Improving GPU Memory Performance with Artificial Barrier Synchronization

Shih-Hsiang Lo, Che-Rung Lee, *Member, IEEE,* Quey-Liang Kao,
I-Hsin Chung, *Senior Member, IEEE,* and Yeh-Ching Chung, *Senior Member, IEEE*

Abstract—Barrier synchronization, an essential mechanism for a block of threads to guard data consistency, is regarded as a threat to performance. This study, however, provides a different viewpoint for barrier synchronization on GPUs: adding barrier synchronization, even when functionally unnecessary, can improve the performance of some memory-intensive applications. We explain this phenomenon using a memory contention model in which artificial barrier synchronization helps reduce memory contention and preserve data access locality. To yield practical applications, we identify a program pattern: artificial barrier synchronization can be used to synchronize the memory accesses when the data locality among threads is violated. Empirical results from three real-world applications demonstrate that artificial barrier synchronization can increase performance by 10% to 20%.

Index Terms—Graphics Processors, Synchronization, Parallel Languages, Resource Contention.

1 Introduction

BARRIER synchronization for a group of processing units (PUs) is a mechanism to stop the execution of PUs at certain points in the program, called synchronization points, until all of the PUs in the group reach those points. When the computational loads among PUs are highly unbalanced, barrier synchronization can dramatically reduce processor utilization. Many performance optimization strategies have been proposed to minimize the number of synchronization points or to enhance the load-balancing among processing units for performance [1], [2], [3], [4], [5].

This study, however, presents a counter-intuitive phenomenon that, for GPUs (Graphics Processing Units), adding unnecessary barriers can enhance application performance. The barrier synchronization operation discussed is the `__syncthreads()` command in CUDA [6], which is effective for all of the threads in a thread block. The phenomenon is first demonstrated in a memory-testing program in which each thread tests different memory cells in a loop. Inserting an unnecessary `__syncthreads()` command at the end of the loop body makes the execution faster.

We investigate and explain this phenomenon using a memory contention model, which shows that artificial barrier synchronization can relieve memory contention and preserve data access locality by synchronizing the memory requests of a thread block. Without barriers, the threads of different progresses may compete for the same cache line, which increases cache misses.

```
kernel(...)
{
    ...
    for (...)
    {
      1. Memory accesses that have data locality
         and contend for the shared-memory system
      2. Simple arithmetic/logic computations
      3. *** Insert an artifical barrier here
    }
    ...
}
```

Fig. 1: Program pattern for the use of artificial barrier synchronization.

For the use of artificial barrier synchronization in applications, this study identifies a program pattern, which is presented in Fig. 1. In the program pattern, the threads access the GPU device memory frequently and perform simple computations in a loop. The accessed data should have a certain level of locality (e.g., data in a consecutive memory space); however, multiple simultaneous-independent memory requests cause the memory contention and pollute the data locality. Under these conditions, adding an artificial barrier at the end of the loop can improve the GPU memory performance.

Our survey indicates that this program pattern appears in real-world applications. As examples, we present two CUDA SDK programs: one for vector addition and one for scalar product, and one bio-informatics application, MUMmerGPU [7], to demonstrate the usefulness of artificial barrier insertion. MUMmerGPU is a GPU program that aligns queried strings with a single reference string.

We conducted experiments on three GPU devices: a

- *S.H. Lo is with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan. E-mail: awatch@gmail.com*
- *C.R. Lee is with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan. E-mail: cherung@cs.nthu.edu.tw*
- *Q.L. Kao is with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan. E-mail: u9662316@oz.nthu.edu.tw*
- *I.H. Chung is with the IBM T.J. Watson Research Center, New York 10598. E-mail: ihchung@us.ibm.com*
- *Y.C. Chung is with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan. E-mail: ychung@cs.nthu.edu.tw*

GeForce GTX 295, a GeForce GTX 480, and a GeForce GTX 690 (see Section 1 of the supplement). The experimental results showed that artificial barrier synchronization can yield performance improvements as high as 22%, 15% and 12% for the vector addition, scalar product and MUMmerGPU, respectively. Performance analyses indicated that the vector addition program receives an improvement when the DRAM is accessed, whereas the scalar product and sequence alignment programs receive an improvement when either the cache or DRAM is accessed.

The remainder of this paper is organized as follows. Section 2 briefly reviews two related topics: barrier synchronization and memory contention. Section 3 describes the observed phenomenon using a memory-testing kernel and presents a memory model to explain this phenomenon. Section 4 presents the program pattern with the necessary conditions for the use of artificial barrier synchronization. Section 5 introduces the three example applications and compares their experimental results with and without artifical barrier synchronization. The last section concludes this study with a list of proposals for future work.

2 Related Work

This study explores the relationship between artificial barrier synchronization and memory performance on modern GPUs. A literature search indicates that no studies have previously investigated this issue. This section briefly provides a survey of two related subjects: barrier synchronization and memory contention.

2.1 Barrier Synchronization

A wide variety of barrier synchronization mechanisms have been reported in the literature. Barrier synchronization mechanisms are specifically tailored for dedicated environments, such as shared-memory multiprocessors [8], [9], multi-clusters [10], [11] and super-computers [12]. Hardware-based implementations provide low latency, whereas software-based or hybrid implementations feature adaptability and scalability.

Apart from the implementations for CPU systems, the CUDA programming model provides a lightweight barrier command, `__syncthreads()`, to coordinate warps (i.e., groups of threads) in the same thread block for GPGPU (General-Purpose computing on Graphics Processing Units). The `__syncthreads()` command incurs low overhead, as indicated in the CUDA programming guide [6] and as reported by Wong et al. [13], to achieve its purpose.

Imposing barrier synchronization on thread execution could lower the utilization of multiprocessors. Research on performance optimization has investigated how to coordinate the thread execution and how to reduce the use of barrier synchronization. GPU-based algorithms [4], [5], for example, replaced block-level synchronization with warp-level synchronization. (A description of warp-level synchronization can be found in Section 1.1 of the supplement.) Another method for eliminating barrier synchronization is to place the variables that originally resided in shared memory to register file [14].

In addition to block-level synchronization, Volkov and Demmel [15] first implemented a global barrier synchronization that enables synchronization across GPU thread blocks. The proposed implementation has less cost than the CPU-based global synchronization (i.e., global synchronization is achieved when the GPU kernel execution finishes). Xiao et al. [16] also presented two global barrier synchronization mechanisms based on the `__syncthreads()` command. Feng et al. [17] investigated the performance and correctness of GPU-based global synchronization.

Burtscher et al. [5] and Alcantara [18] utilized `__syncthreads()` commands to avoid performing too much unwanted memory access. The additional barrier synchronization used in this study can increase memory performance by delaying the necessary memory operations.

2.2 Memory Contention

There has been a significant amount of research regarding resource contention issues. The purpose of reducing contention is not only to improve performance but also to guarantee quality of service (QoS) in the system. For cache contention, Qureshi et al. [19] proposed cache partitioning to minimize the cache misses caused by the co-running applications, whereas Fedorova et al. [20] and Zhuravlev et al. [21] presented cache-aware schedulers that try to fairly execute threads by isolating the cache-sensitive applications to different processors or by varying the scheduling priorities of threads.

In addition to on-chip memory, the DRAM system is a congestion area where multiple threads experience varied execution times. Multi-thread applications can cause inter- and intra-application contention for the DRAM system [22]. Contention for shared resources also occurs in NUMA systems [23], which feature non-uniform memory access latencies and multiple memory controllers. Even for a single thread, read and write requests cause interference in the DRAM memory system if both contend for the memory data bus. A specially designed DRAM-aware controller [24] tries to service the write requests that are likely to hit in the DRAM row buffer. This approach can avoid the performance penalty caused by switching between read and write requests.

To treat contention for the DRAM system, Mutlu et al. [25] presented a fair memory scheduler to equalize the quality of thread interference (which avoids prioritizing one particular thread), whereas Nesbit et al. [26] applied a fair queuing technique to the memory scheduling system. Among the many mechanisms proposed to reduce memory resource contention, Ebrahimi et al. [27] proposed a source throttling approach to directly limit the number of requests that applications can make to the CPU memory system.

GPU programmers experience a common contention situation that is specific to GPUs, the bank conflict of shared memory. The padding technique [28] can address this contention situation. Additionally, Bakhoda et al. [29] observed the contention for the GPU interconnect network and memory controllers using a GPGPU simulator [30].

Reducing the thread block size for a kernel is one method to relieve memory contention for performance. Barrier synchronization, however, provides another method to enhance performance when varied thread block sizes are involved. Adding artificial barrier synchronization not only decreases the number of memory accesses at a given time but also keeps (synchronizes) the memory accesses of threads in phase for data access locality.

3 Performance Improvement from Artificial Barrier Synchronization

3.1 An Example

A memory-testing kernel, MemTest, demonstrates the performance difference before and after applying artificial barrier synchronization. MemTest examines the defects of memory cells in a parallel fashion, similar to the Zero-One algorithm [31]. Specifically, we test a 480-MB global memory array using 30 512-thread blocks. Each thread checks 8192 (4-byte) integers and reports its result in its designated memory space. The output space required for all of the threads is 60 kB (i.e., $30 \times 512 \times 4$).

Algorithm 1: MemTest kernel

```
input  : An array $mem$ to be checked
output : An array $result$ to report the number of
         nonzero memory cells
1  int tid = blockIdx.x $*$ blockDim.x + threadIdx.x;
2  int *ptr = mem + tid;
3  int dist = gridDim.x $*$ blockDim.x;
4  int cnt = 0;
5  for (i=0; i<8; i++) do
6      // perform 1024 integer checks
7      Repeat1024( if(*ptr != 0) cnt++ ; ptr+=dist; );
8      __syncthreads(); // barrier synchronization
9  end
10 result[tid] = cnt; // report the number of nonzeros
```

Algorithm 1 presents the code of MemTest. Initially, each thread computes three values, the unique thread index in this kernel grid (`tid`), the memory address of its first integer (`ptr`) and the distance to the next integer (`dist`). Several built-in variables for the testing include the block index within the grid (`blockIdx.x`), the thread index within the block (`threadIdx.x`), the number of blocks in a grid (`gridDim.x`) and the number of threads in a block (`blockDim.x`). After setting the counter `cnt` to zero, each thread performs 1024 checks per iteration. Each check involves three operations: (1) the thread loads one 4-byte integer, (2) the thread increases its counter by one if the value of the memory cell is nonzero, and (3) the thread computes the address of the next integer. Fig. 2 shows the memory access pattern of the memory-testing kernel.

In Line 8 of Algorithm 1, a `__syncthreads()` command is added. Because each thread handles its individual part

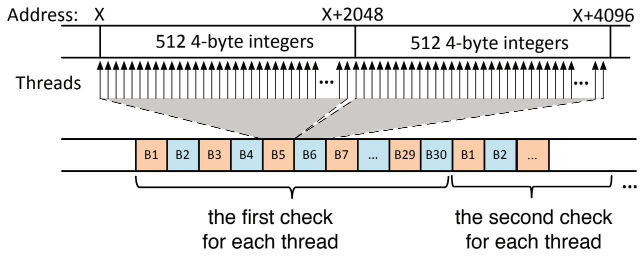


Fig. 2: Memory access pattern of the MemTest kernel using 30 blocks and 512 threads per block.

TABLE 1: Execution time for the MemTest kernel.

barrier synchronization	GTX 295	GTX 480	GTX 690
no	5.624	3.248	3.527
yes	5.208	3.208	3.409

Unit: millisecond

and reports the result in its designated memory space, the `__syncthreads()` command is unnecessary for correctness. This artificial barrier, however, improves the performance. Table 1 shows the execution times for the kernel with and without the `__syncthreads()` command on three GPU devices, GTX 295, GTX 480 and GTX 690. The additional barrier command reduces the kernel execution time by 7%, 1% and 3% on the three GPU devices, respectively, with standard deviation less than 0.3%.

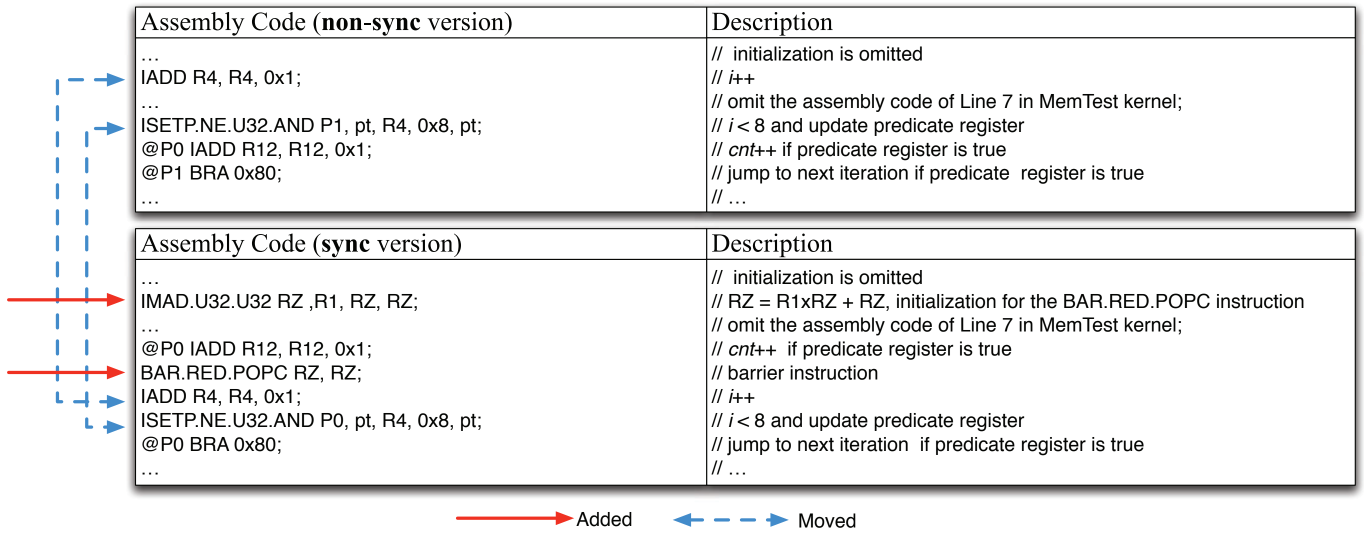
Additional memory-testing kernels that feature different access patterns are given in Section 2 of the supplement.

3.2 Analysis of the Assembly Code

We perform dis-assembly of the binary code of the MemTest kernel using the cuobjdump tool from the CUDA Toolkit [32] so as to understand the `__syncthreads()` command. Fig. 3 shows two differences between the sync version (i.e., the program with the additional barrier synchronization) and the non-sync version. First, the sync version contains a `__syncthreads()` command, which is one `BAR.RED.POPC` instruction along with one IMAD instruction for initialization in assembly language, as expected. Second, the non-sync version reorders the instructions of the original C program, i.e., the arithmetic instructions are moved forward for instruction parallelism.

In addition to those two differences in the assembly code, the `__syncthreads()` command serves two functionalities according to the CUDA technical guide [6]. First, the `__syncthreads()` command stalls the early warps that reach the barrier. Second, it guarantees memory access order (similarly to the `__threadfence()` command [6]).

To determine which one functionality causes the performance changes, we further examine the performance and assembly code of the programs using the `__threadfence()` command and the `BAR.ARV` instruction, in which a thread reports its arrival and continues without waiting for other threads. In brief, we found that the stalling (delay) effect of the barrier command contributes to the performance of the memory-testing kernel.



Assembly Code (**non-sync** version)	Description
…	// initialization is omitted
IADD R4, R4, 0x1;	// *i*++
…	// omit the assembly code of Line 7 in MemTest kernel;
ISETP.NE.U32.AND P1, pt, R4, 0x8, pt;	// *i* < 8 and update predicate register
@P0 IADD R12, R12, 0x1;	// *cnt*++ if predicate register is true
@P1 BRA 0x80;	// jump to next iteration if predicate register is true
…	// …

Assembly Code (**sync** version)	Description
…	// initialization is omitted
IMAD.U32.U32 RZ ,R1, RZ, RZ;	// RZ = R1xRZ + RZ, initialization for the BAR.RED.POPC instruction
…	// omit the assembly code of Line 7 in MemTest kernel;
@P0 IADD R12, R12, 0x1;	// *cnt*++ if predicate register is true
BAR.RED.POPC RZ, RZ;	// barrier instruction
IADD R4, R4, 0x1;	// *i*++
ISETP.NE.U32.AND P0, pt, R4, 0x8, pt;	// *i* < 8 and update predicate register
@P0 BRA 0x80;	// jump to next iteration if predicate register is true
…	// …

Fig. 3: Comparison of the assembly code for the MemTest kernel without and with barrier synchronization. Two instructions are added for the __syncthreads() command, and two instructions are moved for instruction parallelism.

3.3 Memory Contention

Because the GPU warps independently execute in the above example, simultaneous memory requests from multiple warps could cause memory contention for the cache and DRAM system. For the GPUs with caches, two warps that simultaneously request different data with the same cache line increase the cache misses. For the GPUs without caches, many warps could issue distant memory requests (in terms of memory addressing). A wide range of memory accesses, which are typically distributed across several DRAM memory rows, interfere (pollute) the locality of the DRAM row buffer, as discussed by Oh et al. [33] and by Alcantara et al. [34]. Although many memory scheduling strategies, e.g., FR-FCFS [35], have been introduced to increase the row buffer hits, this memory access pattern, which naturally exhibits a low access locality, constrains the overall improvement.

Using barrier synchronization, however, enables all of the warps in a block to synchronously stall and issue requests for data access locality, therefore reducing the aforementioned contention. Fig. 4 shows an example of reducing the memory contention using artificial barrier synchronization. Suppose that a GPU has a direct-mapped cache that consists of 6 128-byte cache lines. The memory segment requested by a warp fits into a cache line. Specifically, a memory request x is mapped to the cache line y, where $y = x$ mod 6.

In Fig. 4(a), for the cache, memory requests 14 and 20 cause the contention for cache line 2 (i.e., 2 = 14 mod 6; 2 = 20 mod 6). Considering the DRAM, memory requests 14 and 15 are close to each other but distant from memory requests 20 and 21 in terms of memory addressing, which interferes with the access locality of the row buffer. In contrast, the artificial barrier synchronization reduces the contention for cache lines 2, 3 and 5 and mitigates the interference in the DRAM system, as shown in Fig. 4 (b).

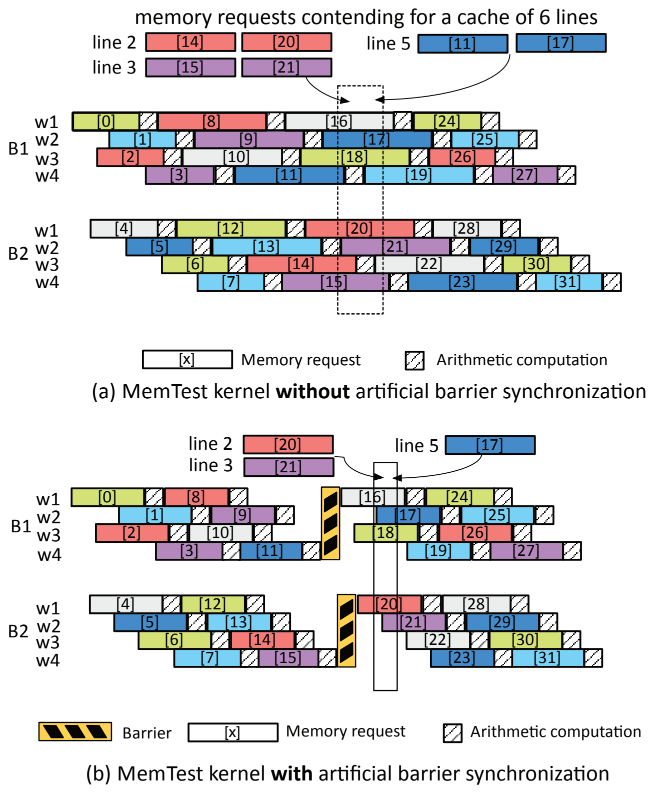

memory requests contending for a cache of 6 lines
line 2 [14] [20]
line 3 [15] [21]
line 5 [11] [17]
B1 w1 w2 w3 w4
B2 w1 w2 w3 w4
[x] Memory request
Arithmetic computation
(a) MemTest kernel **without** artificial barrier synchronization
line 2 [20]
line 3 [21]
line 5 [17]
Barrier
(b) MemTest kernel **with** artificial barrier synchronization

Fig. 4: Illustration of reducing memory contention using artificial barrier synchronization. The MemTest kernel uses two thread blocks, each of which consists of four warps. The dashed-line rectangle in Fig. 4(a) indicates a contention period; the solid-line rectangle in Fig. 4(b) indicates the corresponding contention period after applying artificial barrier synchronization to the kernel. The synchronization case has a shorter execution time than the non-synchronization case because of a reduction in the memory access time, as is the case for memory requests 14 and 20.

TABLE 2: Notation for the execution time analysis.

symbols	Meaning
t	Warp execution time without synchronization
t_s	Warp execution time with synchronization
m	Memory access time without synchronization
m_s	Memory access time with synchronization
c	Computation time
d	Delay time
s	Barrier execution time
n	Number of iterations
H	Memory hit ratio
τ	Time between two consecutive memory access

The data access locality is therefore preserved because of artificial barrier synchronization.

4 A Program Pattern for Artificial Barrier Insertion

The example we presented does not imply that adding arbitrary barriers can accelerate the application performance. In most cases, adding barrier synchronization likely hurts the overall performance. The program pattern presented in Fig. 1 provides a simpler way to guide the use of artificial barrier insertion.

This section presents the analysis of the program pattern and the necessary conditions for inserting barriers into programs for better performance. To clarify the descriptions, Table 2 lists the notation used in this section.

4.1 Analysis of the Program Pattern

The execution time of the program pattern without barrier synchronization can be modeled as

$$t = (m + c) \times n. \tag{1}$$

In contrast, the execution time of the program pattern with barrier synchronization is

$$t_s = (m_s + c + d + s) \times n. \tag{2}$$

The additional barrier synchronization introduces two computational overheads: the delay time (d) to wait for all the warps to reach the synchronization point and the computational cost of the additional barrier instructions (s).

For memory-intensive applications, c and s are relatively small compared with m and m_s. In addition, the delay caused by the barrier synchronization is connected to the execution of instructions, specifically the instructions for memory access. Although GPU devices can hide the memory access time in the computation time, the memory access time dominates the total execution time. Because warps perform memory operations with few and simple arithmetic operations in the program pattern, m or m_s cannot be completely hidden in the computation. The performance difference between the non-sync and sync versions is therefore attributed to the memory access time, which is affected by artificial barrier synchronization.

The memory access time, m or m_s, is modeled as a function of the memory hit ratio H (e.g., the cache hit ratio),

$$\{m, m_s\} = \alpha + (1 - H) \times \beta, \tag{3}$$

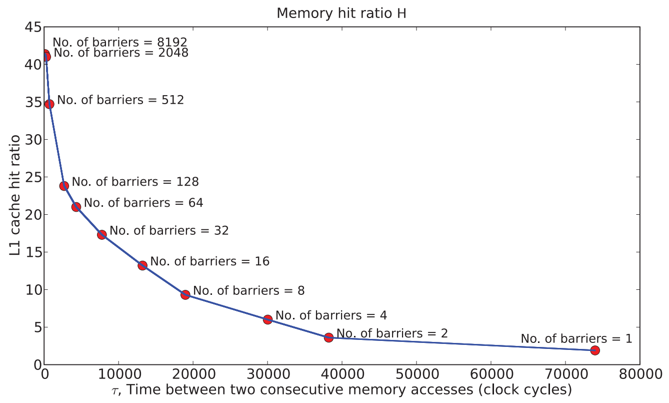


Fig. 5: Relationship between τ and H for the memory-testing kernel on the GTX 480. The results exhibit an effect of locality preservation.

where α is the hit time and β is the miss penalty. The higher the memory hit ratio, the shorter the memory access time.

The memory hit ratio can be further linked with the locality of the consecutive data accesses. Let τ denote the time between two consecutive memory accesses, in which the second memory access hits the pre-fetched data brought by the first access. Assume that the probability of emerging k memory accesses that cause memory misses between those two accesses follows a Poisson distribution with mean λ,

$$P\left(N(\tau) = k\right) = \frac{e^{-\lambda\tau}(\lambda\tau)^k}{k!}, \tag{4}$$

where $N(\tau)$ is the number of emerging memory accesses that pollute the locality of the two memory accesses during the time τ. The memory hit ratio H should be proportional to the probability $P\left(N(\tau) = 0\right) = e^{-\lambda\tau}$. As two warps with data access locality execute diversely (i.e., as τ increases), other memory accesses likely pollute the locality of their memory accesses, thereby decreasing the memory hit ratio and increasing the memory access time.

Fig. 5 shows the relationship between τ and H, as determined via an experiment in which the MemTest kernel with 30 520-thread blocks was executed on the GTX 480. The experiment used 520-thread blocks as an example because two consecutive warps could obtain data from the same 128-byte cache line. We have tried other numbers of threads, and their results are similar. We obtained different values of τ by adjusting the number of iterations in Line 5 and number of repeats in Line 7 in Algorithm 1 to keep 8192 memory accesses per thread but different number of barriers. As illustrated in Fig. 5, the experimental results demonstrate an effect of locality preservation: the smaller the τ, the higher the cache hit ratio.

4.2 Conditions to Apply Artificial Barrier Synchronization

This subsection lists the necessary conditions of the program pattern in Fig. 1 for the insertion of an artificial barrier based on the analysis presented in Section 4.1. As shown in Fig. 1, the program pattern contains three parts in a loop:

1) Threads perform a sequence of device memory accesses. The accessed data originally expose a certain level of locality among threads. As threads issue memory requests independently, their memory requests pollute the data access locality, i.e., H is a decreasing function of τ.
2) Threads perform simple arithmetic or logic computations on the data. The memory access time dominates the total execution time.
3) An additional barrier regulates the warp execution and therefore reduces the interference among threads. Overall, the performance enhancement relies on the condition that the expense of barrier synchronization (i.e., $d + s$) is smaller than the benefit derived from reducing the memory access time (i.e., $m - m_s$).

5 Example Applications

This study uses two CUDA SDK programs and one DNA sequence alignment tool, called MUMmerGPU 2.0 [7], to demonstrate that adding artificial barrier synchronization indeed improves the performance of real-world applications. Our experimental platform contains three GPU devices, the GTX 295, the GTX 480 and the GTX 690. Section 1 of the supplement lists the technical specifications of the two devices. The metric of performance improvement (ρ) is defined as follows:

$$\rho = \frac{(t - t_s)}{t} \tag{5}$$

where t_s is the execution time of an application with artificial barrier synchronization and t is the execution time of an application without artificial barrier synchronization. The execution time recorded for each test was the average obtained over 10 runs.

5.1 CUDA SDK Applications: Vector Addition and Scalar Product

Vector addition and scalar (inner) product computations are basic numerical functions that are widely used in many applications. The vector addition kernel, called VecAdd, loads data, performs additions and then stores the results in the device memory. Similar to the vector addition kernel, the scalar product kernel, called ScalarProd, loads data, performs multiply-and-add instructions and then stores the results in the shared memory. Algorithms 2 and 3 list their pseudo codes.

Because the vector addition kernel in the CUDA SDK supports a small vector size (due to the size limit of a kernel grid), we modified the kernel to allow each thread to perform addition for more than one element. After performing one addition, all of the threads in a block performs barrier synchronization.

For the ScalarProd kernel, the kernel performs the computation for N pairs of vectors concurrently. Each thread block computes the scalar product for one or several pairs of vectors. The memory layout of N pairs of vectors is in two large arrays, each of which comprises consecutive elements of N vectors.

Algorithm 2: VecAdd kernel

```
  input : Two vectors $A$ and $B$
  output: One vector $C = A + B$
1 Let $i$ be the thread index within a kernel grid
2 Let $nIter$ be the number of iterations (elements)
  calculated by each thread
3 Let $dist$ be the distance of the next element to be added
4 repeat
5     C[i] = A[i] + B[i];
6     //*** add __syncthreads() here
7     i = i + dist;
8     nIter--;
9 until nIter=0;
```

Algorithm 3: ScalarProd kernel

```
   input : $N$ pairs of vectors
   output: $N$ scalar product results
 1 Let $shmem$ be a shared memory space of size 1024
   elements;
 2 foreach pair of vectors for the current thread block do
 3     foreach set $s$ of elements for the current thread do
 4         Perform the scalar product for $s$
 5         //*** add __syncthreads() here
 6         Write the result of $s$ to $shmem$
 7     end
 8     Reduce the partial results in $shmem$ by a tree
       reduction
 9     Output the final result for this pair of vectors by
       the first thread within a thread block
10 end
```

5.2 Performance Evaluation of Two CUDA SDK Kernels

5.2.1 *Vector Addition*

We evaluated the effect of artificial barrier synchronization on vector addition by varying the vector length from 2^{16} to 2^{26}, where each thread performs $N/2^{16}$ additions for vector length N. Fig. 6 shows the performance improvements of the vector addition computation yielded by artificial barrier synchronization. As the vector length increases, the performance improvement ρ increases to 22%.

According to the profiling results for the vector addition kernel, the L1 and L2 cache hit ratios are zero because each warp fetches different 128-byte memory segments. Given a vector length, the sync and non-sync versions have the same numbers of DRAM reads and writes, and the same SM (streaming multiprocessor) occupancy. However, the sync version, on average, has a shorter memory access time relative to the non-sync version.

For example, Table 3 lists the memory access time, the time for addition (Line 5 in the VecAdd kernel) and the barrier synchronization (Line 6 in the VecAdd kernel) for a vector length of 2^{26}. (The measured results in Tables 3 and 4 were obtained using a `clock()` command [6], which provides the number of clock cycles required by the device to completely execute a thread.) In this case, the addition

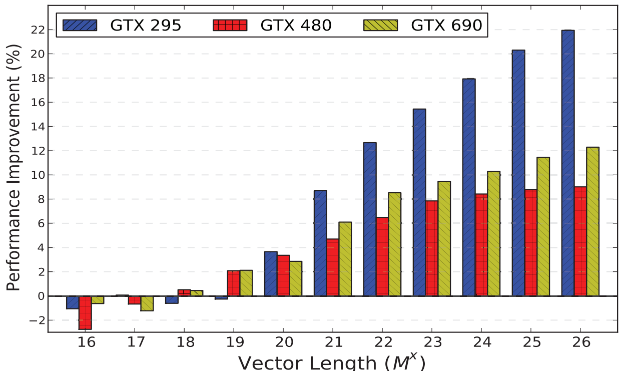


Fig. 6: Performance results for the VecAdd kernel using the barrier synchronization.

TABLE 3: Profile of the VecAdd kernel for vector length 2^{26}.

Operations	GTX 295		GTX 480		GTX 690	
	no-sync	sync	no-sync	sync	no-sync	sync
Memory & Addition (Line 5)	5433	3452	2440	1716	1513	828
Barrier (Line 6)	n/a	555	n/a	527	n/a	503
Reduction ratio (Lines 5–6)	26.3%		8.9%		17.1%	
Improvement ρ	22.0%		8.8%		13.8%	

Unit: GPU clock cycles

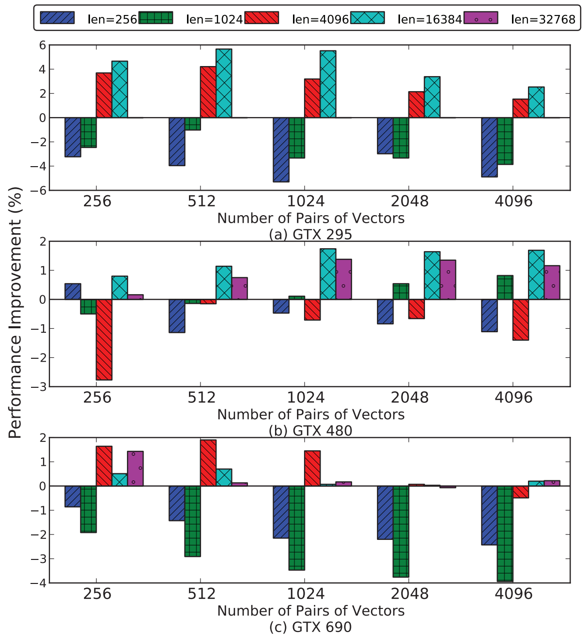


Fig. 7: Performance results for the ScalarProd kernel for regular vectors using barrier synchronization.

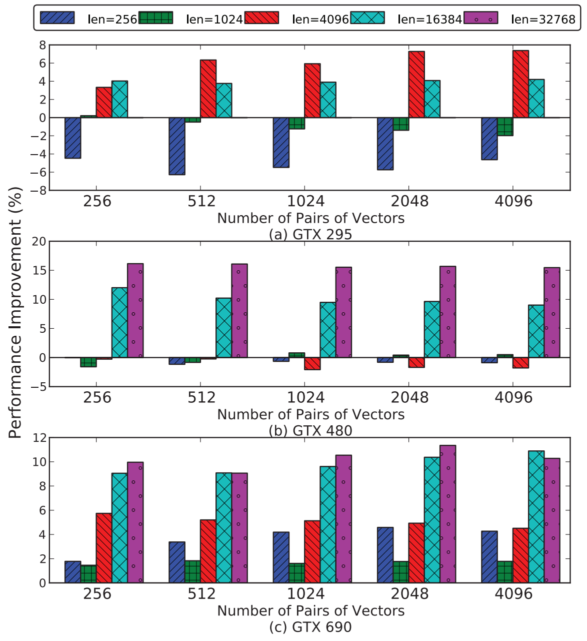


Fig. 8: Performance results for the ScalarProd kernel for irregular vectors using barrier synchronization.

and barrier instructions require only 30–50 and 80–150 clock cycles, respectively, whereas the memory access and delay time dominate the total execution time of the VecAdd kernel. Additionally, Table 3 shows the clock reduction ratios, which correspond to the performance improvements on the GTX 295, the GTX 480 and the GTX 690.

5.2.2 Scalar Product

We evaluated the effectiveness of adding artificial barriers to the scalar product computation using two cases: regular and irregular. In the regular case, all of the vectors have the same length, whereas in the irregular case, the lengths of the vectors can vary. One irregular application is a sparse-matrix vector multiplication [36], in which nonzero elements of a sparse matrix are transformed into irregular vectors. The experimental parameters include the number of vectors and the length of the vectors. In the irregular case, the vectors have a length variance of 20%. Because of the size limitation of the device memory, the maximum length of a vector evaluated on the GTX 480 and the GTX 690 is 2^{15}; the maximum length of a vector on the GTX 295 is 2^{14}. Fig. 7 and 8 show the performance results for the regular and irregular cases.

For the regular case, the additional barrier synchronization yields a small performance increase, but it harms the performance for the cases with shorter vectors, such as for the GTX 295. This study profiled the execution of the ScalarProd kernel for analyzing the above results as follows.

Two primary computational parts in the ScalarProd kernel are the accumulation (i.e., Lines 3–7 in the ScalarProd kernel) and the tree reduction (i.e., Line 8 in the ScalarProd kernel); these components consume most of the total execution time. Table 4 lists the average numbers of clock cycles per thread for the two parts. Barrier synchronization does reduce the time to perform the memory access and the multiple-and-add calculation (denoted as Memory & MAD in Table 4), but the overhead of the barrier diminishes the performance advantage.

It should be noted that barrier synchronization also benefits the tree reduction, which primarily performs tree reduction on shared memory and barrier synchronization to guarantee the data consistency. Table 4 indicates that the performance of the tree reduction part is significantly im-

TABLE 4: Profile of the ScalarProd kernel for 2^{12} vectors of length 2^{15} (GTX 480) and length 2^{14} (GTX 295).

Operations	Regular Case						Irregular Case					
	GTX 295		GTX 480		GTX 690		GTX 295		GTX 480		GTX 690	
	non-sync	sync	non-sync	sync	non-sync	sync	non-sync	sync	non-sync	sync	non-sync	sync
Accumulation (Lines 3–7)	1302092	1295664	2046871	2151725	768774	843731	2526039	2643813	2889167	2730773	1164633	1103736
Memory & MAD (Line 4)	1121866	1063475	1961989	1901816	730065	678805	2331303	2156357	2384287	2237484	1026306	920165
Tree reduction (Line 8)	136478	118574	275709	98779	129124	50668	350016	102731	408093	111333	211422	80701
Reduction ratio (Lines 3–8)	1.7%		3.2%		0.6%		4.7%		16.0%		14.3%	
Improvement ρ	2.2%		1.2%		0.1%		4.1%		15.8%		10.2%	

Unit: GPU clock cycles

proved. This result occurs because the delay overhead, i.e., the time to wait for performing tree reduction, decreases. When the accumulation part uses barrier synchronization, the barrier synchronizes the warps in the accumulation part. All of the warps in the same thread block reach the barrier synchronization of the tree reduction part with slight time differences, resulting in a smaller delay in performing the tree reduction.

For the irregular case, barrier synchronization can create up to 15.8% and 10.2% performance improvements on the GTX 480 and GTX 690, respectively. Two key factors contribute to these improvements. First, because of the variance in the vector length, the barrier stalls threads, which allows them more opportunities to access data that reside in the cache before the data are evicted by other threads. For instance, in the case of 4096 vectors of approximate size 32768, the L1 and L2 cache ratios are enhanced from 13.9% and 14.6% to 24.0% and 19.2%, respectively, according to the hardware profiling results for the GTX 480. Second, as mentioned above, barrier synchronization reduces the delay overhead created in the tree reduction part. For the GTX 295, the performance enhancement occurs primarily because the barrier in the accumulation part reduces the delay overhead in the tree reduction part, as shown in Table 4.

5.3 MUMmerGPU 2.0: A Sequence Alignment Tool

MUMmerGPU, a GPU sequence alignment tool, aligns a set of DNA query sequences to a reference sequence. The alignment tool is used in many practical applications, such as disease genotyping and personal genomics. Based on previous work [37], MUMmerGPU can handle long reference sequences and numerous query sequences.

The alignment kernel in MUMmerGPU, called Align, reports all of the suffixes of query sequences that match the reference sequence for a minimum match length. The reference sequence is preprocessed as a suffix tree and accessed via texture units. Each GPU thread processes a query sequence by traversing the suffix tree until a mismatch is encountered. To avoid redundant searches from the root of the suffix tree, the Align kernel jumps to the node via the suffix link for processing the next suffix string. Note that the Align kernel examines all of the suffix strings of a query from the longest one to the shortest one. Algorithm 4 presents the pseudo code of the Align kernel.

According to the program pattern (in Fig 1), the alignment kernel has two candidate positions to place barriers. The first placement (at Line 8) is after the comparison of the string on an edge and a query, which is performed inside a while-loop execution. The second placement (at Line 11) is before the end of matching one suffix string of a query.

Algorithm 4: Align kernel

```
   input : A block of queries $Q$, the root $r$ of the suffix
           tree, the minimal match length $l$
   output: Alignment results
 1 Let $cur$ be a tree node
 2 Get one query $q$ from $Q$
 3 foreach suffix string $s$ of $q$ and the length of $s > l$ do
 4     if $cur$ is invalid then Set $cur$ as $r$
 5     while exist a child node $c$ of $cur$ that matches $s$ do
 6         Set $parent$ as $cur$ and $cur$ as $c$
 7         repeat match $s$ with $c.edge$ until a mismatch
 8         //*** 1. add __syncthreads() here
 9     end
10     Set $cur$ as the suffix node of $parent$
11     //*** 2. add __syncthreads() here
12     Output the alignment result of $s$
13 end
```

TABLE 5: List of the benchmarks for the Align kernel.

Benchmarks /Genome	Reference Size (Mbp)	Minimum Match Length (bp)	Average Query Length (bp)	Number of Queries
BANTH	5.1	50	100	10^7
CBRIGG	13	100	700	5×10^5
HSILL	16	14	29	5×10^5
LMONO	2.9	20	120	10^6
SSUIS	2	14	36	10^6

5.4 Performance Evaluation of MUMmerGPU

We evaluated the effectiveness of artificial barrier synchronization on the Align kernel using various benchmarks of MUMmerGPU. Table 5 lists the real-world genome references and their default settings [7]. A query generation tool [38] extracts reads (queries) of various lengths from the genome reference. The default number of threads per block is used, and the number of blocks depends on the number of queries. All of the alignment results are verified for correctness. Fig. 9 shows the performance improvements obtained by adding barriers in the first place, in the second place, and in both places.

Overall, by inserting the artificial barriers into the Align kernel, the performance improvement ranges from 0% to

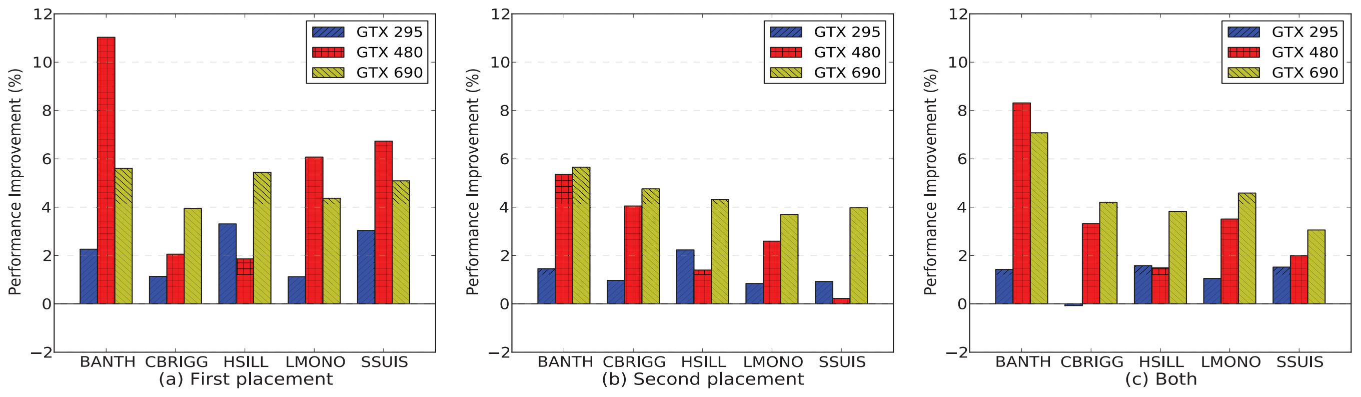


Fig. 9: Performance results for the Align kernel using barrier synchronization.

11%, depending on the benchmarks. Programs with an artificial barrier at the second place typically outperform those with synchronization at the first place. The reason for this difference is that the number of barriers performed is greater when the barrier synchronization is at the first place than when it is at the second place. The first barrier synchronization incurs a greater computational cost. When using both barriers, the program obtains a result that is intermediate between those of the first and second cases.

For the GTX 295, the performance is connected to the efficiency of the DRAM system. In the same benchmark (BANTH), the texture cache hit ratio, approximately 45%, remains the same before and after the barrier is used. The slight performance improvement is attributed to acceleration in the memory access to the query sequences (which are placed in the global memory space). The first placement increases the global memory read throughput from 12.83 GB to 13.01 GB per second; the second placement increases it from 12.85 GB to 13.08 GB per second.

For the GTX 480 and GTX 690, the reduction in execution time corresponds to a reduction in L2 read misses. The L2 read misses result from the global and texture memory requests. We take the benchmark BANTH for the GTX 480 as an example. The cache profiling results demonstrate that the first and second placements enable 10% and 17% reductions, respectively, in L2 read misses. The results indicate that barrier synchronization can alleviate the contention for cache and therefore increase the performance.

6 Conclusions and Future Work

This study used artificial barrier synchronization, which is functionally unnecessary, to enhance the performance of GPU programs. In our model, artificial barrier synchronization relieves contention for the caches and DRAM system and preserves data access locality. The experimental results for three real-world applications proved the effectiveness of adding artificial barrier synchronization to improve memory performance. In addition, this study compared the effects of this technique on different generations of GPU devices and found that the three GPU architectures benefit from the insertion of artificial barrier synchronization under certain conditions.

Although memory contention can be reduced by various methods, barrier synchronization opens a door to mitigating the contention in shared-memory systems. In addition, because GPU barrier synchronization is implemented in hardware, it is a low-overhead solution for programmers to improve the performance of memory-intensive applications.

Our future work includes a number of research directions. First, the exploration of additional program patterns related to barrier synchronization insertion requires systematic approaches. Second, the design and implementation of automatic and intelligent methods to insert barriers into programs is essential to improve the usefulness of this technique. Third, as the number of computing cores that can be packed onto one CPU is increasing, the resource contention caused by massive threads becomes more serious. Similar experiments on CPU memory architectures are worth future research. Finally, other possibilities for relieving the resource contention on GPUs, such as reconfigurable SIMD width, which is a more lightweight approach to synchronize memory accesses for performance, will be included in our future studies.

Acknowledgment

The authors are grateful to NVIDIA's hardware support and would like to thank the National Science Council of Taiwan for financially/partially supporting this research under Contract NSC 101-2115-M-007- 004-MY2. Christopher H. and Roseanna R. are appreciated for their editorial assistance.

References

[1] M. C. Rinard, "Using early phase termination to eliminate load imbalances at barrier synchronization points," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, 2007, pp. 369–386.
[2] R. Gupta, "The fuzzy barrier: a mechanism for high speed synchronization of processors," in *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, 1989, pp. 54–63.

[3] D. Cederman and P. Tsigas, "On dynamic load balancing on graphics processors," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2008, pp. 57–64.

[4] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for gpus," NVIDIA, NVIDIA Technical Report NVR-2008-003, 2008.

[5] M. Burtscher and K. Pingali, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," in *GPU Computing Gems Emerald Edition*, W.-m. W. Hwu, Ed. Morgan Kaufmann, 2011, ch. 6, pp. 75–92.

[6] "Cuda programming guide, 4.2, nvidia." 2011. [Online]. Available: http://developer.nvidia.com/cuda-downloads

[7] C. Trapnell and M. C. Schatz, "Optimizing data intensive gpgpu computations for dna sequence alignment," *Parallel Computing*, vol. 35, no. 8-9, pp. 429–440, 2009.

[8] J. Sartori and R. Kumar, *Low-Overhead, High-Speed Multi-core Barrier Synchronization*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 5952, pp. 18–34.

[9] J. L. Abellan, J. Fernandez, and M. E. Acacio, "Efficient and scalable barrier synchronization for many-core cmps," in *the 7th ACM international conference on Computing frontiers*, 2010, pp. 73–74.

[10] S. Shisheng and H. Kai, "Distributed hardwired barrier synchronization for scalable multiprocessor clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, pp. 591–605, 1995.

[11] T. Vinod, "Fast collective operations using shared and remote memory access protocols on clusters," in *the International Parallel and Distributed Processing Symposium*, N. Jarek and P. Dhabaleswar, Eds., vol. 0, 2003, pp. 84a–84a.

[12] G. Almasi, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of mpi collective communication on bluegene/l systems," in *the 19th annual international conference on Supercomputing*, 2005, pp. 253–262.

[13] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010, pp. 235–246.

[14] A. Rahman, D. Houzet, D. Pellerin, and L. Agud, "Gpu implementation of motion estimation for visual saliency," in *the 2010 Conference on Design and Architectures for Signal and Image Processing*, 2010, pp. 222–227.

[15] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 IEEE/ACM Super Computing*, 2008, pp. 1–11.

[16] S. Xiao and W. C. Feng, "Inter-block gpu communication via fast barrier synchronization," in *the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[17] W. C. Feng and S. Xiao, "To gpu synchronize or not gpu synchronize?" in *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2010, pp. 3801–3804.

[18] D. A. Alcantara, "Efficient hash tables on the gpu," Ph.D. dissertation, University of California, Davis, 2011.

[19] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 423–432.

[20] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007, pp. 25–38.

[21] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, vol. 45, 1736036, 2010, pp. 129–142.

[22] T. Dey, W. Wei, J. W. Davidson, and M. L. Soffa, "Characterizing multi-threaded applications based on shared-resource contention," in *the 2011 IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 76–86.

[23] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for numa-aware contention management on multicore systems," in *the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 557–558.

[24] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Dram-aware last-level cache writeback: Reducing write-caused interference in memory systems," HPS Research Group, The University of Texas at Austin, HPS Technical Report, TR-HPS-2010-002, April 2010.

[25] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 146–160.

[26] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 208–222.

[27] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, 2010, pp. 335–346.

[28] H. Mark, S. Shubhabrata, and O. John D., "Parallel prefix sum (scan) with cuda," in *GPU Gems 3*. Boston MA, USA: Addison-Wesley, 2007, ch. 39, pp. 851–876.

[29] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *the IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[30] Gpgpu-sim. [Online]. Available: http://www.ece.ubc.ca/ aamodt/gpgpu-sim/

[31] M. Breuer and A. Friedman, *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press, 1976.

[32] NVIDIA, "Cuda toolkit 4.2," 2012. [Online]. Available: https://developer.nvidia.com/cuda-toolkit-42-archive

[33] T.-Y. Oh, Y.-S. Sohn, S.-J. Bae, M.-S. Park, J.-H. Lim, Y.-K. Cho, D.-H. Kim, D.-M. Kim, H.-R. Kim, H.-J. Kim, J.-H. Kim, J.-K. Kim, Y.-S. Kim, B.-C. Kim, S.-H. Kwak, J.-H. Lee, J.-Y. Lee, C.-H. Shin, Y. Yang, B.-S. Cho, S.-Y. Bang, H.-J. Yang, Y.-R. Choi, G.-S. Moon, C.-G. Park, S.-W. Hwang, J.-D. Lim, K.-I. Park, J. S. Choi, and Y.-H. Jun, "A 7 gb/s/pin 1 gbit gddr5 sdram with 2.5 ns bank to bank active time and no bank group restriction," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 107–118, 2011.

[34] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Building an efficient hash table on the gpu," in *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2011, ch. 4.

[35] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *the 27th International Symposium on Computer Architecture*, 2000, pp. 128–138.

[36] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 46:1–46:12.

[37] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, no. 1, p. 474, 2007.

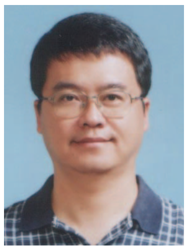
[38] Mummergpu. [Online]. Available: http://sourceforge.net/apps/mediawiki/mummergpu /index.php?title=MUMmerGPU

Shih-Hsiang Lo received his B.S. degree from the Department of Computer Science, National Chengchi University, Taipei, Taiwan, in 2004, and M.S. degree from the Institute of Information Systems and Applications, National Tsing Hua University, Hsinchu, Taiwan, in 2006. He is currently a PhD candidate in the Department of Computer Science at National Tsing Hua University. His research interests are in the areas of GPU computing and high-performance computing, and parallel programming.

Che-Rung Lee received a B.S. and M.S. degrees in Computer Science from National Tsing Hua University Taiwan in 1996 and 2000 respectively, and the Ph.D. degree in Computer Science from University of Maryland, College Park in 2007. He joined the Department of Computer Science at National Tsing Hua University as an assistant professor since 2008. His research interests include numerical algorithms, scientific computing, high-performance computation, and cloud computing. He is a member of IEEE and SIAM.

Quey-Liang Kao received a B.S. and M.S. degree in Computer Science from National Tsing Hua University Taiwan in 2011 and 2012 respectively, and now he is a Ph.D. student in National Tsing Hua University Taiwan. His research interests include numerical algorithms, scientific computing, high-performance computation, and cloud computing.

I-Hsin Chung received the PhD degree in computer science from the University of Maryland, College Park, in 2004, prior to joining IBM Research. He is a research staff member at the IBM Thomas J. Watson Research Center. His research interests include performance tuning, performance analysis, and performance tools. His experience includes designing and developing performance tools on IBM platforms such as IBM Power Systems on AIX and Linux, and the Blue Gene systems. He is a senior member of IEEE.



Yeh-Ching Chung received a B.S. degree in Information Engineering from Chung Yuan Christian University in 1983, and the M.S. and Ph.D. degrees in Computer and Information Science from Syracuse University in 1988 and 1992, respectively. He joined the Department of Information Engineering at Feng Chia University as an associate professor in 1992 and became a full professor in 1999. From 1998 to 2001, he was the chairman of the department. In 2002, he joined the Department of Computer Science at National Tsing Hua University as a full professor. His research interests include parallel and distributed processing, cluster systems, cloud computing, multi-core tool chain design, and multi-core embedded systems. He is a senior member of IEEE computer society.