

Massive Atomics for Massive Parallelism on GPUs

Ian Egielski

Rutgers University
egielski@eden.rutgers.edu

Jesse Huang

Rutgers University
jh3141@gmail.com

Eddy Z. Zhang

Rutgers University
eddy.zhengzhang@cs.rutgers.edu

Abstract

One important type of parallelism exploited in many applications is *reduction type* parallelism. In these applications, the order of the read-modify-write updates to one shared data object can be arbitrary as long as there is an imposed order for the read-modify-write updates. The typical way to parallelize these types of applications is to first let every individual thread perform local computation and save the results in thread-private data objects, and then merge the results from all worker threads in the *reduction* stage. All applications that fit into the *map reduce* framework belong to this category. Additionally, the *machine learning*, *data mining*, *numerical analysis* and *scientific simulation* applications may also benefit from *reduction type* parallelism. However, the parallelization scheme via the usage of thread-private data objects may not be viable in massively parallel GPU applications. Because the number of concurrent threads is extremely large (at least tens of thousands of), thread-private data object creation may lead to memory space explosion problems.

In this paper, we propose a novel approach to deal with shared data object management for *reduction type* parallelism on GPUs. Our approach exploits fine-grained parallelism while at the same time maintaining good programmability. It is based on the usage of intrinsic hardware atomic instructions. Atomic operation may appear to be expensive since it causes thread serialization when multiple threads atomically update the same memory object at the same time. However, we discovered that, with appropriate atomic collision reduction techniques, the atomic implementation can outperform the non-atomics implementation, even for benchmarks known to have high performance non-atomics GPU implementations. In the meantime, the usage of atomics can greatly reduce coding complexity as neither thread-private object management or explicit thread-communication (for the shared data objects protected by atomic operations) is necessary.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—code generation, compilers, optimization

General Terms Performance, Management

Keywords GPU; Atomics; Parallelism; Concurrency

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISMM '14, June 12, 2014, Edinburgh, UK.
Copyright © 2014 ACM 978-1-4503-2921-7/14/06...\$15.00.
<http://dx.doi.org/10.1145/2602988.2602993>

1. Introduction

Parallel applications need to manage shared data object updates efficiently. In many important parallel applications, operations on shared memory objects are commutative and thus the execution order of these operations can be arbitrary [8]. These applications include *map reduce* applications, *machine learning*, *numerical analysis* and *scientific simulation* applications. Parallelism in these applications is typically exploited via the usage of thread-private data structures to hold local computation results. Eventually, a final *reduction* step merges results from all worker threads. In this way, a large task can be split into multiple independent small subtasks that can be executed in parallel. However, this approach may incur large memory space overhead when there are many concurrent threads. This is typically the case in GPU applications, where a kernel function may easily invoke millions of threads.

An alternative parallelization approach is to partition the workload in a way such that the write updates to every data object is at most from one thread. In certain cases, communication among individual threads can be completely eliminated. For instance, in a sparse matrix vector multiplication kernel, the dot product between one entire row in the input matrix and the input column vector results in a single write to one entry in the result vector. Every dot product can be performed by one and at most one thread. However this approach may suffer from the load-unbalancing issue. It is challenging to maintain a balanced workload across all threads when the number of threads is large. In the sparse matrix, the number of non-zero entries in every row may vary a lot. Therefore, if one thread is assigned only one dot product, the amount of multiplication steps (equivalent to the number of non-zero elements in a row) for every thread may vary a lot. It is relatively easier to balance the workload in multi-core architectures that exploit coarse-grained parallelism. This is because we can use a much smaller number of threads and assign multiple dot products to one thread for load balancing. However, the GPU applications exploit fine-grained parallelism and thus it is difficult to achieve load balancing if inter-thread communication is eliminated.

Due to the space explosion and load-unbalancing issues, some prior studies have explored the use of atomic operations in combination with shared data objects with for specific applications [13] [4] [18] or specific memory layer (scratch-pad memory) [6]. The atomic operations help ensure the atomicity of the read, modify and write steps. Therefore, managing the commutative memory updates is purely automatic, requiring no extra algorithmic complexity. The GPU hardware atomic support has also been improved from time to time. For instance, the atomic memory objects can be cached in NVIDIA Fermi GPUs and the L2 cache hit bandwidth is enhanced significantly in NVIDIA Kepler GPUs [3] (for instance by 73% in Geforce GTX680). However, there is a lack of systematic exploration in the application of native atomics to general commutative computations operations. The challenge mainly comes from atomic collision – if multiple concurrent threads attempt to read-

modify-write the same memory location at the same time, then these threads need to be serialized, forfeiting the potential massive parallelism in GPU architectures. It remains unclear whether atomics implementation can be profitable for a more general class of parallel GPU applications, i.e., the applications that already have highly optimized non-atomics implementations. If it is potentially profitable, we need to address the atomic collision challenge.

In this work, we conducted a systematic exploration in the usage of atomic operations for a general class of applications that are abundant in *reduction type* parallelism. We studied the impact of atomic collisions and proposed efficient atomic collision reduction techniques based on two principles. The first principle is to scatter atomic updates to the same memory location at different time intervals to avoid thread serialization. The second principle is to convert atomic collision to parallel computation so that atomic updates can be performed by a few leader threads that do not conflict with each other. In contrast to the traditional perception that atomic operations should mainly be used as synchronization primitives (locks and barriers), our study shows that using atomics for general purpose computation can actually be profitable, in terms of both programmability and efficiency, if used appropriately. We summarize our contributions as follows:

- We discovered that atomic operation can be profitable in parallelizing applications with abundant *reduction type* parallelism, even for parallel GPU applications that already have highly optimized non-atomics implementations, including sparse matrix vector multiplication and parallel summation (reduction).
- We identified two principles for efficient reduction of atomic collisions. Based on these two principles, we designed a set of reduced-collision atomic algorithms. We presented the complexity of our algorithms and discussed the applicability of these algorithms in different scenarios. We built a statistical learning model to choose over the non-atomics, naive atomics and reduced-collision atomics implementations given different atomic collision levels.
- We built a library that implemented these atomic collision reduction algorithms. The functions in this library have simple interfaces. In most cases, programmers can call these functions in a way similar to calling regular native atomic functions.

The rest of the paper is organized as follows. In Section 2 we give the background on GPU programming and present a motivation example. In Section 3, we discuss the performance aspect of generally using atomic operations in computation. We present detailed algorithms for eliminating atomic collision. In Section 4, we present evaluation results. We describe related work in Section 5 and conclude in Section 6.

2. Motivation

Background Throughout this paper we use NVIDIA CUDA terminology to describe the GPU architecture and programming model. A GPU is made up of multiple Streaming Multiprocessors (SMs). An SM is a Single Instruction Multiple Data (SIMD) like processor in which a group of threads execute the same instructions on multiple data elements. A GPU program includes both CPU code and GPU code. The function that runs on the GPU is called a kernel function. A kernel function is typically launched by more than thousands of threads. These threads are divided into small groups to be executed and scheduled on different SMs. Each group is called a thread warp. There is implicit synchronization within a thread warp – in which threads execute in lockstep.

There are two major types of memory on a GPU – on-chip memory and off-chip memory. On-chip memory includes registers, cache, and shared memory. Scratch-pad memory is nearly as

fast as cache memory though it has to be explicitly managed by the programmer. Scratch-pad memory is called *shared memory* in NVIDIA terminology. The off-chip memory of an NVIDIA GPU is partitioned into *global memory* for heap objects, *local memory* for local variables, *constant memory* for constant objects and, *texture memory* for multi-dimensional read-only data objects. We primarily use *shared memory* and *global memory* for our atomics study.

We define *atomic collision* as the event of multiple threads attempting to update the same data object at the same time. These memory updates must be serialized due to the atomicity property of atomic operations. The shared memory atomic instruction is implemented as a loop that atomically updates unique unlocked memory locations at each iteration until one thread warp’s requested atomic memory updates are complete. The number of iterations for a warp is the number of times the most frequent memory location is accessed by threads in the whole thread warp. Common atomic operations include `atomicAdd`, `atomicOr`, `atomicAnd`, `atomicMin`, etc. We focus on the commutative atomic operations.

Sparse Matrix Vector Multiplication Example As a motivation example, we first show the atomics-based implementation of a real application. We then compare the performance between the traditional non-atomics implementation and the atomics-based implementation. We use a highly optimized sparse matrix vector multiplication kernel from CUSP [5], which is an open source C++ library of generic parallel algorithms for sparse linear algebra and graph computations on GPUs. We show the main computation code in the left half of Fig. 1. In this implementation, a thread warp iterates over one or multiple consecutive rows in the input sparse matrix represented by $V [j]$ (at line 1), multiplies every non-zero element by the corresponding element in the input column vector (at line 4), performs segment reduction on the multiplication results and updates the output column vector correspondingly (at lines 5-20). If the last thread in a thread warp at previous iteration obtains the multiplication value to be added to the same output vector entry, then the value is carried into the current iteration (lines 7-8). Otherwise the value from the last iteration needs to be added to the corresponding output vector entry (at lines 9-10). The control flow is relatively complex here since the thread warps may cross input matrix row boundaries.

In the right half of Fig. 1, we show the atomics-based implementation. The multiplication code is exactly the same (at line 4). The difference begins from line 5. We atomically add the multiplication result into the corresponding output vector entry $y[row]$. There is no thread divergence in the code and the total number of lines of code is significantly reduced. Although not included in the code snippet, note that the non-atomics version uses shared memory to hold intermediate reduction results. The `rows[]` and `vals[]` arrays are shared memory arrays. However, our naive atomics implementation does not use *shared memory*.

In Fig. 2, we show the performance comparison results. The experiment is performed on an NVIDIA GTX 680, which is an NVIDIA Kepler card, we denote as *kepler*, and a NVIDIA Tesla C2075, which is a NVIDIA Fermi card, we denote as *Fermi*. We use the non-atomics implementation as baseline and the y axis represents are speedup. If the speedup is greater than 1, *atomics* version is faster, otherwise the *non-atomics* version is faster. Every bar in this graph corresponds to an input matrix. We use the sparse matrix input from matrix market [2] – the standard pool of sparse matrix in real world applications which is typically used for benchmarking sparse matrix programs. In Fig. 2, we can see that, surprisingly, the *naive* atomics implementation is faster for some case(s) on both Fermi and Kepler despite the atomic collisions. For Fermi, in most cases, the atomics version runs slower than the non-atomics version. One reason is that the original non-atomics version used shared memory to store intermediate results, while the naive *atom-*

Non-atomics Implementation

```

1: for(IndexType n = interval_begin + thread_lane; n < interval_end; n += WARP_SIZE)
2: {
3:   IndexType row = I[n]; // row index (i)
4:   ValueType val = V[n] * fetch_x<UseCache>(J[n], x); // A(i,j) * x(j)
5:
6:   if (thread_lane == 0)
7:   {
8:     if(row == rows[idx + 31])
9:       val += vals[threadIdx.x + 31]; // row continues
10:    else
11:      y[rows[idx + 31]] += vals[threadIdx.x + 31]; // row terminated
12:
13:    rows[idx] = row;
14:    vals[threadIdx.x] = val;
15:
16:    if(row == rows[idx - 1]) { vals[threadIdx.x] = val + val + vals[threadIdx.x - 1]; }
17:    if(row == rows[idx - 2]) { vals[threadIdx.x] = val + val + vals[threadIdx.x - 2]; }
18:    if(row == rows[idx - 4]) { vals[threadIdx.x] = val + val + vals[threadIdx.x - 4]; }
19:    if(row == rows[idx - 8]) { vals[threadIdx.x] = val + val + vals[threadIdx.x - 8]; }
20:    if(row == rows[idx - 16]) { vals[threadIdx.x] = val + val + vals[threadIdx.x - 16]; }
21:
22:    if(thread_lane < 31 && row != rows[idx + 1])
23:      y[row] += vals[threadIdx.x]; // row terminated
24:  }

```

Atomics Implementation

```

1: for(IndexType n = interval_begin + thread_lane; n < interval_end; n += WARP_SIZE)
2: {
3:   IndexType row = I[n]; // row index (i)
4:   ValueType val = V[n] * fetch_x<UseCache>(J[n], x); // A(i,j) * x(j)
5:   atomicAdd(y+row, val);
6: }

```

One line of code in atomics version.
16 lines of code in the non-atomics versions.

Figure 1. Atomics based SPMV implementation v.s. non-atomics based SPMV implementation

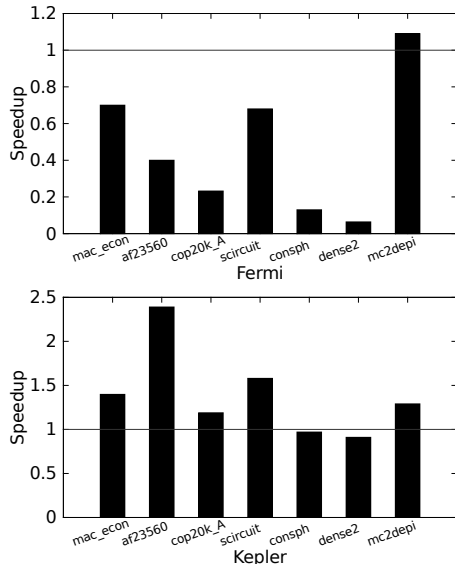


Figure 2. Performance of naive atomics implementation on Kepler and Fermi

ics implementation illustrated in Fig. 1 uses global memory, which is much slower. The other reason is the atomic collision incurred by threads operation on elements at the same row of the matrix. For the Kepler card, we already saw performance improvement for most benchmark matrices. Only two of them have performance slow-down. The atomics hardware support on Kepler has been improved significantly compared to the Fermi generation (especially for multiple atomic updates to single memory locations [3]). Further, the thread divergence is significantly eliminated in the naive atomics version, which also contributes to the performance improvement [19]. The experiment results demonstrated the potential of using atomic operations for programs with abundant *reduction type* parallelism even without applying any atomic collision elimination techniques. In the following Section 3, we show that with appropriate collision-removal techniques, the already atomics-based implementation can be further improved.

3. Performance

In this section, we discuss the performance aspect of applying atomic operations extensively for computing purposes. Atomic collision may degrade atomic performance significantly. We describe our approaches that reduce atomic collisions. There are two major types of approaches. The first type of approach is based on converting atomic collision into computation. This approach leverages local parallel reduction and reduces atomic memory operations. The second type of approach is based on re-scheduling atomic memory operations so that we can scatter potential atomically conflicted accesses over different time intervals. The basic idea is that if the atomic updates of the same memory addresses from different threads are scheduled at the same time, they collide with each other; however, if these threads are scheduled to run at different time intervals, they do not collide with each other. GPU programs typically launch a lot more threads than the number of physical GPU cores for maximal concurrency. Therefore, there is great potential to schedule threads for minimal atomic collisions. We name the first type of approach as *atomic-collision-to-computation*, and the second as *atomic-collision-to-scatter*. We describe the first approach in Section 3.1 and the second one in Section 3.2.

3.1 Convert Atomic Collision To Computation

With the *atomic-collision-to-computation* approach, we first perform local parallel reduction for the threads that atomically access the same memory location. We then let one thread perform atomic updates to every unique memory address based on the local reduction result. We present the design of the *atomic-collision-to-computation* algorithm in two scenarios: (1). the threads that access the same address are placed next to each other as illustrated in Fig. 3 (a), which happens frequently in kernels with structured parallelism such as matrix/vector computation in linear algebra libraries [12]; (2) the threads that access the same memory locations are not placed next to each other as illustrated in Fig. 3 (b), which is the case for kernels with unstructured parallelism such as graph traversal kernels. We refer to the first scenario as the *clustered-collision* case and the second one as the *non-clustered-collision* case.

Clustered-collision Case: One of the major difficulties of the atomic-collision-to-computation approach is that the number of threads accessing every unique memory address may not be uni-

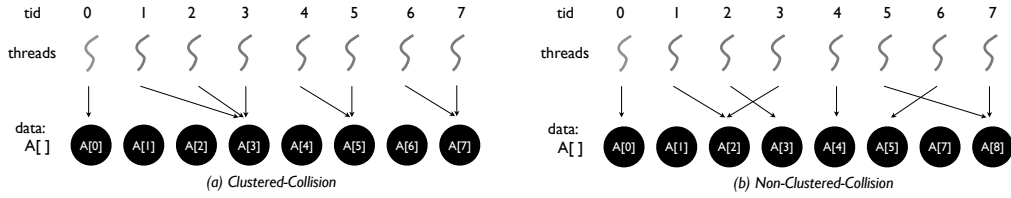


Figure 3. Atomic Collision Pattern

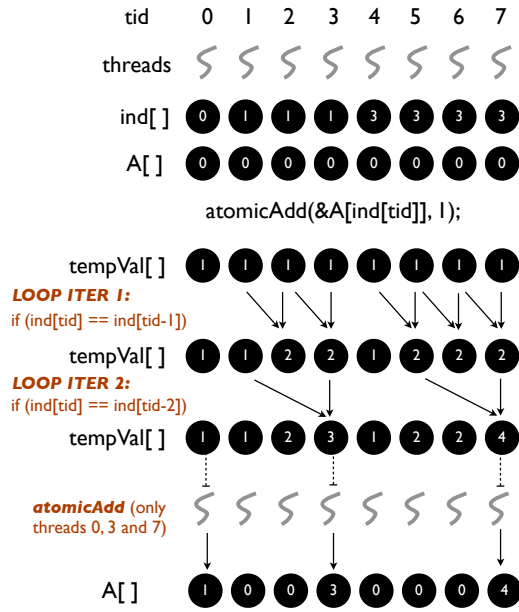


Figure 4. An example of atomic scan and reduce approach

form. Thus the number of local parallel reduction steps for every unique memory address may also vary significantly. Further, the thread boundary for local parallel reduction, the first and the last thread for one unique memory address, is usually not determined during static time. An example is the *sparse matrix vector multiplication* kernel we discussed in Section 2, in which the threads that process the non-zero elements at the same input matrix row may span across different thread warps and blocks. In a word, it is challenging to detect the size and boundary of every thread group that performs local parallel reduction for a unique memory address.

We propose an algorithm for the *clustered-collision* case that can perform the minimal number of local parallel reduction steps and can detect thread boundaries for unique memory addresses at the same time. In this algorithm, within every thread, we check the current thread and the thread with a logical id that is smaller than the current thread by a power of 2. If the two threads' memory addresses are the same, then we perform partial reduction on these two threads, and store the result in the current thread's temporary storage. If the two threads' memory addresses are different, we do not perform any action since we have already crossed the boundary of the thread region corresponding to the same unique memory address. Start from the thread distance of 2^0 , we repeat this process and double the distance at every iteration until we have finished local parallel reduction for the largest thread region.

We describe the detailed algorithm in algorithm 1. We illustrate the algorithm with atomic addition operations, which applies to

Algorithm 1 Atomic scan and reduce algorithm

```

1: procedure ATOMICADDSR( $A[], idx, val, N$ )
2:   // Naive atomic version: atomicAdd(&A[idx],val)
3:   // Allocate temporary storage for addresses and values
4:   alloc tempVal[]; // One value per thread
5:   alloc tempIdx[]; // One integer per thread
6:   alloc tempMask[]; // One integer per thread warp
7:   tempIdx[tid] = idx;
8:   tempVal[tid] = val;
9:   sync(); // Barrier at a given thread scope
10:  tempMask[warpid] =
    ...ballot(tempIdx[tid]==tempIdx[tid-thresh]);
11:  if (tempMask[1:lastWarpid] == 0) then
12:    // If all masks are 0s, use naive atomic and return
13:    atomicAdd(&A[idx], val);
14:    return ;
15:  end if
16:  for  $i = 1$  to  $\log_2 N$  do
17:    if (tempIdx[tid] == tempIdx[tid-2i] && tid  $\geq$  2i) then
18:      tempVal[tid] += tempVal[tid-2i];
19:      sync();
20:    else
21:      sync();
22:      break;
23:    end if
24:  end for
25:  if (tempIdx[tid]  $\neq$  tempIdx[tid+1] && tid  $\leq$  N-2) then
26:    atomicAdd(&A[idx], tempVal[tid]);
27:  end if
28:  sync();
29: end procedure

```

other commutative atomic operations as well. Programmers who want to convert atomic collision to computation can simply replace the original atomic add function *atomicAdd* with this *atomicAddSR* function. The parameters of *atomicAddSR* include the array pointer A , the index idx and the value val , which means the array cell $A[idx]$ needs to be atomically increased by val . The parameter N is the number of threads on which local parallel reductions need to be performed. Note there might be more than one local parallel reduction thread regions within the N threads. This function is executed by every individual thread. In this algorithm, we synchronize after every local (partial) parallel reduction step to make sure the memory updates are visible to all threads within the N threads (at line 19 and 21). If local parallel reduction is performed within every thread warp, the barrier synchronization statement is not necessary since thread warp level synchronization is implicit. We ensure that local parallel reduction is only performed by threads that access the same unique memory address by the condition check at line 17. This can be nicely integrated with the local parallel reduction steps as the number of condition checks is also logarithmic. If the two threads do not pass the condition check at line 17, then we know we crossed

the boundary of the thread region corresponding to the same memory location, we break out of the loop at lines 20-22. Henceforth, the number of local parallel reduction steps for every unique memory address is logarithmic with respect to the thread group size. When all local parallel reduction operations are done, we break out of the loop from line 16 to 24. Therefore we ensure the minimal number of computation steps with this approach. Note that we only perform local parallel reduction if there are at least *thresh* threads which access the same memory location. It is performed with the help of a voting function at lines 10-15. We will describe the selection of the *thresh* variable in *Performance Guarantee* discussion in this section. We will discuss the voting function in more detail at the *Non-Clustered-Collision Case* scenario.

We illustrate the use of our algorithm in Fig. 4. In this example, every thread tries to atomically add a value 1 to a cell in array *A* with the index of *ind[tid]*. The *tid* variable represents the thread index. Every cell in *A[]* array is 0 initially as illustrated in Fig. 4. We also show the values of every cell in array *ind[]*, which corresponds to the location of the array cell every thread needs to access. Assume there are 8 threads. According to *ind[]*, the first thread needs to increment *A[0]* by 1, the second to the fourth thread need to atomically increment *A[1]* by 1, and the fifth to the eighth thread need to increment *A[3]* by 1. The optimal case is to perform 2 local reduction steps for threads 2-4 and threads 5-8, and no reduction for thread 1. We achieve this by using the condition at line 17 in Alg. 1. At *loop iteration 1* in Fig. 4, we only perform addition if the current thread and its precedent thread have exactly the same memory access. Then we save the partial reduction result in temporary array cell *tempVal[tid]* for each thread. At *loop iteration 2*, we perform the same check and partial reduction operations except that we check the current thread and the thread that has a *tid* smaller than the current thread by 2. After two loop iterations, we get the summation results for every element and the elements to its left that correspond to the same memory in array *tempVal[]*. In the last step, we perform atomic add only with thread 0, thread 3 and thread 7 to *A[0]*, *A[1]* and *A[3]* and array *A[]* is completely atomically updated. Our algorithm eliminates atomic collision completely and needs the minimal number of local parallel reduction steps.

Non-Clustered-collision Case: In this case, not only is the amount of atomic collision irregular, but also how it is distributed among threads is irregular – the threads that access the memory address are not necessarily placed next to each other, which makes local parallel reduction boundary even more challenging. We propose an algorithm that identifies the most frequently accessed memory locations with the maximum likelihood. Then we obtain the frequency of these memory addresses in one pass or $\log(N_{\text{warp}})$ passes, with N_{warp} being the number of thread warps. Based on this, we determine if it is necessary to perform local reduction and if so how many local reduction steps is necessary. We name this approach as *atomic vote and reduce*.

We describe the *atomic vote and reduce* algorithm in Alg. 2. Given a local reduction scope (whether it is within every thread warp or every thread block or among all the threads), we first randomly pick a thread and obtain the thread’s memory access location (lines 10-11). Then, we let all threads vote to find out which threads access the same memory address (line 12). Thus, we get the thread access frequency of this memory address. If it is above a threshold (the *thresh* variable in Alg. 2), we perform a local reduction on the corresponding threads and let one leader thread write the local reduction result back atomically (lines 14-17). Otherwise, we use the naive atomic operations (lines 21-22). Similarly, this approach not only applies to atomic addition but also other commutative atomic operations. The barrier synchronization instruction at line 9 is not necessary if we perform atomic vote and reduce at thread warp level as synchronization within a thread

Algorithm 2 Atomic vote and reduce algorithm

```

1: procedure ATOMICADDVR1(A[], idx, val)
2:   // Naive atomic version: atomicAdd(&A[idx],val)
3:   // Allocate temporary storage for addresses and values
4:   alloc tempVal[]; // One value per thread
5:   alloc tempIdx[]; // One integer per thread
6:   alloc tempMask[]; // One mask per thread warp
7:   tempIdx[tid] = idx;
8:   tempVal[tid] = val;
9:   sync(); // Thread barrier at a given thread scope
10:  rid = rand[(scopeid + kbase)% rand_cycle]; //randomly
    selected thread id
11:  sampleIdx = tempIdx[rid];
12:  tempMask[warpid] = _ballot(idx == sampleIdx);
13:  if (freq(tempMask[1:lastWarpid]) ≥ thresh) then
14:    lsum = localReduce(); // Local parallel reduction
15:    if (scopeTid == 0) then
16:      atomicAdd(&A[sampleIdx], lsum);
17:    end if
18:    if (idx ≠ sampleIdx) then
19:      atomicAdd(&A[idx], val);
20:    end if
21:  else
22:    atomicAdd(&A[idx], val);
23:  end if
24: end procedure

```

warp is implicit. In Alg. 2, we illustrated the *atomic vote and reduce* algorithm with the case where we want to pick the most frequently accessed memory address and perform local reduction. We name it as *atomicVR1()*. We also extend it to the case where we want to pick top *m* most frequently accessed memory addresses. We use a loop to sample the top *m* most frequent addresses. The local reductions of these *m* unique addresses happen in parallel. In practice, we find *atomicVR1()* at the thread warp level is most helpful. If the amount of atomic collision is excessive at thread block level or among all running threads, then the non-atomics code version usually outperforms the atomic version (even the reduced atomic collision version). We use a regression tree model described in Section 3.3 to identify different atomic collision level.

We use the random sampling and voting based collision detection approach for the following reason. Given a set of memory addresses accessed by one thread group (group is the scope we mentioned above), the more frequent a memory address appears, the more likely the thread that accesses it will be selected if we draw one thread from this thread group with uniform probability distribution. As the number of thread groups and the number of sampling points increase, the most frequent memory address can be obtained with maximum likelihood according to the *large number theorem*. The number of threads launched by a GPU kernel is typically large which ensures the efficiency of the random sampling and voting based approach. For random number generation, we do not directly use a random number generator during runtime. Instead, we generate a large sequence of random numbers with a full cycle random number generator. We save the sequence in memory. Whenever a kernel is invoked, we pick a location in the sequence randomly as the seed position. The thread groups then use the random number sequence in a round robin fashion. The first thread group in the GPU kernel uses the random number in the seed position. The second thread group uses the number immediately after the seed position. The following thread groups get their random number in a similar fashion and when they hit the end of the sequence, the first random number in the sequence is selected. This

process is described at line 10. The *kbase* is the seed position determined when the kernel is invoked. We use the system time mod the random number cycle to obtain the seed position in the random sequence. Every time the kernel runs, it starts with a different seed position. Therefore the randomness across all thread groups is ensured in a lightweight fashion (no dynamic random number generation is necessary).

Performance Guarantee We choose the **thresh** variable in Alg. 1 and Alg. 2 in a way that guarantees no performance degradation if compared to the naive atomic implementation. Assume the total number of threads in the thread group is N , the maximal number of threads that access the same memory location is x , and the initial setup overhead is s (the voting, boundary checking and counting in Alg. 1 and Alg. 2). We determine the threshold value by finding the minimal of x in the following inequalities:

$$\log_2 x + s \leq x(\text{atomic scan and reduce}) \quad (1)$$

$$(N - x) + \log_2 x + s \leq x(\text{atomic vote and reduce}) \quad (2)$$

In the above inequalities, the left-hand side estimates the total number of computation steps needed if we perform the local reduction, and the right-hand side represents total number of computation steps if we use the naive atomic implementation. For *atomicVR*, in the left-hand side, the $N - x$ component represents, after local parallel reduction on the set of threads that access the most frequently accessed address, the number of extra atomic computation steps needed in the worst scenario (assuming all other threads collide at the same memory address). For both *atomicVR* and *atomicSR*, the $\log_2 x$ component represents the number of computation steps needed for local parallel reduction. The s component represents initial set up overhead. We first change the \leq to $=$ and solve this nonlinear equation for x . Then we set the threshold variable **thresh** as the x value we solved. Following this inequality, we guarantee that our transformed atomics code does not run any slower than the naive atomics code. We obtain the overhead s by checking number of binary instructions that are needed to implement the initial setup and normalize it with respect to the latency of the binary instructions needed to perform every step of local reduction.

3.2 Atomic Collision to Scatter

In the *atomic-collision-to-computation* approach, we perform local reduction operations before we perform any atomic updates. In this section, we present an approach that does not use extra local reduction steps. The basic idea is that two different threads accessing the same unique memory address do not collide atomically if they are scheduled to run at different points in time. Since GPU programs typically launch a lot more threads than physical GPU cores, we can schedule these threads to scatter the potentially conflicting atomic operations over time.

We propose an algorithm that achieves the scattering of potentially conflicting atomic accesses through thread layout transformation. The essential idea is to separate the threads that have conflicting memory accesses from each other as far as possible. The logical thread layout implies the thread co-running pattern; for instance, every thread in a thread warp or a thread block runs at the same time at one SM. The threads from different thread blocks may run at different time intervals, as the GPU hardware only support a limited number of active thread warps at one time. The thread warps are divided into batches, with the size of each batch being the maximal number of active thread warps the GPU can support. These thread batches run sequentially [1]. To enable scattering, we first group threads by the memory addresses they access. We construct one set of threads for every unique memory address. We then reorder threads based on these sets and create a new thread layout. Starting from the first set of threads, we take out one thread

and place it as the first thread in the new thread layout. Next we pick a thread from the second set and place it as the second thread in the new thread layout. We repeat this step and keep appending threads to the new thread layout. If we reach the last set of threads, we restart from the first set. The process stops until we remove all threads from the sets. Finally, we obtain a new order of logical threads. For clustered-collision case, since the threads have already been grouped according to their memory access addresses the set creation step is omitted. For non clustered-collision case, both the set creation step and the reordering step are necessary. If the scattering overhead is non-trivial, we run the scattering algorithm on CPUs. We can overlap the actual computation on GPU with the scattering process on CPU. If necessary, we use the kernel spilling technique described in [19] to enable the overlapping and overcome dependence. Further, we can regroup and reorder thread warps instead of threads. For non-clustered collision cases, we obtain the most frequently accessed address in one warp using the random sampling approach described in Alg. 2 and we label the thread warp with this address. For the clustered case, we associate every thread warp with its first thread’s address since threads are already grouped. These two overhead reduction techniques help us achieve performance improvement or at least no performance degradation by making the thread layout transformation overhead transparent. The algorithm for scattering is illustrated with pseudo-code in Alg. 3.

Algorithm 3 Atomic scatter algorithm

```

1: procedure ATOMICSS(tAddr[], newLayout[], tNum)
2:   //Obtain sets of threads that access unique addresses
3:   wSets = groupThreads(tAddr[]);
4:   i = 0;
5:   while (i < tNum) do
6:     for each set  $\in$  wSets do // In address ascending order
7:       if ( set.size  $\neq$  0 ) then
8:         tId = set.pop();
9:         newLayout[i] = tId;
10:        i++;
11:       end if
12:     end for
13:   end while
14:   return newLayout[];
15: end procedure

```

3.3 Atomic Collision Sampling

Eliminating atomic collision incurs overhead. If there is little atomic collision, there is little benefit in eliminating atomic collision. We should use naive atomics directly in these cases as the overhead for detecting and eliminating collision is non-trivial. If there is too much atomic collision, we should use the non-atomics code version or the reduced-collision atomics. We need to determine for every given program input: (1) whether the atomics code version should be used; (2) if so, whether the reduced atomic collision version or the naive atomics version should be used. Although we can perform a dynamic collision check with the voting approach described in Section 3.1, the check overhead s in inequalities 1 and 2 needs to be incurred regardless of the fact whether atomic collision needs to be reduced or not. We propose statistical learning techniques to model the relationship between the collision statistics and the decision on whether and which atomic code version should be used. Specifically, we use regression tree model [10] for its simplicity and good interpretability. We use three major collision parameters: (1) intra-warp collision level (2) intra-block collision level (3) all-thread collision level. We first define the *maximal collision factor* as the ratio between the maximal access frequency of

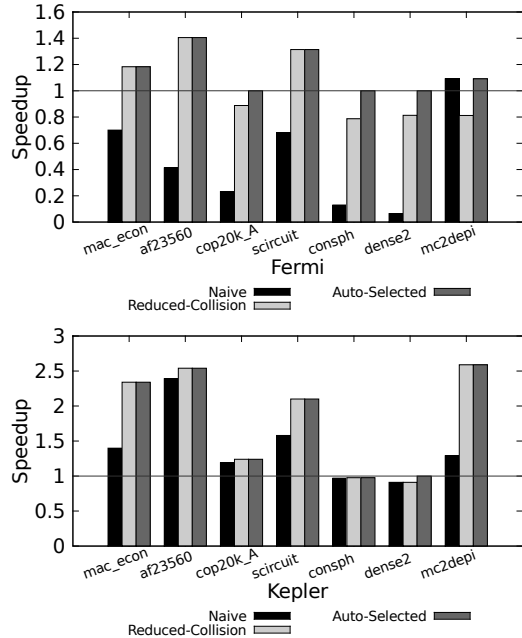


Figure 5. Performance results of sparse matrix vector multiplication after eliminating atomic collision

every unique memory address in a warp/block and the total number of threads in a warp/block. The maximal collision factor can be obtained using the voting step we discussed in Alg. 2. It is lightweight as the process can be easily parallelized by many threads. We define the intra-warp/intra-block collision level as the average of all thread warps’/blocks’ maximal collision factors. We choose maximal collision factor because it is the minimal amount of time the whole thread warp/block needs to finish all atomic updates. The all-thread collision level is the average access frequency of all unique memory addresses. We use a training set of inputs, calibrate a database that stores tuples mainly consisting of the three collision parameters and the corresponding best code version (naive atomics or reduced-collision atomics or no atomics), and build the regression tree model. Given the collision stats of a new program input, we feed it as input to the statistical learning model and outputs the code version to be used. We may also use other factor of a program to build the regression tree model such as the scale of a program. If a program runs a small number of threads such that the running time is trivial, it may not be necessary to perform any optimization.

Discussion The atomicSR, atomicVR and atomicSS algorithms can be applied at different scenarios. For the programs that have inherent clustered-collision pattern such as sparse matrix multiplication, we should always use atomicSR. For the programs that have inherent non-clustered-collision pattern such as image histogramming or graph traversal, we should use the atomicVR algorithm. Both atomicSR and atomicVR algorithms has less complexity than the atomicSS algorithm since it needs to scatter conflicted atomic memory addresses. Also the atomicSS algorithm might incur larger transformation overhead, however it also has greatest potential to improve performance since no extra local reduction needs to take place. Overall, the combination of atomicSR+atomicSS can achieve relatively low overhead and satisfactory performance improvement. The atomicVR is best not to be used in combination with atomicSS since the atomic memory access is likely randomly distributed for non-clustered collision case unless the distribution

of collision statistics is degenerate (which can be detected through the statistical learning model).

We present the performance results of the sparse matrix multiplication example used in Section 2 after we apply the optimization techniques in this section. The sparse matrix vector multiplication program has an inherent clustered collision pattern. Therefore, we apply a combination of the atomic scan/reduce algorithm (atomicSR) and the atomic scattering algorithm (atomicSS). In Fig. 5, we show the naive atomics version, the atomicSR+atomicSS and the final selected version based on the statistical learning model for both Fermi and Kepler. The left bar in every group represents the performance of naive atomics. The middle bar represents the performance after we reduce atomic collisions. The right bar represents the selected version with the statistical learning model. The baseline is the original non-atomics implementation. The graph shows that these techniques improved performance significantly. For Kepler, though naive atomics version is already fast compared to the non-atomic version for a number of benchmarks, reduced-collision atomics can make them even faster. The *mac_econ*, *scircuit* and *mc2depi* matrices have up to 60% improvement. For Fermi, since its intrinsic atomic speed is slower than Kepler [3], the reduced-collision atomic implementation make a big performance difference when compared to the naive atomics implementation almost for every benchmark. Comparing the reduced-collision atomic implementation to the non-atomic implementation, four out of seven cases are better. Three others are slightly worse. That is because these three benchmarks *cop20k_A*, *dense2* and *consph* are relatively denser matrices. The final version selected by the statistical learning model for these matrices is the non-atomics version (with the speedup as 1 in these bars).

4. Evaluation

In this section we evaluate the performance of our atomic algorithms described in section 3 with various important and practical kernels. We conduct our experiments on two GPUs with different hardware atomic instruction latencies. One is an NVIDIA Kepler GPU card – GTX680 with CUDA computing capability 3.0. It has 8 streaming multiprocessors with 192 cores on each of them. There are 65536 registers and 48KB shared memory on each SM. The other one is an NVIDIA Fermi GPU card – Tesla C2075 with CUDA computing capability 2.0. It has 14 streaming multiprocessors (SM) with 32 cores on each of them. Each SM has 32768 registers and 48KB of shared memory. The hardware atomics speed in Kepler card is improved over the Fermi card [3]. Both host machines run 64-bit Linux with kernel version 3.1.10 and CUDA 5.5.

For each benchmark, we have collected data for the original non-atomics implementation, the naive atomics implementation and the reduced-collision atomics implementation. We denote our atomic-collision-to-computation function for clustered-collision case – the “atomic scan and reduce” function as *atomicSR*, and the atomic-collision-to-computation function for non-clustered-collision case – the “atomic vote and reduce” function as *atomicVR*. They are abbreviated as *SR* and *VR* respectively in figures. We denote our atomic-collision-to-scatter function “atomic set scatter” as *atomicSS*, abbreviated as *SS*. The default naive atomic implementation is denoted as *AA*.

As of benchmark applications, we use five important kernels that are commonly used in various applications. They are histogramming [13], merge-sort [14], page-view-count [11], parallel summation [14], and sparse matrix vector multiplication [5]. All benchmarks are highly optimized GPU kernels. They are obtained either from the CUDA linear algebra library for sparse matrices (CUSP) [5], the CUDA SDK [14], or published papers [11]. We describe the features of these five benchmarks as follows:

- **Image Histogramming** Image processing applications extensively use the *histogram* kernel, which counts the frequency of each color for all pixels taken from an input image. We used the image histogramming benchmark optimized by the authors in [13]. The authors provided a warp-private histogram implementation which uses warp-private histograms to store local histogramming results, and a thread-private histogram implementation which does not use any atomics. We extended this benchmark by adding a block-private histogram and a global histogram implementation (which uses no private histogram) in order to test atomic-collision reduction techniques at different memory levels. Both block-private and warp-private histograms are stored in shared memory. The no-private implementation uses a global histogram in device memory for all threads.
- **SPMV** Sparse matrix vector multiplication is from the CUSP [5] library, an open source C++ library of parallel algorithms for sparse linear algebra and graph computations on GPUs. This kernel uses shared memory to enhance performance. Its GPU implementation is much faster than the CPU implementation. In our naive atomics implementation mentioned in Section 2, each thread warp operates on an interval of nonzero elements in the input matrix. The threads fetch the row index, column index, and value of the non-zero input matrix elements. Each thread multiplies the non-zero input matrix element with the corresponding input vector element, and atomically adds the multiplication result to the output vector.
- **Merge Sort** We use *merge sort* from CUDA Thrust library (*Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL)*) as the baseline *non-atomic* implementation. Similarly, this kernel is highly optimized. It iteratively performs local sorting and then merges the partially sorted results at different levels until the entire array is sorted. At every local sort stage, it is typically common that multiple elements have the same value. Sorting these duplicated elements causes increased sorting overhead. Our atomic implementation eliminates redundancy at the end of every local sorting stage so that the following sorting stage sort only unique values from every local sorted group. We use atomic add to count the frequency of each element. Every element is associated with a frequency attribute and it is propagated across all levels of sorting. The frequency information helps restore the array to the original length after the multi-level sorting of all locally unique elements is completed.
- **Page View Count** Page View Count is a *map-reduce* application used to track the number of unique visitors for a given web page. It is from the GPU *map-reduce* benchmark suite Mars [11]. The authors implemented Page View Count by using two invocations of Map Reduce. The first invocation eliminates duplicate page views by mapping each entry to a unique value, globally sorting these values, and then eliminating adjacent duplicates. The second iteration counts the number of remaining unique views for each web page. Our naive atomic implementation extends this benchmark by using atomic reduction between the map and reduce phases of the first iteration of Page View Count in order to eliminate all duplicate entries within each block prior to global sorting. Our block-level atomic redundancy elimination invokes relatively very little overhead and vastly improves the performance of global sorting which is by far the most constricting bottleneck of Mars as the authors themselves mentioned in [11].
- **Summation** The parallel summation kernel is taken from the CUDA computing SDK 5.0 (the reduction kernel), which is carefully optimized with respect to different factors such as shared memory bank conflicts, loop unrolling, etc. The original

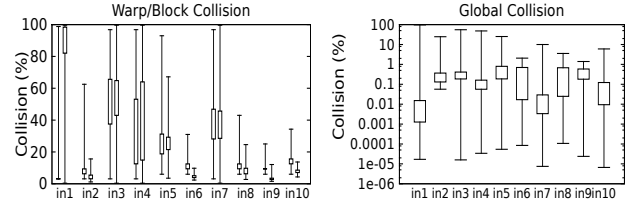


Figure 6. Histogram input statistics

Method	Minimum	Maximum	Average	% Off Opt.
Block P.	0.91	14.74	3.90	9.41%
Warp P.	0.58	6.98	2.03	12.80%
No P.	0.99	6.87	2.30	0.04%

Table 1. Histogramming kernel speedup with regression tree model (kepler)

Method	Minimum	Maximum	Average	% Off Opt.
Block P.	0.68	9.51	3.08	7.82%
Warp P.	0.65	7.61	2.39	8.21%
No P.	0.99	15.40	4.28	0.10%

Table 2. Histogramming kernel speedup with regression tree model (fermi)

version computes a local sum at the block-level at each iteration. It then saves the partial sum for every block in the output array buffer, which becomes the input array for the next iteration of local sum computation. Instead of writing to the block-private data objects, we allow threads to atomically add results to a compacted output array. We vary the level of atomic collision by changing the size of the compacted output array. The program run-time is taken as the total time of all the kernel invocations that are necessary to perform complete summation of a large array.

For the benchmarks that have inherent clustered-collision pattern such as sparse matrix vector multiplication, we use atomicSR within the thread warp scope and atomicSS at the thread warp level. For the benchmarks that have inherent non-clustered-collision pattern such as image histogramming, we use the atomicVR and atomicSS versions at the thread warp level. In our experiments, we found that atomicSS in non-clustered-collision benchmarks do not help as much as it does in clustered-collision benchmarks. Therefore, we do not present the results of atomicSS for non-clustered-collision cases.

We first present the detailed analysis results of two benchmarks. One benchmark has non-clustered-collision pattern. It is the image histogramming kernel. The other benchmark has clustered-collision. It is the sparse matrix vector multiplication kernel. We show the collision statistics for ten representative inputs of every benchmark. And we discuss performance results of various atomic collision reduction techniques.

In the image histogramming kernel, atomic operations are utilized in both the block-private, warp-private, and no-private implementations. We tested the atomic performance on 37 different images with varying degrees of atomic collision levels. We choose to present the results for 10 representative input images. Fig. 6 shows the warp, block, and global collision statistics. In the *Warp/Block Collision* graph, we use box plot to represent the distribution of the maximal collision factors in thread warps/blocks. The bottom of the box correspond to the first quartile and the top of the box

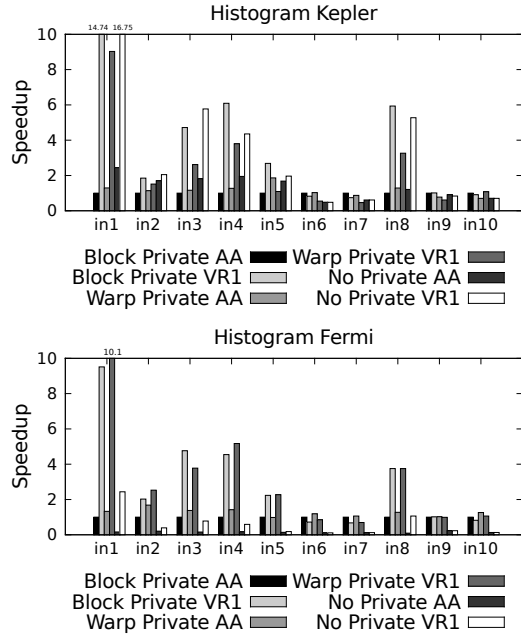


Figure 7. Histogram speedup

correspond to the third quartile. The top bar corresponds to the 100 percentile (the largest max collision factor) and the bottom bar corresponds to the 0 percentile (the smallest max collision factor). The left box plot in every group corresponds to the warp and the right box plot corresponds to the block. The *global collision* graph shows the box plot for the access frequency of unique memory addresses. As can be seen from Fig. 6, the collision statistics vary heavily from image to image. Tables 1 and 2 show the minimum, maximum, and average speedup of the predicted implementation with our statistical learning model (the non-atomic implementation is the baseline). The columns of “% Off Opt.” shows on average how much slower the predicted implementation is compared to the optimal implementation. As both tables show, the average slowdown is extremely small. Even when the predicted code version is suboptimal, in most cases, the performance difference between these two is negligible. Figure 7 shows the speedup of all implementations with respect to the block-private naive atomics implementation. We did not use the original non-atomics as baseline because it is typically much slower even than the naive atomics version [13]. We observe significant speedup in cases with heavy collision and non-trivial speedup even in cases with relatively light collision. Note that on Kepler, the naive no-private version that does not use shared memory is sometimes faster than the block-private and warp-private versions that use shared memory. It is because the Kepler hardware atomic updates are automatically cached and the Kepler’s L2 cache hit bandwidth is much larger. However, after applying our atomic collision reduction techniques, the atomicVR version still performs better in block-private and warp-private versions. The implication is that atomic collision reduction is necessary even when the intrinsic hardware atomic operations are faster. Note that, the number of concurrent threads that are interleaved to execute may also have an impact on the atomics performance. The higher the concurrency level, the more atomics collision overhead can be hidden. The image *in8* has relatively high intra-warp collision level as shown in Fig. 6, however, the collision reduction did not help much on either Fermi or Kepler architecture. That is because image *in8* is the largest and requires a larger number of threads to be launched,

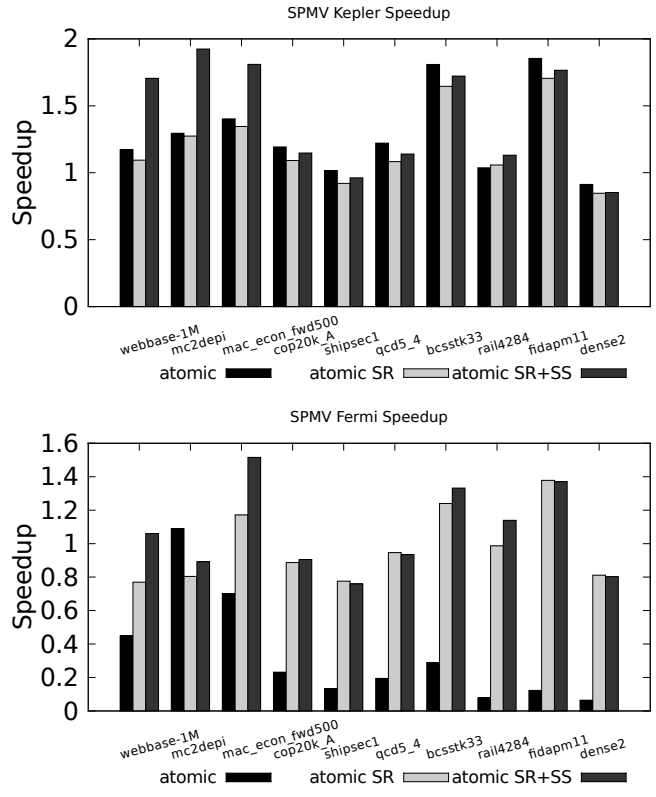


Figure 8. SPMV speedup

Arch.	Minimum	Maximum	Mean	% Off Opt.
Kepler	0.99	1.35	2.46	4.68%
Fermi	0.70	1.04	1.51	8.55%

Table 3. SPMV method prediction speedup and optimality

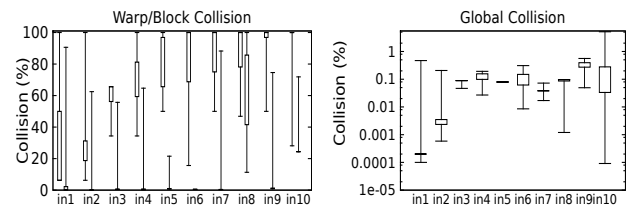


Figure 9. SPMV input statistics

which enables maximal concurrency level. Therefore, the difference between naive atomics and reduced-collision atomics is not that large.

For the sparse matrix vector multiplication kernel, we ran the benchmark on 18 different matrices with different sparsities, sizes, and degrees of atomic collision levels. The result of the benchmarks for 10 representative matrices is shown in Figure 8. The speedups in this chart are relative to the non-atomics kernel. We also present the atomic collision level statistics for each matrix in Figure 9 as a box plot. Similarly, the chart on the left displays the collision levels for thread warps and blocks. The left box plot in every group corresponds to thread warp, and the right box plot corresponds thread block. As usual, this plot shows the min, first

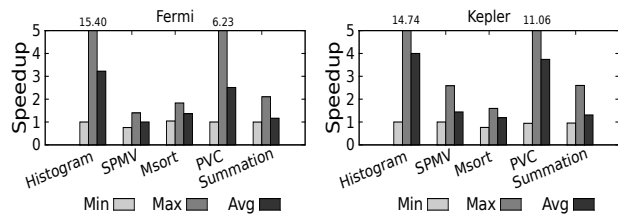


Figure 10. Benchmark speedup summary

quartile, third quartile, and max collision percentage over the set of collisions per warp and block. The right chart shows a box plot for the global collisions level. Overall, we observe that the Kepler architecture has improved atomic addition performance compared to Fermi, which is expected due to an improved L2 cache bandwidth [3]. We used leave-one-out cross validation for statistical learning model. Table 3 show statistics on the average slowdown of the kernel chosen by the decision tree with respect to the fastest implementation. For Fermi architecture, we observe that kernels which use our collision reduction techniques generally offer a significant improvement in performance over the naive atomics kernel. Despite the overall slow performance of naive atomic operations on Fermi, the optimized kernels can even outperform the non-atomic version for some matrices. For Kepler architecture, we observe that the reduced-collision atomic implementations either improved significantly or decreased slightly compared to the naive atomic implementation.

Finally, we present the performance summary for all benchmarks. In Fig 10 we present the minimum, maximum and average speedup across different inputs for each benchmark on both the Kepler and Fermi architectures. The speedup values are normalized with respect to the original code version. In most cases, the original code version is the non-atomic implementation except in the case of image histogramming, where the original code version is the block-private naive atomics version (histogramming already has fast naive atomics implementation published [13] [6]). Non-trivial average speedup is achieved in all benchmarks on both architectures. For most benchmarks, minimal speedup is around 1 (rare case has minimal speedup of 0.65) and about 1.5x-15x maximal speedup is achieved across all benchmarks. For some benchmarks, particularly image histogramming and PVC, we observe a very significant disparity between minimal and maximal speedup. The effectiveness of our atomic collision reduction techniques is obviously heavily dependent on the nature of the program input. Differing inputs may generate heavily contrasting levels of atomic collision. For the sake of fair comparison, our testing input sets have been chosen to represent a broad range of atomic collision from heavy collision to little to no collision on the extreme ends. Our data shows that while presented with a workload generating heavy atomic collision, our techniques improve potential performance tenfold or more (compared to the naive atomic case), and while presented with a workload generating light atomic collision, our techniques induce little to no slowdown (and potentially speedup over the non-atomics version) in most cases. In combination with our regression tree model, our collision reduction techniques have the potential to provide not only vastly simplified programmability but also incredible performance improvement with minimal performance degradation.

5. Related Work

One of the few relevant studies is the hardware extension for efficient atomic vector support [12], where the authors study atomics for SIMD processors in Chip Multi-processors (CMP). Another relevant hardware work is by Gottlieb and others [7], where the fetch-

and-add operation is implemented by an Omega-network for NYU ultracomputer such that memory latency for updates to the same address is logarithmic with respect to the number of cores. There are also limited relevant software studies that systematically explore the usage of atomics for *reduction type* parallelism abundant applications. Most existing software work focuses on application-specific atomic usage, including GPU-MCML[4] – a highly optimized Monte Carlo (MC) code package for simulating light transport, GPU histogramming [16] [15] [13] and GPU graph-cut [18]. Some software work studies atomic collision for a specific memory level, for instance, Gomez-Luna and others [6] optimize atomics operations for scratch-pad memory. To the best of our knowledge, this work is the first one that systematically studies the impact of extensive atomics usage, and explores a variety of atomics collision reduction techniques. The *atomic collision to scatter* approach is relevant to the job swapping idea used in control divergence and memory irregularity removal for GPU programs [19]. The local reduction in *atomicSR* algorithm is similar to the global reduction [9] approach, which does not need to detect boundaries and sizes of local reduction groups. The GPU voting used in *atomicVR* is studied extensively in [17], the techniques in which can help us speedup *atomicVR* even more.

6. Conclusion

In this paper, we proposed to use atomic operations extensively for computation rather than communication on many-core GPUs. We systematically studied the influence of atomic collision on GPU programs and investigated various solutions on the elimination of atomic collisions.

Acknowledgement

We thank Jos E. Moreira for his comments on the draft of this paper. We owe a debt to the anonymous reviewers for their invaluable comments. This material is based upon the work supported by Rutgers University Research Council Grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

- [1] “Cuda occupancy calculator.” NVIDIA. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls
- [2] “Matrix market.” [Online]. Available: <http://math.nist.gov/MatrixMarket/>
- [3] “Whitepaper - nvidia’s next generation cuda compute architecture: Kepler gk110.” [Online]. Available: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>
- [4] E. Alerstam, W. C. Y. Lo, T. D. Han, J. Rose, S. Andersson-Engels, and L. Lilje, “Next-generation acceleration and code optimization for light transport in turbid media using GPUs,” *Biomedical Optics Express*, vol. 1, no. 2, pp. 658–675, 2010.
- [5] N. Bell and M. Garland, “Cusp: Generic parallel algorithms for sparse matrix and graph computations,” 2012, version 0.3.0. [Online]. Available: <http://cusp-library.googlecode.com>
- [6] J. Gomez-Luna, J. M. Gonzalez-Linares, J. I. B. Benitez, and N. G. Mata, “Performance modeling of atomic additions on gpu scratchpad memory,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2273–2282, 2013.
- [7] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, “The nyu ultracomputer—designing a mimd, shared-memory parallel machine (extended abstract),” in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ser. ISCA ’82. Los Alamitos, CA, USA: IEEE

- Computer Society Press, 1982, pp. 27–42. [Online]. Available: <http://dl.acm.org/citation.cfm?id=800048.801711>
- [8] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, “Maximizing multiprocessor performance with the suif compiler,” *Computer*, vol. 29, no. 12, pp. 84–89, Dec. 1996.
- [9] M. Harris, “Optimizing parallel reduction in cuda,” 2007, <http://developer.download.nvidia.com/compute/cuda/1.1/Website/projects/reduction/doc/reduction.pdf>.
- [10] T. Hastie, R. Tibshirani, and J. Friedman, “The elements of statistical learning.” Springer, 2001.
- [11] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, “Mars: a mapreduce framework on graphics processors,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT ’08. New York, NY, USA: ACM, 2008, pp. 260–269.
- [12] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen, “Atomic vector operations on chip multiprocessors,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 441–452.
- [13] C. Nugteren, G.-J. van den Braak, H. Corporaal, and B. Mesman, “High performance predictable histogramming on gpus: exploring and evaluating algorithm trade-offs,” in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 1:1–1:8.
- [14] NVIDIA, “Gpu computing sdk.” NVIDIA. [Online]. Available: <https://developer.nvidia.com/gpu-computing-sdk>
- [15] V. Podlozhnyuk, “Histogram calculation in cuda,” in *Technical Report*. NVIDIA, 2007.
- [16] R. Shams and R. A. Kennedy, “Efficient histogram algorithms for NVIDIA CUDA compatible devices,” in *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, Gold Coast, Australia, Dec. 2007, pp. 418–422.
- [17] G.-J. Van Den Braak, C. Nugteren, B. Mesman, and H. Corporaal, “Gpu-vote: A framework for accelerating voting algorithms on gpu,” in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 945–956. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_92
- [18] V. Vineet and P. Narayanan, “Cuda cuts: Fast graph cuts on the gpu,” in *Proceedings of CVPR workshop on Visual Computer Visions on the GPUs*, 2008.
- [19] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, “On-the-fly elimination of dynamic irregularities for gpu computing,” in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS ’11. New York, NY, USA: ACM, 2011, pp. 369–380.