Lecture 6:

Persistent Threads and Task Queues on GPUs

Modern Parallel Computing

John Owens

EEC 289Q, UC Davis, Winter 2018

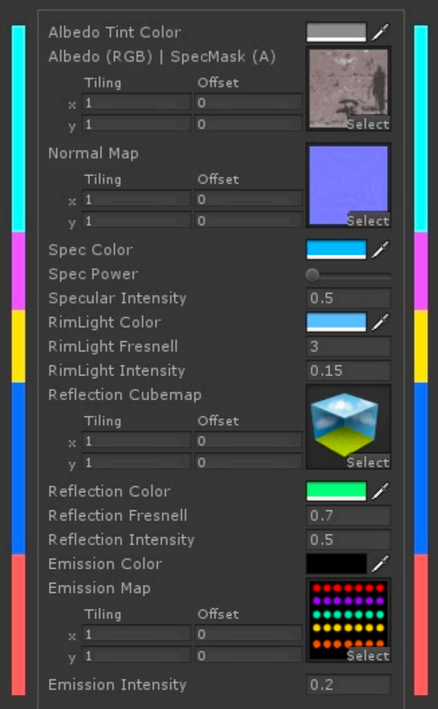
Uberkernels

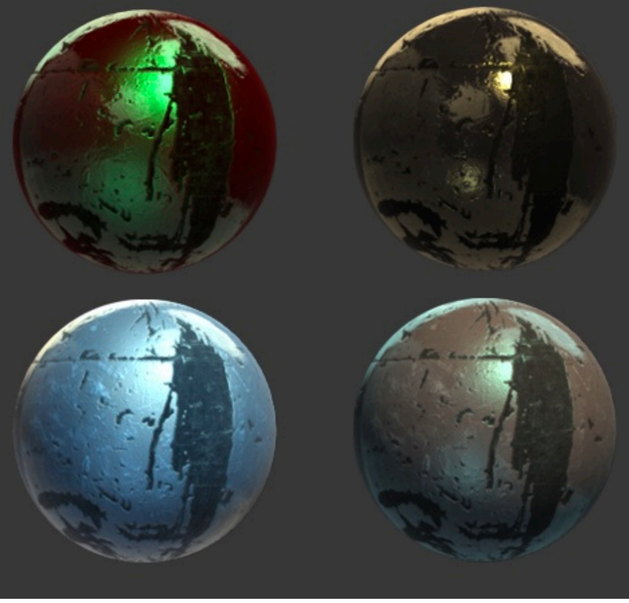
Ubershaders

Uber Shader

The Uber Shader is designed for objects that require complex surface properties like Normal Map, Specularity, Rimlight or Reflection.
It is best used for Metal, Paint, Plastic and other shiny surfaces in general.

There are 5 main surface properties that the Uber Shader can affect: Diffuse, Specular, Rimlight, Reflection and Emissiveness.





Diffuse

Albedo Map

RGB For Diffuse Color
Alpha for Specular/Reflection Intensity

Albedo Tint Color

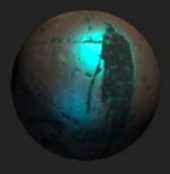
Tints the Diffuse Texture

Normal Map

Normal Map Texture that modulates the Diffuse, Specular and Reflection components

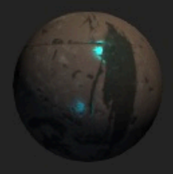
- The Alpha Channel in the Albedo Map is optional but highly recommended since it modulates both the Specular and the Reflection.
- The Albedo Tint Color multiplies with the Albedo Map but it should be used for small corrections rather then dramatic changes - usually to prevent an object from over exposing or slight color tints.
- The Normal Map requires Tangent Space normals.

Specular



Specular Color

Combines with the Light Color to tint the Specular



Specular Power

The size of the Specular

Specular Intensity

The intensity of the Specular

- The Specular Intensity and the Specular Power are modulated by the Alpha Channel in the Albedo Texture using grayscale values: brighter values making the specular smaller and more intense, while darker values increase the size of the specular and decrease the intensity.
- A dark Specular Color decreases the intensity, producing the same effect as using a low value for the Specular Intensity - both can be used for finer control.
- Use high Specular Power for shiny surfaces like metal and low values for soft looking materials like rubber.

Rimlight

Rimlight Color

The color of the Rimlight

Rimlight Fresnel

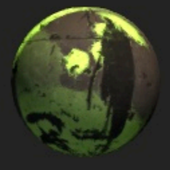
Fresnel for the Rimlight so it can be stronger on the edges

Rimlight Intensity

Modulates the intensity of the Rimlight

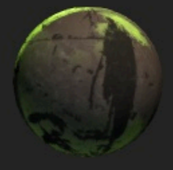
- The Rimlight property is useful when you need to make a prop stand out from the background.
- Used discreetly can be used to fake Global Illumination.
- The Rimlight is Self-Illuminated so it's not affected by any light.

Reflection



Reflection Color

The color of the Rimlight



Reflection Fresnel

Fresnel for the Rimlight so it can be stronger on the edges

Reflection Intensity

Modulates the intensity of the Rimlight

- The Reflection uses a static Cubemap so it doesn't actually reflects the environment real time.
- There are scripts that allow the artist to render a Cubemap from a specific position in a scene, in this way making the highly reflective objects integrate better in the environment.
- The Reflection intensity is modulated by the Alpha Channel in the Albedo Texture.

Emission

Emission Color

The Self-Illuminating color

Emission Map

Texture that can be used for Self-Illumination

Emission Intensity

The intensity of the Self-Illumination

- An Emissive shader doesn't actually emit light so it acts more like an Self-Illumination property.
- Emissiveness can be used for example to make a car headlights/stoplights look turned on but actual lights must be used to lit the environment.
- AnEmission Map can be used to define specific Emissive areas.
- The Emission Color will be multiplied with the Emission Map.
- In the next shader update, the Emission property will be renamed in Self-Illumination.

Uberkernels

```
          Some {threads, blocks} want to run f
          Some {threads, blocks} want to run g

// heterogeneous to
// homogeneous                              // uberkernel
split(c, stream_f,                          switch (c):
      stream_g)                               case 0:
f(stream_f)                                     b = f(a)
g(stream_g)                                 case 1:
combine(stream_f,                               b = g(a)
        stream_g)
```

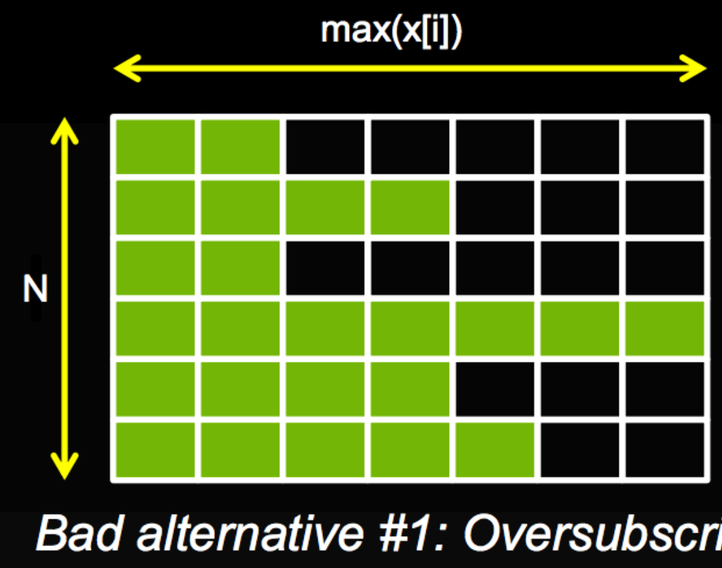
- Issue: Thread divergence
  - Possible fix: instead of thread granularity, use warp or block

Dynamic Parallelism

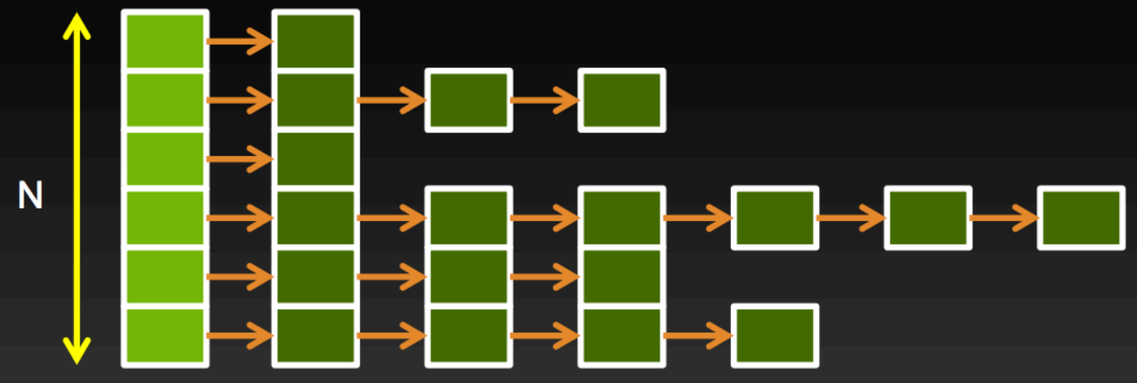
Dynamic Parallelism: Why

The Simplest Impossible Parallel Program

```
for i = 1 to N
    for j = 1 to x[i]
        convolution(i, j)
    next j
next i
```



Bad alternative #1: Oversubscription



Bad alternative #2: Serialisation

Dynamic Parallelism: Why

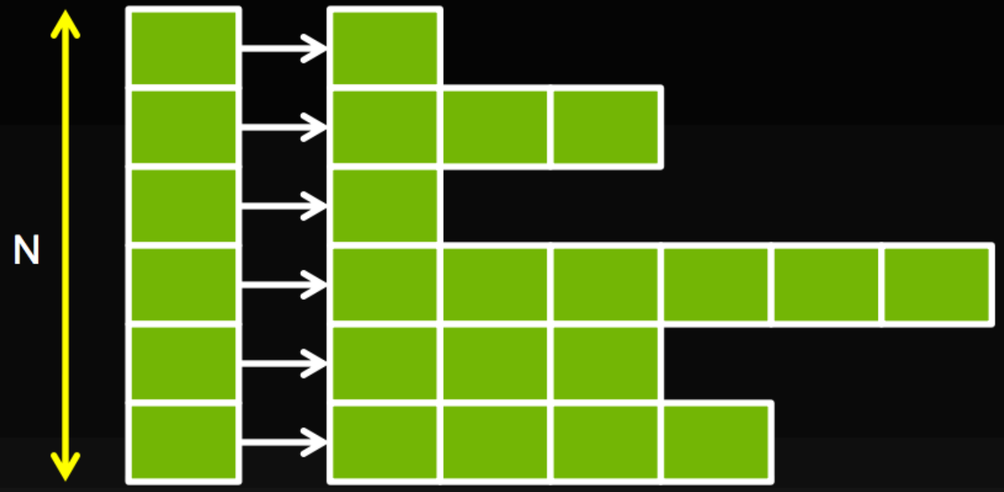
The Now-Possible Parallel Program

Serial Program

```
for i = 1 to N
    for j = 1 to x[i]
        convolution(i, j)
    next j
next i
```

CUDA Program

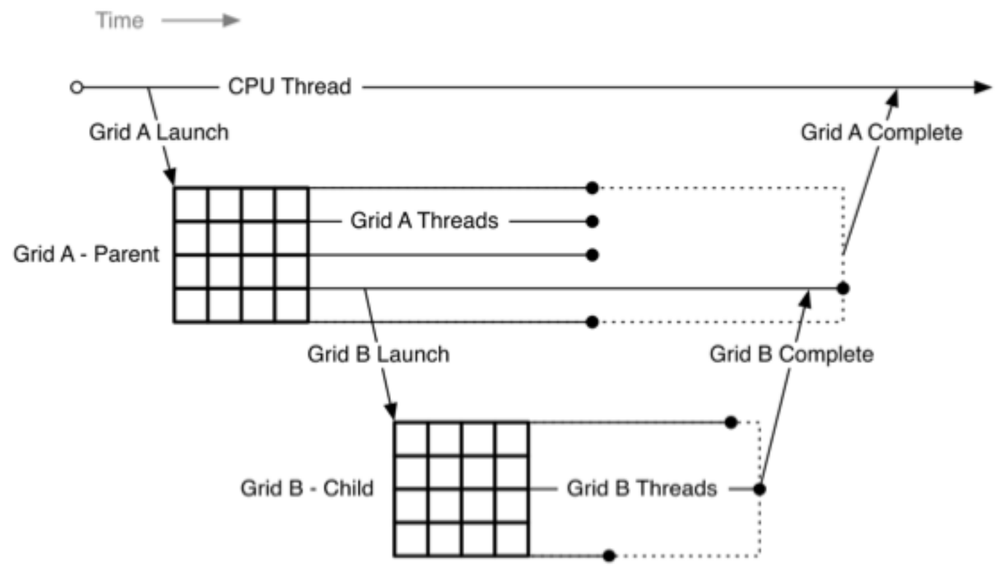
```
__global__ void convolution(int x[])
{
    for j = 1 to x[blockIdx]
        kernel<<< ... >>>(blockIdx, j)
}


    convolution<<< N, 1 >>>(x);
```



Now Possible: Dynamic Parallelism

NVIDIA Dynamic Parallelism

- Launch is per-thread, sync is per-block
- Parent grid launches child grids
- Child grids always complete before parent grids that launched them
- Synchronize and syncthreads calls both available



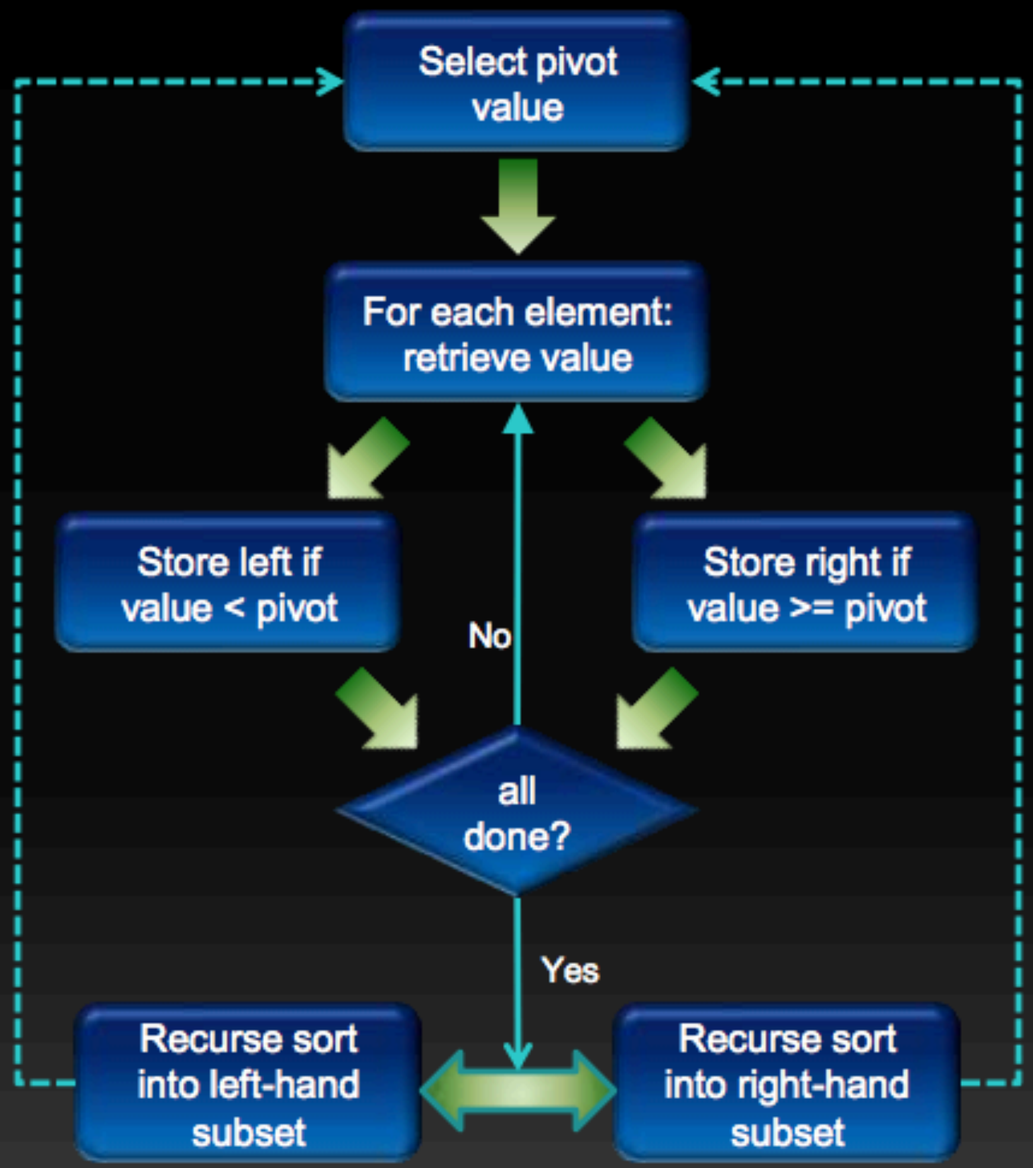

```
// every thread can launch a kernel
// or just one (as below)
if(threadIdx.x == 0) {
  child_k <<< (n + bs - 1) / bs, bs >>> ();
}
```

https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/

Quicksort using dynamic parallelism

Example 2: Parallel Recursion

```
__global__ void qsort(int *data, int l, int r)
{
    int pivot = data[0];
    int *lptr = data+l, *rptr = data+r;

    // Partition data around pivot value
    partition(data, l, r, lptr, rptr, pivot);

    // Launch next stage recursively
    if(l < (rptr-data))
        qsort<<< ... >>>(data, l, rptr-data);
    if(r > (lptr-data))
        qsort<<< ... >>>(data, lptr-data, r);
}
```



Persistent Threads

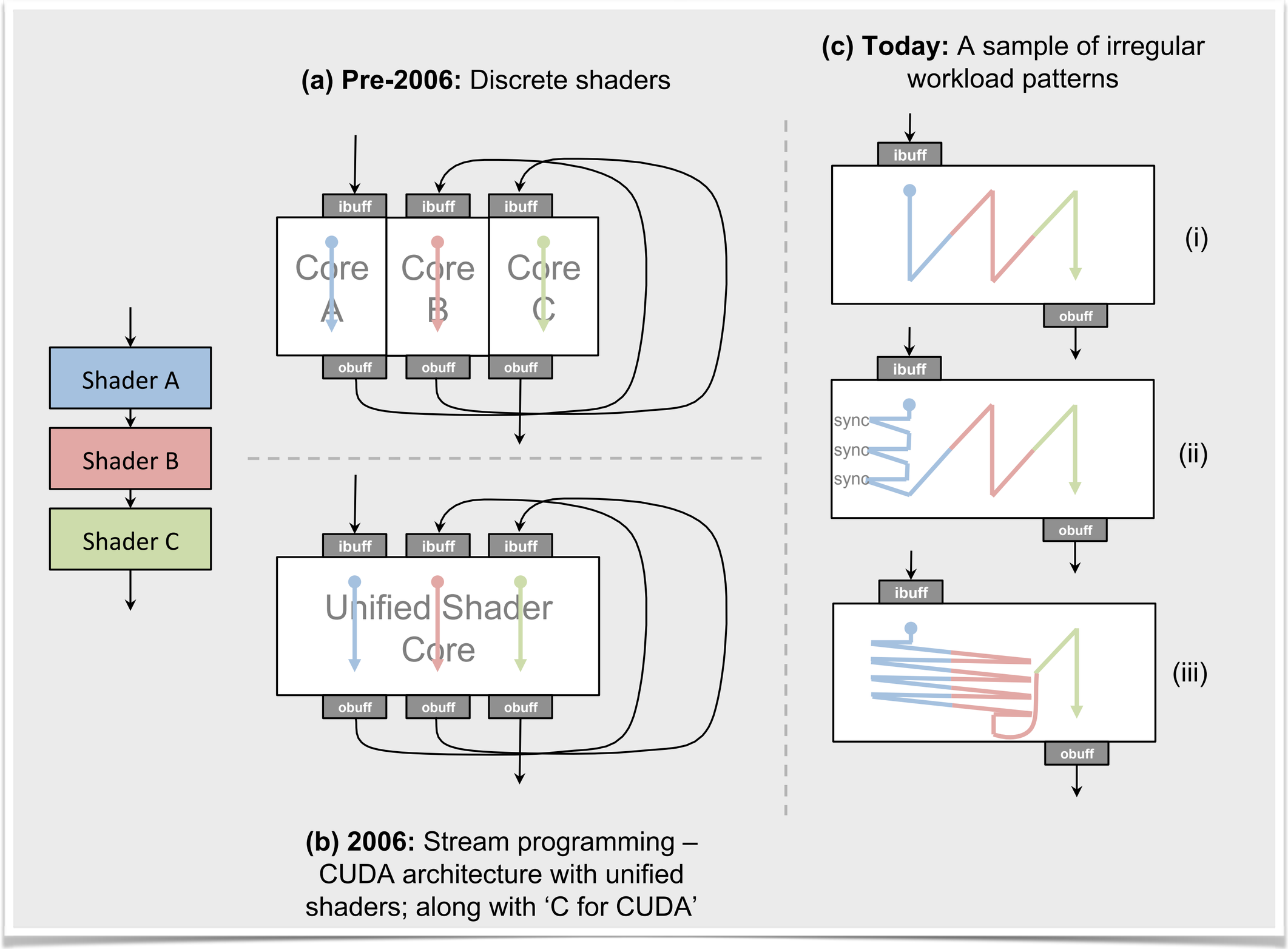
Persistent Threads

- GPU programming model suggests one thread per item
- What if you filled the machine with just enough threads to keep all processors busy, then asked each thread to stay alive until the input was complete?
  - Regular CUDA programming: Each thread fetches one piece of work from a previously determined location
    - fetch my piece of work; process work;
  - PT:

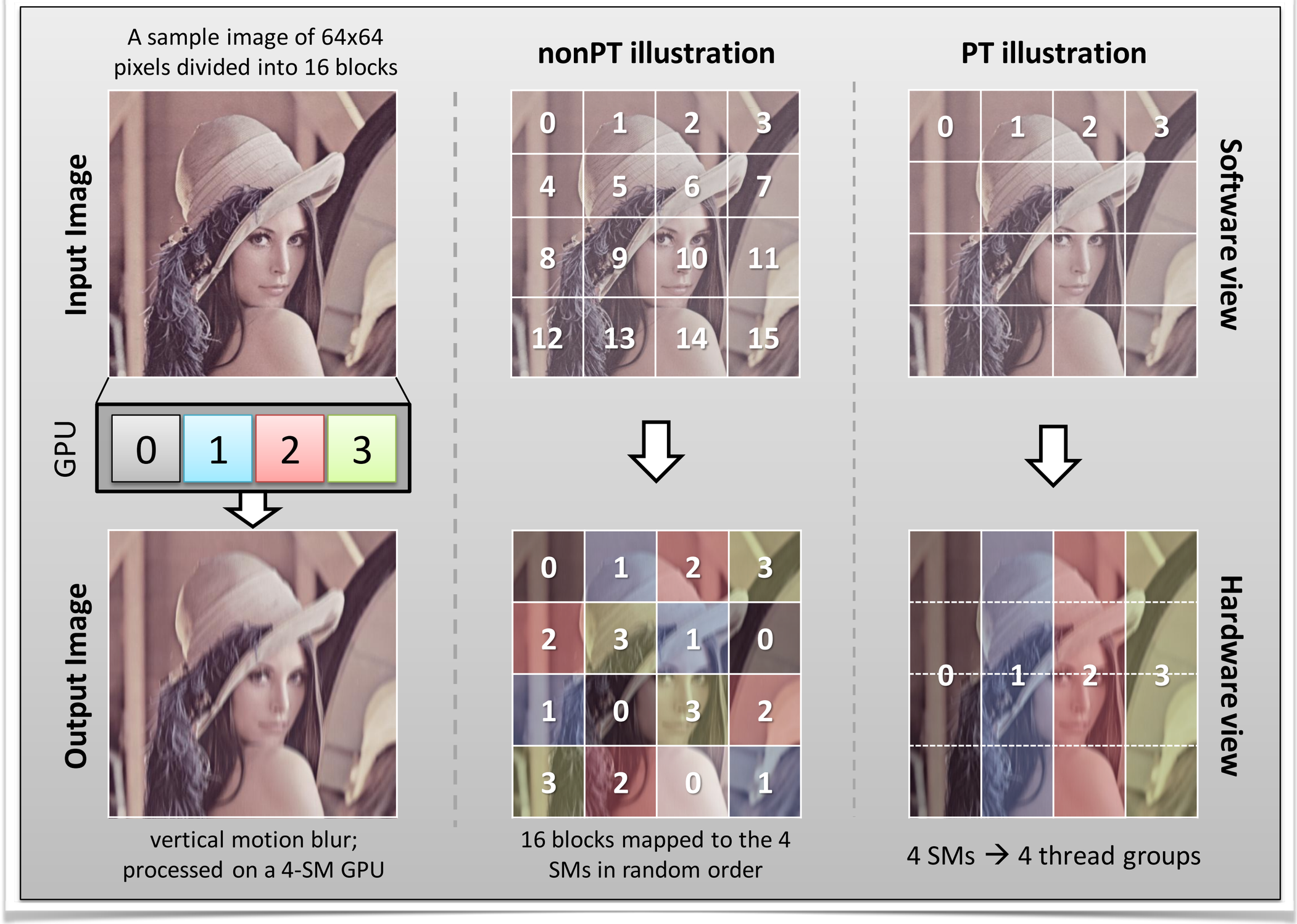
```
loop until no more work available {
  fetch work from queue; process work;
}
```

- Minus: More overhead per thread (register pressure)
- Minus: Violent anger of vendors

Evolution in workloads



Persistent threads: An example



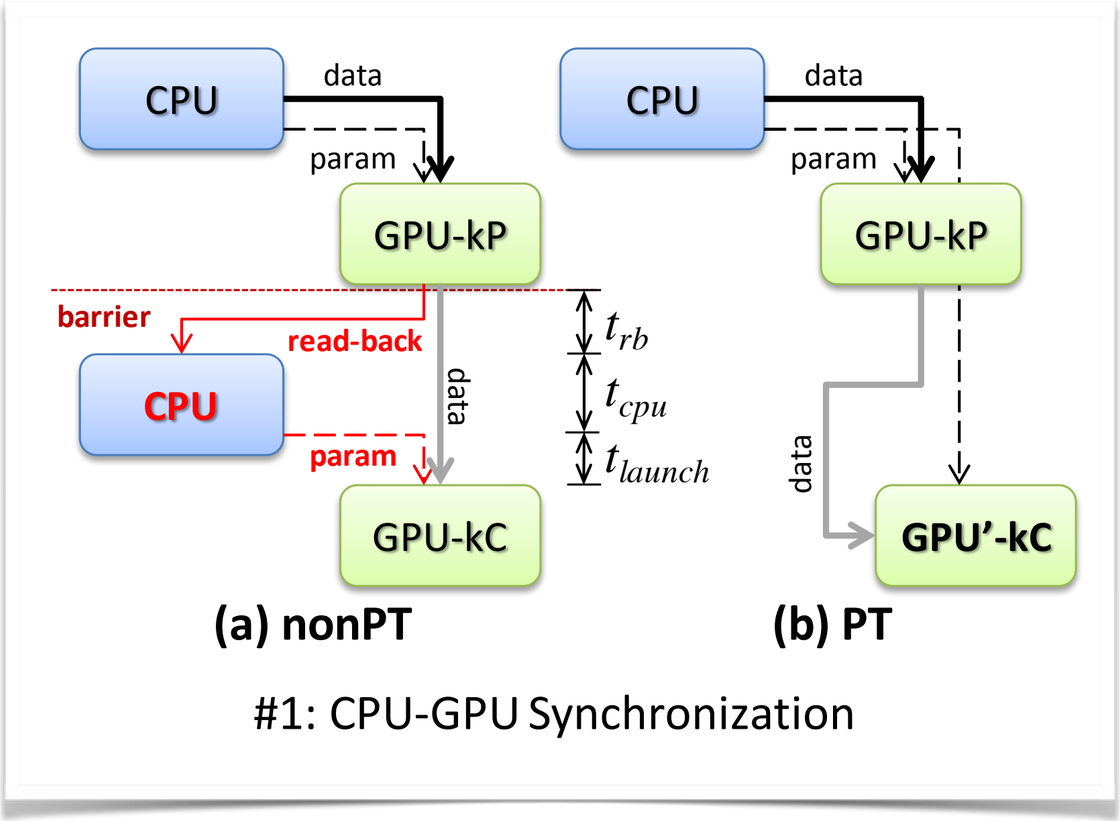
Persistent Threads Use Cases

Use Case	Scenario	Advantage of Persistent Threads
CPU-GPU Synchronization	Kernel A produces a variable amount of data that must be consumed by Kernel B	nonPT implementations require a round-trip communication to the host to launch Kernel B with the exact number of blocks corresponding to work items produced by Kernel A.
Load Balancing	Traversing an irregularly-structured, hierarchical data structure	PT implementations build an efficient queue to allow a single kernel to produce a variable amount of output per thread and load balance those outputs onto threads for further processing.
Maintaining Active State	A kernel accumulates a single value across a large number of threads, or Kernel A wants to pass data to Kernel B through shared memory or registers	Because a PT kernel processes many more items per block than a nonPT kernel, it can effectively leverage shared memory across a larger block size for an application like a global reduction.
Global Synchronization	Global synchronization within a kernel across thread blocks	In a nonPT kernel, synchronizing across blocks within a kernel is not possible because blocks run to completion and cannot wait for blocks that have not yet been scheduled. The PT model ensures that all blocks are resident and thus allows global synchronization.

Kshitij Gupta, Jeff Stuart, and John D. Owens. **A Study of Persistent Threads Style GPU Programming for GPGPU Workloads**.
In *Proceedings of Innovative Parallel Computing*, InPar '12, May 2012

CPU/GPU Synchronization

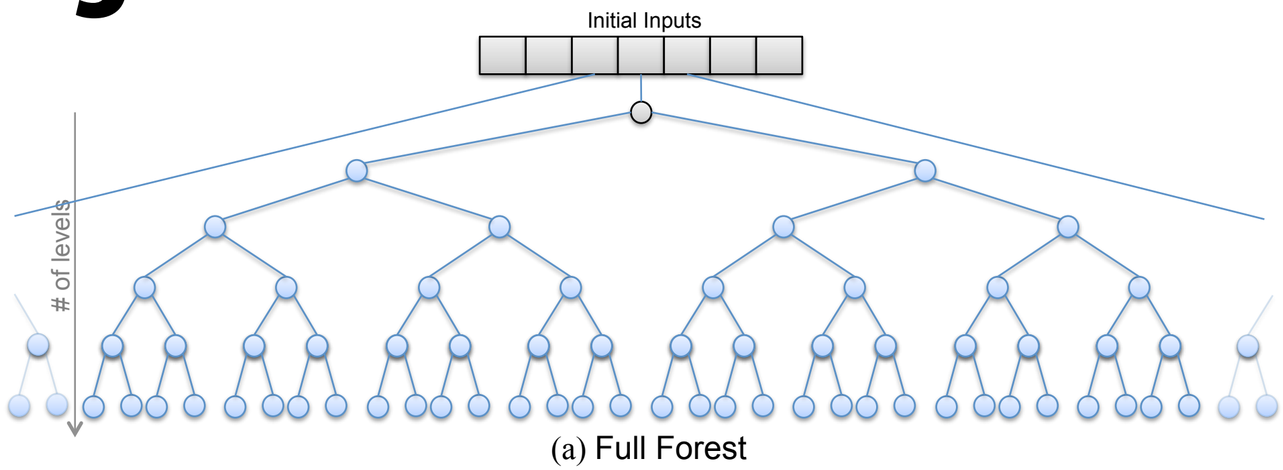
- **Problem:**
  - **Kernel P produces variable amount of output**
  - **Kernel C consumes output of Kernel P**
    - **To properly launch kernel C with correct size, need round-trip to host**
- **Solution:**
  - **Combine P and C into one uberkernel; phases separated by global barrier**



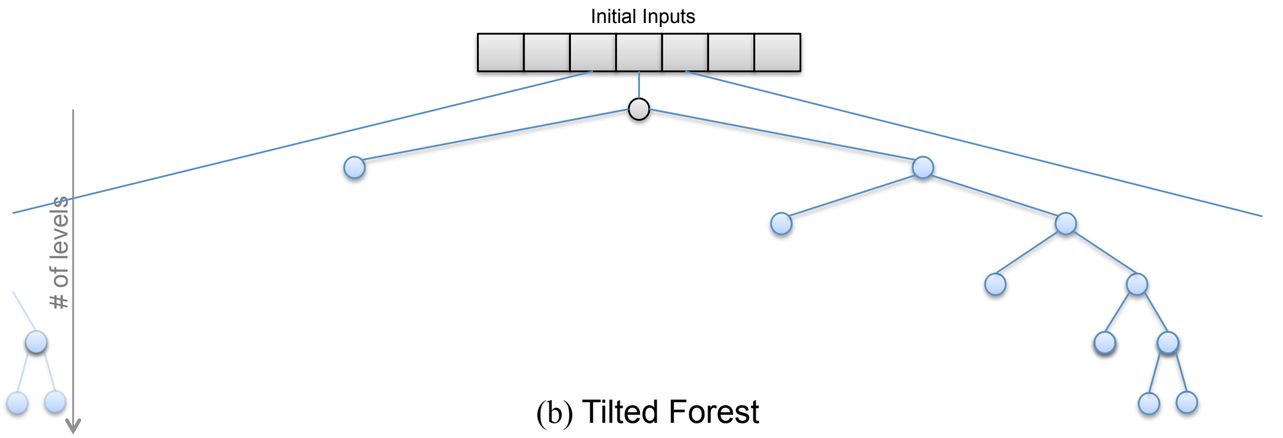
#1: CPU-GPU Synchronization

Load Balancing for Irregular Parallelism

- Consider generating a tree
- Each level is a new kernel
- Problems:
  - Multiple kernels
  - Return control to host for each kernel



(a) Full Forest

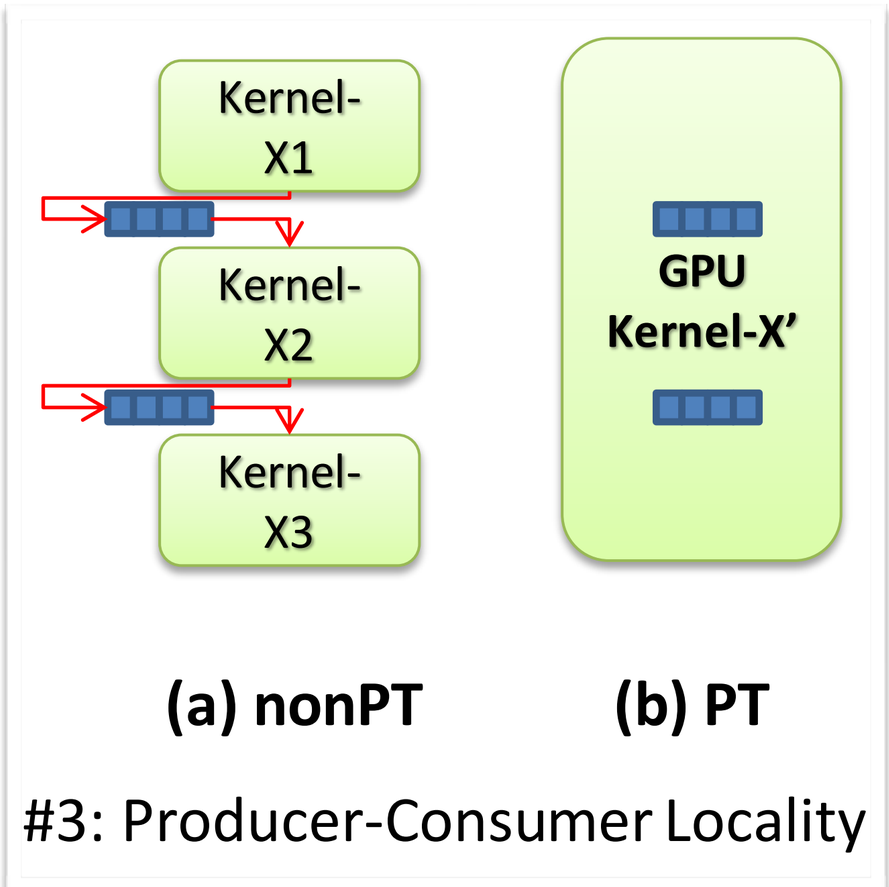


(b) Tilted Forest

- Solution:
  - Single kernel, single global queue for work, threads pull work from bottom of queue, push work onto top of queue

Producer-Consumer Locality

- **Problem:**
  - **Kernel X1 has interesting state in registers**
  - **Kernel X2 wants to inherit that state, but has a size not determined until runtime**
  - **Once Kernel X1 ends, all registers are void**
  - **Have to write all state out to global memory at end of X1, X2, X3**
  - **CUDA has no concept of p-c locality**
- **Solution**
  - **Single PT kernel can store state between kernels if state is *O*(1)**

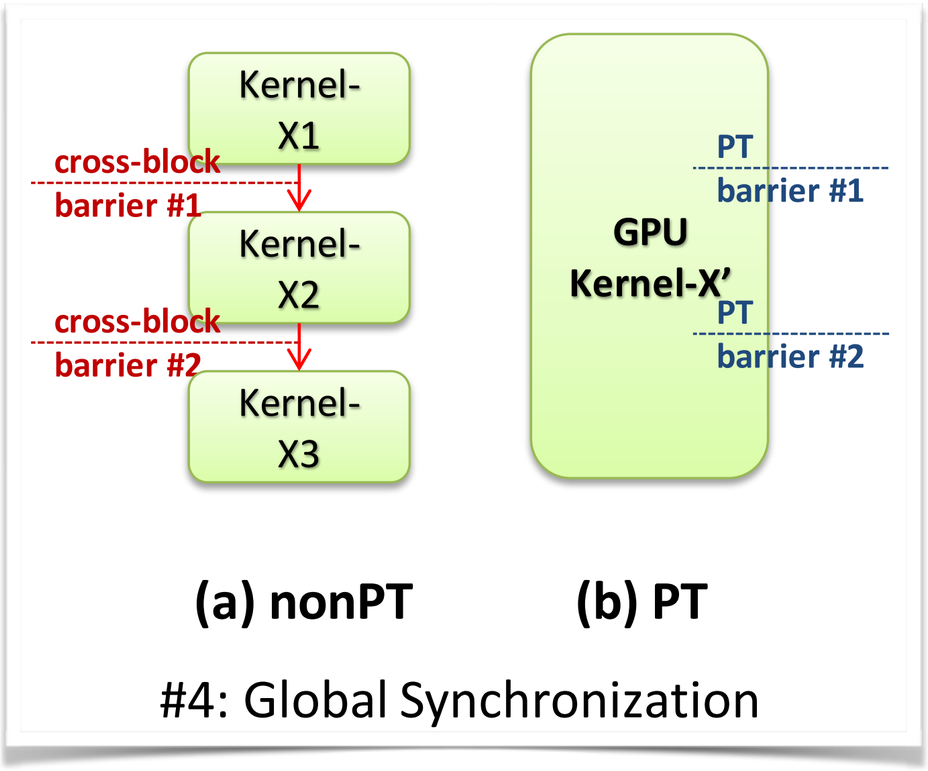


#3: Producer-Consumer Locality

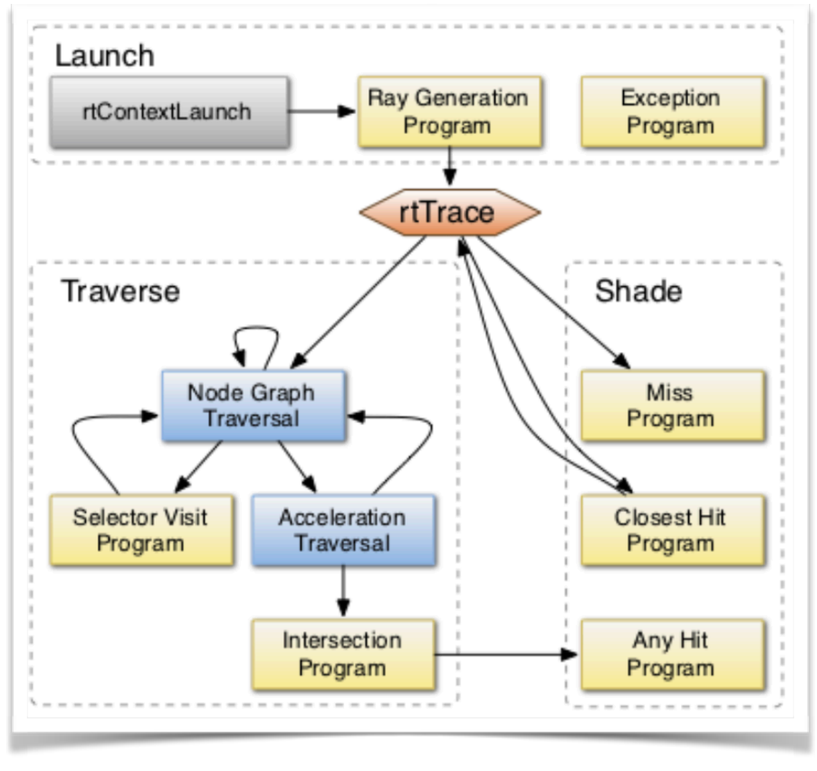
This is the only use case we really liked.

GPU Global Synchronization

- **Problem**
  - **Global barriers in CUDA require a kernel boundary, which is expensive**
- **Solution**
  - **All blocks are resident in a PT kernel**
  - **Can atomically increment a global counter and wait until that counter hits the number of SMs (blocks) on the machine**



NVIDIA OptiX



- Yellow boxes: user-specified
- Blue boxes: OptiX internal

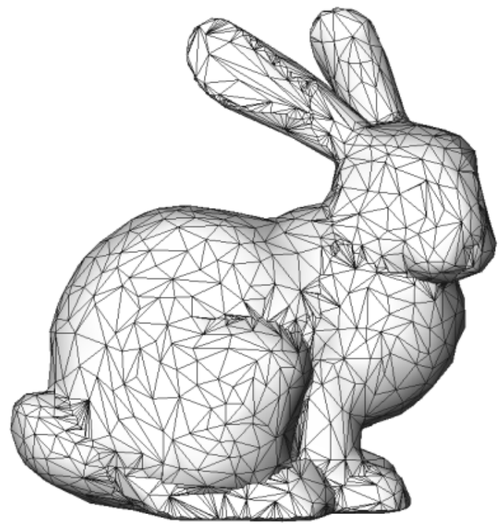
- "The monolithic kernel, or *megakernel*, approach proves successful on modern GPUs [Aila and Laine 2009]. This approach minimizes kernel launch overhead but potentially reduces processor utilization as register requirements grow to the maximum across constituent kernels. Because GPUs mask memory latency with multi-threading, this is a delicate tradeoff. OptiX implements a megakernel by linking together a set of individual user programs and traversing the state machine induced by execution flow between them at runtime."

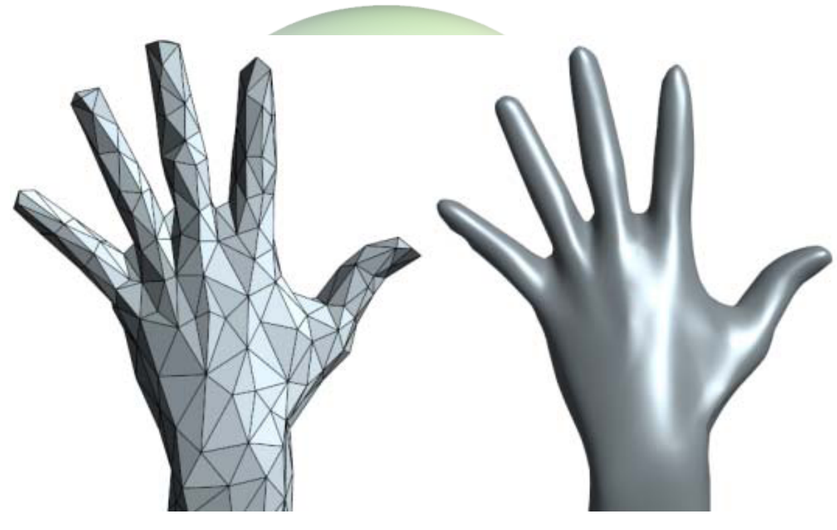
S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: A general purpose ray tracing engine," ACM Transactions on Graphics, vol. 29, no. 4, p. 1, Sep. 2010.

Reyes Dice-and-Split

Representing Smooth Surfaces

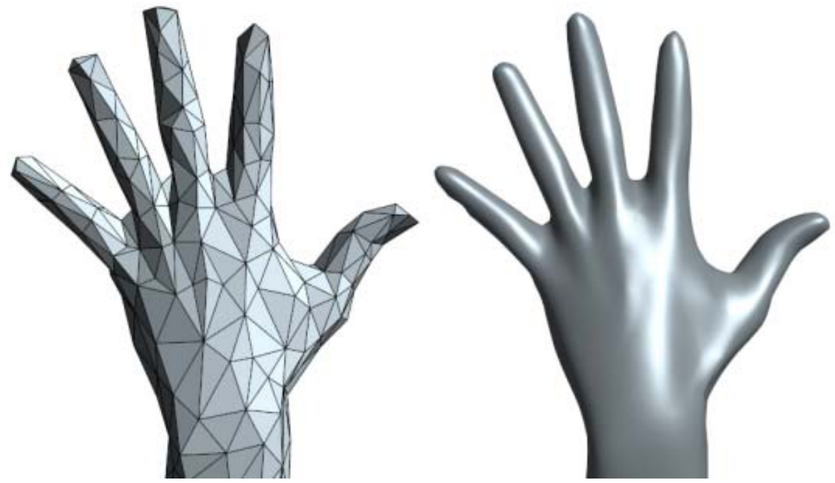
- Polygonal Meshes
  - Lack view adaptivity
  - Inefficient storage/transfer
- Parametric Surfaces
  - Collections of smooth patches
  - Hard to animate
- Subdivision Surfaces
  - Widely popular, easy for modeling
  - Flexible animation

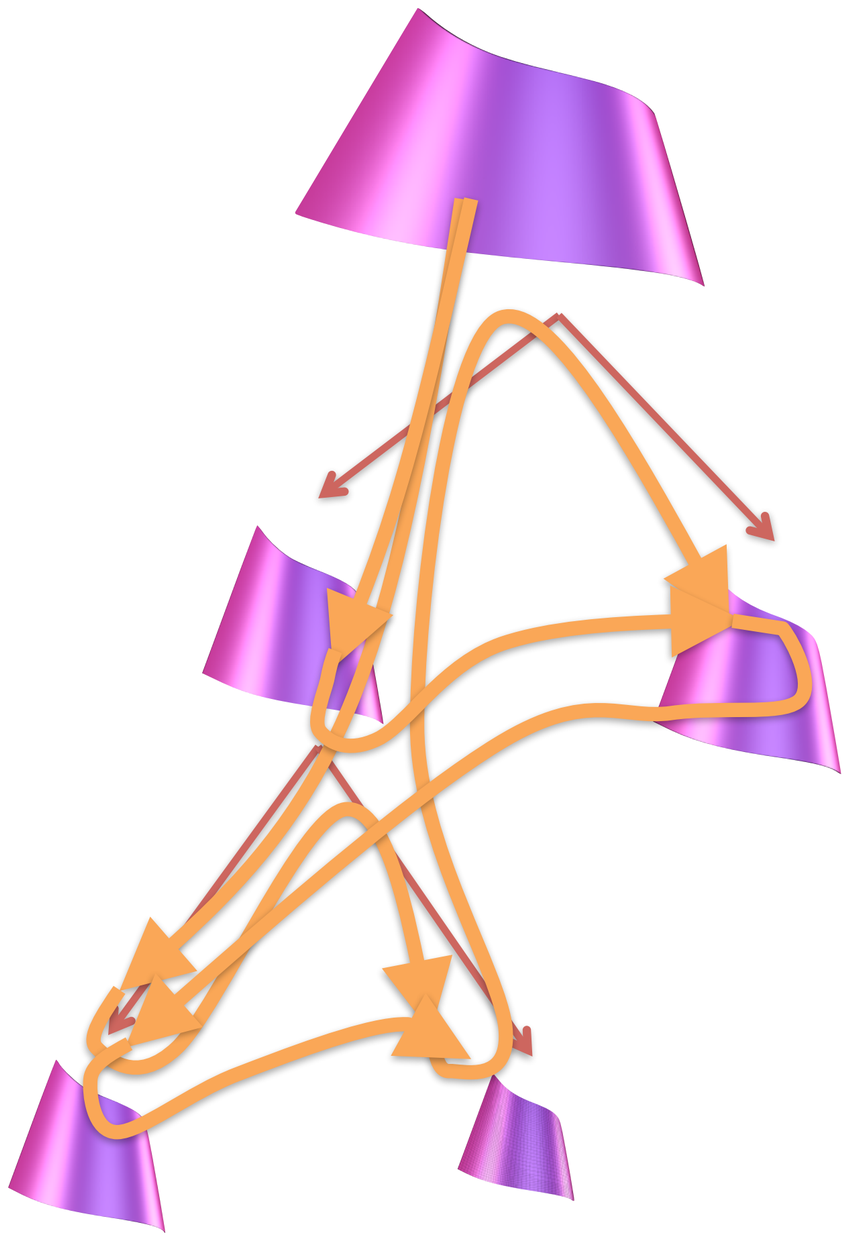




[Boubekeur and Schlick 2007]

Parallel Traversal

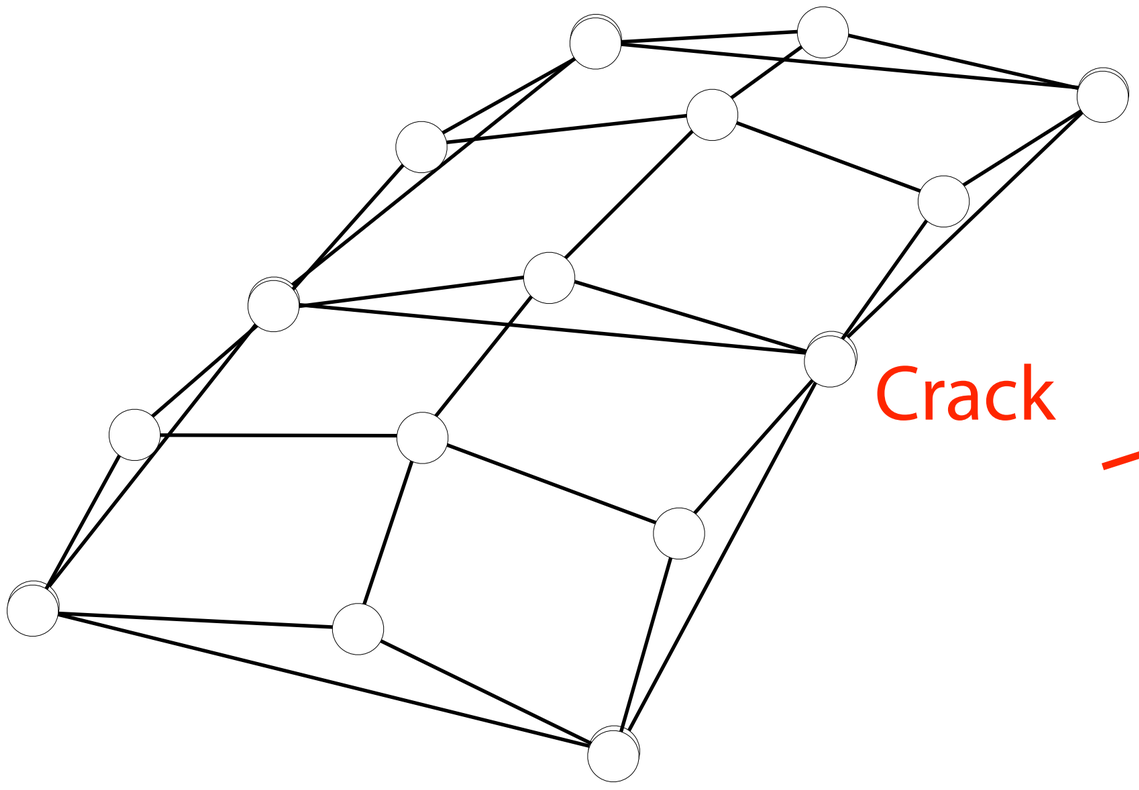




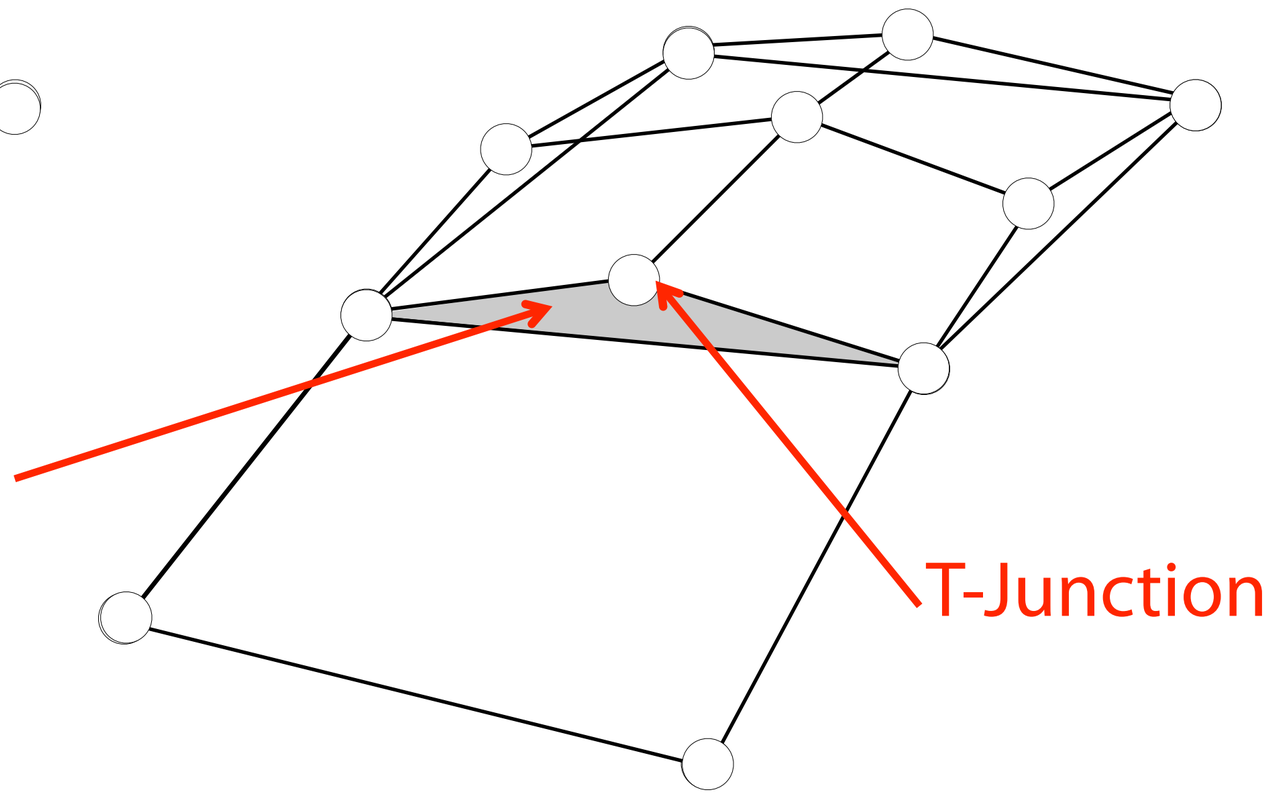
Anjul Patney and John D. Owens. **Real-Time Reyes-Style Adaptive Surface Subdivision**. *ACM Transactions on Graphics*, 27(5):143:1–143:8, December 2008.

Cracks & T-Junctions

Uniform Refinement



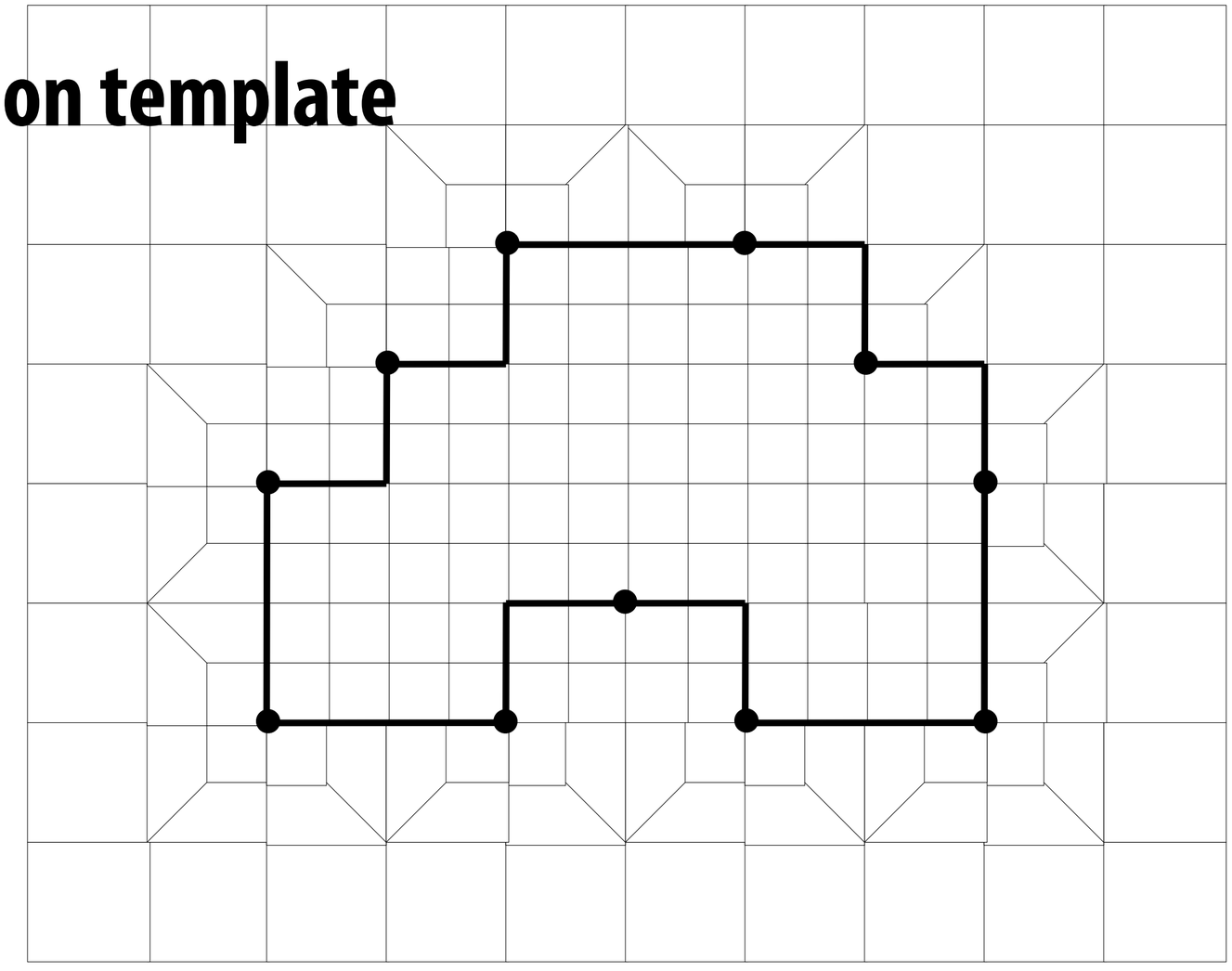
Adaptive Refinement

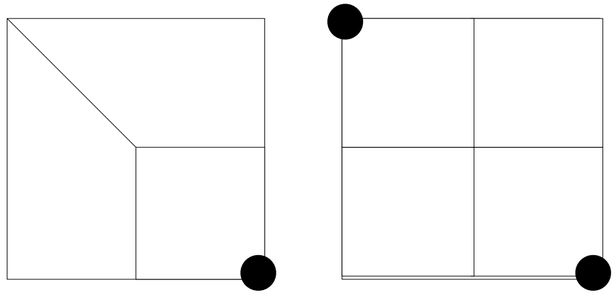


- Previous work was post-process after all steps; ours is inherent within subdivision process; {step, refine, repeat}

Incrementally Resolving Cracks

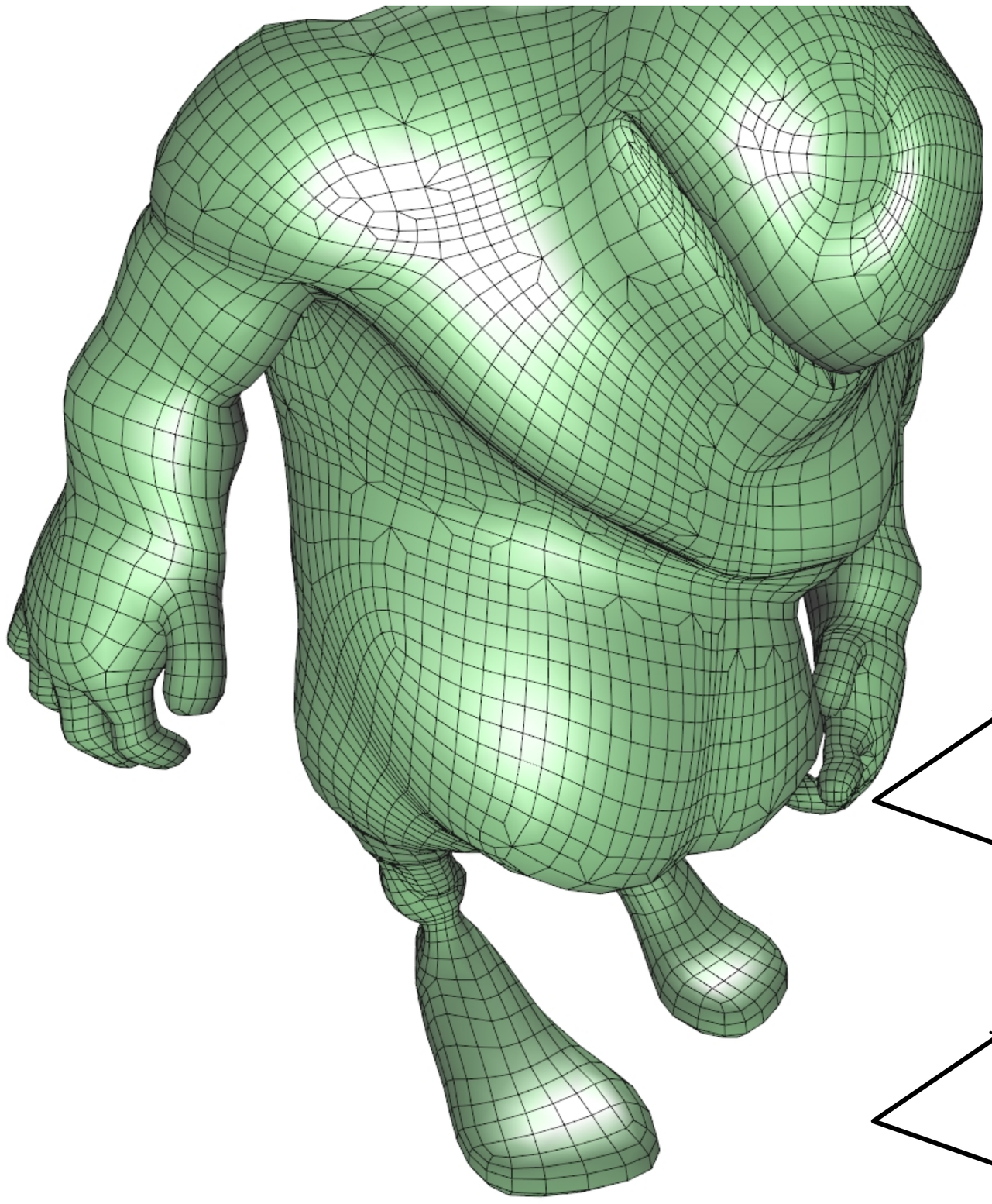
- "Active" Vertices
  - Placed at alternating transition vertices
  - Help choosing a subdivision template

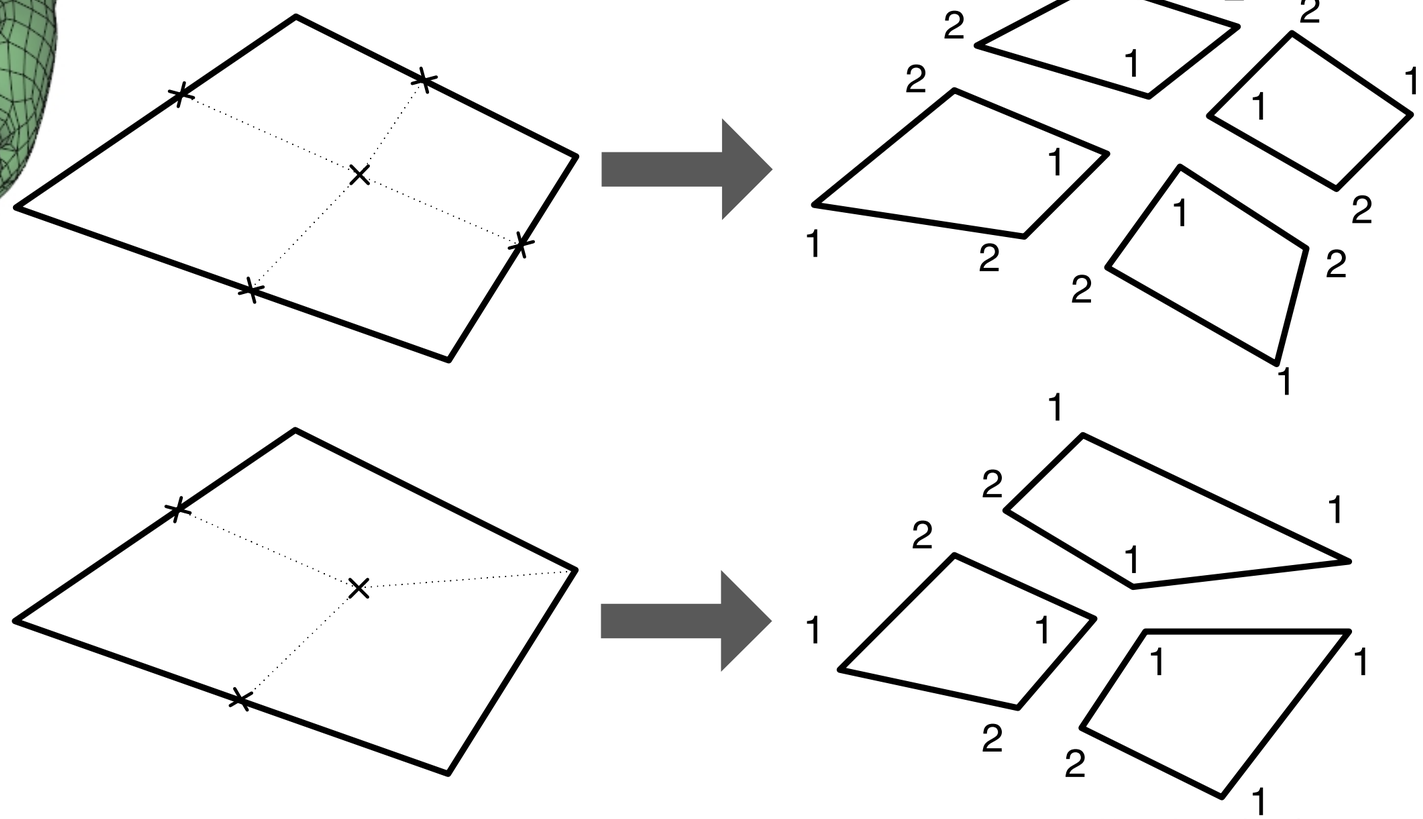





08/03/09 10:17 am
08/03/09 10:18 am

Recursive Subdivision is Irregular

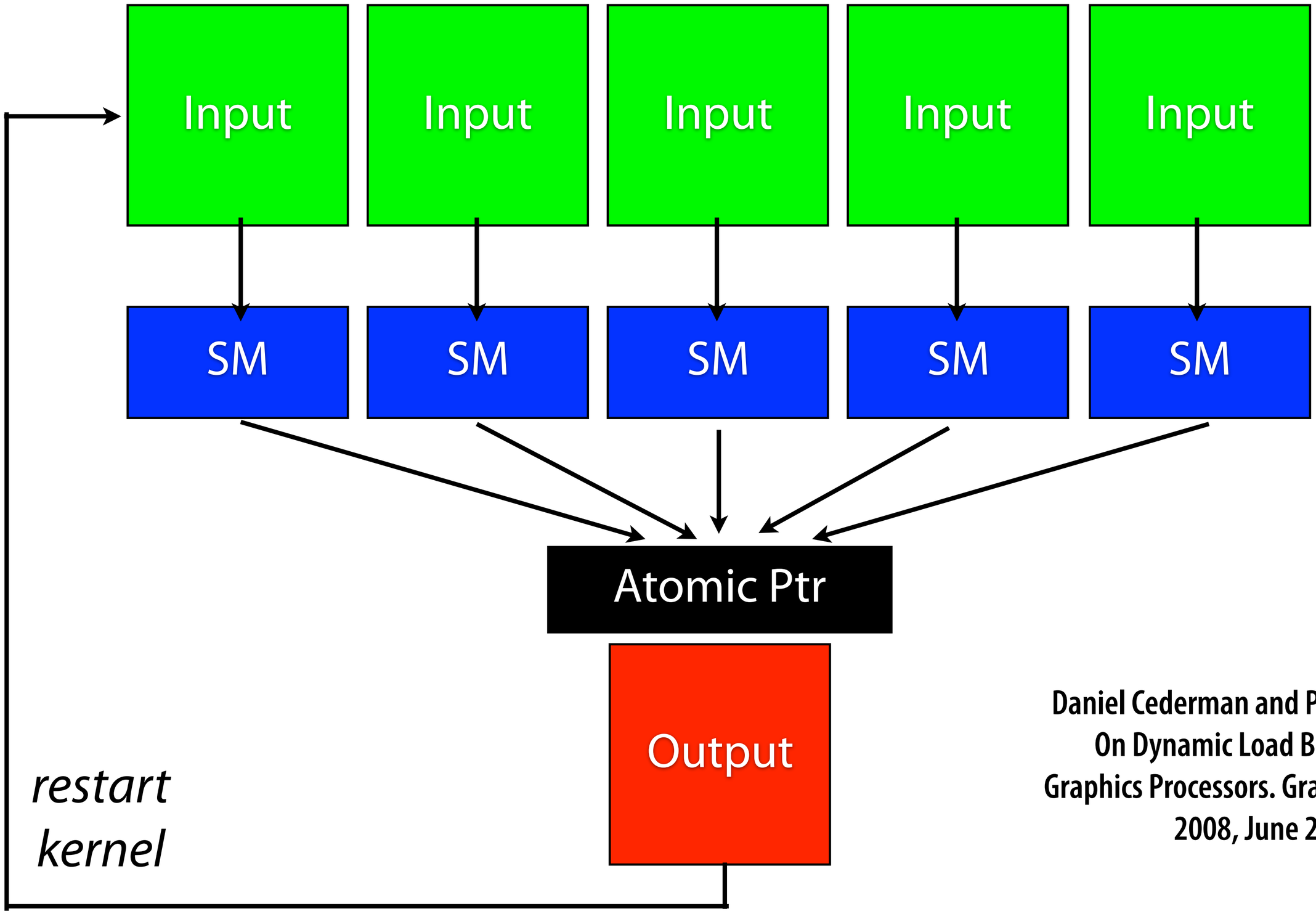




Patney, Ebeida, and Owens. “Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces”. HPG ’09.

Task Queues

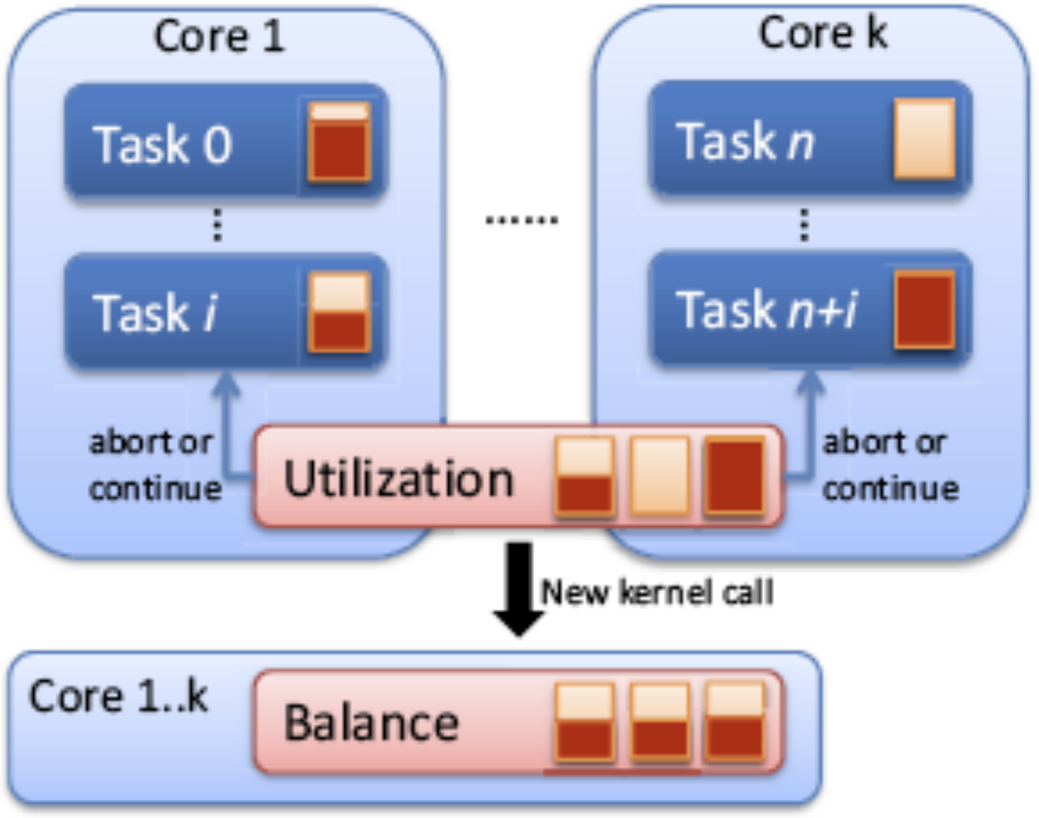
Static Task List



Daniel Cederman and Philippas Tsigas, On Dynamic Load Balancing on Graphics Processors. Graphics Hardware 2008, June 2008.

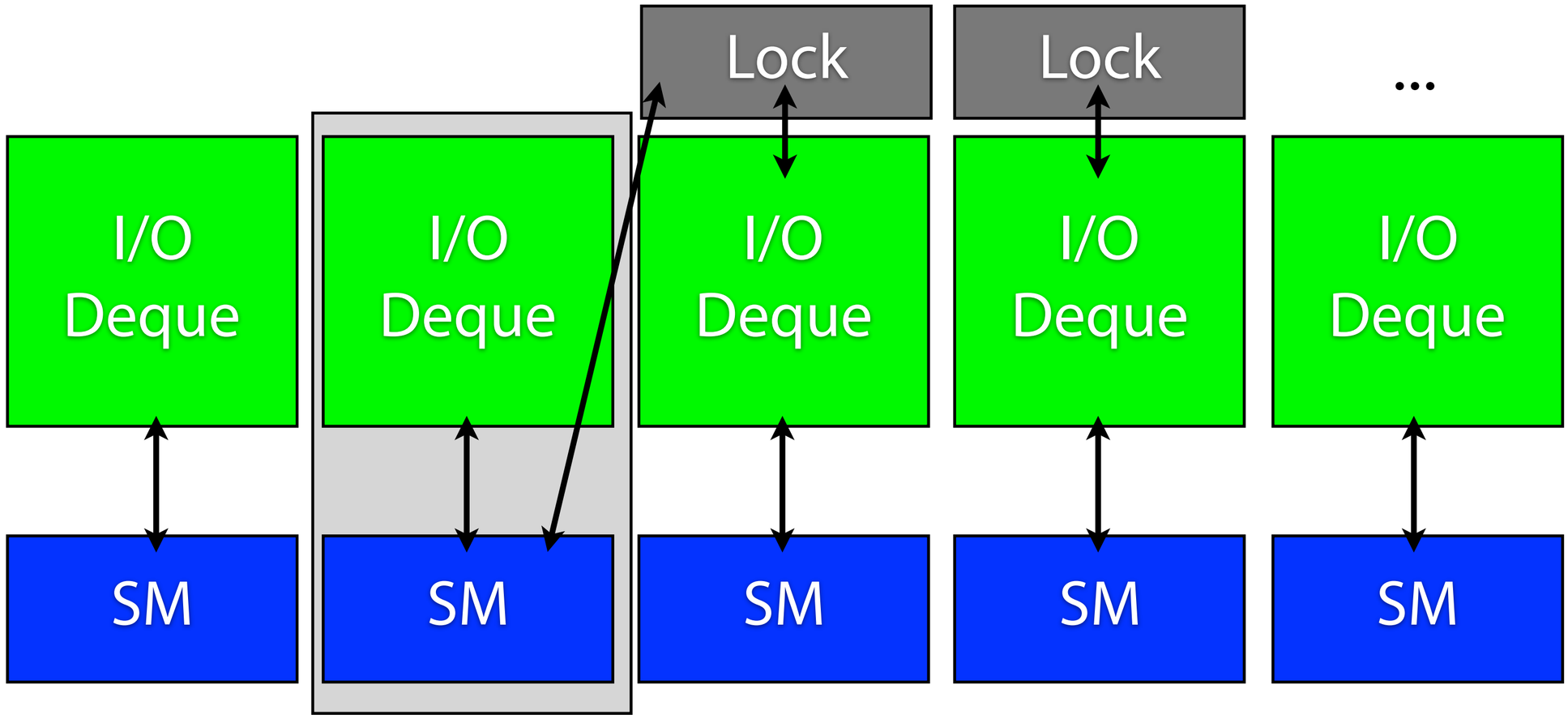
Private Work Queue Approach

- Allocate private work queue of tasks per core
  - Each core can add to or remove work from its local queue
- Cores mark self as idle if {queue exhausts storage, queue is empty}
- Cores periodically check global idle counter
- If global idle counter reaches threshold, rebalance work



gProximity: Fast Hierarchy Operations on GPU Architectures, Lauterbach, Mo, and Manocha, EG '10

Work Stealing & Donating



- Cederman and Tsigas: Stealing == best performance and scalability (follows Arora CPU-based work)
- We showed how to do this with multiple kernels in an uberkernel and persistent-thread programming style
- We added donating to minimize memory usage

Ingredients for Our Scheme

Implementation 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

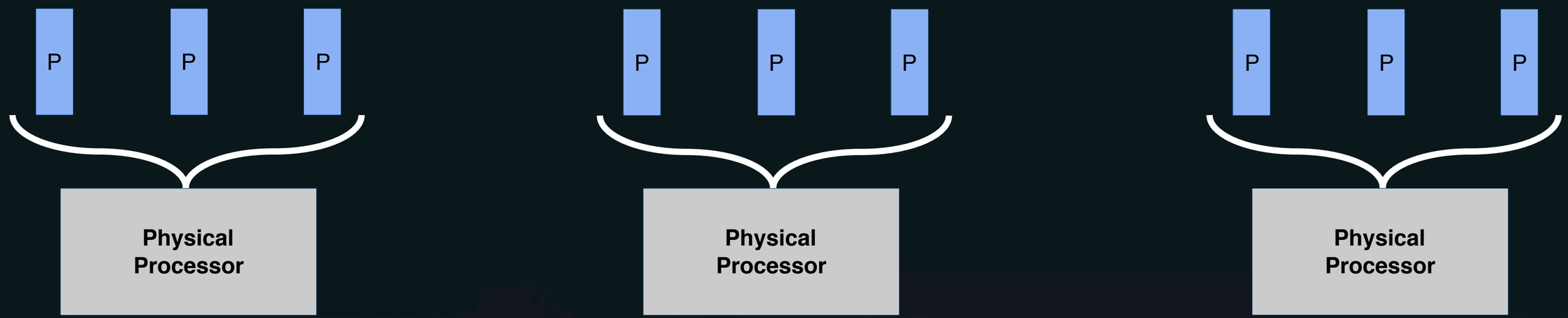
Uberkernels

Task Donation

Stanley Tzeng, Anjul Patney, and John D. Owens. **Task Management for Irregular-Parallel Workloads on the GPU**. In *Proceedings of High Performance Graphics 2010*, pages 29–37, June 2010.

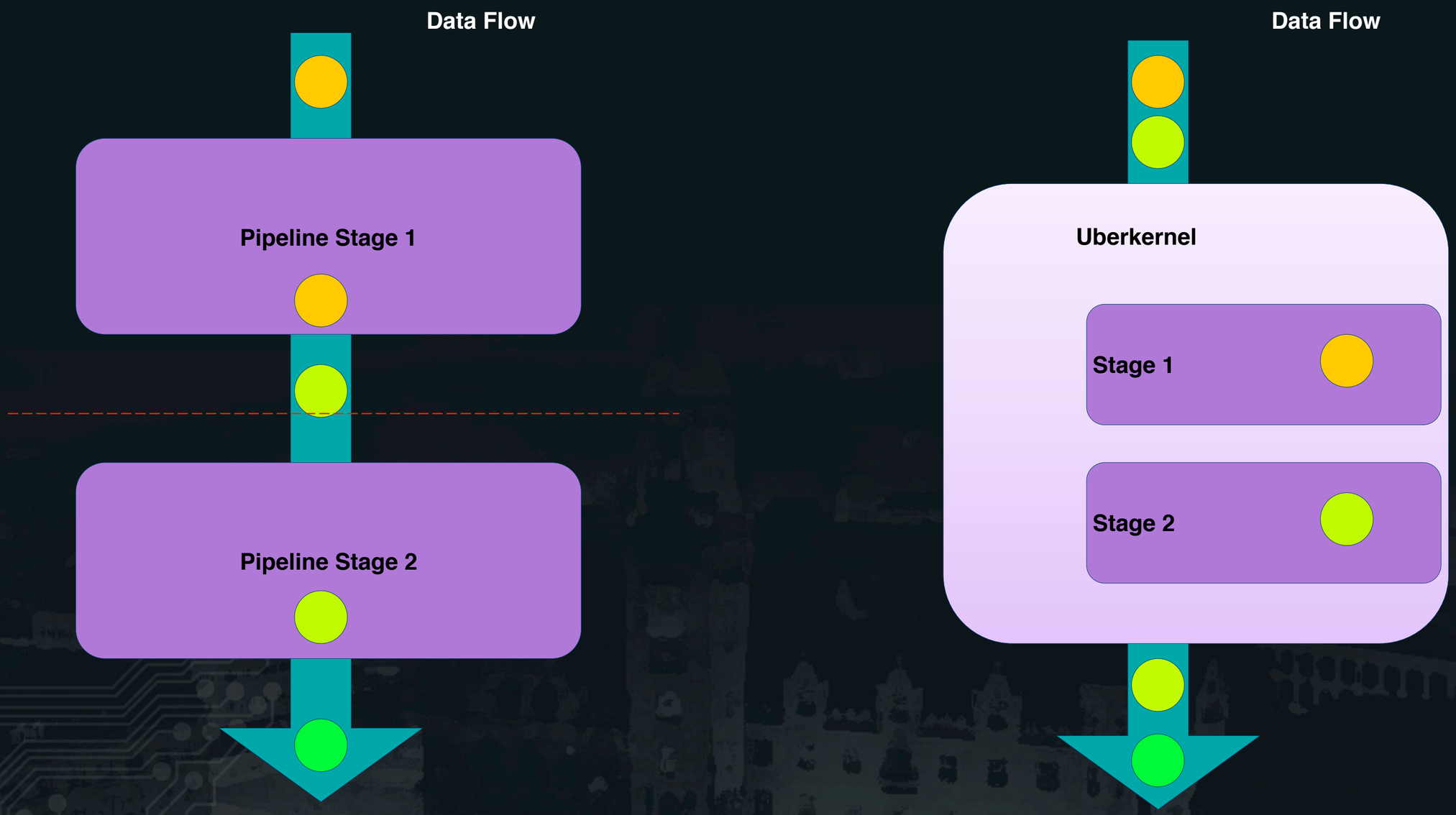
Warp Size Work Granularity

- 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

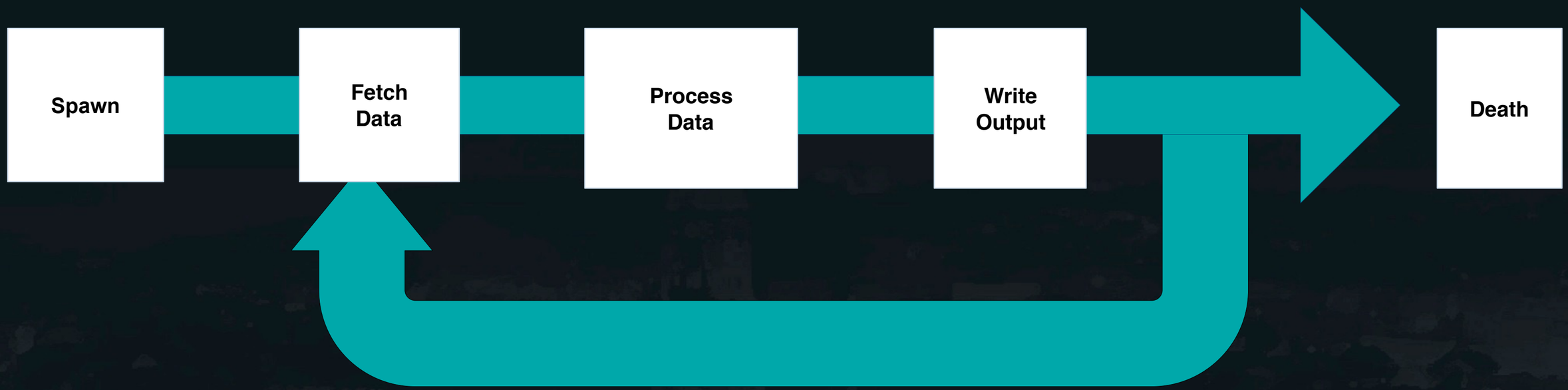
- Problem: Want to eliminate global kernel barriers for better processor utilization
- Uberkernels pack multiple execution routes into one kernel.



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 Persistent Thread:

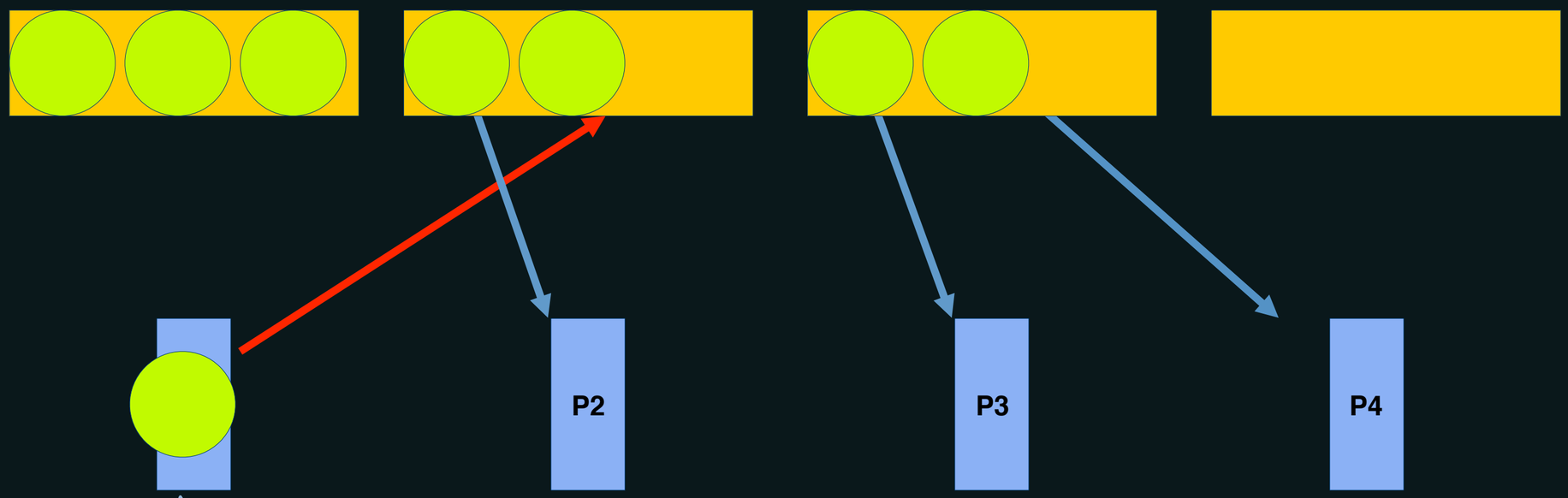


How do we know when to stop?

When there is no more work left

Task Donation

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

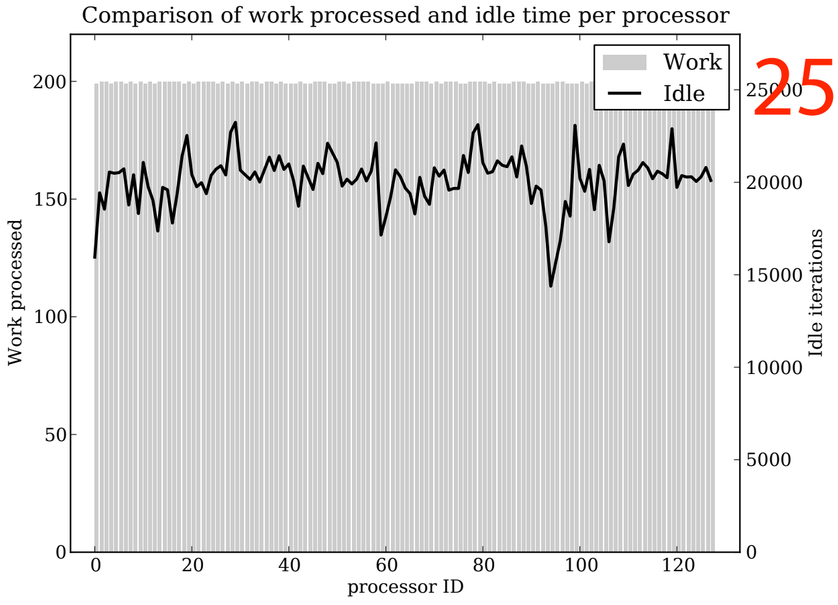


This processor ‘s bin is full and donates its work to someone else’s bin

Smaller memory usage

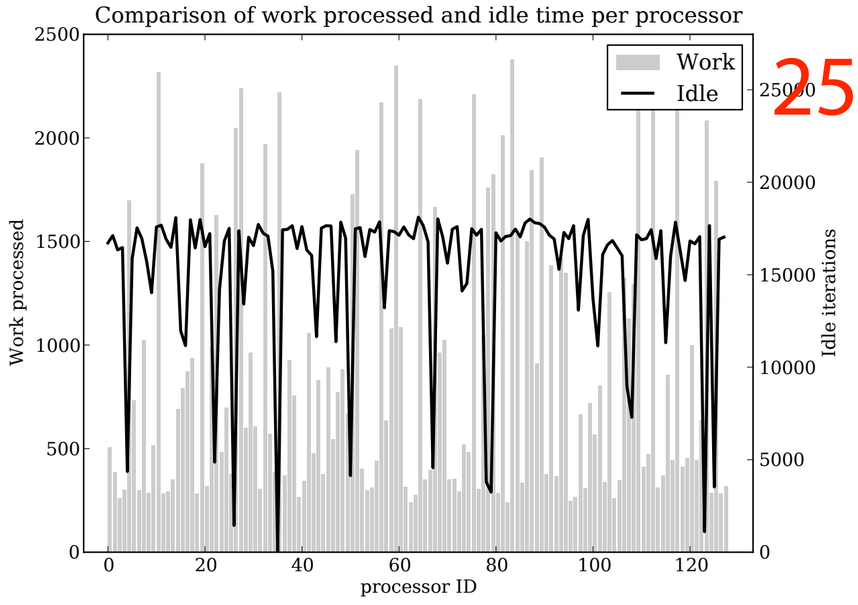
More complicated

Queuing Schemes



25k

(a) Block Queuing

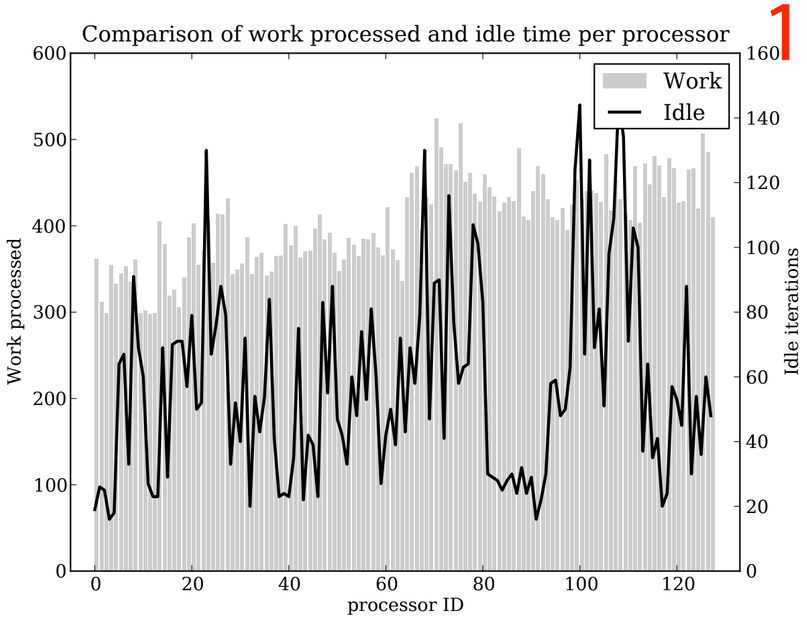


25k

(b) Distributed Queuing

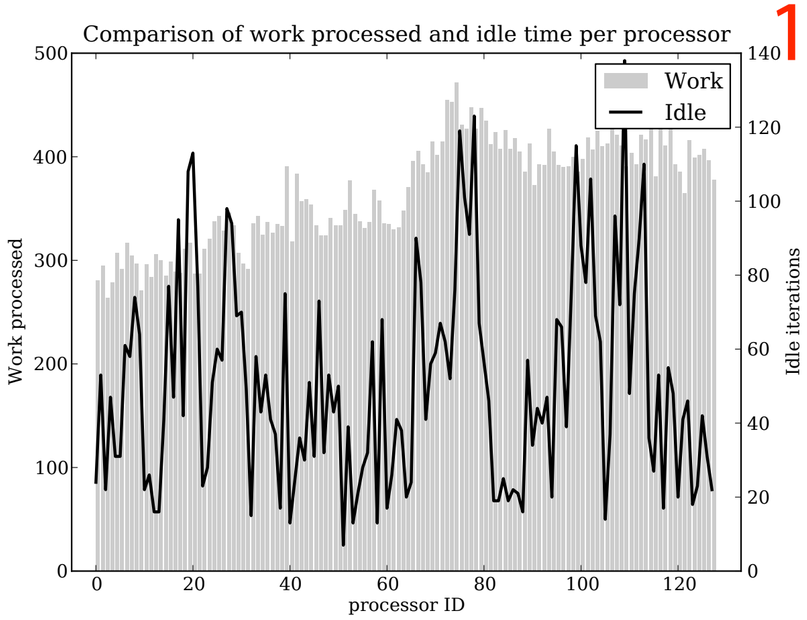
Gray bars: Load balance

Black line: Idle time



160

(c) Task Stealing

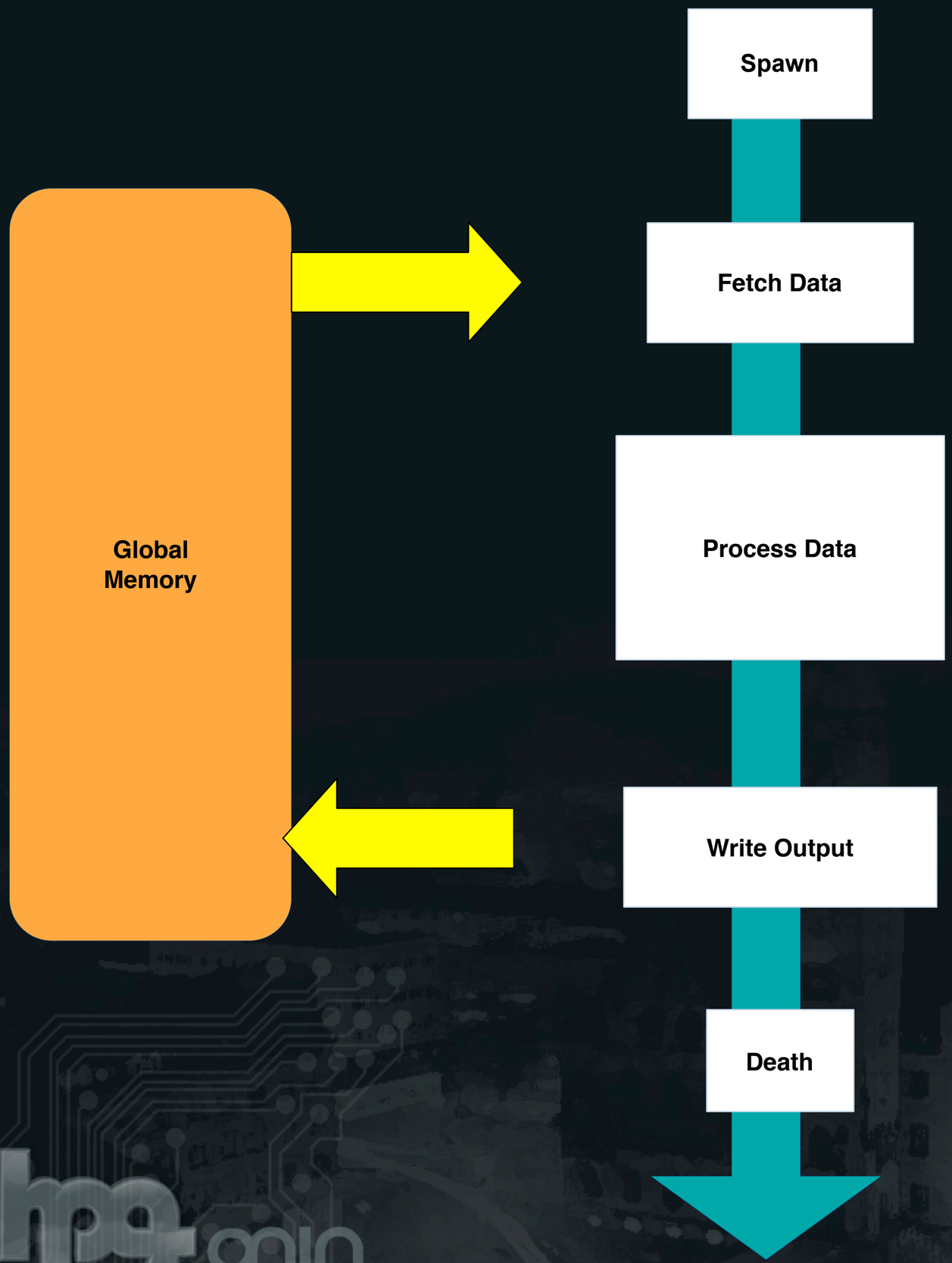


140

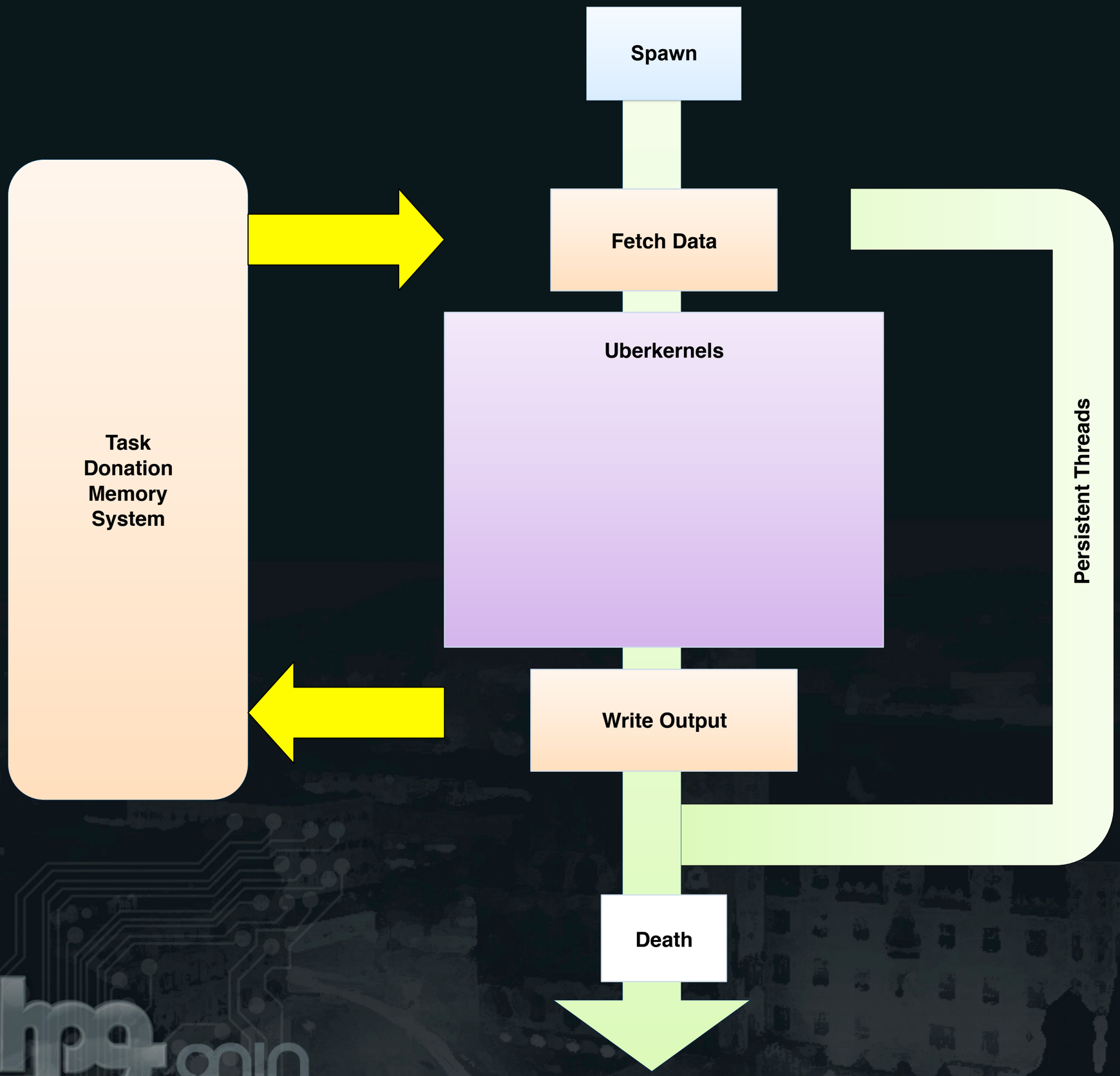
(d) Task Donating

Stanley Tzeng, Anjul Patney, and John D. Owens. Task Management for Irregular-Parallel Workloads on the GPU. In Proceedings of High Performance Graphics 2010, pages 29–37, June 2010.

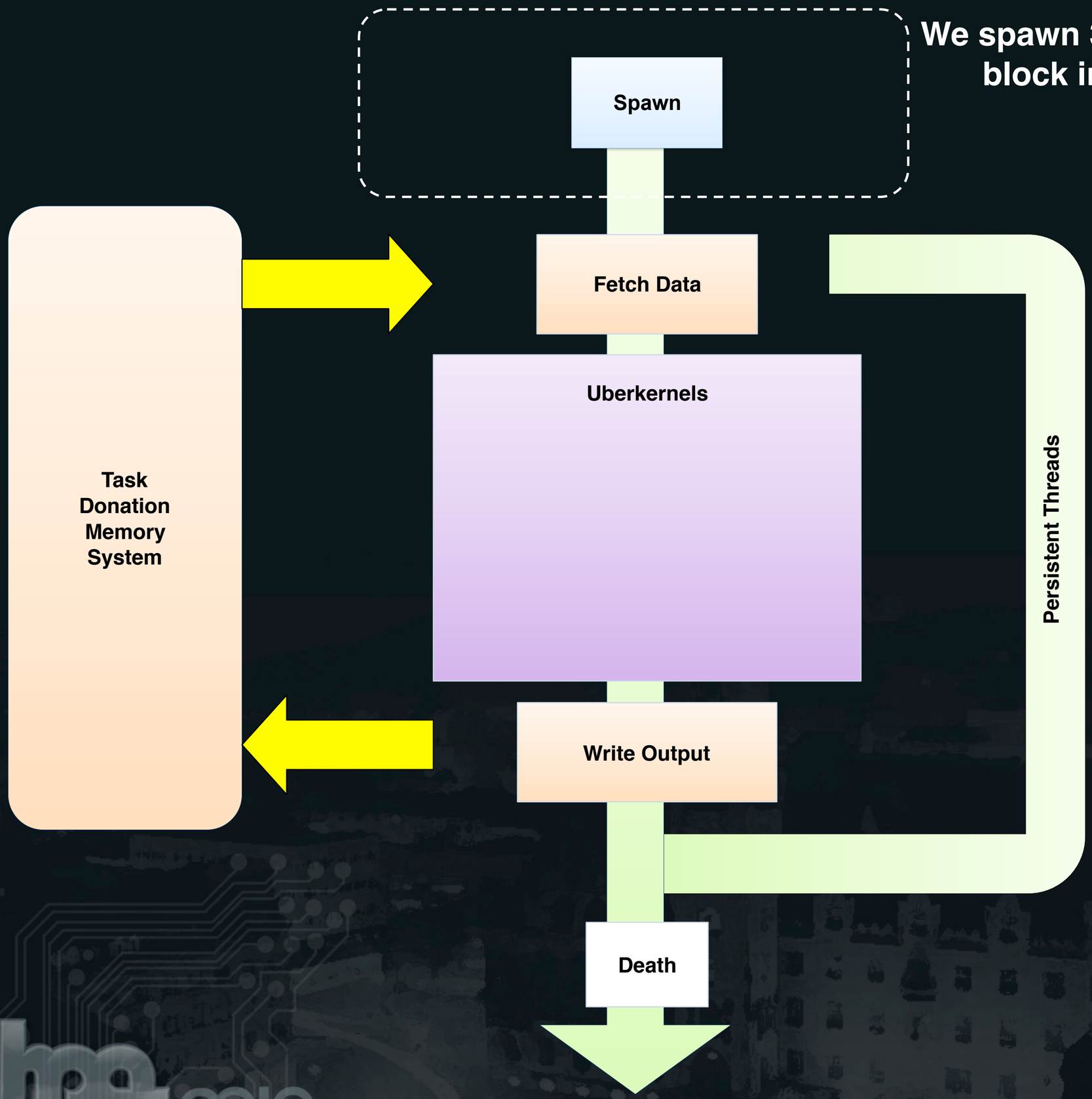
How it All Fits Together



Our Version


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

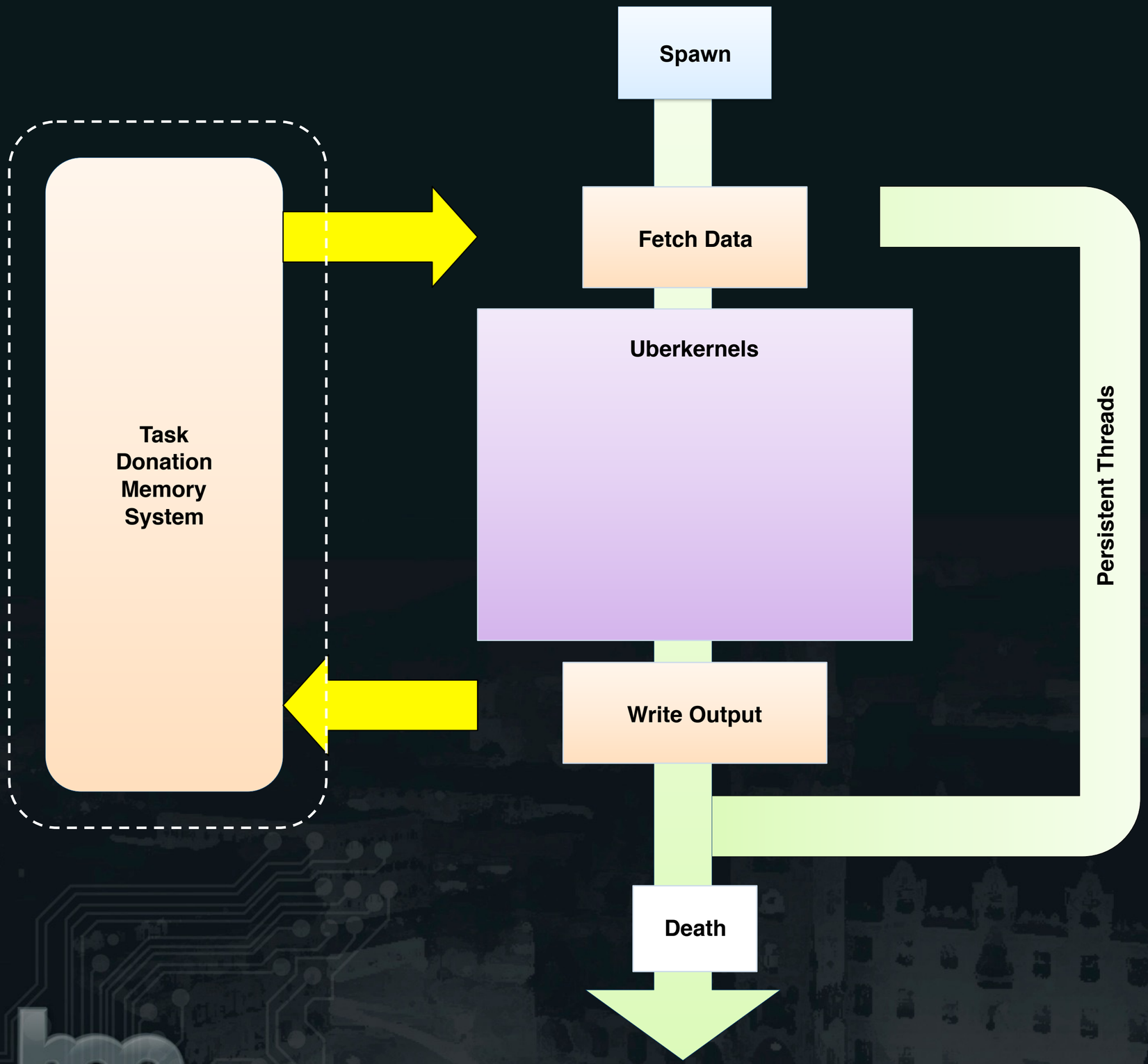
Our Version



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

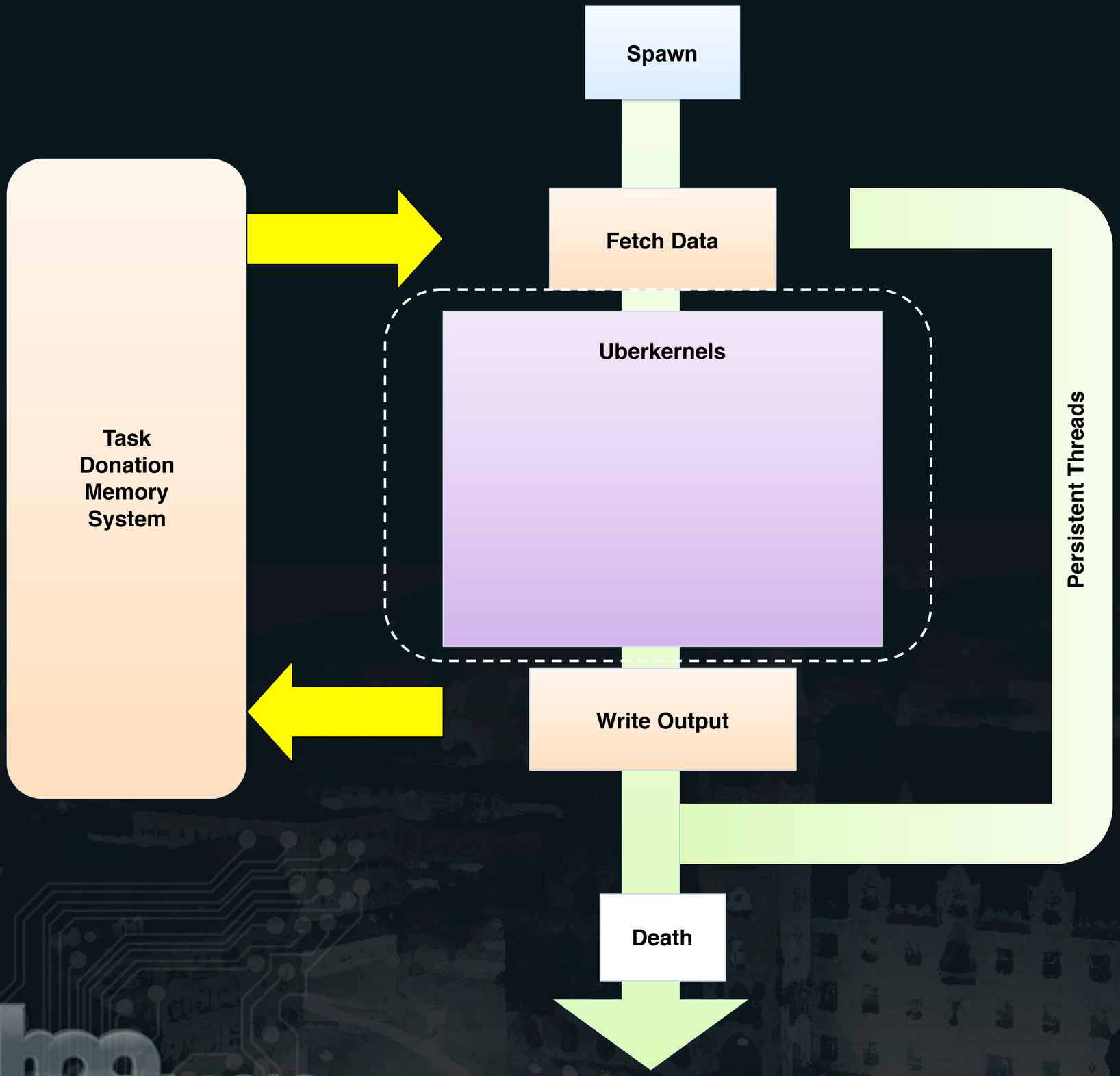
Our Version

Each processor grabs work to process.



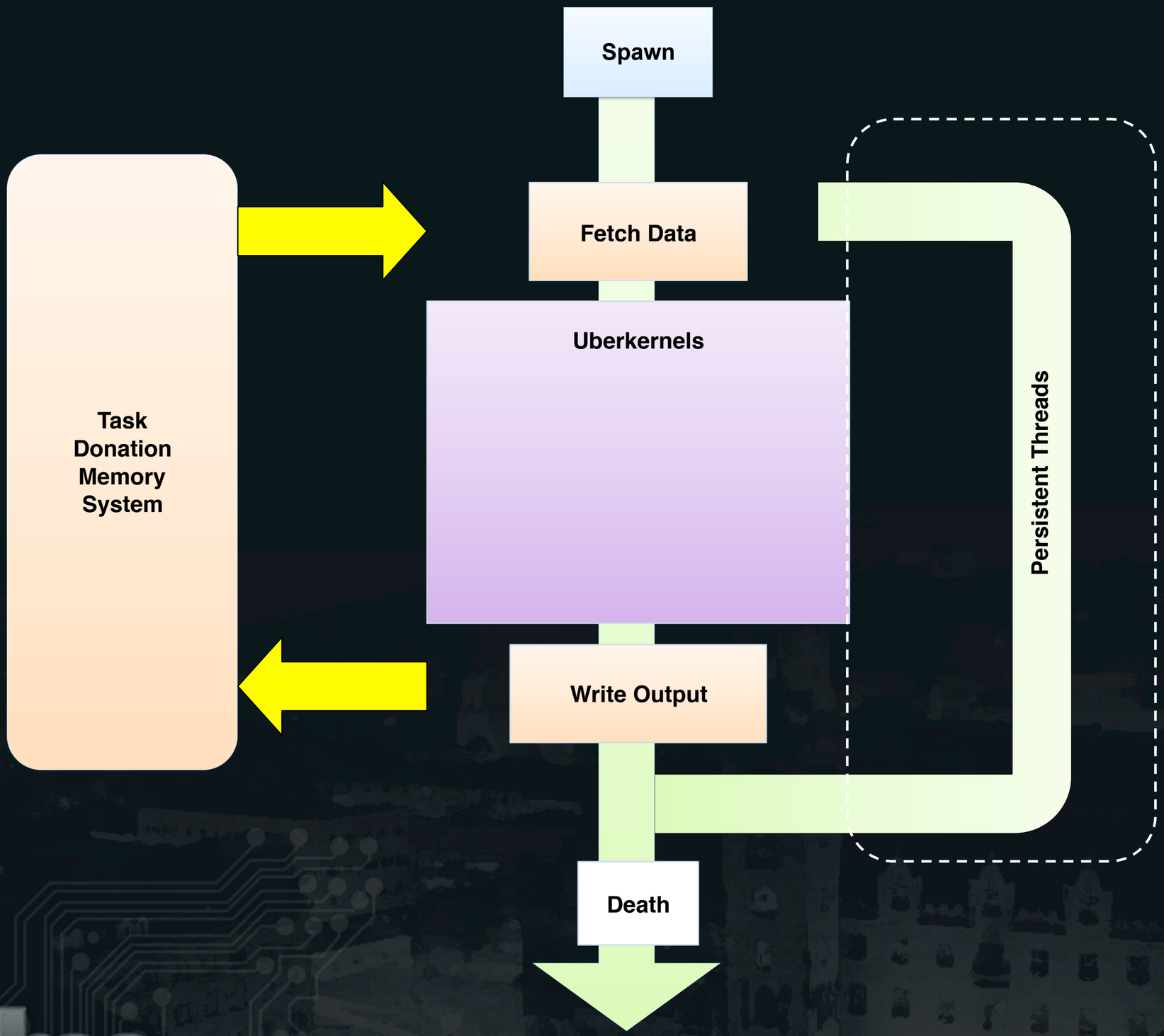
Our Version

Uberkernel decides how to process the current work unit



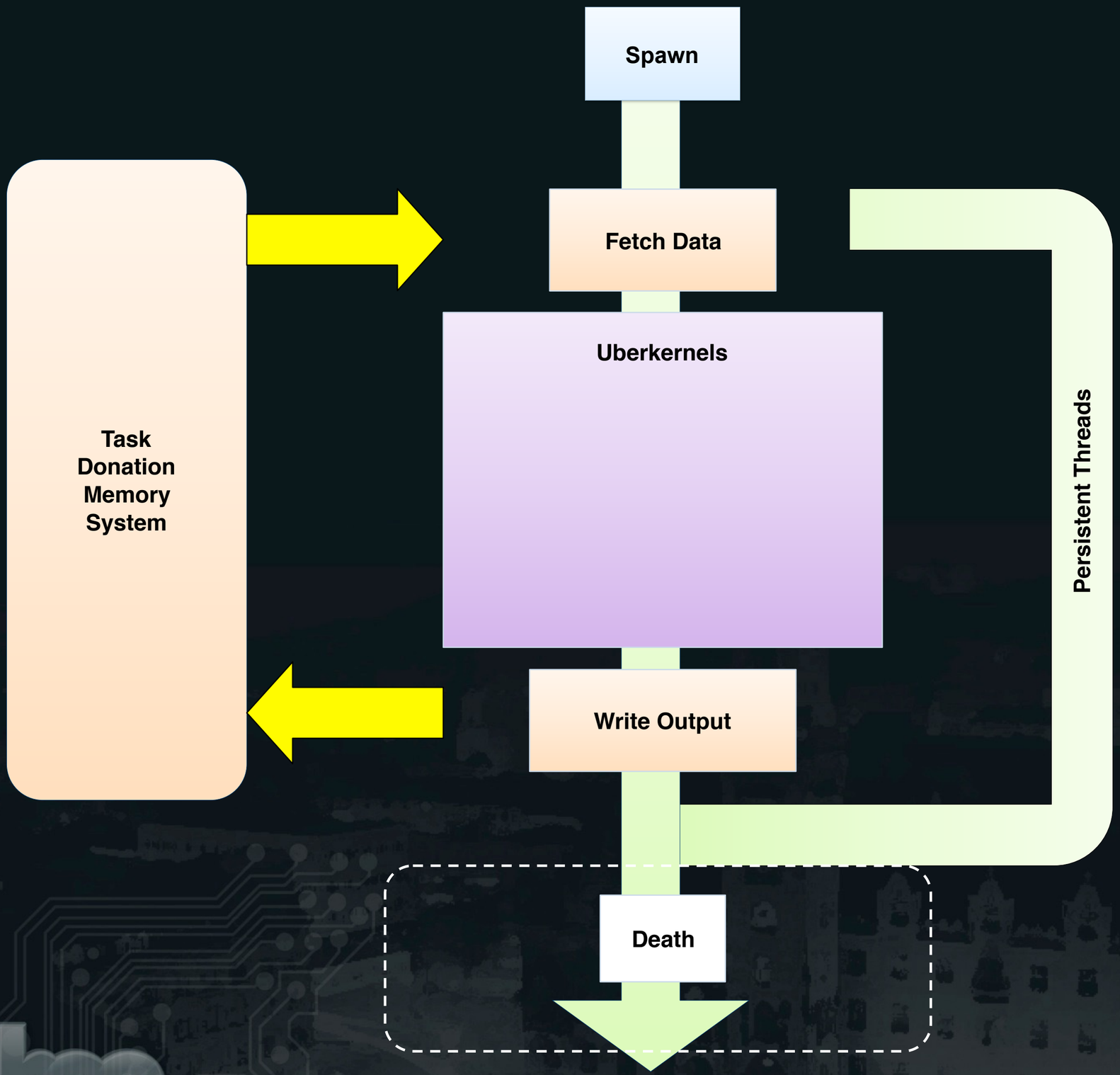
Our Version

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



Next Steps

- Split-and-dice has independent parallel tasks
  - Handling dependencies?
- Split-and-dice is only one stage
  - How do we string together multiple irregular stages?
- Lots of interesting scheduling decisions …
  - Producer-consumer locality
  - Batch sizes for stripmining
  - Data management of large static data
- APIs

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

Whippletree

M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg, "Whippletree: task-based scheduling of dynamic workloads on the GPU," TOG, vol. 33, no. 6, pp. 1–11, Nov. 2014.

UCD EEC 289Q, Winter 2018

Features for Dynamic Irregular Workloads

- "Multiprogramming": a MIMD model that allows simultaneous execution of multiple pipeline stages
- Dynamic work generation, with load balancing
- Locality (including producer-consumer)
- Support for different kinds of parallelism (one thread per element vs. lots of threads per element)

Whippletree Programming Model

- Work-items: data elements to be processed
- Procedures: Process work-items
  - Must specify number of threads needed to process work-item
- Tasks: pairing of work-item with procedure. Tasks can spawn new tasks.
  - Whippletree schedules these
- Programs: Closed sets of procedures

Technologies

- **Persistent threads**
- **Uberkernels (megakernels)**
- **Time-sliced kernels**
- **Dynamic parallelism**

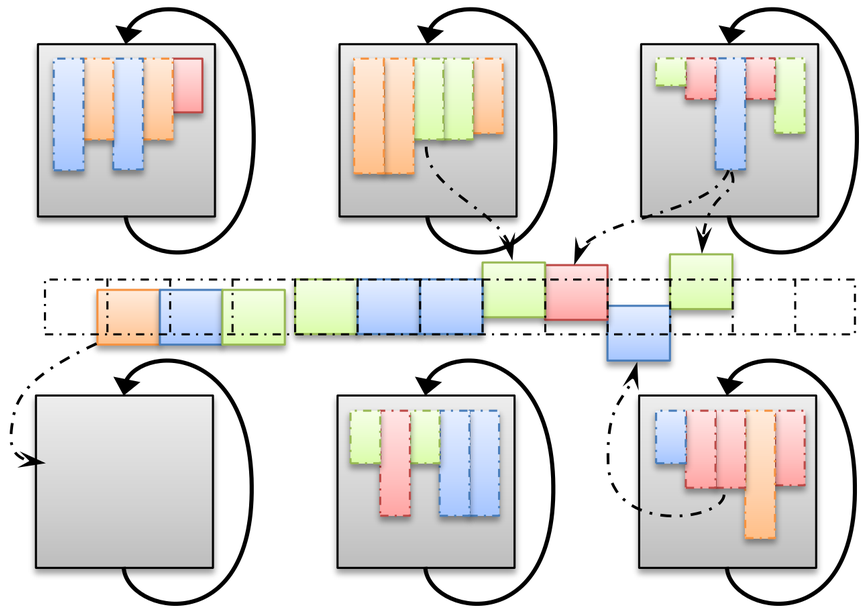


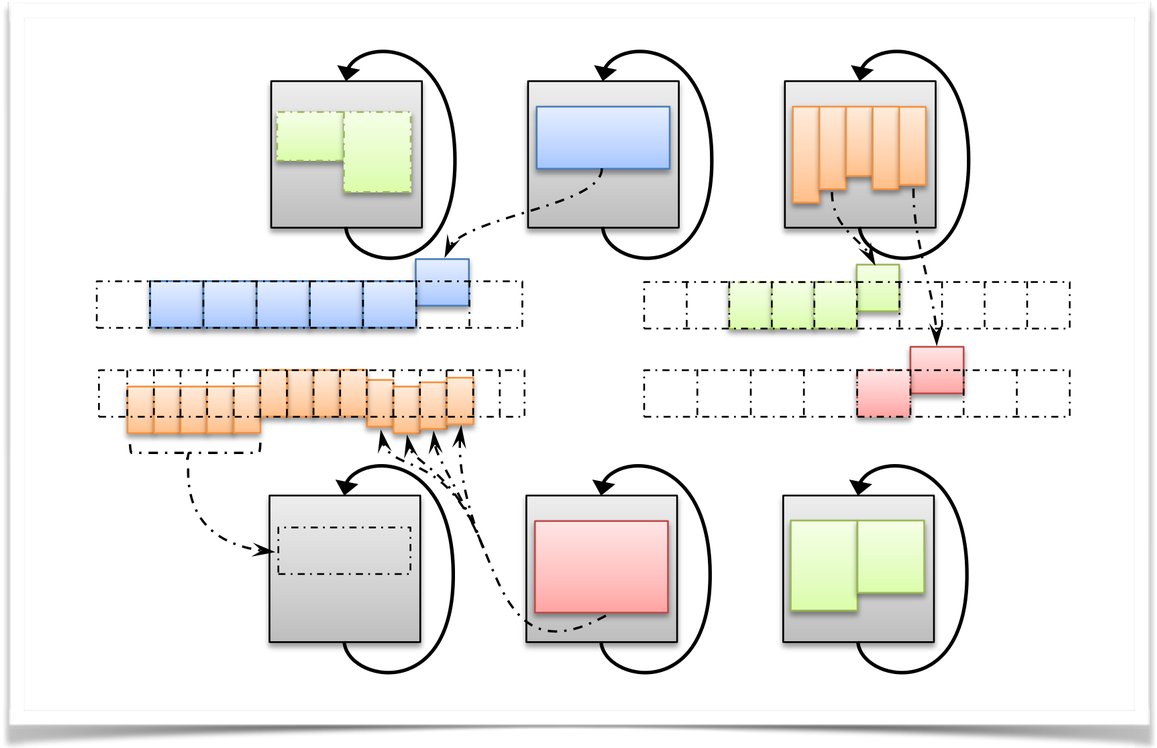
Figure 2: *A persistent megakernel model combines scheduling from a queue implemented in software with persistent worker threads running in a loop until the queue is empty. During execution, new work can be placed in the queue.*

3 Task Types

- Level-0 tasks: must be executed by threads within a warp
  - Warp synchrony, warp-wide instructions, etc.
- Level-1 tasks: must be executed by threads in the same thread block
  - Shared memory, ___syncthreads(), etc.
- Level-2 tasks: single-threaded tasks
  - Arbitrarily assigned to any SM
  - Can be grouped to fill a warp

Whippletree Implementation

- Worker-blocks draw tasks from queues
- One queue per procedure
  - Minimizes divergence even in the presence of different task types
- New tasks can be created during execution



Whippletree Implementation

- Worker-blocks dynamically assign thread to incoming tasks
  - How do we make this decision? What would make sense here?
- Level-1 tasks can have individual barriers