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.
• Hardware responsible for assigning blocks to “SMs” (“streaming multiprocessors” or “cores”—think of them as virtual blocks).

• Different GPUs have different numbers of SMs.
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
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 a warp
  - Newest NVIDIA GPUs add `_syncthreads_count(p)`, `_syncthreads_or(p)`, and `_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

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

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

**Volkov & Demmel (SC ’08):**
- synchronous kernel invocation: 10–14 µs,
- asynchronous: 3–7
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 \( \mu s \)) 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

- 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 $O(\text{number of SMs})$ rather than $O(\text{input size})$
Persistent Threads

- 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

**Recent work in our group. In submission.**

[joint work with Kshitij Gupta and Jeff Stuart]
GPU Lock-Based Synchronization

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>FFT</th>
<th>Smith-Waterman</th>
<th>Bitonic sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>% of time spent on inter-thread communication</td>
<td>17.8%</td>
<td>49.2%</td>
<td>59.6%</td>
</tr>
</tbody>
</table>

GPU Lock-Based Synchronization

GPU Lock-Based Synchronization

GPU Lock-Based Synchronization

GPU Lock-Based Synchronization

GPU Lock-Based Synchronization

GPU Lock-Based Synchronization

GPU Lock-Based Synchronization

GPU Lock-Based Synchronization

GPU Lock-Free Synchronization

• Implementation
GPU Lock-Free Synchronization

• Implementation

![Diagram of GPU Lock-Free Synchronization]

- **Ain**
  - Block #1: Ain[1] = 1
  - Block #2: Ain[2] = 1
  - Block #3: Ain[3] = 1
  - Block #N: Ain[N] = 1

- **Aout**
  - 0 0 0
  - 0
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

![Diagram of GPU Lock-Free Synchronization](image)

- Barrier synchronization
- Blocks and threads with input values Ain and output values Aout

13
GPU Lock-Free Synchronization

- Implementation

<table>
<thead>
<tr>
<th>Block #1</th>
<th>Block #2</th>
<th>Block #3</th>
<th>Block #N</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
<th>Thread #3</th>
<th>Thread #N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ain = 1</td>
<td>Ain = 1</td>
<td>Ain = 1</td>
<td>Ain = 1</td>
</tr>
<tr>
<td><strong>==1</strong></td>
<td><strong>==1</strong></td>
<td><strong>==1</strong></td>
<td><strong>==1</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
</tbody>
</table>

Barrier synchronization
GPU Lock-Free Synchronization

• Implementation

Barrier synchronization
GPU Lock-Free Synchronization

• Implementation
GPU Lock-Free Synchronization

- Implementation

```plaintext

Thread #1  Thread #2  Thread #3  Thread #N

Barrier synchronization
```
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

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: \( X \leftarrow \text{AllocateGPUWord}() \)
2: \( *X \leftarrow 0 \)
3: return \( X \)

function GPU: SpinLock(Lock)
1: Locked \( \leftarrow \text{false} \)
2: while Locked = false do
3: \( \text{OldVal} \leftarrow \text{atomicExch}(\text{Lock}, 1) \)
4: if \( \text{OldVal} = 0 \) then
5: \( \text{Locked} \leftarrow \text{true} \)
6: end if
7: end while

function GPU: SpinUnlock(Lock)
1: \( : \text{atomicExch}({\text{Lock}, 0}) \)
Lots 'o Stats

<table>
<thead>
<tr>
<th></th>
<th>Tesla Reads (ms)</th>
<th>Tesla Writes (ms)</th>
<th>Fermi Reads (ms)</th>
<th>Fermi Writes (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contentious Volatile</td>
<td>0.848</td>
<td>0.829</td>
<td>0.494</td>
<td>0.175</td>
</tr>
<tr>
<td>Noncontentious Volatile</td>
<td>0.590</td>
<td>0.226</td>
<td>0.043</td>
<td>0.029</td>
</tr>
<tr>
<td>Contentious Atomic</td>
<td>78.407</td>
<td>78.404</td>
<td>1.479</td>
<td>1.470</td>
</tr>
<tr>
<td>Noncontentious Atomic</td>
<td>0.845</td>
<td>0.991</td>
<td>0.437</td>
<td>0.312</td>
</tr>
<tr>
<td>Contentious Volatile preceded by Atomic</td>
<td>0.923</td>
<td>0.915</td>
<td>1.473</td>
<td>0.824</td>
</tr>
<tr>
<td>Noncontentious Volatile preceded by Atomic</td>
<td>0.601</td>
<td>0.228</td>
<td>0.125</td>
<td>0.050</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Tesla Reads</th>
<th>Tesla Writes</th>
<th>Fermi Reads</th>
<th>Fermi Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volatiles</td>
<td>1.44×</td>
<td>3.67×</td>
<td>11.49×</td>
<td>6.03×</td>
</tr>
<tr>
<td>Atomics</td>
<td>92.79×</td>
<td>79.12×</td>
<td>3.38×</td>
<td>4.71×</td>
</tr>
<tr>
<td>Volatiles preceded by Atomic</td>
<td>1.54×</td>
<td>4.01×</td>
<td>11.78×</td>
<td>16.48×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Tesla Reads</th>
<th>Tesla Writes</th>
<th>Fermi Reads</th>
<th>Fermi Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contentious Atomics</td>
<td>92.46×</td>
<td>94.57×</td>
<td>2.99×</td>
<td>8.40×</td>
</tr>
<tr>
<td>Noncontentious Atomics</td>
<td>1.43×</td>
<td>4.38×</td>
<td>10.16×</td>
<td>10.76×</td>
</tr>
<tr>
<td>Contentious Volatile preceded by Atomic</td>
<td>1.08×</td>
<td>1.10×</td>
<td>2.98×</td>
<td>4.71×</td>
</tr>
<tr>
<td>Noncontentious Volatile preceded by Atomic</td>
<td>1.02×</td>
<td>1.01×</td>
<td>2.91×</td>
<td>1.72×</td>
</tr>
</tbody>
</table>

- **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

- For each item $i$ (left to right), swap that item with a randomly chosen item $j$ where $j \geq i$

Algorithm 1 Parallel version of Knuth’s algorithm

procedure KnuthPermuteParallel (int a[])

1: for $i$=1 to $n$ do \{in parallel\}
2: $j = \text{rand}(n-i)+i$
3: lock(a[i]); lock(a[j])
4: swap(a[j], a[i])
5: unlock(a[i]); unlock(a[j])
6: end for
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.
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
• 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
The Programmable Pipeline


Split  Dice  Shading  Sampling  Composition

Ray Generation  Ray Traversal  Ray-Primitive Intersection  Shading
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