Weak Execution Ordering - Exploiting Iterative Methods on Many-Core GPUs

Jianmin Chen, Zhuo Huang, Feiqi Su, Jih-Kwon Peir and Jeff Ho
Dept. of Computer and Information Science and Engineering
University of Florida
Gainesville, FL
Email: {jichen, zhuang, fsu, peir, jho}@cise.ufl.edu

Lu Peng
Dept. of Elect. and Computer Eng.
Louisiana State University
Baton Rouge, LA
Email: lpeng@lsu.edu

***Abstract*—On NVIDIA's many-core GPUs, there is no synchronization function among parallel thread blocks. When fine-granularity of data communication and synchronization is required for large-scale parallel programs executed by multiple thread blocks, frequent host synchronization are necessary, and they incur a significant overhead. In this paper, we investigate a class of applications which uses a chaotic version of iterative methods [5], [22] to obtain numerical solutions for partial differential equations (PDE). Such a fast PDE solver is parallelized on GPUs with multiple thread blocks. In this parallel implementation, although frequent data communication is needed between adjacent thread blocks, a precise order of the data communication is not necessary. Separate communication threads are used for periodically exchanging the boundary values with adjacent thread blocks through the global memory. Since a precise order of the data communication is not required, the computation and the communication threads can be overlapped to alleviate the communication overhead. Performance measurements of two popular applications, *Poisson image editing* from computer graphics and *shape from shading* from computer vision, on Tesla C1060 show that a speedup of 4-5 times is achievable for both applications in comparison with the solution using host synchronization.**

I. Introduction

The recent introduction of NVIDIA's many-core GPUs [9], [15] and their programming environment CUDA [12] makes inexpensive desk-top supercomputing available to many users. Historically, a key performance aspect in parallel programming on systems with multiple processing units is to provide an efficient mean of data communication and synchronization in order to satisfy the data dependencies among multiple threads. Traditionally, this requirement is met by direct reads/writes or atomic read-modify-write to shared memory locations from multiple threads to serve the needed communication and synchronization functions. However, due to long memory access latency along with coherency overheads when equipped with local memories or caches, it is essential to minimize data communication and synchronization throughout parallel program execution. The NVIDIA's CUDA GPUs contain a large number of Streaming Processors (SPs) grouped into a number of Streaming Multiprocessors (SMs). The Tesla C1060 GPU [9] has 30 SMs, each with 8 SPs, for a total of 240 processors. To program the Tesla C1060 GPU, NVIDIA's Compute Unified Device Architecture (CUDA) extends ANSI C with a few new program constructs and key words [12]. The programmers can use the CUDA extensions to parallelize the programs into three levels. In the highest level, a *kernel* can be invoked from the host CPU to create a single *grid* to run on the GPU. In order to better exploit software/hardware parallelism, parallel kernel invocations are allowed on the same or multiple GPUs [9]. Since the data communication and synchronization mechanism remains the same with/without parallel kernel invocations, we will omit further discussions at this level. Each kernel has multiple *thread blocks* which can be specified as a three-dimensional array. Each thread block consists of multiple *threads* which can also be specified in three-dimension. A thread block is scheduled to run on one SM independently with other thread blocks. Within each thread block, parallel threads are further grouped into *warps*, each has 32 threads. A warp is the smallest scheduling unit to run on an SM. Multiple warps in the same or different thread blocks can be scheduled on the same SM simultaneously limited only by the hardware resource constraints in each SM.

The CUDA GPU has multiple levels of memory hierarchies. The *global* memory (also called device memory) which is located off-chip with long access latency can be accessed by all thread blocks in a grid. The on-chip *shared* memory with fast access latency is accessible from all threads within a thread block. An efficient usage of the fast but size-limited shared memory becomes critical in reducing expensive global memory accesses. Data that is shared among threads in the same thread block can be moved into the shared memory to exploit reference locality for high performance. The off-chip *local* memory can only be accessed by a single thread. Due to its long latency, the local memory is suitable for certain functions, e.g. register spilling. The CUDA GPU also has *constant* and *texture* memories for the read-only data. Although located off-chip, the data in these two memories can be cached for fast accesses. When a kernel is invoked from the host CPU, the needed data must be copied from the host main memory to the device's global memory before kernel computations can start.

Despite the accessibility of the global memory from all thread blocks, there is no mechanism for synchronization among the thread blocks in a grid. In other words, *precise* data communication and synchronization among threads in different thread blocks within a grid is not permitted. Instead, a barrier

978-1-4244-6022-9/10/$26.00 ©2010 IEEE

synchronization is provided only at a lower level among the threads in the same thread block using *__syncthreads()*. To meet the synchronization requirement among multiple thread blocks, the kernel must exit back to the host. Afterwards, the host may invoke subsequent kernels to continue the program execution. Therefore, frequent host synchronization is necessary for large-scale parallel programs executed by multiple thread blocks when fine-granularity of data communication and synchronization is required among the parallel thread blocks. The Faster Global Barrier in [23] sets *flags* in the global memory, which are accessible from all thread blocks. By setting/testing the flags from multiple thread blocks, a global barrier can be accomplished without going back to host. However, this scheme can only work when all thread blocks are scheduled and executed simultaneously. Many CUDA applications with high parallelism, including those used in this paper, invoke more thread blocks than what the hardware can execute simultaneously.

The host synchronization incurs two noticeable overheads:

1) The lifespan of the shared variables located in the shared memory is confined within a single kernel invocation. Therefore, the data in the shared memory must be saved into the global memory for future reuses before the kernel exits to the host. The saved data is likely to be restored back into the shared memory in the subsequent kernel invocation.
2) Beside the overhead involved in kernel initiation, proper intermediate results may need to be transferred back to the host for controlling the synchronization function.

The requirement for fine-granularity data communication and synchronization through the host degrades the performance of using multiple thread blocks. In this paper, we investigate a class of applications which exhibits high parallelism but requires tight data communication among parallel threads. However, a unique feature in this class of applications is that the precise order of data communication among parallel threads is not required. A typical example involves a partial differential equation (PDE) solver based on chaotic relaxation [5] which uses an iterative algorithm until the solution converges. Although a sequential order of data communication may speed up the convergence, a *weak execution ordering* enables parallel thread blocks to be executed independently without precise data communication and greatly reduces the overhead of using the host synchronization. There are concerns about the PDE solver's accuracy that may be critical for some applications. However, in many applications like image smoothing, editing, mesh smoothing, image inpainting, and so on, the actual proofs of convergence are often not important but empirical observances of convergence are.

In order to accomplish tight data communication among the thread blocks, we use an *asynchronous* data exchange mechanism through the global memory [22]. Each thread block periodically sends newly computed data to the global memory in exchange for the needed data produced by other thread blocks. In this paper, we exploit this inter-block communication mechanism to speed up the iterative algorithm. In handling the data exchanges, two different types of threads, *computation* and *communication* are created in each thread block. The computation threads execute the normal computation functions while the communication threads are dedicated for storing and fetching the data to and from the global memory. It is important to note that this data communication through the global memory can be performed independently in each thread block. There is no particular order among the thread blocks in exchanging the data and hence it is referred to as *asynchronous* data communication. The computation and communication threads can be overlapped to hide the communication overhead. The frequency of data exchanges is based on the ability to overlap the computation and communication threads as well as the balance between the global memory traffic and the convergence speed. Note also that although our study is based on Tesla C1060, the proposed solutions are applicable for all CUDA GPUs including the new Fermi generation [16].

Performance evaluations using a Poisson Image Editing and 3-D Shape from Shading applications show that the *asynchronous* data communication is effective in reducing the requirement for host synchronization with minimal impact on the convergence of the solution. What is more, the tuning of this data communication can significantly improve the performance. By overlapping the communication and computation threads with proper tuning of the thread block scheduling, we have observed that a speedup of 4-5 times is achievable for both applications in comparison with the solution with fine-granularity of host synchronization.The paper is organized as follows. We will first describe the CUDA GPU architecture constraints with NVIDIA Tesla C1060 as an example in Section II. The selected applications for this study are introduced in Section III. This is followed by the details in parallelizing the applications using CUDA in Section IV. Section V presents the performance measurement results for different data communication and synchronization methods. Section VI describes the related work and Section VII concludes the paper.

II. Architecture and Synchronization Overhead

Nvidia's GPUs and CUDA present many constraints in creating and scheduling parallel threads on multiple SMs. We use Tesla C1060 as an example as listed in Table I. It relies on the programmers to optimize the CUDA code for efficiently utilizing the underline hardware. The basic optimization strategy is to fully populate all SPs with simultaneous thread blocks and threads to achieve the best hardware utilization. In addition, it is important to minimize global memory accesses by exploiting data reference locality using the fast registers and shared memory. Unfortunately, the optimization space of CUDA/GPU is multi-dimensional and non-linear [21] such that optimizing one constraint often impact another constraint. Note that NVIDIA provides tools like occupancy calculator [14] and visual profiler [13] to assist programmers in understanding the utilization of the GPU resources in their programs.

TABLE I
RESOURCE CONSTRAINTS OF TESLA C1060

Resource	Limitation
Threads per SM	1024 threads
Thread Blocks per SM	8 blocks
Threads per Warp	32 threads
Warps per SM	32 warps
Threads per Thread Block	512 threads
Registers (32 bits) per SM	16384 registers
Shared Memory per SM	16K bytes

The goal in this paper is to study the performance impact of the data communication and synchronization requirement among parallel thread blocks. In parallelizing the applications, we isolate the data communication and synchronization requirement from other domain of program optimizations on GPUs. We first try our "best effort" to optimize the register and shared memory usage per thread block to be fitted into the available resources in each SM. We also consider the efficiency of accessing the global memory by arranging stride-1 accesses among the threads in a warp. Meanwhile, we consider creating enough schedulable warps and thread blocks to populate the SMs for keeping the SP pipeline busy. Based on the hand-optimized code, we can then evaluate different data communication and synchronization mechanisms for the selected applications. The detailed descriptions of the two selected applications and their parallelization will be given in Section III.

Next, to demonstrate the performance impact of the host synchronization, we experiment with a parallel Breadth First Search (BFS) algorithm [18]. The BFS problem is to find the minimum number of edges needed to reach every vertex in a graph G from a source vertex s. The parallel BFS starts from s by assigning an edge count of 1 to all its neighboring vertexes. Afterwards, the previously found neighbors become the source vertexes for searching their immediate neighbors whose edge counts are equal to 2. This process continues until all vertexes in G are either visited or unreachable from s. Each of these traversal steps is referred as a *level* and is assigned a value that is equal to the number of edges measured from the original source s. During the traversal, a vertex, once visited, is dropped from further considerations.

In the experiment, we limit the parallelization to a single thread block with 512 threads so that the host synchronization can be replaced by *__syncthreads()* within the thread block for barrier synchronization in order to separate the levels of searches. Furthermore, we experiment with a relatively small graph with 3K vertexes in order to fit a small read-only vertex data array in the shared memory to add its reloading overhead in each kernel invocation. During the traversal, two barrier synchronizations are needed at each traversal level. The total amount of synchronizations depends on the connectivity of the graph. In this experiment, multiple graphs are generated using the graph generation tool from [1] with different edge/vertex ratios. Since the number of vertexes is fixed, the traversal levels decrease with the number of edges. However, graphs with more edges increase the time used in searching for the neighboring vertexes; hence they require longer execution time. The pseudo-CUDA kernel program without host synchronization is given in Figure 1.

```
tid = getthreadID;
Finish = false;
while (NOT Finish) do
  Finish = true;
  for all nodes vid calculated by thread tid do
    if (vid is marked as "current level") then
      mark all unvisited neighbors of vid as "next level"
      and set their edge counts;
    end if
  end for
  __syncthread();
  for all nodes vid calculated by thread tid do
    if (vid is marked as "next level") then
      change its mark to "current level";
      mark itself visited;
      Finish = false;
    end if
  end for
  __syncthread();
end while
```

Fig. 1. Psudo-CUDA BFS kernel

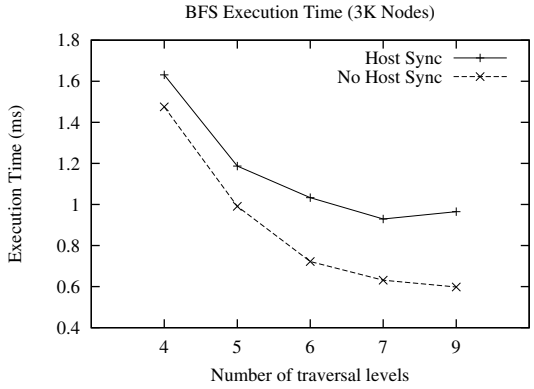


Fig. 2. BFS execution time with/without host synchronization.

In Figure 2, we compare the execution time with and without the host synchronization. The horizontal axis represents the total number of traversal levels, hence the synchronization frequency for the respective graphs. As expected, we can observe that the synchronization through the host, even with a small number of barrier synchronizations, degrades the performance significantly. The execution time gap between host and no-host synchronization increases with the frequency of synchronizations. This comparison can be measured because we limited the BFS parallelization in one thread block. However, it is impossible to avoid the host synchronization when the program has more than one thread block.

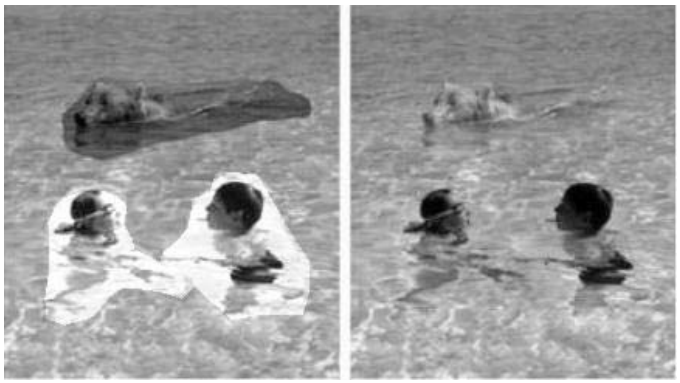


Fig. 3. Inserting objects with complex outlines into a new background: cloning on the left, seamless cloning with Poisson Image Editing on the right. Images taken from [19].

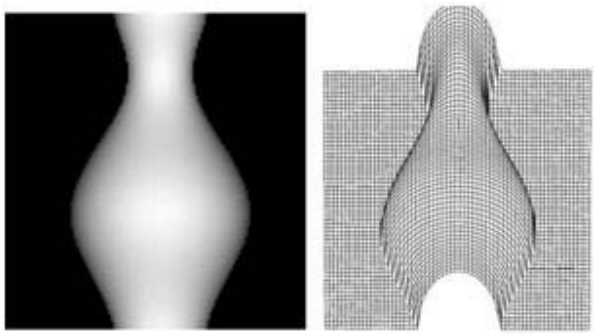


Fig. 4. An example of shape from shading. An image of a vase is shown on the left and the recovered 3-D surface is shown on the right. The height field of the computed 3-D surface is obtained by integrating the normal vector field estimated from the image.

III. Applications using Iterative Method

Iterative methods are frequently used to obtain numerical solutions for partial differential equations (PDE) arising from a variety of engineering and scientific problems. A fast and accurate PDE solver can be valuable for designing computational solutions to a wide range of applications and problems. In this paper, we investigate two applications that require an efficient and accurate PDE solver on grids: *Poisson image editing* from computer graphics and *shape from shading* from computer vision. In both applications, a crucial component is a module that solves a specific PDE on a grid, and the performances of the two algorithms depend critically on the ability of the solver to efficiently and accurately compute the numerical solutions.

A. Poisson Image Editing

Image editing is a computer graphics application that has become increasingly common in our everyday life, especially with the advent of inexpensive digital cameras. Adobe's Photoshop is perhaps the most popular image editing software that has been used widely for editing, modifying and polishing images. A useful image editing function is the ability to seamlessly integrate new image contents into a given background image. Figure 3 shows an interesting example in which new image patches are introduced into the background image of a swimming pool [19]. As the direct cloning of the intensity values produces unacceptable results with visible seams at the patch boundaries that significantly degrades the realism of the image, the challenge here is to develop an efficient and accurate algorithm that can automatically integrate different image contents to produce a realistic-looking composite image. The method proposed in [19] has become popular due to its conceptual simplicity and ease of implementation. By cloning the image gradients directly, the algorithm achieves seemless integration of different image patches by solving a Poisson equation on a grid with Dirichlet boundary condition. If $\boldsymbol{I}$ denotes the intensity values of a given patch in the final composite image, [19] proposes to estimate $\boldsymbol{I}$ by solving the following variational problem:

$$\min_{\boldsymbol{I}} \iint_{\Omega} \| \nabla \boldsymbol{I} - v \|^2 \quad with\, \boldsymbol{I}\,|_{\partial\Omega} = \boldsymbol{I}^*\,|_{\partial\Omega} \tag{1}$$

where v denotes the image gradients of the new patch and $\boldsymbol{I}^*$ is the background image. The corresponding Euler-Lagrange equation is the Poisson equation with Dirichlet boundary condition:

$$\triangle \boldsymbol{I} = \mathbf{div}\ v, \quad \boldsymbol{I}\,|_{\Omega} = g \tag{2}$$

This equation can be solved on the image grid by first converting it into a linear system of equations using a finite-difference scheme that numerically approximates the partial derivatives. The linear system can then be solved using iterative method such as Gauss-Seidel that iteratively updates the value at each grid point using the values at its neighbors. While Gauss-Seidel is a well-established technique for solving PDEs numerically, its convergence rate can be slow and may require many iterations before stopping, particularly for images with large number of pixels. From a practical viewpoint, a fast numerical Poisson equation solver is imperative for developing an interactive image editing software.

B. 3-D Shape from Shading

The second application is a classical problem from computer vision that deals with recovering the 3-D structure of an object from its image. The ability to accurately recover the shape is crucial for many computer vision applications such as object recognition, object detection and other AI applications such as human face recognition, human-computer interaction (HCI) and virtual reality. Many techniques for recovering an object 3-D shape have been developed and a good portion of these algorithms require the solution of partial differential equations.

The basic idea underlying shape from shading [8] is to recover the normal vector field of the observed surface from image intensity values directly using a known shading model, which provides a formula that relates the normal vectors and external illuminations to the observed intensities.

Modeling the 3-D surface as the graph of a height function $\boldsymbol{H(x, y)}$ defined over a grid, the normal vector at a point is a

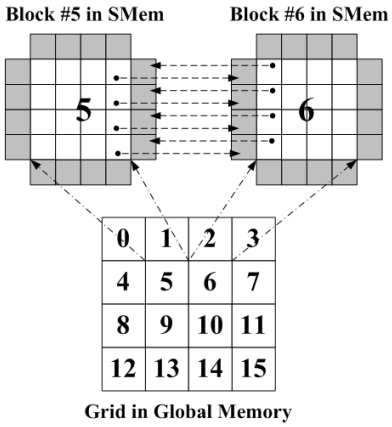


Fig. 5. Data exchange between adjacent thread blocks

function of the partial derivatives of $\boldsymbol{H}, f, g$. The normal vector field is recovered by minimizing the energy functional

$$\begin{aligned}\mathcal{E}(f,g) = \quad & \iint (\boldsymbol{I}(x,y) - R_s(f,g))^2 \, dx\, dy \\ & + \lambda \iint (f_x^2 + f_y^2 + g_x^2 + g_y^2)\, dx\, dy\end{aligned} \tag{3}$$

where *Rs(f, g)* denotes the predicted intensity value. The Euler-Lagrange equation shown in Equation 4 now gives a system of partial differential equations in f and g that can be converted into a linear system of equations using a finite-difference scheme, and both f and g can then be solved simultaneously using an iterative method similar to Gauss-Seidel.

$$\begin{aligned}\nabla^2 f &= -\lambda(E(x,y) - R_s(f,g))\tfrac{\partial R_s}{\partial f} \\ \nabla^2 g &= -\lambda(E(x,y) - R_s(f,g))\tfrac{\partial R_s}{\partial g}\end{aligned} \tag{4}$$

IV. Parallelization of the Applications

For both applications, our focus will be on improving the performance of their core PDE solvers. The numerical solutions for the PDEs are obtained by iteratively updating the values at each grid point with convergence criterion supplied by the users. Typically, the value at a grid point is updated using the values at its neighbors (including itself) from the previous iteration. On a uniprocessor, the updates can be performed only sequentially and the overall running time depends on the size of the grid as well as the specified convergence criterion. However, the Gauss-Seidel-type PDE solver here does not require exact sequential updates in the sense that it is not necessary to use only the values of its neighbors at previous iterative step. Numerically, the exact synchronization is not required since the convergence property of the algorithm would not be fundamentally altered by asynchronous updates that form the core of our parallelization scheme.

The basic idea behind our parallelization scheme is to parallelize the updates in each iterative step using multiple thread blocks. Each thread block produces its own local updates and the updated boundary values are propagated through the global memory to its neighboring thread blocks. As illustrated in Figure 5, the entire data grid in the global memory is divided into a set of smaller rectangular blocks (data blocks), and each data block is loaded into one shared memory and updated by one thread block. Every data block contains a number of *core* nodes plus a few *boundary* nodes, whose values will be used by the corresponding neighboring thread blocks. In the illustrated example, each boundary node between blocks 5 and 6 is marked by a little dot. The shaded nodes represent the boundary data from the neighboring blocks as indicated by the dashed arrows.

A. Weak Ordering in Data Communication

A straight-forward approach for meeting the data communication requirement is to synchronize all thread blocks back to the host after each iterative step on the GPU. The host checks the convergence and decides whether to invoke another kernel call to continue the iterative step. Before exiting, all thread blocks must save the intermediate results from the shared memory to the global memory. In the next iterative step, these results along with boundary nodes computed by other adjacent thread blocks in the previous iterative step are reloaded from the global memory to the shared memory. This host synchronization approach maintains precise data communication among all thread blocks between each iterative step and checks the convergence after each step.

Since precise data communication is not required for the iterative PDE solver, one intuitive approach to alleviate the synchronization and communication overhead is to reduce the frequency of host synchronizations. Instead of each iterative step, a *coarse* synchronization approach synchronizes the thread blocks after n iterative steps, where n can be any arbitrary positive integer. During the n iterative steps, each thread block is executed independently without exchanging data with its neighboring thread blocks. Although simple without data exchange overhead, the coarse synchronization likely slows down the convergence since the boundary values in each thread block cannot be updated using new data from its neighboring thread blocks during the entire n iterative steps.

To speed up the convergence in the coarse synchronization approach, we use *asynchronous* data communication method among threads blocks during n iterative steps between host synchronizations. Each thread block periodically sends the newly computed boundary values to the global memory for its neighboring thread blocks. Meanwhile, the thread block fetches the new boundary values computed by the neighboring thread blocks from the global memory to continue the following iterative steps. We call this data exchange activities as an *inter-block* communication. As there is no specific order for exchanging the data between neighboring thread blocks, each thread block may not always fetch the newest computed boundary values. Nevertheless, as the iterative step moves forward, each data exchange phase will likely get newer data from its neighboring thread blocks.

There are five important performance considerations in this *asynchronous* inter-block data communication method.

TABLE II
COMPUTATION THREADS VS. COMMUNICATION THREADS

Computation Threads	Communication Threads
Initialization Phase: Load 30×20 data nodes into shared memory *__syncthreads()*	*Initialization Phase:* Load the boundary values into shared memory *__syncthreads()*
Main Phase: While not done { Compute *l* iterations *__syncthreads()* }	*Main Phase:* While not done { Store and Load boundary values to/from Global memory *__syncthreads()* }
Ending Phase: Store the new 30×20 data nodes to global memory	*Ending Phase:*

The first consideration is to create separate *communication threads* in each thread block for moving the boundary values from/to the global memory. Separating communication threads from the computation threads overlaps the overhead of data communication with the real computation. Care must be taken, however, to make sure that the increase of the communication threads will not prevent the thread blocks from scheduling due to the resource constraint in each SM.

The second consideration is the need for the *local synchronization* between adjacent phases of data exchanges. Given separate computation and communication threads, the communication threads need to wait for all computations to reach to a proper synchronization point before the data exchange phase can begin. Such local synchronizations synchronize the execution pace between the computation and communication threads and guarantee to store the newest boundary values to the global memory. It pays a small overhead to synchronize all threads in a thread block in each data exchange phase.

The third consideration is the *frequency* of data exchanges within the *n* iterative steps. Clearly, the most important factor is the ability to overlap the communication threads with the computation threads. Too frequent of data exchanges could make the communication threads a bottleneck and slow down the computation threads. Also, although more frequent data changes help the convergence, it increases the global memory traffic and can therefore slow down the iterative step.

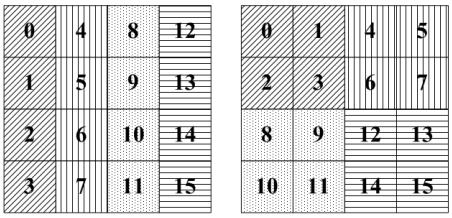
The fourth consideration is the resource constraints in scheduling the grid on the GPU and its impact on inter-block data communication. Given a large problem size, it is conceivable that all the thread blocks in a grid cannot be scheduled at once to the available SMs. Instead, a grid is executed by groups of thread blocks sequentially. Each group consists of a portion of the thread blocks that can be scheduled together. A thread block, once scheduled to run on an SM, will not release its allocated resources until it finishes the entire execution. This constraint further disrupts the asynchronous data communication. Even with periodic data exchanges, new boundary data would not be available if the data is produced by a thread block that has not been scheduled. Therefore, an important strategy in parallelizing the applications is to minimize the boundary communication among multiple scheduling groups of thread blocks.

The fifth consideration is the efficiency of data exchange through the global memory. To improve the efficiency of global memory accesses, NVIDIA's GPU coalesces global memory accesses from 16 threads in a half warp with regular access pattern (e.g. stride-1 accesses). To avoid inefficient data exchange, we consider moving the boundary values with long-stride accesses into a small stride-1 data array to be efficiently written and read to and from the global memory.

B. Choices of Parallelization Parameters

1) Choices of Thread/Data Block Sizes: One obvious way to minimize the data communications is to lower the number of boundary nodes. Simple geometric observations reveal that larger and square blocks are preferred over smaller and rectangular blocks given the same number of nodes. However, the choice of the data block size (number of nodes) for each thread block is also influenced by two other factors. First, the size of the thread block is limited by the size of the shared memory. Second, the global memory accesses from all threads of a half warp can be coalesced into a single memory transaction as long as the access pattern among the threads forms stride-1 accesses. Therefore, for efficient *coalesced* memory accesses, the width of the boundary nodes in the column dimension should be a multiple of 16. Furthermore, the number of threads in each thread block must be confined within the allowable 512 threads. Therefore, with 5 data blocks to be stored in shared memory for computation, we select the data block size of 32×20 to provide the most efficient data communication, meanwhile, each thread computes and updates two data nodes in each iterative step to reduce the number of threads per thread block to 320.

2) Computation vs. Communication Threads: Two types of threads: *computation* and *communication* are used in overlapping the data exchange of the boundary nodes between adjacent thread blocks with the real computation functions. For a problem size 480×480 in our experiment, there are 15×24 thread blocks; each consists of 320 computation threads with 32×20 data nodes. With the limit of 512 threads per thread block, this parallelization leaves a room of 192 threads for data communication. However, since there are only 104 boundary nodes in this configuration, it is reasonable to use 32, 64, or 128 communication threads for overlaping the data exchange. As illustrated in Table II, after loading the needed data block into the shared memory, the computation and communication



(a) Stripe Scheduling **(b) Square Scheduling**

Fig. 6. Stripe vs. square block scheduling (blocks with the same filling are executed concurrently)

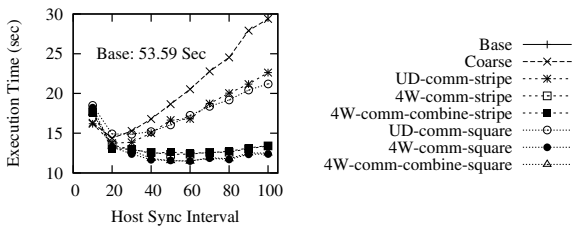


Fig. 7. Execution time of Poisson using eight communication/scheduling schemes

threads execute independently. *__syncthreads()* is used to synchronize all threads in each data exchange phase. Depending on the frequency of data exchange, the computation threads can go through l iterative steps before synchronizing with the communication threads. This synchronization guarantees to store the new updated boundary data to the global memory in each data exchange phase. Such synchronization is necessary, especially when the delays between computation and communication can be very different.

3) Thread Block Scheduling: Due to insufficient hardware resources, scheduling of thread blocks to maximize the benefit of inter-block communication is essential. For Tesla C1060, there are 30 SMs per GPU, hence up to 30 active thread blocks can be active at any given time. However, given a 480×480 grid in our experiment with 32×20 block size, there are 360 thread blocks. Since a thread block, once scheduled, will not release the resources until it finishes the execution, all the thread blocks in a grid are executed by groups sequentially. The sequential group execution impacts the inter-block communication since new boundary data cannot be available if the data is produced by a thread block that has not been scheduled.

From the above discussion, it is clear that the scheduled blocks need to be spatially contiguous on the grid so that inter-block communications are enabling among the active blocks. The thread execution manager schedules the thread blocks according to their block IDs. In this paper, we experiment with two group scheduling techniques: *Stripe* (linear) and *Square* (tiled). In the stripe configuration as shown in Figure 6, the block IDs are assigned so that the set of active blocks at any given time will form a stripe to maximize the inter-block communication on the side with the most efficient coalesced global moves. Similarly, in the square configuration, the set of active blocks will form a square to maximize the boundary nodes among the active blocks.

V. Performance Measurement

To carry out the experiment, both applications, poisson image editing (*Poisson*) and 3D shape from shading (*3D-Shape*) were parallelized with CUDA Version 2.3 and run on NVIDIA Tesla C1060 GPU. Each GPU has 4GB global memory, 30 SMs and clocked at 1.30 GHz. Other resource constraints of Tesla C1060 GPU are listed in Table I. In our experiment, we used a problem size of 480×480 for both applications using the established convergence thresholds reported in the literature [8], [19].

We experiment with *eight* combinations of different synchronization and communication mechanisms, data combination techniques, and block scheduling strategies. As the basis of comparison, the first approach is to synchronize each iterative step through the host (*Base*). The second approach is to initiate coarse synchronization (*Coarse*) through the host after certain number of iterative steps without any data communication in between. The remaining approaches use coarse host synchronizations as the basis but allow data communications among thread blocks between two consecutive host synchronizations. There are three data communication options. First, each thread block only communicates with its up and down neighbors. Given the fact that the nodes in the upper the lower boundaries of a thread block are stored continuously in memory, hence can be coalesced, this option provides efficient data communications. Second, each thread block communicates with all four neighbors to get the needed new data for the boundary nodes. Third, since the left and the right boundaries of each thread block are not consecutive in memory, this option combines the non-consecutive boundary nodes into a small array to shorten the communication cost in communicating with four neighbors. Furthermore, we also compare two block scheduling options, *stripe* and *square* for alleviating the impact on scheduling and executing the grid by groups of thread blocks. Scheduling thread blocks in squares reduces the number of adjacent nodes that have not been scheduled. In combining these communication and scheduling options, the third, fourth, and fifth, approaches use stripe scheduling with up/down communication (*UD-comm-stripe*), 4-way communication (*4W-comm-stripe*), and 4-way with data combining (*4W-comm-combine-stripe*) respectively. Similarly, the sixth, seventh, and eighth, approaches using square scheduling with up/down communication (*UD-comm-square*), 4-way communication (*4W-comm-square*), and 4-way with data combining (*4W-comm-combine-square*).

A. Execution Time and Convergence

We first examine the total execution time and convergence speed of various communication and scheduling mechanisms. In this study, the applications are parallelized with 320 computation threads and 32 communication threads in each thread block. The interval of data communication is set to be every 4 iterative steps. More sensitivity studies will be given in Section

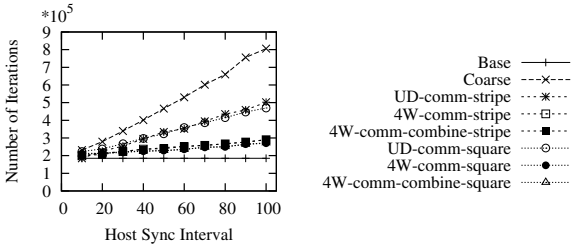


Fig. 8. Number of iterations of Poisson using eight communication/scheduling schemes

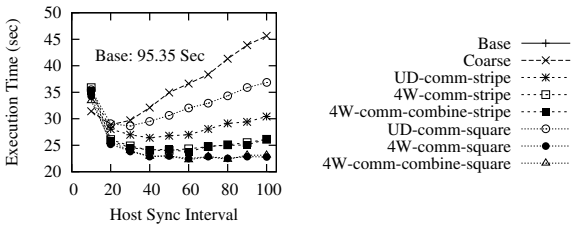


Fig. 9. Execution time of 3D-Shape using eight communication schemes

V-B. Figure 7 and 8 show the total execution time and the number of iterations to reach the convergence of *Poisson* for the eight communication and scheduling combinations. The results are measured over different intervals of host synchronization ranging from every 10 to every 100 iterative steps. Note that due to a much longer execution time (53.6 sec), the *Base* approach of using fine-granularity synchronization through host is not shown in Figure 7. Similarly, Figure 9 and 10 plot the results of *3D-Shape*, where *Base* took 95.4 sec to execute and is also missing in Figure 9.

We can make several important observations. First, we observe a significant execution time improvement using the asynchronous inter-block data communication mechanisms over the fine-granularity host synchronization for both applications. The improvement is also substantial in comparison with *Coarse*, where no data communication occurs between the host synchronizations. Although *Base* has the fastest convergence with respect to the number of iterative steps, the overhead of going back and forth between the host and the GPU increases the execution time per iterative step by about 5 times. With a host synchronization every 60 iterative steps, the overall execution times for *Poisson* are reduced from 53.6 seconds for *Base* to 11.4 seconds for *4W-comm-combine-square*. Similarly

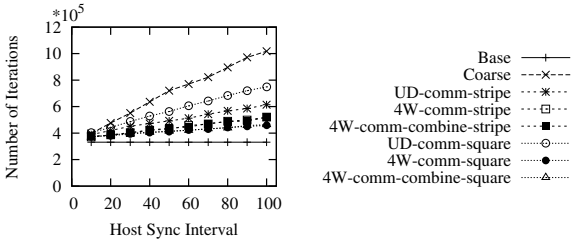


Fig. 10. Number of iterations of 3D-Shape using eight communication/scheduling schemes

for *3D-Shape*, the execution times are reduced from 95.4 seconds to 22.2 seconds. A speedup of 4-5 times is achieved for both applications.

Second, among various data communication and scheduling approaches, the 4-neighbor communication with the square scheduling shows the fastest execution time, followed by the same 4-neighbor communication with stripe scheduling. Both approaches are relatively insensitive to the interval of host synchronization as long as the interval is 20 iterative steps or greater revealing reasonably accurate data communication between two host synchronizations. For the block scheduling, *stripe* suffers from having more boundary nodes adjacent to the thread blocks that have not been scheduled, hence shows slower convergence than that with *square* scheduling.

Third, communicating with only the upper and lower neighbors has worse performance than that of the 4-neighbor communications. Although communications only with upper and lower neighbors can coalesce all global memory accesses and reduce the effective data moves, the lack of a portion of the new boundary values greatly slows down the convergence, hence is not a good communication strategy. Notice also that *square* scheduling can lower the performance of *UD-comm*. This is due to the fact that *stripe* schedules much more neighbors in the up and down directions.

Fourth, the data combining technique for non-stride-1 nodes in the left and right boundaries of each thread block during data communication does not show any noticeable difference. Detailed analysis discloses that data combining indeed lowers the amount of global memory moves. However, depending on the data communication intervals, lengthening the communication may not have impact on the execution time as long as the bottleneck is in the computation threads. More evidences will be given in the following sensitivity study.

Fifth, the general behavior between the two applications with respect to the data communication and block scheduling is very similar. Numerically, however, it takes almost twice as long for *3D-Shade* to converge as that for *Poisson*.

B. Sensitivity Studies

Two essential designs, the number of communication threads and the interval of inter-block communication impact the performance of the asynchronous data communication among multiple thread blocks. In this section, we show the results of sensitivity studies based on these two parameters. Since the behavior of the two applications is similar, only results from *3D-Shade* will be given.

We first simulate different inter-block communication intervals varying from 1 to 10 iterative steps. In this study, we use 32 communication threads with 320 computation threads as described before. The host synchronization interval is fixed at every 60 iterative steps. Figure 11, 12 and 13 show the respective execution time, the total number of iterative steps, and the time per iterative step with different inter-block communication intervals. We can observe that inter-block communication on every iterative step increases the execution time significantly. This is due mainly to the longer

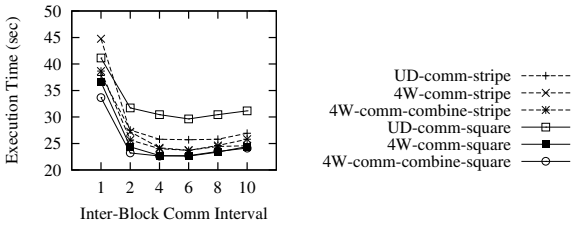


Fig. 11. Execution time of 3D-Shape with different inter-block communication intervals

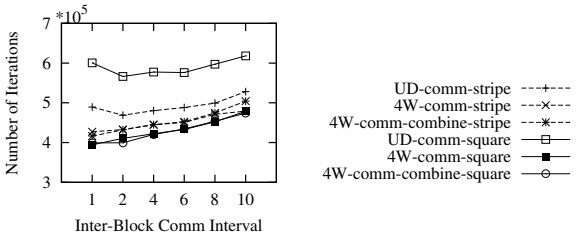


Fig. 12. Number of iterations of 3D-Shape with different inter-block communication intervals

execution time per iterative step as shown in Figure 13. We collect detailed timing in terms of number of cycles for the computation and the communication threads. The computation threads takes roughly 5500 cycles per iterative step while the communication thread of *UD-comm* takes 6000-7000 cycles and *4W-comm* takes 10000-12000 cycles depending on the block scheduling and the boundary data combining schemes. As a result, the communication threads dominate the execution time when the inter-block communication is on each iterative step. When the interval is lengthened to two iterative steps, the execution times of the two thread types are very similar. The computation threads dominate the execution time when the interval becomes 4 iterative steps or longer. The interval of four iterative steps has the fastest execution time because of its ability to totally overlap the communication and computation threads with faster convergence in comparison with longer intervals. Longer intervals reduce the frequency of getting the updated boundary values.

The effect of data combining becomes evident when the execution time is dominated by the communication threads. We can observe in both Figure 11 and 13, the execution times are much faster with data combining for *4W-comm* when the

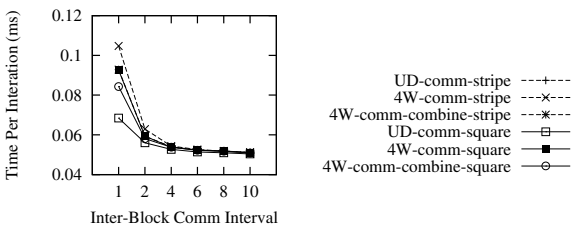


Fig. 13. Execution time per iteration of 3D-Shape with different inter-block communication intervals

inter-block communication interval is 1.

Next, we evaluate the impact on the number of communication threads. We try the inter-block communication interval of 1 and 4 iterative steps. Recall that one-step interval creates the bottleneck in the communication threads. We vary the communication threads from 0 to 128 threads. Since there are 320 computation threads, a single thread block can accommodate up to 192 separate communication threads. Without communication threads, we use the computation threads to accomplish the data movement. The execution time with different number of communication threads are plotted in Figure 14. Note that we do not include the results of *UD-comm* since it does not communicate all needed boundary values and performs worse than *4W-comm*.

Three observations can be made from the results. First, as expected, one-step interval takes much longer execution time than that for the four-step interval. This is again due to the bottleneck in the communication threads with one-step interval. Second, when the computation thread is the bottleneck, integrating the communication function into the computation thread increase the execution time as shown by the results of four-step interval. Third, when the communication thread is the bottleneck in one-step interval, it is advantage to use 128 communication threads to move all 104 boundary values in parallel. In this case, the 0 communication threads can slightly out-perform the 32 communication threads because without communication threads, each computation thread only needs to move up to one boundary node while with only 32 communication threads, each communication thread needs to move multiple boundary values.

VI. Related Work

There has been a large collection of literatures addressing the architecture, programming, parallelization, and performance issues with CUDA on modern many-core GPUs [9], [12], [15], [17], [20], [21]. Many performance of various applications and published papers are displayed in the CUDA Zone [14]. Due to the tight resource constraint on GPUs, program optimization is difficult and nonlinear. In [21], the authors performed an exhaustive search on the optimization space for applications running on NVIDIA GeForce 8800 GPUs. They demonstrated that some small changes in resource allocations could result in significant varying effects for the system performance due to the lack of runtime control for thread scheduling and resource allocation. For high-performance, global memory bandwidth must not be the bottleneck during the execution. Dealing with the memory bandwidth issue using software-managed local memories has been discussed in [20]. A memory model to improve the performance of nested loops on GPUs is reported in [6]. An analytical model for estimating the number of memory requests to show the critical role of global memory access latency in the overall execution time is described in [7]. In [23], a Faster Global Barrier was proposed to avoid going back to host for synchronization. Recently, a GPU simulator is implemented for analyzing performance of CUDA applications [2]. There are many works done in solving

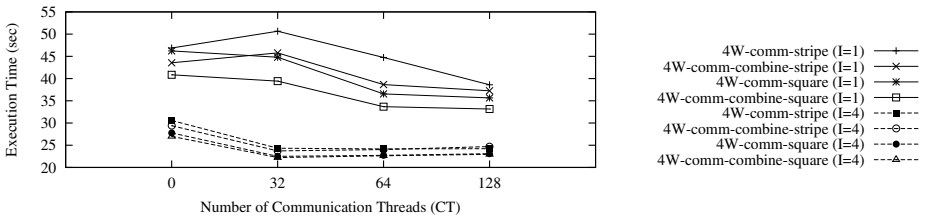


Fig. 14. Execution time of 3D-Shape with different number of communication threads

the problem using the iterative algorithm and its performance on many-core GPUs [3], [5], [10], [11], [22]. A mathematical survey summarizes known convergence conditions for the iterative algorithm [4]. Among these works, [22] implemented the Jacobi's iterative method for the 2-D Poisson equation on a structured grid for a heterogeneous multi-CPU and multi-GPU architecture. They reduced the synchronization bottleneck between iterations by basic fast-and-loose asynchronous algorithms based on chaotic relaxation. However, they did not have separate communication threads and did not consider that thread blocks would be executed in groups which can affect the communication efficiency. In addition, we move the boundary values with long-stride accesses into a small stride-1 data array to be efficiently written and read to and from the global memory. Furthermore, we thoroughly studied the impact of different frequency of local synchronization and communication.

VII. Conclusion

Inexpensive GPU hardware along with the CUDA programming model brings desk-top supercomputing to all users. However, due to the lack of precise data communication and synchronization mechanisms among the parallel thread blocks on a GPU, significant overhead occurs to accomplish such a function through the host CPU. In this paper, we thoroughly study and analyze the inter-block communication and its impact on an important class of applications based on chaotic relaxation of PDE solvers. During parallel computation on the GPU, separate and independent communication threads hide the communication overhead for exchange the needed boundary data. Performance measurement on Tesla C1060 showed that by reducing the host synchronization, a speedup of 4-5 times can be achieved for the two studied applications compared with the host synchronization implementation.

The new FERMI generation of NVIDIA's GPUs increase the transistors count to 3 billion while reduce the number of SMs to 16 [16]. The bigger SM can support more threads (1.5k) with larger shared memory (48K). The proposed asynchronous inter-block communication mechanism will be more efficient on FERMI due to a reduced amount of boundary data required for communication.

Acknowledgment

Omprakash Dhyade provided many useful discussions. Anonymous reviewers gave us helpful comments to improve the presentation of this paper.

References

[1] D. Bader and K. Madduri. Gtgraph: A synthetic graph generator suite. Technical report, National Science Foundation, 2006.
[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software*, April 2009.
[3] A. Balevic, L. Rockstroh, A. Tausendfreund, S. Patzelt, G. Goch, and S. Simon. Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose gpus. In *Computational Science and Engineering, CSE '08*, July.
[4] P. Bughw-sc, A. Frommer, A. Frommer, D. B. Szyld, and D. B. Szyld. On asynchronous iterations, 1999.
[5] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Its Applications*, 2:199–222, April 1969.
[6] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *in Proc. of the ACM/IEEE Conference on Supercomputing (SC06*. ACM Press.
[7] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009.
[8] B. Horn. Robot vision. Technical report, The MIT Press, March 1986.
[9] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.
[10] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 256–265, 2009.
[11] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, 2009.
[12] NVIDIA. Cuda programming guide 2.3.
[13] NVIDIA. Nvidia cuda visual profiler.
[14] NVIDIA. Nvidia cuda zone.
[15] NVIDIA. Nvidia geforce series gtx280, 8800gtx, 8800gt.
[16] NVIDIA. Whitepaper: Nvidia's next generation cudatm compute architecture: Fermi.
[17] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
[18] V. V. P. Harish and P. J. Narayanan. Large graph algorithms for massively multithreaded architectures. Technical report, IIIT, 2009.
[19] P. Pérez, M. Gangnet, and A. Blake. Poisson image editing. *ACM Trans. Graph.*, 22:313–318, July 2003.
[20] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.
[21] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, 2008.
[22] S. Venkatasubramanian and R. W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *Proceedings of the 23rd international conference on Supercomputing*, pages 244–255, 2009.
[23] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11.