# To GPU Synchronize or Not GPU Synchronize?

Wu-chun Feng[†*] and Shucai Xiao[*]

[†]Department of Computer Science
[*]Department of Electrical and Computer Engineering
Virginia Tech
Email: {wfeng, shucai}@vt.edu

***Abstract*— The graphics processing unit (GPU) has evolved from being a fixed-function processor with programmable stages into a programmable processor with many fixed-function components that deliver massive parallelism. By modifying the GPU's stream processor to support "general-purpose computation" on the GPU (GPGPU), applications that perform massive vector operations can realize many orders-of-magnitude improvement in performance over a traditional processor, i.e., CPU.**

**However, the breadth of general-purpose computation that can be efficiently supported on a GPU has largely been limited to highly data-parallel or task-parallel applications due to the lack of explicit support for communication between streaming multiprocessors (SMs) on the GPU. Such communication can occur via the global memory of a GPU, but it then requires a barrier synchronization across the SMs of the GPU in order to complete the communication between SMs.**

**Although our previous work demonstrated that implementing barrier synchronization on the GPU itself can significantly improve performance and deliver correct results in critical bioinformatics applications, guaranteeing the correctness of inter-SM communication is only possible if a memory consistency model is assumed. To address this problem, NVIDIA recently introduced the `__threadfence()` function in CUDA 2.2, a function that can guarantee the correctness of GPU-based inter-SM communication. However, this function currently introduces so much overhead that when using it in (direct) GPU synchronization, GPU synchronization actually performs worse than indirect synchronization via the CPU, thus raising the question of whether "to GPU synchronize or not GPU synchronize?"**

## I. Introduction

Originally, the massive parallelism offered by the GPU only supported calculations for 3D computer graphics, such as texture mapping, polygon rendering, vertex rotation and translation, and oversampling and interpolation to reduce aliasing. However, because many of these graphics computations entail matrix and vector operations, the GPU is also increasingly being used for non-graphical calculations.

Specifically, GPU companies have modified the notion of a stream processor that only performs graphical operations into one that allows for "general-purpose computation" on a graphics processing unit (GPGPU). In addition, programming models such as NVIDIA's Compute Unified Device Architecture [10], AMD/ATI's Brook+ [2], and the recently available OpenCL [8] allow applications to be more easily mapped onto the GPU.

GPUs typically map well only to data-parallel or task-parallel applications whose execution requires relatively minimal communication between streaming multiprocessors (SMs) on the GPU [6], [11], [13], [15]. This proclivity is mainly due to the lack of support for communication between streaming multiprocessors (SMs) on the GPU. While such communication can occur via the global memory of a GPU, it requires a barrier synchronization across the SMs in order to complete the communication between the SMs.

Traditionally, such inter-SM communication on the GPU is achieved by (inefficiently) implementing the barrier synchronization via the host CPU, which we refer to hereafter as *CPU barrier synchronization*. This synchronization occurs by terminating the current GPU-offloaded computation and then re-launching a new GPU-offloaded computation. To reduce this synchronization overhead of having to move from the GPU to the CPU and back to the GPU, we proposed to support barrier synchronization on the GPU itself via our `__gpu_sync()` function [16], [17], which can be implemented in two ways: (1) GPU *lock-based* synchronization and (2) GPU *lock-free* synchronization.

Using our approaches to GPU barrier synchronization in implementing two bioinformatics-related algorithms on the GPU (i.e., dynamic programming (DP) for genomic sequence alignment [12] and bitonic sort (BS) [9]), our GPU lock-based synchronization improved the performance of DP and BS by 11.70% and 16.64%, respectively, when compared to the same algorithms implemented using CPU barrier synchronization. Our GPU lock-free synchronization fared even better and improved performance by 25.47% and 40.39%, respectively [17].

However, our GPU lock-based and lock-free barriers *theoretically* run the risk that write operations performed before our `__gpu_sync()` barrier are *not* completed by the time the GPU is released from the barrier. *In practice*, it is infinitesimally unlikely that this will ever happen given the amount of time that is spent spinning at the barrier, e.g., none of our thousands of experimental runs ever resulted in an incorrect answer. Furthermore, no existing literature has been able to show how to trigger this type of error. But again, it is still *theoretically* possible.

To resolve this theoretical issue, GPU vendors recently introduced functionality that can guarantee that the above risk cannot happen and can thus guarantee the correctness of GPU-based inter-SM communication. In the case of NVIDIA, for example, the `__threadfence()` function was introduced into CUDA 2.2. When `__threadfence()` is called, the calling thread waits until its prior writes to global memory and shared memory are visible to other threads. By integrating this functionality into our `__gpu_sync()` barrier, the correctness of inter-SM communication can be guaranteed.

Unfortunately, this first incarnation of `__threadfence()` incurs so much overhead in our GPU-based barrier synchronizations that it eliminates much of the aforementioned performance improvements in the two algorithms, i.e., DP and BS. That is, CPU barrier synchronization performs as well as or better than the GPU barrier synchronizations in many cases, thus leaving us with the question of whether "to GPU synchronize or not GPU synchronize?"

The rest of the paper is organized as follows. Section II briefly describes the GPU programming model, i.e., CUDA, that we use to study barrier synchronization on the GPU. Section III presents work related to synchronization protocols in the context of multi- and many-core environments. Section IV provides an brief overview of our proposed implementations of GPU barrier synchronization [17]. In Section V, we outline a potential problem with GPU-based barrier synchronization and discuss its current solution as well as its asso-

ciated costs. Section VI then analyzes these costs in synchronization overhead via a micro-kernel benchmark. Finally, Section VII presents our conclusions.

## II. CUDA Programming Model

CUDA, short for Compute Unified Device Architecture, provides a parallel computing architecture that is programmable via industry-standard programming langauges like C. Provided by NVIDIA, CUDA enables users to write multi-threaded programs to run on CUDA-enabled graphics processing units (GPUs). When a program is mapped to the GPU, only the computation-intensive and/or data-parallel parts are parallelized to take advantage of the massive parallelism available in a GPU.[1] These computation-intensive and/or data-parallel parts are implemented as *kernels* and compiled to the device instruction set. In a kernel, threads are grouped as a grid of thread blocks, and each thread block contains a number of threads. *Multiple blocks can be executed on the same SM, but one block cannot be executed across different SMs.*

CUDA provides a data communication mechanism for threads *within* a single block via the barrier function `__syncthreads()`, i.e., intra-SM communication. However, there is no explicit software or hardware support for data communication of threads *across* different blocks, i.e. *inter-SM communication*. Currently, such communication occurs via the global memory of the GPU, and the needed barriers are implemented via the CPU by terminating the current kernel's execution and re-launching a new kernel.

## III. Related Work

Due to the volume of research conducted on synchronization protocols, we necessarily limit our focus to such protocols in the context of multi- and many-core processor environments.

In a multi-core environment, many types of synchronization approaches [1], [3], [5] have been proposed. However, none of them can be directly used on GPUs. This is because multiple GPU thread blocks can be scheduled to execute on a single streaming multiprocessor (SM) simultaneously, and CUDA blocks do not yield their execution. This means that once a thread block is spawned by the CUDA thread scheduler, other blocks cannot start their execution until execution of the spawned block is completed. Thus, deadlocks could occur, and they cannot be resolved in the same way as in a multi-core environment, where a process can yield its execution to other processes. One way to address this problem is to assign only one block per SM, which can be implemented by allocating all the shared memory of an SM for a single thread block [17].

With respect to a GPU many-core environment, CUDA provides a synchronization function `__syncthreads()` to synchronize the execution of different threads *within* a block. However, when a barrier synchronization is needed *across* different blocks, programmers traditionally use a kernel launch as a way to *implicitly* barrier synchronize [4], [7].

Besides kernel launches, Stuart et al. [14] propose a protocol for data communication across multiple GPUs, i.e., inter-GPU communication. Though this approach can be used for inter-SM communication, its performance will be quite poor because, in this approach, data needs to be transferred to the host memory first and then copied back to the device memory, which is unnecessary for data communication across different SMs on a single GPU card.

[1]On the NVIDIA GTX 280, up to 1024 threads can be active per streaming multiprocessor (SM). With 30 SMs on the GTX 280, this means that 30,720 threads can be simultaneously active on a GPU.

## IV. GPU-Based Barrier Synchronization

In [16], [17], we propose two methodologies for GPU-based barrier synchronization: (1) GPU *lock-based* synchronization and (2) GPU *lock-free* synchronization. For the former, a mutually exclusive (mutex) variable controls the execution of different blocks on SMs. Once a block finishes its computation on an SM, it atomically increments the mutex variable. Only after all thread blocks finish their computation will the mutex variable be equal to the target value and the barrier complete. In contrast, GPU lock-free synchronization uses one distinct variable to control *each* block, thus eliminating the need for different blocks to contend for the single mutex variable. By eliminating the single mutex variable, the need for atomic addition is removed.

### *Application Performance Improvement with GPU Synchronization*

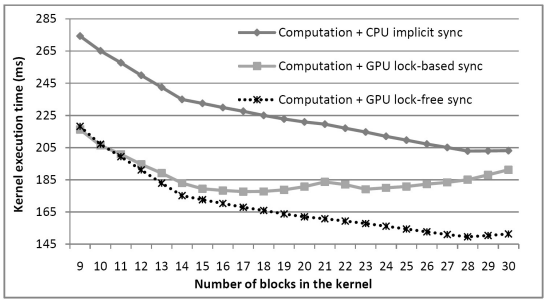
To evaluate the performance of our GPU synchronization approaches, we implemented them in two bioinformatics-related algorithms — dynamic programming (DP) for genomic sequence alignment (specifically the Smith-Waterman algorithm [12]) and bitonic sort (BS) [9] — and ran them on a GeForce GTX 280 video card. The GTX 280 consists of 30 SMs, where each SM contains 8 processing cores running at 1.3 GHz, for a total of 240 processing cores. The on-chip memory of each SM comprises 16K registers and 16KB shared memory, which can only be accessed by threads on the SM. Outside the SMs on the rest of the GPU card, there is 1GB GDDR3 global memory with a bandwidth of 141.7 GB/second. Atop this GPU, we use the NVIDIA CUDA 2.2 SDK toolkit. The host system for the GPU contains a 2.2-GHz Intel Core 2 Duo CPU with 2MB L2 cache and 2×2GB DDR2 SDRAM. The operating system on the host machine is 64-bit Ubuntu GNU/Linux 8.10.

Figure 1 shows the kernel execution time with different synchronization approaches and the time variation versus the number of blocks in the kernel, averaged over three runs. As shown in Figure 1, the kernel execution time improves by 25.47% and 40.39% for the dynamic programming of Smith-Waterman and bitonic sort, respectively, when GPU lock-free synchronization across 30 blocks is used instead of CPU barrier synchronization. When comparing the two GPU synchronization methods to each other, the performance of the GPU lock-free synchronization is consistently better than that of the GPU lock-based synchronization. The more blocks that are configured in the kernel, the larger the performance difference is. This behavior is due to the atomic addition in the GPU lock-based synchronization, which can only be executed serially. Thus, the more blocks that are in the kernel, the more time that is needed to execute the barrier synchronization function. In contrast, all operations in the GPU lock-free synchronization can be executed in parallel, and the time needed for the barrier synchronization is independent of the number of blocks.

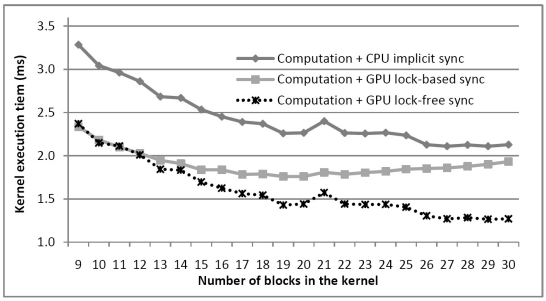
## V. Cost of Guaranteeing GPU Synchronization Correctness

Theoretically, our GPU lock-based and lock-free barriers run the infinitesimal risk (at least on the GeForce GTX 280) that write operations performed before our `__gpu_sync()` barrier are *not* completed by the time the GPU is released from the barrier. Why? The CUDA SDK function `__syncthreads()` that is used in the barrier synchronization function can only guarantee writes to shared memory and global memory visible to threads of the *same* block, it cannot do so for threads across *different* blocks.

To address this problem, NVIDIA introduced `__threadfence()` in CUDA 2.2. When `__threadfence()` is called, the calling thread

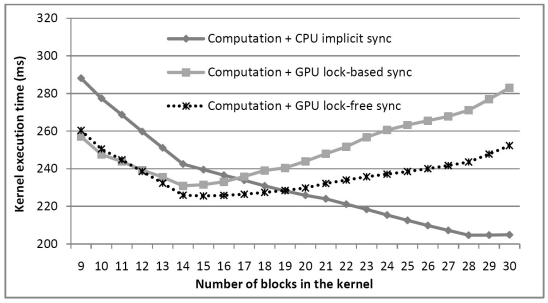



(a) Dynamic Programming: Smith-Waterman

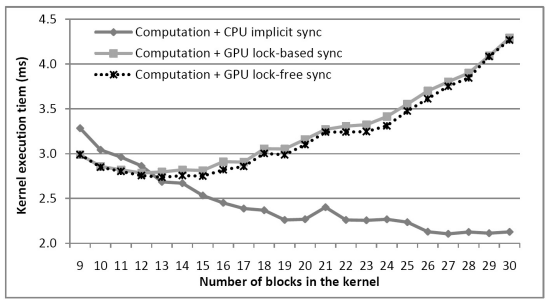



(b) Bitonic Sort

Fig. 1. Kernel Execution Time versus Number of Blocks in the Kernel




(a) Dynamic Programming: Smith-Waterman




(b) Bitonic sort

Fig. 2. Kernel Execution Time versus Number of Blocks in the Kernel with `__threadfence()`

waits until its prior writes to global memory and shared memory are visible to other threads. By integrating this functionality into our `__gpu_sync()` barrier, the correctness of inter-SM communication can be guaranteed.

However, the overhead incurred by calling `__threadfence()` in our `__gpu_sync()` barrier significantly impacts the performance of the barrier. Figure 2 shows the performance of the same two algorithms in Section IV with `__threadfence()` called in `__gpu_sync()`. Clearly, the kernel execution time with GPU synchronization increases substantially. Compared to the dynamic programming (DP) implementations with CPU implicit synchronization, the GPU lock-based and lock-free implementations only perform better when the number of blocks in the kernel is less than 17 and 19, respectively. For bitonic sort (BS), the number of blocks needs to be less than 13 in both cases. Furthermore, with `__threadfence()` added into our GPU-based barrier `__gpu_sync()`, the kernel execution time for GPU synchronization increases further if there are more blocks configured in the kernel. This is because the execution time of `__threadfence()` increases if threads in more blocks call it. With 30 blocks configured in the kernel, compared to the CPU synchronization, the kernel execution times increase by 23.14% and 100.70% for DP and BS, respectively.

## VI. Fine-Grained Analysis of GPU Barrier Synchronization

In this section, we analyze the barrier synchronization overhead by partitioning it into the time consumed for each operation within the synchronization. As an example, we analyze GPU lock-based synchronization, which contains a superset of all the operations that are used in the GPU lock-free synchronization. With respect to the barrier implementation in [17], there are four types of operations in the GPU lock-based synchronization, and its synchronization overhead can be expressed as

$$T_S = t_a + t_c + t_s + t_f \quad (1)$$

where $t_a$ is the overhead of atomic add, $t_c$ is the mutex variable checking time, $t_s$ is the time consumed by `__syncthreads()`, and $t_f$ is the `__threadfence()` execution time. Unfortunately, the execution times for these component operations cannot be measured directly on the GPU. Thus, we use an indirect approach to infer the times. Specifically, we measure the kernel execution time across different scenarios, and then calculate the execution time of each of the above operations. Based on the kernel execution time model in [17], a kernel's execution time can be expressed as

$$T = t_O + t_{Com} + t_S \quad (2)$$

where $t_O$ is the kernel launch time, $t_{Com}$ is the computation time, and $t_S$ is the synchronization time. By combining Equations (1) and (2), the kernel execution time can be represented as

$$T = t_O + t_{Com} + t_a + t_c + t_s + t_f \quad (3)$$

From Equation (3), we can calculate the overhead of a particular operation, e.g., `__threadfence()` by measuring the kernel execution time both with and without `__threadfence()` and taking the time difference as the overhead of `__threadfence()`.

With the above indirect approach, we use a micro-benchmark to measure kernel execution times in different scenarios. The micro-benchmark calculates the average of two floats over 10,000 iterations. If CPU synchronization is used, each kernel calculates the average once, and the kernel is launched 10,000 times; while for GPU synchronization, the kernel is launched only once, and there is a 10,000-iteration `for` loop used in the kernel with the GPU barrier function called in each loop. In addition, the micro-benchmark is set with each thread calculating one element, no matter how many threads and blocks are set in the kernel. The more blocks and threads are set, the more elements are computed, i.e., weak scaling. So, the computation time should be approximately constant. As before, each result is the average of three runs.

We then measure kernel execution times in the following scenarios:

1) Sum of the kernel launch and computation time, i.e., $t_1 = t_O + t_{Com}$, which is the kernel execution time of a GPU implementation but without the barrier function `__gpu_sync()`.
2) Kernel execution time with one `atomicAdd` called in each block, i.e., $t_2 = t_a$.
3) Sum of the time for kernel launch, computation, and `__syncthreads()`, i.e., $t_3 = t_O + t_{Com} + t_s$.
4) Kernel execution time of the GPU lock-based synchronization without `__threadfence()`, so $t_4 = t_O + t_{Com} + t_s + t_a + t_c$.
5) Kernel execution time of the GPU lock-based synchronization with `__threadfence()`, thus $t_5 = t_O + t_{Com} + t_s + t_a + t_c + t_f$.

Figure 3 shows the measured execution times of $t_1$ to $t_5$ noted above. With these times, the overhead of the four types of operations in the GPU lock-based synchronization can be calculated as:

1) Time for executing the atomic add is $t_2$, i.e., $t_a = t_2$;
2) Time of the mutex variable checking is $t_c = t_4 - t_3 - t_2$;
3) Time consumption of `__syncthreads()` is $t_s = t_3 - t_1$;
4) Overhead of the function `__threadfence()` is $t_f = t_5 - t_4$.

Thus, for 10,000 times of execution, $t_s = 0.541$, $t_a = 2.300 \times n$, $t_c = 5.564$, and $t_f = 0.333 \times n + 7.267$, where $n$ is the number of blocks in the kernel, and the units are in *milliseconds*.

These results show that the intra-block synchronization function `__syncthreads()` consumes very little time compared the other three operations. With 10,000 times of execution, the execution time is only 0.541 ms, which is about 70 clock cycles per call on the GTX 280 and is a constant value that is unrelated to the number of threads that call it. Similarly, the execution time of the mutex variable checking is 5.564 ms for 10,000 iterations and is unrelated to the number of threads. On the other hand, if we analyze the `atomicAdd()` and `__threadfence()` functions, their execution times are directly related to the number of blocks that call them, as shown in Figure 3. In addition, they are more expensive than the other two operations. Hence, to improve the performance of the GPU lock-based synchronization, efficient atomic add and memory flush functions are needed.

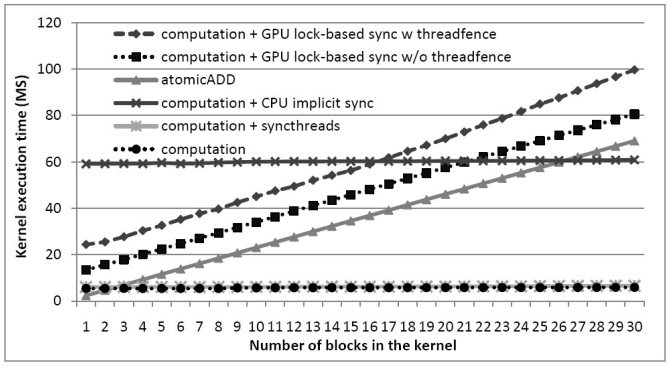



Fig. 3. Profile of GPU Lock-Based Synchronization via a Micro-Benchmark

## VII. Conclusion

Using two bioinformatics-related algorithms, we demonstrate the efficacy of our inter-SM communication via GPU-based barrier synchronization. While our GPU lock-based and lock-free barriers *theoretically* run the risk that write operations performed before the barrier are *not* completed by the time the GPU is released from the barrier, the likelihood of this happening on the GeForce GTX 280 generation of video cards is infinitesimally small *in practice*.

To address this theoretical issue, NVIDIA recently introduced a new function called `__threadfence()`, which guarantees that all writes to shared memory and global memory are visible to other threads. But because the overhead that `__threadfence()` incurs is so high, the performance advantage of GPU synchronization with `__threadfence()` is less clear. For a smaller number of blocks, it still performs better than CPU synchronization, but for a large number of blocks, CPU synchronization performs better.

So, to answer the question of whether "to GPU synchronize or not GPU synchronize?" We grudgingly conclude that one *should* GPU synchronize (with or without `__threadfence()`) on the current generation of NVIDIA video cards, i.e., GTX 280. For the next-generation Fermi GPU and its projected efficient implementations of atomic operations and memory flushing via `__threadfence()`, the answer will be a more definitive 'yes' to GPU synchronize.

## References


[1] J. Alemany and E. W. Felten. Performance Issues in Non-Blocking Synchronization on Shared-Memory Multiprocessors. In *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*, August 1992.

[2] AMD/ATI. Stream Computing User Guide. April 2009. `http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf`.

[3] G. Barnes. A Method for Implementing Lock-Free Shared-Data Structures. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, June 1993.

[4] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proc. of Supercomputing (SC)*, October 2008.

[5] R. Gupta and C. R. Hill. A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree. *International Journal of Parallel Programming*, 18(3):161–180, 1989.

[6] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proc. of the IEEE Int'l Conf. on High Performance Computing*, December 2007.

[7] M. Hussein, A. Varshney, and L. Davis. On Implementing Graph Cuts on CUDA. In *First Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

[8] Khronos. OpenCL: Parallel Computing for Heterogeneous Devices, 2009. `http://www.khronos.org/developers/library/overview/opencl_overview.pdf`.

[9] H. W. Lang. Bitonic Sort. 1997. `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm`.

[10] NVIDIA. NVIDIA CUDA Programming Guide-2.2, 2009. `http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf`.

[11] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 73–82, Feb. 2008.

[12] T. Smith and M. Waterman. Identification of Common Molecular Subsequences. In *Journal of Molecular Biology*, April 1981.

[13] G. M. Striemer and A. Akoglu. Sequence Alignment with GPU: Performance and Design Challenges. In *IPDPS*, May 2009.

[14] J. A. Stuart and J. D. Owens. Message Passing on Data-Parallel Architectures. In *23rd IEEE Int'l Parallel & Distributed Processing Symp.*, May 2009.

[15] V. Volkov and B. Kazian. Fitting FFT onto the G80 Architecture. pages 25–29, April 2006. `http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6-report.pdf`.

[16] S. Xiao, A. Aji, and W. Feng. On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In *Proc. of the 15th Int'l Conf. on Parallel and Distributed Systems*, December 2009.

[17] S. Xiao and W. Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (to appear)*, April 2010.