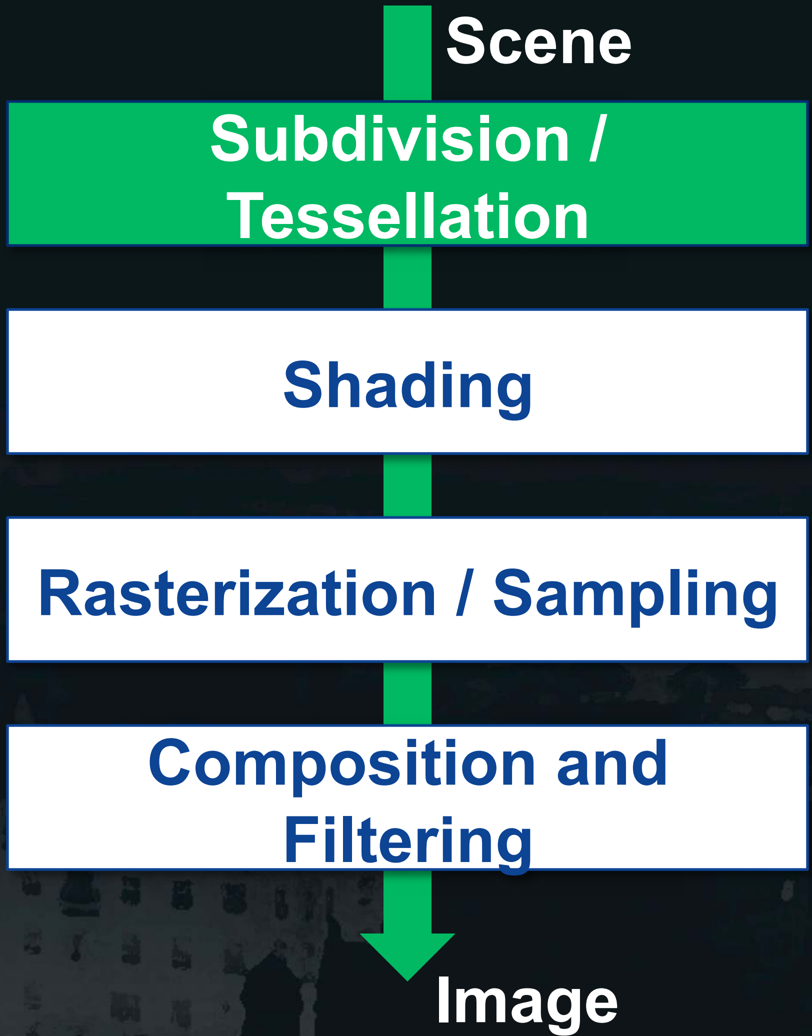
APPLICATION: REYES

Pipeline Overview

Start with smooth surfaces

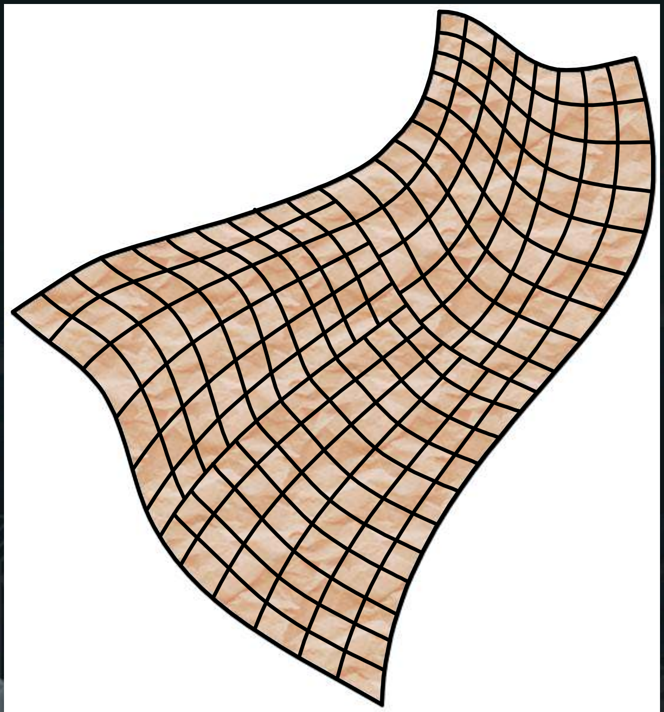
Obtain micropolygons

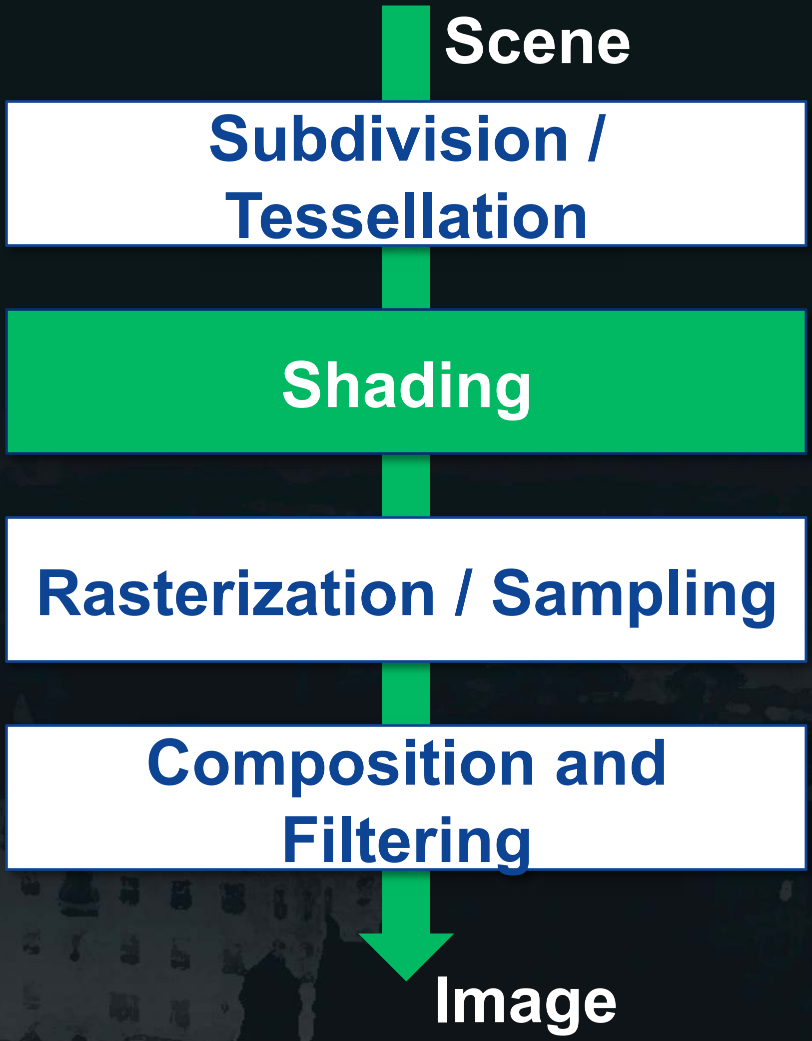




Pipeline Overview

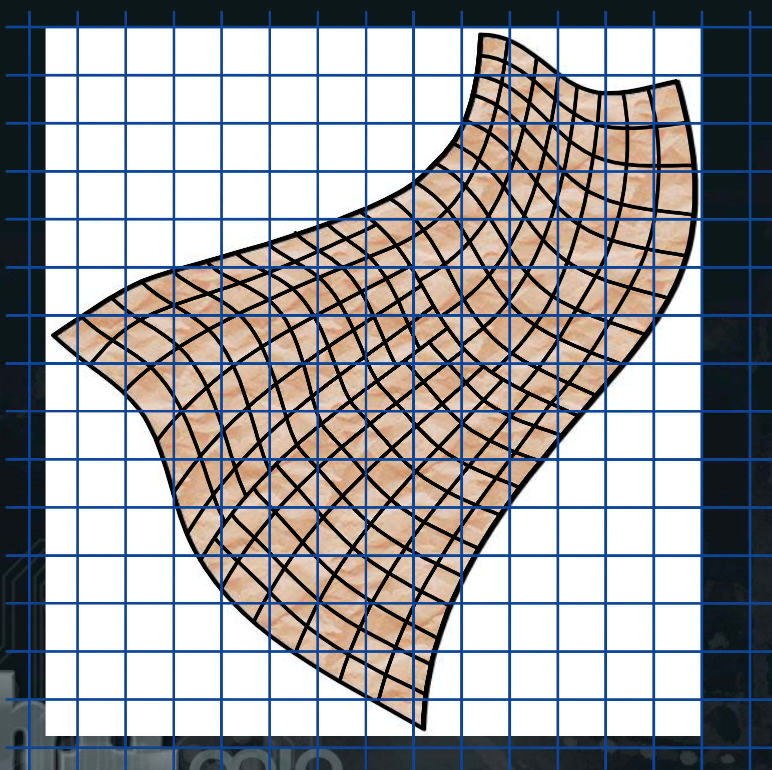
Shade micropolygons

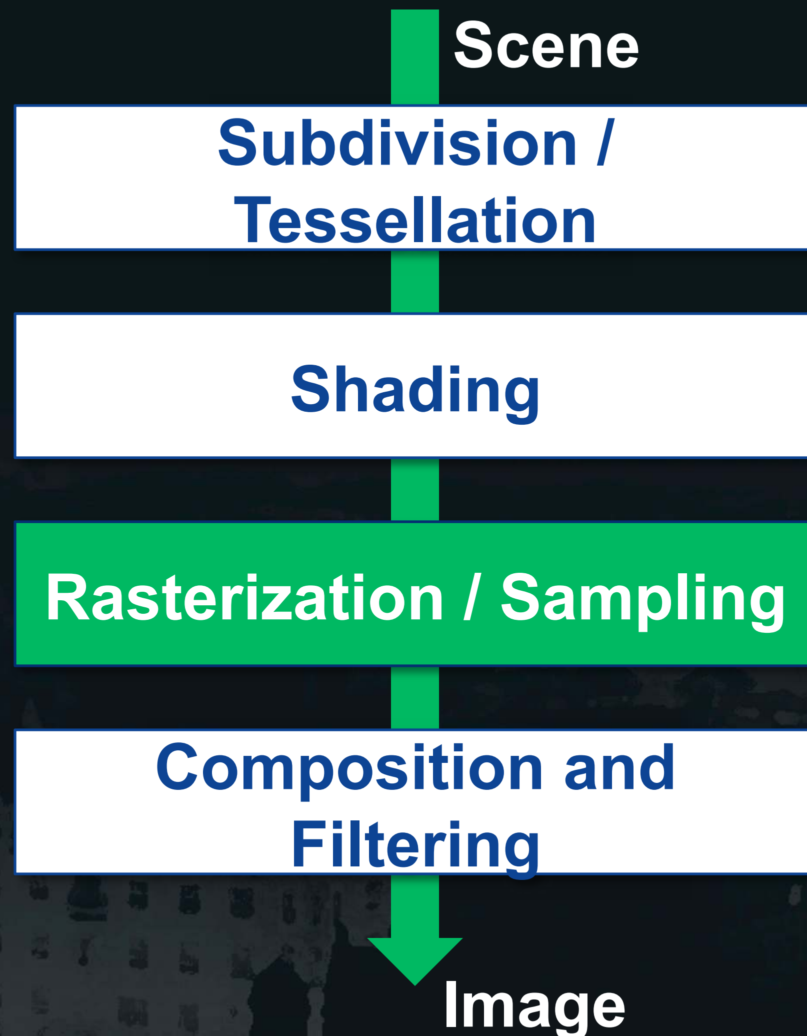




Pipeline Overview

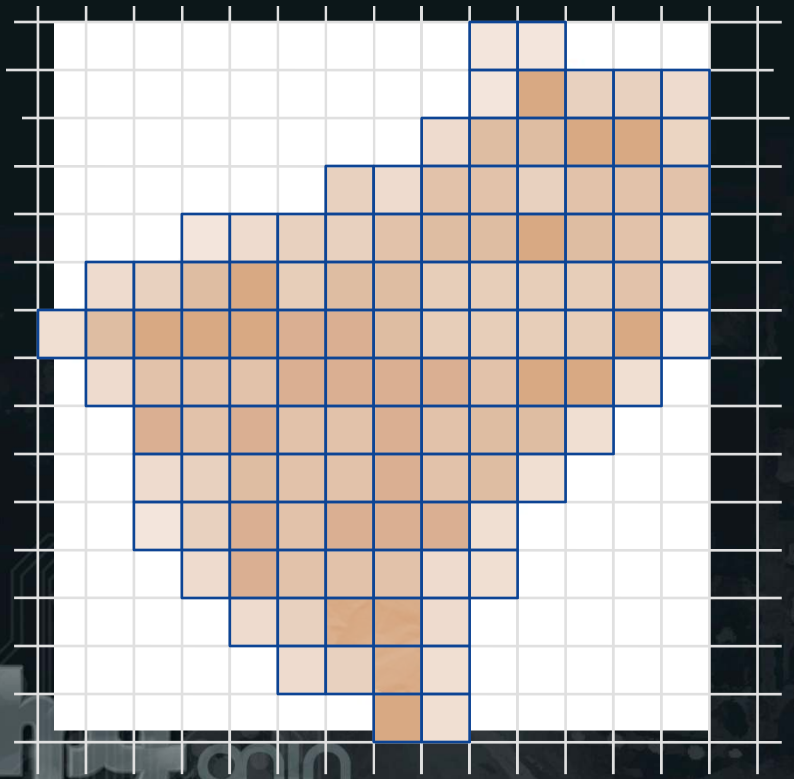
Map micropolygons to screen space

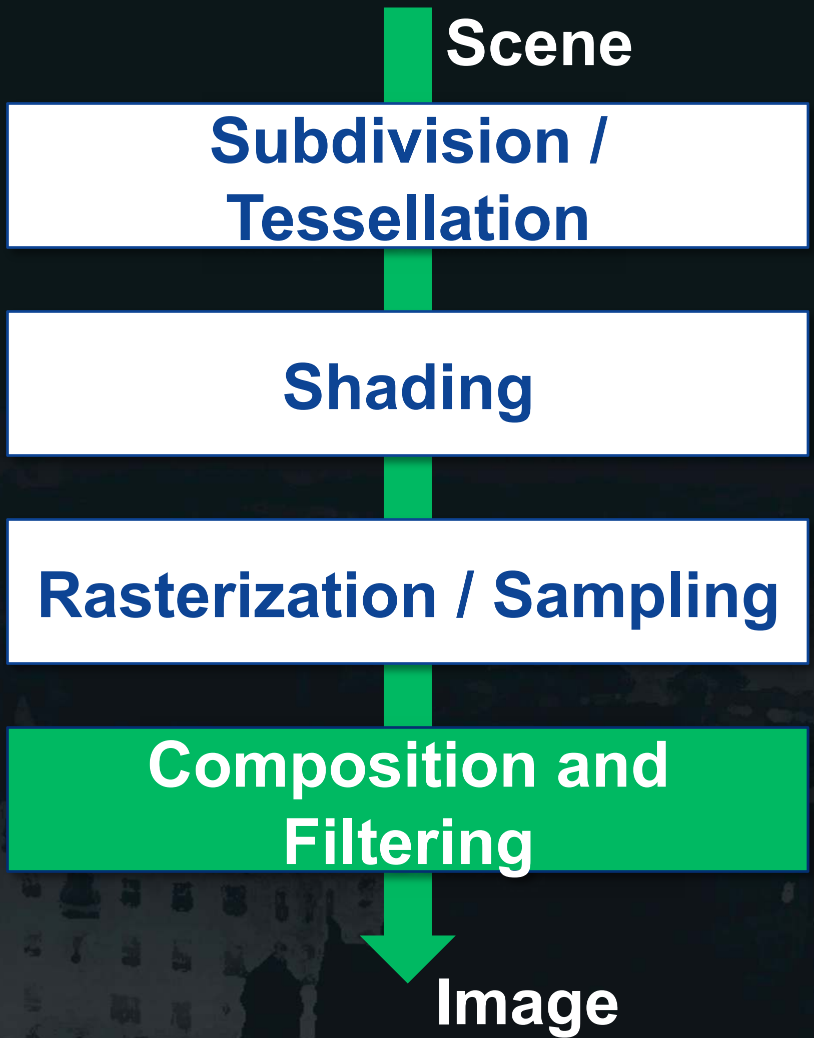




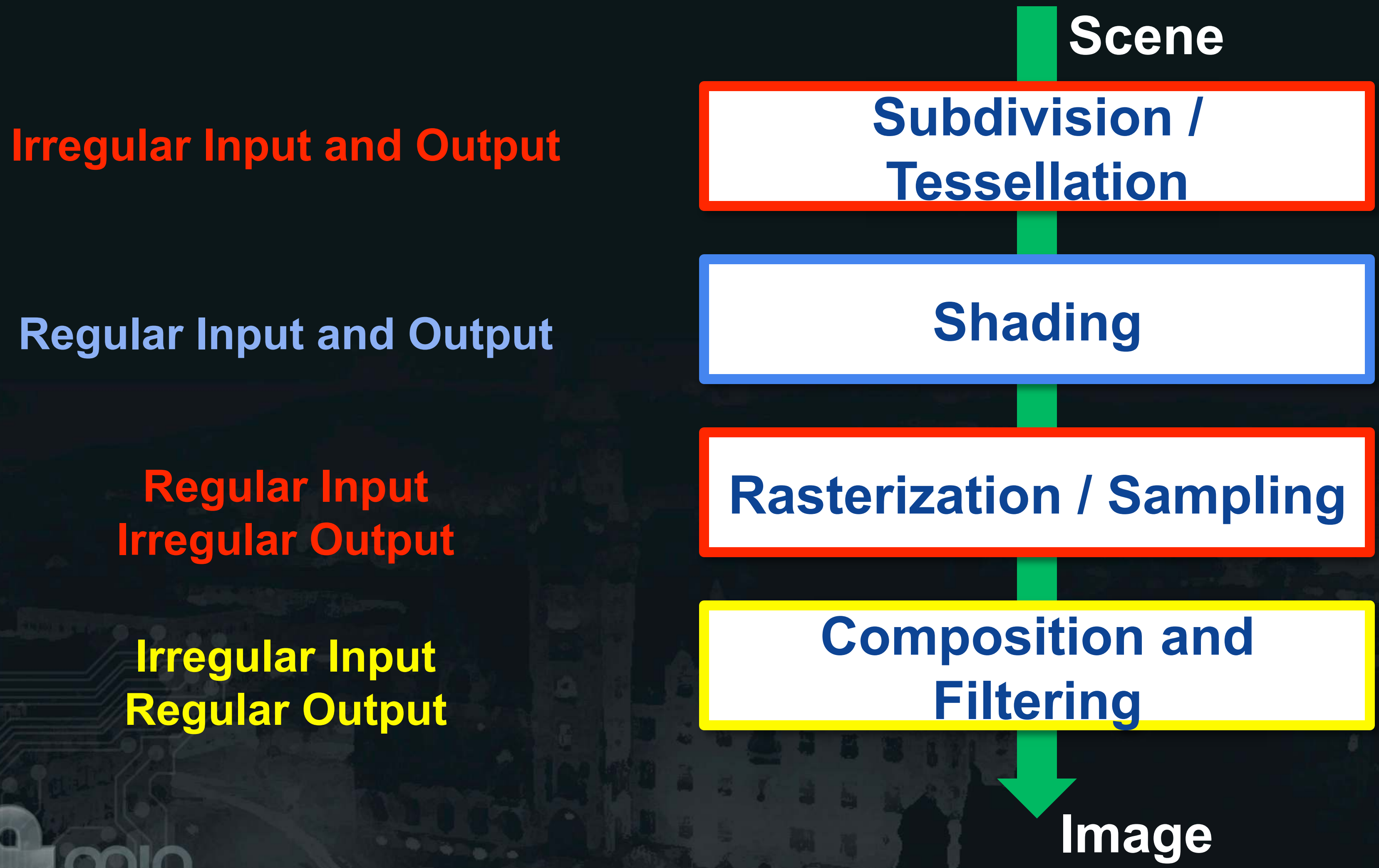
Pipeline Overview

Reconstruct pixels from obtained samples





Pipeline Overview

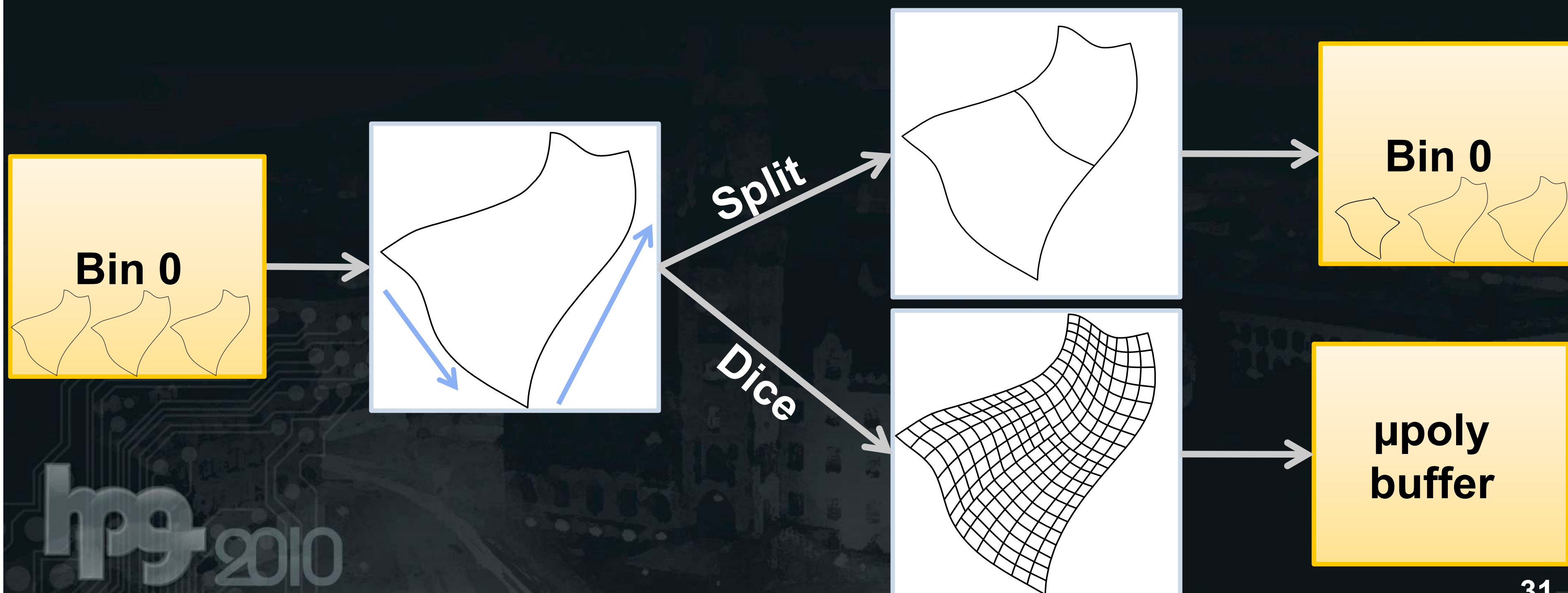


Split and Dice

- We combine the patch split and dice stage into one kernel.
- Bins are loaded with initial patches.
- 1 processor works on 1 patch at a time. Processor can write back split patches into bins.
- Output is a buffer of micropolygons

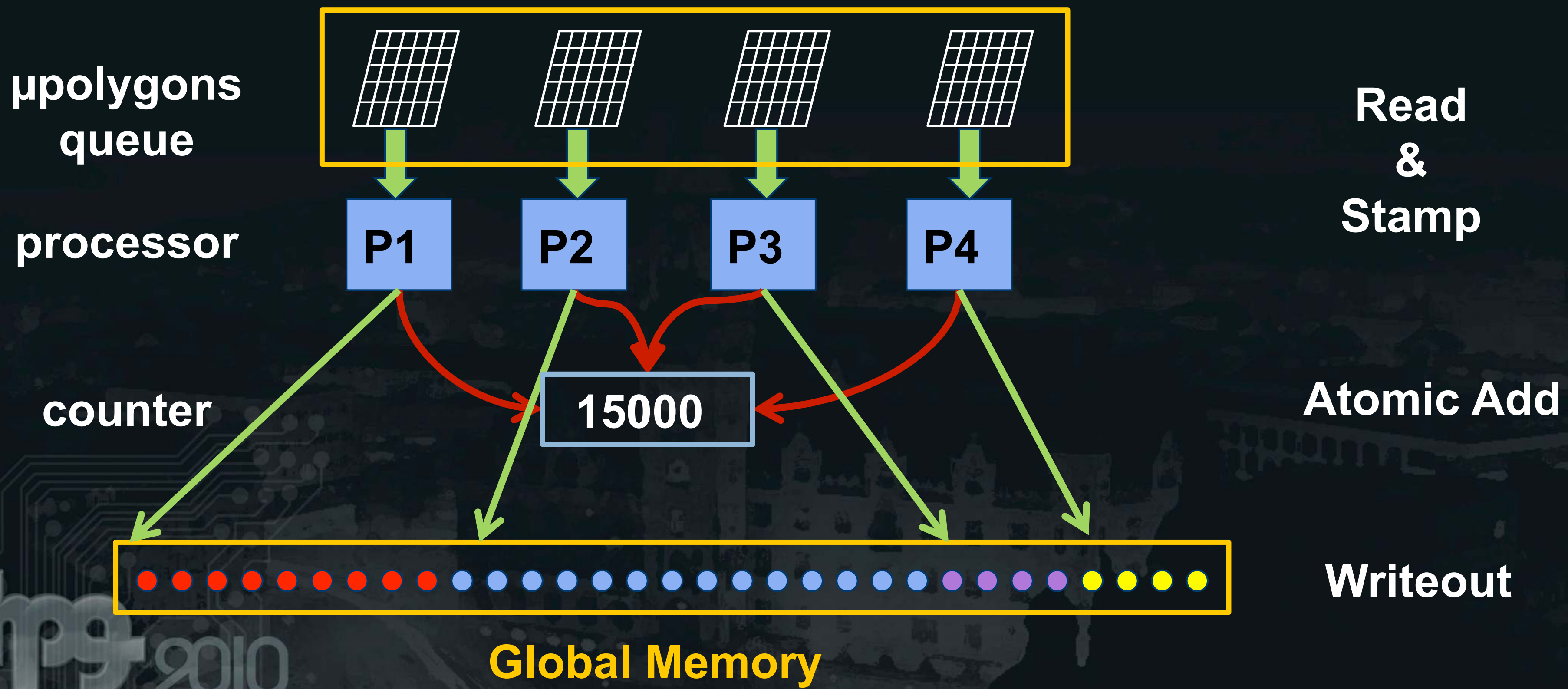
Split and Dice

- 32 Threads on 16 CPs – 16 threads each work in u and v
- Calculate u and v thresholds, and then go to uberkernel branch decision:
  - Branch 1 splits the patch again
  - Branch 2 dices the patch into micropolygons



Sampling

- Stamp out samples for each micropolygon. 1 processor per micropolygon patch.
- Since only output is irregular, use a block queue.
- Write out to a sample buffer.



Sampling

- Stam… 1 proce…
- Since … queue.
- Write…

Q:
Why are you using a block queue? Didn't you just show block queues have high contention when processors write back into it?

μpolygons queue

processo…

counter

A:
True. However, *we're not writing into this queue!* We're just reading from it, so there is no writeback idle time.

Read & Stamp

Atomic Add

Writeout

Smooth Surfaces, High Detail

16 samples per pixel
>15 frames per second on GeForce GTX280

What's Next

- What other (and better) abstractions are there for programmable pipelines?
- How is future GPU design going to affect software schedulers?
- For Reyes: What is the right model to do GPU real time micropolygon shading?

Acknowledgments

- Matt Pharr, Aaron Lefohn, and Mike Houston
- Jonathan Ragan-Kelley
- Shubho Sengupta
- Bay Raitt and Headus Inc. for models
- National Science Foundation
- SciDAC
- NVIDIA Graduate Fellowship

Thank You