Towards Scalar Synchronization in SIMT Architectures

by

Arun Ramamurthy

B.Eng, McMaster University, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Masters of Applied Science

in

THE FACULTY OF GRADUATE STUDIES
(Electrical And Computer Engineering)

The University Of British Columbia
(Vancouver)

September 2011

Abstract

An important class of compute accelerators are graphics processing units (GPUs). Popular programming models for non-graphics computation on GPUs, such as CUDA and OpenCL, provide an abstraction of many parallel scalar threads. Contemporary GPU hardware groups 32 to 64 scalar threads as a single warp or wavefront and executes this group of scalar threads in lockstep. The inherent mismatch between scalar programming model and vector hardware creates a challenge when developing applications that employ synchronization on the GPU. This challenge arises from the use of a hardware stack to manage control flow divergence among scalar threads.

This thesis explains the porting of the Apriori benchmark to a GPU which led to the research on synchronization in SIMT hardware. It then proposes instruction set and hardware changes that simplify the implementation of mutual exclusion when porting multiple-instruction, multiple data (MIMD) programs with synchronization to accelerators employing single-instruction, multiple thread (SIMT) hardware. These instructions when compared with more complex software only solutions, achieve similar performance. This thesis also implements and evaluates queue based mutual exclusion on SIMT hardware.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my parents and my brother for motivating and inspiring me to pursue a higher education and for their constant support throughout the endeavour. I would like to thank my supervisor Dr.Tor Aamodt for his guidance. I want to acknowledge and thank Wilson Fung and other members of my research lab for their help and encouragement. Also, I could not have completed my degree without the support of my friends and thank them deeply. Last but not least, I thank the owners of curry point restaurant and their spicy potatoes for providing me with delicious sustenance during my time at UBC.

Chapter 1

Introduction

GPUs were originally conceived to speed up graphics processing. They are designed to exploit data parallelism and provide performance by maximizing throughput. However, the replacement of fixed function pipelines with programmable hardware in the mid 2000s [26] has allowed them to be used to speed up general purpose compute applications [20, 37, 39]. This branch of computing is popularly known as General Purpose GPU computing or GPGPU. Each generation of GPU hardware has increased the language features available to programmers and has steadily moved GPGPU into the mainstream. Several manufacturers are now even offering integrated CPU-GPU chips [12]. The work presented in this thesis looks at synchronization on GPUs and how it can be improved.

GPUs follow a Single Instruction Multiple Thread (SIMT) execution model [27]. In this model multiple threads are grouped together and executed in lock-step, meaning they execute the same instruction with different data. The groups of threads are known *warp* (NVIDIA terminology) or *wavefront* (ATI terminology). Control flow divergence in each warp is handled by a hardware SIMT stack [10, 25]. When a warp encounters a branch instruction, two entries are pushed on the stack. Each entry represents a sub-group of threads, grouped based on the results of the branch instructions i.e one entry for threads that took the branch and one for threads that did not. The sub-group of threads whose entry is at the top of the SIMT stack are executed first. Their execution continues up to a reconvergence point, a pre-determined instruction that is common to both sub-groups of threads. When a

sub-group reaches the reconvergence point, the top entry is popped from the SIMT stack. This allows the other sub group of threads to execute code, reach the reconvergence point and pop the top entry from the SIMT stack. Now the sub-groups are merged again and all of them will execute the remaining code. This serialization, where sub-groups of threads are executed one after each other, causes unexpected behaviour when synchronization code is implemented in a single threaded paradigm is executed in a SIMT fashion.

The thesis examines the implementations of various synchronization mechanisms in the context of SIMT. It then proposes instructions to simplify the implementation of certain synchronizations. The rest of this chapter describes the motivation behind this thesis, lists its contributions and explains the baseline GPU architecture.

1.1 Motivation

Example 1 shows the pseudo code of a spin lock. This kind of synchronization code which is written in the context of a single thread is what we refer to as *scalar synchronization*. We will now show what happens when this scalar synchronization code is executed in a SIMT architecture. In this example, all threads in a warp attempt to acquire a lock. To simplify the explanation, we assume that all the threads are attempting to acquire the same lock. Figure 1.1(a) shows the hardware stack before execution of the loop. When a thread acquires the lock, it exits the *while loop* and diverges from other threads in the warp. The reconvergence point for this while loop is immediately after the while loop and before the critical region (the reason for which is explained in more detail in Section 1.3.2 of this Chapter). Thus, the thread that obtained the lock, will wait to reconverge with the other subgroup of threads before executing the critical section and the lock release operation. Figure 1.1(b) shows the hardware stack after a thread has acquired the lock. Since the thread that has acquired the lock has already reached the RPC, it does not have an entry on the stack. The other threads in the warp are now waiting for a lock that is not available. Since they cannot acquire it, they will not be able to reach the RPC and this causes a deadlock.

Similarly the implementation of other blocking synchronizations such as mu-

Example 1 Pseudo code of a spin lock

```
while(acquire(lock));    (A)
//CRITICAL SECTION       (B)
Release(lock);           (C)
```

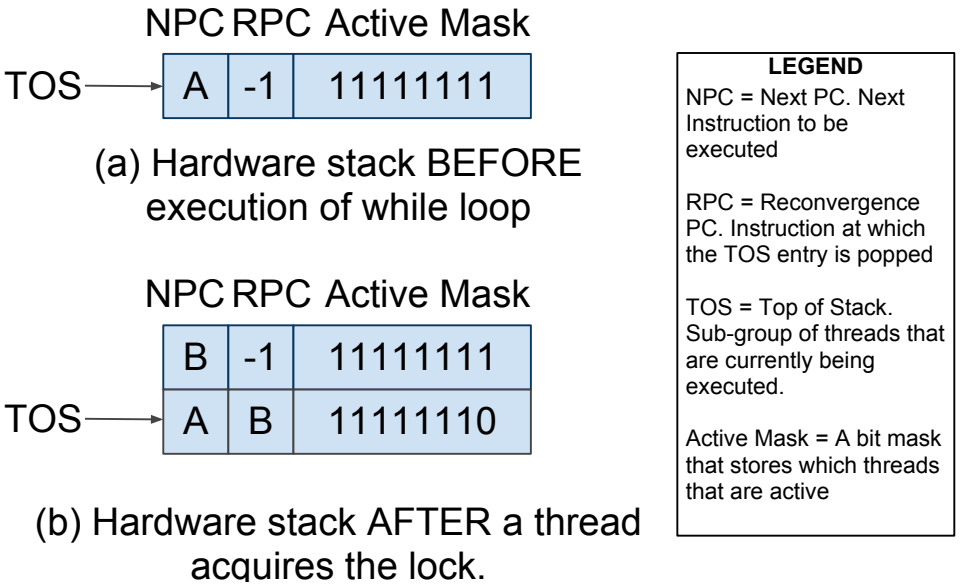

Figure 1.1: Hardware Stack for Example 1

texes, semaphores and barriers requires threads to spin in loops until they can satisfy the condition of the synchronization. When this synchronization takes place between threads in the same warp these loops can cause deadlocks unless they are tailored to consider the reconvergence mechanism. Even if threads in the same warp are not competing for the same resources, the synchronization acts on a per warp basis instead of the per thread basis expected by the programmer. The work presented in this thesis addresses these issues and works towards the goal of using scalar synchronization code in SIMT architectures.

Contemporary GPU hardware is architected with graphics applications in mind. These applications are inherently parallel and where synchronization is required, raster graphics specific hardware is employed. The goal of a graphics application

is to render a three dimensional scene onto the screen. This involves calculation of vertices by a vertex processor whose output is fed into a rasterizer, which in turn produces pixel fragments [27, 32]. Each pixel fragment represents the contribution of an object in the scene to a pixel's colours. Pixel shader programs are responsible for determining the colour of each fragment. A process known as alpha blending is performed on the output of the pixel shader programs and it calculates the final colour of each pixel [46]. The primary need for synchronization occurs in this alpha bending phase when two different objects might want to update the same pixel. This requires an exclusive read-modify-write operation on a memory location [46] and the microarchitecture handles this with atomic functions [27, 31]. At first glance these atomic functions seem ideal to implement various synchronizations. However, these atomic functions are not designed to interact with the scalar control flow of a shader program and can cause unexpected behavior. In this thesis, we propose adding new instructions that are capable of operating in co-operation with the SIMT stack to simplify the implementation of synchronization mechanisms.

1.2 Contributions

This thesis makes the following contributions:

1. It explains the Apriori benchmark and the challenges faced in porting it to a GPU. Porting Apriori led to the research on implementing synchronizations on a GPU

2. It proposes new instructions that perform lock and unlock operations and simplify the implementation of mutexes on GPUs

3. It explores the implementation of queuing synchronization requests on a GPU.

4. It evaluates these instructions and the effect of queuing synchronizations on several applications including Apriori.

5. It identifies and fixes an issue where recursive functions did not execute correctly with a post dominator based SIMT stack.

1.3 Background

In this thesis we study the use of GPUs as compute accelerators. CUDA [35, 37] and OpenCL [20] are two programming models we use for this purpose. This section describes the hardware architecture of contemporary GPUs and the compute model they use.

1.3.1 Baseline GPU architecture

A GPU is designed to exploit data parallelism by executing a large number of threads in parallel. A GPU program is executed by launching a *kernel* from the CPU. This kernel comprises the code to be executed and the number of threads that should execute it. These threads are divided into *thread blocks*, with the total number of blocks and number of threads per block specified by the programmer. Each thread block is assigned to be executed on a *shader core*. These shader cores form the backbone of the architecture. A shader core picks a group of threads and executes them in lock-step i.e. they execute the same instruction, albeit with different data. This model of execution is called Single Instruction Multiple Thread (SIMT) [27] and the group of threads are known as a *warp* or *wavefront*. These thread groupings are fixed and threads cannot be move around between warps. The shader core is capable of executing warps from different thread blocks and the number of thread blocks that a core can execute simultaneously is dependent on the specific GPU. If a kernel contains more thread blocks than can be scheduled they have to wait until other thread blocks finish executing before being scheduled. Figure 1.2 depicts this baseline GPU architecture.

A shader core can be thought of as its own independent unit of execution with dedicated resources. It consists of an Instruction fetch and decode unit that is shared among multiple *SIMT lanes* as seen in Figure 1.3. Each of these lanes has its own ALUs and is responsible for executing the given instruction for a single thread. The shader core has a single register file which is banked and can be accessed in parallel by all the threads.

Each thread in the GPU has access to several different memory spaces. To begin with, a thread has its own private *local memory*. Even though this memory space is per thread, it is actually stored in the global DRAM. Next, each shader

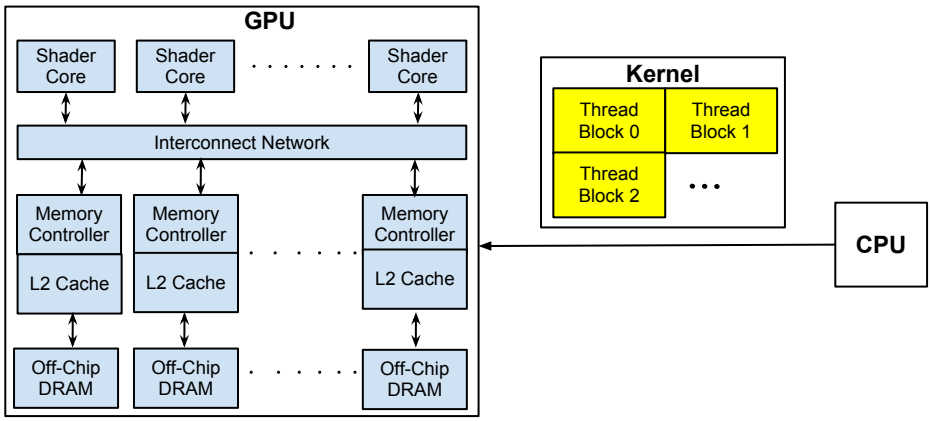

Figure 1.2: Baseline Architecture

core has its own fast *shared memory* and it can be used by threads inside of a thread block to communicate with each other. The off chip DRAM memory is known as *global memory* and is accessible by all threads on the GPU. Threads from different blocks cannot communicate other than by writing values to global memory.

As mentioned before, shader cores execute threads in a warp in lock-step. When a warp encounters a branch instruction, it can cause control flow divergence if not all threads in the warp take the branch. This control flow divergence is handled by the use of a hardware stack. Section 1.3.2 of this chapter will give more details on this stack.

1.3.2 Stack Based Reconvergence Mechanism

Note that this mechanism is not exactly the same as described in [10, 25], but behaves in essentially the same way.

When a warp is scheduled for execution the SIMD hardware reads the stack for this warp. It uses the information in the top of the stack (*TOS*) to determine which instruction to execute and the threads that should execute it. Each stack entry consists of:

1. Next PC (NPC): The address of the next instruction to be executed by this warp.

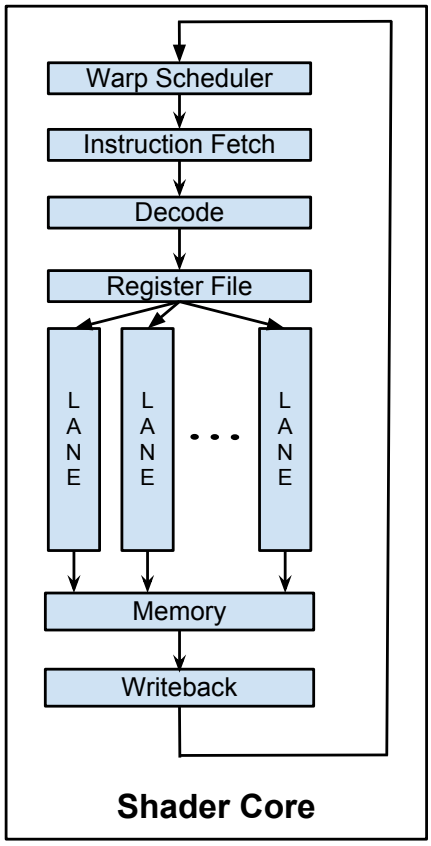

Figure 1.3: Shader Core

2. Active Mask: A bit mask specifying which threads in the warp should execute this next instruction.

3. Reconvergence PC (*RPC*): The address of the instruction where these threads converge with other threads in the warp.

Figure 1.4 illustrates an examples of how the stack handles control flow divergence. Figure 1.4(a) shows the control flow graph (CFG) of a program where the labels refer to instructions of the program. Figure 1.4(b) shows the contents of the stack as the warp executes the program. At the start of the program the TOS entry for the warp indicates that all threads in the warp must execute the instruction at address X. Once this instruction has been executed, the stack is updated to reflect

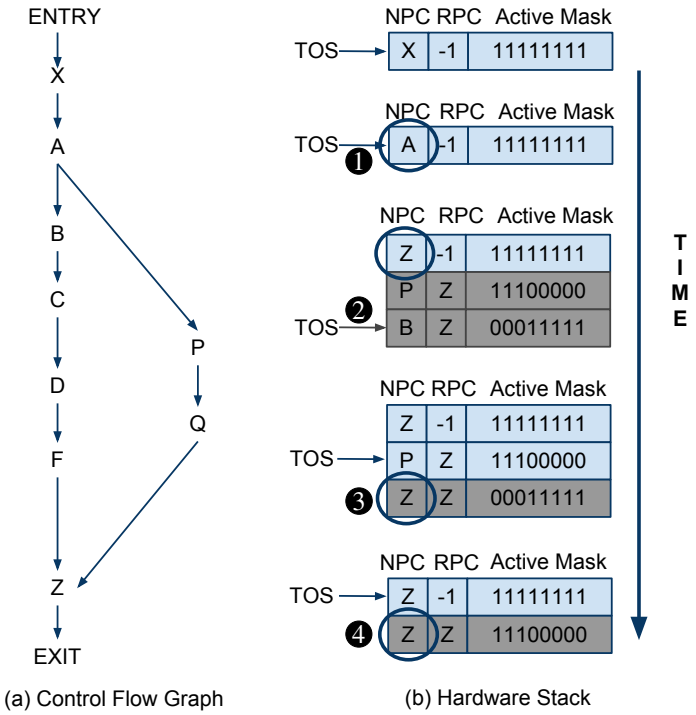

Figure 1.4: Control Flow Graph and Hardware Stack (Note: This Stack Based Reconvergence Mechanism Can Handle Arbitrary Levels of Nested Control Flow and Loops)

the PC of the next instruction (❶). When the next instruction at *A* is executed the threads in the warp diverge, with some threads going to *P* and other threads to *B*. Two entries are added on to the stack (❷) (in alternate implementation special instructions are used to manage the stack [10, 24]). The *Active Mask* of one entry contains the bit mask of the threads that follow the control flow path of the branch and its *NPC* is the target of the branch. The *Active Mask* of the next entry represents the threads that did not take the branch and its *NPC* is the address of the instruction that follows the branch. The *NPC* of the original TOS stack entry and the RPC of both the new entries is the address of the instruction where the the threads reconverge and return to being executed in lock-step. This reconvergence PC is determined by the assembler using control flow analysis to find the immediate post dominator [33] of the branch instruction.

In a control flow graph, a node M is said to post-dominate a node N if all paths to the exit starting at node N must pass through M. In the control flow graph for our program F post dominates C as all paths to the exit node from C must pass through F. The RPC of a branch instruction is determined by analysing the control flow graph of a program and finding the immediate post dominator of that branch instruction. In our example Z and F are the immediate post dominators and thus the *RPC* of branch instructions A and C respectively.

The stack is also capable of handling nested control flow and if a group of divergent threads reaches another branch instruction, two additional entries are pushed on to the stack in the same manner as described above. The stacks are sized to handle complete divergence where every thread in the warp is executing a different PC and thus can handle any level of nested control flow. The SIMD hardware continues executing the threads in the TOS entry until the *NPC* of this entry matches the RPC (❸). At this point, the TOS is updated by popping a stack entry. Once the next group of threads reach the *RPC* (❹), the TOS is updated again. Now the two groups of threads from branch A have reconverged and will continue execution in lock-step.

This stack-based divergence mechanism is typically considered a hardware detail and the programming model for GPUs abstracts it away [21, 37]. Chapter 3 examines the issues faced when implementing synchronization in the presence of this hardware stack. However, before we examine synchronization, it is necessary to explain the atomic functions that are used to implement them.

1.3.3 Atomic Functions

Our Baseline architecture provides hardware support for atomic functions which are available in both CUDA [37] and OpenCL [21]. An atomic function is one which is guaranteed to be executed without interference from other threads. It can perform a read-modify-write operation on any 32- bit or 64-bit address residing in global or shared memory. Two functions of particular importance to us are:

1. **AtomicExchange (Address,Value)** abbreviated in this thesis as *atomicExch*. It replaces the contents at memory location *Address* with *Value*.

2. **AtomicCompareAndSwap (Address, Compare, NewValue)** abbreviated in this thesis as *atomicCAS*. It reads the memory location *Address*, checks if its value is equal to *Compare* and if so, swaps it for *NewValue*. It also returns the original contents at *Address*.

These functions are implemented in hardware as a single instruction in the ISA. Before we examine the implementation of synchronization on a GPU, Chapter 2 describes the Apriori benchmark which helped discover the problems with synchronization presented in Chapter 3.

Chapter 2

Apriori Benchmark

Apriori comes from the RMS-TM [19] benchmark suite and is part of an emerging class of algorithms knows as Recognition, Mining and Synthesis. These algorithms have been tailored towards CMPs [19] and hence present opportunities for the use of GPUs. Apriori was ported over to a GPU as it is a scalable parallel algorithm. It also requires fine grained synchronization in its parallel region, making it ideal to evaluate the instruction that will be proposed in Chapter 4. This chapter describes the algorithm, the methodology used and challenges faced in porting it

2.1 Algorithm

Apriori is an Association Rule Mining algorithm [3]. The algorithm works on databases where each record contains a list of items sorted in ascending order. These items are taken from a known fixed set of items. For example if we assume 10 types of items, a record in the database could be (1,4,6,8,10) and another might be (2,4,8,9).

The goal of the algorithm is to find the most frequent sub-sets in a given database of records. It starts by forming sets of two known as *candidate sets* some examples of candidate sets are (1,2), (1,3) ,(1,4), (4,7), etc. The *support* level for each set is its frequency of occurrence among all the records of the given database. The algorithm scans the database and increments a support counter for each candidate set. A *minimum support* level is specified as input parameter to the algorithm

and any candidate sets below that frequency level are dropped. The remaining candidate sets are then used to generate new candidate sets of size three in accordance with the Apriori property. This property states that if K is the size of the current candidate set, all sub sets of candidate set size $(K+1)$ must also meet the minimum support level [3]. For example, let us say that after counting the support level the candidate sets remaining are (1,2), (1,4), (1,6), (2,6), (3,7), (5,7). A candidate set of (1,2,4) would **not** be included in the new list as (2,4) did not meet the minimum support level. However, a candidate set (1,2,6) would be included. Each iteration of the algorithm repeats this process and increases the size of the candidate set by one. The algorithm terminates if the candidate set list contains only one set after support level have been counted or if it cannot generate a new candidate set list (as per the Apriori property).

Once a candidate set list has been created, each candidate set is inserted into a *Hash Tree* data structure. Each node of a *Hash Tree* contains a hash table that indexes the nodes below it and the leaves of the *Hash Tree* contain candidate sets. The hash function is applied to the element of the candidate set with the same depth as the node. For example, in a candidate set of (1,4,6,8,9,15) 1 will be hashed in the root node which will lead to a leaf node where 4 will be hashed followed by 6 etc until a leaf node is reached. An example of the *Hash Tree* data structure is shown in Figure2.1.

The *HashTree* grows when the number of candidate sets in a leaf node crosses a pre-set threshold. A process known as *rehashing* converts the leaf node into an internal node and distributes its candidate sets among newly created leaf nodes. The original Apriori benchmarks used transactions when inserting or rehashing the *Hash Tree* to enable parallelism. This insertion into the *Hash Tree* and the generation of the candidate sets formed the *AprioriGen* function. Counting the frequency of candidate sets in the database was done by the *subset* function. These functions were the main focus of porting the benchmark to a GPU. The following section will detail the methodology used to do so.

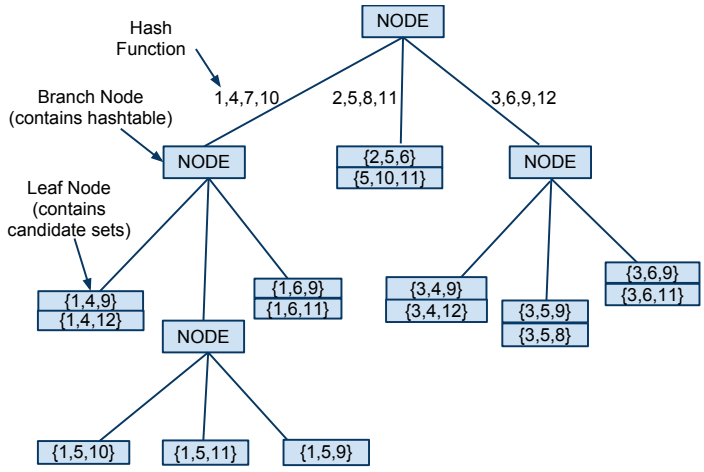

Figure 2.1: Example of a HashTree

2.2 Methodology

The original benchmark was written in C++ and used Intel STM[41] compiler primitives to implement transactional memory[17]. It was ported to run on a GPU using CUDA 3.1[35, 37]. Initially the benchmark was used for research on transactional memory on GPUs. It was then adopted for research on synchronization in GPUs. The original RMS-TM version of the benchmark will here be referred to as Apriori-Orig. The CUDA version that uses transactions will be called Apriori-TM and the CUDA version that used locks will be called Apriori-Lock

2.2.1 Dynamic Memory Allocator on the GPU

The linked list used to store the candidate sets and the *Hash Tree* data structures both required dynamic memory allocation. At the time of porting, CUDA did not support the use of *new* and *delete*. Hence it was necessary to create a parallel dynamic memory allocator for the GPU. The original benchmark was instrumented and profiled to obtain the **maximum** memory needed **per iteration** for both the linked list and hash tree data structures. The memory required for several temporary data structures was also determined. Each type of data structure was given a *empty list* that was a statically allocated array of that data structure. A separate counter variable for each *empty list* was used to index the array. The calls to *new*

were replaced by a function that performed an **atomic** increments on the counter and returned an index to the array. The atomics were used as threads running in parallel could request for new objects at the same time and this ensured each thread received a different index for the *empty list*. The counters were reset every iteration and the same memory was reused.

2.2.2 Race Condition in Apriori

Apriori-Orig was to be used as a gold standard to verify correct execution of the CUDA version. Each iteration of the benchmark printed the number of candidate sets in the list before insertion in the *Hash Tree* and also the remaining number after counting the frequency. The Apriori algorithm is deterministic and given fixed input parameters, should have the same output for every run. However, in Apriori-Orig, some iterations produced different number of candidate sets across multiple runs.

The issue was the improper use of transactions in the benchmark. The original benchmark accessed a variable outside the critical section, that could potentially be changed inside the critical section. This boolean variable *leaf* was a member of each *Hash Tree* node and indicated whether the node was a leaf node or not. The variable was accessed outside the transaction to determine if the *Hash Tree* should be traversed further or if insertion should be attempted. During insertion, if the *rehash* process described in section 2.1 occurs, the value of the leaf variable will change inside the transaction. This led to a race condition among the threads and produced different outputs. The critical section was expanded by moving the transaction boundaries to include the access of the *leaf* variable. This resulted in consistent outputs for Apriori-Orig with no impact on performance. After this fix was applied, Apriori-Orig became the gold standard.

2.2.3 Recursive Functions in Reconvergence Stack

Apriori used recursive functions to traverse the *Hash Tree* data structure. These recursive functions did not execute as expected with the reconvergence mechanism described in Section 1.3.2. In this mechanism, two divergent thread groups re-converge when they have both reached a pre-determined RPC (reconvergence PC).

However, this mechanism does not take into account the level of recursion of a thread and can erroneously converge thread groups. Example 2 shows a code snippet where this can occur. Figure 2.2 & 2.3 shows the operation of the stack where the subscripts for the NPC and RPC entries show the level of nesting. It is to be noted that entries in these figures where $NPC = RPC$ are shown for illustration purposes. A real hardware stack would not add such entries as they have already reconverged and will simply be removed when they become the TOS entry. In this example we assume that the *RecursiveFunction* is initially called from a location outside the function with parameter $X = 0$. In this example all the threads in a warp reach the branch instruction at B and execute it. This causes the threads to diverge as seen in Figure 2.2(a). The threads with $thread.id > 4$ take the branch, execute the function call at C and call *RecursiveFunction* again, as seen in Figure 2.2(b). When these threads reach the branch instruction at B, they do not take the branch and the NPC becomes D. This will result in the reconvergence mechanism thinking that the threads have reached their RPC and the TOS entry of the stack will be popped as seen in Figure 2.2(c). All the threads will now execute the code at *D* and exit the function. This will result in $X = 3$ for threads with $thread.id > 4$ and $X = 2$ for the other threads. This is incorrect as threads with $thread.id > 4$ should have returned $X = 4$. These threads should have executed the code at *D before* reconverging in addition to having executed it after reconvergence.

Example 2 Reconvergence Stack and Recursive Functions

```
RecursiveFunction (int x )
{
    x=x+1                          (A)
    if (x==1 && thread.id>4)       (B)
       RecursiveFunction(x)        (C)
    x=x+1                          (D)
    return x                       (E)
}
```

To rectify this problem, a new stack entry is added. Example 2 is reused to explain this mechanism. Figure 2.3(a) shows the stack before threads with $thread.id > 4$ execute the function call at *C*. After the call, a new entry known

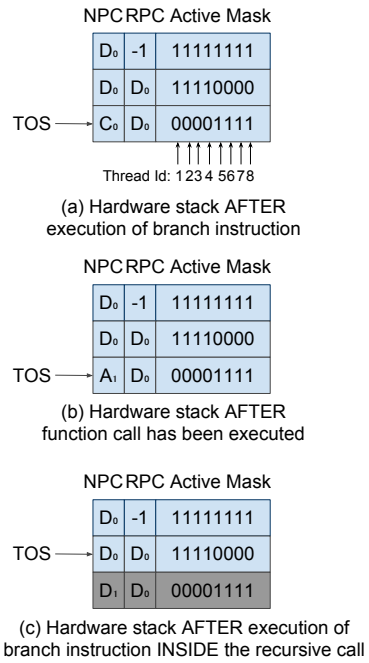

Figure 2.2: Hardware Stack for Example 2 (Grey Indicates Entry That is Being Popped from Stack).

as *call entry* is added to the stack as shown in Figure 2.3(b). This entry has the same Active Mask as the previous entry, the NPC corresponds to the start of the function and the RPC is set to -1. The NPC of the stack entry below the *call* entry is changed to the return PC of the function. The threads with $thread.id > 4$ will now execute till E and return from the function, at which point the *call entry* is removed from the stack. The two entries below the *call entry* have $NPC = RPC$ and will be removed by the reconvergence mechanism as seen in Figure 2.3(c). The entire warp will then execute the code at D and exit the function. Since the threads with $thread.id > 4$ executed the code at D twice, they will correctly return $X = 4$. Thus, the call entry creates a boundary around function calls and prevents reconvergence of threads with different nesting depths. It is to be noted that any divergence within the function will be handled in the normal way by adding more entries to the stack above the *call* entry.

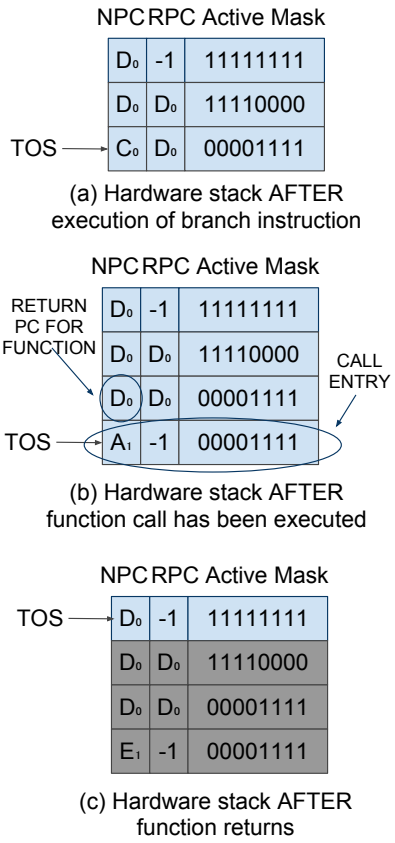

Figure 2.3: Hardware Stack That Uses *Call* Entry

2.2.4 Locks to Protect Insertion into Hash Tree

Apriori-TM, the first GPU version used transactions to allow parallel insert of candidate sets into the Hash tree. This benchmark was used for work on implementing a transactional memory system on a GPU. However, the development of Apriori-Lock required implementing spin locks on a GPU. The next chapter details the challenges associated with this and other synchronizations.

Chapter 3

Problems with Synchronization in SIMT hardware

Contemporary GPU hardware achieves performance by maximizing throughput per unit hardware cost. They are perfect for applications that have a large amount of data parallelism but require little synchronization. However, there exists a large class of parallel applications that require synchronization to take advantage of their data parallelism [23]. The synchronization mechanism used by traditional parallel programs can broadly be classified into the following categories:

1. **Mutex**: Algorithms where a thread can only enter a critical section after acquiring exclusive access to one or more locks.

2. **Semaphores**: Threads *wait* until a resource is available and *signal* when they are finished with the resource.

3. **Barriers**: A barrier is a point in the source code where a thread must stop and cannot continue until all other threads have reached the same location.

The rest of this chapter details how the reconvergence based hardware stack explained in Chapter 1.3.2 complicates the implementation of the above synchronization methods.

3.1 Mutex

Mutexes are used to avoid the simultaneous use of shared data by multiple threads. They can be subdivided into:

1. **Blocking**: The lock operation on the mutex does not return until the lock is acquired.

2. **Non-blocking**: The lock operation attempts to acquire the mutex once and return regardless of success or failure.

The implementation of non-blocking mutexes is straightforward and can be achieved by the use of atomics described in Chapter 1.3.3. This work shows how the implementation of blocking mutexes on a GPU does not behave as expected. Specifically, it demonstrates how a programmer cannot ignore the SIMT behavior as advocated by contemporary GPU programming models [21, 37].

Example 3 shows the CPU implementation of a spinlock. The atomicCAS function as described in Section 1.3.3 is also available on CPUs [22] and is a popular method to implement this lock [30].

Example 3 Spin lock using atomic compare and swap

```
//Spin until lock is acquired
while(atomicCAS(lock,-1,Unique_ID)!=-1));
//CRITICAL SECTION
atomicExch(lock,-1); \\Release Lock
```

In Example 3, the unique ID would be a unique index assigned to the thread at the beginning of the program that does not change for the lifetime of the program. The threads spin on the while loop until one of them is able to replace the value of the lock with its unique ID. At this point, no other threads can enter the critical section as the atomicCAS does not succeed for them and mutual exclusion is guaranteed. Since GPUs provide support for atomic functions, a programmer wishing to port CPU code that uses locks would reuse the existing CPU version of the spin lock and expect it to execute correctly on the GPU. However, the code does not behave as expected and leads to deadlock among the threads.

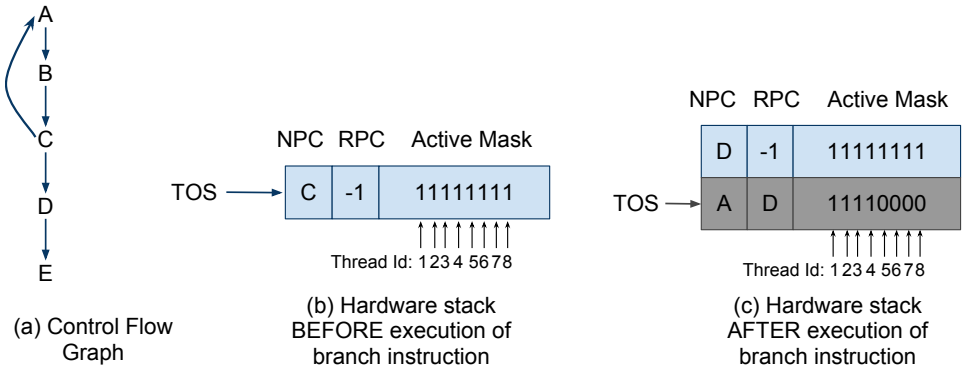

Figure 3.1: Control Flow Graph and Hardware Stack for Example 4

Example 4 shows pseudo assembly code for Example 3. The code has been labelled with letters which correspond to the control flow graph for this code shown in Figure 3.1(a). The hardware stack before the execution of the branch instruction at *C* is show in Figure 3.1(b). In Example 4 at *A*, threads 1 and 5 are contending for lock *W*, threads 2 and 6 are contending for lock *X*, threads 3 and 7 are contending for lock *Y* and threads 4 and 8 are contending for lock *Z*

Example 4 Pseudo assembly code of a spin lock on a GPU

```
Begin Loop:
  atomic.CAS $address,$compare,$new,$old;    (A)
  compare  $old, -1, $result;                (B)
  Branch  ($result == 1 ) Go To Begin Loop;  (C)
  //Critical Section Code                    (D)
  atomic.exch $address,-1 //Release Lock     (E)
```

After the execution of the branch instruction, the warp consists of two groups of threads, one group consists of the threads that acquired locks (threads 5,6,7,8) and the other is all the threads that did not. For the purpose of this discussion we will refer to a subgroup of threads in a warp as a *subwarp*. The *NPC* of the subwarp containing the threads that acquired locks is *D*, which is the same as the RPC, therefore this subwarp has reconverged and no entry is pushed to the stack. The locks acquired by these threads are not released since this subwarp has not

reached the instruction *E*. Next, the SIMD hardware pushes a stack entry for the other subwarp as shown in Figure 3.1(c) (the shaded entry indicates a newly added stack entry). This allows threads 1,2,3 and 4 to execute and contend for the locks they require. Since the locks have not yet been released, the threads waiting for the lock cannot exit the loop and reach the RPC at *D*. Thus, the stack entry is never removed and results in deadlock.

The problem of code not behaving as expected is relevant even if all the threads in the warp were trying to acquire different locks. In this scenario, all threads in the warp trying to acquire the lock, have to succeed and reconverge before executing the critical section. This leads to warp level synchronization as opposed to the thread level synchronization expected by the programmer. Also, even if each thread in the warp contends for a different lock, we could still get deadlocks. Consider two warps A and B which have eight threads. The corresponding threads in each warp contend for the same lock. Threads 1 to 4 of warp A and threads 5 to 8 of warp B succeed in acquiring the locks they are contending for. Now threads 5 to 8 in warp A are waiting for locks held by warp B. Similarly threads 1 to 4 in warp B are waiting for locks acquired by warp A. Since neither warp can make forward progress, a deadlock is encountered.

Example 5 Implementation of a spin lock on a GPU [42]

```
loop_continue=true;
while(loop_continue){
  if(atomicCAS(&lock,-1,thread_Index)==-1){
    //critical section
    atomicExch(lock,-1); //Release Lock
    loop_continue=false; // exit loop
  }
}
```

To overcome this issue, a programmer who has a good understanding of the reconvergence stack mechanism would have to code the locks in a manner such that a RPC is reachable by both the subwarps. One such implementation is shown in Example 5 [42]. The pseudo assembly code obtained from compiling Example 5 is shown in Example 6. The hardware stack entries before and after the execution of

Example 6 Pseudo assembly code of Example 5

```
  store $loop_exit,true;                        (A)
Outer Loop:
  atomic.CAS $address,$compare,$new,$old;       (B)
  compare  $old, -1, $result;                   (C)
  Branch  ($result == 0 ) Go To Check Loop;     (D)
  //Critical Section Code                       (E)
  atomic.exch $address,-1; //Release Lock       (F)
  store $loop_exit,false;                       (G)
Check Loop:
  compare $loop_exit,true,$result;              (H)
  Branch ($result == 1 )goto Outer Loop;        (I)
  exit;                                         (J)
```

this branch are shown in Figure 3.2(a) and Figure 3.2(b). In this example the subwarp of threads that acquired the lock can execute the critical section and release the lock before reconverging with the other subwarp at *H* as shown in Figure 3.2(c). When the next branch instruction at *I* is encountered, the threads that have executed the critical section diverge and the new entries pushed on to the stack are show in Figure 3.2(d). Note that if the *NPC* of a subwarp is the same as the *RPC*, an entry is not pushed on to the stack. The threads that have finished execution wait for the rest of the threads at *J*. In the next iteration of the while loop, it is possible that only a subset of the threads acquire the lock. If this happens the warp diverges again at instruction *D* and a new entry is pushed on to the hardware stack as shown in Figure 3.2(e). Once again, the threads that acquired the lock, complete the critical section, reconverge at H, diverge again at I and wait for remaining threads at J. This process continues until all threads in the warp have completed their critical section thus achieving a spin lock among the threads in a warp.

The code provided in Example 5 could conceivably be reused as macros for programs where the *lock* and *unlock* operations are at the beginning and end of the critical sections. However this is not always the case, **radiosity** from the SPLASH-2 benchmark suite is one such application. An excerpt of code from **radiosity** is shown in Example 7. Adding another *while* loop to replace *A* would require the *break* statement at *E* to be moved below *F*. Additional code to predicate the

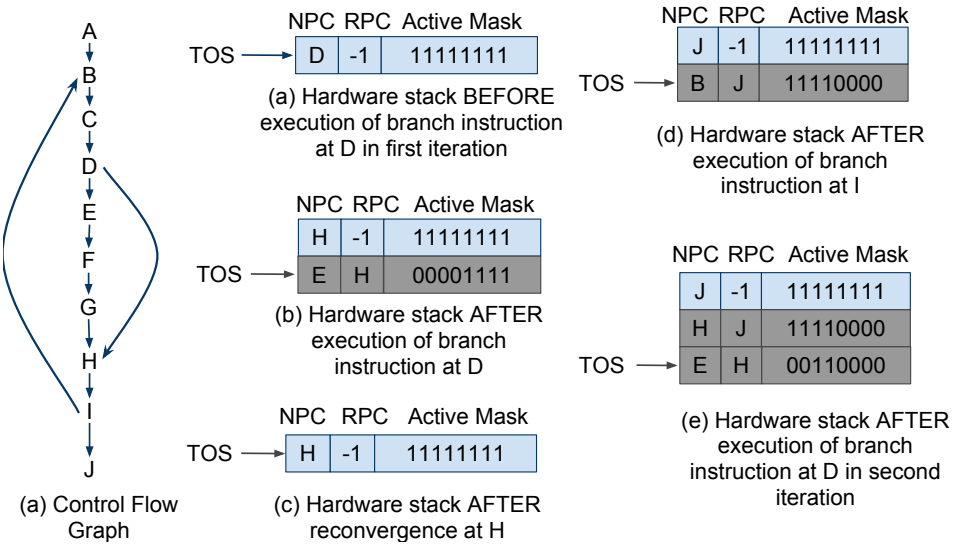

Figure 3.2: Control Flow Graph and Hardware Stack for Example 6

execution of this break on the execution of *C* also needs to be added.

Every time programmers encounter a different combination of *lock* and *unlock* operations separated by control flow, they would have to spend time adapting the code to a SIMT model. This adds development overhead to porting applications onto a GPU and could deter the more widespread adoption of GPUs for general purpose compute applications.

3.2 Semaphores

Semaphores are used to protect access to shared resources and differ from locks by allowing an arbitrary resource count. In a semaphore the *signal* operation increments a resource count and the *wait* operation checks for resource availability before proceeding. Semaphores are divided into two categories:

1. **Busy waiting**: The *wait* operation repeatedly checks if the resource count is available and returns only when it has acquired a resource and decremented the count. The *signal* operation just increments the resource count.

Example 7 Excerpt from Radiosity application in SPLASH-2 benchmark suite

```
while (...){
  if(...){
      LOCK();                    (A)
      if( ... ){                 (B)
                  .
                  .              (C)
                  .
              UNLOCK();          (D)
              break ;            (E)
      }
      UNLOCK();                  (F)
  }
}                                (G)
```

2. **Blocking**: The *wait* operation checks if the resource count is zero and if not puts the thread to sleep. It also inserts it into a waiting queue. The *signal* operation wakes up the first thread in the queue or increments the resource count if the queue is empty.

The blocking semaphores are used in operating systems where resources can be conserved by putting threads to sleep. Since contemporary GPUs do not yet provide this functionality, this work focuses on *busy waiting* semaphores and shows how their implementation is complicated by the interaction of the SIMT stack and the hardware based reconvergence mechanism described in Chapter 1.3.2

Example 8 shows one way a counting semaphore could be implemented on a CPU. It is necessary for the *wait* operation to acquire a lock and ensure that only one thread at a time is updating the semaphore. In this example, the mutex is implemented such that the RPC *G* for the branch at *A* is reachable by both the divergent subwarps. However, in this case, even this is insufficient and subtleties in the code make the semaphore behave differently then expected.

A thread that acquires the lock and decreases the semaphore exits the outer while loop and waits to reconverge with other threads at *I*. This means a thread that has acquired the semaphore does not execute the critical section until all other threads in the warp trying to acquire the semaphore have also done so.

Once again, synchronization code written for a single threaded model does not behave as expected when executed in a SIMT model. Also, it reiterates that the code structure provided in Example 5 is not sufficient to implement every form of synchronization.

Semaphores are an important synchronization mechanism and a solution to implement them on GPUs is left for future work.

Example 8 Implementation of a Semaphore

```
lock=-1 ; // lock to protect the semaphore.
s=10; // Number of resources available
void kernel(){
    Wait(s)
    //Critical Section
   Signal(s)
}
void Wait( int s){
bool loop_continue=true;
while(loop_continue){
  //LOCK
  if ( atomicCAS(lock,-1,unique ID)==-1){   (A)
     if(s>0){                               (B)
        s--;                                (C)
        loop_continue=false;                (D)
      }                                     (E)
    //UNLOCK
    atomicExch(lock,-1);                    (F)
    }                                       (G)
  }                                         (H)
}                                           (I)
void Signal(int s){
  atomicCAS(lock,-1,unique ID);
  s++;
  atomicExch(lock,-1);
}
```

3.3 Barriers

Barriers are meant to provide a global synchronization point that all threads in a kernel have to reach before proceeding further. Contemporary GPU programming

models provide some support for synchronization of threads within a thread block but not across the whole kernel [21, 37]. They propose implementing synchronization barriers by breaking down the algorithm before and after the barrier into separate kernel calls. However, launching a kernel has some overhead associated with it and this method becomes inefficient for algorithms that do not have enough work in between barriers. Furthermore, the more kernels that an algorithm is broken into, the more data structures that will have to be placed in global memory. This reduces the chances to optimize the kernel performance by placing data in the low latency shared memory.

Xiao et al. proposed implementing barriers in software by using an array in global memory to synchronize all thread blocks [49]. However, this solution cannot be used when there are more thread blocks in a kernel than can be scheduled on a core. This can only be handled by context switching the thread blocks that have reached the barrier with waiting blocks. Contemporary GPUs do not yet provide support for such operations. Also, the proposed solution uses the intrinsic thread block barrier in its implementation of the global kernel barrier. Wong et al. showed that the barrier operates on a per warp level as opposed to the per thread level as expected by a programmer [47]. Thus by extension, the use of Xiao et al.'s solution in different control flow paths could also produce unexpected behaviour.

We would like to identify *barriers* on a GPU as an important part of the problems with synchronization on a GPU. However providing hardware support for implementing them is beyond the scope of this work.

Chapter 4

Proposed Lock and Unlock Instruction

The goal of the solution is to aid programmers by simplifying the implementation of synchronization. To address the problems described in Chapter 3.1, we propose adding two instructions to the ISA:

1. Lock $address
2. Unlock $address

These instructions are exposed to the programmer through corresponding API calls and enables them to maintain the level of abstraction detailed in the programming model. It allows them to focus on optimizing the code for a GPU instead of reimplementing synchronization mechanisms. Example 9 shows how these instructions can be used to implement the spinlock shown in Example 3

Example 9 Implementation of a spinlock with new instructions

```
Lock (lockAddress);   (A)
//Critical Section    (B)
Unlock (lockAddress); (C)
// More Code          (D)
```

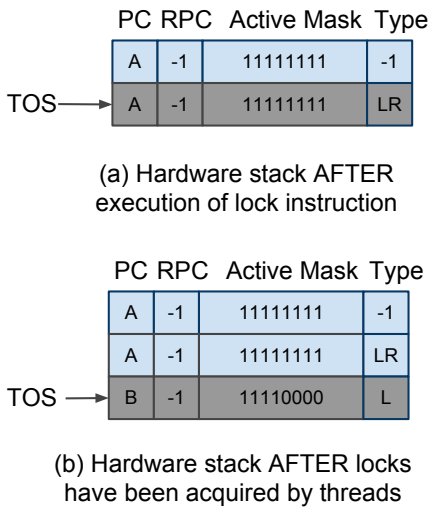

Figure 4.1: Hardware Stack During Execution of Lock Instruction (Grey Indicates Newly Added Stack Entries).

4.1 Lock Instruction

The goal of this instruction is to push entries on the stack that represent the threads trying to acquire a lock and the threads that succeeded in doing so. These entries coordinate the requirements of synchronization with the reconvergence mechanism.

The instructions send out an atomic memory request which performs the same function as the atomicCAS described in Chapter 1.3.3. The request replaces the value of the lock with the unique thread index if the lock is available and sends a reply to the core indicating the success or failure of acquiring the lock. Once the write requests have been sent a *lock retry* entry is pushed on to the stack. An extra field is added to the stack to indicate the *lock retry (LR)* entry as displayed in Figure 4.1(a). The *Active Mask* of this entry is copied from the previous TOS entry and represents all the threads that have sent out write requests to acquire the lock. The *NPC* is the PC of the lock instruction and the *RPC* of this entry is set to -1. This prevents the entry from being popped by the reconvergence mechanism. The warp is taken out of the scheduling queue and is rescheduled only when all threads have received a reply for their atomic write request.

If any of the threads have succeeded in acquiring the lock, a *lock entry(L)* is pushed on to the stack as show in Figure 4.1(b). The bits of the *Active Mask* are set to 1 for all the threads that acquired the lock. The *NPC* is the address of the instruction following the lock instruction and the RPC is set to -1 similar to the *lock retry* entry. The TOS entry of the stack now allows threads that have acquired their locks to execute the critical section.

However, if none of the threads succeeded in acquiring the lock, no new entry is added on to the stack. In this case the TOS entry of the stack is the *lock retry* entry and when the warp is executed the threads will again attempt to acquire locks. This process is repeated until at least one of the threads is able to acquire a lock.

4.2 Unlock Instruction

The *unlock* instruction determines when entries are popped from the stack. The reconvergence mechanism of the hardware stack pops entries from the stack when there *NPC* is equal the *RPC*. Since, the *lock* and *lock retry* entries have an *RPC* of -1, these entries never satisfy this condition and cannot be popped by the reconvergence mechanism.

The instruction sends out an atomic request which functions like the atomicExch described in Chapter 1.3.3. It release's the lock by setting its value to -1. The *Active Mask* of the *lock* entry is XORed with that of the of the *lock retry* entry and then it is popped from the stack as shown in Figure 4.2.

Since the boundaries of the critical section are not known when requesting the lock, a thread waits until it reaches the unlock instruction. The instruction following the unlock instruction (*D* in Example 9) is to be executed when all threads have finished their critical section. Thus the *NPC* of the entry below the *lock retry* entry is update to the this PC as shown in Figure 4.2.

Once the *lock* entry has been removed, the *Active Mask* of the *lock retry* entry is checked. If it is zero, all threads have finished executing their critical section and this entry can be popped from the stack. The next time this warp is scheduled, the *NPC* of the TOS entry is the instruction after the *unlock* instruction and the warp execute code after the critical section.

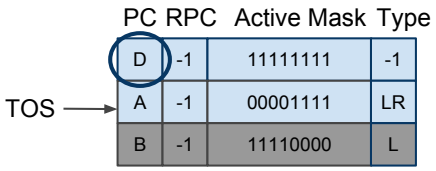

Figure 4.2: Hardware Stack After Execution of Unlock Instruction (Grey Indicates Stack Entries That Have Been Popped).

4.3 Interaction of Control Flow and Synchronization.

The unlock instruction described above will function when there is no divergence among the threads entering the critical section. However, as seen in the excerpt of code from radiosity in Example 10, this is not always the case.

Example 10 Excerpt from Radiosity application in SPLASH-2 benchmark suite

```
while (...){
  if(...){
      LOCK();                    (A)
      if( ... ){                 (B)
                  .
                  .              (C)
                  .
              UNLOCK();          (D)
              break ;            (E)
      }
      UNLOCK();                  (F)
  }
}                                (G)
```

In this example, we are presented with two issues:

1. The branch instruction inside the critical region has an *RPC* that is after the unlock instruction. When a divergent thread group from this branch reaches the unlock instruction, the stack will contain two additional entries above the *lock* entry and cannot directly pop the lock entry.

2. The multiple unlock instructions mean that after all threads finish their critical section, they will not all continue execution from the same PC.

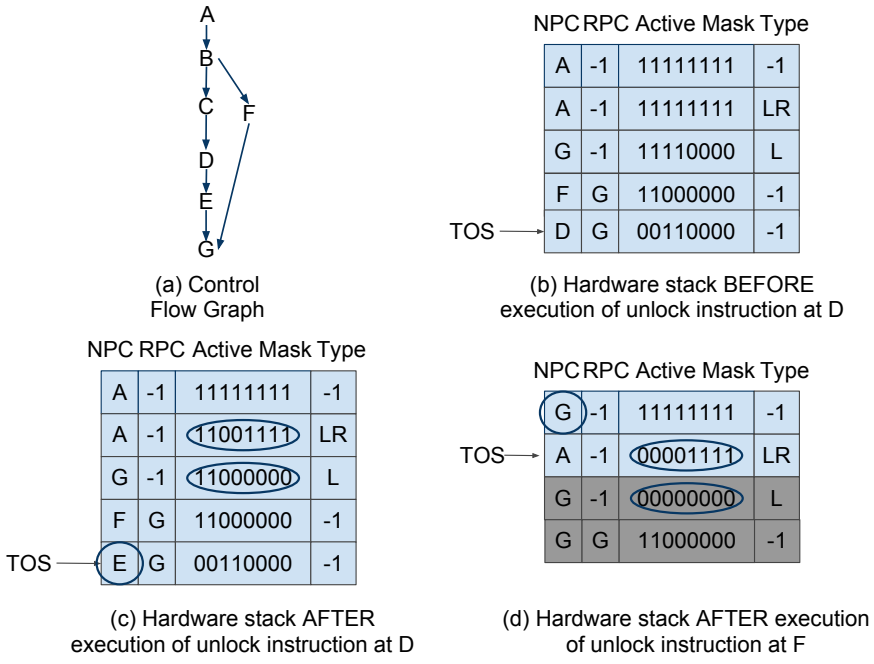

Figure 4.3: Control Flow Graph and Hardware Stack for Example 10 (Grey Indicates Stack Entries That Have Been Popped)

To handle the above issues, the *unlock* instruction is adjusted to **not** pop the *TOS* entry. Figure 4.3(a) shows the CFG for Example 10 and Figure 4.3(b) shows the hardware stack before execution of the *unlock instruction* at *D*. When the *unlock* instruction is executed, the stack entries below the *TOS* entry are searched to find the *lock* and *lock retry* entry. The *Active mask* of the *TOS* entry will be XORed with these two entries to indicate that they have finished the critical section as show in Figure 4.3(c). Instead of popping the *TOS* entry, the *NPC* will be updated. The subwarp continues execution until it reaches *G*, the RPC. Now, the reconvergence mechanism will pop the TOS entry. Next, the other subwarp executes the unlock instruction at *F*. The process described above is repeated and the *lock* entry becomes the *TOS* entry as shown in Figure 4.3(d). However, since its *Active mask* is zero, it is also popped from the stack. The *NPC* of the entry below the *lock retry* entry is also updated to the *NPC* of the *lock entry*. This last step ensures that

threads that diverged inside the critical section reach their RPC and continue at the correct PC once they have finished their critical section.

4.4 Nested Locks

Mutual exclusion is not restricted to one lock. Threads might need to acquire several locks before entering a critical region or acquire another lock during their execution. This would require management of multiple *lock* and *lock retry* entries on the stack. The subtle interactions of these entries is explained using Example 11

Example 11 Nested Locking

```
Lock(lockAddress_1);                    (A)
Lock (lockAddress_2);                   (B)
// Critical Section                     (C)
if (..){                                (D)
      // Some code                      (E)
     Unlock(lockAddress_2);             (F)
}else{
     // Other code                      (G)
     Unlock(lockAddress_2);             (H)
}
//More Code                             (I)
Unlock(lockAddress_1);                  (J)
```

In this example there are three locks at addresses X, Y and Z. Thread 1 requires lock X and Y and thread 2 requires lock Y and Z. A classic synchronization method to avoid deadlocks is to introduce ordering among the locks. In this example the address for locks $Z > Y > X$ and the threads have to acquire the lock with the lower address first. At A, thread 1 in the warp acquire lock at address X and thread 2 acquires lock at address Y. At B, thread 2 acquires Z but thread 1 cannot acquire Y and waits. Thread 2 executes the code from C and release lock Z at F. As per the mechanism described in Chapter 4.3, the *lock entry* is popped from the stack. The *lock retry* entry for the lock at B is at the *TOS* and thread 1 will contend for lock Y. However, thread 2 has not released lock Y as it has not executed the unlock at J. This causes a deadlock even though lock ordering is supposed to prevent it. This scenario can also occur between threads of different warps if they are holding

locks required by each other at different nesting depths.

To avoid deadlocks, we introduce a mechanism where threads that have acquired locks must release all of them before allowing any other threads in the warp to contend for locks. This is done with the use of counters to track the *nesting depth* which is defined as the number of locks a thread has not yet released. A per warp *lock counter* is incremented every time a *lock retry* is pushed on the stack. A per thread *unlock counter* is incremented whenever an unlock instruction is executed. The *unlock counter* is subtracted from the lock counter to determine the *nesting depth* of the thread. This *nesting depth* along with the *unlock counter* will determine which stack entries should be modified.

Example 11 is reused to explain this mechanism. Figure 4.4(a) shows the control flow graph for this example. Threads 1,2,3,4,7 and 8 acquire locks at *A* and threads 1,2,3,4 acquire locks at *B*. At *D*, the branch instruction causes thread 1 and 2 to diverge as shown in Figure 4.4(b). After executing the unlock at *F*, the *unlock counters* for threads 1,2. Once the *unlock counter* is incremented, the *nesting depth* is obtained by subtracting it from the *lock counter*.

If the *unlock counter* is 1, the threads have just released the most recently acquire lock and still have more locks to release. In this case, the *Active Mask* of the *TOS* entry is XORed with the *lock retry* entry as shown in Figure 4.4(c). The *lock* entry is left unchanged and will be used to execute the code at *I*. Since the RPC for the branch at *D* is *I*, the stack entry for threads 1,2 is poped after executing *F* (if the RPC was at a later point, the threads would have continued execution and no changes would have been made to the *TOS*). Threads 3 and 4 execute the unlock instruction at *H*, repeat the above process and converge with threads 1 and 2 at *I* as shown in Figure 4.4(d). Since the *unlock counters* are per thread, all four threads will now have the same value of 1. These threads continue to *J* and execute the unlock instruction, at which point their unlock counters will be incremented to 2.

When the *unlock counter* is greater than 1, the *nesting depth* is calculated by subtracting the *lock counter* from the *unlock counter*. The *nesting depth* (zero in this case) is used to find the*lock retry* entry (bottom most in this case) for the lock being released. The *Active Mask* of the *TOS* entry is XORed with **all** stack entries between the top most *lock retry* entry and the *lock retry* entry is found using the *nesting depth* as seen in Figure 4.4(e). This stack adjustment reflects the fact that

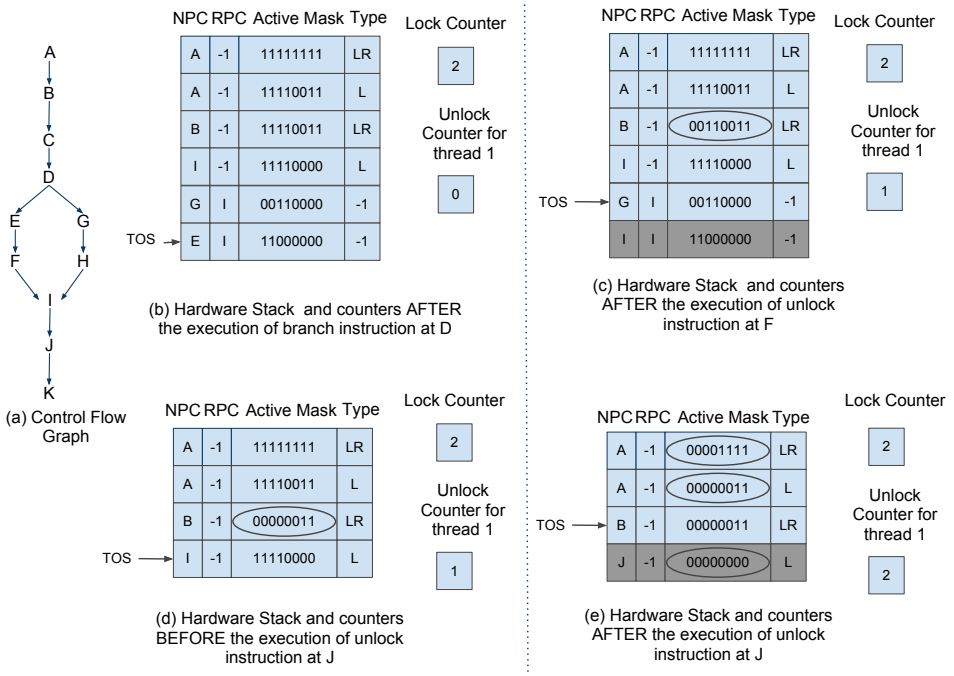

Figure 4.4: Control Flow Graph and Hardware Stack for Example 11 (Grey Indicates Stack Entries That Have Been Popped)

these threads have already executed the code at *I*.

This mechanism can handle any level of nesting and continues till the *nesting depth* value is zero. This indicates that the last lock has been released and the *Active Mask* of the *TOS* entry is XORed with the top most *lock entry*. In this example, the *TOS* and *lock entry* are the same when the *nesting depth* becomes zero. Therefore, the *active mask* becomes zero and the *lock* entry is removed from the stack as shown in Figure 4.4(e). Now threads 1,2,3 and 4 have released all the locks they have acquire and threads 7,8 can contend for the locks at *B*. These threads will repeat the same mechanism till they have executed both unlock instructions. At this point, the active mask of all entries above the bottom most *lock retry* entry will be zero and the entries will be removed. This will allows threads 5 and 6 to contend for locks at *A*.

The per thread *unlock counter* is reset when the nesting depth becomes zero and the *lock counter* is decremented when a *lock retry* entry is popped from the stack.

The PC of the instructions at which all threads should continue after finishing the critical section is updated when the *lock counter* value becomes zero.

Note, the mechanism described here is different from that of section 4.3 where there is control flow divergence and only one lock. The *lock counter* is examined to determine which mechanism to use. If it is greater than 1, the process described here is used, otherwise the unlock instruction behaves as described in section 4.3. This gives us the flexibility to handle nested locks and some control flow divergence.

4.5 Limitations of the Solution

In examining the PARSEC [9] and SPLASH-2 [48] multithreaded benchmarks suites, we could not identify any synchronization code that could not be implemented by the proposed instructions. However, in the interest of providing a comprehensive solution we identify areas where the solution falls short.

Consider a scenario when a divergent instruction outside the critical section has an *RPC* inside the critical section as shown in Example 12. The *RPC* for the branch *A* is *E* and upon execution of this branch, two stack entries are added with the *RPC E*. The subwarp that branched into *B* completes its critical section, removes the *lock* and *lock retry* entries at *F* and update its NPC to *G*. Now the entry for this subwarp cannot be popped as it cannot reach its *RPC* at *E* and this leads to deadlock.

One way of overcoming this issue is to use compiler directives to mark the *lock* and *unlock* instructions. The compiler must then ensure that any branches outside this region must have an *RPC* that is also outside this region. In Examples 12, if the *RPC* of the branch instruction at *A* is set to *G* instead of *E*, there will be no deadlock. Moving the *RPC* of a branch instruction would have a negative effect on the SIMT efficiency of this code as it causes larger sections of code to be executed by diverged subwarps and not the whole warp. The magnitude of this effect would depend on the size of the critical section and is the trade off for simplifying the implementation of this scenario on a GPU. These compiler modifications are outside the scope of this work.

Nested locking suffers from the same issue along with an added problem. Ex-

Example 12 Code scenario that limits the solution

```
if( ...){                    (A)
    Lock (lockAddress);      (B)
    //Critical Section       (C)
}else{
    Lock (lockAddress);      (D)
}
//  More code                (E)
Unlock(lockAddress);         (F)
// More Code                 (G)
```

ample 13 shows code that seems like it could be handled by the proposed solution. However, let us take a case where threads 1 and 2 acquire locks X and Y at A. Then, threads 1 and 2 diverge at B with only thread 1 wanting to acquire lock Y at C. This causes a deadlock as thread 2 cannot execute until thread 1 reaches the RPC at F and cannot release lock Y. This creates a situation where a thread is waiting for a lock held by another thread which is not at the top of the reconvergence stack. Since this other thread is not active, it cannot make forward progress and causes deadlock.

Example 13 Code scenario that limits nested locking solution

```
Lock(lockAddress1)           (A)
if( ...){                    (B)
    Lock (lockAddress2);     (C)
    //Critical Section       (D)
    Unlock(lockAddress2);    (E)
}
//  More code                (F)
Unlock(lockAddress1);        (G)
```

Chapter 9 discusses future work that could address these and other shortcomings.

Chapter 5

Queuing Lock Requests

The lock instruction described in section 3.1 can be categorized as a *busy wait* synchronization. This means every time a thread receives a reply that indicates failure to acquire the lock, it will send out another request. This is repeated until it succeeds in acquiring the lock. If threads are waiting for a lock that is currently being held, they will use up resources sending lock requests and receiving failure replies for the duration of the critical region. These requests cause contention in the interconnect and memory system and degrade performance. This situation is even more drastic on a GPU where programs can contain thousands of threads, potentially consuming a significant amount of resources. Queuing up the requests for an acquired lock until its released would eliminate the need for failure messages. This is the motivation behind the research into lock queuing on GPUs.

5.1 Related Work

To the authors knowledge, queuing of lock requests on a GPU is a novel proposal. However, there is significant literature on lock queueing in the context of CPUs, which is discussed in this section.

5.1.1 Queue on Sync Bit Primitive

Goodman et al. propose the first hardware queue synchronization primitive known as *Queue on Sync Bit* [15]. A sync bit as described by Goodman et al "enforces

mutual exclusion when a particular protocol is followed by the parallel tasks". The QOSB primitive adds the thread requesting synchronization to a syncbit queue. It is used in addition with a *Test and Set* operation to implement synchronizations. Example 14 taken from Goodman et al.[15] shows a spin lock implemented using the QOSB.

Example 14 Spin Lock using OQSB from Goodman et al. [15]

```
procedure lock (addr)
begin
    QOSB(addr)
    while(TEST_AND_SET(addr)) do
       QOSB(addr)
end
```

In this example, the QOSB primitive adds the thread to the queue and the Test and Set operation returns true if the thread is at the head of the queue. The QOSB primitive is ignored if the thread is already in the queue. The Test and Set spins on a flag local to the thread which avoids sending messages over the interconnect. When a request reaches the head of the queue, the requesting thread is informed, which allows the Test and Set to succeed.

5.1.2 Array Based Lock Queuing

Anderson proposed a software mechanism that uses array based lock queuing [5]. A thread requesting a lock, reads a counter for that lock, performs an atomic increment and gets a unique sequence number. When a thread has finished executing its critical section, it passes the lock to the next sequence number. Example15 shows pseudo code for the lock and unlock instructions as provided by Anderson [5]. Graunke and Thakkar [16] independently proposed a similar algorithm that uses atomic swaps instead of increments. In both these proposals, threads spin on *local* locations which avoids sending messages over the interconnect.

Experimental results prove that array based locking scales better than locks implemented with test and set. However this mechanism was not compared to the QOSB proposal from Goodman et al. [15]. The storage required for array based locks is proportional to the number of threads being executed.

Example 15 Spin Lock from Anderson [5]

```
Init        flags[0]:= HAS_LOCK;
            flags[1...P-1]:= MUST_WAIT;
            queueLast:=0;

Lock        myPlace:= ReadAndIncrement (queueLast);
            while(flags[myPlace mod P] == MUST_WAIT);

Unlock      flags[myPlace mod P] := MUST_WAIT;
            flags[(myPlace + 1) mod P] := HAS_LOCK;
```

5.1.3 Lock Queuing Using Linked Lists

Mellor-Crummey et al. proposed using linked lists to queue lock requests [30]. This software proposal called the MCS lock allocates a record for each thread requesting a lock in a memory location local to the thread. The lock address contains a pointer to the tail of a queue of these records. Each new request is added to the end of the queue and the tail pointer is updated. The thread that releases the lock uses these links to pass the lock to the next thread. Example 16 taken from [30] shows the implementation of this lock. Craig [11] and Magnusson et all [28] also proposed lock queues that used linked lists with each thread linked to its predecessor instead of its successor.

Experiments by the Mellor-Crummey et al. proved that the MCS lock performed better than array based locking on non cache coherent machines. This is because spinning on *local* locations cannot directly be implemented without coherent caches. On cache coherent machines, both methods performed similarly. The MCS lock requires space linear only to the number of locks being acquired by a processor which is an improvement over array based locking.

5.2 Lock Queuing on GPUs

The prior work on lock queuing took advantage of existing cache coherence protocols or relied on fast locally accessible memory. At the moment GPUs do not have either of these. L1 caches are not coherent and the shared memory per shader core is accessible only to threads in a CTA. Hence, it cannot be used to commu-

Example 16 MCS lock from [30]

```
type qnode=record
   next:qnode
   locked:Boolean
type lock=qnode

//parameter I below points to a qnode record allocated
//(in a enclosing scope) in shared memory locally accessible
// to the invoking processor
procedure acquire_lock(L: lock, I: qnode)
    I->next=nil
    predecessor:qnode= fetch_and_store (L,I)
        if predecessor != nil //queue was non empty
    I->locked=true
    predecessor->next=I
    repeat while I->locked //spin

procedure release_lock(L: lock, I:qnode)
   if I->next=nil //no known succesor
      if compare_and_swap (L, I, nil)
          return // compare and swap return true iff it swapped
      repeat while I->next=nil //spin
   I->next->locked=false
```

nicate between threads of different blocks or shader cores. Instead to implement lock queuing on GPUs we propose adding additional hardware to the memory controllers.

5.2.1 Queueing Requests in GPU Memory Partition

A GPU memory partition consists of the memory controller, L2 cache and ports to the off chip DRAM. The memory controller is responsible for processing incoming memory requests, either from the cache or by forwarding these requests to the DRAM.

A table is added to this memory controller that stores the head and tail pointers for queues. When a thread sends a lock request using the proposed instructions, the memory controller will check this table to see if the lock address has a queue. If there is no queue for this address, a new entry and a new queue record are allocated

and the table is updated. However, if a queue already exists, a new queue record is allocated and added to the end of the list. Storing the tail pointer speeds up insertion by eliminating the need to traverse the whole list. Each queue record stores the unique thread identifier and a pointer to the next queue record. The queue record themselves can be stored in memory and cached in the L2.

If a queue is created or a queue record moves to the head of the queue, the lock is available and the request can be processed. The queue record of the thread holding the lock remains at the head of the queue for the duration of the critical section. It is popped from the queue by the unlock instruction at which point the lock passes to the thread in the next queue record.

This design is based on the MCS lock queue described in section 5.1 of this chapter and the storage space required is linear to the number of threads acquiring locks. However, in our experiments we use an idealized model where this is no limitation on the size of a queue or the number of queues. A more realistic model could limit both of these factors and implement a back off mechanism for threads when the storage limit is reached. This is an area of future work and is discussed in Chapter 9. Additionally due to time constraints, we do not model the DRAM operations associated with adding and removing and entries from the queue. We hypothesize that If these queuing operations become a bottleneck to performance, an additional SRAM could be added to store the queues and therefore these additional operations are not a significant design constraint.

5.2.2 Interaction of Nested Locks with Lock Queuing

Queueing of lock requests for nested locks creates a deadlock condition similar to the one described in Chapter 4.5. Without lock queuing, when threads in a warp request locks, the warp waits for all replies to come back and only schedules threads that have acquired the locks. However, with lock queuing, any thread that receives a reply will immediately be scheduled for execution. This cause divergence with the top stack entry containing the active thread. It is possible that during the execution of the critical section, other threads in the warp receive replies indicating successful lock acquisition. If the active thread requires a lock that has been acquired by a thread not at the top of the stack, it causes a deadlock. This limitation

prevents us from implementing lock queuing for nested locks.

Chapter 6

Methodology

A modified version of GPGPU-Sim (version 2.1.1b) [7] is used to model the proposed changes. GPGPU-Sim is a cycle accurate GPU simulator that does performance simulations of CUDA and OpenCL programs. The next section gives an overview of the simulator followed by the change made to implement the proposed instructions.

6.1 Background on GPGPU-Sim

GPGPU-Sim models the architecture described in Chapter 1.3. Applications to be executed on the simulator use the same tool chains that a normal GPU program would use. CUDA programs are compiled with *nvcc* (a NVIDIA compiler) and OpenCL program with any C compiler that supports the OpenCL extensions. The compiler generates an executable containing *PTX* code that is to run on the GPU and host code that runs on the CPU. Upon execution, the program would normally use the CUDA or OpenCL runtime libraries to interface with the GPU. However, GPGPU-Sim implements the API for CUDA and OpenCL providing custom shared runtime libraries. By dynamically linking the executables to these custom libraries, the GPU code is executed on the simulator instead of the GPU hardware.

GPGPU-Sim provides a functional simulator that executes the PTX assembly code and a performance simulator that models the GPU microarchitecture. Parallel Thread Execution (PTX) is a pseudo assembly language used by NVIDIA and is

described as an ISA for general purpose parallel thread execution[38]. As the name suggests, the functional simulator executes the PTX instructions and mimics what a GPU program would expect from hardware. It also provides input to the performance simulator that models the Shader Core pipeline, interconnect and memory systems. Each of these subsystem are described in the following sections.

6.1.1 Shader Core

The shader cores consist of a five stage in order pipeline with no forwarding. The stages are simulated in reverse order, which is a well known simulator design. It avoids having two copies of the same pipeline register. During the execution of a stage, the pipeline register that feeds the next stage is checked. If the register is non-empty, it indicates a stall in the next stage and the current stage is stalled as well. Regardless of a stall, the current stage is executed as normal. However, the pipeline registers are copied to the next stage only when a stall is not detected.

The fetch stage selects a warp for execution from a pool of ready warps. A ready warp is one where no active threads in the warp are waiting for memory operations to complete. It fetches the instructions for the warp from the instructions buffer and copies the PC and thread IDs into the pipeline of the decode register. It also marks the warp as issued to prevent it from being rescheduled until it has passed through the pipeline.

The decode stage functionally executes instruction for all active threads in the warp. It determines the Next PC for every thread and this information is used by the performance model to handle branch divergence. The branch divergence model used is the model described in Chapter 1.3.2. The functional simulator also determines the addresses for any memory operations required by the thread. The performance simulator then coalesces these requests and generates memory request packets. Coalescing is a process where memory requests from scalar threads are checked to see if they are contiguous. If so, the requests are combined into fewer wide memory accesses. This reduces the number of messages required between shader cores and DRAM and increases efficiency

The execute stage is not modelled in GPGPU-Sim as the functional execution is already performed in the decode stage. The simulator moves on to the memory

stage instead. This stage examines the coalesced memory requests and forwards them to the appropriate memory unit. Requests are classified as shared memory or global memory. For shared memory requests, bank conflicts are checked and the requests are returned after the appropriate number of cycles. For global memory requests, the local cache is first checked for hits. If any accesses miss in the cache, the memory stage forwards a request to the memory system through the interconnect. The warp is marked as waiting for memory operations and will not be scheduled until a reply is received.

The writeback processes replies from the memory system. It updates the status of warps to ready when all its outstanding requests have been received. This will allow the fetch stage to continue execution of the warp.

6.1.2 Interconnect

The interconnect system transports messages from the shader core to the memory system and vice versa. *Booksim* a flit level simulator is used to implement this model [1]. GPGPU-Sim interfaces with the interconnect at the memory stage of the shader core and the L2 cache of the memory unit. The interconnect system can be configured through configuration files and can model different topologies, routing model, virtual channels, flow control models etc. It is only aware of the size of the packet being transmitted and does not operate on its contents.

6.1.3 Memory Unit

The memory space of a GPU is split into several memory partition units. Each unit contains, the L2 cache, memory controller and DRAM chips for a portion of the memory space. Before being injected into the interconnect, memory requests are decoded to determine which memory partition unit they must be sent to. On arrival at a unit, requests that miss in the L2 cache are forwarded to the DRAM. The DRAM access model is detailed and takes into account the latency for row accesses, column accesses, pre charges etc.

The atomic instructions described in Chapter 1.3.3 are implemented in GPGPU-Sim by using callback functions. In this model, the decode stage generates memory requests for atomic instructions and sets function pointers. These functions are then

executed when the memory request has reached the memory partition unit. This mimics the execution of atomics in hardware, which is performed by ALU units in the memory controller.

6.2 Implementing Lock and Unlock Instructions

To simplify the implementation and avoid modification to the nvcc compiler, the *lock* and *unlock* instructions are modelled as function calls. This allows us to use the instructions in benchmarks by adding empty function definitions. The functional calls are intercepted by the simulator and treated as instructions. Similar to the execution of atomics, they set callback functions and generate memory requests in the decode stage. Each warp keeps count of the lock requests sent out and the warp is marked as *not ready* till all outstanding requests have been processed. This prevents the fetch unit from scheduling the warp for execution. The memory requests generated are also marked to indicate that they are lock or unlock requests.

When the memory requests reach the memory partition unit, they are processed by the memory controller. It is here that the callback functions are executed, which check the availability of locks and acquire or release the locks. The replies generated by these requests are forwarded back to the shader cores through the interconnect. Upon arrival at the shader cores, the writeback unit examines the replies to determine which threads have acquired the locks. The writeback unit waits till all outstanding lock/unlock requests for active threads in a warp have returned. The writeback unit then performs the updated to the PDOM mask as described in Chapter 4.1 and 4.2. The warp is then marked ready and can be scheduled by the fetch unit for execution.

6.3 Implementing Queuing of Lock Requests

To implement lock queuing, the memory controller was modified to create a map of addresses and queues. Every lock request is checked against this map. If the queue is empty, the request is added to the head of the queue and processed. If the queue is non-empty the request is added to the queue and removed from the memory pipeline. Unlock instructions are allowed to pass through the memory pipeline as normal. After the execution of an unlock instruction, the requests at

the head of the queue is removed and the next request is inserted into the memory pipeline. If this next request cannot be inserted into the DRAM memory access queue due to resources constraints, it will be inserted into a separate buffer. On the next cycle, requests in this buffer are given priority over incoming memory requests from shaders to avoid starvation of lock requests.

In this implementation, outstanding lock requests in a warp might not all come back at the same time. The writeback unit is modified to account for this. If a reply is received indicating success in acquiring a lock, the PDOM mask of the warp is updated and the warp can be scheduled for execution regardless of the number of outstanding requests. This means that replies to lock requests can come in when a warp is executing the critical section. If this happens, the writeback unit will not update the PDOM mask. Instead, when the warp finishes executing an unlock instruction, the other active threads in the warp trying for locks are checked. If any of them have acquired locks, the PDOM stack is updated accordingly and the warp is scheduled for execution. This ensures that threads in a warp requesting different locks can receive replies in any order.

6.4 Baseline Configuration

The modified GPGPU-Sim models the architecture described in Chapter 1.3.1. Table 6.1 shows the major configuration parameters used. Section 6.5 discusses the benchmarks used in this study. This configuration is similar to an NVIDIA's Fermi architecture [36] and is used to evaluate the proposed instructions.

6.5 Benchmarks

Apriori, *BarnesHut* and *interac* are the benchmarks used in this thesis. They are discussed in the following sections.

6.5.1 BarnesHut N-Body algorithm

N-Body algorithms calculates the force on a particles from all other particles in the system. The Barnes-Hut N-Body algorithm has an $O(nlogn)$ when compared to a direct sum algorithm of order $O(n^2)$ [8]. The algorithm is given the location of

Table 6.1: GPGPU-Sim Configuration

# Streaming Multiprocessors	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L2 Unified Cache	1MB/Memory Channel, 64B line, 64-way assoc.
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Branch Divergence Method	PDOM [14]
Warp Scheduling Policy	Loose Round Robin
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

each particle in the system as an input. It starts by creating a cube that encloses all the particles in the system. The cube is then divided into 8 equal quadrants knows as cells. Each cell is repeatedly divided into 8 cells until each particle in the system is in its own cell. This information is stored by creating an *Oct-tree*, a tree where each node has eight children. The nodes of the tree are the cells and the leaves are the particles. When calculating force on a given particle, the algorithm only considers particles from nearby cells individually. Particles from distant cells are treated as a single large particle located at the centre of mass for these distant particles.

This algorithm was ported to CUDA by Burtscher et al. [29]. It uses atomics to implement synchronization when multiple threads access the Oct-tree. A thread traverses the tree to a leaf node before attempting insertion. Each leaf must be locked by a thread before inserting a particle. If it is an empty leaf, the particle is inserted. If not, the leaf is converted into a node and new leafs are added to it. The existing particle and the particle being inserted are then distributed to these leaf nodes. Example 17 shows the synchronization code used for this purpose. This

Example 17 Synchronization in BarnesHut from [29]

```
// initialize
cell = find insertion point(body);
child = get insertion index(cell, body);
if (child != locked) {
// ACQUIRE LOCK
 if (child == atomicCAS(&cell[child], child, lock)) {
     //critical section
     child=-2 // RELEASE LOCK
   }
}
syncthreads(); // wait for other warps to finish insertion
```

synchronization is non-blocking meaning it returns after a single attempt, regardless of success or failure in acquiring the lock. The *if (child!=locked)* statement before the lock, checks the lock availability and reduces contention by not sending a request if the lock is already taken. If the lock availability check fails or if the thread cannot acquire the lock, it traverses the Oct-tree again to a leaf node(this traversal code is omitted from Example 17. Example 18 shows the same code with the proposed instructions.

As explained in Chapter 4.1, the lock instructions does not return until the lock is successfully acquired. This results in a subtlety in Example 18 which is best explained by an example. Two threads X and Y are contending for the same leaf. This leaf already has a particle in it but is not locked by any other thread. Since the lock is available, both threads pass the lock check and execute the lock instruction. Thread X succeeds in acquiring the lock, upon which it expands the Oct-tree and converts the leaf into a node. Thread X releases the lock and it is then acquired by Thread Y. Thread Y must not insert into this node as it is not a leaf. Additional code was added to the check for this scenario and immediately release the lock if it is encountered.

This situation does not occur in the original code show in Example 17, where thread Y would have failed to acquire the lock on the first attempt and re-traversed the tree. The lock availability check would prevent thread Y from attempting the lock on the same node again and it will spin in the tree traversal loop. When

thread X release the lock, it would have already expanded the tree and thread Y will traverse to a new leaf node before trying to acquiring a lock. This form of synchronization that returns after a single attempt is called a *try lock*. Extending the proposed instructions to implement *try locks* is an area of future work and is discussed in Chapter 9.

Burtscher et al. state "As long as at least one thread per warp succeeds in acquiring a lock, thread divergence temporarily disables all the threads in the warp that did not manage to acquire a lock until the successful threads have completed their insertions and released their locks" [29]. From this we infer that the authors required an understanding of the SIMT behaviour to optimize this code. The *syncthreads()* in Example 17 is another example of the same. The reason for which is described by Burtscher et al. as "This barrier throttles warps that would otherwise most likely not be able to make progress but would slow down the memory access of the successful threads". The benchmark was chosen to show that the proposed instructions can achieve similar performance and free the author from understanding the intricacies of the SIMT behaviour.

Example 18 Synchronization using proposed instructions in BarnesHut

```
// initialize
cell = find insertion point(body);
child = get insertion index(cell, body);
if (child != locked) {
     lock (child) // ACQUIRE LOCK
       //critical section
     unlock (child) // RELEASE LOCK
}
```

6.5.2 Apriori

A data mining application from the Minebench benchmark suite [34] described in Chapter 2.

6.5.3 Interac

A microbenchmark that simulates bank transactions that run in parallel. A transaction consists of two accounts, with money withdrawn from one and deposited into the other account. Before executing a transaction, a thread must acquire locks on both the bank accounts. To avoid deadlock, threads acquire the locks in order i.e the lock for the smaller account number is acquired first. This is a classic deadlock avoidance technique used when threads have to acquire more than one lock. Example 19 shows the nested locks implemented using atomics and loops. We would like to note that the atomic implementation required some trial and error due to compiler optimization of loops. These optimizations caused the divergence problems described in Chapter 3.1. This is another example of complications that can be avoided by using the proposed instructions as shown in Example 20.

Example 19 Synchronization using atomics in Interac

```
while(!transaction_done) {
  if(has_acc1 || atomicCAS(&acc1->lock, 0, 1) == 0) {
    has_acc1 = 1;
    if(atomicCAS(&acc2->lock, 0, 1) == 0) {
      src->balance -= action->amount; // do transaction
      dest->balance += action->amount;
      acc2->lock = 0; // release locks
      acc1->lock = 0;
      transaction_done = 1;
    }
  }
}
```

Example 20 Synchronization using proposed instructions in Interac

```
lock(A);
lock(B) ;
src->balance -= action->amount; // do transaction
dest->balance += action->amount;
unlock(B); //locks release;
unlock(A);
```

Chapter 7

Experimental Results

The instructions proposed in Chapter 4.1 are evaluated on the benchmarks described in Chapter 6.5. Section 7.1 discusses the comparison of the proposed instructions against synchronizations using atomics. Section 7.2 shows the results of implementing lock queuing.

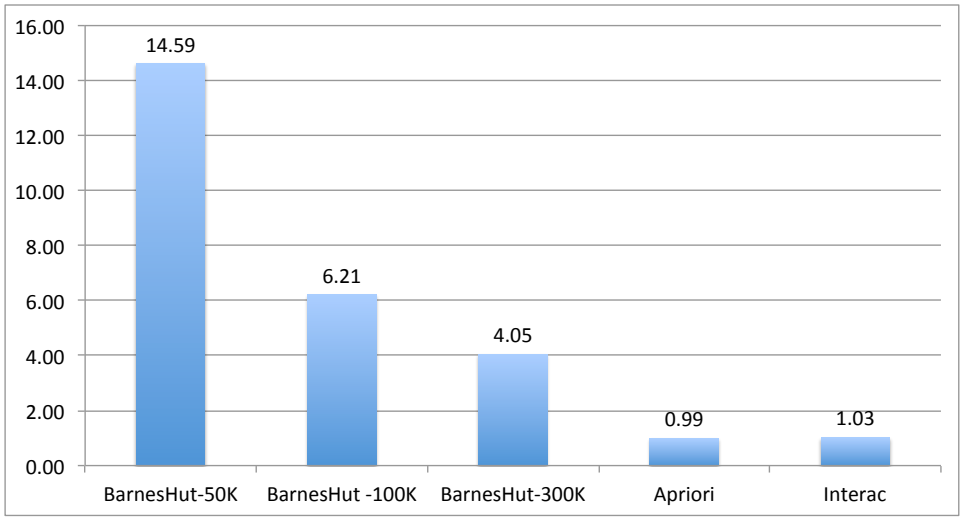

Figure 7.1: Normalized Execution Cycles of Proposed Instructions Against Unmodified Benchmarks That Use Atomics for Synchronizations

7.1 Performance of Lock and Unlock Instructions

Figure 7.1 shows the number of execution cycles of the proposed instructions normalized to the unmodified benchmarks that use atomics for synchronizations. In Figure 7.1, the number following BarnesHut is the size of the input data set (the number of particles being simulated) Apriori and interac achieve our goal of simplifying the implementation of synchronizations while providing similar performance. The slowdowns experience by BarnesHut can be attributed to two factors.

First, as explained in Chapter 6.5.1, our proposed instructions implement blocking synchronization and may result in threads acquiring locks, but performing no insertion into the tree. Even though no insertions take place, threads spend time waiting to acquire locks as opposed to the original benchmark where they are in a tree traversal loop. This wait time and the cycles required to execute the critical section with no insertion contribute to the slow downs seen for BarnesHut. Figure 7.2 shows the number of critical sections executed where no insertions take place normalized to the total number of critical sections executed. From this figure, we can infer that there is a direct relationship between the number of wasteful critical sections executed and the slowdown achieved.

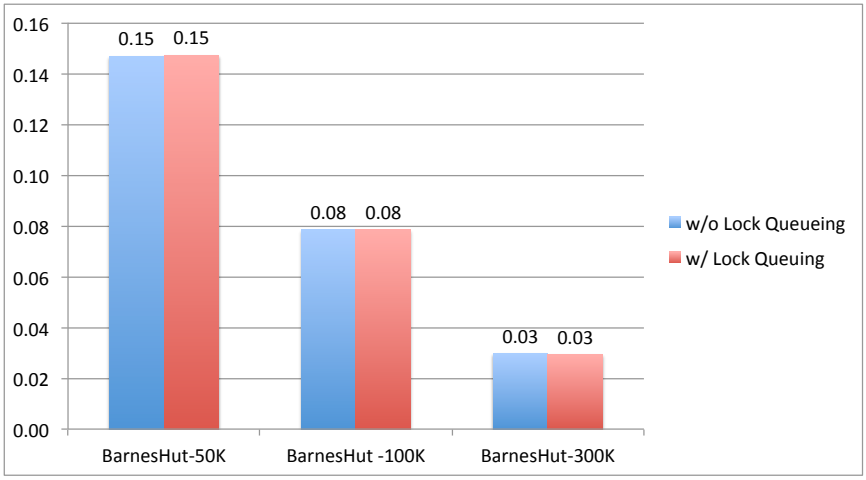

Figure 7.2: Number of Critical Sections That Do Not Insert into the Tree Normalized to Total Critical Sections Executed

Secondly, the proposed lock instruction keeps sending out requests till it ac-

quires the lock. Even with the lock availability check described in Chapter 6.5.1, it results in a large number of retries for lock requests. This causes contention in the interconnect and memory system and is another factor in the performance degradation. The implementation of lock queuing greatly reduces this contention and the results are discussed in section 7.2.

7.2 Performance of Lock Queuing

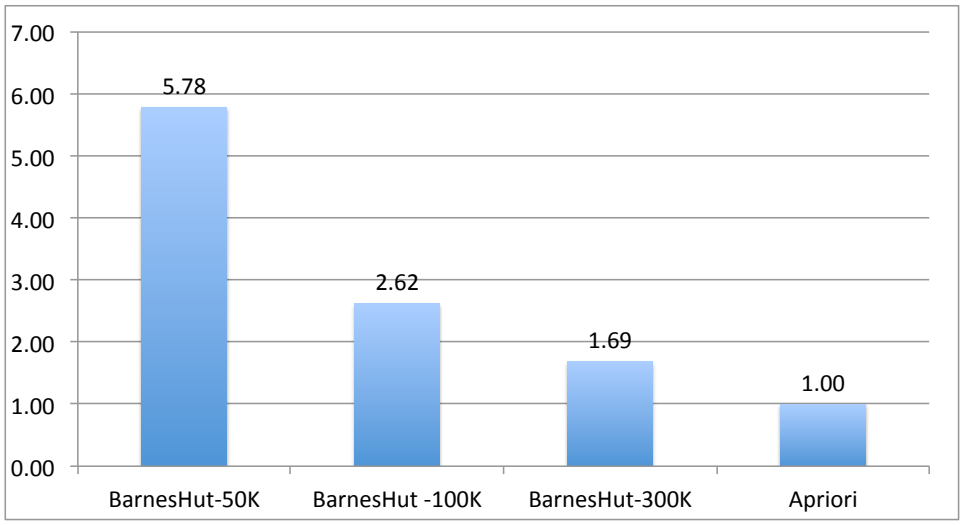

Figure 7.3: Normalized Execution Cycles of Proposed Instructions with Lock Queuing Against Unmodified Benchmarks That Use Atomic for Synchronizations

Figure 7.3 shows the execution cycles of the proposed instructions with lock queuing (as described in Chapter 5.2) normalized to the synchronization with atomics. The Apriori benchmark is not bottlenecked by contention in the interconnect and memory system and implementing lock queuing does not affect its performance by much. The complexities involved in implementing lock queuing for nested locks (as described in Chapter 5.2.2) prohibited the inclusion of the interac benchmark in this experiment.

As expected, all BarnesHut data sets show a significant improvement in performance with the implementation of lock queueing. The data packets sent by the

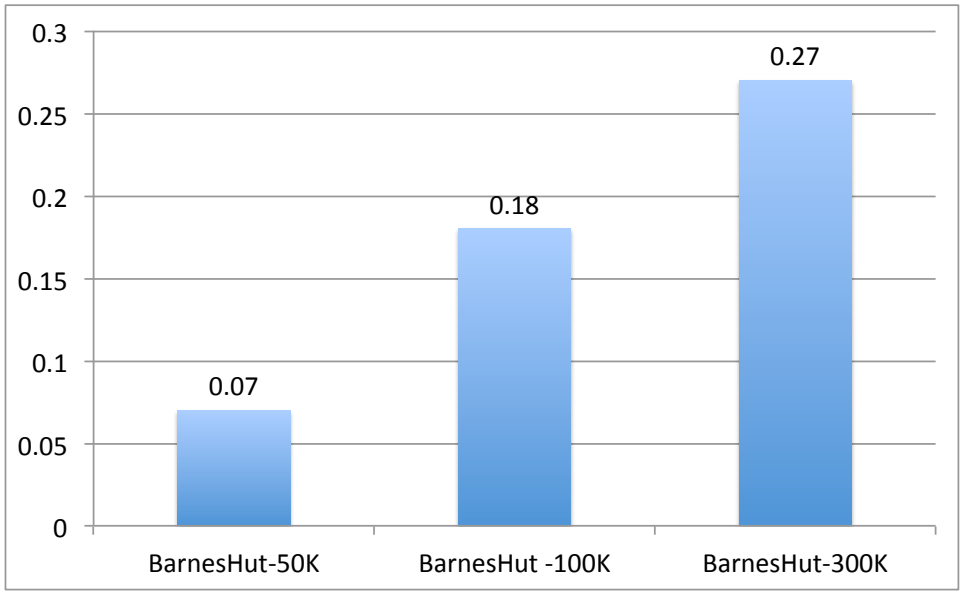

Figure 7.4: Number of Writes of an Implementation with Lock Queuing Normalized to One Without

lock and *unlock* instructions are modelled as memory writes in our experiments. Figure 7.4 shows the number of writes executed in the implementation with lock queuing normalized to the implementation without. These reductions in write traffic reduce the contention and contribute to the improved performance. However, the implementation of lock queuing does not effect the number of wasteful critical sections executed as shown in Figure 7.2. The remaining performance slowdown can be attributed to these critical section. It may be possible to eliminate this slowdown by implementing *try locks*, which is an area of future work and discussed further in Chapter 9.

7.3 Hardware Cost

The main cost of implementing the proposed instructions is the changes required to the hardware stack. Since the instructions can update entries that are not at the *TOS*, it changes the behaviour of the stack mechanism. Additional logic and registers would be needed to implement stack traversal capabilities. This logic will

Table 7.1: Storage Used for Lock Queuing

BarnesHut-50K	0.34 MB
BarnesHut-100K	0.63 MB
BarnesHut-100K	1.77 MB
Apriori	0.006 MB

only be used by the proposed instructions and the rest of the system will continue to use the stack in the traditional manner. Support for nested locking would require additional registers to store the per thread and per warp counter as described in Chapter 4.4.

In our experiments with lock queuing we use an idealized model where there is no limit on the number of requests that can be queued. Each queue request consists of an integer to store the thread ID and a pointer to the next queue request. Assuming 4 bytes each to store an integer and the pointer, Table 7.1 shows the **total** storage space in MB used by benchmarks to store requests. Even with this idealized model, the storage required is not a significant constraint.

The existing hardware stack is sized to handle divergence of every thread in the warp. Since the proposed instructions can add more entries to the stack, it could have more entries than the number of threads in the warp. To handle these extra synchronization entries, the depth of the stack would need to be increased with overflow being handled by spilling the stack to memory.

Chapter 8

Related Work

8.1 Synchronization using Full/Empty Bits in Memory

One method of implementing fine grained synchronization in multiprocessor architecture is to add extra bits to each memory location. This bit, known as the full/empty bit along with special instructions can implement synchronization directly in hardware. These synchronization instructions complete only if the memory location is in a pre-determined state or else they return a failure message. For example, a load to a memory location will check if the bit is full and a store will check if the bit is empty before executing. These instructions also change the full/empty bit appropriately. Tera [4] and Sparcle [2] are two multiprocessor architectures where this was implemented. The obvious overhead of this method is the storage required as every memory location requires a full/empty bit. In addition these instructions implemented non-blocking synchronization and software mechanisms were still required to manage retries.

8.2 Register Based Synchronization Mechanisms

The M-Machine[13, 18] associated full/empty bits with registers instead of memory locations. Writes to a register only succeed if bit is set to empty and threads could explicitly clear the bit after reading the register. This allowed waiting threads to stall as opposed to spinning, which is a form of blocking synchronization. Ex-

perimental results showed that register based communication performed better than associating bits with memory [18]. The Cray X-MP[6] also used this mechanism but decoupled the synchronization bits from the data registers by using separate one bit synchronization registers. These registers could be used to protect any of the data registers.

8.3 Synchronization State Buffer

The *synchronization state buffer (SSB)* proposed by Zhu et al. [50] is based on the observation that the number of synchronization variables in use at a given time, is usually smaller than the number of memory locations. It uses a separate storage buffer to keep track of the full/empty bits for the memory locations in use. In the context of producer-consumer synchronizations, if a producer's write operation arrives first, a SSB entry is allocated and a *success* message is returned to the thread. Any subsequent loads from consumers will also receive a *success* message. However, if a load from a consumer arrives first, a *wait* message is returned to the thread. The thread then goes to sleep until it is woken up by a write operation. This reduces the overhead of adding one bit to every memory location. The extended SSB from Ributzka et al. [40] builds on this proposal by eliminating the return messages. If a consumer's read arrives first, it stores the request in the E-SSB and does not return anything until the producer's write operation is complete. The processor uses scoreboarding to keep track of outstanding memory requests. This mechanism will allow it to schedule other instructions that are not related to the synchronization operation. This can be considered a form of lock request queueing, similar to methods described in Chapter 5.1.

8.4 Synchronization Primitives in SMT processors

Tullsen et al. [45] proposes a synchronization mechanism for simultaneous multi-threaded processors (SMT) with explicit *acquire* and *release* operations. This work takes into account the special properties of an SMT processor and tries to reduce the usage of all shared resources. It uses a *lock box* which contains **one entry** per hardware thread. Since threads cannot be blocked on more than one lock, this is sufficient to keep track of lock requests. When a thread fails to acquire a lock

using the *acquire* operation, the lock address and thread ID are stored in the lock box and the thread is blocked. When a *release* operation for a lock is executed, the processor checks the lock box and allows a thread blocked on this lock to execute. This mechanism differs from our proposal in that it does not have to deal with a reconvergence stack.

Chapter 9

Future Work And Conclusion

This chapter summarizes and the concludes the work presented. Finally it discusses areas of future work that could address the shortcomings identified throughout this paper.

9.1 Summary and Conclusion

The implementation of synchronizations on GPUs is complicated by the SIMT execution model of the GPU. We show how mutexes, semaphore and barriers do not behave as expected because of this SIMT model and the reconvergence stack it uses. We propose instructions that simplify the implementation of mutexes on a GPU. These instructions modify the hardware stack in a novel way, using special entries to manage the retries for lock requests. Further, the proposed solution is amended to handle scenarios where synchronization is interleaved with control flow (as seen in certain benchmarks). We expose the subtleties of implementing nested locking in a SIMT stack and provide a solution that can handle some common cases. To improve the performance of these requests, we propose implementing lock queuing in GPUs. This minimizes the synchronization messages sent over the interconnect and the pressure on the memory system. Finally we evaluate the performance of these instructions as compared to synchronizations implemented using atomics and show that have similar performance. These instructions represent the first step towards providing a wide range of synchronization primitives

for GPUs. They will aid the development of application on a GPU and improve programmer productivity.

9.2 Future Work

In this section we discuss proposals that can improve the performance of these instruction. We also present other areas of research that this work could lead to.

9.2.1 Backoffs for Lock Requests

An approach often used in CPUs is to introduce delays between synchronization requests. Delays could either be inserted after the lock has been released or after every request to acquire the lock. The duration of the delay can range from a simple constant to being increased by linear or exponential factors. In [5], Anderson explores these alternatives and concludes that exponential backoff provides the best performance and scalability for spin locks. Backoff algorithms have generally not been combined with queuing lock requests in the CPUs and are seen as orthogonal proposals. However in a GPU, we could implement backoffs for threads when the storage for queued locks requests is full. Extending the proposed instructions to implement Backoff and evaluating its performance is an area of future work.

9.2.2 Try Locks

As described in Chapter 7.1, the BarnesHut algorithm implements a form of *try lock*. In the implementation using atomics, only a single attempt is made and the program executes alternate code on failure. However, certain algorithm could require that the lock is tried for a certain duration of time or number of attempts before giving up. Scott et all have proposed and evaluated several version of *try locks* including solutions that timeout in lock queues [43, 44]. In order to adopt this work to GPUs, future work should investigate how to remove threads from the stack retry mechanism and ensure that they execute alternate code, if any.

9.2.3 Alternative to the Reconvergence Stack

As explained in Chapter 1.3.2, a reconvergence stack is used to handle control flow in a SIMT execution model. This means that during the execution of a program, not all threads in a GPU maybe active. These non-active threads can cause problems with synchronization if they are holding resources required by active threads as described in Chapter 4.5. These problems could be avoided if all thread in the GPU are allowed to make forward progress. Fung et al. propose a mechanism that eliminates the hardware stack by dynamically grouping threads into warps [14]. In this proposal the scheduler examines all threads in a thread block and creates a warp from threads executing the same instruction. This means a stack is not required and upon control flow divergence, threads in the warp are assigned to different warps based on the result of the branch. This mechanism allows all threads to make forward progress. This eliminates the problem of non-active threads holding resources and causing deadlocks. Adopting this mechanism to implement synchronizations and manage retries is an area of future work.

Bibliography

[1] Booksim interconnection network simulator. http://nocs.stanford.edu/booksim.html.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: an evolutionary processor design for large-scale multiprocessors. *Micro, IEEE*, 13(3):48 –61, jun 1993. ISSN 0272-1732. http://dx.doi.org/10.1109/40.216748doi:10.1109/40.216748.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. pages 487–499, 1994.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. *SIGARCH Comput. Archit. News*, 18: 1–6, June 1990. ISSN 0163-5964. http://dx.doi.org/http://doi.acm.org/10.1145/255129.255132doi:http://doi.acm.org/10.1145/255129.255132. URL http://doi.acm.org/10.1145/255129.255132.

[5] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, January 1990. ISSN 1045-9219. http://dx.doi.org/10.1109/71.80120doi:10.1109/71.80120. URL http://portal.acm.org/citation.cfm?id=628891.628973.

[6] M. August, G. Brost, C. Hsiung, and A. Schiffleger. Cray x-mp: the birth of a supercomputer. *Computer*, 22(1):45 –52, jan 1989. ISSN 0018-9162. http://dx.doi.org/10.1109/2.19822doi:10.1109/2.19822.

[7] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Int'l Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pages 163–174, April 2009.

[8] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, Dec. 1986. http://dx.doi.org/10.1038/324446a0doi:10.1038/324446a0. URL http://dx.doi.org/10.1038/324446a0.

[9] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[10] B. W. Coon et al. United States Patent #7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture (Assignee NVIDIA Corp.), April 2008.

[11] T. Craig. Building fifo and priority-queuing spin locks from atomic swap. Technical report, 1993.

[12] David Kanter. AMD Fusion Architecture and Llano. http://www.realworldtech.com/page.cfm?ArticleID=RWT062711124854p=2.

[13] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and Whay. The m-machine multicomputer, 1995.

[14] W. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. 40th IEEE/ACM Int'l Symp. on Microarchitecture*, 2007.

[15] J. R. Goodman, M. K. Vernon, and P. J. Wwst. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. pages 64–75, 1989.

[16] G. Graunke and S. Thakkar. *Synchronization algorithms for shared-memory multiprocessors*, pages 26–35. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995. ISBN 0-8186-6502-5. URL http://portal.acm.org/citation.cfm?id=201711.201712.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. http://dx.doi.org/http://doi.acm.org/10.1145/165123.165164doi:http://doi.acm.org/10.1145/165123.165164. URL http://doi.acm.org/10.1145/165123.165164.

[18] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee. Exploiting fine-grain thread level parallelism on the mit multi-alu processor. In *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, pages 306–317, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8491-7. http://dx.doi.org/http://dx.doi.org/10.1145/279358.279399doi:http://dx.doi.org/10.1145/279358.279399. URL http://dx.doi.org/10.1145/279358.279399.

[19] G. Kestor, S. Stipic, O. S. Unsal, A. Cristal, and M. Valero. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

[20] Khronos Group. OpenCL. http://www.khronos.org/opencl/.

[21] *The OpenCL Specification*. KHRONOS group, 1.1 edition, 2010.

[22] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, PODC '86, pages 218–228, New York, NY, USA, 1986. ACM. ISBN 0-89791-198-9. http://dx.doi.org/http://doi.acm.org/10.1145/10590.10609doi:http://doi.acm.org/10.1145/10590.10609. URL http://doi.acm.org/10.1145/10590.10609.

[23] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen. Atomic vector operations on chip multiprocessors. *Computer Architecture, International Symposium on*, 0: 441–452, 2008. ISSN 1063-6897. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/ISCA.2008.38doi:http://doi.ieeecomputersociety.org/10.1109/ISCA.2008.38.

[24] A. Levinthal and T. Porter. Chap - A SIMD Graphics Processor. In *11th Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 77–82, 1984.

[25] A. Levinthal and T. Porter. Chap - a simd graphics processor. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 77–82, New York, NY, USA, 1984. ACM. ISBN 0-89791-138-5. http://dx.doi.org/http://doi.acm.org/10.1145/800031.808581doi:http:

//doi.acm.org/10.1145/800031.808581. URL http://doi.acm.org/10.1145/800031.808581.

[26] E. Lindholm, M. J. Kligard, and H. P. Moreton. A user-programmable vertex engine. In *SIGGRAPH'01*, pages 149–158, 2001.

[27] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39–55, March-April 2008.

[28] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *In Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171. IEEE Computer Society, 1994.

[29] Martin Burtscher and Keshav Pingali. An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm. *Chapter 6 in GPU Computing Gems Emerald Edition*, pages 75–92, January 2011.

[30] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991. ISSN 0734-2071. http://dx.doi.org/http://doi.acm.org/10.1145/103727.103729doi:http://doi.acm.org/10.1145/103727.103729. URL http://doi.acm.org/10.1145/103727.103729.

[31] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: high-speed rendering using image composition. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 231–240, New York, NY, USA, 1992. ACM. ISBN 0-89791-479-1. http://dx.doi.org/http://doi.acm.org/10.1145/133994.134067doi:http://doi.acm.org/10.1145/133994.134067. URL http://doi.acm.org/10.1145/133994.134067.

[32] J. Montrym and H. Moreton. The geforce 6800. *Micro, IEEE*, 25(2):41 – 51, march-april 2005. ISSN 0272-1732. http://dx.doi.org/10.1109/MM.2005.37doi:10.1109/MM.2005.37.

[33] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[34] R. Narayanan, B. zs. Ikylmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *2006 IEEE*

International Symposium on Workload Characterization, pages 182–188, 2006.

[35] J. Nickolls et al. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar.-Apr. 2008.

[36] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. NVIDIA, October 2009.

[37] *NVIDIA CUDA Programming Guide*. NVIDIA Corporation, 3.1 edition, 2010.

[38] *NVIDIA Compute PTX: Parallel Thread Execution ISA Version 2.1*. NVIDIA Corporation, CUDA Toolkit 3.1 edition, 2010.

[39] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.

[40] J. Ributzka, Y. Hayashi, J. B. Manzano, and G. R. Gao. The elephant and the mice: the role of non-strict fine-grain synchronization for modern many-core architectures. In *ICS'11*, pages 338–347, 2011.

[41] B. Saha, A. reza Adl-tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *In Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, pages 187–197. ACM Press, 2006.

[42] Sarnath. Spinlock on a GPU. http://forums.nvidia.com/index.php?showtopic=98444&st=40/, June 2009.

[43] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *IN PROCEEDINGS OF THE 21TH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING*, pages 31–40. ACM, 2002.

[44] M. L. Scott and W. N. S. III. Scalable queue-based spin locks with timeout. In *PPOPP*, pages 44–52, 2001.

[45] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 54 –58, jan 1999. http://dx.doi.org/10.1109/HPCA.1999.744326doi: 10.1109/HPCA.1999.744326.

[46] B. A. Wallace. Merging and transformation of raster images for cartoon animation. In *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '81, pages 253–262, New York, NY, USA, 1981. ACM. ISBN 0-89791-045-1. http://dx.doi.org/http://doi.acm.org/10.1145/800224.806813doi:http://doi.acm.org/10.1145/800224.806813. URL http://doi.acm.org/10.1145/800224.806813.

[47] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Int'l Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pages 235–246, March 2010.

[48] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM. ISBN 0-89791-698-0. http://dx.doi.org/http://doi.acm.org/10.1145/223982.223990doi:http://doi.acm.org/10.1145/223982.223990. URL http://doi.acm.org/10.1145/223982.223990.

[49] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization.

[50] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 35–45, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. http://dx.doi.org/http://doi.acm.org/10.1145/1250662.1250668doi:http://doi.acm.org/10.1145/1250662.1250668. URL http://doi.acm.org/10.1145/1250662.1250668.