Inter-Block GPU Communication via Fast Barrier Synchronization

Shucai Xiao and Wu-chun Feng

Department of Computer Science
Virginia Tech
{shucai, wfeng}@vt.edu

Abstract

The graphics processing unit (GPU) has evolved from a fixed-function processor with programmable stages to a programmable processor with many fixed-function components that deliver massive parallelism. Consequently, GPUs increasingly take advantage of the programmable processing power for general-purpose, non-graphics tasks, i.e., general-purpose computation on graphics processing units (GPGPU). However, while the GPU can massively accelerate data parallel (or task parallel) applications, the lack of explicit support for inter-block communication on the GPU hampers its broader adoption as a general-purpose computing device.

Inter-block communication on the GPU occurs via global memory and then requires a 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, incurs significant overhead. Thus, we seek to propose more efficient methods for inter-block communication. To systematically address this problem, we first present a performance model for the execution of kernels on GPUs. This performance model partitions the kernel's execution time into three phases: (1) kernel launch to the GPU, (2) computation on the GPU, and (3) inter-block GPU communication via barrier synchronization. Using three well-known algorithms — FFT, dynamic programming, and bitonic sort — we show that the latter phase, i.e., inter-block GPU communication, can consume more than 50% of the overall execution time.

Therefore, we propose three new approaches to inter-block GPU communication via barrier synchronization, all of which run only on the GPU: GPU simple synchronization, GPU tree-based synchronization, and GPU lock-free synchronization. We then evaluate the efficacy of each of these approaches in isolation via a micro-benchmark as well as integrated with the three aforementioned algorithms. For the micro-benchmark, the experimental results show that our GPU lock-free synchronization performs 7.8 times faster than CPU explicit synchronization and 3.7 times faster than CPU implicit synchronization. When integrated with the FFT, dynamic programming, and bitonic sort algorithms, our GPU lock-free synchronization improves the performance by 8%, 24%, and 39%, respectively, when compared to the more efficient CPU implicit synchronization.

Categories and Subject Descriptors D.1.3[Software] [*Programming Techniques*]: Concurrent Programming

General Terms Design, Performance, Model

Keywords Parallel computing, CUDA, kernel execution-time model, GPU synchronization

1. 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 GPU (GPGPU). Programming models such as NVIDIA's Compute Unified Device Architecture (CUDA) [22] and AMD/ATI's Brook+ [1] 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 4.2), inter-block communication via barrier synchronization can consume as much as 50% of the total kernel execution time, as shown in Table 1.

Hence, in contrast to previous work that mainly focuses on optimizing the GPU computation, we focus on 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.

Algorithms	FFT	SWat	Bitonic sort
% of time spent on inter-block communication	19.6%	49.7%	59.6%

Table 1. Percent of Time Spent on Inter-Block Communication

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

We propose two types of GPU synchronization, one with locks and one without. For the former, we use one or more mutual-exclusive (mutex) variables and an atomic add operation to implement *GPU simple synchronization* and *GPU tree-based synchronization*. For 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 inter-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.

Overall, the contributions of this paper are three-fold. First, we propose a set of 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 demonstrate performance improvements of 8%, 24%, and 39%, respectively, over the traditional CPU synchronization approach.

The rest of the paper is organized as follows. Section 2 provides an overview of the NVIDIA GTX 280 architecture and CUDA programming model. The related work is described in Section 3. Section 4 presents our performance model for kernel execution time. Section 5 describes our GPU synchronization approaches. In Section 6, we give a brief description of the algorithms that we use to evaluate our proposed GPU synchronization approaches, and Section 7 presents and analyzes the experimental results. Section 8 concludes the paper.

2. 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 (SPs). 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 a 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, i.e., threads executing within a SM, 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.

3. 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. Volkov et al. [30] try to accelerate the FFT by designing a hierarchical communication scheme that minimizes 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 by 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 used graphics primitives [15, 14] in a task parallel fashion. More recent work uses CUDA, but again, largely in a task parallel manner [18, 19, 26], e.g., running a single (constrained) problem instance per SM in order to eliminate the need for inter-block communication — with the tradeoff, of course, being that the size of the problem is limited by the size of shared memory, i.e., 16 KB. That is, because multiple problem instances share the GPU resources, the problem size is *severely* restricted.

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 [17, 8, 11, 2, 3], 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, the 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 synchronizations, i.e., GPU simple synchronization and GPU tree-based synchronization. These approaches leverage 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 methods 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. In contrast, our proposed GPU synchronization approaches guarantee the completion of memory accesses with the existing memory access model in CUDA. 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.

4. A Model for Kernel Execution Time and Speedup

In general, the kernel's execution time on GPUs consists of three components — *kernel launch time*, *computation time*, and *synchronization time*, which can be represented as

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

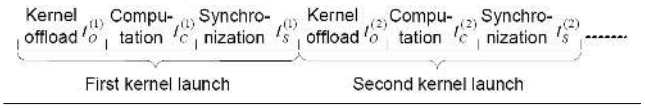


Figure 1. Total Kernel Execution Time Composition

where M is the number of kernel launches, $t_O^{(i)}$ is the kernel launch time, $t_C^{(i)}$ is the computation time on a GPU, 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, the thread organization (number of threads per block and number of blocks per grid) in the kernel, etc.

From Equation (1), an algorithm can be accelerated by decreasing any of the three time components. Since the kernel launch time is much smaller compared to the other two (as described in Sections 4.2 and 4.3), 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

$$\begin{aligned} S_T &= \frac{T}{t_C + (T - t_C)/S_S} \\ &= \frac{1}{\left(\frac{t_C}{T}\right) + \left(1 - \frac{t_C}{T}\right)/S_S} \\ &= \frac{1}{\rho + (1-\rho)/S_S} \end{aligned} \quad (2)$$

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 is the synchronization time of the CPU implicit synchronization, which is our baseline as mentioned later. S_S is the synchronization speedup. In Equation (2), the smaller the ρ is, the more speedup can be gained with the same S_S. In practice, different algorithms have different ρ. For example, for the three algorithms used in this paper, FFT has a ρ value larger than 0.8, while Swat and bitonic sort both have a ρ value about 0.5. Since most of the previous works were focusing on optimizing the computation, i.e., decreasing the computation time t_C. The more optimization is performed on an algorithm, the smaller ρ will become. At this time, if we decrease the synchronization time, large kernel execution speedup can be obtained.

In this paper, we will focus on decreasing the synchronization time. This is due to three facts: 1) There have been a lot of works [19, 25, 15, 6, 10] to decrease the computation time. Techniques such as shared memory usage, divergent branch removing have been widely used. 2) No work has been done to decrease the synchronization time for less-data-dependent 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 15), which results in a small ρ.

Currently, the available synchronization approach is the CPU synchronization approach, in which, depending on whether the function `cudaThreadSynchronize()` is called, there are two methods — *CPU explicit synchronization* and *CPU implicit synchronization*. In addition, our proposed GPU synchronization includes *GPU simple synchronization*, *GPU tree-based synchronization*, and *GPU lock-free synchronization*. In the following subsections, we present the kernel execution time model of the CPU explicit/implicit synchronization and the GPU synchronization. In the next section, the proposed GPU synchronization approaches will be

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

(a) CPU explicit synchronization

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

(b) CPU implicit synchronization

Figure 2. CPU Explicit/Implicit Synchronization Function Call

described in detail and their time consumption is modeled and analyzed quantitatively.

4.1 CPU Explicit Synchronization

Figure 2(a) shows the pseudo-code of the CPU explicit synchronization, in which `__kernel_func()` is the kernel function, and the *implicit* barrier is implemented by terminating the current kernel execution and launching the kernel again, then a barrier exists between the two kernel launches. In addition, in the CPU explicit synchronization, the function `cudaThreadSynchronize()` is used. According to the CUDA programming guide [22], "it will block until the device has completed all preceding requested task", and then a new kernel can be launched. So, in the CPU explicit synchronization, the three operations of a kernel's execution are executed sequentially across different kernel launches, and the time needed for multiple times of kernel execution is the same as that in Equation (1), i.e.

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

where, M is the number of kernel launches, $t_O^{(i)}$ is the kernel launch time, $t_C^{(i)}$ is the computation time, and $t_{CES}^{(i)}$ is the synchronization time (the kernel returning time is included.) for the i^{th} kernel launch, respectively. Since the function `cudaThreadSynchronize()` across multiple kernel launches is useless and causes overhead to the kernel's execution, in practice, the CPU explicit synchronization is not usually used.

4.2 CPU Implicit Synchronization

Compared to the CPU explicit synchronization, the function `cudaThreadSynchronize()` is not called in the CPU implicit synchronization as shown in Figure 2(b). As we know, kernel launch is an asynchronous operation, i.e., the kernel function `__kernel_func()` will return before its computation finishes, and if there are multiple kernel launches, subsequent kernel launches will be performed without waiting for the completion of their previous kernel's computation. So, in the CPU implicit synchronization, except for the first kernel launch, the following kernel launches are pipelined with the computation of their previous kernel launches. In the CPU implicit synchronization, the time composition of multiple times of kernel execution is shown in Figure 3 and the total kernel execution time can be represented as

$$T = t_O^{(1)} + \sum_{i=1}^{M} \left(t_C^{(i)} + t_{CIS}^{(i)} \right) \tag{4}$$

where, M is the number of kernel launch times, $t_O^{(1)}$ is the kernel launch 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,

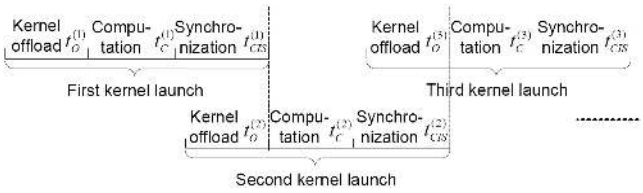


Figure 3. Kernel Execution Time Composition of CPU Implicit Synchronization

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

Figure 4. GPU Synchronization Function Call

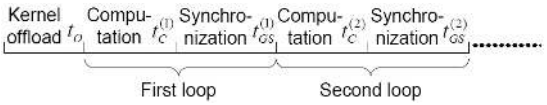


Figure 5. Kernel Execution Time Composition of GPU Synchronization

respectively. Since kernel launch time is pipelined for all kernel launches except the first one, less time is needed to execute the same kernel for the same number of times compared to the CPU explicit synchronization.

4.3 GPU Synchronization

In the GPU synchronization, synchronization across different blocks is implemented by calling a synchronization function `__gpu_sync()` (described in detail in Section 5) without terminating the kernel's execution and re-launching the kernel. Pseudo-code of the GPU synchronization is shown in Figure 4. Here, 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. The function `__gpu_sync()` is the *explicit* barrier for the inter-block communication. With the GPU barrier function `__gpu_sync()`, the kernel `__kernel_func1()` will be launched only once. As a result, it avoids executing the costly operation — kernel launch multiple times, and it is possible to save time for the kernel's execution compared to the CPU synchronization strategy. The time composition of the kernel execution is shown in Figure 5 and represented as

$$T = t_O + \sum_{i=1}^{M} \left(t_C^{(i)} + t_{GS}^{(i)} \right) \tag{5}$$

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.

Compared to the CPU implicit synchronization, if the time needed to execute the barrier function `__gpu_sync()` is less than the synchronization time of the CPU implicit synchronization, less time is needed to execute `__device_func()` for the same number of times compared to that with the CPU implicit synchronization approach.

5. Proposed GPU Synchronization

Since in CUDA programming model, the execution of a thread block is non-preemptive, care must be taken to avoid dead locks 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 the unscheduled thread blocks will not be able to reach the barrier without preemption. Our solution to this problem is to have an 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 a 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 three alternative GPU synchronization designs: *GPU simple synchronization, GPU tree-based synchronization, GPU lock-free synchronization*. The first two are lock-based designs that make use of mutex variables and CUDA atomic operations. The third design uses a lock-free algorithm that avoids the use of expensive CUDA atomic operations.

5.1 GPU Simple Synchronization

The basic idea of GPU simple synchronization is to use a global mutex variable to count the number of thread blocks that reach the synchronization point. As shown in Figure 6[2], 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.

```
1  //the mutex variable
2  __device__ int g_mutex;
3
4  //GPU simple synchronization function
5  __device__ void __gpu_sync(int goalVal)
6  {
7      //thread ID in a block
8      int tid_in_block = threadIdx.x * blockDim.y
9                         + threadIdx.y;
10
11     // only thread 0 is used for synchronization
12     if (tid_in_block == 0)    {
13         atomicAdd(&g_mutex, 1);
14
15         //only when all blocks add 1 to g_mutex will
16         //g_mutex equal to goalVal
17         while(g_mutex != goalVal)    {
18             ......
19         }
20     }
21     __syncthreads();
22 }
```

Figure 6. Code snapshot of the GPU Simple Synchronization (Some code details have been omitted to enhance readability.)

In the GPU simple synchronization, the execution time of the barrier function `__gpu_sync()` consists of two parts — atomic addition and checking of `g_mutex`. The atomic addition can only be executed sequentially by different blocks, but the `g_mutex` checking can be executed in parallel. Assume there are N blocks in the kernel, the time of each atomic addition is t_a, and the time of the `g_mutex` checking is t_c. If all blocks finish their computation at the same time as shown in Figure 7, the time to execute `__gpu_sync()` is

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

where N is the number of blocks in the kernel. From Equation (6), the cost of GPU simple synchronization increases linearly relative to N.

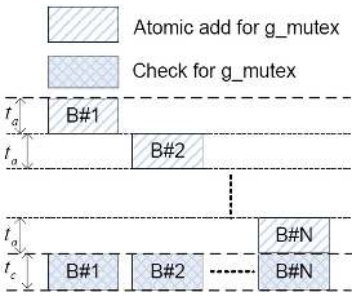


Figure 7. Time Composition of GPU Simple Synchronization

5.2 GPU Tree-Based Synchronization

One way to improve the performance of GPU simple synchronization is to increase the concurrency of updating the global mutex variable. For this purpose, we propose *GPU tree-based synchronization* approach. Figure 8 shows a 2-level tree-based GPU synchronization method. In this figure, thread blocks are divided into m groups with block number n_i for group i, $(i = 1, \cdots, m)$. For each group, a separate mutex variable `g_mutex_`i is assigned to synchronize its blocks with the GPU simple synchronization method. After blocks within a group are synchronized, another mutex variable `g_mutex` is used to synchronize all the blocks in the m groups.

Similar to the model we used for GPU simple synchronization, the time to execute the `__gpu_sync()` function can be represented as

$$t_{GTS} = (\hat{n} \cdot t_a + t_{c1}) + (m \cdot t_a + t_{c2}) \tag{7}$$

where $\hat{n} = \max_{i=1}^{m}(n_i)$ and m are the number of atomic add operations that are executed sequentially in the first and second levels, if all blocks finish their computation simultaneously. t_{c1} and t_{c2} are the time to check the mutex variables in the first and second levels, respectively. To obtain the least time to execute the synchronization function `__gpu_sync()`, the number of groups m is calculated as

$$m = \lceil \sqrt{N} \rceil \tag{8}$$

With the group number m, if m^2 equals N, then there are m blocks in each group, i.e., $n_i = m, (i = 1, \cdots, m)$; Otherwise, for the first $m-1$ groups, $n_i = \lfloor N/(m-1) \rfloor, (i = 1, \cdots, m-1)$, and for the last group m, $n_m = N - \lfloor N/(m-1) \rfloor \cdot (m-1)$. For the GPU 3-level tree-based synchronization, the number of groups and the number of blocks in each group can be calculated in a similar way.

When the number of blocks is small, the GPU 2-level tree-based method may incur more overhead than the GPU simple synchronization approach. However, as the number of blocks gets larger, the advantage of the tree-based approach is obvious. For example, if we only consider the time for atomic addition, when $N > 4$, the GPU 2-level tree-based approach will outperform the GPU simple

[2] Because volatile variables are not supported in atomic functions in CUDA, an `atomicCAS()` function should be called within the `while` loop to prevent the `while` loop from being compiled away.

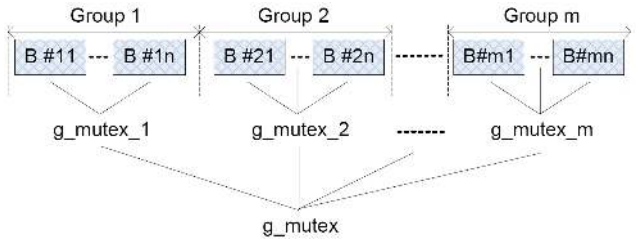


Figure 8. GPU Tree-based Synchronization

synchronization approach, assuming all blocks finish their computation at the same time. In practice, because there are more checking operations in the GPU 2-level tree-based synchronization approach than the GPU simple one, the threshold value will be larger than 4. The same case is for the comparison of the GPU 2-level and 3-level tree-based synchronization approaches.

5.3 GPU Lock-Free Synchronization

While scaling better than GPU simple synchronization, GPU tree-based synchronization still relies on costly CUDA atomic operations. In this section, we propose a lock-free synchronization algorithm 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.

As shown in Figure 9[3], our lock-free synchronization algorithm 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`, 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. The leading thread in each block continues its execution once it sees the corresponding element in `Arrayout` is set to `goalVal`.

It worths noting that in the step 2 above, rather than having one thread to check all elements of `Arrayin` *in serial*, we use N threads to check the elements *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 GPU simple synchronization.

From Figure 9, there is no atomic 'add' 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 10,

[3] Similarly, because volatile variables are not supported in atomic functions in CUDA, an `atomicCAS()` function should be called within the `while` loop to prevent the `while` loop from being compiled away.

```
1  //GPU lock-free synchronization function
2  __device__ void __gpu_sync(int goalVal,
3              int *Arrayin, int *Arrayout)
4  {
5      // thread ID in a block
6      int tid_in_block = threadIdx.x * blockDim.y
7                       + threadIdx.y;
8      int nBlockNum = gridDim.x * gridDim.y;
9      int bid =  blockIdx.x * gridDim.y + blockIdx.y;
10
11     // only thread 0 is used for synchronization
12     if (tid_in_block == 0)    {
13         Arrayin[bid] = goalVal;
14     }
15
16     if (bid == 1)    {
17         if (tid_in_block < nBlockNum)    {
18             while (Arrayin[tid_in_block] != goalVal)  {
19                 ......
20             }
21         }
22         __syncthreads();
23
24         if (tid_in_block < nBlockNum)    {
25             Arrayout[tid_in_block] = goalVal;
26         }
27     }
28
29     if (tid_in_block == 0)    {
30         while (Arrayout[bid] != goalVal)    {
31             ......
32         }
33     }
34     __syncthreads();
35 }
```

Figure 9. Code snapshot of the GPU Lock-free Synchronization (Some code details have been omitted to enhance readability.)

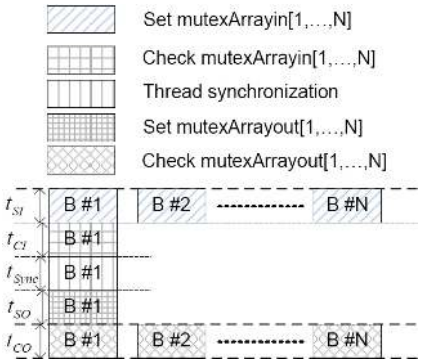


Figure 10. Time Composition of GPU Lock-free Synchronization

the execution time of `__gpu_sync()` is composed of five parts and calculated as

$$t_{GLS} = t_{SI} + t_{CI} + t_{Sync} + t_{SO} + t_{CO} \quad (9)$$

where, t_{SI} is the time for setting an element in `Arrayin`, t_{CI} is the time to check an element in `Arrayin`, t_{Sync} is the intra-block synchronization time, and t_{SO} and t_{CO} are the time for setting and checking an element in `Arrayout`, respectively. From Equation (9), execution time of `__gpu_sync()` is unrelated to the number of blocks in a kernel.

5.4 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 10000 times is used. In other words, in the CPU synchronization, each kernel calculates the mean once and the kernel is launched 10000 times; in the GPU synchronization, there is a 10000-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 11. 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 11, 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 overlapped. 2) Even for the CPU implicit synchronization, a lot of synchronization time is needed. From Figure 11, 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 simple synchronization, the synchronization time is linear to the number of blocks in a kernel, and more synchronization time is needed for kernels with a larger number of blocks, which matches very well to Equation (6) in Section 5.1. Compared to the CPU implicit synchronization, when the block number is less than 24, its synchronization time is less; otherwise, its synchronization time is more. The reason is that, as we analyzed in Section 5.1, more blocks means more atomic add operations are needed for the synchronization. 4) Compare the GPU simple synchronization to the GPU tree-based synchronization, if the block number is less than 11, the 2-level tree-based synchronization needs more time than the GPU simple synchronization; otherwise, its synchronization time is less than the GPU simple synchronization. Time consumption relationship of the 2-level and 3-level tree-based synchronization approaches is similar except the block number threshold is 29. 5) For the GPU lock-free synchronization, since there are no atomic 'add' operations used, all the operations can be executed in parallel, which makes it unrelated to the number of blocks in a kernel, i.e., the synchronization time is almost a constant value. Furthermore, its synchronization time is much less (for more than 3 blocks set in the kernel) than that of all other synchronization methods.

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 GPU synchronization approaches are compared and analyzed.

6. 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.

6.1 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

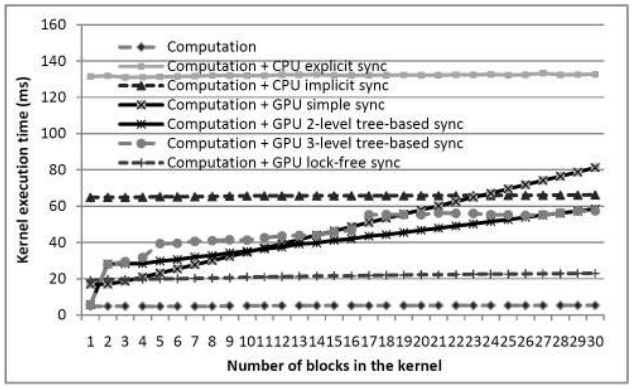


Figure 11. Execution Time of the Micro-benchmark.

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.

6.2 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 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 alignment 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.

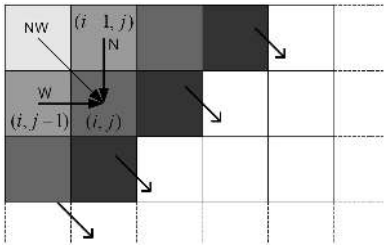


Figure 12. Wavefront Pattern and Dependency in the Matrix Filling Process.

Only after the previous anti-diagonals are computed can the current one be calculated as shown in Figure 12. 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 and it is the object to be parallelized.

6.3 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 $O\left(n\log^2(n)\right)$ comparators and has a delay of $O\left(\log^2(n)\right)$.

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.

7. Experiment Results and Analysis

7.1 Overview

To evaluate the performance of our proposed GPU synchronization approaches, we implement them in the three algorithms described in Section 6. 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 approach for each algorithm, their performance is evaluated in three aspects: 1) Kernel execution time decrease caused by our proposed GPU synchronization approaches and its variation against the number of blocks in the kernel; 2) According to the kernel execution time model in Section 4, we calculate the *synchronization time* of each synchronization approach. Similarly, the synchronization time variation against the number of blocks in the kernel is presented; 3) Corresponding to the best performance of each algorithm with each synchronization approach, the percentages of the computation time and the synchronization time are demonstrated and analyzed.

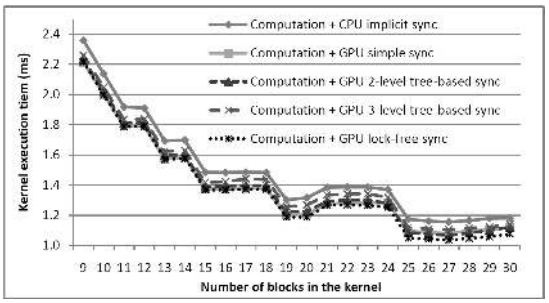
Our experiments are performed on a GeForce GTX 280 GPU card, which has 30 SMs and 240 processing cores with the clock speed 1296MHz. The on-chip memory on each SM contains 16K registers and 16KB shared memory, and there are 1GB GDDR3 global memory with the bandwidth of 141.7GB/Second on the GPU card. 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 operation 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.

7.2 Kernel Execution Time

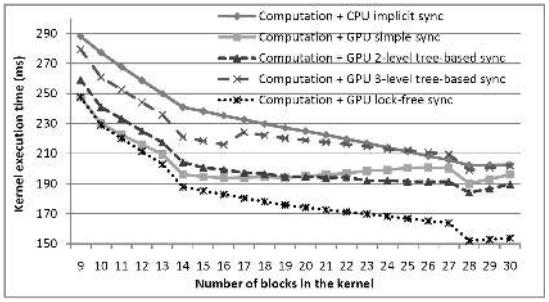
Figure 13 shows the kernel execution time decrease with our proposed synchronization approaches and its variation versus the number of blocks in the kernel. Here, we display the kernel execution time with the block number from 9 to 30. This is due to two reasons: One is, if the number of blocks is less than 9, the performance is much worse than that with block number larger than 9; On the other hand, if a GPU synchronization approach is used, the maximum number of blocks in a kernel is 30. In addition, for the CPU implicit synchronization, we run the programs with block number from 31 to 120 with step of one block, but find that performance with 30 blocks in the kernel is better than all of them, so we do not show the performance with block number larger than 30 for the CPU implicit synchronization. The number of threads per block is 448, 256, and 512 for FFT, SWat, and bitonic sort, respectively. Figure 13(a) shows the performance variation of FFT, Figure 13(b) displays the results of SWat, and Figure 13(c) is for bitonic sort.

From Figure 13, 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 can 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 1.18ms to 1.076ms, which is an 8.81% decrease. For SWat and bitonic sort, this value is 24.1% and 39.0%, 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. So the impact on the total kernel execution time caused by the synchronization time decrease in FFT is not as much as that in SWat and bitonic sort.

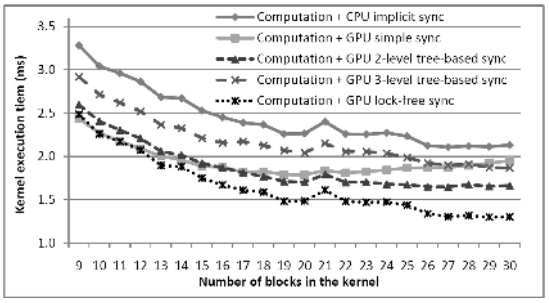
In addition, among the implementations with the proposed GPU synchronization approach, 1) With more blocks set in the kernel, kernel execution time decrease rate of the GPU simple synchronization is not as fast as the other three (GPU 2-level tree-based, 3-level tree-based, and lock-free synchronization approaches). In FFT, when the block number is less than 24, kernel execution time with the GPU simple synchronization is less than that of the 2-level tree-based synchronization; Otherwise, performance with the 2-level tree-based synchronization is better. Similarly, the threshold values of SWat and bitonic sort are both 20; 2) If we consider the GPU 2-level and 3-level tree-based synchronization approaches, performance with the former is always better than that with the latter. This is because, as we discussed in Section 5.2, if the number of blocks in the kernel is small, the mutex variable checking in all 3 levels will make it need more time than the GPU 2-level tree-based approach; 3) In the three algorithms, performance with the GPU lock-free approach is always the best. The more blocks are set



(a) FFT

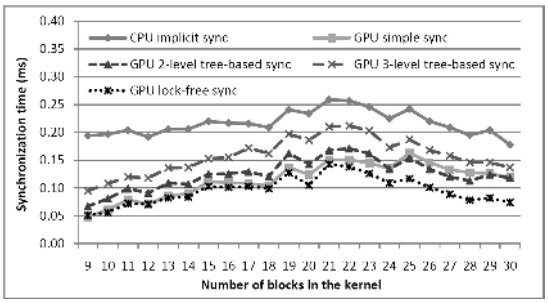


(b) SWat

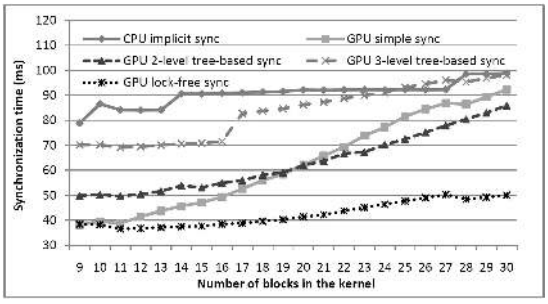


(c) Bitonic sort

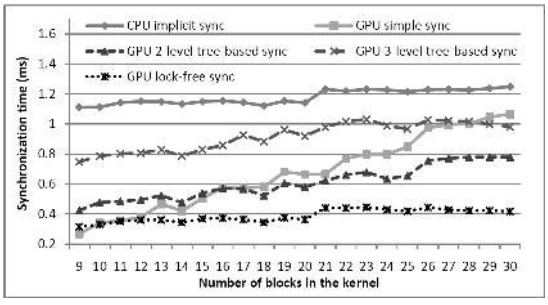
Figure 13. Kernel Execution Time versus Number of Blocks in the Kernel



(a) FFT



(b) SWat



(c) Bitonic sort

Figure 14. Synchronization Time versus Number of Blocks in the Kernel

in the kernel, the more performance improvement can be obtained compared to the other three GPU synchronization approaches. The reason is the time needed for the GPU lock-free synchronization is almost a constant value and no additional time is needed for the synchronization when there are more blocks set in the kernel.

7.3 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 implementation method, 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, the time of each synchronization method in the three algorithms are shown in Figure 14. Similar as Figure 13, we show the number of blocks in the kernel from 9 to 30. Figure 14(a), 14(b), and 14(c) are for FFT, SWat, and bitonic sort, respectively.

From Figure 14, in SWat and bitonic sort, synchronization time matches the time consumption models in Section 5. First, the CPU implicit synchronization approach needs the most time while the GPU lock-free synchronization takes the least time. 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 simple and 2-level tree-based synchronization approaches, the synchronization time increases with the increase of the number of blocks in the kernel; and the time increase of the GPU simple synchronization is faster than that of the GPU 2-level tree-based synchronization; Fourth, the GPU 3-level tree-based synchronization method needs the most time among the proposed GPU synchronization methods. For FFT, though the synchronization time variation is not regular versus the number of blocks in the kernel, their difference across different synchronization approaches is the same as the other two algorithms. The reason for the irregularity is caused by the property of the FFT computation, which needs more investigation in the future.

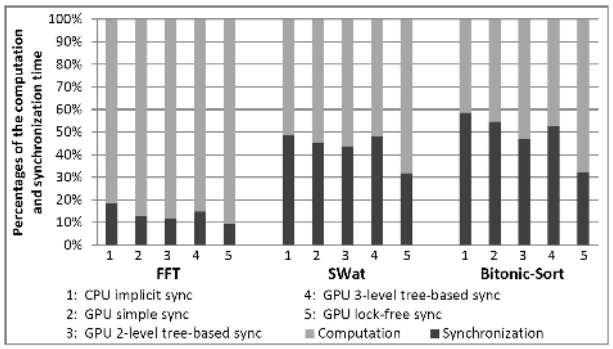


Figure 15. Percentages of Computation Time and Synchronization Time

7.4 Percentages of the Computation Time and the Synchronization Time

Figure 15 shows the performance breakdown in percentage of 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 13, 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 percentage is about 50% percent in SWat and bitonic sort. This indicates that inter-block communication time occupies a large part of the total execution time in some algorithms. As a result, 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 about 50% to about 30% in SWat and bitonic sort, but that of FFT is much less, from about 20% to about 10%. The reason is similar, i.e., synchronization time decrease does not impact the total kernel execution time much because its percentage in the total kernel execution time is small.

8. Conclusion

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

Second, we propose three synchronization approaches: GPU simple synchronization, GPU tree-based synchronization and GPU lock-free synchronization. The first two use mutex variables 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 efficiency with the aforementioned performance model.

We evaluate the three 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. In addition, the time needed for each GPU synchronization approach matches the time consumption model well. Finally, 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 about half of the total execution time. This demonstrates that for data-parallel algorithms with considerable data communication, decreasing synchronization time is as important as optimizing computation.

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 obtained for their parallelization on multi- and many-core architectures.

References

[1] AMD Brook+ Presentation. In *SC07 BOF Session*, November 2007.

[2] 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.

[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'2008*, pages 1–12, October 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'2008*, October 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, Feb 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. LU, QR and Cholesky Factorizations Using Vector Capabilities of GPUs. *Technical Report, UCB/EECS-2008-49, EECS Department, University of California, Berkeley, CA*, May 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`.