Inter-Block GPU Communication via Fast Barrier Synchronization

Shucai Xiao* and Wu-chun Feng*†

*Department of Electrical and Computer Engineering
†Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24061
Email: {shucai, wfeng}@vt.edu

***Abstract*—While GPGPU stands for general-purpose computation on graphics processing units, the lack of explicit support for inter-block communication on the GPU arguably hampers its broader adoption as a general-purpose computing device. Inter-block communication on the GPU occurs via global memory and then requires barrier synchronization across the blocks, i.e., inter-block GPU communication via barrier synchronization. Currently, such synchronization is only available via the CPU, which in turn, can incur significant overhead.**

We propose two approaches for inter-block GPU communication via barrier synchronization: GPU lock-based synchronization and GPU lock-free synchronization. We then evaluate the efficacy of each approach via a micro-benchmark as well as three well-known algorithms — Fast Fourier Transform (FFT), dynamic programming, and bitonic sort. For the micro-benchmark, the experimental results show that our GPU lock-free synchronization performs 8.4 times faster than CPU explicit synchronization and 4.0 times faster than CPU implicit synchronization. When integrated with the FFT, dynamic programming, and bitonic sort algorithms, our GPU lock-free synchronization further improves performance by 10%, 26%, and 40%, respectively, and ultimately delivers an overall speed-up of 70x, 13x, and 24x, respectively.

I. Introduction

Today, improving the computational capability of a processor comes from *increasing its number of processing cores* rather than increasing its clock speed. This is reflected in both traditional multi-core processors and many-core graphics processing units (GPUs).

Originally, GPUs were designed for graphics-based applications. With the elimination of key architecture limitations, GPUs have evolved to become more widely used for general-purpose computation, i.e., general-purpose computation on the GPU (GPGPU). Programming models such as NVIDIA's Compute Unified Device Architecture (CUDA) [22] and AMD/ATI's Brook+ [2] enable applications to be more easily mapped onto the GPU. With these programming models, more and more applications have been mapped to GPUs and accelerated [6], [7], [10], [12], [18], [19], [23], [24], [26], [30].

However, GPUs typically map well only to data or task parallel applications whose execution requires minimal or even no inter-block communication [9], [24], [26], [30]. Why? There exists no explicit support for inter-block communication on the GPU. Currently, such inter-block communication occurs via global memory and requires a barrier synchronization to complete the communication, which is (inefficiently) implemented via the host CPU. Hereafter, we refer to such CPU-based barrier synchronization as *CPU synchronization*.

In general, when a program (i.e., kernel) executes on the GPU, its execution time consists of three phases: (1) kernel launch to the GPU, (2) computation on the GPU, and (3) inter-block GPU communication via barrier synchronization.[1] With different approaches for synchronization, the percentage of time that each of these three phases takes will differ. Furthermore, some of the phases may overlap in time. To quantify the execution time of each phase, we propose a general performance model that partitions the kernel execution time into the three aforementioned phases. Based on our model and code profiling while using the current state of the art in barrier synchronization, i.e., CPU implicit synchronization (see Section IV), inter-block communication via barrier synchronization can consume more than 50% of the total kernel execution time, as shown in Table I.

TABLE I
Percent of Time Spent on Inter-Block Communication

Algorithms	FFT	SWat	Bitonic sort
% of time spent on inter-block communication	17.8%	49.2%	59.6%

(SWat: Smith-Waterman)

Hence, in contrast to previous work that mainly focuses on optimizing the GPU computation, we focus on reducing the inter-block communication time via barrier synchronization. To achieve this, we propose a set of *GPU synchronization* strategies, which can synchronize the execution of different blocks without the involvement of the host CPU, thus avoiding the costly operation of a kernel launch from the CPU to GPU. To the best of our knowledge, this work is the first that systematically addresses how to better support more general-purpose computation by significantly reducing the inter-block

[1]Because inter-block GPU communication time is dominated by the inter-block synchronization time, we will use *inter-block synchronization time* instead of inter-block GPU communication time hereafter.

communication time (rather than the computation time) on a GPU.

We propose two types of GPU synchronization, one with locks and the other without. For the former, we use one mutual-exclusive (mutex) variable and an atomic add operation to implement *GPU lock-based synchronization*. With respect to the latter, which we refer to as *GPU lock-free synchronization*, we use two arrays, instead of mutex variables, and eliminate the need for atomic operations. With this approach, each thread within a single block controls the execution of a different block, and the intra-block synchronization is achieved by synchronizing the threads within the block with the existing barrier function `__syncthreads()`.

We then introduce these GPU synchronization strategies into three different algorithms — Fast Fourier Transform (FFT) [16], dynamic programming (e.g., Smith-Waterman [25]), and bitonic sort [4] — and evaluate their effectiveness. Specifically, based on our performance model, we analyze the percentage of time spent computing versus synchronizing for each of the algorithms.

Finally, according to the work of Volkov et al. [29], correctness of inter-block communication via GPU synchronization cannot be guaranteed unless a memory consistency model is assumed. To solve this problem, a new function `__threadfence()` is introduced in CUDA 2.2. This function will block the calling thread until prior writes to global memory or shared memory visible to other threads [22]. It is expected that additional overhead will be caused by integrating `__threadfence()` into our barrier functions. From our experiment results, when the number of blocks is more than 18 in the kernel, performance of all three algorithms are worse than that with the CPU implicit synchronization. As a result, though barriers can be implemented efficiently in software, guaranteeing the inter-block communication correctness with `__threadfence()` causes a lot of overhead, then implementing efficient barrier synchronization via hardware or improving the memory flush efficiency become necessary for efficient and correct inter-block communication on GPUs. It is worth noting that even without `__threadfence()` called in our barrier functions, all results are correct in our thousands of runs.

Overall, the contributions of this paper are four-fold. First, we propose two GPU synchronization strategies for inter-block synchronization. These strategies do *not* involve the host CPU, and in turn, reduce the synchronization time between blocks. Second, we propose a performance model for kernel execution time and speedup that characterizes the efficacy of different synchronization approaches. Third, we integrate our proposed GPU synchronization strategies into three widely used algorithms — Fast Fourier Transform (FFT), dynamic programming, and bitonic sort — and obtain performance improvements of 9.08%, 25.47%, and 40.39%, respectively, over the traditional CPU synchronization approach. Fourth, we show the cost of guaranteeing inter-block communication correctness via `__threadfence()`. From our experiment results, though our proposed barrier synchronization is efficient, the low efficacy of `__threadfence()` causes a lot of overhead, especially when the number of blocks in a kernel is large.

The rest of the paper is organized as follows. Section II provides an overview of the NVIDIA GTX 280 architecture and CUDA programming model. The related work is described in Section III. Section IV presents the time partition model for kernel execution time. Section V describes our GPU synchronization approaches. In Section VI, we give a brief description of the algorithms that we use to evaluate our proposed GPU synchronization strategies, and Section VII presents and analyzes the experimental results. Section VIII concludes the paper.

II. Overview of CUDA on the NVIDIA GTX 280

The NVIDIA GeForce GTX 280 GPU card consists of 240 streaming processors (SPs), each clocked at 1296 MHz. These 240 SPs are grouped into 30 streaming multiprocessors (SMs), each of which contains 8 streaming processors. The on-chip memory for each SM contains 16,384 registers and 16 KB of shared memory, which can only be accessed by threads executing on that SM; this grouping of threads on an SM is denoted as a *block*. The off-chip memory (or device memory) contains 1 GB of GDDR3 global memory and supports a memory bandwidth of 141.7 gigabytes per second (GB/s). Global memory can be accessed by all threads and blocks on the GPU, and thus, is often used to communicate data across different blocks via a CPU barrier synchronization, as explained later.

NVIDIA provides the CUDA programming model and software environment [22]. It is an extension to the C programming language. In general, only the compute-intensive and data-parallel parts of a program are parallelized with CUDA and are implemented as *kernels* that are compiled to the device instruction set. A kernel must be launched to the device before it can be executed.

In CUDA, threads within a block can communicate via shared memory or global memory. The barrier function `__syncthreads()` ensures proper communication. We refer to this as *intra-block communication*.

However, there is no explicit support for data communication across different blocks, i.e., *inter-block communication*. Currently, this type of data communication occurs via global memory, followed by a barrier synchronization via the CPU. That is, the barrier is implemented by terminating the current kernel's execution and re-launching the kernel, which is an expensive operation.

III. Related Work

Our work is most closely related to two areas of research: (1) algorithmic mapping of data parallel algorithms onto the GPU, specifically for FFT, dynamic programming, and bitonic sort and (2) synchronization protocols in multi- and many-core environments.

To the best of our knowledge, all known algorithmic mappings of FFT, dynamic programming, and bitonic sort take the same general approach. The algorithm is mapped onto the GPU in as much of a "data parallel" or "task parallel" fashion as possible in order to minimize or even eliminate inter-block communication because such communication requires an expensive barrier synchronization. For example, running a single (constrained) problem instance per SM, i.e., 30 separate problem instances on the NVIDIA GTX 280, obviates the need for inter-block communication altogether.

To accelerate FFT [16], Govindaraju et al. [6] use efficient memory access to optimize FFT performance. Specifically, when the number of points in a sequence is small, shared memory is used; if there are too many points in a sequence to store in shared memory, then techniques for coalesced global memory access are used. In addition, Govindaraju et al. propose a hierarchical implementation to compute a large sequence's FFT by combining the FFTs of smaller subsequences that can be calculated on shared memory. In all of these FFT implementations, the necessary barrier synchronization is done by the CPU via kernel launches. Another work is that of Volkov et al. [30], which tries to accelerate the FFT by designing a hierarchical communication scheme to minimize inter-block communication. Finally, Nukada et al. [20] accelerate the 3-D FFT through shared memory usage and optimizing the number of threads and registers via appropriate localization. Note that all of the aforementioned approaches focus on optimizing the GPU computation and minimizing or eliminating the inter-block communication rather than by optimizing the performance of inter-block communication.

Past research on mapping dynamic programming, e.g., the Smith-Waterman (SWat) algorithm, onto the GPU uses graphics primitives [14], [15] in a task parallel fashion. More recent work uses CUDA, but again, largely in a task parallel manner [18], [19], [26] or in a fine-grain parallel approach [31]. In the task parallel approach, no inter-block communication is needed, but the problem size it supports is limited to 1K characters. While the fine-grain parallel approach can support sequences of up to 7K characters, inter-block communication time consumes about 50% of the total matrix filling time. So if a better inter-block synchronization method is used, performance improvements can be obtained.

For bitonic sort, Greβ et al. [7] improve the algorithmic complexity of GPU-ABisort to $O(n \log n)$ with an adaptive data structure that enables merges to be done in linear time. Another parallel implementation of the bitonic sort is in the CUDA SDK [21], but there is only one block in the kernel to use the available barrier function `__syncthreads()`, thus restricting the maximum number of items that can be sorted to 512 — the maximum number of threads in a block. If our proposed inter-block GPU synchronization is used, multiple blocks can be set in the kernel, which in turn, will significantly increase the maximum number of items that can be sorted.

Many types of software barriers have been designed for shared-memory environments [1], [3], [8], [11], [17], but none of them can be directly applied to GPU environments. This is because multiple CUDA thread blocks can be scheduled to be executed on a single SM and the CUDA blocks do not yield to the execution. That is, blocks run to completion once spawned by the CUDA thread scheduler. This may result in deadlocks, and thus, cannot be resolved in the same way as in traditional CPU processing environments, where one can yield the waiting process to execute other processes. One way of addressing this is our GPU lock-based barrier synchronization [31]. This approach leverages a traditional shared mutex barrier and avoid deadlock by ensuring a one-to-one mapping between the SMs and the thread blocks.

Cederman et al. [5] implement a dynamic load-balancing method on the GPU that is based on the lock-free synchronization method found on traditional multi-core processors. However, this scheme controls task assignment instead of addressing inter-block communication. In addition, we note that lock-free synchronization generally performs worse than lock-based methods on traditional multi-core processors, but its performance is better than that of the lock-based method on the GPU in our work.

The work of Stuart et al. [27] focuses on data communication between multiple GPUs, i.e., inter-GPU communication. Though their approach can be used for inter-block communication across different SMs on the same GPU, the performance is projected to be quite poor because data needs to be moved to the CPU host memory first and then transferred back to the device memory, which is unnecessary for data communication on a single GPU card.

The most closely related work to ours is that of Volkov et al. [29]. Volkov et al. propose a global software synchronization method that does not use atomic operations to accelerate dense linear-algebra constructs. However, as [29] notes, their synchronization method has not been implemented into any real application to test the performance improvement. Furthermore, their proposed synchronization cannot guarantee that previous accesses to all levels of the memory hierarchy have completed. Finally, Volkov et al. used only one thread to check all *arrival* variables, hence serializing this portion of inter-block synchronization and adversely affecting its performance. In contrast, our proposed GPU synchronization approaches guarantee the completion of memory accesses with the existing memory access model in CUDA. This is because a new function `__threadfence()` is added in CUDA 2.2, which can guarantee all writes to global memory visible to other threads, so correctness of reads after the barrier function can be guaranteed. In addition, we integrate each of our GPU synchronization approaches in a micro-benchmark and three well-known algorithms: FFT, dynamic programming, and bitonic sort. Finally, we use multiple threads in a block to check all the *arrival* variables, which can be executed in parallel, thus achieving a good performance.

IV. A Model for Kernel Execution Time and Speedup

In general, a kernel's execution time on GPUs consists of three components — *kernel launch time*, *computation time*,

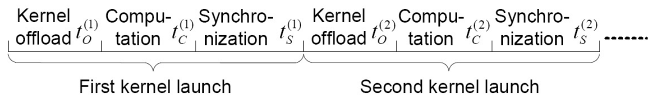


Fig. 1. Total Kernel Execution Time Composition

```
for() {
    __kernel_func<<<grid, block>>>();
    cudaThreadSynchronize();
}
```

(a) CPU explicit synchronization

```
for() {
    __kernel_func<<<grid, block>>>();
    //without cudaThreadSynchronize()
}
```

(b) CPU implicit synchronization

Fig. 2. CPU Explicit/Implicit Synchronization Function Call

```
__global__ void __kernel_func1()
{
    for () {
        __device_func();
        __gpu_sync();
    }
}
```

Fig. 3. GPU Synchronization Function Call

and *synchronization time*, which can be represented as

$$T = \sum_{i=1}^{M} \left(t_O^{(i)} + t_C^{(i)} + t_S^{(i)} \right) \tag{1}$$

where M is the number of kernel launches, $t_O^{(i)}$ is the kernel launch time, $t_C^{(i)}$ is the computation time, and $t_S^{(i)}$ is the synchronization time for the i^{th} kernel launch as shown in Figure 1. Each of the three time components is impacted by a few factors. For instance, the kernel launch time depends on the data transfer rate from the host to the device as well as the size of kernel code and parameters. For the computation time, it is affected by memory access methods, thread organization (number of threads per block and number of blocks per grid) in the kernel, etc. Similarly, the synchronization time will be different with different synchronization approaches used.

Figure 2 shows the pseudo-code of implementing barrier synchronization via kernel launches, where Figure 2(a) is the function call of *CPU Explicit Synchronization* and Figure 2(b) is for *CPU Implicit Synchronization*. As we can see, in the CPU explicit synchronization, the kernel function `__kernel_func()` is followed by the function `cudaThreadSynchronize()`, which will *not* return until all prior operations on the device are completed. As a result, the three operations — kernel launch, computation, and synchronization are executed sequentially in the CPU explicit synchronization. In contrast, in the CPU implicit synchronization, `cudaThreadSynchronize()` is *not* called. Since kernel launch is an asynchronous operation, if there are multiple kernel launches, kernel launch time can be overlapped by previous kernels' computation time and synchronization time. So, in the CPU implicit synchronization approach, except for the first kernel launch, subsequent ones are pipelined with computation and synchronization of previous kernel's execution, and the execution time of multiple kernel launches can be represented as

$$T = t_O^{(1)} + \sum_{i=1}^{M} \left(t_C^{(i)} + t_{CIS}^{(i)} \right) \tag{2}$$

where, M is the number of kernel launches, $t_O^{(1)}$ is the time for the first kernel launch, $t_C^{(i)}$ and $t_{CIS}^{(i)}$ are the computation time and synchronization time for the i^{th} kernel launch, respectively.

With respect to the *GPU Synchronization*, Figure 3 shows the pseudo-code of how functions are called. In this approach, a kernel is launched only once. When barrier synchronization is needed, a barrier function `__gpu_sync()` is called instead of re-launching the kernel. In Figure 3, the function `__device_func()` implements the same functionality as the kernel function `__kernel_func()` in Figure 2, but it is a device function instead of a global one, so it is called on the device rather than on the host. In the GPU synchronization, kernel execution time can be expressed as

$$T = t_O + \sum_{i=1}^{M} \left(t_C^{(i)} + t_{GS}^{(i)} \right) \tag{3}$$

where, M is the number of barriers needed for the kernel's execution, t_O is the kernel launch time, $t_C^{(i)}$ and $t_{GS}^{(i)}$ are the computation time and synchronization time for the i^{th} loop, respectively.

From Equations (1), (2), and (3), an algorithm can be accelerated by decreasing any of the three time components. With the properties of kernel launch time considered[2], we ignore the kernel launch time in the following discussion. If the synchronization time is reduced, according to the Amdahl's Law, the maximum kernel execution speedup is constrained by

$$\begin{aligned} S_T &= \frac{T}{t_C + (T - t_C)/S_S} \\ &= \frac{1}{\left(\frac{t_C}{T}\right) + \left(1 - \frac{t_C}{T}\right)/S_S} \\ &= \frac{1}{\rho + (1-\rho)/S_S} \end{aligned} \tag{4}$$

where S_T is the kernel execution speedup gained with reducing the synchronization time, $\rho = \frac{t_C}{T}$ is the percentage of the computation time t_C in the total kernel execution time T, $t_S = T - t_C$ is the synchronization time of the CPU implicit

[2]Three properties are considered. First, kernel launch time can be combined with the synchronization time in the CPU explicit synchronization; Second, it can be overlapped in CPU implicit synchronization; Third, kernel is launched only once in the GPU synchronization.

synchronization, which is our baseline as mentioned later. S_S is the synchronization speedup. Similarly, if only computation is accelerated, the maximum overall speedup is constrained by

$$S_T = \frac{1}{\rho/S_C + (1-\rho)} \tag{5}$$

where S_C is the computation speedup.

In Equation (4), the smaller the ρ is, the more speedup can be gained with a fixed S_S; while in Equation (5), the larger the ρ is, the more speedup can be obtained with a fixed S_C. In practice, different algorithms have different ρ values. For example, for the three algorithms used in this paper, FFT has a ρ value larger than 0.8, while SWat and bitonic sort have a ρ of about 0.5 and 0.4, respectively. According to Equation (5), corresponding to these ρ values, if only the computation is accelerated, maximum speedup of the three aforementioned algorithms are shown in Table II. As can be observed, very low speedup can be obtained in these three algorithms if only the computation is accelerated. Since most of the previous work focuses on optimizing the computation, i.e., decreases the computation time t_C, the more optimization is performed on an algorithm, the smaller ρ will become. At this time, decreasing the computation time will not help much for the overall performance. On the other side, if we decrease the synchronization time, large kernel execution speedup can be obtained.

TABLE II
POSSIBLE MAXIMUM SPEEDUP WITH ONLY COMPUTATION ACCELERATED

Algorithms	FFT	SWat	Bitonic sort
ρ	0.82	0.51	0.40
Possible maximum speedup	5.61	2.03	1.68

In this paper, we will focus on decreasing the synchronization time. This is due to three facts:

1) There has been a lot of work [6], [10], [15], [19], [25] proposed to decrease the computation time. Techniques such as shared memory usage and divergent branch removing have been widely used.
2) No work has been done to decrease the synchronization time for algorithms to be executed on a GPU;
3) In some algorithms, the synchronization time consumes a large part of the kernel execution time (e.g., SWat and bitonic sort in Figure 12), which results in a small ρ value.

With the above model for speedup brought by synchronization time reduction, we propose two GPU synchronization approaches in the next section, and time consumption of each of them is modeled and analyzed quantitatively.

V. PROPOSED GPU SYNCHRONIZATION

Since in CUDA programming model, the execution of a thread block is non-preemptive, care must be taken to avoid deadlocks in GPU synchronization design. Consider a scenario where multiple thread blocks are mapped to one SM and the active block is waiting for the completion of a global barrier. A deadlock will occur in this case because unscheduled thread blocks will not be able to reach the barrier without preemption. Our solution to this problem is to have a one-to-one mapping between thread blocks and SMs. In other words, for a GPU with 'Y' SMs, we ensure that at most 'Y' blocks are used in the kernel. In addition, we allocate all available shared memory on an SM to each block so that no two blocks can be scheduled to the same SM because of the memory constraint.

In the following discussion, we will present two alternative GPU synchronization designs: *GPU lock-based synchronization* and *GPU lock-free synchronization*. The first one uses a mutex variable and CUDA atomic operations; while the second method uses a lock-free algorithm that avoids the use of expensive CUDA atomic operations.

A. GPU Lock-Based Synchronization

The basic idea of GPU lock-based synchronization [31] is to use a global mutex variable to count the number of thread blocks that reach the synchronization point. As shown in Figure 4, in the barrier function `__gpu_sync()`, after a block completes its computation, one of its threads (we call it the *leading thread*.) will atomically add 1 to `g_mutex`. The leading thread will then repeatedly compare `g_mutex` to a target value `goalVal`. If `g_mutex` is equal to `goalVal`, the synchronization is completed and each thread block can proceed with its next stage of computation. In our design, `goalVal` is set to the number of blocks N in the kernel when the barrier function is first called. The value of `goalVal` is then incremented by N each time when the barrier function is successively called. This design is more efficient than keeping `goalVal` constant and resetting `g_mutex` after each barrier because the former saves the number of instructions and avoids conditional branching.

```
//the mutex variable
__device__ volatile int g_mutex;

//GPU lock-based synchronization function
__device__ void __gpu_sync(int goalVal)
{
    //thread ID in a block
    int tid_in_block = threadIdx.x * blockDim.y
                     + threadIdx.y;

    // only thread 0 is used for synchronization
    if (tid_in_block == 0)   {
        atomicAdd((int *)&g_mutex, 1);

        //only when all blocks add 1 to g_mutex
        //will g_mutex equal to goalVal
        while(g_mutex != goalVal)   {
            //Do nothing here
        }
    }
    __syncthreads();
}
```

Fig. 4. Code snapshot of the GPU Lock-Based Synchronization

In the GPU lock-based synchronization, the execution time of the barrier function `__gpu_sync()` consists of three

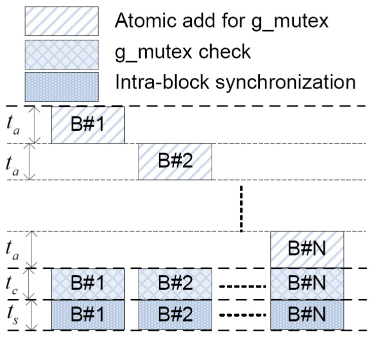


Fig. 5. Time Composition of GPU Lock-Based Synchronization

parts — atomic addition, checking of `g_mutex`, and synchronization of threads within a block via `__syncthreads()`. The atomic addition can only be executed sequentially by different blocks, while the `g_mutex` checking and intra-block synchronization can be executed in parallel. Assume there are N blocks in the kernel, the intra-block synchronization time is t_s, time of each atomic addition and `g_mutex` checking is t_a and t_c, respectively, if all blocks finish their computation at the same time as shown in Figure 5, then the time to execute `__gpu_sync()` is

$$t_{GBS} = N \cdot t_a + t_c + t_s \tag{6}$$

where N is the number of blocks in the kernel. From Equation (6), the cost of GPU lock-based synchronization increases linearly with N.

B. GPU Lock-Free Synchronization

In the GPU lock-based synchronization, the mutex variable `g_mutex` is added with the atomic function `atomicAdd()`. This means the addition of `g_mutex` can only be executed sequentially even though these operations are performed in different blocks. In this section, we propose a lock-free synchronization approach that avoids the use of atomic operations completely. The basic idea of this approach is to assign a synchronization variable to each thread block, so that each block can record its synchronization status independently without competing for a single global mutex variable.

As shown in Figure 6, our lock-free synchronization approach uses two arrays `Arrayin` and `Arrayout` to coordinate the synchronization requests from various blocks. In these two arrays, each element is mapped to a thread block in the kernel, i.e., element i is mapped to thread block i. The algorithm is outlined into three steps as follows:

1) When block i is ready for communication, its leading thread (thread 0) sets element i in `Arrayin` to the goal value `goalVal`. The leading thread in block i then busy-waits on element i of `Arrayout` to be set to `goalVal`.
2) The first N threads in block 1 repeatedly check if all elements in `Arrayin` are equal to `goalVal`, with thread i checking the i^{th} element in `Arrayin`. After all elements in `Arrayin` are set to `goalVal`,

```
//GPU lock-free synchronization function
__device__ void __gpu_sync(int goalVal,
  volatile int *Arrayin, volatile int *Arrayout)
{
    // thread ID in a block
    int tid_in_blk = threadIdx.x * blockDim.y
                   + threadIdx.y;
    int nBlockNum = gridDim.x * gridDim.y;
    int bid =  blockIdx.x * gridDim.y + blockIdx.y;

    // only thread 0 is used for synchronization
    if (tid_in_blk == 0)    {
        Arrayin[bid] = goalVal;
    }

    if (bid == 1)    {
        if (tid_in_blk < nBlockNum)    {
            while (Arrayin[tid_in_blk] != goalVal){
                //Do nothing here
            }
        }
        __syncthreads();

        if (tid_in_blk < nBlockNum)    {
            Arrayout[tid_in_blk] = goalVal;
        }
    }

    if (tid_in_blk == 0)    {
        while (Arrayout[bid] != goalVal)    {
            //Do nothing here
        }
    }
    __syncthreads();
}
```

Fig. 6. Code snapshot of the GPU Lock-Free Synchronization

each checking thread then sets the corresponding element in `Arrayout` to `goalVal`. Note that the intra-block barrier function `__syncthreads()` is called by each checking thread before updating elements of `Arrayout`.
3) A block will continue its execution once its leading thread sees the corresponding element in `Arrayout` is set to `goalVal`.

It is worth noting that in the step 2) above, rather than having one thread to check all elements of `Arrayin` *in serial* as in [29], we use N threads to check the elements of `Arrayin` *in parallel*. This design choice turns out to save considerable synchronization overhead according to our performance profiling. Note also that `goalVal` is incremented each time when the function `__gpu_sync()` is called, similar to the implementation of the GPU lock-based synchronization. Finally, this approach can be implemented in the Brook+ programming model of AMD/ATI GPUs in the same way, where an intra-block synchronization function `syncGroup()` is provided.

From Figure 6, there is no atomic operation in the GPU lock-free synchronization. All the operations can be executed in parallel. Synchronization of different thread blocks is controlled by threads in a single block, which can be synchronized efficiently by calling the barrier function `__syncthreads()`. From Figure 7, the execution time of `__gpu_sync()` is composed of six parts and calculated as

$$t_{GFS} = t_{SI} + t_{CI} + 2t_s + t_{SO} + t_{CO} \tag{7}$$

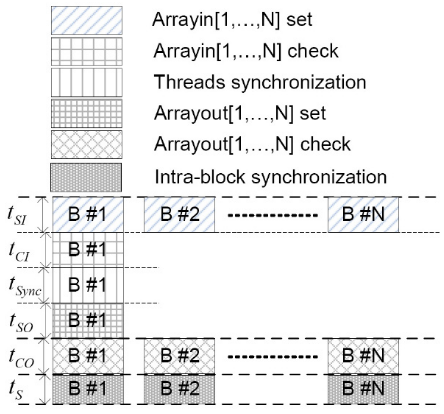


Fig. 7. Time Composition of GPU Lock-Free Synchronization

where, t_{SI} is the time for setting an element in `Arrayin`, t_{CI} is the time to check an element in `Arrayin`, t_s is the intra-block synchronization time, t_{SO} and t_{CO} are the time for setting and checking an element in `Arrayout`, respectively. From Equation (7), execution time of `__gpu_sync()` is unrelated to the number of blocks in a kernel[3].

C. Synchronization Time Verification via a Micro-benchmark

To verify the execution time of the synchronization function `__gpu_sync()` for each synchronization method, a micro-benchmark to compute the mean of two floats for 10,000 times is used. In other words, in the CPU synchronization, each kernel calculates the mean once and the kernel is launched 10,000 times; in the GPU synchronization, there is a 10,000-round `for` loop used, and the GPU barrier function is called in each loop. With each synchronization method, their execution time is shown in Figure 8. In the micro-benchmark, each thread will compute one element, the more blocks and threads are set, the more elements are computed, i.e., computation is performed in a weak-scale way. So the computation time should be approximately constant. Here, each result is the average of three runs.

From Figure 8, we have the following observations: 1) The CPU explicit synchronization takes much more time than the CPU implicit synchronization. This is due to, in the CPU implicit synchronization, kernel launch time is overlapped for all kernel launches except the first one; but in the CPU explicit synchronization, kernel launch time is not. 2) Even for the CPU implicit synchronization, a lot of synchronization time is needed. From Figure 8, the computation time is only about 5ms, while the time needed by the CPU implicit synchronization is about 60ms, which is 12 times the computation time. 3) For the GPU lock-based synchronization, the synchronization time is linear to the number of blocks in a kernel, and more synchronization time is needed for a kernel with a larger number of blocks, which matches very well to Equation (6) in Section V-A. Compared to the CPU implicit synchronization, when the block number is less than 24, its synchronization time is less; otherwise, more time is needed for the GPU lock-based synchronization. The reason is that, as we analyzed in Section V-A, more blocks means more atomic add operations should be executed for the synchronization. 4) For the GPU lock-free synchronization, since there are no atomic operations used, all the operations can be executed in parallel, which makes its synchronization time unrelated to the number of blocks in a kernel, i.e., the synchronization time is almost a constant value. Furthermore, the synchronization time is much less (for more than 3 blocks set in the kernel) than that of all other synchronization methods.

[3]Since there are at most 30 blocks that can be set on a GTX 280, threads that check `Arrayin` are in the same warp, which are executed in parallel. If there are more than 32 blocks in the kernel, more than 32 threads are needed for checking `Arrayin` and different warps should be executed serially on an SM.

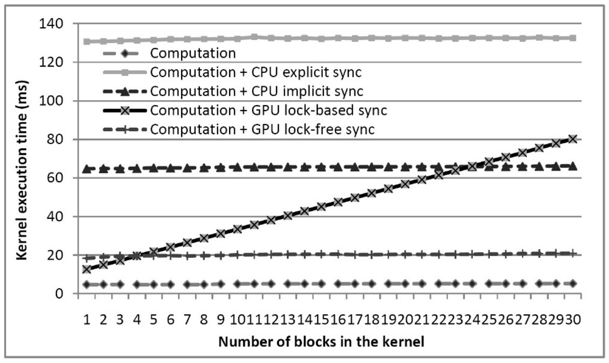


Fig. 8. Execution Time of the Micro-benchmark.

From the micro-benchmark results, the CPU explicit synchronization needs the most synchronization time, and in practice, there is no need to use this method. So in the following sections, we will not use it any more, i.e., only the CPU implicit and two GPU synchronization approaches are compared and analyzed.

VI. Algorithms Used for Performance Evaluation

Inter-block synchronization can be used in many algorithms. In this section, we choose three of them that can benefit from our proposed GPU synchronization methods. The three algorithms are Fast Fourier Transformation [16], Smith-Waterman [25], and bitonic sort [4]. In the following, a brief description is given for each of them.

A. Fast Fourier Transformation

A Discrete Fourier Transformation (DFT) transforms a sequence of values into its frequency components or, inversely, converts the frequency components back to the original data sequence. For a data sequence $x_0, x_1, \cdots, x_{N-1}$, the DFT is computed as $X_k = \sum_{i=0}^{N-1} x_i e^{-j2\pi k \frac{i}{n}}$, $k = 0, 1, 2, \cdots, N-1$, and the inverse DFT is computed as $x_i = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{j2\pi i \frac{k}{n}}$, $i = 0, 1, 2, \cdots, N-1$. DFT is used in many fields, but direct DFT computation is too slow to be used in practice. Fast Fourier Transformation (FFT) is a fast way of DFT computation. Generally, computing DFT directly by the definition takes $O\left(N^2\right)$ arithmetical operations, while FFT takes only $O\left(N\log\left(N\right)\right)$ arithmetical operations. The computation difference can be substantial for long data sequence, especially when the sequence has thousands or millions of points. A detailed description of the FFT algorithm can be found in [16].

For an N-point input sequence, FFT is computed in $\log\left(N\right)$ iterations. Within each iteration, computation of different points is independent, which can be done in parallel, because they depend on points only from its previous iteration. On the other hand, computation of an iteration cannot start until that of its previous iteration completes, which makes a barrier necessary across the computation of different iterations [6]. The barrier used here can be multiple kernel launches (CPU synchronization) or the GPU synchronization approaches proposed in this paper.

B. Dynamic Programming: Smith-Waterman Algorithm

Smith-Waterman (SWat) is a well-known algorithm for local sequence alignment. It finds the maximum alignment score between two nucleotide or protein sequences based on the Dynamic Programming paradigm [28], in which the segments of all possible lengths are compared to optimize the alignment score. In this process, first, intermediate alignment scores are stored in a DP matrix M before the matrix is inspected, and then, the local alignment corresponding to the highest alignment score is generated. As a result, the SWat algorithm can be broadly classified into two phases: (1) matrix filling and (2) trace back.

In the matrix filling process, a scoring matrix and a gap-penalty scheme are used to control the alignment score calculation. The scoring matrix is a 2-dimensional matrix storing the alignment score of individual amino acid or nucleotide residues. The gap-penalty scheme provides an option for gaps to be introduced in the alignment to obtain a better alignment result and it will cause some penalty to the alignment score. In our implementation of SWat, the *affine gap* penalty is used in the alignment, which consists of two penalties — the *open-gap* penalty, o, for starting a new gap and the *extension-gap* penalty, e, for extending an existing gap. Generally, an open-gap penalty is larger than an extension-gap penalty in the affine gap.

With the above scoring scheme, the DP matrix M is filled in a *wavefront* pattern, i.e. the matrix filling starts from the northwest corner element and goes toward the southeast corner element. Only after the previous anti-diagonals are computed can the current one be calculated as shown in Figure 9. The calculation of each element depends on its northwest, west, and north neighbors. As a result, elements in the same anti-diagonal are independent of each other and can be calculated in parallel; while barriers are needed across the computation of different anti-diagonals. For the trace back, it is essentially a sequential process that generates the local alignment with the highest score. In this paper, we only consider accelerating the matrix filling because it occupies more than 99% of the execution time.

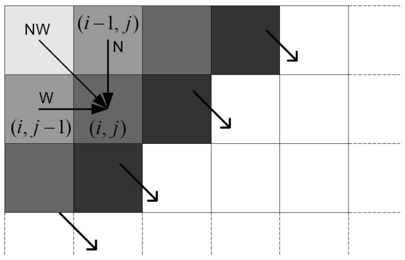


Fig. 9. Wavefront Pattern and Dependency in the Matrix Filling Process.

C. Bitonic Sort

Bitonic sort is one of the fastest sorting networks [13], which is a special type of sorting algorithm devised by Ken Batcher [4]. For N numbers to be sorted, the resulting network consists of $O\left(n\log^2\left(n\right)\right)$ comparators and has a delay of $O\left(\log^2\left(n\right)\right)$.

The main idea behind bitonic sort is using a divide-and-conquer strategy. In the divide step, the input sequence is divided into two subsequences and each sequence is sorted with bitonic sort itself, where one is in the ascending order and the other is in the descending order. In the conquer step, with the two sorted subsequences as the input, the bitonic merge is used to combine them to get the whole sorted sequence [13]. The main property of bitonic sort is, no matter what the input data are, a given network configuration will sort the input data in a fixed number of iterations. In each iteration, the numbers to be sorted are divided into pairs and a compare-and-swap operation is applied on it, which can be executed in parallel for different pairs. More detailed information about bitonic sort is in [4]. In bitonic sort, the independence within an iteration makes it suitable to be executed in parallel and the data dependency across adjacent iterations makes it necessary for a barrier to be used.

VII. Experiment Results and Analysis

A. Overview

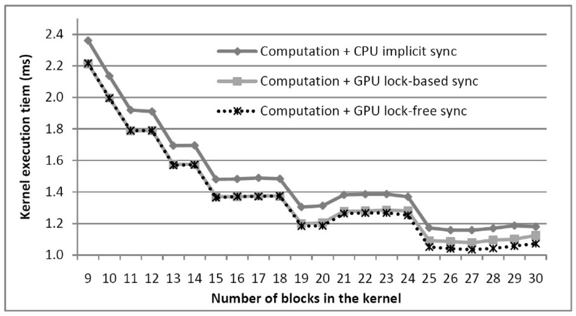
To evaluate the performance of our proposed GPU synchronization approaches, we implement them in the three algorithms described in Section VI. For the two CPU synchronization approaches, we only implement the CPU implicit synchronization because its performance is much better than the CPU explicit synchronization. With implementations using each of the synchronization approaches for each algorithm, their performance is evaluated in four aspects: 1) Kernel execution time decrease brought by our proposed GPU synchronization approaches and its variation against the number of blocks in the kernel; 2) According to the kernel execution time partition model in Section IV, we calculate the *synchronization time* of each synchronization approach. Similarly, the synchronization time variation against the number of blocks in kernels is presented; 3) Corresponding to the best performance of each algorithm with each synchronization approach, the percentages of computation time and synchronization time are demonstrated and analyzed; 4) The costs of guaranteeing inter-block communication correctness via `__threadfence()` on GPUs are shown.

Our experiments are performed on a GeForce GTX 280 GPU card, which has 30 SMs and 240 processing cores with the clock speed 1296MHz. The on-chip memory on each SM contains 16K registers and 16KB shared memory, and there are 1GB GDDR3 global memory with the bandwidth of 141.7GB/Second on the GPU card. For the host machine, The processor is an Intel Core 2 Duo CPU with 2MB of L2 cache and its clock speed is 2.2GHz. There are two 2GB of DDR2 SDRAM equipped on the machine. The operating system on the host machine is the 64-bit Ubuntu GNU/Linux distribution. The NVIDIA CUDA 2.2 SDK toolkit is used for all the program execution. Similar as that in the micro-benchmark, each result is the average of three runs.

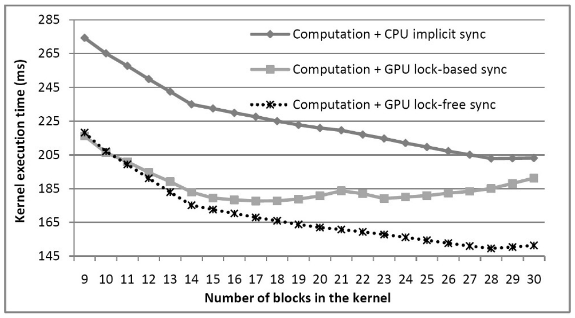
B. Kernel Execution Time

Figure 10 shows the kernel execution time decrease with our proposed GPU synchronization approaches and its variation versus the number of blocks in the kernel. Here, we demonstrate the kernel execution time with the block number from 9 to 30. This is due to, when the number of blocks in the kernel is larger than 30 or less than 9, kernel execution times are more than that with block number between 9 and 30. Also, if a GPU synchronization approach is used, the maximum number of blocks in a kernel is 30. In our experiments, the number of threads per block is 448, 256, and 512 for FFT, SWat, and bitonic sort, respectively. Figure 10(a) shows the performance of FFT, Figure 10(b) is for SWat, and Figure 10(c) displays the kernel execution time of bitonic sort.

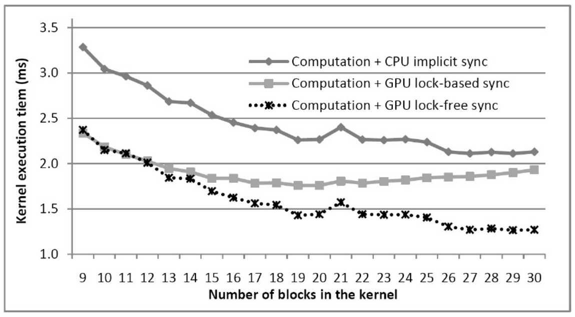
From Figure 10, we can see that, first, with the increase of the number of blocks in the kernel, kernel execution time will decrease. The reason is, with more blocks (from 9 to 30) in the kernel, more resources can be used for the computation, which will accelerate the computation; Second, with the proposed GPU synchronization approaches used, performance improvements are observed in all the three algorithms. For example, compared to the CPU implicit synchronization, with the GPU lock-free synchronization and 30 blocks in the kernel, kernel execution time of FFT decreases from 1.179ms to 1.072ms, which is an 9.08% decrease. For SWat and bitonic sort, this value is 25.47% and 40.39%, respectively. Table III shows the speedup increase corresponding to the performance improvement by comparing to a sequential implementation of each algorithm. As we can see, speedup of FFT increases from 62.50× with the CPU implicit synchronization to 69.93× with the GPU lock-free synchronization. Similarly, speedup of SWat and bitonic sort increases from 9.53× to 12.93× and from 14.40× to 24.02×, respectively. Third, kernel execution time difference between the CPU implicit synchronization and the proposed GPU synchronization of FFT is much less than that of SWat and bitonic sort. This is due to, in FFT, the computation load between two barriers is much more than that of SWat and bitonic sort. According to Equation (4), kernel execution time change caused by the synchronization time decrease in FFT is not as much as that in SWat and bitonic sort.



(a) FFT



(b) SWat



(c) Bitonic sort

Fig. 10. Kernel Execution Time versus Number of Blocks in the Kernel

TABLE III
ADDITIONAL SPEEDUP OBTAINED BY BETTER SYNCHRONIZATION APPROACHES

Algorithms	FFT	SWat	Bitonic sort
Speedup with **CPU implicit** synchronization	62.50	9.53	14.40
Speedup with **GPU lock-based** synchronization	67.14	10.89	17.27
Speedup with **GPU lock-free** synchronization	69.93	12.93	24.02

In addition, among the two implementations with our proposed GPU synchronization approaches, 1) With more blocks set in the kernel, kernel execution time decrease rate of the GPU lock-based synchronization is not as fast as the GPU lock-free synchronization. This is compatible with Equation (6), i.e., as more blocks are configured, more time is needed for the GPU lock-based synchronization. 2) In the three algorithms, performance with the GPU lock-free synchronization is always the best. The more blocks are set in the kernel, the more performance improvement can be obtained if compared to the GPU lock-based synchronization approach. The reason is the time needed for the GPU lock-free synchronization is almost a constant value, but synchronization time will increase in the GPU lock-based synchronization when more blocks are set in the kernel.

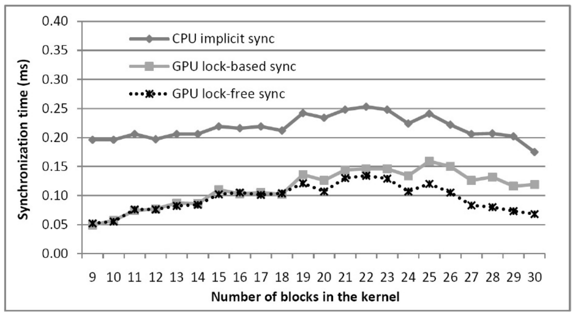
C. Synchronization Time

In this section, we show the synchronization time variation versus the number of blocks in the kernel. Here, the synchronization time is the difference between the total kernel execution time and the computation time, which is obtained by running an implementation of each algorithm with the GPU synchronization approach, but with the synchronization function `__gpu_sync()` removed. For the implementation with the CPU synchronization, we assume its computation time is the same as others because the memory access and the computation is the same as that of the GPU implementations. With the above method, time of each synchronization method in the three algorithms is shown in Figure 11. Similar as Figure 10, we show the number of blocks in the kernel from 9 to 30. Figures 11(a), 11(b), and 11(c) are for FFT, SWat, and bitonic sort, respectively.

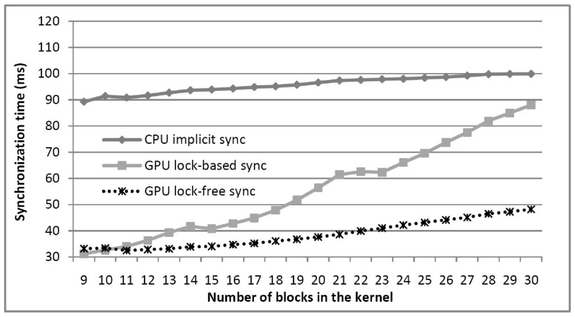
From Figure 11, in SWat and bitonic sort, synchronization time matches the time consumption models as expressed in Equations (6) and (7) in Section V. First, the CPU implicit synchronization approach needs the most time while the GPU lock-free synchronization consumes the least. Second, the CPU implicit and the GPU lock-free synchronization has good scalability, i.e., the synchronization time changes very little with the change of the number of blocks in the kernel. Third, for the GPU lock-based synchronization approach, the synchronization time increases with the increase of the number of blocks in the kernel. With 9 blocks in the kernel, time needed for the GPU lock-based synchronization is close to that of the GPU lock-free synchronization; When the number of blocks increases to 30, synchronization time becomes much larger than the GPU lock-free synchronization, but it is still less than that of the CPU implicit synchronization. For FFT, though the synchronization time variation is not regular versus the number of blocks in the kernel, differences across different synchronization approaches are the same as that in the other two algorithms, i.e., as more blocks are configured in kernel, more time is needed for the GPU lock-based synchronization than the GPU lock-free synchronization. The reason for the irregularity is caused by the property of the FFT computation, which needs more investigation in the future.

D. Percentages of the Computation Time and the Synchronization Time

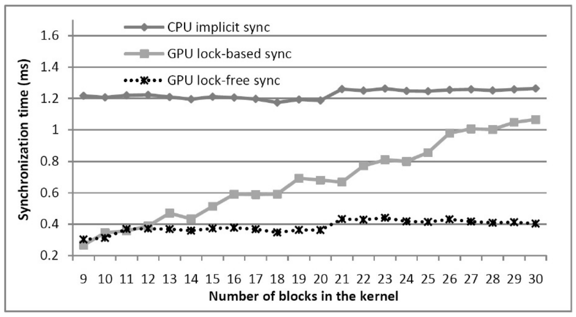
Figure 12 shows the performance breakdown in percentage of the three algorithms when different synchronization approaches are used. As we can see, percentage of the synchronization time in FFT is much less than that in SWat and bitonic sort. As a result, synchronization time changes have a less impact on the total kernel execution time compared to SWat and bitonic sort. This is compatible with what are shown in Figure 10, in which, for FFT, kernel execution time is very close with different synchronization approaches used; while the kernel execution time changes a lot in SWat and bitonic sort; In addition, with the CPU implicit synchronization approach used, synchronization time percentages are about 50% and 60% in SWat and bitonic sort, respectively. This indicates that inter-block communication time occupies a large part of the total execution time in some algorithms. Thus, decreasing the synchronization time can improve the performance greatly in some algorithms; Finally, with the GPU lock-free synchronization approach, percentage of the synchronization time decreases from 49.2% to 31.1% in SWat and from 59.6% to 32.7% in bitonic sort, respectively, but that of FFT is much less, from 17.8% to 8.0%. The reason is similar, i.e., synchronization time decrease does not impact the total kernel execution time as much as the other two algorithms because its percentage in the total kernel execution time is



(a) FFT



(b) SWat



(c) Bitonic sort

Fig. 11. Synchronization Time versus Number of Blocks in the Kernel

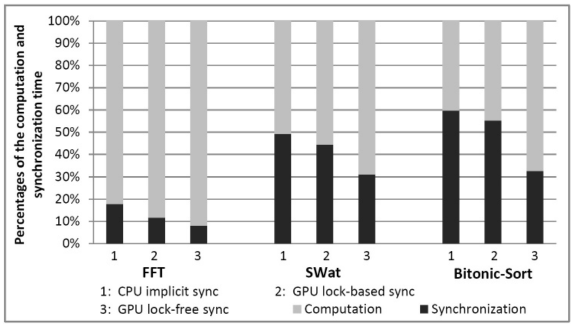


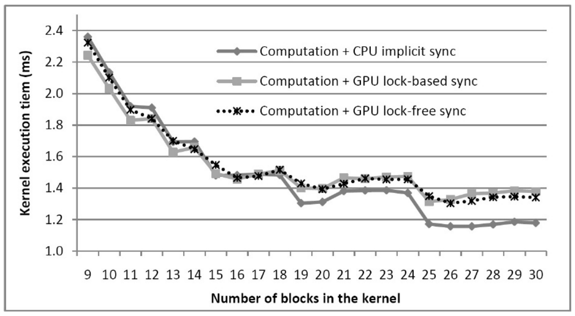
Fig. 12. Percentages of Computation Time and Synchronization Time

small.

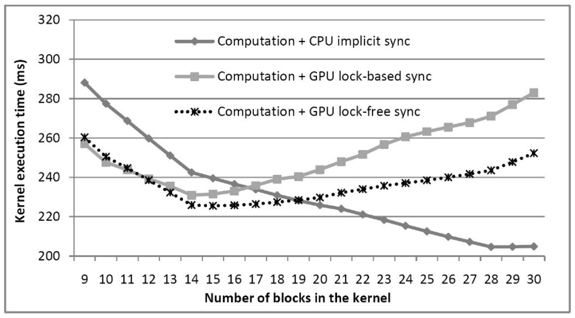
E. Costs of Guaranteeing Inter-Block Communication Correctness

As described in [29], the barrier function cannot guarantee that inter-block communication is correct unless a memory consistency model is assumed. To remedy this problem, CUDA 2.2 provides a new function `__threadfence()`. This function can "guarantee all writes to shared or global memory visible to other threads [22]". If it is integrated in our proposed GPU barrier synchronization functions, then all writes to shared memory or global memory will be read correctly after the barrier synchronization function. However, as we can expect, overhead will be caused when `__threadfence()` is called. Figure 13 shows the kernel execution time versus the number of blocks in kernels with `__threadfence()` called in our barrier synchronization function `__gpu_sync()`, where Figures 13(a), 13(b), and 13(c) are for FFT, SWat, and bitonic sort, respectively.

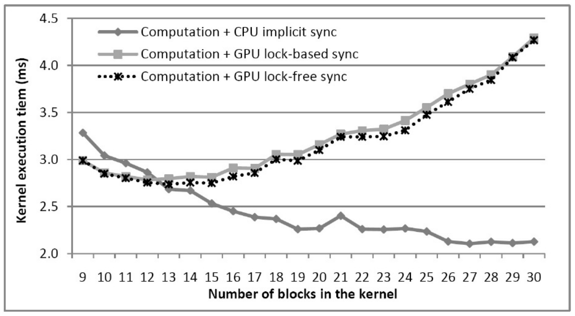
As we can see, a lot of overhead is caused by `__threadfence()`. The more block are configured in kernels, the more overhead is caused, which can even exceed the kernel execution time with the CPU implicit synchronization. Consider the GPU lock-free synchronization, from Figure 13(a), for FFT, when the number of blocks in the kernel is larger than 14, more time is needed to execute the kernel with the GPU lock-free synchronization. The threshold is 18 and 12 for SWat and bitonic sort, respectively. From these results, though the barrier can be implemented in software efficiently, the cost of guaranteeing correctness with the function `__threadfence()` is very high, which means guaranteeing writes to shared memory or global memory to be read correctly via `__threadfence()` is not an efficient way. It is worth noting that even without `__threadfence()` called in our barrier functions, all program results are correct with thousands of runs. Thus, the likelyhood of incorrect inter-block data communication is effectively 0, which arguably obfuscates the needed for `__threadfence()` on the GTX 280. However, this is not expected on the next generation of NVIDIA GPU "Fermi", on which, with a more efficient implementation of `__threadfence()` and a different architecture, it is needed for correct inter-block data communication.



(a) FFT



(b) SWat



(c) Bitonic sort

Fig. 13. Kernel Execution Time versus Number of Blocks in the Kernel with `__threadfence()` Called

VIII. Conclusion

In the current GPU architecture, inter-block communication on GPUs requires a barrier synchronization to exist. Till now, most previous GPU performance optimization studies focus on optimizing the computation, and very few techniques were proposed to reduce inter-block communication time, which is dominated by barrier synchronization time. To systematically solve this problem, we first propose a performance model for kernel execution on a GPU. It partitions kernel execution time into three components: kernel launch time, computation time, and synchronization time. This model can help to design and evaluate various synchronization approaches.

Second, we propose two synchronization approaches: GPU lock-based synchronization and GPU lock-free synchronization. The GPU lock-based synchronization uses a mutex variable and CUDA atomic operations, while the lock-free approach uses two arrays of synchronization variables and does not rely on the costly atomic operations. For each of these methods, we quantify its efficacy with the aforementioned performance model.

We evaluate the two GPU synchronization approaches with a micro-benchmark and three important algorithms. From our experiment results, with our proposed GPU synchronization approaches, performance improvements are obtained in all the algorithms compared to state of the art CPU barrier synchronization, and the time needed for each GPU synchronization approach matches the time consumption model well. In addition, based on the kernel execution time model, we partition the kernel execution time into the computation time and the synchronization time for the three algorithms. In SWat and bitonic sort, the synchronization time takes more than half of the total execution time. This demonstrates that for data-parallel algorithms with considerable inter-block communication, decreasing synchronization time is as important as optimizing computation. Finally, we show the performance degradation caused by `__threadfence()` to guarantee inter-block communication correctness. From the results, though barrier synchronization can be implemented via software efficiently, guaranteeing data writes to shared memory and global memory visible to all other threads via `__threadfence()` is inefficient. As a result, better approaches such as efficient hardware barrier implementation or memory flush functions are needed to support efficient and correct inter-block communication on a GPU.

As for future work, we will further investigate the reasons for the irregularity of the FFT's synchronization time versus the number of blocks in the kernel. Second, we will propose a general model to characterize algorithms' parallelism properties, based on which, better performance can be obtained for their parallelization on multi- and many-core architectures.

Acknowledgment

We would like to thank Heshan Lin, Jeremy Archuleta, Tom Scogland, and Song Huang for their technical support and feedback on the manuscript.

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] K. E. Batcher. Sorting Networks and their Applications. In *Proc. of AFIPS Joint Computer Conferences*, pages 307–314, April 1968.

[5] D. Cederman and P. Tsigas. On Dynamic Load Balancing on Graphics Processors. In *Proc. of the 23rd ACM SIGGRAPHEUROGRAPHICS Symp. on Graphics Hardware*, pages 57–64, June 2008.

[6] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proc. of Supercomputing*, pages 1–12, November 2008.

[7] A. Greb and G. Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In *IPDPS*, April 2006.

[8] 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.

[9] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proc. of the IEEE International Conference on High Performance Computing*, December 2007.

[10] E. Herruzo, G. Ruiz, J. I. Benavides, and O. Plata. A New Parallel Sorting Algorithm based on Odd-Even Mergesort. In *Proc. of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 18–22, 2007.

[11] I. Jung, J. Hyun, J. Lee, and J. Ma. Two-Phase Barrier: A Synchronization Primitive for Improving the Processor Utilization. *International Journal of Parallel Programming*, 29(6):607–627, 2001.

[12] G. J. Katz and J. T. Kider. All-Pairs Shortest-Paths for Large Graphs on the GPU. In *Proc. of the 23rd ACM SIGGRAPHEUROGRAPHICS Symp. on Graphics Hardware*, pages 47–55, June 2008.

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

[14] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-Sequence Database Scanning on a GPU. *IPDPS*, April 2006.

[15] Y. Liu, W. Huang, J. Johnson, and S. Vaidya. GPU Accelerated Smith-Waterman. In *Proc. of the 2006 International Conference on Computational Science, Lectures Notes in Computer Science Vol. 3994*, pages 188–195, June 2006.

[16] C. V. Loan. Computational Frameworks for the Fast Fourier Transform. In *Society for Industrial Mathematics*, 1992.

[17] B. D. Lubachevsky. Synchronization Barrier and Related Tools for Shared Memory Parallel Programming. *International Journal of Parallel Programming*, 19(3):225–250, 1990. 10.1007/BF01407956.

[18] S. A. Manavski and G. Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, 2008.

[19] Y. Munekawa, F. Ino, and K. Hagihara. Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU. In *Proc. of the 8th IEEE International Conference on BioInformatics and BioEngineering*, pages 1–6, October 2008.

[20] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth Intensive 3-D FFT Kernel for GPUs Using CUDA. In *Proc. of Supercomputing*, November 2008.

[21] NVIDIA. CUDA SDK 2.2.1, 2009. http://developer.download.nvidia.com/compute/cuda/2_21/toolkit/docs/CUDA_Getting_Started_2.2_Linux.pdf.

[22] 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.

[23] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W. W. Hwu. GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. In *Proc. of the Conference on Computing Frontiers*, pages 273–282, May 2008.

[24] 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, February 2008.

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

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

[27] J. A. Stuart and J. D. Owens. Message Passing on Data-Parallel Architectures. In *IPDPS*, May 2009.

[28] M. A. Trick. A Tutorial on Dynamic Programming. 1997. http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html.

[29] V. Volkov and J. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proc. of Supercomputing*, November 2008.

[30] 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.

[31] S. Xiao, A. Aji, and W. Feng. On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In *Proc. of the 15th International Conference on Parallel and Distributed Systems (ICPADS)*, December 2009.