TOWARDS SHARED MEMORY CONSISTENCY MODELS FOR GPUS

by

Tyler Sorensen

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Bachelor of Science

School of Computing

The University of Utah

March 2013

FINAL READING APPROVAL

I have read the thesis of Tyler Sorensen in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place.

Date

Ganesh Gopalakrishnan

ABSTRACT

With the widespread use of GPUs, it is important to ensure that programmers have a clear understanding of their shared memory consistency model i.e. what values can be read when issued concurrently with writes. While memory consistency has been studied for CPUs, GPUs present very different memory and concurrency systems and have not been well studied. We propose a collection of litmus tests that shed light on interesting visibility and ordering properties. These include classical shared memory consistency model properties, such as coherence and write atomicity, as well as GPU specific properties e.g. memory visibility differences between intra and inter block threads. The results of the litmus tests are determined by feedback from industry experts, the limited documentation available and properties common to all modern multi-core systems. Some of the behaviors remain unresolved. Using the results of the litmus tests, we establish a formal state transition model using intuitive data structures and operations. We implement our model in the Murphi modeling language and verify the initial litmus tests. As a preliminary study, we restrict our model to loads and stores across global and shared memory along with two of the memory fences given in the NVIDIA PTX, `__thread_fence` and `__thread_fence_block`. Finally, we show real world examples of code that make assumptions about the GPU shared memory consistency model that are inconsistent with our proposed model.

To all my long suffering professors and mentors

CONTENTS

ABSTRACT . . . . . **i**

LIST OF FIGURES . . . . . **v**

LIST OF TABLES . . . . . **vi**

ACKNOWLEDGEMENTS . . . . . **vii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Related Work 5
1.2 Roadmap 6

2. BACKGROUND **8**

2.1 GPU Basics 8
2.2 Memory Consistency Models and Litmus Tests 10
2.3 Memory Fences 11
2.3.1 CUDA Fences 12
2.4 Operational Semantic Models and Implementation 13
2.4.1 Murphi Implementation 13

3. LITMUS TESTS AND PROPERTIES **15**

3.1 Notation 15
3.2 Coherence 16
3.2.1 Weak Coherence 17
3.3 Simple Visibility 18
3.3.1 Shared Memory 19
3.3.2 Global Memory 20
3.3.3 Putting it together 20
3.4 Write Atomicity and Cumulative Fences 21

4. GPU OPERATIONAL SEMANTIC MODELS **23**

4.1 Visual Description and Data Structures of Model 23
4.2 Annotated Operational Semantics 24
4.3 Alternative Model 42

4.4 Murphi Model 43
4.4.1 Results 45

5. EXAMPLES 47

5.1 MAGMA 47
5.2 Graph Traversal 48
5.3 Software Memory Model 50
5.4 Detecting Memory Model Violations 51

6. CONCLUSION 52

REFERENCES 53

LIST OF FIGURES

1.1 (a) Illustration of sequentially consistent executions and (b) a non sequentially consistent execution. 3

1.2 Classic IRIW litmus test. If the assert can be true then this shows a write atomicity violation. 6

2.1 Visual representation of how a vector addition routine might happen on a GPU. 9

2.2 Graphical representation of the relevant features of a GPU. 10

3.1 Litmus test that can test for coherence violation. 17

3.2 Graphical representation of coherence litmus test, a violating cycle is emphasized. 18

3.3 Litmus test that can test for weak coherence violation. 19

3.4 Establishing a causal relationship between threads. 19

3.5 Litmus test establishing dynamic properties of GPU fences. 20

3.6 Large litmus test combining both GPU fences. 21

4.1 Visual description of our shared memory consistency model for GPUs. 25

4.2 Visual description of our shared memory consistency model for GPUs. 43

LIST OF TABLES

1.1 Table of modeled instructions. 4
4.1 All bit combinations and their meanings. 42
4.2 The results of the litmus tests when ran on the models. 46

ACKNOWLEDGEMENTS

This project would not have been possible without many individuals who provided assistance in the creation and development of the project, personal support and/or financial support. The following is a short list of people I would personally like to thank without whom this document would have never seen fruition.

First and Foremost, my Professor, mentor and role model Professor Ganesh Gopalakrishnan for giving me the awesome and life-changing opportunity to get involved in undergraduate research. The experiences I've had over the last few years working with Professor Gopalakrishnan have given me a love for learning I never thought I could have. His astute insights into this project gave it momentum and substance. His tireless devotion to his students will not be forgotten.

My parents and other family members whose unwavering support and patience throughout my life has lead me to where I am today.

Professor Suresh Venkatasubramanian, who generously took time out of his schedule this semester to teach me how to write a thesis. Because of his invaluable weekly one-on-one feedback, this document (a) actually got finished and (b) is much less of an organizational train wreck than it would have been otherwise.

Dr. Vinod Grover, NVIDIA, for his expertise in this area. His contributions from industry gave this project a much needed grounding in reality.

Professor Zvonimir Rakamaric and Professor Stephen Siegel, for their heavy contributions in my formal methods education, personalized mentoring and valuable feedback.

Peng Li, PhD student in the Gauss Group for his work on GKLEE (a GPU verification tool) which got me starting working on GPU verification topics. His intensive GPU knowledge and willingness to answer questions was a major factor in both the inception and development of this work.

Geof Sawaya, RE in the Gauss Group whose consistent quality (and quantity) of work and good attitude will always be an inspiration to me.

Kathryn Rodgers, Colleague, for her corrections and comments without which this document would not have passed a middle school English class.

All the Gauss Group members for providing me with a stimulating environment and for forcing me to work harder than I ever have in my life trying to reach the precedence they have set.

Lastly, and not exclusive from the aforementioned, my friends for their generous support throughout my education and consistent reminders that life is meant to be enjoyed.

CHAPTER 1

INTRODUCTION

A Graphics Processing Unit (GPU) is a microprocessor designed with many cores and high data bandwidth [15, p.3-5]. These devices were originally used for graphics acceleration, particularly in 3D games; however, the high arithmetic throughput of these microprocessors had potential to be used in other applications. In late 2006, NVIDIA released the first GPU that supported the CUDA architecture [27, p.6]. CUDA allowed programmers to develop general purpose code to run on a GPU. Since then the use of GPUs has and is continuing to grow in many aspects of modern computing. These devices have been used in many applications including medical imaging [29], radiation modeling [10] and molecular simulations [6]. Current research is showing innovative new algorithms for GPUs efficiently solving fundamental problems in computer science e.g. Merrill et. al. recently published an optimized graph traversal algorithm using GPUs[21]. The most recent results of the TOP500 project, which ranks and documents the current most powerful 500 computers[1], states that a total of 62 the computers on the list are using accelerators or co-processor technology. Furthermore, the number 1 and 8 ranked computers have NVIDIA GPU co-processors. Statistics on the number of papers published from a popular research hub `www.hgpu.org` also shows a rising trend in GPU popularity. Less than 600 papers were published in 2009 describing applications developed for GPUs; in 2010 this rose to 1000 papers and both 2011 and 2012 saw over 1200 papers.

GPUs are highly concurrent devices that allow multiple processors to share data and as such, they are prone to the same notorious concurrency memory issues as their traditional

[1] see `http://www.top500.org`

CPU counterparts. Concurrency bugs, such as data races and deadlocks, can be very difficult to detect due to the non-deterministic execution of processes, that is, a bug may appear in one run and not in another even with the exact same input data [22]. In some cases these concurrency bugs have gone completely undetected until deployment and have caused substantial damage in real world systems [30, 13, 18]. One source of non-deterministic concurrency bugs is a misunderstanding of the architecture's *shared memory consistency model* i.e. what values can be read from shared memory when issued concurrently with other reads and writes [28, p.1]. A programmer may expect that every concurrent execution (even if a data race is present) to be equivalent to a sequential interleaving of the concurrent instructions, a property known as *sequential consistency* [16] (see Figure 1.1). This, however, is not always the case as many modern architectures *relax* sequential consistency for performance gains [9] and the underlying hardware is allowed to re-order and execute certain instructions non-deterministically according to the particular architecture's shared memory consistency model. The Power architecture [11] is one such example. Furthermore, data written to shared memory by a process `p0` is not always guaranteed to be visible to a different process `p1` immediately. To alleviate some of these issues, many architectures provide *memory barrier* instructions which can guarantee certain visibility and ordering properties [2]. If a programmer is to avoid costly and elusive concurrency bugs, he or she must understand the architecture's shared memory consistency model and the guarantees (or lack thereof) provided.

Shared memory consistency models for traditional CPUs have been well studied over the years [28] and continues to be an area of rich research. However, GPUs have a memory hierarchy system and concurrent model that is substantially different from that of a traditional CPU. For example, in NVIDIA GPUs a thread[2] can access two different sections of memory, *shared* and *global*. All threads on the GPU can read and write to the same global memory while only certain other threads in the GPU read and write to the same shared memory (see Section 2.1 for more details) [15, p.77]. In fact the PTX (Parallel

[2]Both threads and processes refer to an independent sequence of execution, however threads run in some type of shared memory environment.

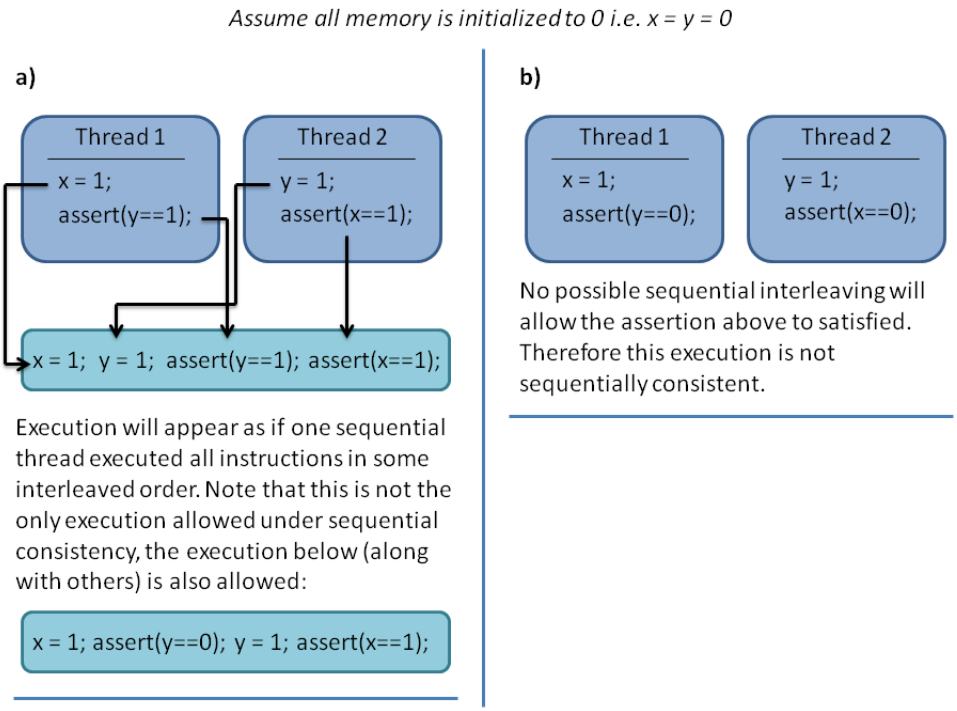

Figure 1.1: (a) Illustration of sequentially consistent executions and (b) a non sequentially consistent execution.

Thread eXecution) ISA from NVIDIA defines four different memory state spaces that can be read from. Out of these state spaces, two are thread local, one is global and one is shared within a *block* i.e. only shared between a few specific threads [25, p.127]. This is a noticeable departure from CPU memory models where fewer memory spaces are involved. Furthermore, the memory barrier instructions for NVIDIA GPUs have different guarantees for the different types of memory and threads [24, p.88]. Despite these differences, two of the most popular GPU frameworks, CUDA [23] and OpenCL [14], do not provide detailed memory model documentation leaving GPU memory consistency largely unstudied.

This work presents a variety of *litmus tests* i.e. short executions which are allowed, disallowed or guaranteed, that we believe illustrate the behavior of two of the CUDA memory fence types across shared and global memory on NVIDIA GPUs. While other GPU vendors exist, NVIDIA was chosen due to its dominant usage, abundance of examples, and

valuable contributions from industry contacts. For the remainder of this paper, GPU will refer to an NVIDIA GPU. These litmus tests illustrate both *static* and *dynamic* properties of fences as well as classical memory consistency properties such as *coherence*. Then two operational memory consistency models for GPUs are presented which are consistent with previously discussed litmus test. The two models deviate on some controversial behavior; for example, the first model maintains classical coherence properties [28] while the second model exhibits more relaxed behavior based on industry feedback. These models were implemented in the Murphi modeling language [7] and formally describe interactions of loads, stores, and memory fences, which lack detailed documentation in existing CUDA literature [24], across both types of memory. These Murphi models are made publicly available. As a preliminary study, our model contains a subset of the real instruction set (see Table 1.1 for modeled instructions), yet we are able to show interesting properties between them. Note that this work attempts to describe the *hardware* consistency model which pertains to how the hardware manages instructions; the *software* consistency model dictates how the compiler can manipulate instructions. Other than a motivating example in Section 5.3 the software consistency model is not addressed in this work. Finally, to emphasize the need for a formal GPU memory model, this work shows real world examples of incorrect (with respect to our model) and poorly defined CUDA code [21, 5]. As a final introductory note, we would like to emphasize that the model presented in this paper, while developed in collaboration with NVIDIA, is not an official NVIDIA endorsed model. None of the other models developed for traditional CPUs have been declared "official" by their respective vendors for various reasons beyond the scope this paper. It is our hope that

Table 1.1: Table of modeled instructions.

Instruction	Description
`ld.global`	Loads a value from the global memory space
`ld.shared`	Loads a value from the shared memory space
`st.global`	Stores a value to the global memory space
`st.shared`	Stores a value to the shared memory space
`thread_fence`	To be discussed
`thread_fence_block`	To be discussed

our work will aid GPU memory consistency efforts to provide a more complete model that both developers and manufacturers could agree on. Perhaps one way to view this work is: *what is a reasonable CUDA memory model that passes the litmus tests we present?*

1.1 Related Work

Using litmus tests are an intuitive way to understand memory consistency models and are used in official industry documentation [31]. Litmus tests have also been studied formally [20] and have been shown to describe important properties of memory systems such as model equivalence. More specifically, several concise litmus tests are used to show a great deal about the consistency model. For example, the classic **IRIW** [3] (see Figure 1.2) test can show *write atomicity* violation, that is, if this execution is allowed, then threads can see updates to different addresses in different orders and there is no global total order of stores. `Thread3` will see the memory at location `x` updated before the memory at location `y`, while `Thread4` will see the opposite memory update order. One can see the extra attention that developing in an architecture that allows this behavior would need. Additionally, simple *causality* tests are shown that describe how causal relationship can be determined between threads. That is, if `Thread1` writes a new value to address `a` and `Thread2` reads the new value from `a` then we know that `Thread1`'s write caused thread `Thread2`'s read. These causal relationships are important when reasoning about litmus tests as they can provide some execution order between threads. We use similar litmus tests to show properties of our GPU operational models. Murphi has been used in the past to model concurrent shared memory consistency protocols. As a notable example, Murphi was used in verification studies of the Cray SV2 cache coherence protocol [1]. More specifically, the Murphi model was used to verify *coherence*, *forward progress*, and *exclusivity* of this protocol. We discuss coherence in depth in 3.2 and similarly use our models to reason about coherence properties of GPUs. Murphi was also used in validating a complex cache protocol used in Intel processors [17]. Between these two studies, Murphi is shown to be an effective tool in reasoning about memory systems.

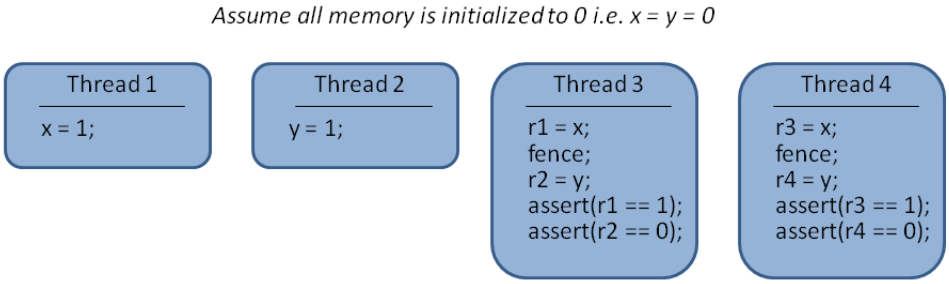

Figure 1.2: Classic IRIW litmus test. If the assert can be true then this shows a write atomicity violation.

Alglave et. al. present a thorough study on fences in weak memory models [2] which formally defines *static*, *dynamic* and *cumulative* properties of memory fences which we refer to in our work. We borrow the idea representing litmus tests as graphs with cycles showing violations of properties in Section 3.2. While their work presents axiomatic consistency models as opposed to the operations memory consistency models presented in this work, our work shares a similar goal in showing the complex interactions of memory fences on weak memory models. Finally, their work uses the Power architecture as its main focus and does not mention GPUs, which is our contribution.

1.2 Roadmap

The presentation of this work is organized as follows. Chapter 2 presents the required background necessary for the healthy digestion of the rest of this document. This includes a primer on the GPU concurrency and memory system and some required prerequisites on shared memory consistency models and litmus tests. We also give some background on memory fences and their properties as well as operational semantics. The next two chapters present our model in the top-down order that it was developed i.e. we explicitly state behaviors we would like to see in the form of litmus tests, given in Chapter 3, and build our model to satisfy those tests, given in Chapter 4. Some behaviors of the model have yet to be resolved, which we take in consideration in both chapters. The operational semantics of our model are given in detail in Chapter 4 along with a discussion of the Murphi

implementation. Real-world examples of code that make memory consistency assumptions that are not in line with our model are shown in Chapter 5 along with the fixes that make the code consistent with our model. We end with a conclusion which summarizes this document and discusses future work.

CHAPTER 2

BACKGROUND

2.1 GPU Basics

This is a very basic overview of the GPU architecture. We only consider one dimensional blocks and grids and restrict memory to global and shared. More complete treatments of this material are available [27, 15] for the motivated reader.

GPUs have a fundamentally different architecture than traditional CPUs including complex memory and concurrency systems. In Flynn's taxonomy [8], a GPU has a SIMD (single instruction multiple data) architecture, that is, many processing units execute the same instruction stream on different data. A thread in a GPU executes a shared instruction stream on a processing unit and has access to a `thread_id` variable to facilitate accessing its own data. In CUDA, the C level interface to program NVIDA GPUs, this variable can be given by `threadIdx.x`. For example, a CUDA program to add two integer vectors $z = x + y$ might look like:

```
__global__ void add_vectors(int *x, int *y, int *z) {
   z[threadIdx.x] = x[threadIdx.x] + y[threadIdx.x];
}
```

Figure 2.1 shows a graphical representation of the execution of the `add_vectors` routine on a GPU.

One important aspect of concurrent programming is how different threads can work together to accomplish a task. This involves synchronization and shared memory between threads. To this end, GPU threads are split into *blocks*. Each block contains the same

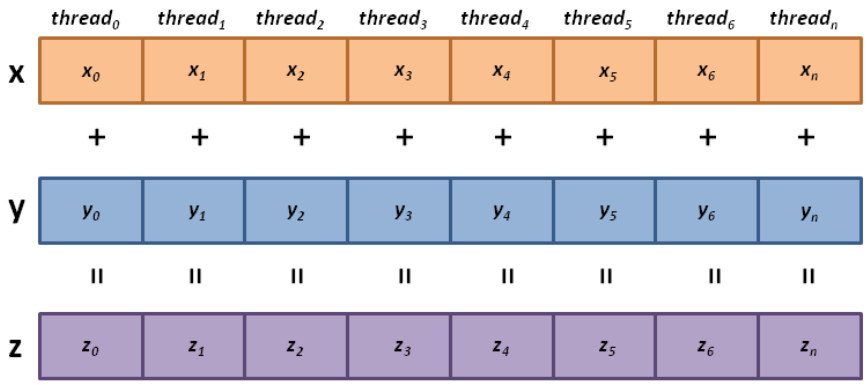

Figure 2.1: Visual representation of how a vector addition routine might happen on a GPU.

number of threads and thus satisfies the equation : $blocks * block\_size = total\_threads$. The block size and number of blocks[1], can be adjusted at launch time. Threads within the same block can share certain types of memory and synchronize with each other. The `block_id` variable can be accessed in CUDA by the variable `blockIdx.x`. Only threads within the same block are given a synchronization barrier which can be invoked by the CUDA command `__syncthreads()`. GPUs have several memory spaces but this work is restricted to *global* and *shared* memory. Global memory is accessible to all threads on the GPU and is persistent between GPU launches from within the same program. On the other hand, shared memory is only shared between threads in the same block i.e. each block has its own shared memory. Shared memory is not persistent between GPU launches. As one might suspect, global memory is larger but slower than shared memory. Global memory is dynamically allocated and initialized on the CPU before the GPU launch while shared memory is declared in the GPU code using the CUDA keyword `__shared__`. Finally a graphical representation of the discussed features of the GPU is given in Figure 2.2.

[1] often called the *grid* size.

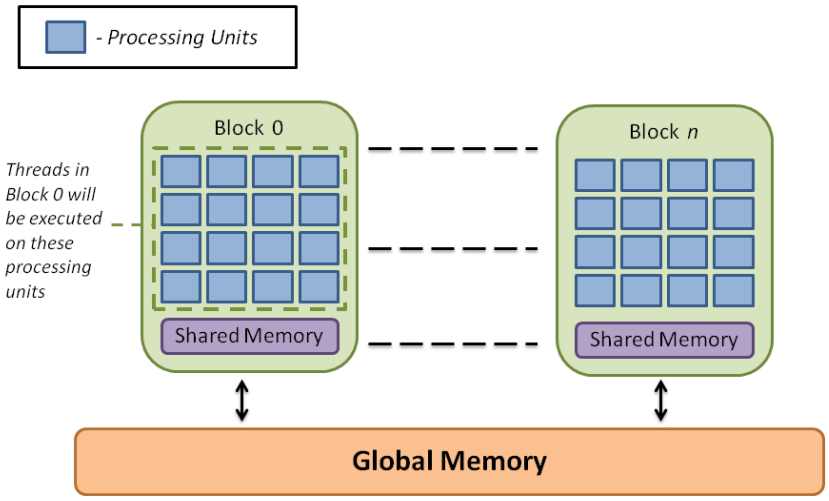

Figure 2.2: Graphical representation of the relevant features of a GPU.

2.2 Memory Consistency Models and Litmus Tests

Shared memory consistency models (or memory models for short) for a particular architecture define the correct behavior of shared memory behavior without explicitly mentioning the physical cache or other layers of memory. Traditionally there have been two ways to do this:

1. **Axiomatic Models**: These models define relationships between *actions* and are normally drawn as graph-like structures with the relationships as edges and actions as nodes. Actions are usually memory operations such as load, write, or memory fence. Examples of relationships are program order, data dependency etc. Other than a brief mention of a structure obtained by a prior model (see Figure 3.2) we will not discuss axiomatic models in this work.

2. **Operational Models**: These models define behavior in terms of the *operation* of the system. That is, given the current state of the system and, optionally, an operation, e.g. `load`, the operational model will provide all possible steps the system could take. These steps are known as operational semantics and they are more formally defined in Section 2.4

If one memory model `M1` imposes fewer constraints than another `M2`, we say `M1` is *weaker* `M2`. Memory models become weaker by removing constraints, which is known as *relaxing* the model. A model that is strictly stronger than another will only allow a subset of the legal executions of the weaker model. The strongest memory model is sequential consistency (see Figure 1.1) which only allows executions which can be represented as a sequential interleaving of instructions.

While providing precise operational or axiomatic models yield unambiguous mathematical structures, an intuitive way to understand a given memory model is through litmus tests. Litmus tests are short programs evaluated under a memory model and usually contain assertions about values read from memory. Litmus tests can either be guaranteed (meaning all assertions pass), disallowed (at least one assertion will always fail), or, given the non-determinism of concurrent systems, allowed but not guaranteed. As mentioned in the introduction (see Section 1.1) litmus tests have been widely studied and can show interesting properties of the memory model. We use litmus tests as our main method of specifying the high level behavior of our model.

2.3 Memory Fences

A memory fence is a low level instruction that enforces ordering on memory instructions. As mentioned in Chapter 1 many modern architecture support out-of-order execution i.e. memory instructions can be reordered for performance gains. Memory fence instructions can be inserted to combat this non-intuitive behavior and restore orderings. Memory fences have been well studied [2] and have two distinct properties:

1. **Static Properties**: These properties enforce program order within one thread of execution. For example, these properties would decide if instructions before the memory fence are allowed to be reordered after the memory fence or similarly, if instructions after the fence can be reordered before.

2. **Dynamic Properties**: These properties enforce memory visibility across threads. Informally, these properties dictate if the writes to shared memory prior to the fence must be visible to other threads after the execution of the fence.

Fences may also be *cumulative*, a dynamic property which enforces visibility of not only the values written by the calling thread but also the values written by other threads currently visible to the calling thread. Consider the classic **IRIW** test in Figure 1.2. In `Thread3` and `Thread4` the static properties of the fence are enforcing the two loads to happen in program order. If the fence has cumulative dynamic properties, then the assertions can never be satisfied by the following argument: w.l.o.g. one of the fences must execute before the other; say `Thread3`'s fence executes first and say `r1 == 1` (from which we know `x == 1`). Then the memory fence will enforce the visibility of x to `Thread4` even though `Thread3` did not write it, thus making the assertions unsatisfiable.

Memory fences are the best way to enforce orderings in architectures with weak memory models. Understanding the subtle behaviors enforced by properties is crucial to writing correct code; furthermore, by reasoning about fences in terms of static and dynamic properties we can gain insights into the entire shared memory consistency model.

2.3.1 CUDA Fences

CUDA provides three memory fences, documented [24] as follows:

- `__threadfence_block()` *waits until all global and shared memory accesses made by the calling thread prior to* `__threadfence_block()` *are visible to all threads in the thread block.*
- `__threadfence()` *waits until all global and shared memory accesses made by the calling thread prior to* `__threadfence()` *are visible to: All threads in the thread block for shared memory accesses, All threads in the device for global memory accesses.*
- `__threadfence_system()` *waits until all global and shared memory accesses made by the calling thread prior to* `__threadfence_system()` *are visible to: All threads in the thread block for shared memory accesses, All threads in the device for global memory accesses, Host threads for page-locked host memory accesses (see Section 3.2.4.3).*

While some high level dynamic properties are discussed, this documentation fails to mention any static or cumulative properties of these fences. This work aims to formally model and thus capture the behavior of `__threadfence_block()` and `__threadfence()` along with all of their properties.

2.4 Operational Semantic Models and Implementation

Operational semantic models describe systems in a rigorous, unambiguous manner using mathematical notation. A state of the system (denoted by σ) is described by a valid assignment to all the global variables in the system e.g. to refer to global variable x in a state σ we say $\sigma.x$. We limit our discussion of operation semantic models to *structural* operational semantic models [26]. This semantic model describes the system by formally defining each possible atomic step of the system. The notation generally used for an atomic step is a horizontal bar. On the top of the bar is a predicate $P : X \rightarrow \{true, false\}$ where $X \subseteq S$ where S is the global variables of the system. Underneath the bar is a transition $\sigma \rightarrow \sigma'$ where σ is the current state and σ' is a new state which may contain different assignments to system variables. It is also common notation for $\sigma[(z := v)]$ where $z \in S$ and $v \in Domain(z)$ to mean the state σ with variable z updated to the value of v. As an example consider a system with x and y as global variables and say the system increments x by 2 and y by 1 while $x < 50$. The operational semantics for this behavior would be:

$$\frac{\sigma.x < 50}{\sigma \rightarrow \sigma[x := x + 2, y := y + 1]} \tag{2.1}$$

A complete operational semantic model will consist of many atomic steps. If the predicate on top of the bar is true in a given state i.e. $P : \sigma \rightarrow true$ then we say that transition is *enabled*. If a transition is enabled, then the state update underneath the bar is allowed to happen and the system moves to state σ'. Note that it is possible for multiple transitions to be enabled in a single state; this leads to a non-deterministic system i.e. even if given the exact same starting state, an execution may give different intermediate or ending states. Given the non-deterministic nature of concurrent code, structural operational semantic models are often developed to reason about these types of systems.

2.4.1 Murphi Implementation

The Murphi modeling language [7] provides intuitive syntax and powerful techniques for implementing and reasoning about non-deterministic structural operational semantic models. A Murphi Model consists of four main sections:

1. **System Variables**: This is where all the global variables of the system are declared and defined. Murphi supports integer and boolean data types as well as arrays and records of multiple data. Functions may also be defined which are executed atomically and can modify system variables.

2. **Start State**: This initializes all the system variables to some starting value.

3. **Guarded Transitions**: This describes the behavior of the model and is a direct mapping of structural operational semantic atomic steps. The general form of these guarded transitions is:

   ```
   <expr> ==> <expr>*
   ```

   where the right hand side `expr` should be a predicate on a subset of the system variables and the left hand side `expr` should be modifying system variables to reflect a new state. For example, using the Pascal-like syntax of Murphi, the formal transition in equation (2.1) would be implemented as:

   ```
   x < 50 ==> x := x + 2; y := y + 1;
   ```

4. **Invariants**: These are predicates that must remain true in all states of the system.

When given a model and executed, Murphi will explore all possible states and executions, even in non-deterministic models, and check invariants. If an invariant ever becomes false during an execution, Murphi will report the error along with an execution trace. In short, Murphi is a tool for implementing and reasoning about structured operational semantic models. Section 4.2 gives the behavior of the GPU consistency model in the syntax of Murphi guarded transitions; however, it should be clear that these guarded transitions are a simple implementation of mathematical operational semantics.

CHAPTER 3

LITMUS TESTS AND PROPERTIES

This chapter presents a series of litmus tests that we believe the GPU shared memory consistency model should adhere to. While extensive testing on actual hardware is still in a preliminary state, we believe the GPU memory model should satisfy the following tests by one or more of the following criteria:

- Desired properties studied with respect to traditional CPU memory models.
- The limited documentation provided (see Section 2.3.1).
- High level discussion with contacts at NVIDIA.
- Limited testing on hardware.

Once the high level behavior of the GPU memory model is described through litmus tests we proceed in Chapter 4 to define an operational semantic memory model that satisfies these litmus tests. In a way, we are attempting to reverse engineer the GPU memory model, starting with the behavior we hope to see and concluding with the actual model. This approach produces a memory model built around the programmers perspective and thus is more amiable to developers. Finally, some conflicting views on the outcome of several of the litmus tests arose during the course of this work. The implications of these conflicts yield additional discussion and have a dramatic effect on the resulting model.

3.1 Notation

Because we attempt to keep the notation as clean and intuitive as possible, we will not write the litmus tests in the NVIDIA PTX [25] language as would be required to run the

tests on hardware, but rather in a higher level more intuitive language. Furthermore, many interesting properties and behaviors can be seen using small litmus tests consisting of only several threads, memory locations, and values which allows us to restrict their domains to a readable format. In each test it will be noted which memory space addresses belong to and which block threads belong to. Occasionally these details are kept generic and a case analysis is presented examining the different possibilities. The notations kept consistent across tests are:

- **Registers**: Each thread has its own registers and cannot read or write from any other thread's registers. They are labeled rx where x is an integer.
- **Values**: For simplicity, values that can be stored in registers and memory locations are integers.
- **Instructions**: As noted in Table 1.1, we model a small subset of the actual PTX instruction set. A store instruction is given by $x = y$ where x is a memory location and y is a value. A load instruction is given by $x = y$ where x is a register and y is a memory location. We restrict our study to only two of the CUDA memory fences, `__threadfence_block()` which is abbreviated as `TFB()` and `__threadfence()` which is abbreviated as `TF()`. Finally to establish a causal relationship between threads we use c-like if statements.
- **Assertions**: Assertions take a Boolean expression as an argument and pass if, and only if, the Boolean expression evaluates to true. If it is possible for an assertion to fail, then the entire litmus test fails.

3.2 Coherence

Coherence is a property studied in traditional CPU memory consistency models that aims to make memory caching operationally invisible to the programmer. That is, a programmer should not be able to determine which memory values were cached based on the results of load and store operations. The threads and memory in the coherence litmus tests have been left generic as coherence is a desired property in any shared memory situation

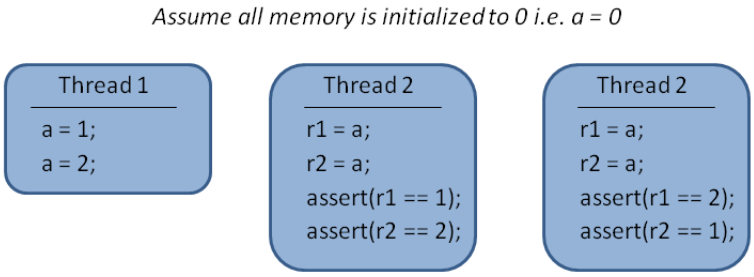

Figure 3.1: Litmus test that can test for coherence violation.

and thus we assert that it must be maintained between all memory spaces that threads share. It has been shown that this property is equivalent to sequential consistency with respect to *a single address* [28, p.14]. To this end we propose a *classical coherence* litmus test (see Figure 3.1). We assert that this litmus test must *always* fail if classical coherence is to hold as there is no sequentialization of these instructions that satisfy the assertions. As further evidence that this test violates classical coherence, we use an equivalent relational coherence definition, given by Alglave et. al. [2] but not explicitly shown here, to further show that this litmus test does indeed test for coherence violations. Again, we do not formally go into how the graph is constructed, but rather it is shown in support for our litmus test as we encountered some disagreement about the classical definition of coherence. A violating cycle in the relations graph (see Figure 3.2) substantiates our test.

3.2.1 Weak Coherence

While the above definition of coherence seems to be prevalent in the literature [28, 2], discussion with industry experts revealed that some architectures allow load operations from the same address to be reordered which violates the classical coherence litmus test. To this end we developed a notion of *weak coherence* which we define as:

Weak Coherence : sequential consistency with respect to a single address if loads within a thread are separated (and thus ordered) by a memory fence.

Figure 3.3 shows a litmus test similar to the classical coherence test which tests for weak coherence violations. Notice that fences are inserted for their static property of keeping the loads in program order.

3.3 Simple Visibility

We now turn our attention to simple dynamic properties of the GPU fences, i.e. what should other threads expect to see in memory after a thread executes a memory fence. The strategy here is to establish a causal relationship between threads so one thread knows when the other has issued a memory fence. We do this by having one thread issue a fence followed by a write. Now a separate thread observes the memory location of the first thread's final write. If the observing thread sees the write issued by the first thread, we know the memory fence and, by the static properties of fences, all instructions before the memory fence in the first thread have been executed (see Figure 3.4). Since our study restricts to GPU memory space into only two sections (shared and global) we consider each space separately and then bring them together with a larger test. The behavior in the following tests follows directly from the limited documentation of the fences and has been confirmed by industry experts.

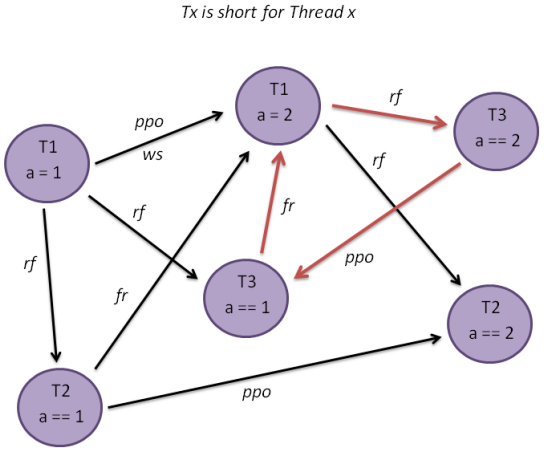

Figure 3.2: Graphical representation of coherence litmus test, a violating cycle is emphasized.

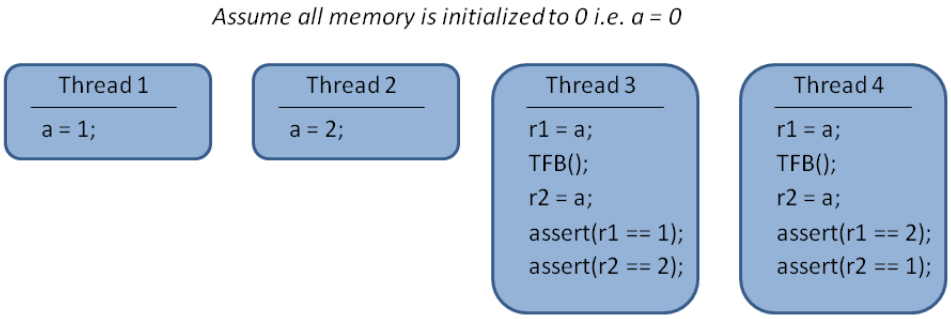

Figure 3.3: Litmus test that can test for weak coherence violation.

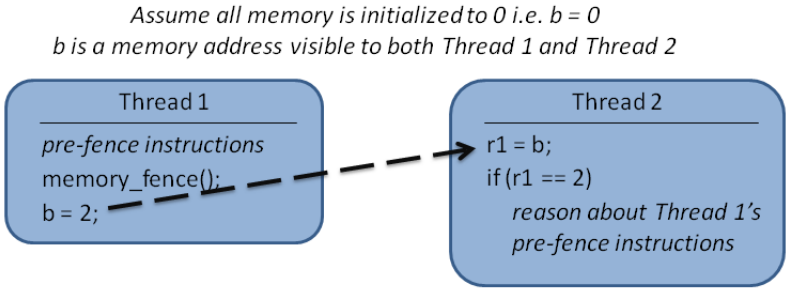

Figure 3.4: Establishing a causal relationship between threads.

3.3.1 Shared Memory

Recall that threads inside the same block have a shared memory space known as *shared* memory. We propose a litmus test in Figure 3.5 to show simple visibility across shared memory for threads within the same block. The memory fence in `Thread2` is a `TFB()` for its static properties only, that is, to ensure the loads remain in program order. The memory fence in `Thread1` is left unspecified so we can analyze the test under the different fences. If the memory fence is a:

- `No-op`: No guarantees are made and the assertions may pass or fail.
- `__threadfence_block()`: The assertions in this litmus must always pass.
- `__threadfence()`: The assertions in this litmus must always pass.

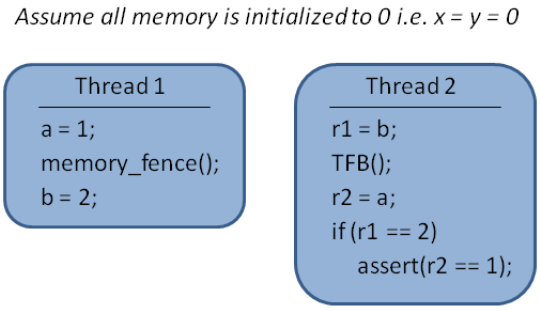

Figure 3.5: Litmus test establishing dynamic properties of GPU fences.

3.3.2 Global Memory

We can use the exact same test as seen in Figure 3.5 for global memory, however, since all threads on the device share global memory, we must consider the separate cases when `t0` and `t1` are in the same block and when they are not. We show a similar fence case analysis as before for both.

- *T0 and T1 are in the same block:*
  - `No-op`: No guarantees are made and the assertions may pass or fail.
  - `__threadfence_block()`: The assertions in this litmus must always pass.
  - `__threadfence()`: The assertions in this litmus must always pass.
- *T0 and T1 are not in the same block:*
  - `No-op`: No guarantees are made and the assertions may pass or fail.
  - `__threadfence_block()`: No guarantees are made and the assertions may pass or fail.
  - `__threadfence()`: The assertions in this litmus must always pass.

3.3.3 Putting it together

When creating a test that combines `__threadfence()` and `__threadfence_block()` we run into an interesting issue which leads us to our next section. Consider the test in

Figure 3.6. It is fairly easy to follow the causal relations between threads, but should this assertion be guaranteed? That is, does `Thread2`'s `__threadfence()` apply to `thread1`'s stores that are visible to `Thread2`? Or, another way this assertion could be satisified is if memory updates have a global order and `Thread2`'s visibility of `Thread1`'s stores implies all threads see `Thread1`'s stores. This is a source of conflict in discussion and is discussed next.

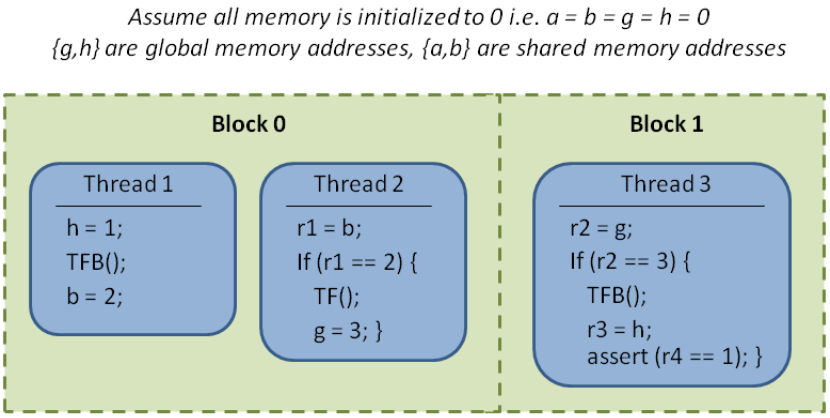

Figure 3.6: Large litmus test combining both GPU fences.

3.4 Write Atomicity and Cumulative Fences

Write atomicity was briefly discussed and shown in Figure 1.2 with the classical IRIW litmus test which we reference now. Write atomicity requires that one thread's store is visible to all other threads at the same time. It is clear that in a system supporting write atomicity this test cannot pass as one of the stores to `x` or `y` will become visible first and the other threads cannot see them out of order. Another factor that dictates whether or not an architecture passes is if the fences are cumulative or not. Recall that cumulative fences not only guarantee the visibility of their own stores, but also the visibility of other stores visible to them. *If* GPU fences are indeed cumulative, we maintain that `__threadfence()` and `__threadfence_block()` maintain their same strength as shown in

the previous section. That is, `__threadfence()` acts both within and across blocks while `__threadfence_block()` only acts within blocks but on both global and shared memory. That is, assuming cumulative fences, the IRIW test in Figure 1.2 cannot pass if the memory fence is a `__threadfence_block()` and the threads all belong to the same block regardless of shared or global memory. If the threads are not in the same block, then the memory must be global and the fence must be a `__threadfence()` to ensure the assertion never passes. If the fences are not cumulative then there are no guarantees about if the assertions will pass or fail. As mentioned earlier, there are conflicting views if the GPU fences are cumulative or not.

As a final note, notice that the above tests suggest that `__threadfence_block()` is strictly weaker than `__threadfence()`. That is, executions allowed using `__threadfence()` are a subset of the executions allowed using `__threadfence()_block()`.

CHAPTER 4

GPU OPERATIONAL SEMANTIC MODELS

In this section we present our shared memory consistency model that satisfies the desired behaviors shown by the litmus tests in the previous chapter. We give a visual description of the data structures in the model and then formally present the transition rules in Murphi syntax. We briefly show an alternative model which has different behavior with respect to some of the unresolved issues discussed in the previous chapter. We conclude the chapter by discussing the Murphi implementation and the results of model checking the litmus tests.

4.1 Visual Description and Data Structures of Model

A visual description of the model is given in Figure 4.1 which will be helpful for understanding the operational semantics of the model. We begin by discussing the data structures used in the model. Each thread has its own

- Program (an array of instructions) along with a program counter. These are given in the model by: `PGM : array[blocks] of array[threads] of array [program_length] of instructions` or a three dimensional array indexed by block, thread and program counter. Similarly the program counter is given by a two dimensional array indexed by block and thread: `PC : array[blocks] of array[threads] of program_length;`
- Set of registers given in the model by: `REG : array[blocks] of array[threads] of array [registers] of data`. Again, a three dimensional array indexed by block, thread, and register number.

- Copy of shared and global memory [1] which are simply arrays indexed by addresses given in the model by: `SM : array[blocks] of array[threads] of smem` and `GM : array[blocks] of array[threads] of gmem` respectively. `smem` and `gmem` are simply data locations with the additional flags as seen in the figure and discussed in the next list.

- Set of unissued loads (called the load pool) given in the model by: `LP : array[blocks] of array[threads] of load_instruction`.

- Instruction queues for each global and shared memory address given in the model by: `(S|G)QA : array[blocks] of array[threads] of array [addrs] of Instructions`. where `SQA` are the shared memory instructions and `GQA` are the global memory instructions.

Each memory location (per thread) also has a set of Boolean flags which are:

- `B` - Borrowed bit

- `LS` - Locally shared

- `GS` - Globally shared (global memory only)

4.2 Annotated Operational Semantics

We now discuss the operational semantics of the model, that is, we describe atomic operations and the guards must be satisfied in order to enable the operation (see Section 2.4). Note that we will employ the Murphi syntax for expressing the guarded transition system. Because of its intuitive nature, we do not give detailed documentation on all the Murphi syntax but any confusion can be resolved with the official Murphi documentation. In the quest for succinctness, the model shown below is slightly simpler than the actual model as the actual model contains additional mechanisms to aid with verification; additionally we do not define all functions and predicates used below but rather give them with sufficient

[1] in the figure `a` and `b` are shared addresses and `g` and `h` are shared addresses.

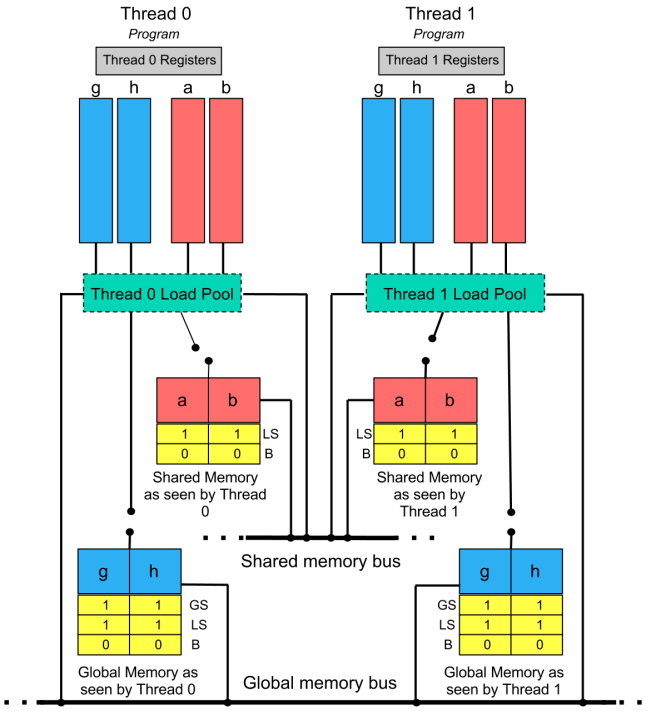

Figure 4.1: Visual description of our shared memory consistency model for GPUs.

names or comments to express their semantics. English summaries of the different sections of system are given to aid in understanding. Intuitively, we begin with the initial state.

For the intial state, all memory locations are initialized to 0. The borrowed flag is set to false and locally and globally shared flags are set to true. Additionally, all program counters (`PC`) and registers are set to 0. Instruction queues and load pools are empty and memory fields are initialized (the previously mentioned flags and data values). Programs can be custom input by the user here (see Section 4.4). This is given in the start state of the Murphi model as:

```
For b:blocks do
  For t:threads do
     PC[b][t] := 0;        -- Program counter
  end;
end;
For b:blocks do
  For t:threads do
    initlp(LP[b][t]); -- initialize load pool to empty.
    For a:addrs do
      initq(SQA[b][t][a]); -- initialize each queue to empty;
      initq(GQA[b][t][a]); -- initialize each queue to empty;
    endfor;
  endfor;
endfor;
--initializing memory, values to 0,shared (not needing to share) and not borrowed.
For b:blocks do
  For t:threads do
    For a:addrs do
      GM[b][t][a].Data  :=  0;
      GM[b][t][a].LS    :=  True;
      GM[b][t][a].GS    :=  True;
      GM[b][t][a].B     :=  False;
      SM[b][t][a].Data  :=  0;
      SM[b][t][a].LS    :=  True;
      SM[b][t][a].B     :=  False;
    end;
  end;
end;
--Insert Custom program and registers here
```

Next we present operations that progress the thread forward in the program, i.e. increment the program counter. A `no-op` simply increments the program counter while loads and stores get stored in the instruction queue of the address they are operating on. Finally fence instructions get enqueued in all of the calling threads' instruction queues. Notice that the rulesets create a rule for each thread in the model (a technique that is used throughout) and `ins` is simply an alias to the current instruction. We use the function `enqueue(queue, value)` to enqueue instructions. Finally notice that instructions have an `Op` field containing what kind of instruction it is, an `addr` field for which address it operates on and a `Global` field indicating if it acts on global or shared memory.

```
ruleset b : blocks do
  ruleset t : threads do
    Alias ins : PGM[b][t][ PC[b][t] ] Do
      Rule "Fire a noop (simply incrpc)"
      --
         ins.Op = no-op  ==> PC[b][t] := PC[b][t] + 1;
      EndRule;
      Rule "Fire a ld from global, store in global queue"
      --
          ins.Op = load  &
          ins.Global     ==>
                                  enqueue(GQA[b][t][ins.addr], ins);
                                  PC[b][t] := PC[b][t] + 1;
      Endrule;
```

```
Rule "Fire a ld from shared, store in shared queue"
--
    ins.Op = load   &
    !ins.Global  ==>
                            enqueue(SQA[b][t][ins.addr], ins);
                            PC[b][t] := PC[b][t] + 1;
Endrule;
Rule "Fire a st from global, store in global queue"
--
    ins.Op = store & ins.Global
                  ==>
                enqueue(GQA[b][t][ins.addr], ins);
                PC[b][t] := PC[b][t] + 1;
Endrule;
Rule "Fire a st from shared, store in shared queue"
--
    ins.Op = store & !ins.Global
                  ==>
                enqueue(SQA[b][t][ins.addr], ins);
                PC[b][t] := PC[b][t] + 1;
Endrule;
```

```
      Rule "Fire a TF."
      --
         (ins.Op = TF)  ==>
                              For a : addrs Do
                                enqueue(SQA[b][t][a], ins);
                                enqueue(GQA[b][t][a], ins);
                              Endfor;
                              PC[b][t] := PC[b][t] + 1;
      Endrule;
      Rule "Fire a TFB."
      --
         (ins.Op = TFB)  ==>
                              For a : addrs Do
                                enqueue(SQA[b][t][a], ins);
                                enqueue(GQA[b][t][a], ins);
                              Endfor;
                              PC[b][t] := PC[b][t] + 1;
      Endrule;
    Endalias;
  endruleset;
endruleset;
```

The next set of operations describes what happens when load and store instructions are dequeued from the instruction queues. Store instructions go out to the thread's memory resetting the borrowed and shared flags while load instructions go through the *load pool*. The load pool can block an address `a` if it contains any operation with `a` as an argument. We define a predicate `lp_not_locking(Load_Pool,address)` which returns true if load pool is not blocking else false. We use a few more additional functions in these rules, namely `dequeue(Queue)` which removes the head of a queue, `head(Queue)` which returns the head of a queue and finally a function which can add an instruction to the load pool

`addtoLoadPool(Load_Pool,instruction)`. Finally the store instructions have a `Data` field containing the value to store.

```
ruleset b : blockT do
  ruleset t : threadT do
    ruleset a : addrT do
      --Rules with the shared memory queues
      Alias Q : SQA[b][t][a] do
        Rule "Drain a store of a shared queue."
        --
          lp_not_blocking(LP[b][t],a) &
          head(Q).Op = store ==>
                                    SM[b][t][a].Data  := head(Q).Data;
                                    SM[b][t][a].B     := False;
                                    SM[b][t][a].LS    := False;
                                    dequeue(Q);
        EndRule;
        Rule "Drain a load of a shared queue."
        --
          head(Q).Op = load ==>
                                    addtoLoadPool(LP[b][t],head(Q));
                                    dequeue(Q);
        EndRule;
      Endalias;
```

```
      Alias Q : GQA[b][t][a] do
      --Rules with the global memory queues
        Rule "Drain a store of a global queue."
        --
          lp_not_blocking(LP[b][t],a) &
          head(Q).Op = store ==>
                                    GM[b][t][a].Data  := head(Q).Data;
                                    GM[b][t][a].B     := False;
                                    GM[b][t][a].LS    := False;
                                    GM[b][t][a].GS    := False;
                                    dequeue(Q);
        EndRule;
        Rule "Drain a load of a global queue."
        --
          head(Q).Op = load ==>
                                    addtoLoadPool(LP[b][t],head(Q));
                                    dequeue(Q);
    Endruleset;
  Endruleset;
Endruleset;
```

Moving on, we discuss the operations dealing with the load pool issuing instructions. Note that the load pool is unordered ad its function is to allow reordering of loads, even from the same address. This supports the litmus test for weak coherence (see Section 3.2.1). We also allow load instruction to *borrow* memory values from other thread's memory if they have not been shared (as indicated by the flag). Once an address has been borrowed, the thread must update it's own memory with the value and not allow borrowing the same address again until another operation overrides the borrowed bit. These memory properties are indicated by the memory flags discussed earlier. Note that the borrowing thread is also not responsible for sharing the value. Furthermore there are different rules

for borrowing global memory intra and inter block. This borrowing mechanism allows for the write atomicity violation behavior. The memory is loaded to the register stored in the `reg` field of the instruction type. Finally the instruction is removed from the load pool with the following function `removeFromLoadPool(Load_Pool,instruction)`.

```
ruleset b : blocks do
  ruleset t : threads do
    ruleset i : loadPool do
      Alias a   : i.addr do
        Alias reg : i.reg do
        --First the cases loading from local memory
          Rule "non-det execute a load from the load pool
          from local global memory"
          --
          i.Global ==>
                                  REG[b][t][reg] := GM[b][t][a].Data;
                                  removeFromLoadPool(LP[b][t], i);
          EndRule;
          Rule "non-det execute a load from the load pool
          from local shared memory"
          --
          !i.Global ==>
                                  REG[b][t][reg] := SM[b][t][a].Data;
                                  removeFromLoadPool(LP[b][t], i);
          EndRule;
```

```
ruleset b1 : blocks do
  ruleset t1 : threads do
    Rule "non-det execute a load from the load pool
    from others global memory out of block"
    --
    i.Global                &
    GM[b][t][a].B = False   &
    b != b1                 &
    !GM[b1][t1][a].GlobalShared
                    ==>
                         REG[b][t][reg]    := GM[b1][t1][a].Data;
                         GM[b][t][a].Data  := GM[b1][t1][a].Data;
                         GM[b][t][a].LS    := True;
                         GM[b][t][a].B     := True;
                         GM[b][t][a].GS    := True;
                         removeFromLoadPool(LP[b][t], i);
    EndRule;
```

```
  Rule "non-det execute a load from the load pool
  from others global memory inside block"
  --
  i.Global                &
  GM[b][t][a].B = False   &
  b = b1 & t != t1        &
  !GM[b1][t1][a].LS
                ==>
                     REG[b][t][reg]    := GM[b1][t1][a].Data;
                     GM[b][t][a].Data  := GM[b1][t1][a].Data;
                     GM[b][t][a].S     := True;
                     GM[b][t][a].B     := True;
                     GM[b][t][a].GS    := True;
                     removeFromLoadPool(LP[b][t], i);
  EndRule;
  Rule "non-det execute a load from the load pool
  from others shared memory inside block"
  --
  !i.Global               &
  SM[b][t][a].B = False   &
  b = b1 & t != t1        &
  !SM[b1][t1][a].LS
                ==>
                     REG[b][t][reg]    := SM[b1][t1][a].Data;
                     SM[b][t][a].Data  := SM[b1][t1][a].Data;
                     SM[b][t][a].S     := True;
                     SM[b][t][a].B     := True;
                     removeFromLoadPool(LP[b][t], i);
EndRule;
```

```
            Endruleset;
          Endruleset;
        EndAlias;
      EndAlias;
    Endruleset;
  Endruleset;
Endruleset;
```

The next set of transitions we discuss are the nondeterministic sharing of memory (if flags dictate memory needs to be shared). Memory is shared per-location to all threads the memory location is visible to i.e. all threads for global memory, intra block threads for shared memory. Shared memory locations reset the memory flags associated with that memory in both the sharing thread and all threads being shared with. It is possible for global memory to only be shared to inter block threads as the weaker fence (`__thread_fence_block()`) only enforces global memory to be shared intra block.

```
ruleset b : blocks do
  ruleset t : threads do
    ruleset a : addrs do
      Rule "share an completely unshared datum from global memory"
      --
      !GM[b][t][a].LS & !GM[b][t][a].GS
                              ==>
                               For b2 : blocks Do
                                 For t2 : threads Do
                                   if b != b2 | t != t2 then
                                     GM[b2][t2][a].Data := GM[b][t][a].Data;
                                     GM[b2][t2][a].LS   := True;
                                     GM[b2][t2][a].GS   := True;
                                     GM[b2][t2][a].B    := False;
                                   endif;
                                 endfor;
                               endfor;
                               GM[b][t][a].GS := True;
                               GM[b][t][a].LS := True;
      Endrule;
```

```
Rule "share a partially shared datum from global memory"
--
GM[b][t][a].LS & !GM[b][t][a].GS  ==>
                        for b2 : blocks Do
                          for t2 : threads Do
                            if b != b2 then
                              GM[b2][t2][a].Data := GM[b][t][a].Data;
                              GM[b2][t2][a].LS   := True;
                              GM[b2][t2][a].GS   := True;
                              GM[b2][t2][a].B    := False;
                            endif;
                          endfor;
                        endfor;
                        GM[b][t][a].GS := True;
                        GM[b][t][a].LS := True;
Endrule;
Rule "share a datum from shared memory to others in the block"
--
!SM[b][t][a].LS & !SM[b][t][a].B
                       ==>
                        for t2 : threads Do
                          if t2 != t then
                            SM[b][t2][a].Data := SM[b][t][a].Data ;
                            SM[b][t2][a].LS   := True;
                            SM[b][t2][a].B    := False;
                          endif;
                        endfor;
                        SM[b][t][a].LS := True;
Endrule;
```

```
   Endruleset;
 Endruleset;
Endruleset;
```

The final set (phew!) of instructions we present is the execution of the memory fences from the instructions queues. In order for a thread to drain a fence, the fence instruction must be at the head of all instruction queues of the calling thread and the load pool must be empty. This ensures the local ordering of instructions or the *static* properties of the fences. The weaker fence, `__thread_fence_block()` or `TFB`, enforces all memory not locally shared to be shared to intra block threads. The stronger fence, `__thread_fence()` or `TF`, enforces the same with the additional constraint that global memory must be shared inter block as well. These are the *dynamic* properties of the fences. We use the function `is_empty(List)` which returns true if the list is empty, false otherwise. Finally we use two extra functions `Merge_Shared(source,dest)` and `Merge_Global(source_mem,dest_mem,source_block,dest_block)` to aid these transitions. These functions merge two sets of memory i.e. based on flags in memory, copy memory from one threads memory to another and reset flags. Due to their importance, we include their definitions before the transitions. The `lsmT` type is a local shared memory, similarly for `lgmT` and global memory. Note that when values in memory are merged, their borrowed and shared flags are overwritten.

```
--First the merging functions
PROCEDURE Merge_Shared(source : lsmT; var dest : lsmT);
--Merge local shared memories source and dest
begin
  For a  : addrs Do
    if !source[a].LS
    then
      dest[a].Data  := source[a].Data;
      dest[a].LS    := True;
      dest[a].B     := False;
    endif;
  endfor;
end;
PROCEDURE Merge_Global(source : lgmT; var dest : lgmT;
bsource : block; bdest : block);
--Merge global memories source and dest
begin
  For a  : addrT Do
     if (!source[a].LS) |
        ((!source[a].GS) & (bdest != bsource))
     then
       dest[a].Data  := source[a].Data;
       dest[a].LS    := True;
       dest[a].GS    := True;
       dest[a].B     := False;
     endif;
  endfor;
end;
```

```
--Now the transition rules
ruleset b : blocks do
  ruleset t : threads do
    Rule "Drain TFB fence."
    --
    (For a : addrs Do
      head(SQA[b][t][a]).Op = TFB    &
      head(GQA[b][t][a]).Op = TFB
      Endforall)                     &
      is_emptyp(LP[b][t])  ==>
                           --atomically dequeue all
                           For a : addrs Do
                             dequeue(SQA[b][t][a]);
                             dequeue(GQA[b][t][a]);
                           Endfor;
                           --Share memory within block
                           For t2 : threads Do
                             if t != t2 then
                               Merge_Shared(SM[b][t], SM[b][t2]);
                               Merge_Global(GM[b][t], GM[b][t2], b, b);
                             endif;
                           endfor;
                           --Mark as merged
                           For a : addrs Do
                             SM[b][t].LS := True;
                             GM[b][t].LS := True;
                           endfor;

    Endrule;
```

```
Rule "Drain all TF fences when together."
--
  (For a : addrs Do
      head(SQA[b][t][a]).Op = TF   &
      head(GQA[b][t][a]).Op = TF
   Endforall)                      &
   is_emptyp(LP[b][t])  ==>
                       For a : addrs Do
                         dequeue(SQA[b][t][a]);
                         dequeue(GQA[b][t][a]);
                       Endfor;
                       --Share globally
                       For t2 : threads Do
                         if t != t2 then
                           Merge_Shared(SM[b][t], SM[b][t2]);
                         endif;
                         For b2 : blocks Do
                           if t != t2 | b != b2 then
                             Merge_Global(GM[b][t], GM[b2][t2], b, b2);
                           endif;
                         endfor;
                       endfor;
                       --Mark as merged
                       For a : addrs Do
                         SM[b][t].LS := True;
                         GM[b][t].LS := True;
                         GM[b][t].GS := True;
                       endfor;
Endrule;
```

```
  Endruleset;
Endruleset;
```

Finally, as the careful reader may have noticed, not all combinations of flags on memory are possible. In fact only four combinations of the global memory bits and three combinations on shared memory. The combinations and meanings are listed in Table 4.1.

Table 4.1: All bit combinations and their meanings.

Global Memory			
B	**LS**	**GS**	**Meaning**
0	0	0	The memory location has not been shared at all and needs to be shared both locally and globally.
0	1	0	The memory location has been shared intra block and still needs to be shared inter block
0	1	1	The memory location does not need to be shared and has not been borrowed (the initial state)
1	1	1	The memory location has been borrowed and as such the owner thread does not have to share it
Shared Memory			
B	**LS**	**GS**	**Meaning**
0	0	NA	The memory location has not been shared yet
0	1	NA	The memory location has not been borrowed and does not need to be shared (initial state)
1	1	NA	The memory location has been borrowed and as such the owner thread does not have to share it

4.3 Alternative Model

Because of the unresolved behavior pertaining to the results of some of the litmus tests, we designed an alternative model which is strictly stronger than the model presented above. We will refer to the new model as the *strong* model and the previous model as the *weak* model. We will simply present a visual description of the strong model (see Figure 4.2) and note some of the major differences. Notice that each thread does not have its own view of memory, rather each block shares its own view of global and shared memory (this

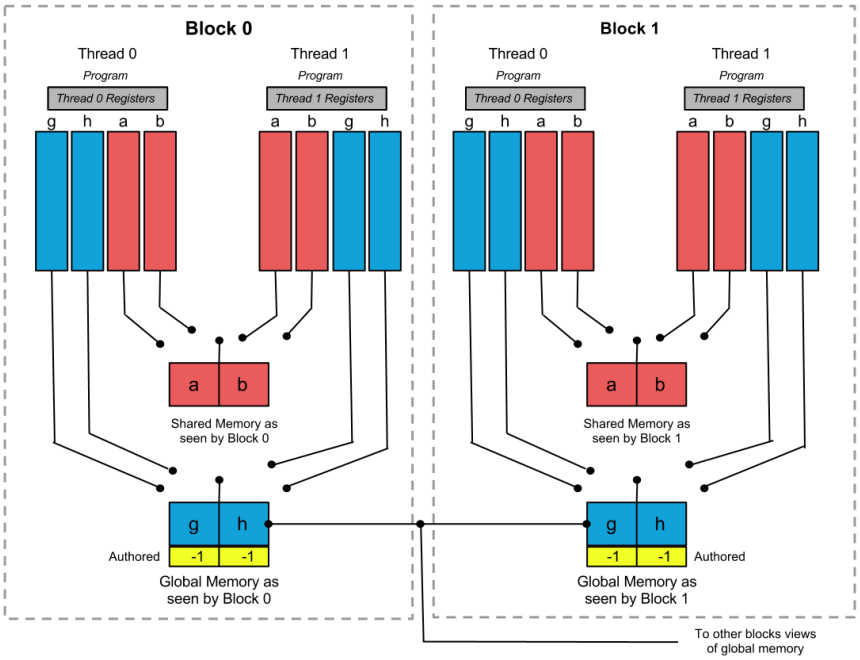

Figure 4.2: Visual description of our shared memory consistency model for GPUs.

enforces write atomicity). Also note the lack of a load pool and only one flag per memory location which leads to a simpler memory sharing system. The behavioral differences are discussed in the next section as the results of the Murphi model checking. We believe that regardless of resolution of the unresolved behavior, we will be able to combine mechanisms from each model to get the desired properties.

4.4 Murphi Model

The models described above were implemented in the Murphi modeling language and can be used to check the executions of litmus tests described in the previous chapter. This is done by initializing each threads program array with the desired instructions in the initial state and then adding invariants. Some changing of global constants is also required for each test. An example of the encoding of the weak coherence violation test (seen in Figure 3.3) across global memory intra block is shown below along with the invariant. Recall the

invariant has to be true for every state, however, we are only concerned with the state of the system after all the threads have finished executing; to this end, we add the predicate `All_Threads_Done()` which returns true if all threads have executed all their instructions, meaning their instruction queues are empty.

```
--T1
PGM[0][0][0].Op     := store;
PGM[0][0][0].addr   := 0;
PGM[0][0][0].Data   := 1;
PGM[0][0][0].Global := true;
--T2
PGM[1][0][0].Op     := store;
PGM[1][0][0].addr   := 1;
PGM[1][0][0].Data   := 2;
PGM[1][0][0].Global := true;
--T3
PGM[2][0][0].Op     := load;
PGM[2][0][0].addr   := 0;
PGM[2][0][0].reg    := 1;
PGM[2][0][0].Global := true;
--
PGM[2][0][1].Op     := load;
PGM[2][0][1].addr   := 0;
PGM[2][0][1].reg    := 2;
PGM[2][0][1].Global := true;
--T4
PGM[3][0][0].Op     := load;
PGM[3][0][0].addr   := 0;
PGM[3][0][0].reg    := 1;
PGM[3][0][0].Global := true;
```

```
--
PGM[3][0][1].Op      := load;
PGM[3][0][1].addr    := 0;
PGM[3][0][1].reg     := 2;
PGM[3][0][1].Global  := true;
--
Invariant
  !(All_Threads_Done() &
    REG[2][0][1] = 1  & REG[2][0][2] = 2 &
    REG[3][0][1] = 2  & REG[3][0][2] = 1 )
```

4.4.1 Results

The results of the litmus tests on the models (as seen in Table 4.2) were as expected. The litmus tests presented below are all discussed in Chapter 3. Because this project is not about the performance of model checkers, we do not record execution time or the number of states explored. We will note, however, that all tests executed in under a minute and the weak model had a much larger state space.

We end with some notes about the results. Notice that the simple visibility tests have the same results across both models; this is due to both the simplicity of the tests and the clear documentation on these matters. Also notice that shared memory visibility is only tested intra block as those threads are the only ones that share shared memory. The strong model does not allow either the classical or relaxed coherence violations while the weak model can show a classical coherence violation. Similar behavior is seen with write atomicity violations. Note that the strong model does not need cumulative fences as write atomicity implies cumulative fences. The weak model does not have cumulative fences as executing fences only shares values authored by the calling thread (see Section 4.2) which directly leads to the allowed write atomicity violation result.

Table 4.2: The results of the litmus tests when ran on the models.

✓- Guaranteed ✗ - Not Allowed ?- Allowed but not Guaranteed		
Test	**Weak Model**	**Strong Model**
Classical Coherence Violation (inter and intra block)	?	✗
Relaxed Coherence Violation (inter and intra block)	✗	✗
Simple Visibility Across Shared Memory With No Fence	?	?
Simple Visibility Across Shared Memory With `TFB`	✓	✓
Simple Visibility Across Shared Memory With `TF`	✓	✓
Simple Visibility Across Global Memory With No Fence (inter and intra block)	?	?
Simple Visibility Across Global Memory With `TFB` (intra block)	✓	✓
Simple Visibility Across Global Memory With `TFB` (inter block)	?	?
Simple Visibility Across Global Memory With `TF` (inter and intra block)	✓	✓
Write Atomicity Violation IRIW (inter and intra block)	?	✗

CHAPTER 5

EXAMPLES

In this section we show several examples of code that makes assumptions about the GPU shared memory consistency model which are inconsistent with our model. We show how to fix the offending code using our proposed model.

Synchronization operations are expensive and algorithms that can jettison these operations by taking advantage of low level atomics or other hardware features can gain substantial performance increases at the expense of exposing the memory model [4]. In GPUs, a popular technique is exploiting *warp-based scheduling*. That is, in NVIDIA GPUs, threads execute in a lock-step manner in groups of 32 [27, p.106]. This allows unsynchronized code in groups of 32 threads to execute on hardware while producing deterministic results. While our model does not take warps into consideration, we maintain that even when instructions are executed (or dequeued from the instruction queues) in a lock-step manner the memory visibility still behaves as we presented in our model. We analyze at a high level several CUDA programs that exploit this technique under the visibility properties of our shared memory consistency model.

5.1 MAGMA

MAGMA is a dense linear algebra library similar to LAPACK but for architectures with GPU accelerators [5]. The following code was found in a GPU kernel function in the file `zgemvt_kernel_fermi.cu`:

```
/* sdata is a shared memory array which is NOT declared volatile */
if(tx < 32)  sdata[tx] += sdata[tx + 32];
if(tx == 0)
   for(int i=1;i<32;i++)
        sdata[tx] += sdata[tx + i];
```

This code attempts to exploit warp based programming by allowing thread `tx==0` to do unsynchronized work on the `sdata` array. The final `for` loop done only by thread `tx==0` assumes that the data in `sdata` is visible after multiple other threads issue stores to it in the statement `sdata[tx]+=sdata[tx+32];`. Given the shared memory consistency model we presented, this is not the case. At a high level, the shared memory behavior here is similar to the `No-op` fence litmus test in Section 3.3.1 and a `__thread_fence_block()` is needed before the final `if` statement. The corrected code is:

```
if(tx < 32)  sdata[tx] += sdata[tx + 32];
__thread_fence_block();
if(tx == 0)
    for(int i=1;i<32;i++)
        sdata[tx] += sdata[tx + i];
```

5.2 Graph Traversal

Recently Merrill et. al. developed a novel new graph traversal algorithm specifically for NVIDIA GPUs [21]. As previously discussed, this algorithm exploits warp based scheduling for performance gains; in particular, one technique used several times in this algorithm we refer to as a *warp vote*. Generic code for this technique is given below:

```
//All threads in a warp write to the same location.
array[warp_id] = thread_id;
//Only one thread should make it past the conditional.
if (array[warp_id] == thread_id)
   //Do unsynchronized work w.r.t. other threads
   //in the warp.
```

In this technique all the threads within a warp write to the same memory location; however, due to warp-based scheduling all of the threads will attempt to write at once and only one of the writes will succeed. All of the threads will progress to the conditional at the same time, but only the thread whose write succeeded will get past the conditional. In short, threads in a warp vote for themselves to do unsynchronized work, and only one is elected, hence the name *warp vote*. Similar to the first example, the problem lies in the assumption that shared memory writes are immediately visible to other threads in the block. While the authors make note that the shared memory is declared `volatile`, we feel that `volatile` is a historically confusing[1] and an informal construct. Furthermore, the C11 ISO standard maintains that `volatile` should be used only for hardware accesses, not for thread communication[2] [12]. Since CUDA is generally compiled with a C compiler and given our formal modeling of GPU fences, we feel a more portable and robust solution would be to place a `__thread_fence_block()` between the memory write and conditional which gives the correct code of:

```
array[warp_id] = thread_id;
__thread_fence_block();
if (array[warp_id] == thread_id)
   //Do unsynchronized work w.r.t. other threads
   //in the warp.
```

[1] The keyword `volatile` is substantially different in Java, C, and C#

[2] While the standard doesn't exactly say this outright, it does not mention volatile at all in the context of concurrency

5.3 Software Memory Model

Finally, while the entirety of this work has been based on the hardware memory model, we now briefly show an example motivating the need for a software memory model which attempts to exploits warp based scheduling. Consider the following CUDA code:

```
/* x and y are allocated and initialized in the host code such that:
 * x[i] = y[i] = i
 * The kernel is launched with 1 block and 64 threads */
__global__ void kernel(int* x, int* y) {
  int index = threadIdx.x;
  y[index] = x[index] + y[index];
  if (index != 63 && index != 31)
    y[index+1] = 1111;
}
```

It is easy to see that under warp based scheduling there is no race as each thread accesses its own data element with the line `y[index] = x[index] + y[index]`. The conditional then weeds out the two threads at the warp edges and writes a flag value to location `index+1`. This means we would expect to see the flag value `1111` everywhere but index 0 and 32 i.e. `y = [0,1111, ..., 64, 1111, ....  1111, 1111]`. However, when compiled with regular `nvcc` with CUDA 4.2 we see the output: `y = [0, 2, 4, 6, 8, ...  126]` or `y[i] = 2*i`. Furthermore we see this output consistently, which is uncharacteristic for a race. To add to the confusion, when compiled with `nvcc` with CUDA 4.1 or with debug symbols i.e. `nvcc -g -G` we get the expected output. Speaking with an industry expert revealed that the `nvcc` compiler does not take into account warp-based scheduling and expects race-free programs under warp size of one. In order to fix this code to show the correct output, `x` and `y` could be labeled as volatile or either of the memory fences could be inserted immediately after the `y[index]=x[index]+y[index]` instruction[3]. Like the

[3] Although from Section 5.2 it should be clear we prefer the memory fence fix.

others, this program is similar to the `No-op` fence litmus test in Section 3.3.1 and the corrected code is:

```
__global__ void kernel(int* x, int* y) {
  int index = threadIdx.x;
  y[index] = x[index] + y[index];
  __thread_fence_block();
  if (index != 63 && index != 31)
    y[index+1] = 1111;
}
```

5.4 Detecting Memory Model Violations

While we do not have a tool that takes raw CUDA code as input and outputs memory model analysis, we propose a way to isolate potential trouble areas in code. Because weak memory model behaviors are invisible under well synchronized code, it is only when races (with warp size being one) are present that programmers have to even think about the memory model. The tool GKLEE [19] is a concolic executor for CUDA programs that has the option to model warp size equal to one. GKLEE can be run to show any unsynchronized areas in CUDA code that may need to be evaluated under the weak memory model. In fact this method was used in finding the MAGMA example in Section 5.1.

CHAPTER 6

CONCLUSION

This GPU shared memory consistency model is currently being reviewed by industry experts and is expected to grow and change based on feedback. This work provides a set of litmus tests that not only test for classical shared memory consistency properties, but also take into consideration the unique concurrency and memory systems of GPUs. Using the results of the litmus tests, determined by industry feedback and documentation, we provided a reasonable operational model. We implemented our model in the Murphi modeling language and verified the properties proposed by the initial litmus tests. Finally we have shown a need for a GPU memory model using real-world examples that make assumptions about local instruction orderings and memory visibilites across threads. It is our hope that this initial work will (1) educate developers that the GPU memory model needs to be considered when writing unsynchronized GPU code and (2) motivate conversation and developments in the area related to GPU memory models including robust and portable development of fast non-blocking algorithms and verification of such code. Future work includes an axiomatic model to compliment the operational model, a more complete treatment of the PTX intermediate code, comparing the GPU and C11 memory models, and a language-level model for CUDA compilation that is related to GPU hardware memory models.

REFERENCES

[1] D. Abts, D. J. Lilja, and S. Scott, *Toward complexity-effective verification: A case study of the Cray SV2 cache coherence protocol*, 2000.

[2] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, *Fences in weak memory models*, in Proceedings of the 22nd international conference on Computer Aided Verification, CAV'10, Berlin, Heidelberg, 2010, Springer-Verlag, pp. 258–272.

[3] H.-J. Boehm and S. V. Adve, *Foundations of the C++ concurrency memory model*, SIGPLAN Not., 43 (2008), pp. 68–78.

[4] ——, *You don't know jack about shared variables or memory models*, Commun. ACM, 55 (2012), pp. 48–54.

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

[6] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, *Implementing molecular dynamics on hybrid high performance computers - short range forces*, Computer Physics Communications, 182 (2011), pp. 898–911.

[7] D. Dill, *The Murphi verification system*, in Computer Aided Verification, vol. 1102 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1996, pp. 390–393.

[8] M. J. Flynn, *Some computer organizations and their effectiveness*, Computers, IEEE Transactions on, C-21 (1972), pp. 948 –960.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy, *Performance evaluation of memory consistency models for shared-memory multiprocessors*, SIGARCH Comput. Archit. News, 19 (1991), pp. 245–257.

[10] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, *Radiation modeling using the uintah heterogeneous cpu/gpu runtime system*, in Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond, XSEDE '12, New York, NY, USA, 2012, ACM, pp. 4:1–4:8.

[11] IBM, *Power ISA 2.06*, 2009. `http://www.power.org/documentation/power-isa-version-2-06-revision-b`.

[12] ISO/IEC, *Programming languages - C, draft N1570*, 2011. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[13] M. B. Jones, *What really happened on Mars?*, 1997. Retrieved Feb. 2013 from http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/.

[14] Khronos Group, *OpenCL: Open Computing Language*. http://www.khronos.org/opencl.

[15] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed., 2010.

[16] L. Lamport, *How to make a correct multiprocess program execute correctly on a multiprocessor,*, Computers, IEEE Transactions on, 46 (1997), pp. 779–782.

[17] J. W. O. Leary, M. Talupur, and M. R. Tuttle, *Protocol verification using flows: An industrial experience*, in Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD'09, IEEE, 2009, pp. 172–179.

[18] N. G. Leveson and C. S. Turner, *An investigation of the Therac-25 accidents*, Computer, 26 (1993), pp. 18–41.

[19] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, *GKLEE: concolic verification and test generation for GPUs*, in Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '12, New York, NY, USA, 2012, ACM, pp. 215–224.

[20] S. Mador-Haim, R. Alur, and M. M. K. Martin, *Litmus tests for comparing memory consistency models: how long do they need to be?*, in Proceedings of the 48th Design Automation Conference, DAC '11, New York, NY, USA, 2011, ACM, pp. 504–509.

[21] D. Merrill, M. Garland, and A. Grimshaw, *Scalable gpu graph traversal*, SIGPLAN Not., 47 (2012), pp. 117–128.

[22] R. H. B. Netzer and B. P. Miller, *What are race conditions?: Some issues and formalizations*, ACM Lett. Program. Lang. Syst., 1 (1992), pp. 74–88.

[23] NVIDIA, *Compute Unified Device Architecture (CUDA): Version 4.2 (May 2012)*. http://developer.nvidia.com/cuda/cuda-downloads.

[24] ——, *CUDA C Programming Guide: Version 5.0 (October 2012)*. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[25] ——, *Parallel Thread Execution ISA: Version 3.1 (September 2012)*. http://docs.nvidia.com/cuda/parallel-thread-execution.

[26] G. D. Plotkin, *A structural approach to operational semantics*, (1981). Computer Science Department, Aarhus University Denmark.

[27] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, Boston, MA, USA, 1st ed., 2010.

[28] 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.

[29] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, and B. P. Sutton, *Accelerating advanced MRI reconstructions on GPUs*, in Proceedings of the 5th conference on Computing frontiers, CF '08, New York, NY, USA, 2008, ACM, pp. 261–272.

[30] U.S.-Canada Power System Outage Task Force, *Final report on the August 14, 2003 blackout in the United States and Canada: Causes and recommendations*, 2004. `https://reports.energy.gov/`.

[31] D. L. Weaver and T. Germond, *The SPARC Architecture Manual: Version 9 (1994)*. `http://www.sparc.com/standards/SPARCV9.pdf`.