# Synchronization on Manycore Machines

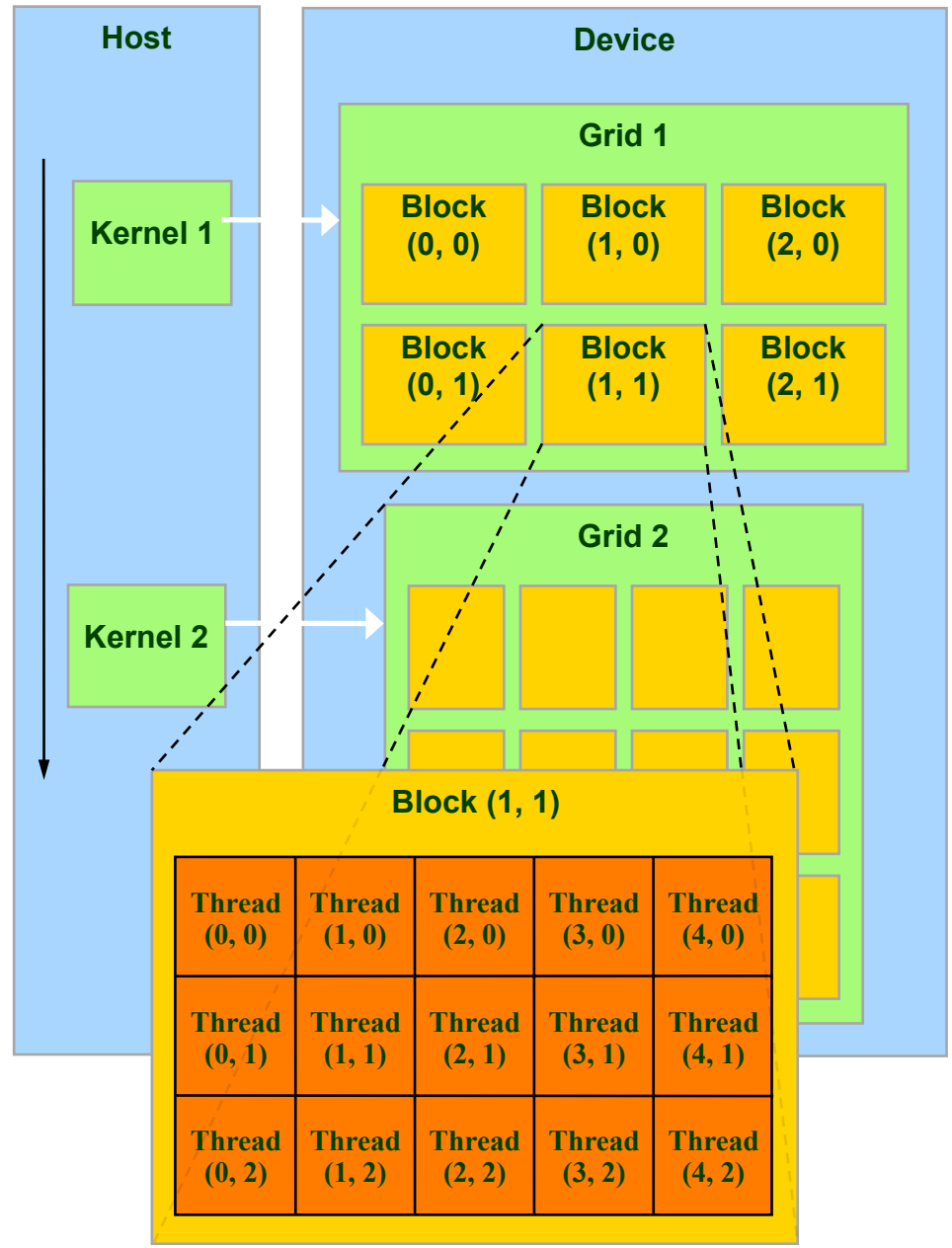
John Owens
Associate Professor, Electrical and Computer Engineering
University of California, Davis

# Announcements

- If anyone's going back to Boston near a T station immediately after the end of the conference on Friday, I'd love a ride. (Faster than the train alternative. I'm happy to get back ASAP.)
- Stuff I'm not talking about but might be interesting to some of you:
  - Tridiagonal solvers
  - bzip2-style lossless compression
  - Heterogeneous multi-node global-illumination rendering (substitute your hard heterogeneous problem here)

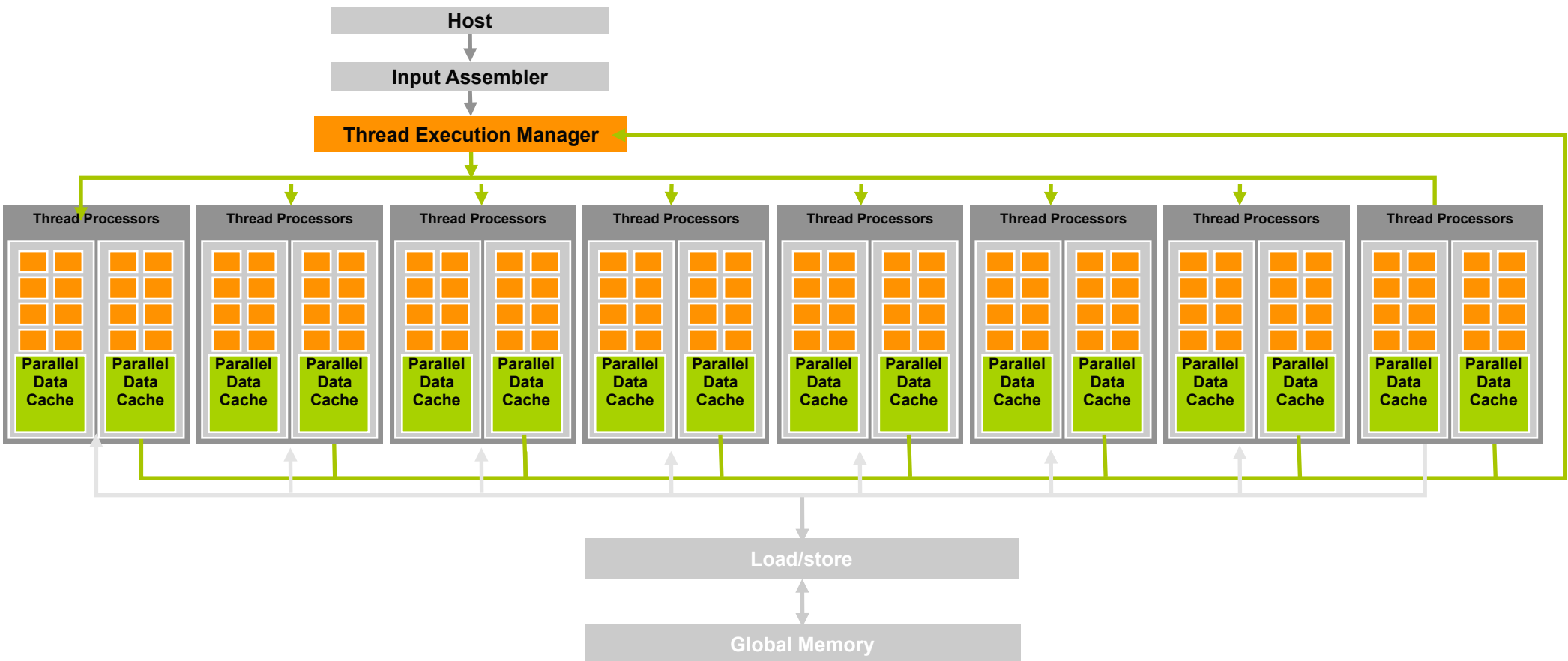
# GPU Programming Model

- A kernel is executed as a grid of thread blocks
- A thread block is a fixed-maximum-size (~512) batch of threads that can cooperate with each other by:
  - Efficiently sharing data through shared memory
  - Synchronizing their execution
- Two threads from two different blocks cannot cooperate
  - Blocks are *independent*

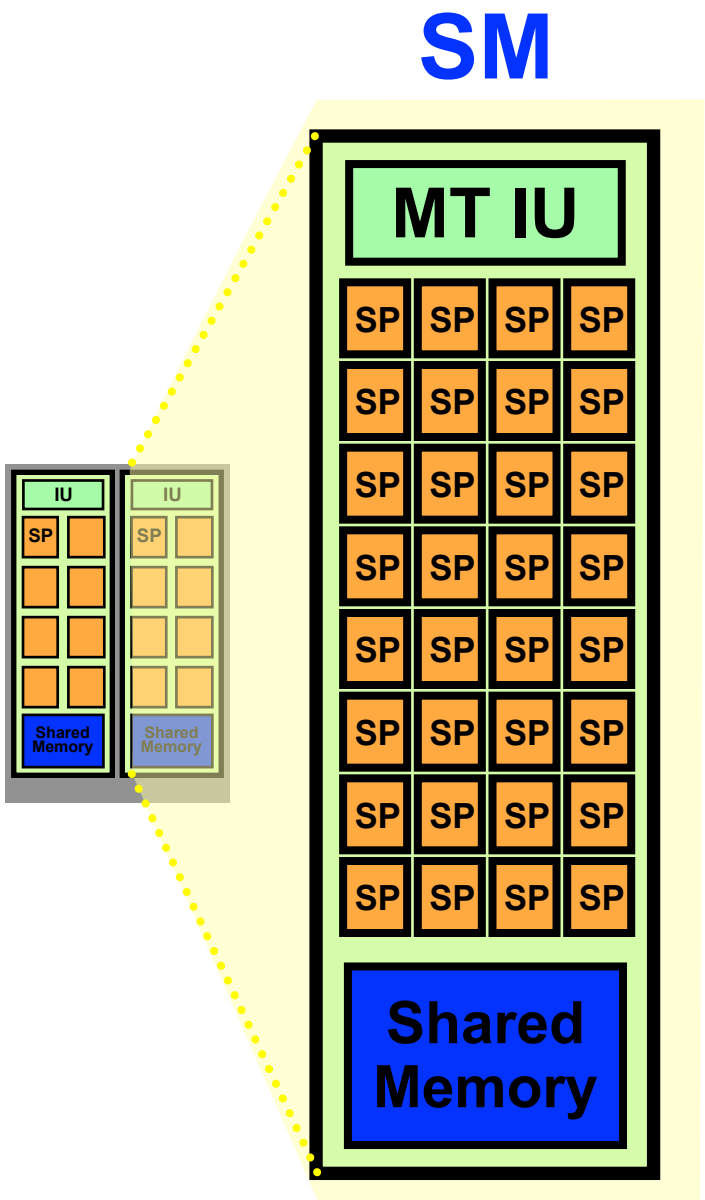


# GPU Hardware, High Level

- Hardware responsible for assigning blocks to “SMs” (“streaming multiprocessors” or “cores”—think of them as virtual blocks).
- Different GPUs have different numbers of SMs.

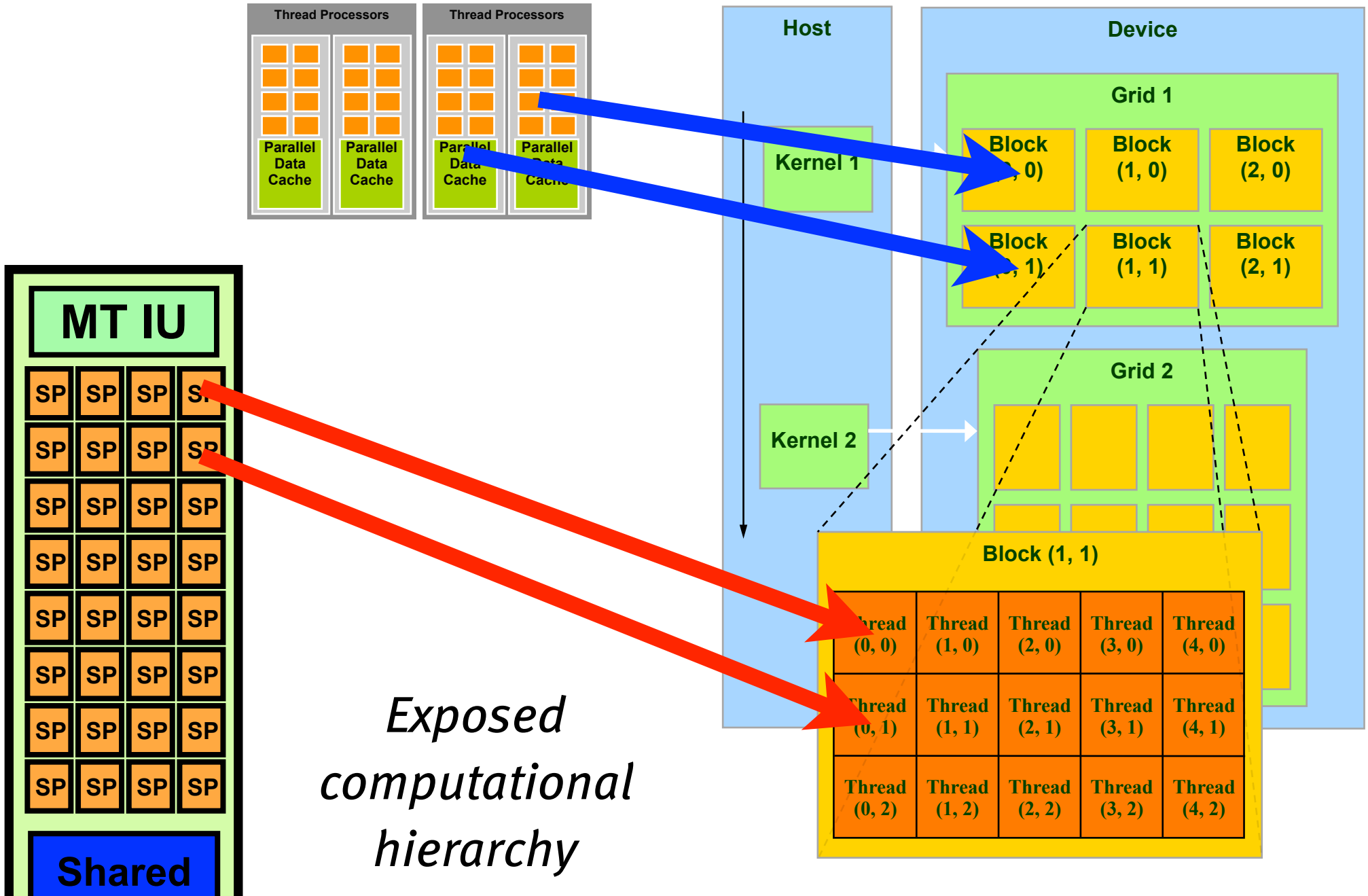


# SM Multithreaded Multiprocessor



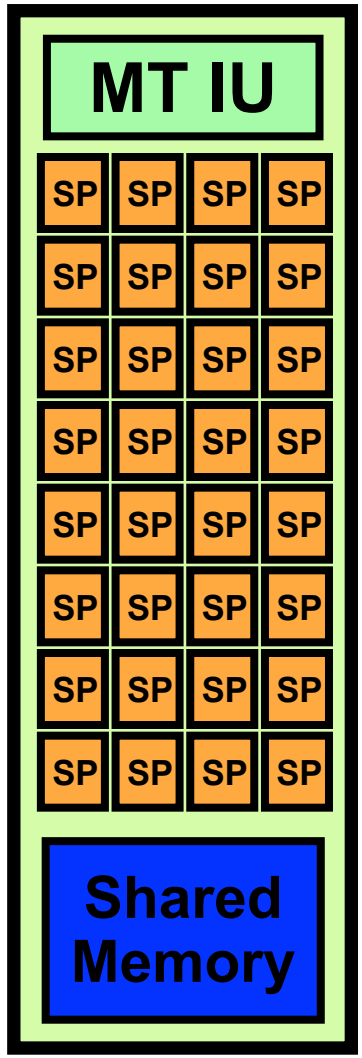

- **Each SM runs a *block* of threads**
- SM has 32 SP Thread Processors
  - Run as a “warp” in lockstep
  - 99 GFLOPS peak x 16 SMs at 1.544 GHz (1 MAD/clock/SP)
  - IEEE 754 32-bit floating point
- Scalar ISA
- Up to 768 threads, hw multithreaded
- 16 or 48 KB shared memory, 48 or 16 KB hardware-managed cache

# Mapping SW to HW



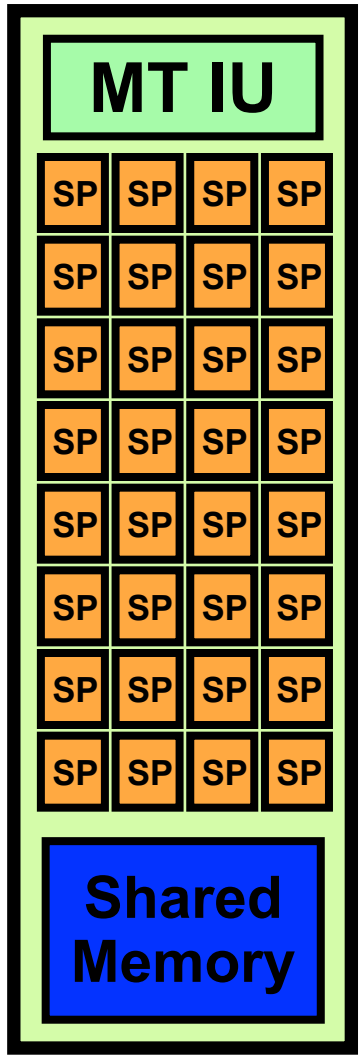

*Exposed computational hierarchy*

# Synchronization Toolbox (1)



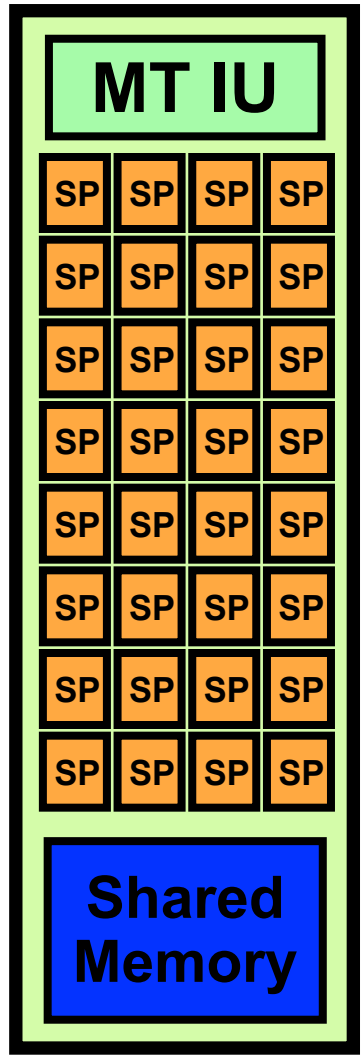

- Within a thread block & within a warp:
  - In hardware, warps run synchronously
  - Hardware manages branch divergence (idle threads go to sleep)
  - The width of a warp is only vaguely exposed by the programming model
    - Different for different vendors (Intel: 16, NVIDIA: 32, AMD: 64)
  - Warps have *_all*, *_any*, *_ballot* hw intra-warp functions

# Synchronization Toolbox (2)




- Within a thread block & across warps:
  - *_syncthreads* is a barrier for threads within a warp
    - No need to synchronize between threads within warp
    - Newest NVIDIA GPUs add *_syncthreads_count(p)*, *_syncthreads_or(p)*, *_syncthreads_and(p)* for predicate *p*
  - *_threadfence_block:* all memory accesses are visible to all threads within block
  - *_threadfence:* all memory accesses visible to all threads on GPU
  - *_threadfence_system:* all memory accesses visible to threads on GPU and also CPU

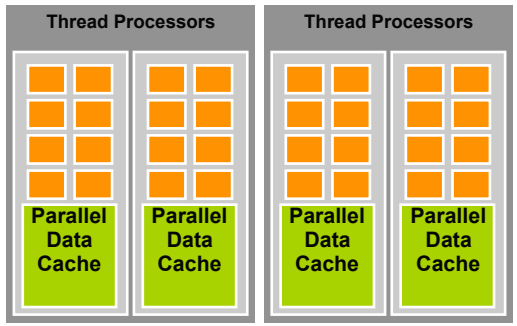
# Synchronization Toolbox (3)




- Threads within a block can read/write shared memory
  - Best approximation of shared-memory model is CREW: concurrent reads, exclusive write
    - Hardware makes no guarantees about who will win if concurrent writes
  - Memory accesses can be guaranteed to compile into actual read/write with *volatile* qualifier
  - Atomics on shared memory: 32b, 64b ints; 32b float for *exch* and *add*
  - *add*, *sub*, *exch*, *min*, *max*, *inc*, *dec*, *CAS*, bitwise {*and*, *or*, *xor*}

# Synchronization Toolbox (4)

- Threads within a block can read/write global memory
  - Same atomics as shared memory
  - Memory accesses can be guaranteed to compile into actual read/write with *volatile* qualifier
    - Fermi has per-block L1 cache and global L2 cache
    - On Fermi, *volatile* means "bypass L1 cache"
- Implicit global-memory barrier between *dependent kernels*




*Volkov & Demmel (SC '08): synchronous kernel invocation: 10–14 µs, asynchronous: 3–7*

```
vec_minus<<<nblocks, blksize>>>(a, b, c);
vec_dot<<<nblocks, blksize>>>(c, c);
```

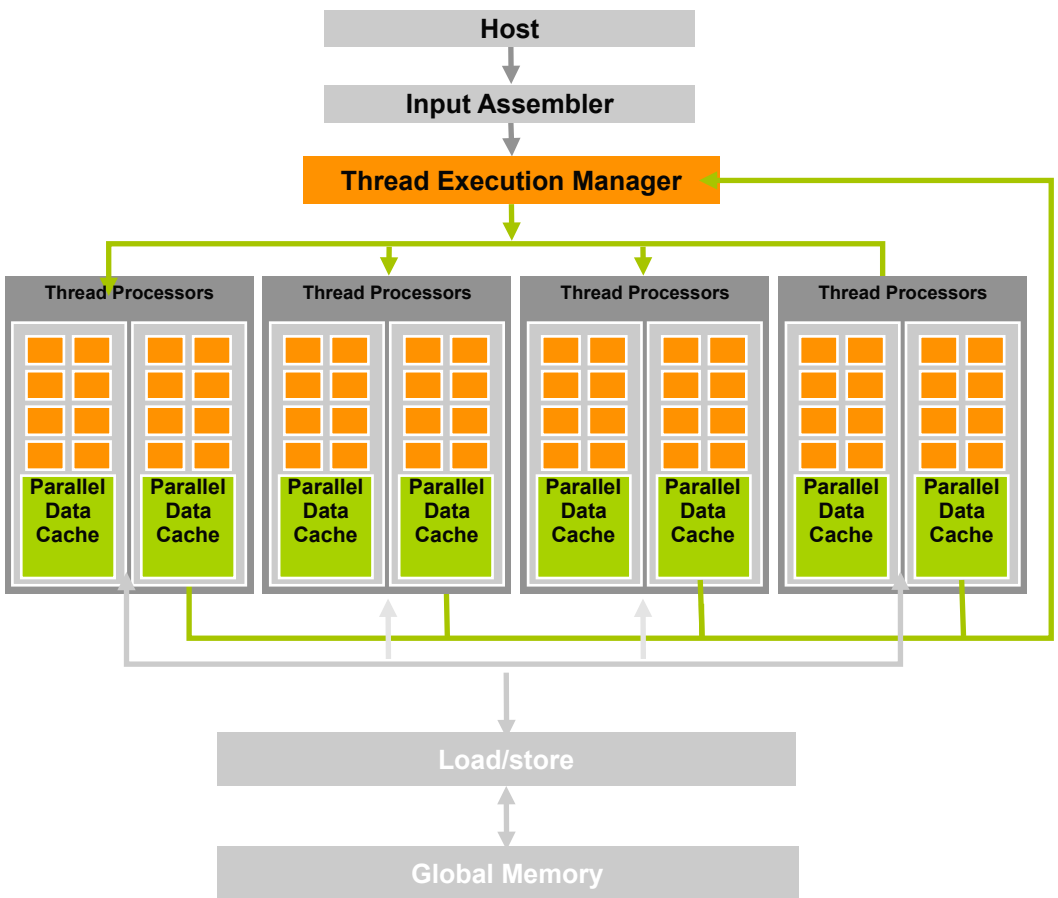
- ***No other synchronization instructions! Why? Let's pop up a level and talk about CUDA's goals.***

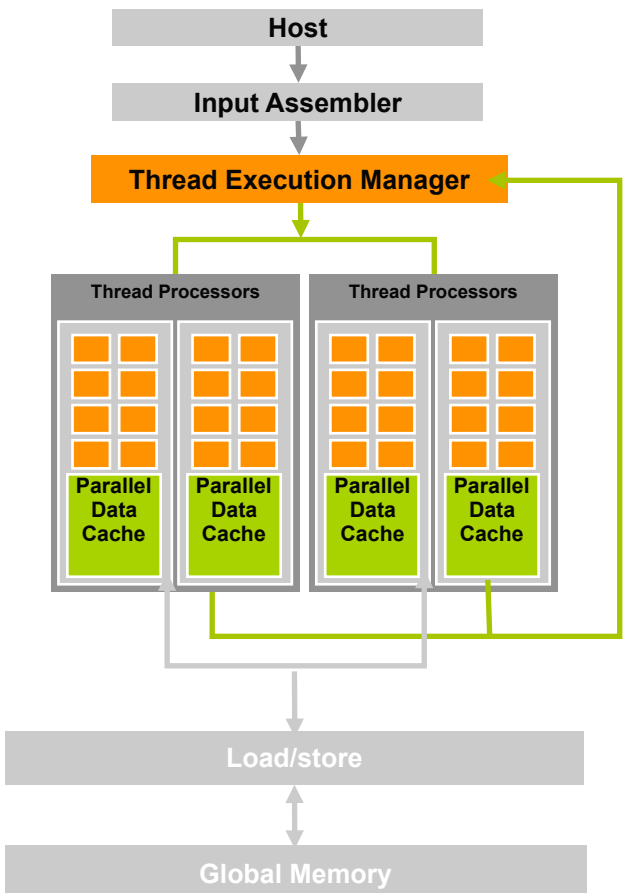
# Big Ideas in the GPU Model

1. One thread maps to one data element (lots of threads!)
2. Write programs as if they run on one thread
3. CPUs *mitigate* latency. GPUs *hide* latency by switching to another piece of work.
4. **Blocks within a kernel are *independent***

# Scaling the Architecture

- Same program runs on both GPUs
- Scalable performance!



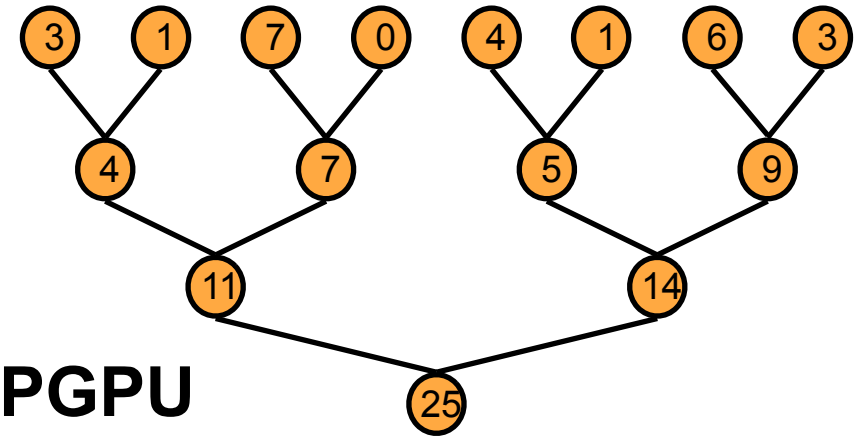
# Consequences of Independence

- *Any* possible interleaving of blocks must be valid
  - Blocks presumed to run to completion without preemption
  - Can run in any order
  - Can run concurrently OR sequentially
- Therefore, blocks may *coordinate* but not *synchronize* or *communicate*
  - Can’t have a global barrier: blocks running to completion may block other blocks from launching
  - Can’t ask block A to wait for block B to do something, or for B to send to A: A might launch before B

# Outline

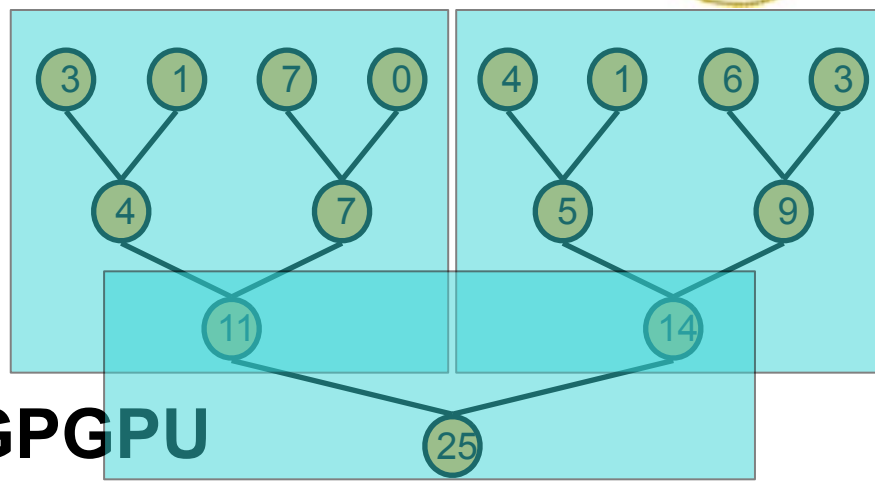
- Persistent threads
- Persistent thread global barriers
- Spin-locks for shared resources
- Higher-order (and better) synchronization primitives
- Hardware biases (permutation)
- Work queues

# Tree-Based Parallel Reductions




- Commonly done in traditional GPGPU
  - Ping-pong between render targets, reduce by 1/2 at a time
  - Completely bandwidth bound using graphics API
  - Memory writes and reads are off-chip, no reuse of intermediate sums
- CUDA solves this by exposing on-chip shared memory
  - Reduce blocks of data in shared memory to save bandwidth

# Tree-Based Parallel Reductions

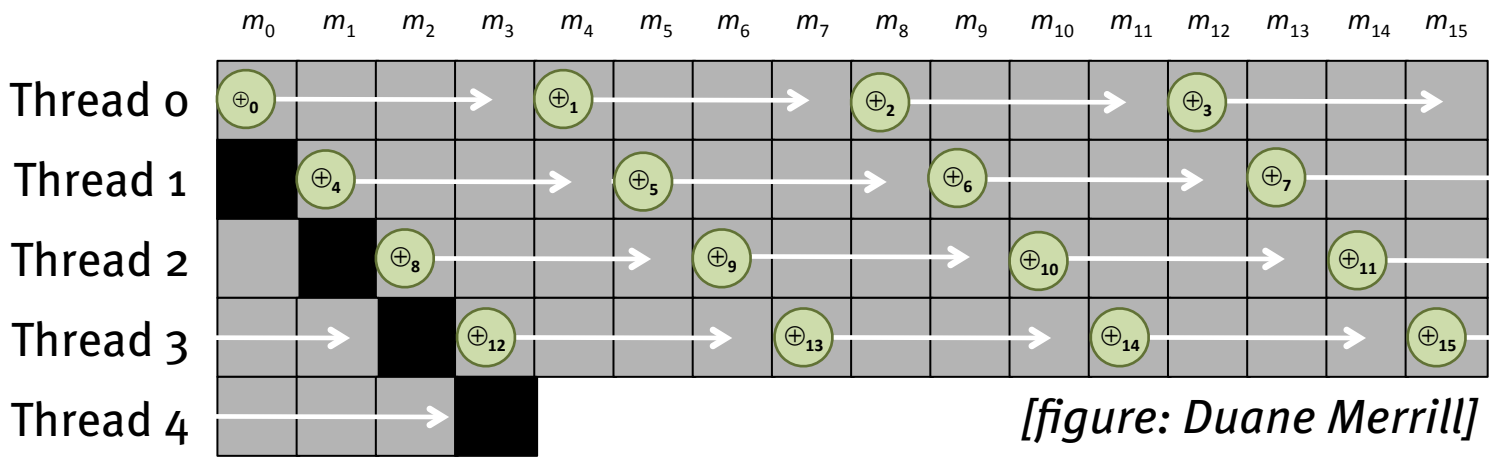



- Commonly done in traditional GPGPU
  - Ping-pong between render targets, reduce by 1/2 at a time
  - Completely bandwidth bound using graphics API
  - Memory writes and reads are off-chip, no reuse of intermediate sums
- CUDA solves this by exposing on-chip shared memory
  - Reduce blocks of data in shared memory to save bandwidth

# Traditional reductions

- Ideal: *n* reads, 1 write.
- Block size 256 threads. Thus:
  - Read *n* items, write back *n*/256 items. (Kernel 1)
  - Implicit synchronization between kernels, and possibly round-trip communication (400 µs) to CPU to launch second kernel.
  - Read *n*/256 items, write back 1 item. If too big for one block, recurse. (Kernel 2)
    - Or could sum using an atomic add, but we'll ignore that for the moment.

# Persistent Threads




*[figure: Duane Merrill]*

- GPU programming model suggests one thread per item
- What if you filled the machine with just enough blocks to keep all processors busy, then asked each thread to stay alive until the input was complete?
  - More like a traditional CPU program
  - Essentially replaces hardware scheduler with software
- Reduction example: now intermediate results are *O*(number of SMs) rather than *O*(input size)

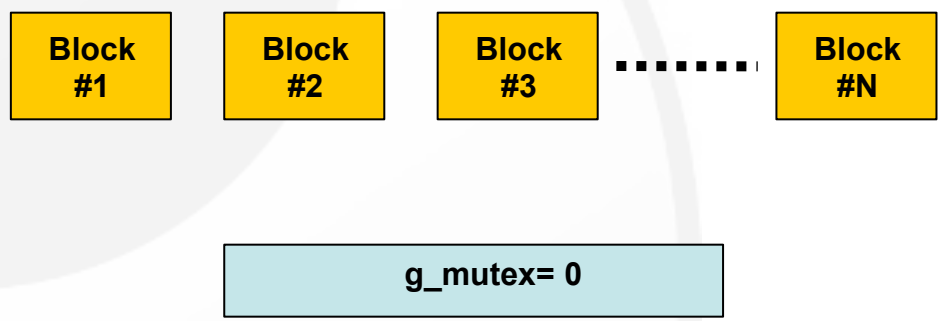
# Persistent Threads

*Recent work in our group. In submission.*

- Use cases:
  - Avoid CPU-GPU round-trip synchronization
  - Load-balancing: PT can use a software queue to (re)distribute irregularly-{produced,consumed} work
  - Producer-consumer locality within kernel
  - Cheaper global synchronization (next slide)
- Minus: More overhead per thread (register pressure)
- Minus: Violent anger of vendors

*[joint work with Kshitij Gupta and Jeff Stuart]*

# GPU Lock-Based Synchronization



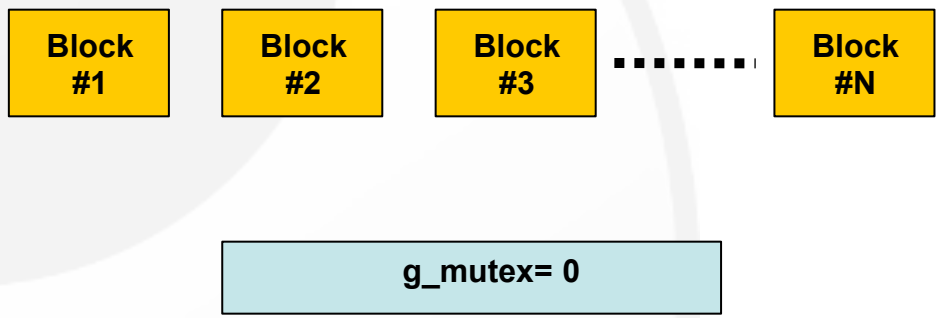





*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

# GPU Lock-Based Synchronization

| Algorithms | FFT | Smith-Waterman | Bitonic sort |
|---|---|---|---|
| % of time spent on inter-thread communication | 17.8% | 49.2% | 59.6% |

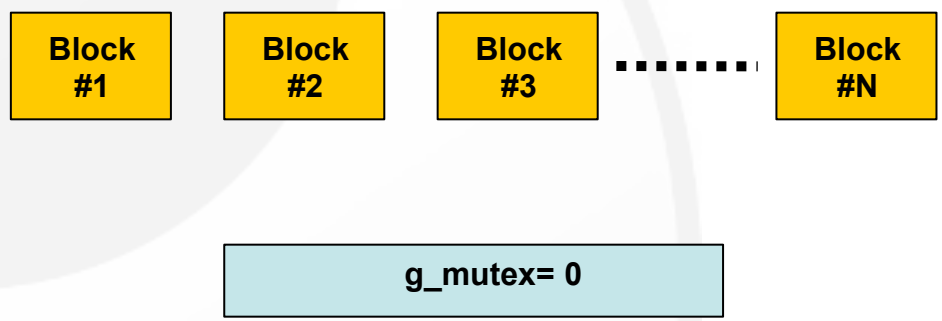







*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

# GPU Lock-Based Synchronization

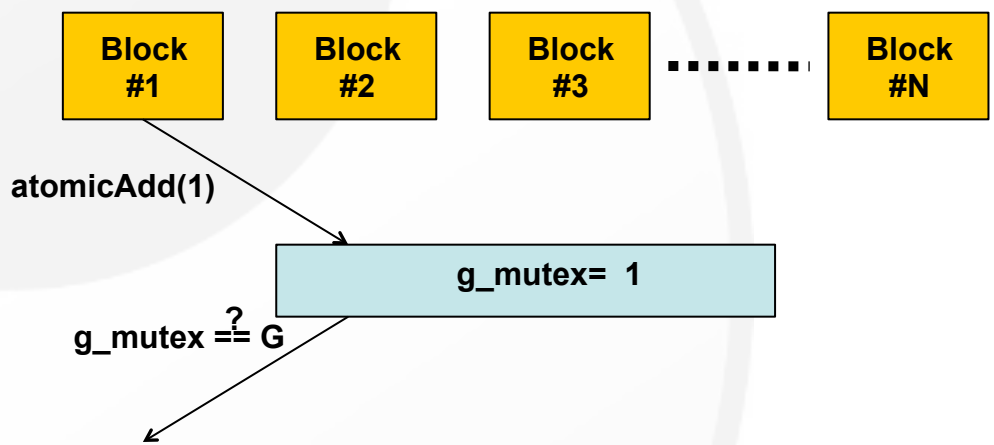







*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

# GPU Lock-Based Synchronization

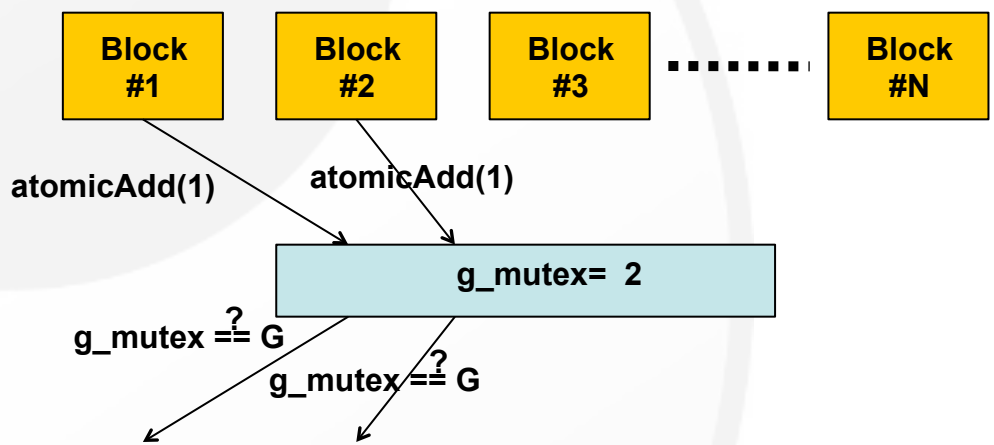







*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

# GPU Lock-Based Synchronization

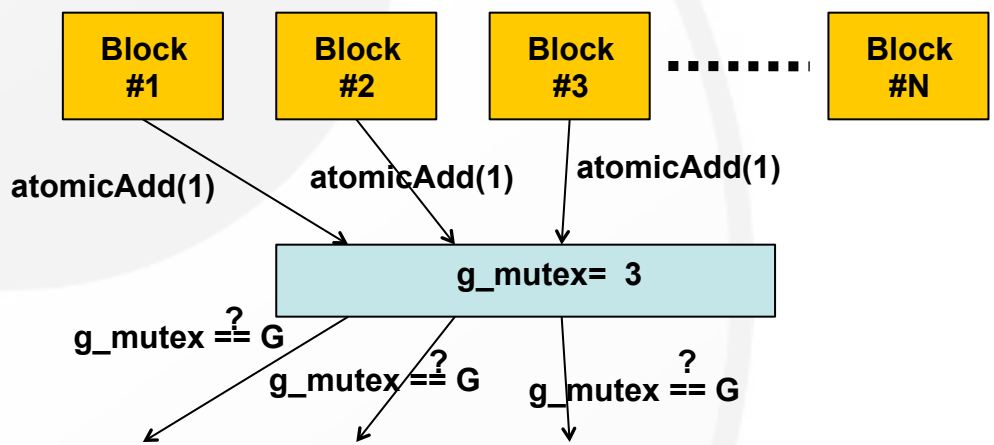







*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

# GPU Lock-Based Synchronization

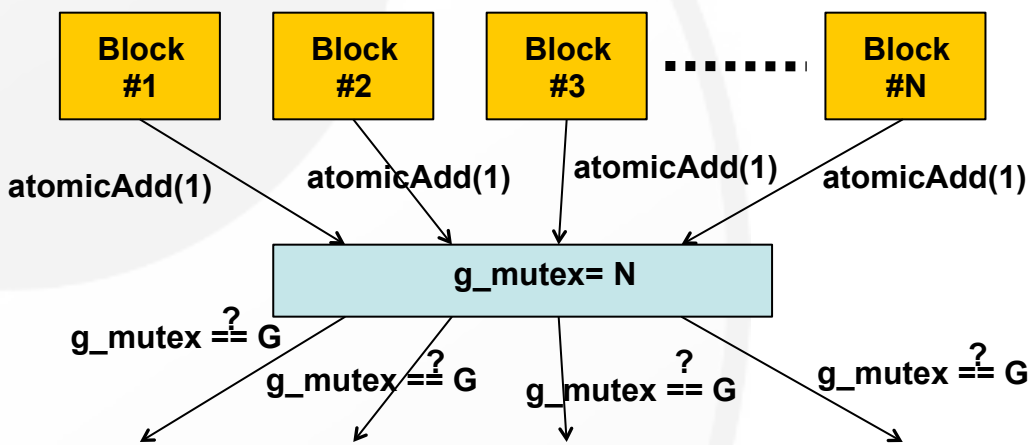







*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

# GPU Lock-Based Synchronization

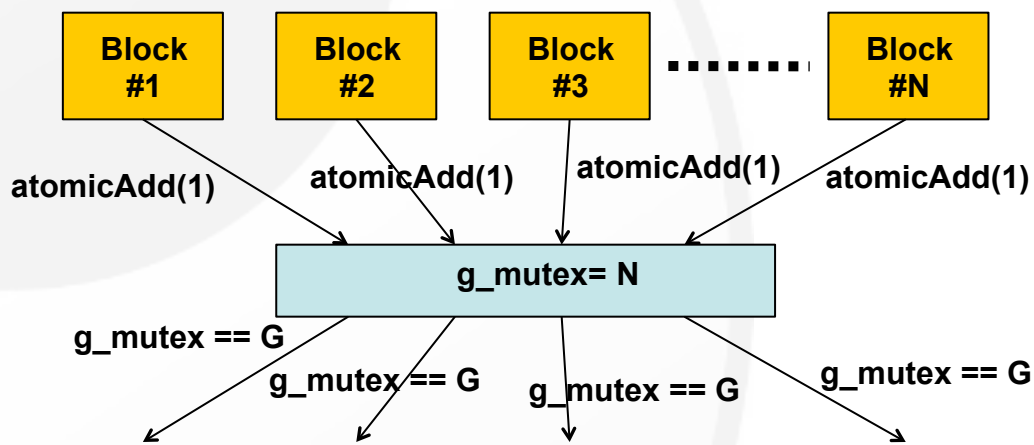







*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

# GPU Lock-Based Synchronization

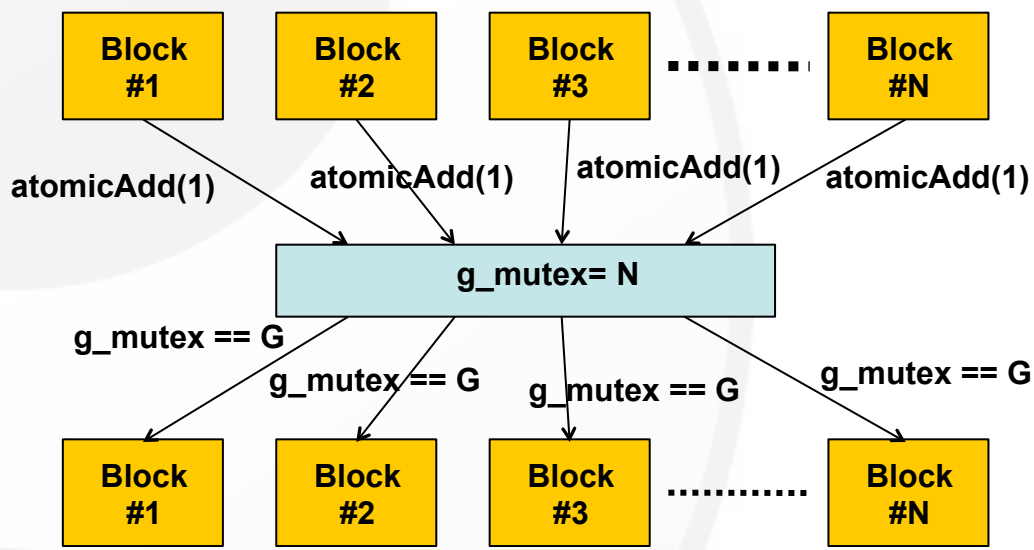







*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

# GPU Lock-Based Synchronization

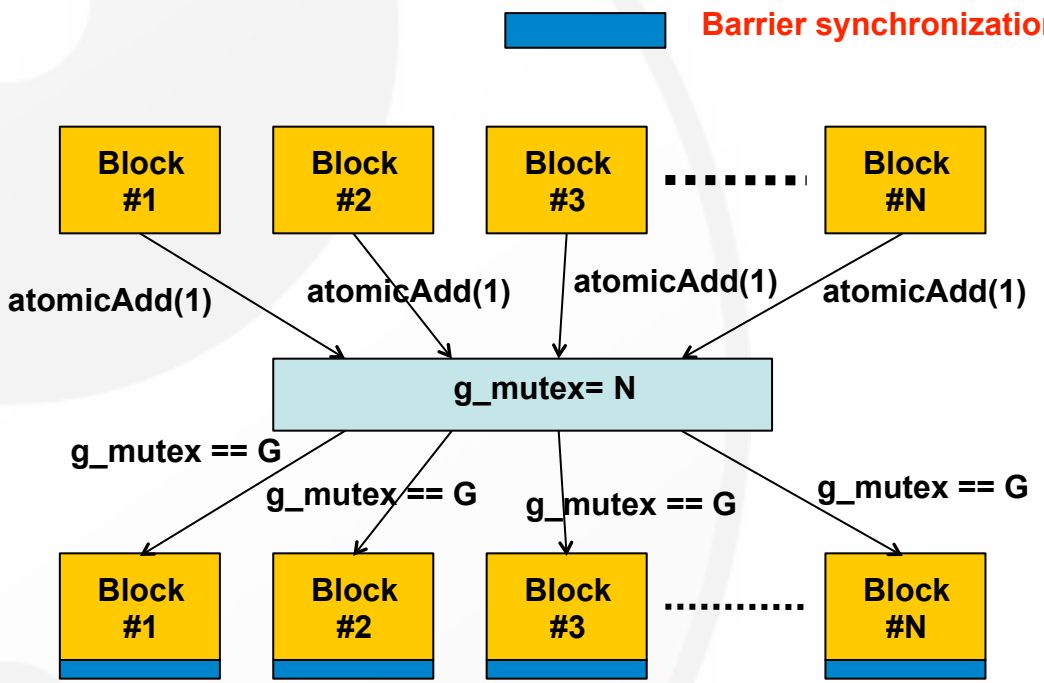







*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

# GPU Lock-Based Synchronization



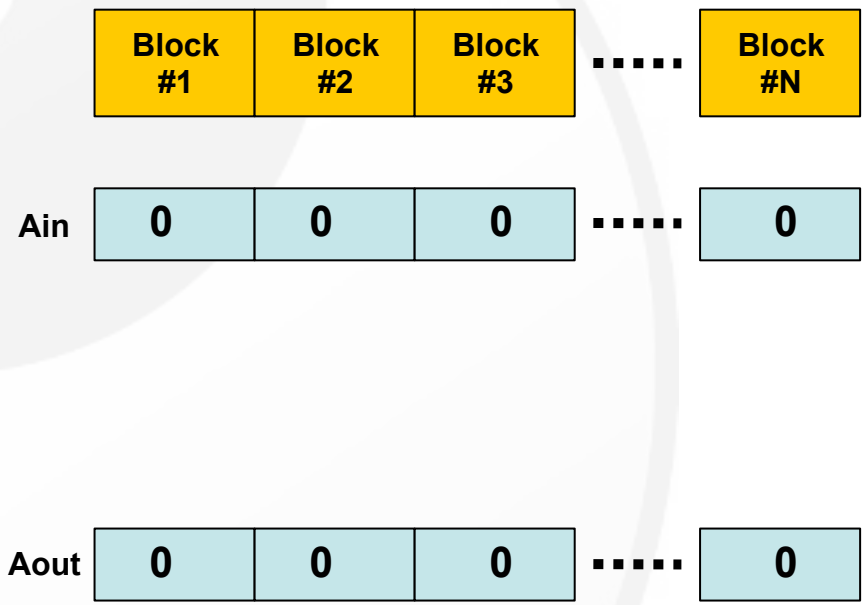





*S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In IPDPS, Apr. 2010.*

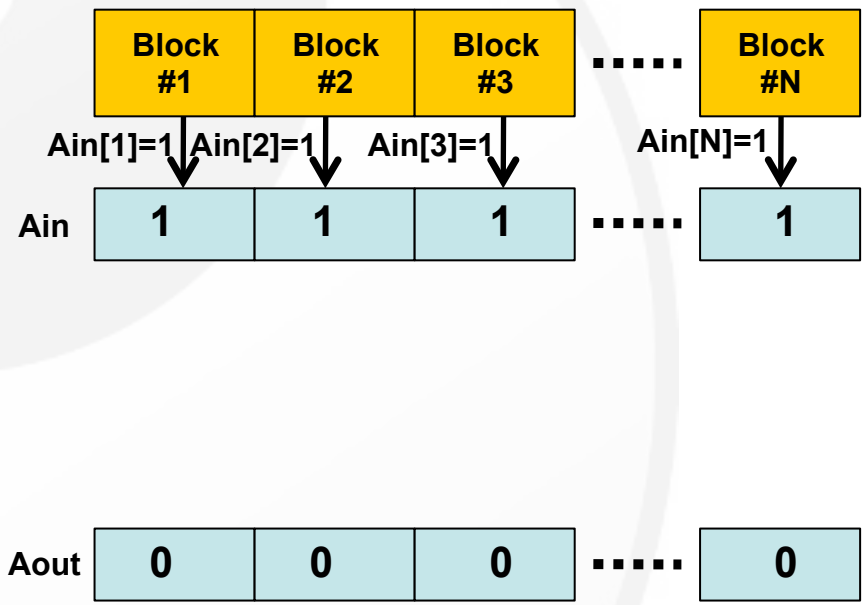
# GPU Lock-Free Synchronization

- Implementation



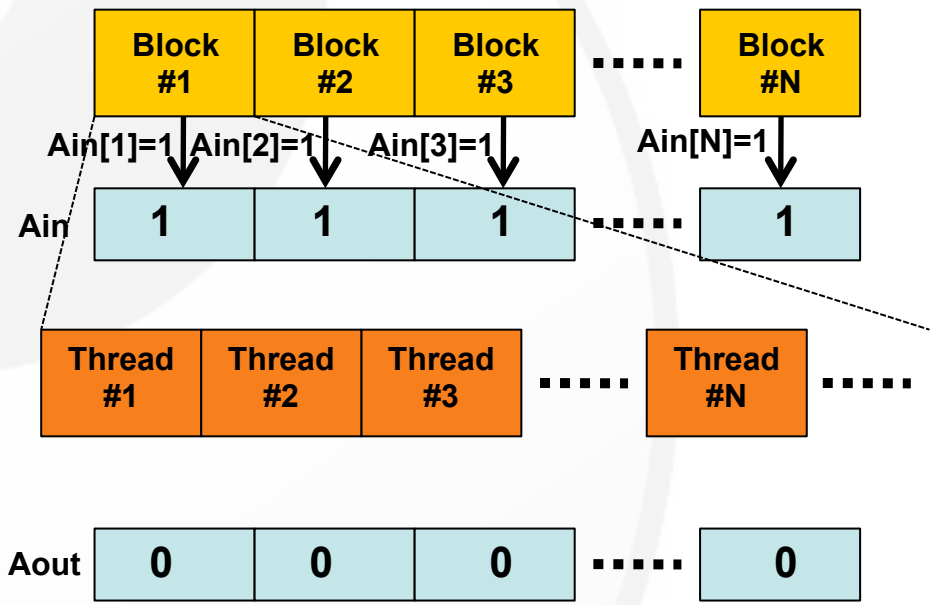
# GPU Lock-Free Synchronization

- Implementation



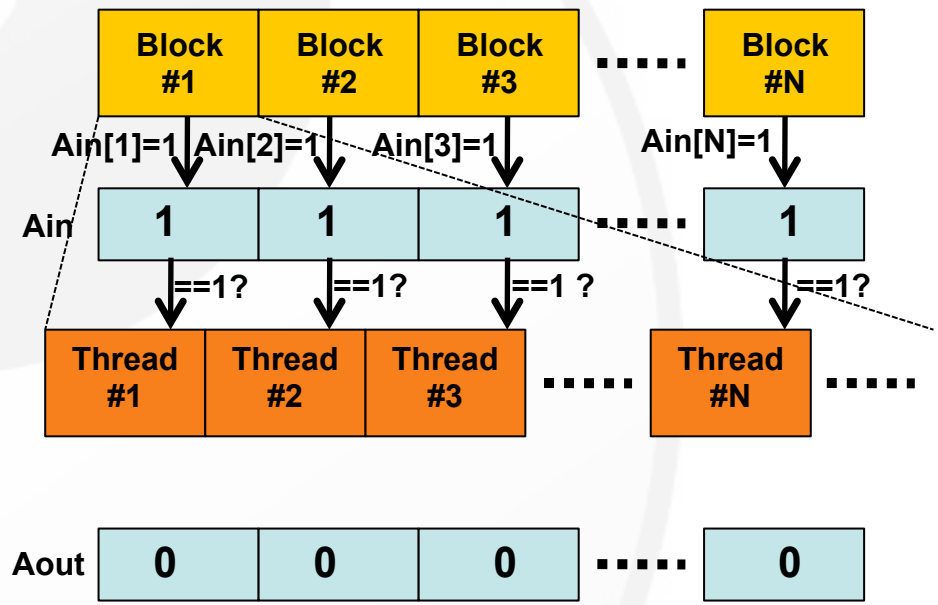
# GPU Lock-Free Synchronization

- Implementation



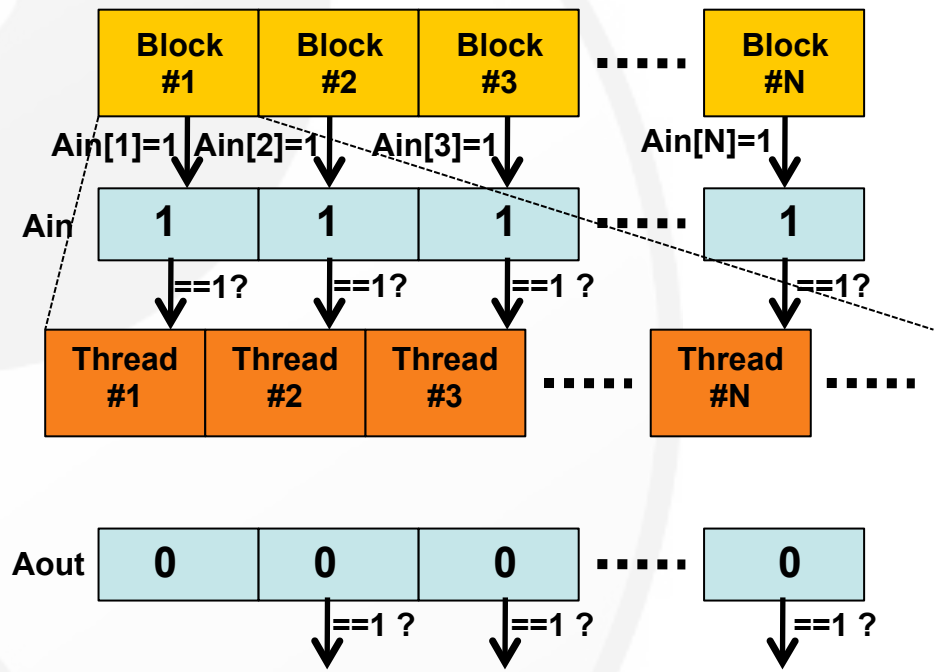
# GPU Lock-Free Synchronization

- Implementation



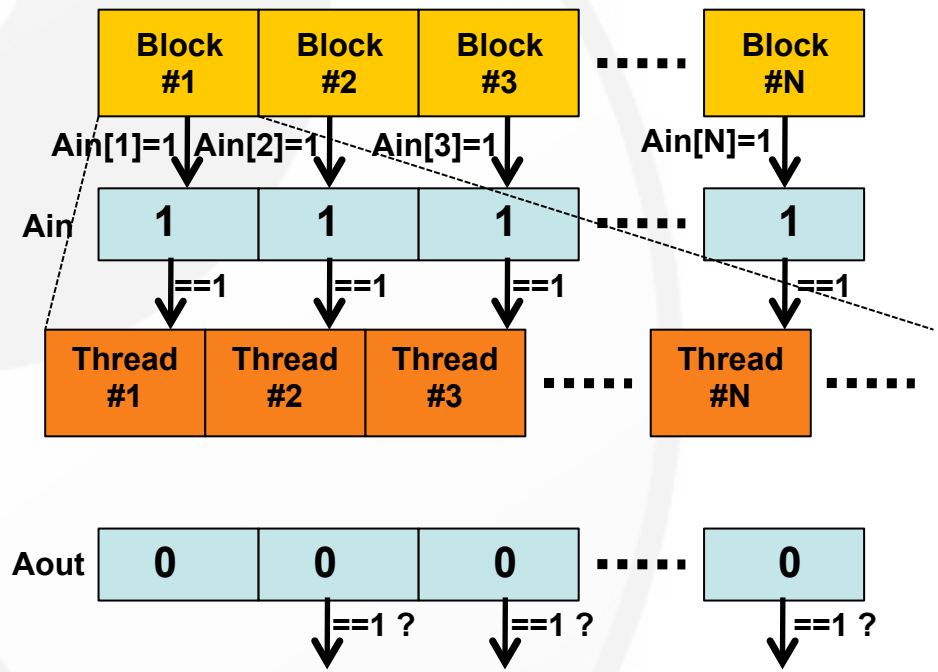
# GPU Lock-Free Synchronization

- Implementation



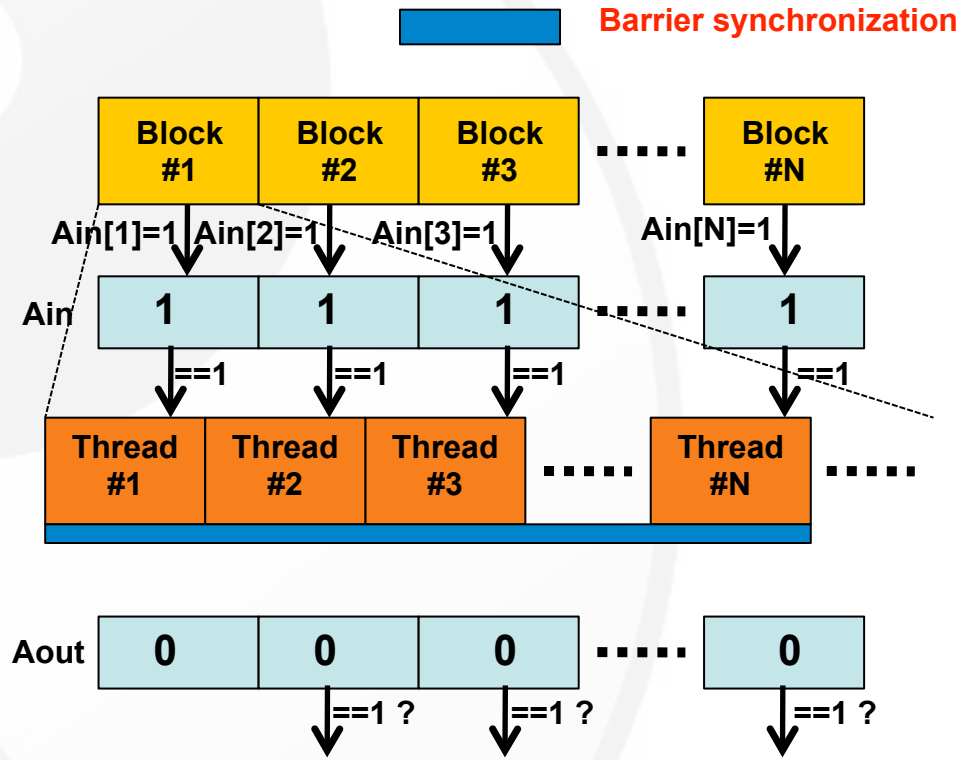
# GPU Lock-Free Synchronization

- Implementation



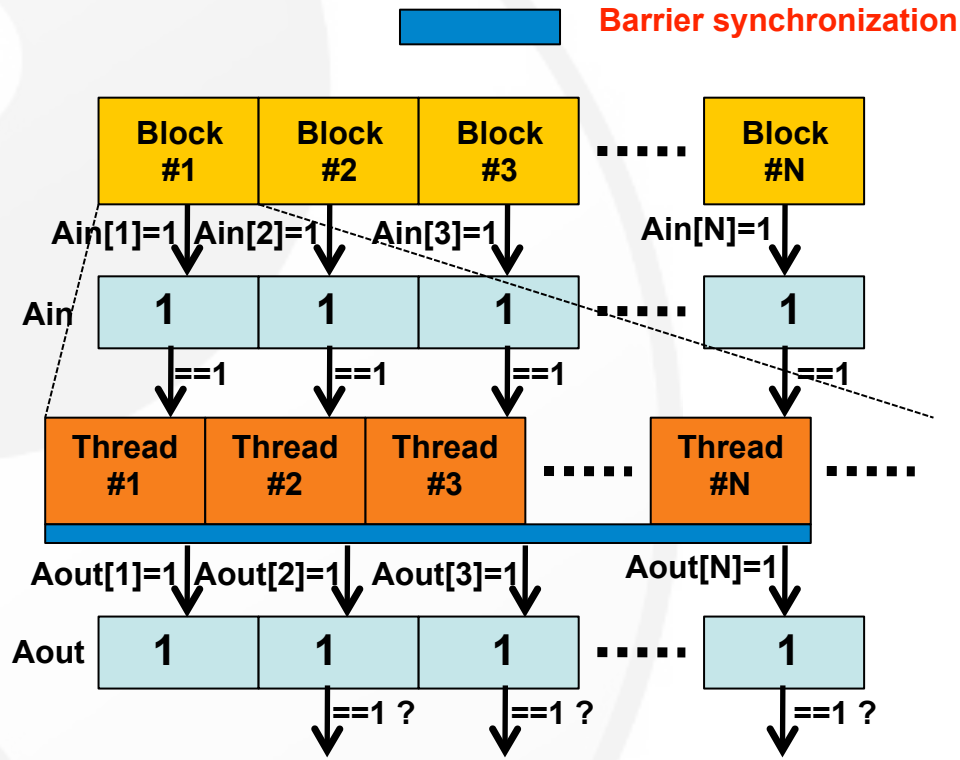
# GPU Lock-Free Synchronization

- Implementation



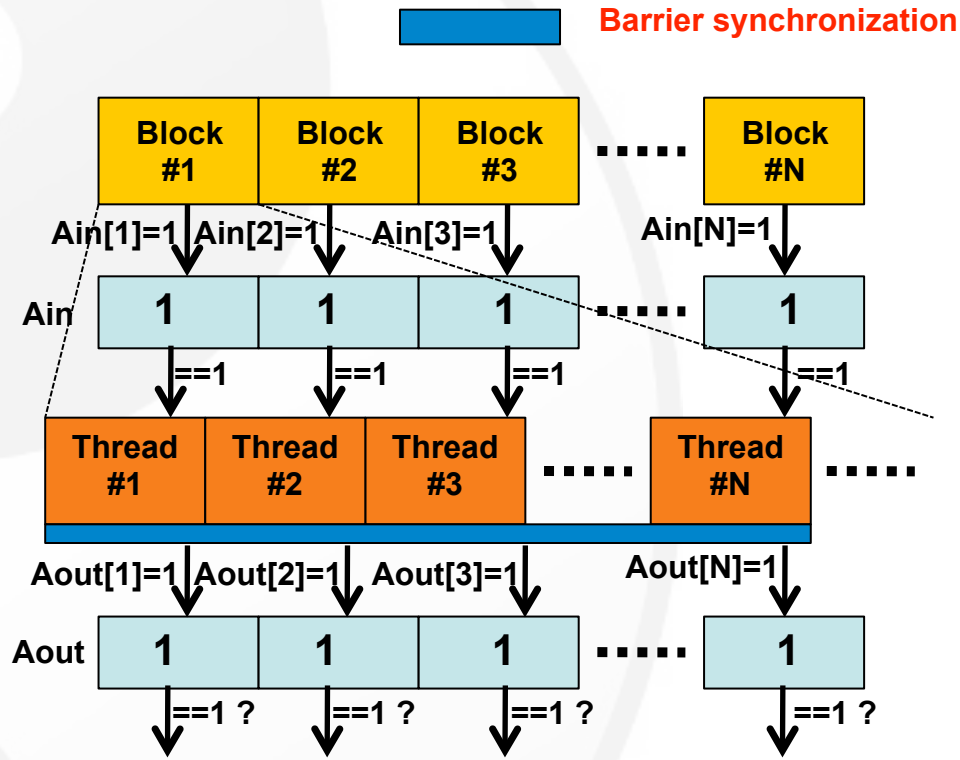
# GPU Lock-Free Synchronization

- Implementation



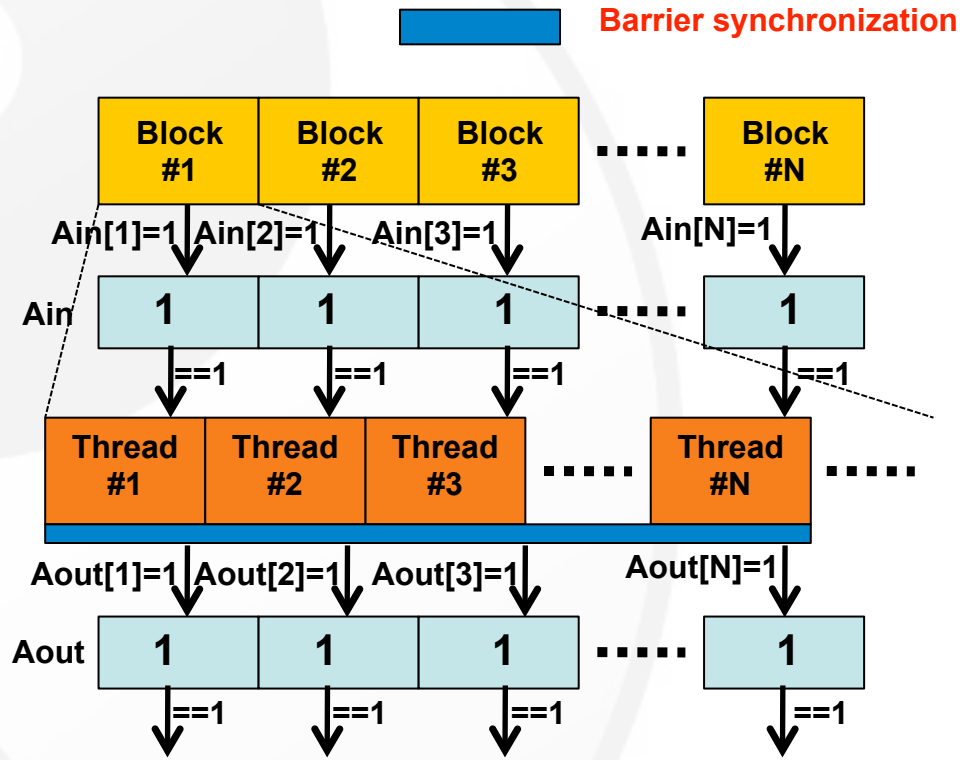
# GPU Lock-Free Synchronization

- Implementation



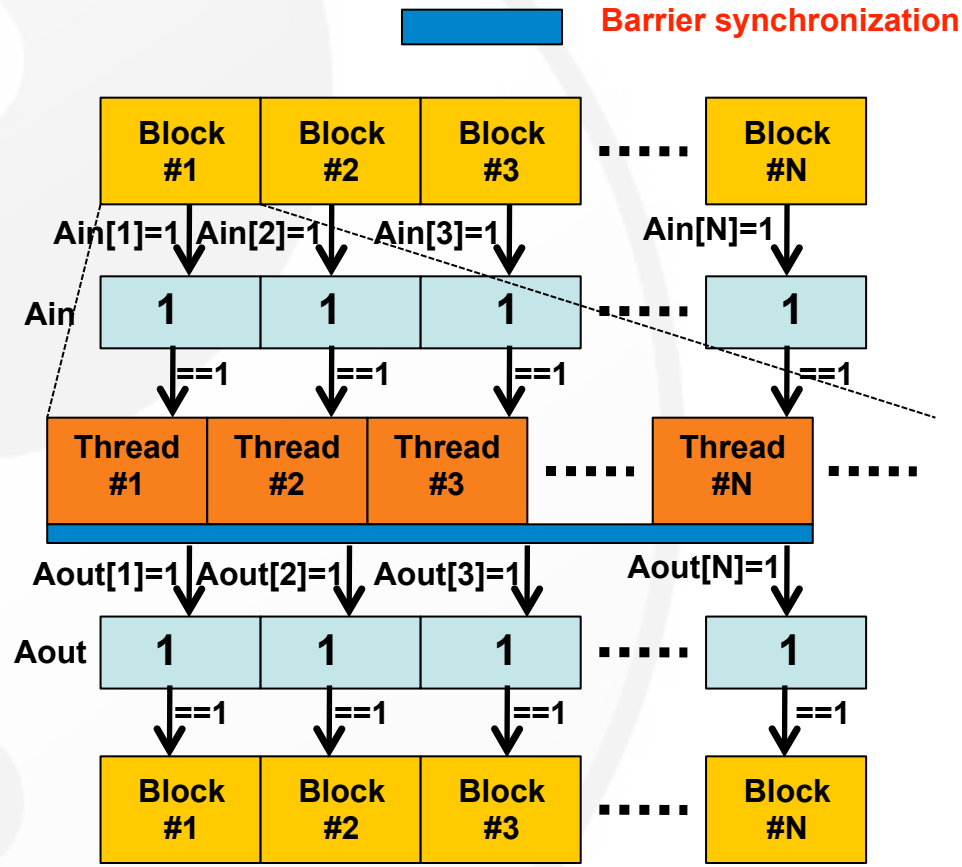
# GPU Lock-Free Synchronization

- Implementation



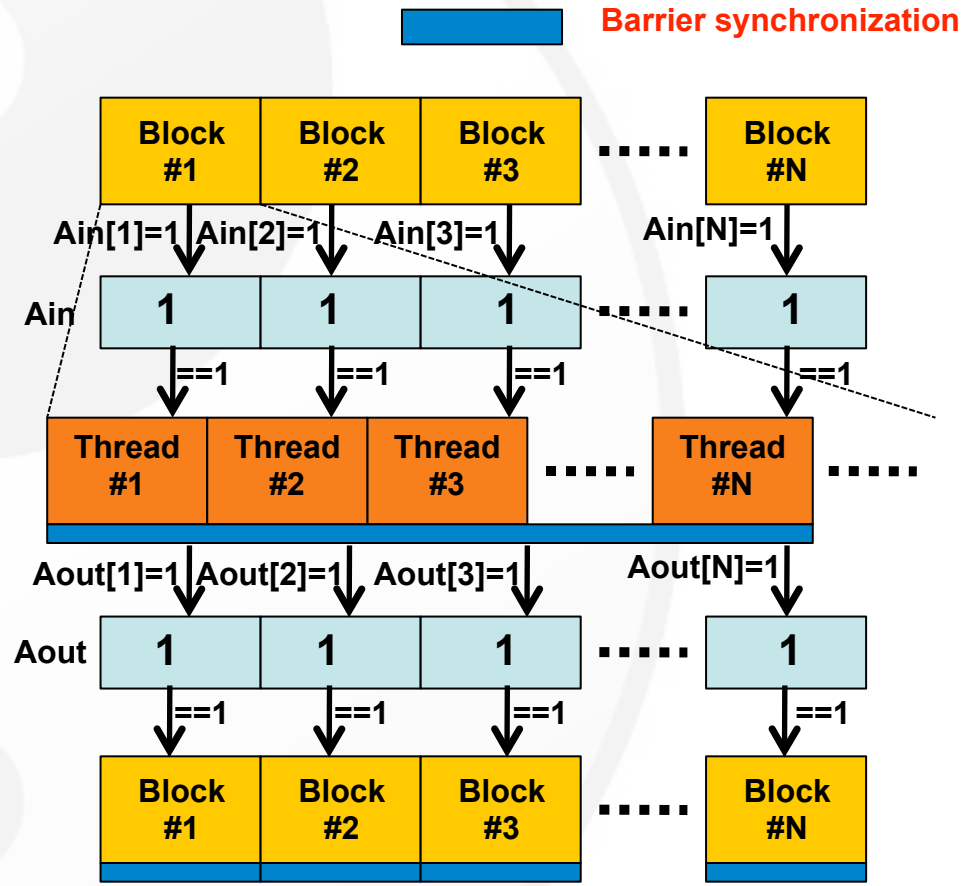
# GPU Lock-Free Synchronization

- Implementation



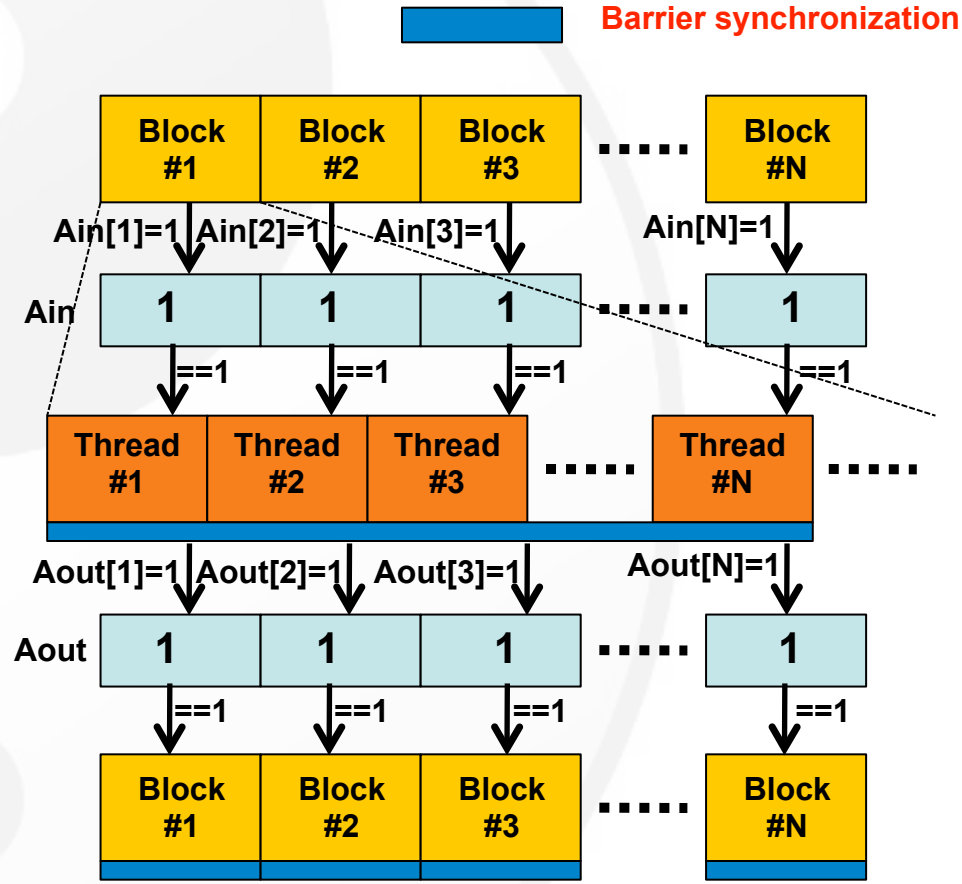
# GPU Lock-Free Synchronization

- Implementation



# GPU Lock-Free Synchronization

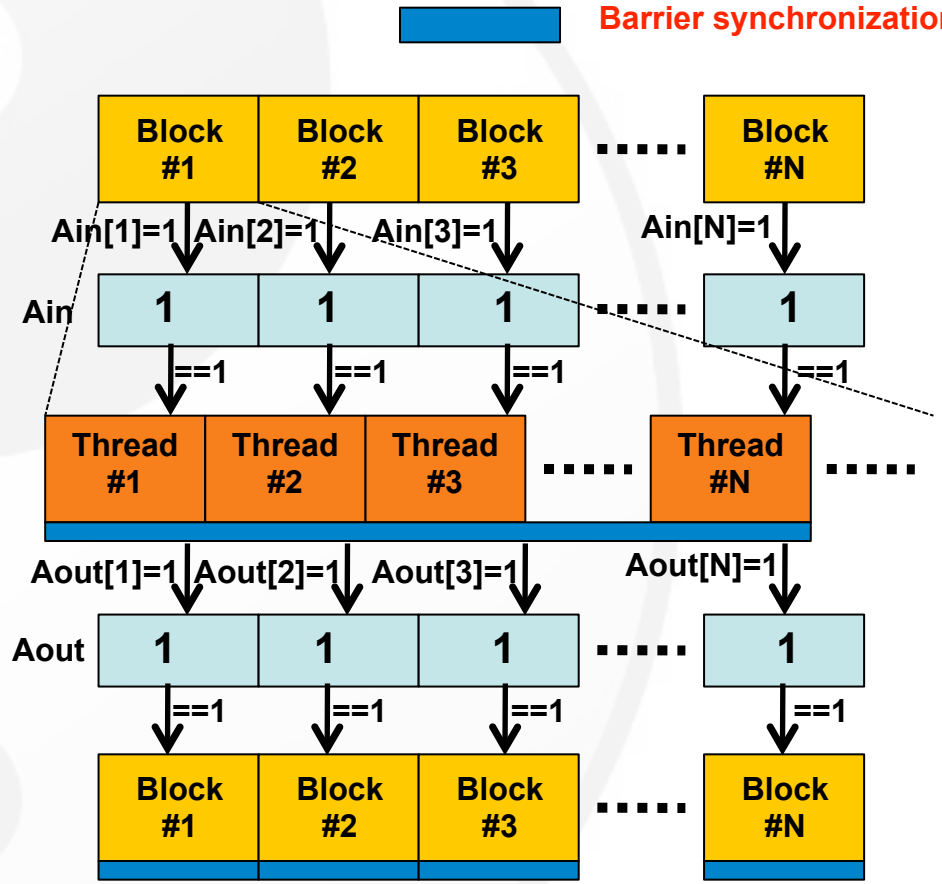
- Implementation




Note: Goal value is `1` for the first time, and then increased by `1` each time __gpu_sync() is called.

# GPU Lock-Free Synchronization

- Implementation




*Volkov & Demmel (SC '08): synchronous kernel invocation: 10–14 µs, asynchronous: 3–7. This method: 1.3–2 µs.*

Note: Goal value is `1` for the first time, and then increased by `1` each time __gpu_sync() is called.

# Spin Lock

- Lock stores 0 if unlocked, 1 if locked
- To lock, swap 1 with lock
  - Succeeded if we get a 0 back
  - Otherwise try again
- To unlock, swap 0 with lock
  - More predictable than volatile-write + threadfence
- Bad: High atomic contention

**function** CPU: CreateSpinLock

```
1: X ← AllocateGPUWord()
2: *X ← 0
3: return X
```

**function** GPU: SpinLock(Lock)

```
1: Locked ← false
2: while Locked = false do
3:     OldVal ← atomicExch(Lock, 1)
4:     if OldVal = 0 then
5:         Locked ← true
6:     end if
7: end while
```

**function** GPU: SpinUnlock(Lock)

```
1: : atomicExch(Lock, 0)
```

# Lots ‘o Stats

*Recent work in our group. Unpublished. On arxiv. Looking for venue.*

| | Tesla Reads (ms) | Tesla Writes (ms) | Fermi Reads (ms) | Fermi Writes (ms) |
|---|---|---|---|---|
| Contentious Volatile | 0.848 | 0.829 | 0.494 | 0.175 |
| Noncontentious Volatile | 0.590 | 0.226 | 0.043 | 0.029 |
| Contentious Atomic | 78.407 | 78.404 | 1.479 | 1.470 |
| Noncontentious Atomic | 0.845 | 0.991 | 0.437 | 0.312 |
| Contentious Volatile preceded by Atomic | 0.923 | 0.915 | 1.473 | 0.824 |
| Noncontentious Volatile preceded by Atomic | 0.601 | 0.228 | 0.125 | 0.050 |

| | Tesla Reads | Tesla Writes | Fermi Reads | Fermi Writes |
|---|---|---|---|---|
| Volatiles | 1.44× | 3.67× | 11.49× | 6.03× |
| Atomics | 92.79× | 79.12× | 3.38× | 4.71× |
| Volatiles preceded by Atomic | 1.54× | 4.01× | 11.78× | 16.48× |

| | Tesla Reads | Tesla Writes | Fermi Reads | Fermi Writes |
|---|---|---|---|---|
| Contentious Atomics | 92.46× | 94.57× | 2.99× | 8.40× |
| Noncontentious Atomics | 1.43× | 4.38× | 10.16× | 10.76× |
| Contentious Volatile preceded by Atomic | 1.08× | 1.10× | 2.98× | 4.71× |
| Noncontentious Volatile preceded by Atomic | 1.02× | 1.01× | 2.91× | 1.72× |

- Important parameters for synchronization design:
  - Atomic:volatile ratio, especially under contention. Are spin locks viable?
  - Contentious:noncontentious ratio. Do sleeping algorithms make sense?
  - Atomic capture: Does an atomic hold a cache line hostage?

# Synchronization Primitive Design

```
function GPU: SpinMutexLock(Mutex)
 1: Acquired ← false
 2: while Locked = false do
 3:     OldVal ← atomicExch(Mutex, 1)
 4:     if OldVal = 0 then
 5:         Acquired ← true
 6:     else if Acquired = false∧ UseBackoff = true then
 7:         Backoff()
 8:     end if
 9: end while
```

```
function GPU: FAMutexLock(Mutex)
 1: TicketNumber ← atomicInc(Mutex.ticket)
 2: while TicketNumber ≠ Mutex.turn do
 3:     Backoff()
 4: end while
```

*this design also ensures fairness: service in order of arrival*

- Evaluated designs for barrier, mutex, semaphore
- General strategies:
  - Minimize atomics
  - Avoid contentious atomics
  - Sleeping is often a win

*[joint work with Jeff Stuart]*

# Knuth's Algorithm for shuffling

*Recent work in our group. Unpublished. Written in an unconventional style. Looking for venue.*

- For each item *i* (left to right), swap that item with a randomly chosen item *j* where *j* ≥ *i*

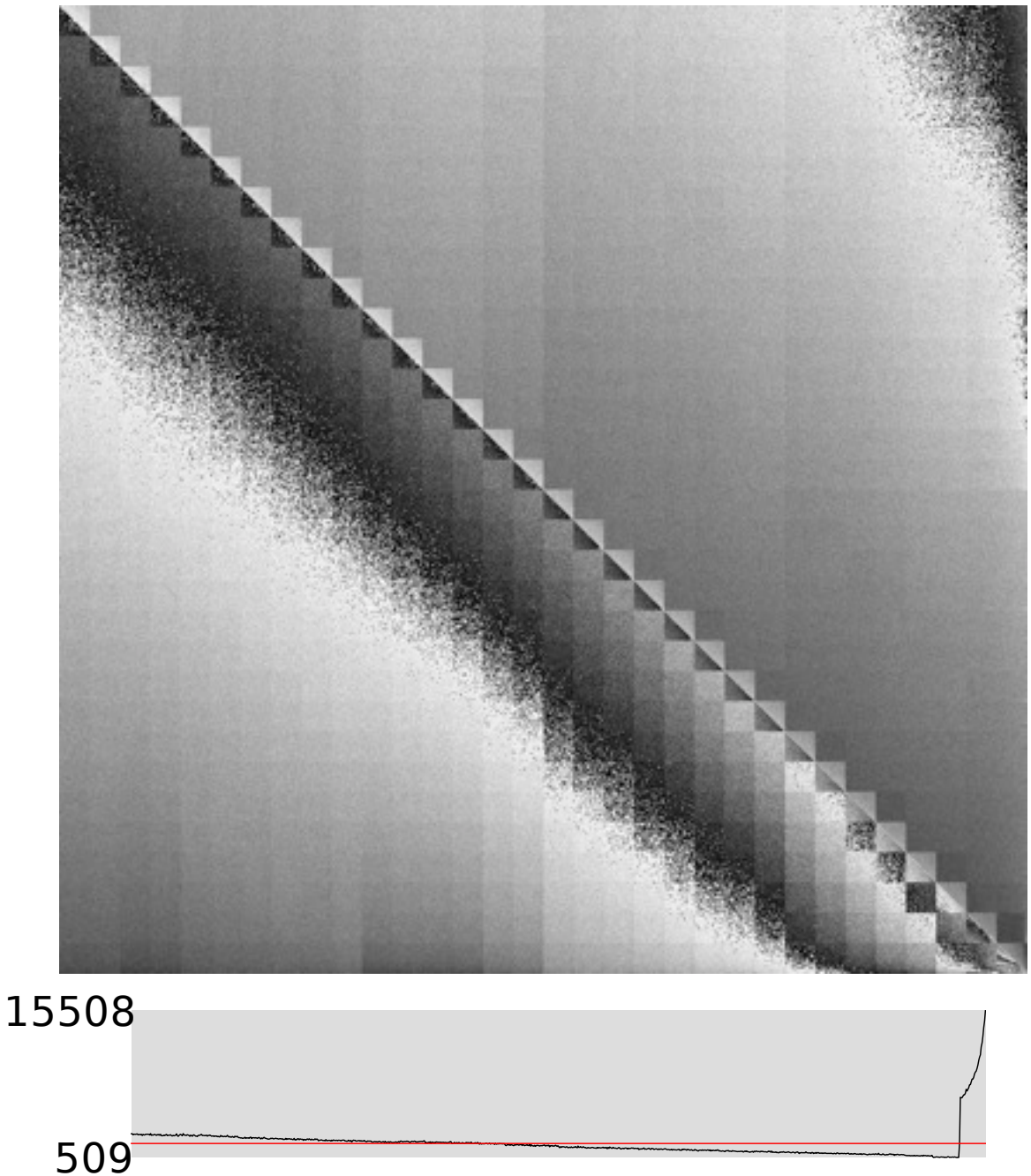
**Algorithm 1** Parallel version of Knuth's algorithm

```
procedure KnuthPermuteParallel (int a[])
1: for i=1 to n do {in parallel}
2:     j = rand(n-i)+i
3:     lock(a[i]); lock(a[j])
4:     swap(a[j], a[i])
5:     unlock(a[i]); unlock(a[j])
6: end for
```

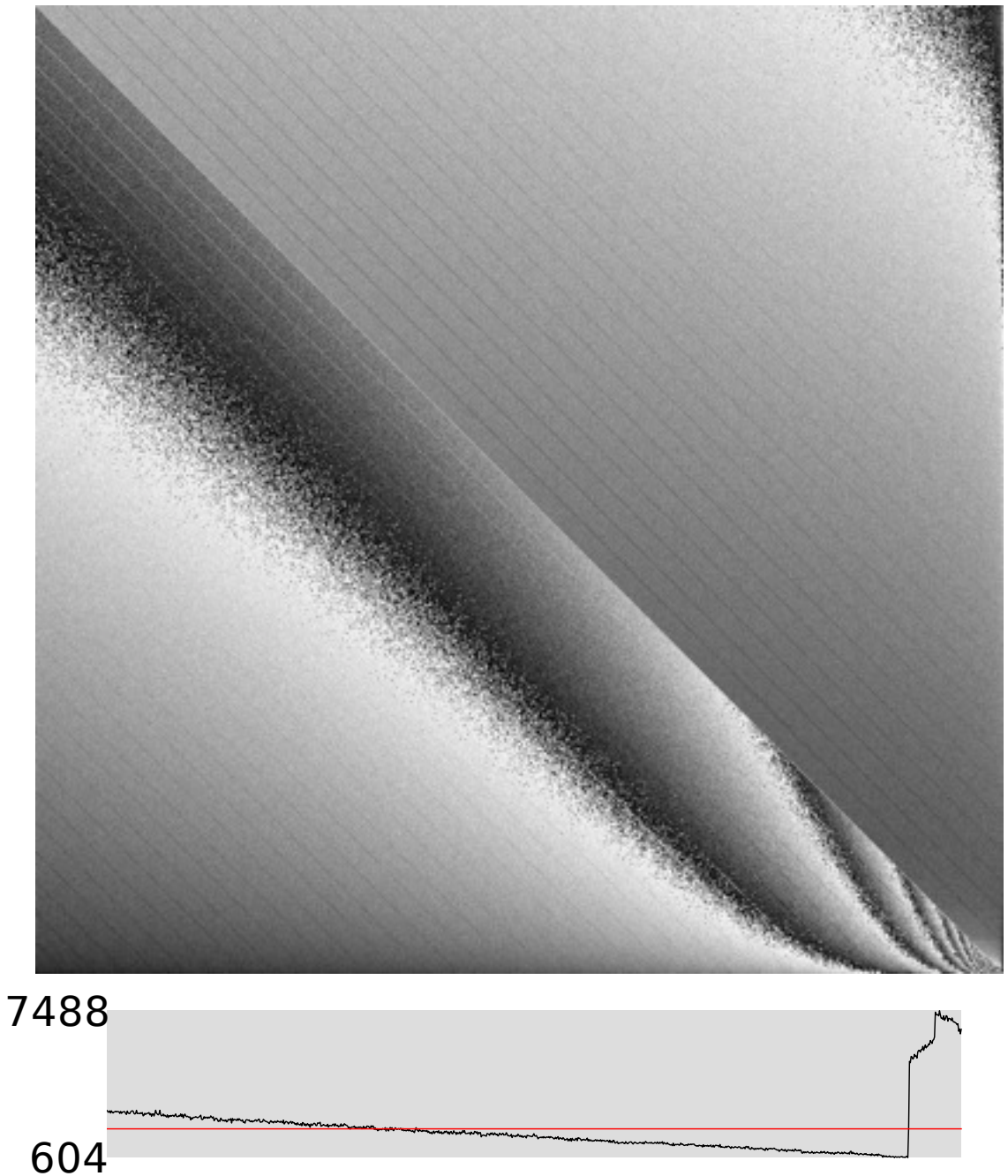
*[joint work with Andrew Davidson and Anjul Patney]*

*How do you tell if it worked?*

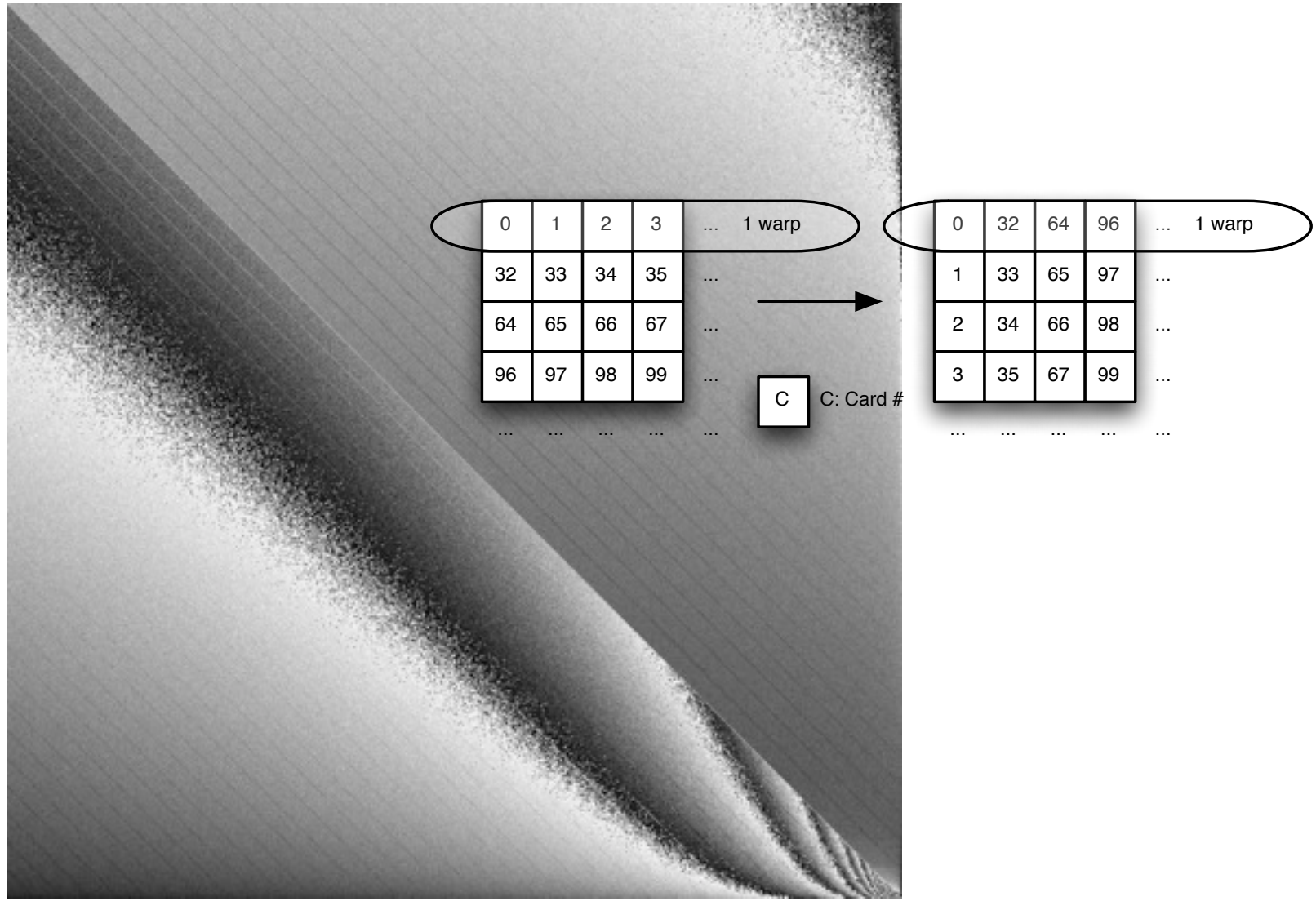
# Original implementation



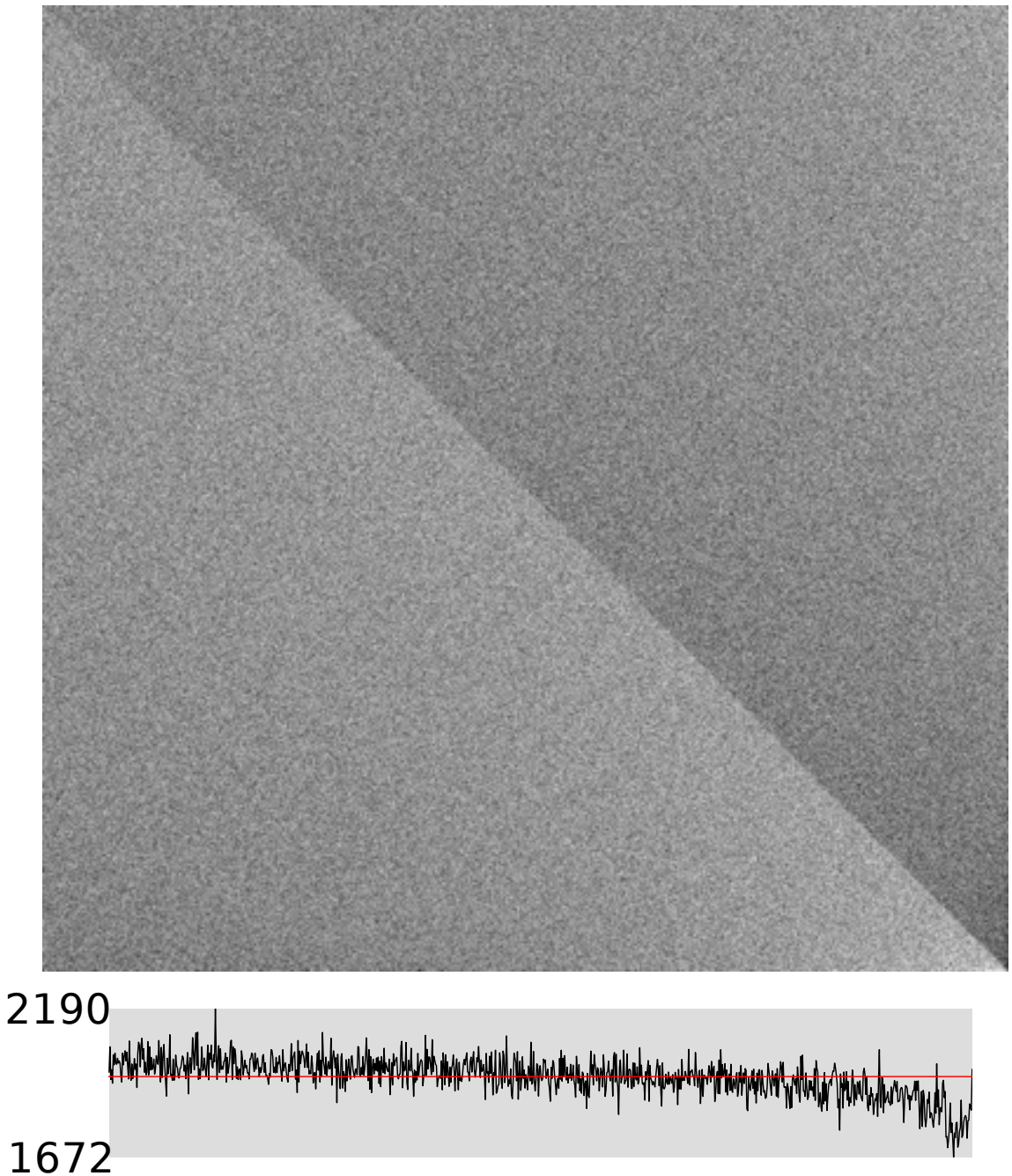
# Reindex warps



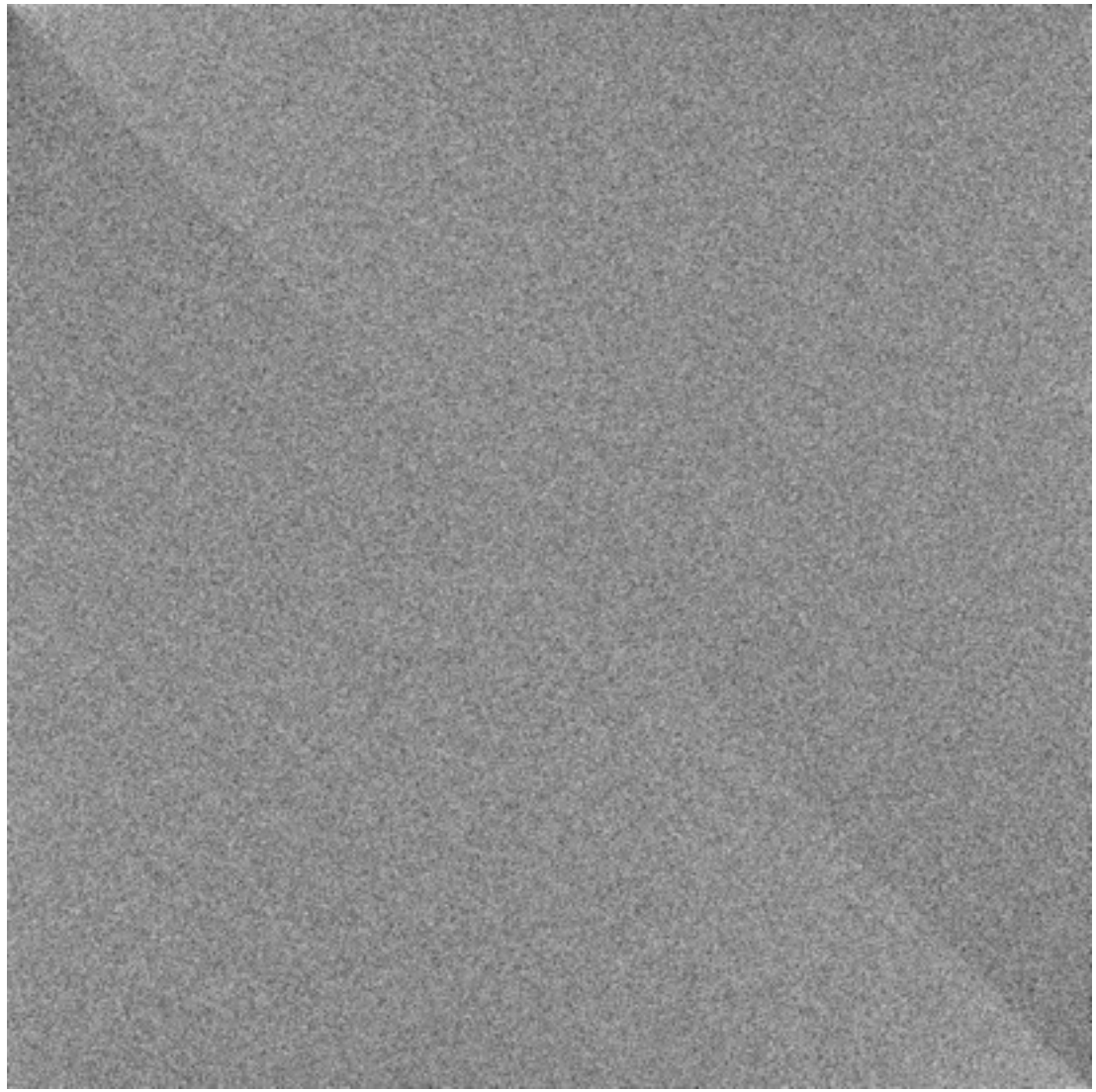
# Reindex warps

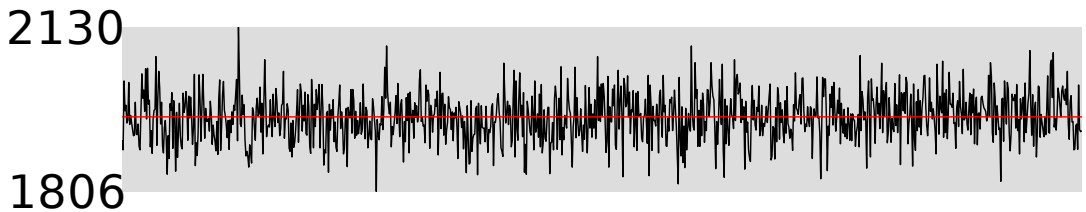


# Random waits on warps

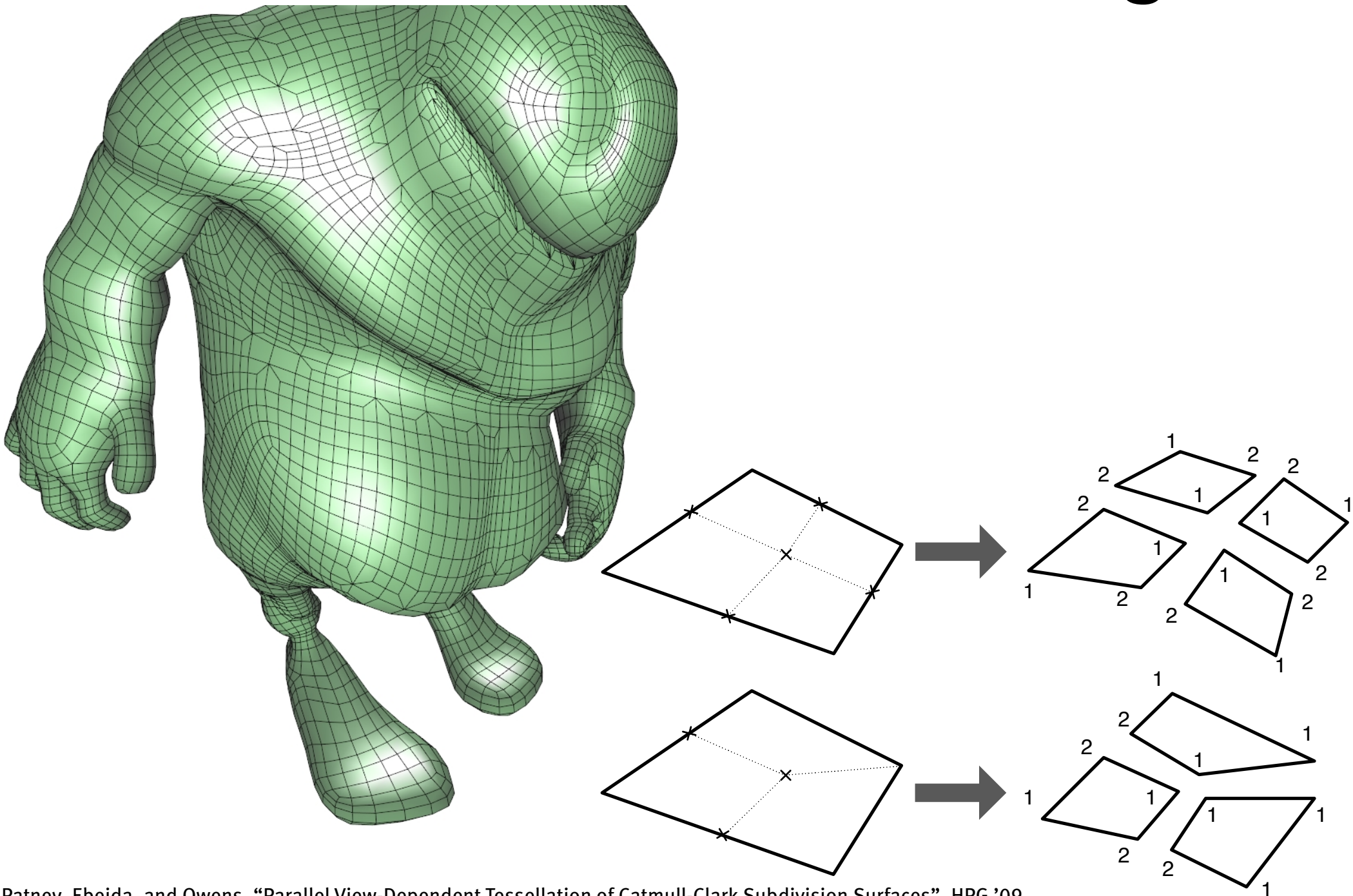


# Random swap left/right





# Recursive Subdivision is Irregular



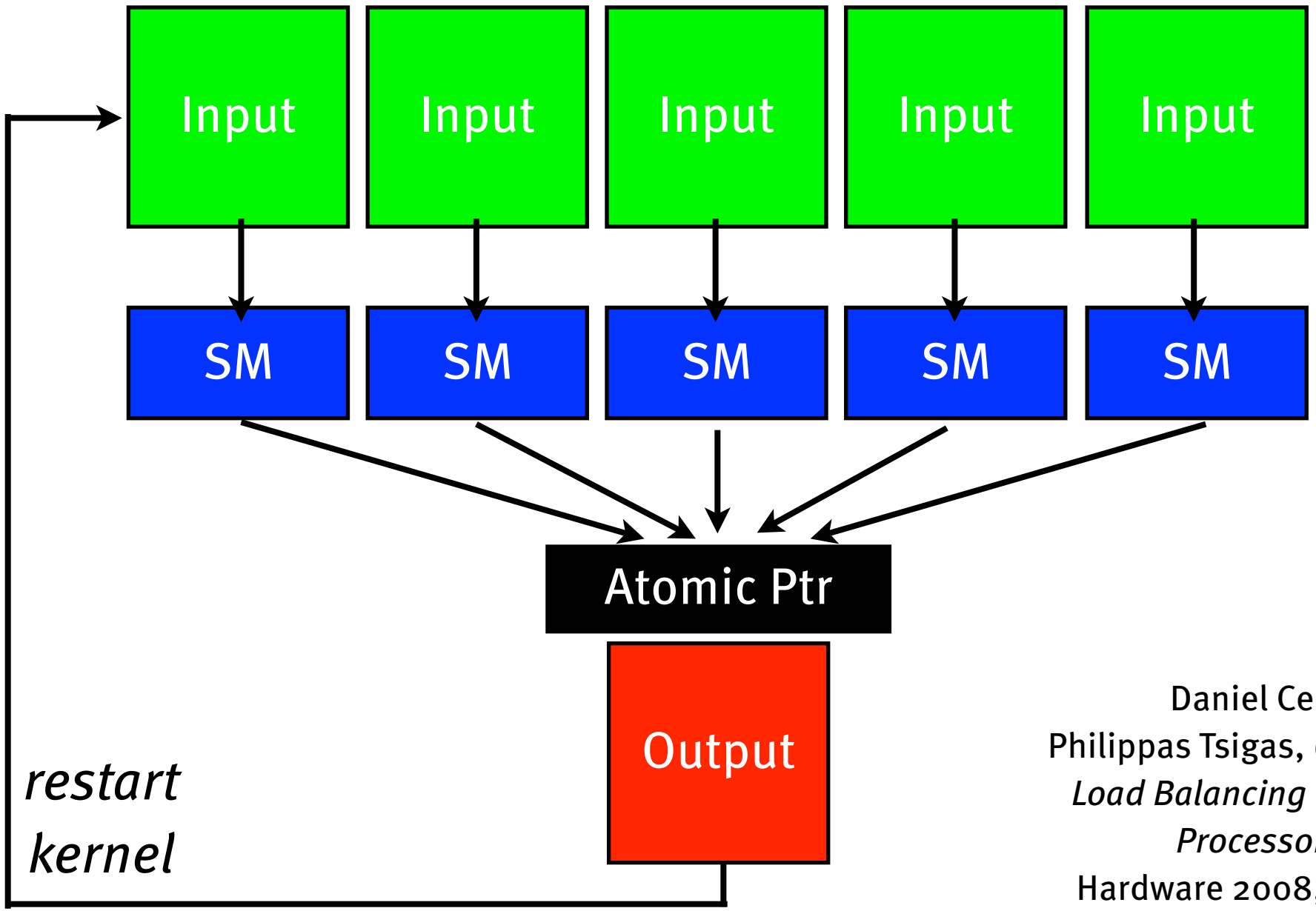

Patney, Ebeida, and Owens. “Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces”. HPG ’09.


08/03/09 10:18 am


08/03/09 10:18 am


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

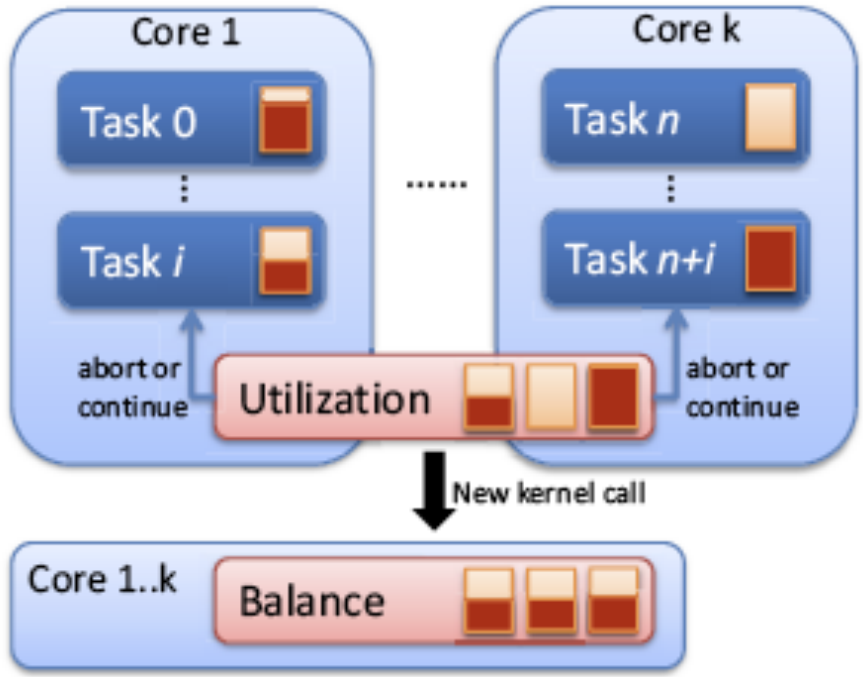
# Static Task List




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

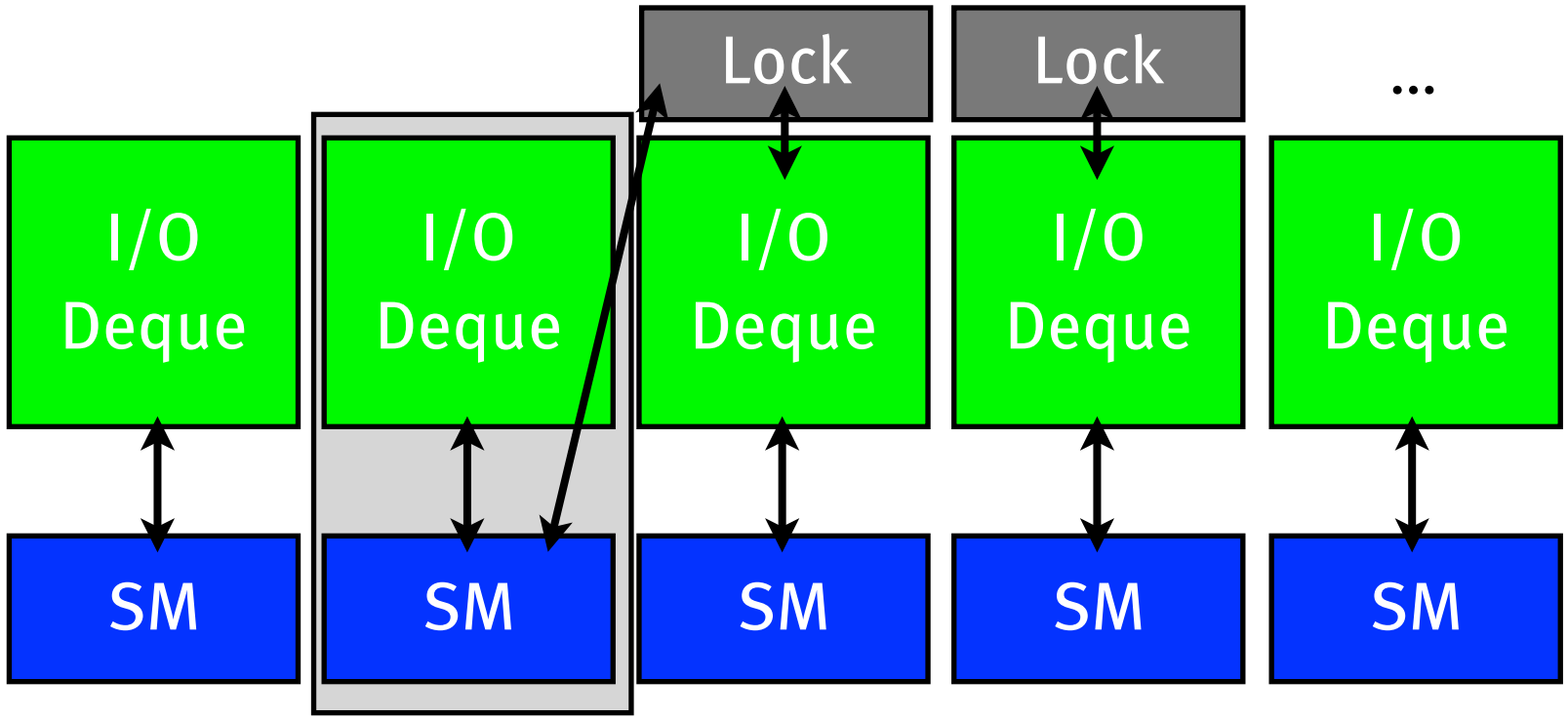
# Private Work Queue Approach

- 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

Stanley Tzeng, Anjul Patney, and John D. Owens. **Task Management for Irregular-Parallel Workloads on the GPU.** HPG '10.

# 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?

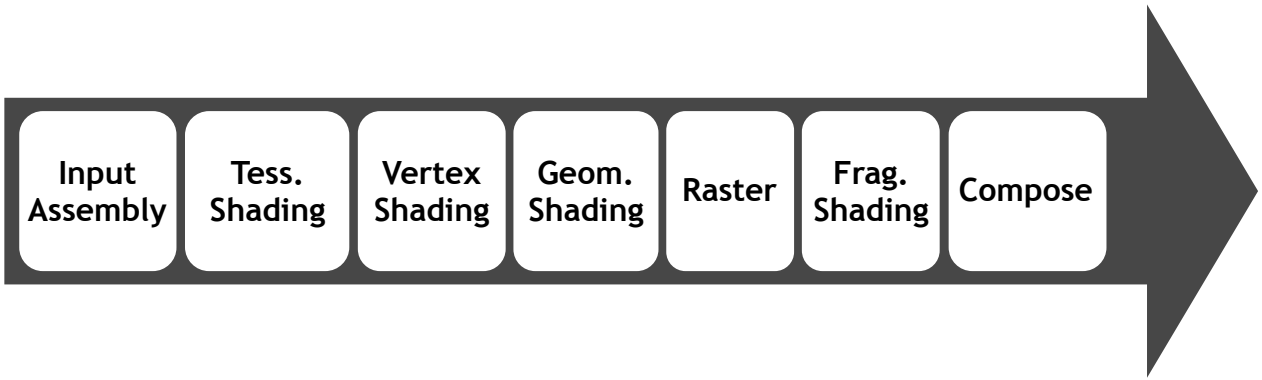
**Warp Size**
**Work Granularity**

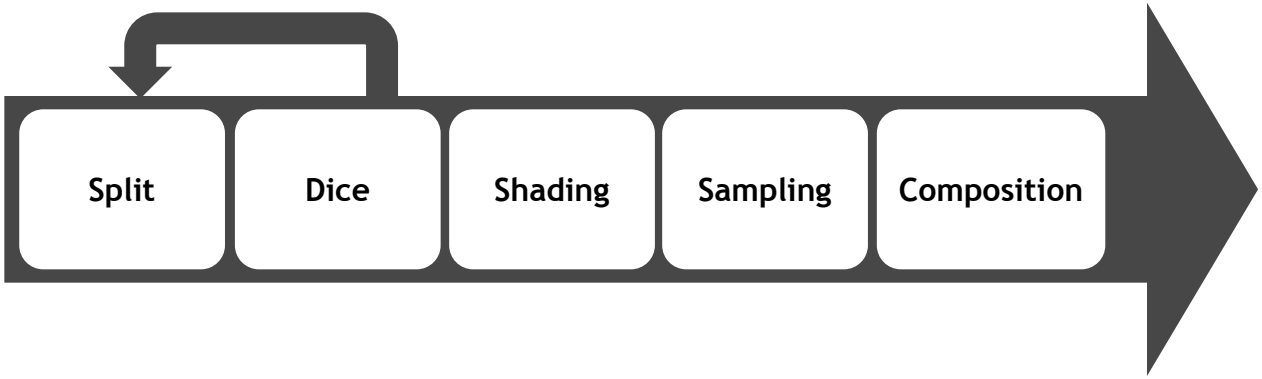
**Persistent Threads**

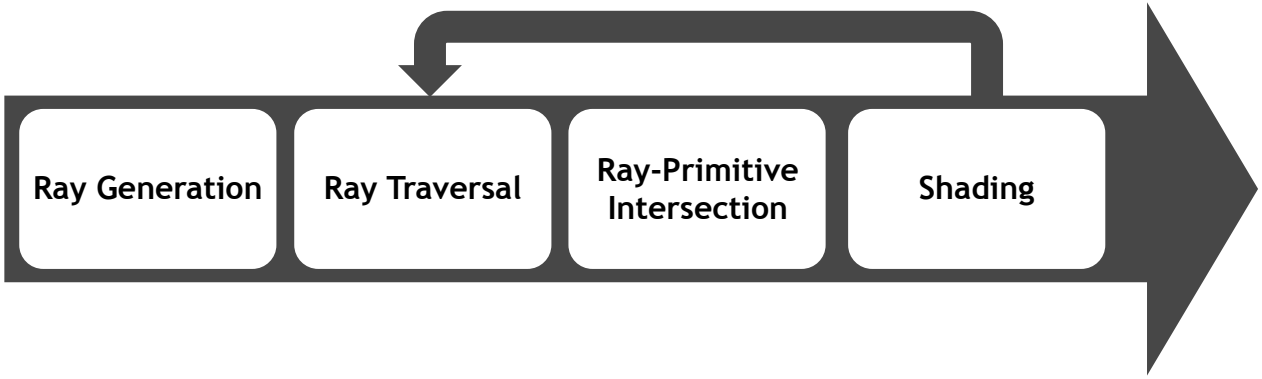
**Uberkernels**

**Task Donation**

# The Programmable Pipeline

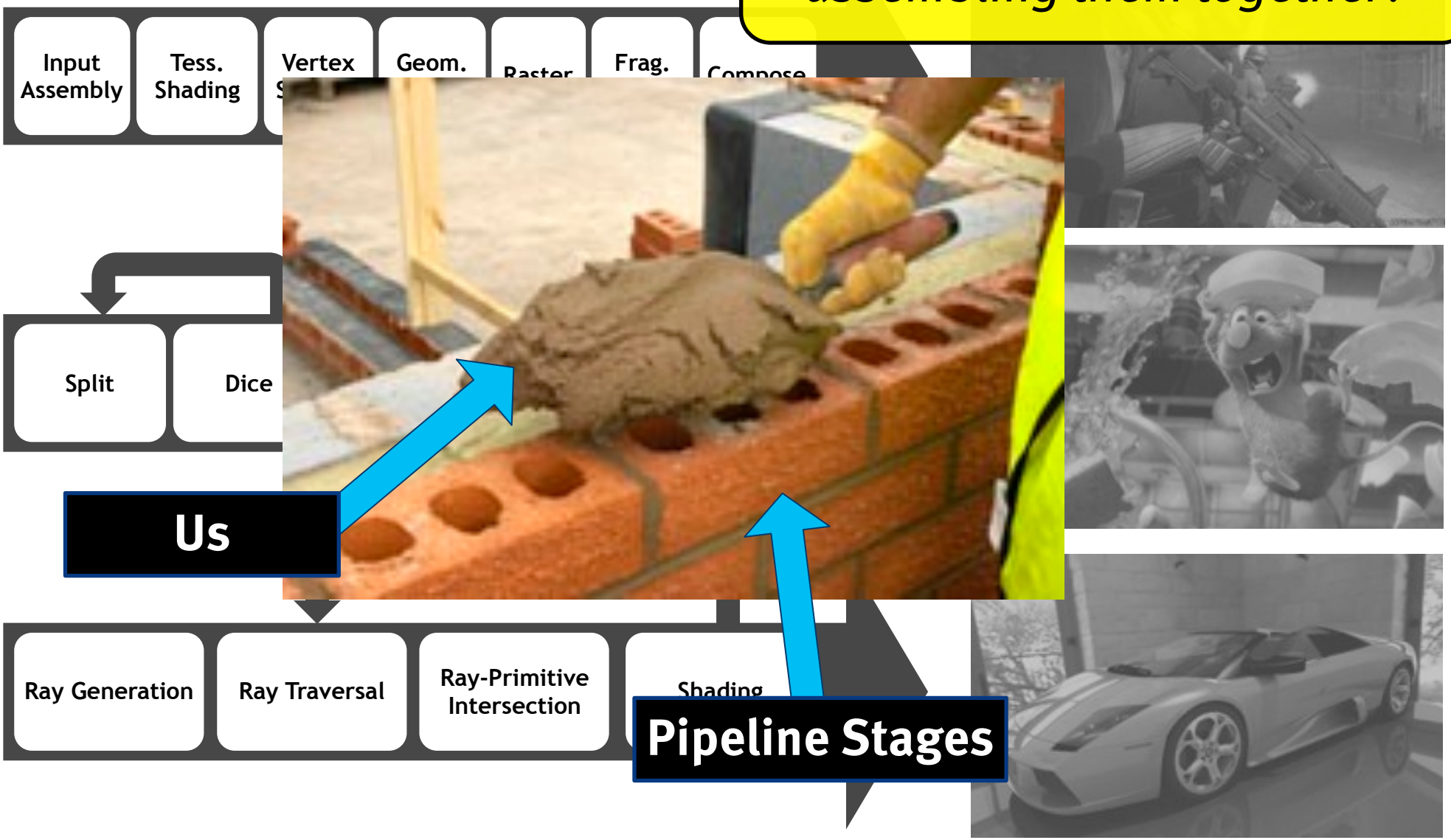



# The Programmable Pipeline

*Bricks & mortar: how do we allow programmers to build stages without worrying about assembling them together?*



# Thanks to ...


- The organizers for the invitation
- Wu Feng, Jeff Stuart, Duane Merrill, Anjul Patney, and Stanley Tzeng for helpful comments and slide material.
- Funding agencies: Department of Energy (SciDAC Institute for Ultrascale Visualization, Early Career Principal Investigator Award), NSF, Intel Science and Technology Center for Visual Computing, LANL, BMW, NVIDIA, HP, UC MICRO, Microsoft, ChevronTexaco, Rambus

“If you were plowing a field, which would you rather use? Two strong buffalo or 1024 chickens?”

—Seymour Cray