Fine-Grained Synchronizations and Dataflow Programming on GPUs

Ang Li, Gert-Jan van den Braak, Henk Corporaal
Eindhoven University of Technology
Eindhoven, Netherlands
ang.li@tue.nl, g.j.w.v.d.braak@tue.nl, h.corporaal@tue.nl

Akash Kumar
National University of Singapore
Singapore
akash@nus.edu.sg

ABSTRACT

The last decade has witnessed the blooming emergence of many-core platforms, especially the graphic processing units (GPUs). With the exponential growth of cores in GPUs, utilizing them efficiently becomes a challenge. The data-parallel programming model assumes a single instruction stream for multiple concurrent threads (SIMT); therefore little support is offered to enforce thread ordering and fine-grained synchronizations. This becomes an obstacle when migrating algorithms which exploit fine-grained parallelism, to GPUs, such as the dataflow algorithms.

In this paper, we propose a novel approach for fine-grained inter-thread synchronizations on the shared memory of modern GPUs. We demonstrate its performance and compare it with other fine-grained and medium-grained synchronization approaches. Our method achieves 1.5x speedup over the warp-barrier based approach and 4.0x speedup over the atomic spin-lock based approach on average. To further explore the possibility of realizing fine-grained dataflow algorithms on GPUs, we apply the proposed synchronization scheme to *Needleman-Wunsch* – a 2D wavefront application involving massive cross-loop data dependencies. Our implementation achieves 3.56x speedup over the atomic spin-lock implementation and 1.15x speedup over the conventional data-parallel implementation for a basic sub-grid, which implies that the fine-grained, lock-based programming pattern could be an alternative choice for designing general-purpose GPU applications (GPGPU).

Categories and Subject Descriptors

C.1.2 [**Computer System Organization**]: Multiple Data Stream Architectures—*SIMD*; D.1.3 [**Software**]: Concurrent Programming—*Parallel programming*

Keywords

Fine-grained synchronization; Spin-lock; GPU; Dataflow

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS'15, June 08 - 11, 2015, Newport Beach, CA, USA.
Copyright © 2015 ACM 978-1-4503-3559-1/15/06 ...$15.00.
http://dx.doi.org/10.1145/2751205.2751232.

1. INTRODUCTION

To harness the unprecedented computational capacity of modern multiprocessor architectures, a program must be partitioned and executed by multiple threads that communicate via shared memory or interconnection network. To ensure correctness, however, operations from various threads must obey certain order restrictions imposed by program logic. Synchronization is the process referring to this coordination issue, during which timing information is exchanged among participant threads.

Synchronization can be further classified as *thread cooperation* and *thread contention* [1]. Thread cooperation enforces read-after-write data dependencies between cooperative threads, which is accomplished by producer-consumer primitives in general. Thread contention, on the other hand, ensures exclusive manipulation of the shared data so that program consistency is preserved. Atomic operations are provided for this purpose. The major difference between the two classifications is that thread cooperation emphasizes access order while thread contention stresses mutual exclusion. In this paper, unless stated otherwise, the word *synchronization* is specially referred to thread cooperation.

Synchronization is not free. It can consume a significant fraction of the execution time due to parallelism degradation, as threads may stall at barriers or spin at locks [2, 3]. Furthermore, the synchronization process itself induces overhead, such as the communication delay and memory traffic for enquiring and releasing locks, the operation overhead for updating mutexes, the storage cost for synchronization variables, etc.

Such overhead is particularly significant for algorithms that exploits *fine-grained parallelism* (e.g. dataflow algorithms) as the occurrence of synchronizations in these algorithms is much more frequent than in other applications [4]. As a result, numerous works have been proposed to alleviate the *fine-grained synchronization* overhead, from both architectural [5, 6, 7] and algorithmic perspectives [8, 9, 10].

Starting from the last decade, the graphics processing unit (GPU) has evolved to be applied on general purpose applications [11, 12]. However, traditional data-parallel programming models for GPUs assume single instruction stream for all concurrent threads (SIMT) and little support is offered to enable elaborate thread cooperation. This becomes an obstacle when migrating dataflow applications which exploits fine-grained parallelism to GPUs.

GPU threads are organized in a hierarchy of three levels: *thread*, *warp* and *block*. Accordingly, three different granularities are addressed for GPU synchronizations:

- *coarse-grained:* synchronizations among thread blocks.
- *medium-grained:* synchronizations among warps in thread blocks.
- *fine-grained:* synchronizations among threads in thread blocks.

GPU currently provides hardware support for medium-grained warp barriers [13]. It also offers fine-grained atomic operations on global and shared memory [14]. However, the existing atomic operation based synchronization scheme, as will be seen, exhibits poor performance; using it incurs significant overhead.

In this paper, we propose a fine-grained, highly efficient thread synchronization mechanism on the shared memory of NVIDIA Fermi GPUs [14]. Instead of seeking to reduce the occurrence of synchronizations, we look into an atomic instruction itself from a lower level standpoint. By reassembling the *micro-instructions* that comprise an atomic operation, we develop an approach that can set up a *producer-consumer* communication channel between cooperative threads in a thread block with much less overhead than the atomic spin-lock based implementation. We validate the correctness and demonstrate the effectiveness of the proposed approach through comparisons with other fine-grained and medium-grained synchronization approaches. Further, to explore the possibility of realizing thread-level dataflow algorithms on GPUs, we apply the proposed synchronization scheme in *Needleman-Wunsch* – a 2D wavefront application that contains a large amount of cross-loop data dependencies. The performance we obtained proves that the fine-grained, lock-based programming pattern could be an alternative choice for designing GPGPU applications.

This paper makes the following contributions:

- We show the inefficiency of the atomic spin-locks and propose a novel lock mechanism (called **tiny-lock**) that shows much better performance with no memory cost.
- We use the tiny-lock to build highly efficient producer-consumer primitives for fine-grained data synchronizations between cooperative threads in a thread block.
- We address two architectural factors that can lead to deadlocks: one is the structural conflicts between thread ordering and SIMD execution; one is lock alias.
- We show how to realize lock-based dataflow computing on GPUs using a wavefront application. This is the first time, to the best of our knowledge, that a fine-grained dataflow model has been reported to be efficiently implemented at the lowest *thread level* of GPUs.

2. RELATED WORK

For **coarse-grained** synchronizations on GPUs, Xiao et al. proposed three schemes [15]: a simple version, a tree-based version, and a lock-free version. The simple version leveraged a global-shared mutex via global memory atomic operations. The tree-based version improved the simple version by synchronizing progressively along the tree branches. The lock-free version allocated a monitor thread block to coordinate synchronizations among working thread blocks. Their work was later extended by Stuart et al. to build a set of course-grained synchronization primitives [16].

In terms of **medium-grained** synchronizations, although the block-wise barrier *__syncthreads()* is widely adopted, it was not until recently that a warp-to-warp synchronization approach has been developed. It relies on the *sync-arrive* barrier pair [17]: *bar.sync* is a blocking operation that suspends the current warp until all desired warps have arrived at the barrier. *bar.arrive* is a non-blocking operation that signals the arrival of the current warp to the barrier. In [18], Bauer et al. proposed a producer-consumer communication model based on this barrier-pair that could coordinate data movement from a producer warp to a consumer warp via shared memory buffers. They further applied this medium-grained synchronization approach to a chemical application [19]. The performance was demonstrated and the implementation was straightforward using the PTX embedding technique. However, for this approach, although the number of synchronization threads is parameterizable, it has to be a multiple of warp size [17] (32 for all present CUDA GPUs), meaning that the granularity is warp, not thread. Furthermore, only 16 barrier instances are available per thread block [17], making these barriers very precious and limited for frequent usage, such as in a context of dataflow programming.

Regarding **fine-grained** synchronizations, the only approach till now, to the best of our knowledge, is through the spin-locks, which are constructed using the atomic operations in global memory and shared memory. However, the performance of such atomic spin-locks is poor and their utilization is highly discouraged [20]. In fact, the lack of highly efficient, fine-grained synchronization mechanisms has already become an obstacle that disturbs the broad adoption of GPUs for general purpose applications [16, 21, 22].

3. LOCK UNIT

In this section, we briefly describe the architecture of the lock unit in GPU shared memory and the associated operations.

3.1 Architecture

The *shared memory* (i.e. *scratchpad memory*) in a GPU is a small on-chip storage shared among all processing units in a streaming multiprocessor (SM). It serves as a communication interface for fast data exchanging between different threads of a thread block. Being on-chip, the shared memory has much higher bandwidth and shorter accessing latency compared to the *global memory* (or *main memory*) of GPUs. Therefore, optimizations which can shift global memory access to shared memory access are highly advised by the CUDA programming guide [13].

The lock mechanism that enables fast atomic access is implemented in the shared memory, under the help of a module called "*lock unit*", in Fermi GPUs (see Figure 1). According to the associated patent [23], the lock bits are flags indicating the present lock status for the corresponding locations in the main storage (i.e. the Storage Resource in Figure 1). The lock bit is set so that other updating requests to that location are refused. For space concern, multiple locations in the main storage are *aliased* to a single lock bit. A hash function is implemented to perform the mapping, ensuring that successive *words* are mapped to distinct lock bits. For Fermi GPUs, a total of 1024 independent lock bits are provided for the 16KB (or 48KB based on configuration) shared

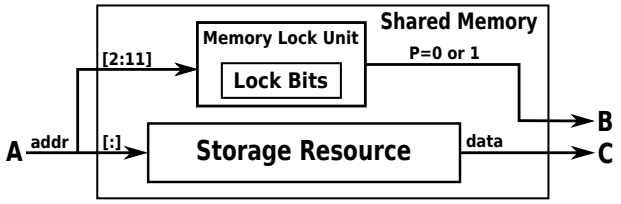

Figure 1: Shared memory lock unit. Terminal A reads the request memory address and looks it up in the storage resource. The fetched data is returned to terminal C that connects to a general register. Meanwhile, the 2-to-11 bits of the data address is used to retrieve associated lock bit from the lock unit. The value of lock bit is returned to terminal C, which connects to a predicate register.

memory. Word addresses with a stride of 1024 are aliased to the same lock bit. When a memory request being delivered (to terminal A in Figure 1), the 2-to-11 bits of the data address is labeled as the lock address and redirected to the lock unit. Gomez-Luna et al. discusses this mapping mechanism exhaustively in [24] and report a number of 1024 lock bits. We confirm this value experimentally when testing deadlocks (see Section 4.4).

Regarding such a design, the following characteristics are highlighted for our proposal:

- **Efficiency:** Accessing the lock units does not require extra pipeline-stages or decision logic in the critical path because it is performed in parallel with the ordinary data access. So no extra delay is induced.
- **Flexibility:** The lock unit is not configured to track the ownership of the locks. It is the program's responsibility to honor the lock bits and to prevent illegal access to the locked locations in the main storage.

3.2 Operations

Listing 1 shows the low-level assembly sequence (SASS) generated by *cuobjdump* for the atomic instruction "*atomicAdd()*" to the shared memory of Fermi GPUs, in CUDA runtime (i.e. *atom.shared.add* instruction in PTX [17]). It indicates that the high-level *"atomic"* instruction is essentially comprised by a series of low-level SASS operations:

```
/*00a0*/  LDSLK P0, R1, [R0];
/*00a8*/  @P0 IADD R4, R1, 0x1;
/*00b0*/  @P0 STSUL [R0], R4;
/*00b8*/  @!P0 BRA 0xa0;
```

Listing 1: SASS code for atomicAdd()

- **LDSLK** loads data from address [R0] to a general register R1. It also reads the associated lock bit to a 1-bit predicate register P0. (In Figure 1, R0 is connected to A, R1 is connected to C, P0 is connected to B.) Therefore, P0 equals true implies that the target lock is successfully acquired by the current thread. Meanwhile, the lock bit in the lock unit toggles to 0, disabling subsequent locking requests. Here, "LDS" stands for loading from shared memory while "LK" means loading the lock bit simultaneously.
- Based on P0=1 (@P0), **IADD** adds 0x1 to R1 and stores the sum to R4. Note that threads in a warp may diverge here if some of them fail to acquire the locks in the present locking test (@!p0).
- Also with P0=1, **STSUL** stores R4 to [R0] and triggers the lock unit to reset the lock bit. "STS" stands for storing to shared memory while "UL" means unlocking simultaneously.
- **BRA** is the branch operation that jumps to instruction address 0xa0, which is the entry of the atomic procedure. In this way, the threads failed to obtain locks in the current test rotate back and redo the atomic process. Meanwhile, the finished threads have to wait beyond this BRA operation until all divergent threads in the warp have reached so as to continue lockstep execution.

Regarding these operations, it should be noted that:

- The default value of a lock bit is 1, indicating that it is free for fetching. LDSLK resets the lock bit to 0 while STSUL sets the lock bit to 1. It is infeasible to set the lock bit via LDSLK or reset the lock bit via STSUL. There is no alternative way to set or reset a lock bit.
- To release a lock bit, a thread **must** store a value to the corresponding memory location simultaneously. The store overwrites the original content.

4. FINE-GRAINED SYNCHRONIZATION

In this section, we present the fine-grained synchronization mechanism. We first describe our motivation and then propose the *tiny-lock*, based on which we show our fine-grained synchronization scheme.

4.1 Motivation

Our approach is motivated by the observation that an atomic instruction in the shared memory is comprised of multiple low-level SASS operations (Section III-B). Therefore, we can *reassemble these SASS operations in a different way to build other efficient synchronization procedures.*

4.2 Tiny-Lock

Fine-grained synchronization relies on fine-grained locks. Listing 2 illustrates a common implementation [25] of the fine-grained spin-locks based on atomic instructions.

```
__device__ inline void lock(int* p_mutex){
  while(atomicCAS(p_mutex,0,1)!=0);//compare and swap
}
__device__ inline void unlock(int* p_mutex){
  atomicExch(p_mutex,0);//exchange
}
```

Listing 2: Baseline implementation: atomic spin-locks [25]

In Listing 3, we show the SASS sequence of the baseline implementation. To make it more clear, we draw the corresponding control-flow-graph (CFG) in Figure 2. There are two loops in the *Lock* routine: the small loop is spinning for a lock bit. It is embedded in *atomicCAS()*. The big loop, which corresponds to the *while* statement, is the actual iteration for the user-defined mutex variable stored in the main storage of the shared memory.

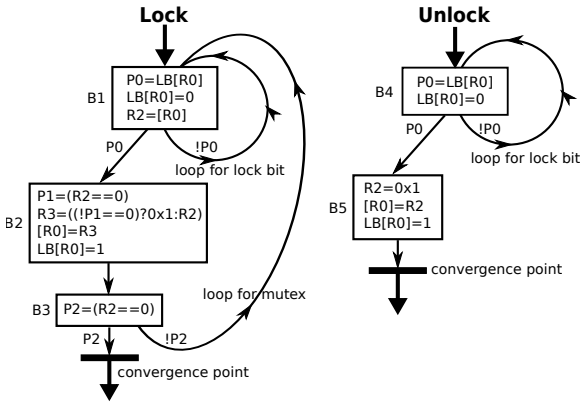

Figure 2: CFG of atomic spin-locks. LB stands for lock unit. Convergence point is the place where divergent threads of a warp rejoin to proceed lock-step execution. In the *Lock* routine, the big loop is for acquiring the user-defined mutex while the small loop is for acquiring the lock bit of the mutex. P2 being a replicate of P1 is the result of direct translation from two ptx instructions by the ptxas assembler. In the *Unlock* routine, the atomic update to the mutex (i.e. the small loop) is a must; otherwise, the updated result may be overwritten unexpectedly by another thread who acquires the lock bit but not the mutex. Since that thread needs to write a value to the mutex for releasing the lock bit, it uses a dated value obtained when fetching the lock bit as it is unaware of the latest update.

```
// ==================== Lock ====================
/*0060*/  SSY 0x98; //set convergence point
/*0068*/  LDSLK P0, R2, [R0];
/*0070*/  @P0 ISETP.EQ.U32.AND P1, pt, R2, RZ, pt;
/*0078*/  @P0 SEL R3, R2, 0x1, !P1;
/*0080*/  @P0 STSUL [R0], R3;
/*0088*/  @!P0 BRA 0x68; //atomicCAS loop
/*0090*/  ISETP.EQ.AND.S P2, pt, R2, RZ, pt;
/*0098*/  @!P2 BRA 0x60; //while loop
          //converge to proceed lockstep execution
/*00a0*/  ...
// ================== Unlock ==================
/*00b0*/  LDSLK P0, RZ, [R0];
/*00b8*/  @P0 MOV32I R2, 0x1;
/*00c0*/  @P0 STSUL [R0], R2;
/*00c8*/  @!P0 BRA 0xb0;
```

Listing 3: SASS code of atomic spin-locks

This is a recursive design: the user-defined mutex acts as an intermediate layer to realize the required locking functionality (i.e. the big loop) whereas the lock bit of the mutex is leveraged to ensure atomic updates to the mutex (i.e. small loop). Such a design behaves quite well when the mutex serves as a semaphore, but is probably redundant when only a single-bit lock is required – why not exploit the lock bit directly?

```
// ==================== Lock ====================
/*0000*/  LDSLK P0, RZ, [R0];
// ================== Waitlock ==================
/*0010*/  LDSLK P0, RZ, [R0];
/*0018*/  @!P0 BRA 0x10;
// ================== Unlock ==================
/*0020*/  STSUL [R0], RZ;
```

Listing 4: Proposed fine-grained lock

We show the novel design in Listing 4. It is called **tiny-lock**. There are two primitives for locking: *Lock* simply fetches without verifying the locking result. It is used when the programmer guarantees the acquisition of the target locks (e.g. in an initialization scenario). Otherwise, *Waitlock* has to be applied, which repeatedly fetches the lock until eventually succeeds. *Unlock* stores 1 to the lock bit for release.

Such a design completely eliminates the space cost for the user-defined mutex. It also avoids the big loop in the *Lock* routine and the small loop in the *Unlock* routine. Compared to the baseline implementation, it has the following advantages:

- **Time Delay**, the proposed design reduces the static number of SASS operations by 75% for *Lock* and *Unlock*; and by 50% for one iteration of *Waitlock* (although the dynamic number of operations executed by waitlock depends on the waiting time experienced). Meanwhile, the lock unit is accessed in parallel with the shared storage (*Efficiency* in Section 3.1), so the maximum delay for accessing locks is equal to an ordinary memory read or write. Furthermore, this delay can be completely hidden in certain scenarios, e.g. the read-after-write data synchronizations.

- **Storage Cost**, since the lock unit is isolated from the main storage, our scheme does not require any shared memory storage. In comparison, the baseline implementation has to explicitly allocate a word as an intermediate mutex. Further, since only the lock bit is of interest, in many cases (see Section 5 and Section 6) we can read the content of the memory location to the zero register in *Lock* or write the original value back in *Unlock* so that no register is used either.

- **Memory Traffic**, there is only one load transaction for *Lock* and one store transaction for *Unlock*. For *Waitlock*, unlike the baseline implementation that writes the original value back to the mutex if the lock is not obtained (B2 in Figure 2 when R2!=0), our approach does not produce any write traffic when locking. Further, it does not produce computation traffic like the baseline implementation (e.g. operations 0x0070, 0x0078 and 0x00b8 in Listing 3).

4.3 Fine-Grained Synchronization

All concurrent programming models offer programmers the ability to control the order of dataflow from different threads. However, conventional SIMT programming model assumes weak inter-dependencies among threads that relies on barriers to enforce thread ordering. However, barriers are either coarse-grained or medium-grained in GPUs, which are too coarse for thread-to-thread synchronizations. Therefore, fine-grained locks have to be used for such synchronizations.

```
void producer(){
  lock(&mutex);   //initialize
  ...
  shared_buffer = put; //store to channel
  unlock(&mutex); //signal consumer to load
}
void consumer(){
  ...
  lock(&mutex);   //wait producer to store
  get = shared_buffer; //load from channel
  unlock(&mutex); //finalize
}
```

Listing 5: Fine-grained synchronizations based on atomic spin-locks

Listing 5 illustrates how an atomic spin-lock is used for read-after-write synchronizations – *the producer thread acquires the mutex in advance and releases it after writing to*

the shared buffer so that when consumer thread obtains the mutex, it can read safely.

Here, a 1-bit lock is already sufficient to accomplish the job. However, as discussed earlier and will be seen in the experiments, the atomic spin-lock incurs significant time/space/traffic overhead which makes it too costly for frequent inter-thread synchronizations. The proposed tiny-lock design significantly reduces such overheads and is therefore the ideal option upon which to construct the fine-grained synchronization scheme. Its implementation is shown in Listing 6.

```
// =============== producer ===============
/*0000*/  LDSLK P0, RZ, [R0];//initialize
...
/*0010*/  STSUL [R0], R4;//store to channel and
                         //unlock
// =============== consumer ===============
...
/*0100*/  LDSLK P0, R2, [R0];//wait and load from
                             //channel
/*0108*/  @!P0 BRA 0x100;//spinning
/*0200*/  STSUL [R0], R2;//finalize
```

Listing 6: Fine-grained synchronizations based on lock bits

This is the one-to-one synchronization scheme, which can be extended further to one-to-many and many-to-one conditions: the producer alternatively signals all its consumers or the consumer waits for all its producers.

4.4 Deadlock

Programmers must be careful when using fine-grained locks in GPUs because it is easy to generate deadlocks. Besides general causes from algorithmic aspects, there are two special scenarios that may lead to deadlocks for GPUs. We label them *SIMD Deadlock* and *Alias Deadlock*.

4.4.1 SIMD Deadlock

This kind of deadlock is due to a structural conflict between inter-thread synchronizations and SIMD-lockstep execution. Consider the following scenario: what if the producer and consumer threads are from the same warp? The answer is — a *deadlock*. The general explanation is that lockstep stresses synchronous execution whereas thread cooperation enforces consumer-after-producer (i.e. read-after-write) order, which is essentially asynchronous. Therefore, if the synchronizing threads are from the same warp, we need a divergence mechanism to separate the producer and the consumer's execution paths. In addition, for the producer, the lock and unlock operations must be within the same divergent segment, or in other words, the unlock operation must be the post-dominator for the lock operation before the next convergence point. Otherwise, the producer will wait at that convergence point for the consumer to join in order to proceed to execute the unlock instruction, whereas the consumer is waiting to acquire the lock before it can step to the convergence point. Here the inter-waiting produces a deadlock.

In fact, such deadlocks occur more often than just for synchronizations. Consider a warp executing lock function in Listing 2. The convergence point is well beyond the while loop (see the black barrier in Figure 2). If two or more threads in the warp are contending for the same mutex (not lock bit), due to atomicity, only one of them can acquire. However, this thread has to be blocked at convergence point, waiting for other threads to join. Meanwhile, the remaining threads are adversely waiting for that thread to release the mutex (via calling the unlock function) before they can proceed. Here, the same reason leads to a deadlock: *the SIMD convergence point is earlier than unlock*. To circumvent this problem, a direct implementation for the baseline scheme is shown in Listing 7. In this way, the release of the mutex (i.e. *atomicExch(p_mutex,0)*) can be performed before the warp convergence point, which is right after the while loop.

```
__device__ void producer_consumer(int* p_mutex){
  bool finished = false;
  while(!finished){
    if(atomicCAS(p_mutex,0,1)==0){
      finished=true;
      ... // critical section
      atomicExch(p_mutex,0);
} } }
```

Listing 7: Intra-warp synchronizations based on atomic spin-locks

And for our scheme in Listing 6, the predicate register can be manipulated to include unlock operation into the same divergent path, as shown in Listing 8.

```
// producer-consumer
/*00a0*/  SSY 0x110;
/*00a8*/  LDSLK P0, R2, [R0];
...       @P0 ... //critical section
/*0100*/  @P0 STSUL [R0], R2;
/*0108*/  @!P0 BRA 0xa8;
```

Listing 8: Intra-warp synchronizations based on lock bits

Although we successfully circumvent this deadlock at programming level, another problem still remains – performance degradation. As GPU adopts lane-masks to switch between divergent branches for a warp, the performance is impaired when each divergent branch has to be executed sequentially. Here, the producer lane has to wait until the consumer lane finishes. Even worse, if the consumer is in turn a producer of another synchronization also in the same warp, such as in a "scan" operation, then all the former producers have to be blocked until the final consumer finishes the synchronization. In the worst case, the performance drops by 32 folds (e.g. a propagation chain). Unless a perfect pipeline can be formed (i.e. producers start working on new data but execute in a lockstep with the consumers), some threads will be idle. The problem here is that the dispatch units only issue warp instructions, which is too coarse-grained for elaborate intra-warp coordination.

Summarizing, for synchronizations between threads of different warps, we use the lock/unlock primitives in Listing 5 and Listing 6. Both the producer and consumer can proceed immediately after the synchronization. But for synchronizations involving threads from the same warp, the critical sections in Listing 7 and Listing 8 are necessary. The producers have to wait until all their direct or indirect consumers accomplish their synchronizations and arrive at the convergence point. Although performance suffers, the fine-grained scheme is still better than a medium-grained approach as consumers from other warps can be signaled as soon as the required data is produced, instead of waiting for the whole warp that contains the consumer to be finished.

4.4.2 Alias Deadlock

This kind of deadlock is due to lock bit aliasing. There are two conditions: First, suppose a thread already holds the

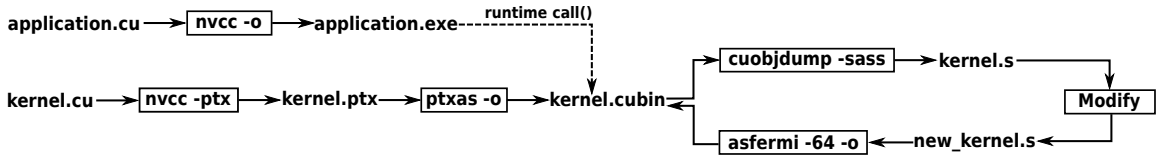

Figure 3: Experiment workflow. The application is written in CUDA driver-API that can load cubin object file at runtime. The kernel is first developed in CUDA-C and compiled to PTX code via *NVCC*. The PTX file is then assembled to a cubin binary via *ptxas* which is marked as the base binary. After that, the human-readable SASS routine is dumped from the base binary through *cuobjdump*. We modify this routine manually to insert the producer-consumer instructions, which is re-assembled to an updated cubin file for the driver-API to load.

lock bit of a memory location, say [M], but is trying to fetch from its aliased location (e.g. [M+1024], see Section 3.1). Then, the thread will trap in a circle because it is attempting to get a lock bit from itself. Based on our experiments, such a conduct immediately leads to a deadlock. However, the positive side is that such an experiment confirms the stride of lock bit alias is 1024 [24].

Second, we need to ensure the producer acquires the lock before its consumer (see Listing 5 and Listing 6). As warps are not synchronously executed in an SM, this is achieved by placing a coarse-grained barrier (i.e. *__syncthreads()*) after the initialization phase for the whole thread block. Lock-bit aliasing generates deadlock because the warp obtaining the aliased lock waits at the block-wise barrier for other warps, including the failed warp, while the failed warp is waiting for the aliased lock before it can reach the barrier.

Although alias deadlock is easy to understand, it is one of the major restrictions for the proposed synchronization scheme: *to avoid alias deadlock, only 1024 locks can be utilized safely.* This number is smaller than the allocatable threads for an SM (i.e. 1536 threads) and much smaller than the entries of the shared memory (i.e. 4096 or 12,288). Given the fact that an SM can accommodate several thread blocks, the volume of usable lock bits can significantly limit the number of thread blocks an SM could support, hence degrading the performance for a large data size (see Section 6).

4.5 Warp-shared Lock Bit

When fine-grained lock bits are exploited for the situations of medium-grained synchronizations, it is possible to share a single lock bit for the whole warp, which reduces the demand for lock bits by a factor of 32. The idea is to exploit the warp-wise voting instructions [17]. Listing 9 provides the implementations for the lock and unlock routines, based on which the readers can further construct warp synchronization primitives.

```
// ==================== Lock ====================
//R0 is the same for all threads across the warp
/*0000*/  LDSLK P0, RZ, [R0];
// ================== Waitlock ==================
//If any thread acquires the lock bit, continue
/*0010*/  LDSLK P0, RZ, [R0];
/*0018*/  VOTE.ANY RZ, P1, P0;
/*0020*/  @!P1 BRA 0x10;
// ================== Unlock ==================
//Thread 0 in the warp releases the shared lock
/*0020*/  S2R R1, SR_LaneId; //Load lane_id
/*0028*/  ISETP.EQ P0, pt, R1, RZ, pt;//lane_id=0?
/*0030*/  @P0 STSUL [R0], RZ;
```

Listing 9: Warp-shared lock bit scheme

For *Lock*, any thread in the warp may acquires the lock eventually, but we know one of them must obtain it. For *Waitlock*, after acquiring, all threads are enforced to participate in a warp-wise vote. If any thread successfully acquires the target lock (i.e. P0=1), the voting result is true (i.e. P1=1). Then the whole warp quits the spinning loop and proceeds lockstep execution. Otherwise, the warp rotates back and try again. For *Unlock*, it may be too expensive to let every thread perform the release operation since a 32-degree bank conflict and lock conflict can be generated [24]. Further, if there are multiple threads waiting for the lock, releasing it 32 times (due to conflict) may potentially violate the consistency between the waiting threads. The method here is to find a representative. Here the ISETP instruction and predicate register P0 are used to select thread 0 for releasing. Note it is not feasible to let the representative thread acquire the lock for the whole warp because the remaining 31 threads may fail to make their writings observable by other warps due to the weekly-ordered memory model [13]. However, such a design is not a problem if an atomic-spin lock is shared for the whole warp as it enforces the order in the memory.

5. VALIDATION

In this section, we validate the correctness and demonstrate the effectiveness of our fine-grained synchronization scheme. We use a NVIDIA GTX-570 GPU as the test platform. It contains 15(SM)x32 CUDA cores with compute capacity 2.0 (Fermi). The CUDA toolkit version is 4.0. In terms of tools, *cuobjdump* is employed to generate the SASS code of the target kernel. We then modify the SASS code to insert our lock operations. However, to reproduce the *cubin* binary for the updated SASS code, an SASS assembler is necessary. Since *ptxas* only accepts PTX code, we use an open-source SASS assembly tool named *asfermi* [26] instead. This is also the reason why we restrict to Fermi – *asfermi does not support other architectures right now.* The detailed workflow is depicted in Figure 3.

```
for (i=0;i<32*N;i++)
  A[i+32] = A[i] + independent_computation(i);
```

Listing 10: Validation kernel (serial version)

The loop shown in Listing 10 is used for validation. It contains a parallel independent computation phase and a serial dependent reduction phase. It is derived from the kernel developed by Tullsen et. al. [6] that represents a common map-reduce pattern. In order to compare with the medium-grained synchronization approaches (see Section 2), we extend the dependency distance from 1 to the size of a warp (i.e. 32). Meanwhile, since only 16 warp barriers are available in a thread block (see Section 2), N is set to be 16.

The whole loop is parallelized and mapped to 16 warps for concurrent execution. We compare the proposed tiny-lock implementation (i.e. *tiny_lock*, Section 4.3) with the atomic

Scheme	Shared Memory Cost	Lock Bit Used
atom_lock	2048 bytes	512 (implicit)
warp_barr	0	0
shrd_lock	128 bytes	32 (implicit)
tiny_lock	0	512 (explicit)
warp_vote	0	32 (explicit)

Table 1: Resource cost

spin-lock implementation (i.e. *atom_lock*, Section 4.2), the medium-grained sync-arrive barrier implementation (i.e. *warp_barr*, Section 2), the shared lock-bit implementation (i.e. *warp_vote*, Section 4.5) as well as a shared spin-lock implementation (a warp shares a common spin-lock, i.e. *shrd_lock*). The core of the kernels for atomic spin-lock based, sync-arrive barrier based and tiny-lock based implementations are shown in Listing 11, Listing 12 and Listing 13 respectively.

```
__shared__ int A[32*N], mutex[32*N];
  lock(mutex[tid]);//producer initially locks
  __syncthreads();//ensure producer gets lock first
  /* ============== Reduction Phase ==============*/
  if (wid > 0) lock(mutex[tid-32]); //consumer waits
  A[tid]=A[tid-32]+independent_computation(tid);
  unlock(mutex[tid]);    //producer releases
  unlock(mutex[tid-32]); //finalize
```

Listing 11: Atomic spin-lock based version (CUDA code)

```
__shared__ int A[N*32];
  int tid = threadIdx.x; int wid = tid>>5; //log32=5
  /* ============== Reduction Phase ==============*/
  if (wid>0)
    asm("bar.sync %0,%1;"::"r"(wid-1),"r"(64));
  A[tid+32]=A[tid]+independent_computation(tid);
  asm("bar.arrive %0,%1;"::"r"(wid),"r"(64));
```

Listing 12: Warp barrier based version (PTX embedded CUDA code)

```
/*0000*/ LDSLK P0,RZ,(A[tid]);//producer init locks
/*0008*/ BAR.RED.POPC RZ,RZ;  //block barrier
/* ============== Reduction Phase ==============*/
/*0100*/ ISETP.EQ P0, pt, (wid), RZ, pt;
/*0108*/ @P0 BRA 0x120; // warp_0 breaks
/*0110*/ LDSLK P1,R1,(A[tid-32]); //consumer waits
/*0118*/ !@P1 BRA 0x110;
/*0120*/ IADD R2,R1,(independent_computation(tid));
/*0128*/ STSUL (A[tid]),R2; //producer releases
/*0130*/ @!P0 STSUL (A[tid-32]),R1; //finalize
```

Listing 13: Tiny-lock based version (SASS code)

To be simple, we set *independent_computation()* to immediately return its thread index. Therefore, if we measure the elapsed time for the reduction phase, it is the *raw delay for 16 times' synchronizations and additions in sequence*. Figure 4 illustrates the measured execution time in cycles for the reduction phase for the 5 schemes. Table 1 lists the cost of resource for each scheme.

As can be seen, our tiny-lock based approach is 4.0x times faster than the atomic spin-lock based scheme and is 1.5x times faster than the warp barrier scheme. Meanwhile, warp voting is shown to be an expensive operation (it actually induces thread divergence in a warp) although the sharing saves many lock bits. Finally, picking a warp-representative thread reduces space cost at the expense of performance loss. Table 2 summarizes the 5 schemes.

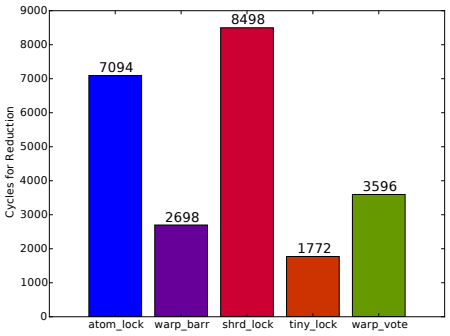

Figure 4: Execution time for the reduction phase in cycles

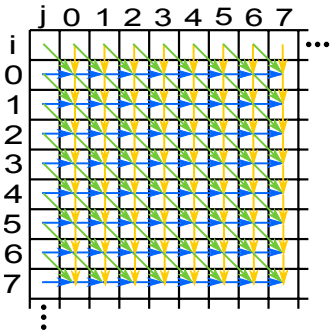

Figure 5: Dependence graph for the *Needleman-Wunsch* algorithm. The green arrows denote dependencies with the north-west neighbors. The yellow arrows refer to dependencies with north elements. The blue arrows indicate dependencies with the west grid-points. The first row and column of the grid are the initial values.

6. WAVEFRONT APPLICATION

In this section, we use the *Needleman-Wunsch* algorithm [27, 28] from the *Rodinia* benchmark [29] as an example to describe how to efficiently implement a dataflow algorithm on GPUs using the proposed fine-grained, tiny-lock based synchronization schemes. The application is to find the best alignment between protein or nucleotide sequences in bioinformatics. Its core computation is:

$$S(i,j) = max \begin{cases} S(i,j-1) - k \\ S(i-1,j-1) + p(i,j) \\ S(i-1,j) - k \end{cases} \quad (1)$$

where S is 2D grid and $p(i,j)$ is a predefined reference field. As can be seen, the computation of each grid-point has true data dependencies on its north, west and north-west neighbors. The dependence graph is shown in Figure 5.

The data-parallel model relies on wavefront propagation to resolve such dependencies. In [30], Lamport et al. show that, for a multi-dimensional volume, given a value f, all points laid in the hyperplane satisfying $i + j + ... = f$ can be processed in parallel while all their dependent points fulfill $i + j + ... = f - 1$. By stepping along the incremental direction of f and processing all elements associated, data dependencies can be respected. So far, all the existing implementations of wavefront applications on GPUs adopt this data-parallel pattern [22, 31, 32]. Figure 6 illustrates the processing trace of this pattern for the *Needleman-Wunsch* algorithm.

Scheme	Granularity	Performance	Memory Cost	Resource	Programmability
atom_lock	fine	x1.0	128 bytes/warp	4096/12,288 locations per SM	CUDA runtime
warp_barr	medium	x2.6	0	16 barriers per thread_block	PTX/embedded_PTX
shrd_lock	medium	x0.8	4 bytes/warp	4096/12,288 locations per SM	CUDA runtime
tiny_lock	fine	x4.0	0	1024 lock bits per SM	assembly
warp_vote	medium	x2.0	4 bytes/warp	1024 lock bits per SM	assembly

Table 2: Summary of synchronization schemes

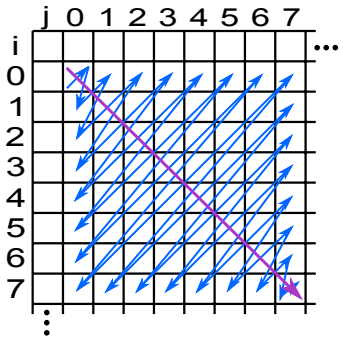

Figure 6: Working trace for wavefront parallel pattern. The wavefront direction coincides with the diagonal of the grid. In each wavefront step, the points along the anti-diagonal can be processed in parallel.

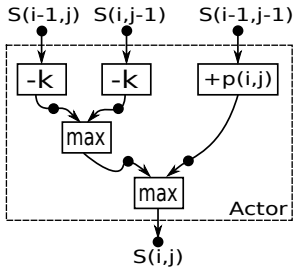

Figure 7: Dataflow graph. The actor is the computation shown in Equation.1. When the required operands $S(i-1,j)$, $S(i,j-1)$ and $S(i-1,j-1)$ are ready, the actor can fire. The arcs across the dashed box denote the dependencies with other actors, which are also the places that require synchronization.

However, the data-parallel propagation approach confronts two problems: first, as the points that can be processed in parallel are along the line that is perpendicular to the diagonal, the computation workload for each propagation step is quite unbalanced, especially for SIMD processing. Second, since the grid-points are normally sequentially stored along the axises of the gird in memory, data access in each step is cache unfriendly and cannot be coalesced for effective global memory fetch.

The major factor leading to the irregular computation and memory access is the rigorous 2D data-dependencies, which can be naturally and effectively resolved by a static dataflow model. A dataflow model describes the computation of each point as an *actor* which is executed by a GPU thread. The actor *fires* when all the operands it requires are available. Many actors may fire simultaneously, thus achieving high-level asynchronous concurrency. The dataflow graph for the application is shown in Figure 7. Since the computation of an actor is relatively simple, we concentrate on the communication part: *how to effectively synchronize between actors.*

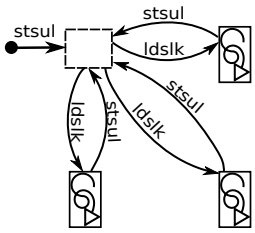

Figure 8: Using shared channel for synchronization.

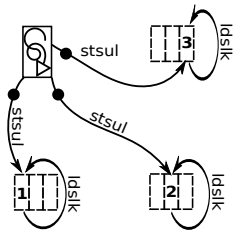

Figure 9: Using private channel for synchronization.

There are two approaches: One is *resource-preferred*, which means a common synchronization channel is shared among the three consumers of a producer (Figure 8). Recall the synchronization process in Listing 5 and Listing 6, the producer thread acquires the lock of the channel buffer first. Then, its three consumers (south, east and south-east neighbors) spin at the channel (it is also possible that they spin at other channels). When the producer fires, it releases a token to the channel. An arbitrary waiting consumer may acquire the token, but as other consumers may still wait for the token, it must restore the token back to the channel after usage. Since three consumers share one synchronization channel, a single lock is enough. However, due to the sharing of the token, a consumer may false-wait for other consumer(s) to restore the token before can fire (In fact, it only has to wait for the producer, but there is no way for it to distinguish).

The other approach is *performance preferred*, meaning that each synchronization uses an isolated channel so that the consumers are independent of each other (Figure 9). So it is possible that the consumers can start firing earlier and they do not have to restore the token afterwards, which may benefit performance. The expense is three times the lock resource cost.

```
if (ty!=0 && tx!=0){
  lock(&mutex[ty][tx]);
  __syncthreads();
  while(!finished){
    if(!north_sync &&
       atomicExch(&mutex[ty-1][tx],1)==0){
      north = s[ty-1][tx];//get north operand
      north_sync = true;
      atomicExch(&mutex[ty-1][tx],0);}
    if(!west_sync &&
       atomicExch(&mutex[ty][tx-1],1)==0){
      west = s[ty][tx-1];//get west operand
      west_sync = true;
      atomicExch(&mutex[ty][tx-1],0);}
    finished = north_sync && west_sync;//all ready?
    if(finished){//fire
      s[ty][tx]=MAX(s[ty-1][tx-1]+p[ty][tx],
                    north-k,west-k);
      unlock(&mutex[ty][tx]);}//put self
} }
```

Listing 14: Atomic-based lock version

```
/*00e8*/  LDSLK P0, RZ, [R7];//lock self
/*00f0*/  BAR.RED.POPC RZ, RZ;
/*00f8*/  SSY 0x170;
/*0100*/  @!P1 LDSLK P1, R11, [R7+-0x4];//get west
/*0108*/  @P1 STSUL [R7+-0x4], R11;//restore token
/*0110*/  @!P3 LDSLK P3, R10, [R7+-0x80];//get north
/*0118*/  @P3 STSUL [R7+-0x80], R10;//restore token
/*0120*/  PSETP.AND.AND P2, pt, P3, P1, pt;//ready?
/*0128*/  @P2 LDS R12, [R7+-0x84];//fire
/*0130*/  @P2 ISETP.GE.AND P4, pt, R10, R11, pt;
/*0138*/  @P2 IADD R12, R12, R4;
/*0140*/  @P2 SEL R13, R10, R11, P4;
/*0148*/  @P2 IADD R13, R13, -R15;
/*0150*/  @P2 ISETP.GE.AND P5, pt, R13, R12, pt;
/*0158*/  @P2 SEL R8, R13, R12, P5;
/*0160*/  @P2 STSUL [R7], R8;//put self
/*0168*/  @!P2 BRA 0x100;
```

Listing 15: Fine-grained lock naive version

In our implementation, concerning the lock bits are limited and a shortage of locks may restrict the volume of actors, we adopt the resource-preferred approach. Meanwhile, for a point $S(i,j)$, it depends on $S(i-1,j-1)$, but as $S(i-1,j)$ and $S(i,j-1)$ also depend on $S(i-1,j-1)$, if any token(s) from $S(i-1,j)$ or $S(i,j-1)$ acquired, $S(i-1,j-1)$ can be safely loaded. The core part of the implementations based on atomic spin-locks and tiny-locks are shown in Listing 14 and Listing 15. To avoid intra-warp synchronization deadlocks (Section 4.4.1), the critical section scheme is used. Furthermore, the thread block configuration is set to be 32x32 to fully leverage the 1024 lock bits of an SM (also to avoid deadlocks due to alias, see Section 4.4.2).

We use the same outer framework as the original code and test the three implementations (data-parallel, atomic spin-lock dataflow, tiny-lock dataflow) on the GTX-570 platform.

The execution time of the kernels are listed in Table 3. As can be seen, our tiny-lock based implementation is far more efficient than the atomic spin-lock approach, with as much as 296x speedup for the 1984x1984 data grid. Compared with the original data-parallel implementation, our tiny-lock method achieves more than 1.15x speedup on small size data grid (less than 248x248), but is slower for larger sizes.

The scalability problem here is incurred by the restrictions on the number of threads and lock bits in an SM. In the data-parallel design, one warp is already sufficient to process a sub-grid, so one thread block contains only 32 threads. However, for the dataflow design, this number is 1024. Consequently, for a large grid size, more sub-grids can be processed simultaneously in the data-parallel approach, as an SM can sustain 8 thread blocks at a time for Fermi. For the dataflow approaches, however, an SM can only support one thread block (In fact, the maximum number of resident threads per SM is 1536 for Fermi, but there are only 1024 lock bits), which severely limits the exploitable parallelism at the thread block level.

If the new generation GPUs integrate more lock bits and allow more threads for a SM, the data-flow scheme could achieve superior performance than the data-parallel scheme, even for large grid sizes.

7. LIMITATIONS

Here we evaluate the limitations of the proposed synchronization scheme. First, in order to use it, one has to do low-level **SASS assembly programming**, which requires significant efforts. The coding process is error-prone and can easily lead to deadlocks, while debugging is almost impossible. However, this situation can be significantly improved if NVIDIA provides specific PTX instructions or CUDA functions to manipulate lock bits. This can also resolve the second limitation – **portability**. As no official SASS assembler is available, although the idea is general, our real hardware testing has to rely on the open-source *asfermi* that only functions smoothly for a portion of instructions for Fermi architecture. Since Kepler has dramatically improved the atomic functionality, we expect the proposed scheme can work more efficiently on the Kepler architecture. The third limitation is the **number of usable lock bits**, which restricts the parallelism and scalability that can be achieved on GPUs.

Grid Size	Atomic-Lock	Data-Parallel	Tiny-Lock
31x31	175 μs	57 μs	49 μs
62x62	466 μs	58 μs	49 μs
124x124	1050 μs	58 μs	50 μs
248x248	2285 μs	59 μs	51 μs
496x496	5052 μs	72 μs	79 μs
992x992	14757 μs	79 μs	109 μs
1984x1984	48808 μs	80 μs	165 μs

Table 3: Execution time for atomic-lock, data-parallel and tiny-lock based Implementations.

8. CONCLUSION

In this paper we proposed a highly efficient lock mechanism on the shared memory of NVIDIA Fermi GPUs. By reassembling the SASS micro-operations that comprise an atomic instruction, we developed a highly efficient, low cost lock approach that can be leveraged to set up a fine-grained producer-consumer synchronization channel between cooperative threads in a thread block. This is the first time that the SASS instructions comprising an atomic operation are used independently to form new synchronization primitives. Furthermore, we showed how to implement a dataflow algorithm on GPUs using a real 2D-wavefront application. This is the first work that explores the possibility of applying lock-based dataflow-style programming model on GPUs.

Although programming with locks for the current platform/assembler is low-level and deadlock-prone, our work is already sufficient to show the possibility and potential of lock-based dataflow programming for GPUs. We expect more developers, especially architects and library writers to see such potential and participate in exploring and simplifying the programmability of this new design pattern.

9. ACKNOWLEDGMENTS

This work was supported by Singapore Ministry of Education Academic Research Fund Tier 1 (No.R263-000-B02-112). The authors would like to thank all the anonymous reviewers for their insightful suggestions and feedback.

10. REFERENCES

[1] Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education, 2006.

[2] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1), 1990.

[3] Thomas E Anderson, Edward D Lazowska, and Henry M Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *Computers, IEEE Transactions on*, 38(12), 1989.

[4] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. *The impact of synchronization and granularity on parallel systems*, volume 18. ACM, 1990.

[5] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 35–45. ACM, 2007.

[6] Dean M Tullsen, Jack L Lo, Susan J Eggers, and Henry M Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *High-Performance Computer Architecture. Proceedings. Fifth International Symposium On*, pages 54–58. IEEE, 1999.

[7] William E Cohen, Henry G Dietz, and JB Sponaugle. Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors. In *Parallel Processing. International Conference on*, volume 1. IEEE, 1994.

[8] Alexandru Nicolau, Guangqiang Li, and Arun Kejariwal. Techniques for efficient placement of synchronization primitives. In *ACM Sigplan Notices*, volume 44, pages 199–208. ACM, 2009.

[9] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *Computers, IEEE Transactions on*, 36(12), 1987.

[10] Martin C Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4), 1999.

[11] W Hwu Wen-Mei. *GPU Computing Gems Emerald Edition*. Elsevier, 2011.

[12] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5), 2008.

[13] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[14] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. Fermi architecture white paper. `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`.

[15] Shucai Xiao and Wu-Chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Parallel and Distributed Processing, IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[16] Jeff A Stuart and John D Owens. Efficient synchronization primitives for GPUs. *arXiv preprint arXiv:1110.4623*, 2011.

[17] NVIDIA. PTX: Parallel Thread Execution ISA Version 4.0. `http://docs.nvidia.com/cuda/parallel-thread-execution/index.html`.

[18] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 12. ACM, 2011.

[19] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: leveraging warp specialization for high performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2014.

[20] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013.

[21] Feng Ji and Xiaosong Ma. Using shared memory to accelerate mapreduce on graphics processing units. In *Parallel and Distributed Processing Symposium, IEEE International*, pages 805–816. IEEE, 2011.

[22] Ashwin M Aji and Wu-Chun Feng. Accelerating data-serial applications on data-parallel GPGPUs: a systems approach. Technical report, TR-08-24, Computer Science, Virginia Tech, 2008.

[23] Brett W Coon, Peter C Mills, John R Nickolls, and Lars Nyland. Lock mechanism to enable atomic updates to shared memory, November 8 2011. US Patent 8,055,856.

[24] Juan Gomez-Luna, José Maria González-Linares, Jose Ignacio Benavides Benitez, and Nicolas Guil Mata. Performance modeling of atomic additions on GPU scratchpad memory. *Parallel and Distributed Systems, IEEE Transactions on*, 24(11), 2013.

[25] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[26] Yunqing Hou. Asfermi: An assembler for the NVIDIA Fermi instruction set. `http://code.google.com/p/asfermi/`, 2011.

[27] Carlos ER Alves, Edson Norberto Cáceres, Frank Dehne, and Siang W Song. A parallel wavefront algorithm for efficient biological sequence comparison. In *Computational Science and Its Applications*, pages 249–258. Springer, 2003.

[28] Hsien-Yu Liao, Meng-Lai Yin, and Yi Cheng. A parallel implementation of the smith-waterman algorithm for massive sequences searching. In *Engineering in Medicine and Biology Society. 26th Annual International Conference of the IEEE*, volume 2, pages 2817–2820. IEEE, 2004.

[29] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[30] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2), 1974.

[31] Simon J Pennycook, Gihan R Mudalige, Simon D Hammond, and Stephen A Jarvis. Parallelising wavefront applications on general-purpose GPU devices, 2010.

[32] George Teodoro, Tony Pan, Tahsin M Kurc, Jun Kong, Lee AD Cooper, and Joel H Saltz. Efficient irregular wavefront propagation algorithms on hybrid CPU–GPU machines. *Parallel computing*, 39(4), 2013.