ICS: U: Towards Shared Memory Consistency Models for GPUs

Tyler Sorensen
University of Utah
t.sorensen@utah.edu

Jade Alglave
University College London
j.alglave@ucl.ac.uk

Ganesh Gopalakrishnan
University of Utah
ganesh@utah.edu

Vinod Grover
NVIDIA
vgrover@nvidia.com

I. Problem and Motivation

A Graphics Processing Unit (GPU) is a compute accelerated microprocessor designed with many cores and high data bandwidth [12, p. 3-5]. These devices were originally used for graphics acceleration; however, their high arithmetic throughput and energy efficiency made them attractive for use in other applications. In 2006, NVIDIA released their first general-purpose GPU that supported the CUDA architecture [19, p. 6] which allowed programmers to develop applications more easily that run on GPUs. Since then, GPUs have continued to be used in many applications and are present in devices ranging from the top supercomputers [22] to smartphones and tablets [27].

GPUs are concurrent shared memory devices and share many of the concurrency considerations as their multicore CPU counterparts. One example, and our focus, is the architecture's shared memory consistency model (or *memory model* for short) which governs the values that can be read from memory when issued concurrently with other reads and writes.

While memory models for CPUs have been well studied [3, 24], GPUs have a substantially different concurrency and memory system. Despite this, GPU vendor documentation on memory models remains sparse. For example, the documentation for NVIDIA's low level intermediate language PTX [16] has less than two pages describing the memory fences and no examples. Without a well specified memory model, programmers cannot write robust and portable code using custom synchronization idioms (e.g. locks, non-blocking data structures). Furthermore expensive and critical scientific simulation results could be cast into serious doubt.

This work presents several contributions towards better understanding of GPU memory models. First we propose a variety of *litmus tests* for GPUs. These tests describe relaxed coherence and shed light on scope-sensitive memory fence behavior — a unique aspect of GPUs. Next we discuss the operational semantics for a putative GPU operational memory model which we dub the Utah GPU (or UGPU) model. This proposed model captures the semantics of basic load, store, and scoped fence instructions. The model is implemented in the Murphi modeling language [7] and is able to run our GPU litmus tests. This work was originally presented in [23].

While the UGPU model can reason about an idealistic GPU memory model, it is unclear how well it describes deployed hardware and hence, how useful it is to the practitioner. To this end, we present ongoing work to extend the *litmus* hardware testing tool [2] to test GPUs. With this, we show that GPUs implement weak memory models with subtle scoped properties unseen in CPU models; we then compare the UGPU model to our experimental observations. We show that older NVIDIA chips (Kepler) implement a controversial relaxed coherence behavior while newer chips (Maxwell) do not. Finally, we show that a published implementation of a GPU spin-lock [19, p. 250-258] does not provide sufficient synchronization to implement what is generally considered correct for a lock.

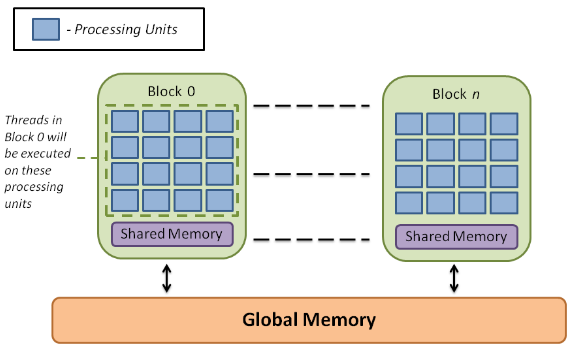


Fig. 1. Thread hierarchy and memory regions of a GPU architecture.

II. Background and Related Work

A. GPU Architecture

GPU architectures have several key differences from their CPU counterparts. Developers have explicit access to the location of threads in the GPU thread hierarchy and can design programs using this information; threads that share finer grained levels of the hierarchy enjoy accelerated interactions and additional functionality. For example, one level of the hierarchy is called a *block*. A GPU program often has many blocks, and threads residing in the same block have access to a fast region of memory called *shared* memory. Threads in different blocks cannot access the same shared memory region and must use the slower *global* memory region to share data [15, p. 10-12]. The thread hierarchy and memory regions of a single GPU device are shown in Figure 1.

B. Memory Models, Litmus Tests and Testing

For a given program and architecture, a memory model defines the set of values that the load instructions are allowed to return. That is, it specifies all possible behaviors of shared memory interactions. Our work describes an operational-semantic (or *operational* for short) memory model, which

initial state: x = 0, y = 0	
Thread 1	Thread 2
`a: *x = 1;`	`c: *y = 1;`
`b: r1 = *y;`	`d: r2 = *x;`
assert: $r1 = 0 \land r2 = 0$	

Fig. 2. Classic Store Buffering Litmus test. This assertion cannot be satisfied under an SC memory model.

means that the system is described as an abstract machine. Given the current state of the system, the operational model will provide all possible transitions the system could take and how the system state is updated based on the transition; examples of operational models include [20, 21]. Memory models may alternatively be defined in an *axiomatic* style where constraints are described on sets and relations over memory actions; for examples of this type of model see [1, 3, 14].

An intuitive way to understand memory models is through *litmus tests*, i.e. short concurrent programs with an assertion about the final states of registers and memory. Litmus tests are evaluated under a memory model and can be allowed (the assertion sometimes passes) or disallowed (the assertion never passes). Figure 2 shows a litmus test known as *store buffering*. Litmus tests are prevalent illustrations in memory model literature [1, 3, 17].

Many programmers reason about concurrent programs under the *sequentially consistent* memory model (or simply SC), first defined by Lamport in 1979 [13]. That is, a concurrent execution must correspond to some interleaving of the instructions. However, modern multiprocessors (e.g. x86, ARM) implement *weak memory models*, where executions may not correspond to an interleaving. Weak architectures provide *fence* instructions to restore orderings. Figure 3 shows how one would reason about the store buffering litmus test under SC. While the litmus test assertion will never be allowed under SC, it is observable on x86 chips if fences are not used [17].

Some possible interleavings of store buffering litmus test		
Interleaving 1	Interleaving 2	...
`a: *x = 1;`	`a: *x = 1;`	`c: ...`
`b: r1 = *y;`	`c: *y = 1;`	`a: ...`
`c: *y = 1;`	`b: r1 = *y;`	`b: ...`
`d: r2 = *x;`	`d: r2 = *x;`	`d: ...`
final: $r1 = 0 \land r2 = 1$	final: $r1 = 1 \land r2 = 1$	...

Fig. 3. Examining some possible interleaving of the store buffering litmus test (see Figure 2). The litmus test assertion cannot be satisfied in SC.

The GPU testing framework we present is an extension of the Litmus tool [2] which runs litmus tests on different CPU architectures, (e.g. x86 and ARM). ARCHTEST and TSOTool [6, 8] are earlier memory model testing tools.

C. GPU Memory Models

The past few years have seen considerable activity in academia w.r.t. GPU memory models [9–11]. We consider this work part of that effort and hope to see the same level of rigorous testing and modeling applied to GPU memory models as CPU memory models have enjoyed [3, 21, 24]. We outline some recent work in this area:

In June 2013, Hower et al. proposed an SC for RF (i.e. a race-free program only has SC executions) memory model for GPUs [10]. Using scoped atomic operations, they build a happens-before relationship and use it to define a race which they call a *heterogeneous race*.

Also in June 2013, Hechtman and Sorin [9] showed that in a particular model of GPU and for common programming idioms on GPUs, weak memory consistency has negligible benefits. Because of this, SC is an attractive choice for their model of GPUs. Our testing work shows that despite their analysis, current GPUs do implement weak memory models.

In January 2014, Hower et al. [11] presented two SC for heterogeneous-RF memory models named HRF-direct and HRF-indirect. The first is suited for traditional GPU programs and current language standards while the latter is forward-looking to irregular GPU programs and new standards.

Our work differs from Hower et al. in that we investigate the memory model implemented on deployed GPUs. As such, we are able to test and reason about programs executed on deployed hardware. Additionally, we attempt to give semantics to all programs — *regardless of data races.*

III. Approach and Uniqueness

A. GPU Litmus Tests

GPU litmus tests extend CPU litmus tests in that they must specify the GPU thread hierarchy and memory regions discussed in Section II-A, namely:

- *GPU Thread hierarchy* - In this study we consider only single GPU device tests and restrict the threads to being either in the same block or different blocks.
- *GPU Memory regions* - In this study we consider *shared* and *global* memory regions and restrict our tests to a single memory region, i.e memory locations are either are shared or global per test.

We note that these are not all the configurations to consider, e.g. in 3+ threaded tests, some threads may be in the same block and others may be in different blocks. Similarly the same test may contain both shared and global memory locations. While our model gives semantics to all programs, we are less confident in its behavior under these configurations and leave exploration of these behaviors to future work.

GPUs have *scoped* fences, where ordering constraints are limited to certain levels in the GPU thread hierarchy. The CUDA fences [15, p. 92] we consider in this study are:

- `__thread_fence_block` **(TFB)** - This fence enforces orderings within the issuing GPU block.
- `__thread_fence` **(TF)** - This fence enforces orderings within the entire GPU device.

While we have investigated more tests (see Section V), we only discuss two tests in this paper. Both have been previously studied on CPUs, e.g. in [3]. When specifying the litmus test, we give the traditional CPU test with a generic `fence` instruction. In Section IV we parameterize these tests over the GPU memory regions, thread hierarchy and scoped fences.

1) Message Passing (MP): Figure 4 shows the message passing (MP) litmus test, in which one thread writes some data followed by a flag while the other thread reads the updated flag but does not read updated data. This test describes a handshake idiom and published locking methods can be distilled to similar tests (see Section IV-B). We investigate which GPU fence is required to disallow this test (and thus actually implement the expected handshake) under different GPU parameters.

initial state: data = 0, flag = 0	
Thread 1	Thread 2
`a: *data = 1;`	`d: r1 = *flag;`
`b: fence;`	`e: fence;`
`c: *flag = 1;`	`f: r2 = *data;`
assert: flag = 1 $\wedge$ data = 0	

Fig. 4. Specification of the Message Passing (MP) litmus test. We use a generic `fence` instruction which we parameterize for different scoped fences.

2) Relaxed Coherence: Coherence has been defined as SC for a single address (see e.g. [24, p. 14]). However some architectures (e.g. Sparc RMO [26, p. 265-267]) allow reads from the same address to be reordered; this can be seen in the Coherence Read-Read (or CoRR) litmus test (shown in Figure 5), in which Thread 2 is able to read the updated value from memory followed by a read which returns older data. This behavior has been controversial in CPU memory models as it is observable on certain ARM chips but confirmed as buggy behavior by the vendor [3, 4]. Initial discussions suggested this behavior may be intentionally allowed on GPUs and thus we implemented our model to account for it. However, due to recent feedback and testing results we have recently reopened this issue and now believe that it should *not* be allowed. This postulation is consistent with our testing results which show that CoRR is observable on Kepler architectures (2012) but not Maxwell architectures (2014).

initial state: x = 0	
Thread 1	Thread 2
`a: *x = 1;`	`b: r1 = *x;`
	`c: fence;`
	`d: r2 = *x;`
assert: r1 = 1 $\wedge$ r2 = 0	

Fig. 5. Specification of the read-read coherence (CoRR) litmus test. We use a generic `fence` instruction which we parameterize for different scoped fences.

B. UGPU Operational Model

Here we describe our putative GPU memory model dubbed the UGPU model. *We make no claim that this model is endorsed to be the actual NVIDIA hardware memory model.* Figure 6 shows the data structures and communication in the UGPU model. Specifically, each thread contains:

- *Global and Shared Address Queues*: A queue for each address. When a thread executes a load or store instruction from the program, the instruction is enqueued in the queue for the address it references. Instructions are dequeued to memory non deterministically allowing memory accesses from different addresses to be re-ordered. When a fence is executed, a special instruction denoting which type of fence (TF or TFB) is enqueued in all address queues of the issuing thread.
- *Load Array*: An unordered array of load instructions. This allows for relaxed coherence in which loads from the same address can be reordered. To enforce full coherence (e.g. disallow the CoRR test) this structure simply needs to be removed and the loads will be ordered by the above queues.
- *Shared Memory*: An array of shared memory. The shared memory is connected to all threads in the block.
- *Global Memory*: An array of global memory. The global memory is connected all threads in the device.

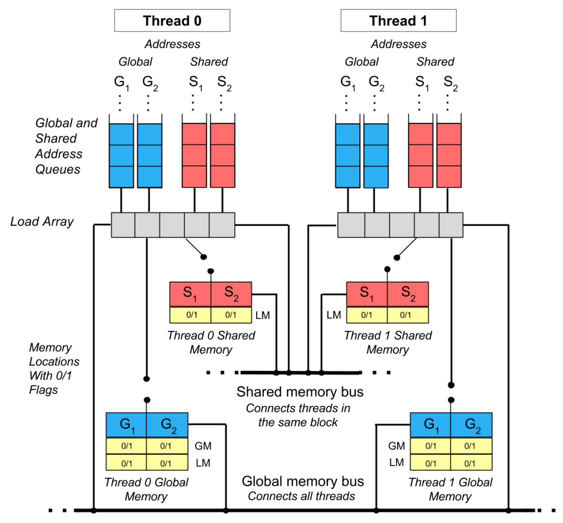


Fig. 6. A high level view of the data structures and communication in the UGPU memory model. This shows two threads in the same block where (G_1, G_2) are global addresses and (S_1, S_2) are shared addresses.

Each thread has its own view of memory to allow *write atomicity* violations [24, p. 69], i.e. threads may see updates to memory in different orders. Due to space constraints, we leave this discussion to materials referenced in Section V.

Memory locations have flags which enforce consistency and coherence (similar to a MESI protocol [18]). Fence instructions use these flags to determine which values need to be distributed to which scope. These flags are:

- *Locally Modified (LM)* - The location has been modified and needs to be distributed within the block.
- *Globally Modified (GM)* - The location has been modified and needs to be shared globally. Not on Shared memory as blocks have disjoint shared memory.

These flags on global memory give the model its scoped properties. When a thread issues a fence that provides intra-block ordering constraints (TFB), the thread must distribute all locally modified memory locations within the block. The TF fence distributes both globally and locally modified values to all threads in the GPU. In the case where the data is globally, but not locally modified, the TF fence distributes the memory to all threads *not* in the same block; this preserves coherence. Being locally modified, but not globally modified is an invalid state in our model as this would indicate that values were distributed inter-block before intra-block; we are unaware of any GPU fence that enforces such behavior.

Our model is implemented in the Murphi modeling language [7]. Rules are given in the form of a predicate and action separated by the symbol ==>. If the predicate is true, the action may execute. As an example, we show the rule for when a thread reads a global memory access from the program and enqueues it in the address queue. The complete model is pointed to in Section V.

```
Ruleset t: thread do
  Rule "Read global load or store from
        program and put in global queue"
  Alias ins:Program[t][ProgramCounter[t]]

  (ins.type = load | ins.type = store) &
  ins.memory = Global
                      ==>
  GlobalAddrQueue[t][ins.addr].enq(ins);
  ProgramCounter[t]++;
```

C. GPU Testing Framework

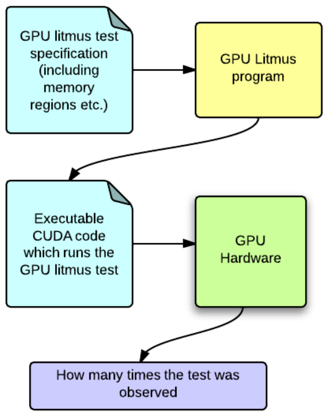
We have extended the Litmus memory model testing tool [2] to test GPUs. This tool takes a test specification along with GPU memory region and thread hierarchy information. It generates executable CUDA code that runs the GPU litmus test under system stress designed to trigger weak memory effects. The flow of the tool is shown in Figure 7. This tool allows us to compare the UGPU model with actual GPU hardware and test more complicated idioms used in deployed code.

We investigated GPU specific ways to stress the memory system in hopes of triggering weak memory behaviors. For example, we randomly place poorly aligned memory accesses (known as *bank conflicts* [15, p. 77]) next to optimally aligned accesses. These heuristics are crucial for testing and without them we were unable to observe any weak behaviors. Additionally, NVIDIA does not provide a way to program machine level instructions. To ensure our tests are unoptimized by the compiler, we check the assembly produced from a complied binary using `cuobjdump` against the test specification.

IV. Results and Contributions

A. Running Tests

We report on running the two litmus tests (MP and CoRR) on the UGPU model and on two production GPUs with different architectures — a GTX Titan (Kepler) and a GTX 750 (Maxwell). For the hardware tests, we run the tests 100,000 times and report the number of times we observe the weak behavior. We report results under different GPU parameters including fence type, memory region and GPU thread hierarchies. Our naming convention is the name of the test (MP or CoRR) followed by the fence type (TF or TFB) in parenthesis; the memory region and thread hierarchy will



GPU litmus test specification (including memory regions etc.)
GPU Litmus program
Executable CUDA code which runs the GPU litmus test
GPU Hardware
How many times the test was observed

Fig. 7. A high level flow chart of how the GPU Litmus tool is used to test GPU memory consistency.

be explicitly provided. In Figure 8 we show the results of running the tests when the threads are in the same GPU block. In Figure 9 we report the results of the tests when the threads are in different GPU blocks.

We observe that for these two families of tests, our model and observations agree exactly for the Kepler chip. The Maxwell chip shows far fewer weak behaviors e.g. only inter-block MP (no fences) is observable. However, documentation and discussions lead us to believe that it is the architectural intent to allow many of these behaviors; namely MP (no fences) should be allowed both inter and intra block and MP (TFB) should be allowed inter-block. We do not observe them either because our testing was unable to reveal them, or because the behavior is not implemented yet but may be on future chips. Finally, CoRR is observable on Kepler but not Maxwell. As stated in Section III-A2, we believe this to be intentional and the UGPU model should be revisited to reflect this change.

We further observe the memory region GPU parameter does not make a difference if the behavior is allowed or disallowed, i.e. tests allowed with shared memory are also allowed with global memory. However, the GPU thread hierarchy parameter *does* make a difference, e.g. MP (TFB) is observable inter-block, but not intra-block. While TFB is not documented to provide inter-block ordering, our results show that it reduces the number of violations observed in MP.

Results of running MP and CORR intra-block

Test	Memory Region	UGPU Model	Observed on: Kepler	Maxwell
MP (no fences)	Shared	YES	2265	0
	Global	YES	3142	0
MP (TF or TFB)	Shared	NO	0	0
	Global	NO	0	0
CoRR (no fences)	Shared	YES	4368	0
	Global	YES	9707	0
CoRR (TF or TFB)	Shared	NO	0	0
	Global	NO	0	0

Fig. 8. Results for running MP and CoRR intra-block on the UGPU model and hardware. Recall TFB and TF are the scoped GPU fences. The observation columns reports how many times the test was seen out of 100,000 runs.

Results of running MP and CORR inter-block

Test	UGPU Model	Observed on: Kepler	Maxwell
MP (no fences)	YES	4388	290
MP (TFB)	YES	1860	0
MP (TF)	NO	0	0
CoRR (no fences)	YES	1033	0
CoRR (TFB)	NO	0	0
CoRR (TF)	NO	0	0

Fig. 9. Results for running MP and CoRR intra-block on the UGPU model and hardware. Recall TFB and TF are the scoped GPU fences. The observation columns reports how many times the test was seen out of 100,000 runs.

B. CUDA by Example Spin Lock

The book *CUDA by Example* presents an inter-block mutex implementation using atomic CAS and atomic exchange operations. [19, p. 250-258] implemented as:

```
__device__ void lock(int mutex) {
  while( atomicCAS(mutex, 0, 1) != 0 ); }

 __device__ void unlock(int mutex) {
  atomicExch(mutex, 0);  }
```

We distill this spin lock into a simple litmus test we call CAS spin-lock (or CAS-SL), shown in Figure 10; it is similar to the MP tests with the addition of RMW atomic instructions and a conditional. The `m` memory location is the mutex and `d` is data accessed in the critical section. The test begins in a state where Thread 1 has the mutex. Thread 1 stores a value to `d` and then releases the mutex via an atomic exchange. Thread 2 attempts to acquire the lock with a CAS instruction, then checks to see if the lock was acquired successfully (via the conditional). If the lock was acquired, Thread 2 attempts to read the global data in `d`. The final constraint describes an execution where Thread 2 acquires the lock (`r1 = 0`) but does not see the updated value (`r2 = 0`), a behavior many would consider incorrect for a mutex.

initial state: m = 1, d = 0

```
Thread 1               Thread 2
a: *d = 1;             d: r1 = CAS(m,0,1)
b: fence;              e: if (r1 == 0) {
c: Exch(m,0);          f:     fence;
                       g:     r2 = *d;   }
```

assert: $r1 = 0 \land r2 = 0$

Fig. 10. Specification of the CAS spin-lock (CAS-SL) litmus test. Exch and CAS are atomic exchange and atomic compare-and-swap respectively. Again, we use a generic `fence` that we parameterize with GPU fences.

While the UGPU model does not have the machinery to handle atomic RMWs or conditionals, our testing tool supports these operations. Figure 11 shows the results of running CAS-SL inter-block on global memory. Without fences, we observe that Thread 1 may not see the updated memory value on Kepler chips. While the Maxwell chip does not reveal this behavior, we believe the architectural intent is to allow it. The *CUDA by Example* implementation is given without fences and is vulnerable to this behavior.

Some architectures give control flow (e.g. in Figure 10, the branch in Thread 2) ordering constraints similar to fences; however, our testing in this area is preliminary. As such, we conservatively suggest a fence even with conditionals.

Results of running CAS-SL inter-block

Test	Observed on: Kepler	Maxwell
CAS-SL (no fences)	2260	0
CAS-SL (TFB)	1672	0
CAS-SL (TF)	0	0

Fig. 11. Results for running CAS-SL inter-block on GPU hardware. We highlight that the fence-less test is vulnerable to weak behavior.

C. Contributions and Future Work

In this work, we have described GPU litmus tests and the extra parameters they require. We then gave several tests providing intuition about scoped fences and the relaxed coherence implemented on some production GPUs. We presented a putative operational GPU memory model with an executable Murphi implementation followed by a GPU testing tool which we used to compare our model to hardware. Using this tool, we identified a published spin-lock implementation [19, p. 250-258] which is lacking the proper fence instructions to provide sufficient synchronization for what is generally considered correct for a lock. Additionally, we have identified other related bugs in available code, namely another spin lock [25] and non-synchronous (or volatile) intra-warp data races [5] where the compiler may reorder instructions. We document these issues in materials referred to in Section V. We feel this body of work contributes substantially to the understanding, testing, and overall awareness of GPU memory models.

Currently we are undertaking a comprehensive study on GPU memory models (with several additional authors) which includes this testing work. The larger project involves systematically generating GPU litmus tests and modeling GPU memory models in the Herd axiomatic generic framework [3].

V. Supplementary Material and Acknowledgements

Discussion of the findings in [5] and details on the operational memory model including the implementation, more GPU litmus tests and GPU write atomicity can be found at: http://www.cs.utah.edu/~tylers/gpu_mmodel

Detailed GPU testing write-ups with even more GPU litmus tests, including a detailed report on the *CUDA by Example* spin-lock and the issue in [25] can be found at: http://www.cs.utah.edu/~tylers/gpu_testing

We acknowledge the valuable discussions provided by the other members involved in the comprehensive GPU project, namely: Mark Batty, Alastair Donaldson, John Wickerson, and Daniel Poetzl. We also thank Luc Maranget for his help with the GPU extensions to the Litmus testing tool. Finally we thank our NVIDIA contacts for feedback and discussion.

References

[1] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. CAV'10, pages 258–272. Springer-Verlag, 2010.

[2] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. TACAS'11/ETAPS'11, pages 41–44. Springer-Verlag, 2011.

[3] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. TOPLAS. ACM, 2014.

[4] ARM. Cortex-A9 MPCore, programmer advice notice, read-after-read hazards. ARM Reference 761319. http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf. 2011.

[5] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997.

[6] W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Inc., 1992.

[7] D. Dill. The Murphi verification system. CAV'96, pages 390–393. Springer-Verlag, 1996.

[8] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu. Tsotool: A program for verifying memory systems using the memory consistency model. ISCA '04. IEEE Computer Society, 2004.

[9] B. A. Hechtman and D. J. Sorin. Exploring memory consistency for massively-threaded throughput-oriented processors. ISCA'13, pages 201–212. ACM, 2013.

[10] D. R. Hower, B. M. Beckmann, B. R. Gaster, B. A. Hechtman, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Sequential consistency for heterogeneous-race-free. MSPC'13. ACM, 2013.

[11] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous-race-free memory models. ASPLOS'14, pages 427–440. ACM, 2014.

[12] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2010.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, pages 690–691, Sept. 1979.

[14] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for power multiprocessors. CAV'12, pages 495–512. Springer-Verlag, 2012.

[15] NVIDIA. CUDA C Programming Guide: Version 5.0 (July 2013). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[16] NVIDIA. Parallel Thread Execution ISA: Version 3.2 (July 2013). http://docs.nvidia.com/cuda/pdf/ptx_isa_3.2.pdf.

[17] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: X86-tso. TPHOLs '09, pages 391–407. Springer-Verlag, 2009.

[18] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. ISCA '84. ACM, 1984.

[19] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[20] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. PLDI '11, pages 175–186. ACM, 2011.

[21] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *CACM*, pages 89–97, 2010.

[22] T. . S. Sites. Top 10 sites for November 2013. http://www.top500.org/lists/2013/11/.

[23] T. Sorensen, G. Gopalakrishnan, and V. Grover. Towards shared memory consistency models for gpus. ICS'13, pages 489–490. ACM, 2013.

[24] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[25] J. A. Stuart and J. D. Owens. Efficient synchronization primitives for GPUs. *CoRR*, 2011. http://arxiv.org/pdf/1110.4623.pdf.

[26] D. L. Weaver and T. Germond. The SPARC Architecture Manual: Version 9 (1994). http://www.sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz.

[27] Wikipedia. iPad. http://en.wikipedia.org/wiki/IPad.