Using Massive Atomics Operations for Massively Parallel GPU Applications: Inevitable or Indispensable?

Abstract

The parallelization process for a sequential applications involves handling of concurrent shared memory object updates. One important type of parallelism is exploited when the order of the memory updates to the same location does not change the output of the program. This type of parallelism is reduction type parallelism. It typically exists in many important applications such as *data mining*, *numerical analysis* and *scientific simulation*. The implementation of these applications for multi-core architectures is typically accomplished by using thread(s) private data objects to hold partial results and applying a sequential final stage to aggregate the partial results. Porting this type of applications to massively parallel GPU processors faces new challenges. One major challenge is work partitioning, the target of which is minimal communication between individual threads that run in parallel and less work in the sequential reduction stage to aggregate all partial results. However, when the number of threads explodes to thousands of or millions of, the workload partitioning becomes much more complicated than that of less than ten threads. This may ultimately lead to load unbalancing or extra control code for handling boundary cases in irregular applications. Extra control code, may on one hand lead to increased coding complexity, and on the other hand, runtime thread divergence and thus serious performance degradation.

In this paper, we propose a novel approach to handle concurrent shared memory objects on GPUs. This approach not only yields good performance, but also good programmability. This approach uses atomic operations extensively. Atomic operations for shared memory updates are known to be expensive and may cause serialization for parallel threads. However, we discovered that, with appropriate atomic collision elimination techniques, we can achieve similar performance or even better performance than the traditional non-atomics involved implementation. We implemented these techniques as a library of functions with simple interface. The programmers can call these procedures to perform shared memory object updates without worrying about the order of these operations or workload balancing, while achieving significant performance gains brought by the massive parallelism in GPUs.

1. Introduction

Parallel applications need to handle shared memory object updates efficiently and correctly. In an important type of parallel applications, the operations to the shared memory objects are commutative and thus the memory update order does not matter [4]. Examples include *data mining*, *machine learning*, *numerical methods* and most of the applications that fit into the *map-reduce* model. Parallelism in this applications is exploited in the way that every thread uses thread-private data structure to hold the partial computation results so that a large task can be split into small subtasks to run in parallel. A final stage aggregates the partial results from individual threads and performs the shared memory object(s) update correspondingly.

In the multi-core implementation of these applications, the workload is typically partitioned in a way that the updates to the same shared memory object are spread to as few threads as possible such to minimize inter-thread communication and thus reduced work in the final aggregation stage. In certain cases, communication between individual thread is completely avoided. For instance, in a sparse matrix vector multiplication kernel, the dot product of a row vector in the matrix and the input vector results in a single entry in the output vector, which can be completely assigned to one thread. This approach of minimizing inter-thread communication, can be implemented with ease for multi-core architectures, but not necessarily for many-core architectures. There are two major issues. The first issue is load balancing. It is challenging to maintain balanced workload and minimal thread communication at the same time if the number of threads explodes to thousands of or millions of or even more. For instance, in the sparse matrix vector multiplication kernel, the number of non-zero entries varies from row to row in the matrix and every row maps to one reduced entry in the output vector. To ensure minimal inter-thread communication, we let one thread handle one or multiple rows. It is easy to balance the workload if we have a few threads. However, if we have many fine-grained threads, one thread may be assigned to one or very few rows and it's difficult to ensure load balancing since the number of entries per-row may not be balanced. The second issue is work partitioning. It's challenging to assign workload to every co-running thread while ensuring minimal inter-thread communication when the number of threads is large. For instance, in the data clustering kernel, the centroid of every cluster depends on the data points in the cluster, and thus the workload partition approach that ensures minimal inter-thread communication is to assign a cluster to every thread. However, the number of clusters may be small as opposed to the total number of data points and input dependent. Similarly, if we have two or four or eight threads, we can pack one or more clusters to every thread. If we have millions of threads, we may run out of the number of clusters to assign to every thread.

Due to the aforementioned reasons, when an application is ported from the sequential or multi-core implementation to the many-core GPU implementation, the programmers create more thread-private data objects to minimize thread communication and

ensure balanced fine-grained workload among different threads. This results in large memory space to store intermediate results. Another consequence is the extra control code added to handle the boundary case when the commutative operations to the same shared memory object is spreads across different threads. The control code may cause serious performance degradation due to runtime thread divergence in GPUs [15] since GPU threads run in lock-step groups. The code becomes more complex and any further optimization a challenging task.

When the extra required space to hold thread-private data objects becomes prohibitive and/or the code to handle boundary cases in applications with unstructured parallelism becomes over complicated, an inevitable approach is the use of native hardware atomic instructions. A hardware atomic instruction ensures the atomicity of the read-modify-write process. For instance, an atomic add instruction loads a variable from the memory, add a number to it, and write it back to the memory without being interrupted in the middle. Native atomic operations help manage the commutative memory updates automatically without programming efforts. The example applications of using atomic operation include the histogramming kernel [8], bio-simulation kernel [2] and graph-cut kernel [14].

Can we use atomic operations generically in reduction-parallelism abundant applications, even when we have enough memory space to store extra thread private objects or the code to handle boundary cases is feasible? Atomic operation can be very expensive compared its non-atomic counter part. It can potentially serialize parallel threads when different threads try to update the same memory location at the same time, known as atomic collisions. Atomic operations have been avoided as much as possible in traditional multicore parallel applications. Their uses have been mainly restricted to synchronization primitives such as locks, mutex and barrier. In this paper, we performed a systematic study on the general use of atomic operations for non synchronization type tasks. In contrast to the traditional perception that atomic operations should be avoided in computation as much as possible, our study shows that these operations can actually be profitable instead of harmful if used appropriately. Not to mention that it can help alleviate the burden on programmers. This is based on the two major observations: (1) if atomic updates to the same location are scattered in different time intervals, then the parallel threads will not be serialized; (2) atomic updates to the same memory location, can be converted to parallel reduction and one atomic update. The key challenge is then how can we detect atomic conflicts efficiently, scatter them and/or convert them into parallel computation in dynamic and irregular applications.

The previous studies on atomic operations are mainly on enhancing the performance from the hardware perspective [7]. A few software studies have explored using atomics for computation in specific applications such as bio-simulation [2], graphics manipulation [14], and image histogramming [8]. However, they use the native atomic operations for a certain type of workload without further exploring into the systematic approaches to eliminate atomic conflicts. Other related work include the study on enhancing applications with a lot of commutative operation to shared memory objects – mainly in the category of *map-reduce* applications [5]. Most of them apply to distributed systems instead of single computing node.

In this paper, we study the impact of using atomic operations in computation extensively for the first time. We discovered the counter-intuitive fact that atomic operations can not only increase the programmability of massively parallel GPUs but also help enhance the performance if used appropriately. We explored the different approaches of eliminating atomic conflicts at various granularities. We performed complexity and bound analysis for different algorithms. We also analyzed the trade-off between performance, programmability and space-efficiency when combining atomics with the traditional atomics free implementation. Our contributions can be summarized as follows:

- For the first time, we studied the use of atomic operation for shared memory tasks using commutative operations. We explored the impact of inter-warp collision and intra-warp atomic collision, and their implications on using atomics smartly in general purpose computation.
- We proposed efficient approaches to reduce atomic collisions at different granularities. We studied the complexity of different algorithms, analyzed the performance bounds, and provided guidance for choosing algorithm in different cases.
- We built a library that implemented the atomic collision elimination techniques with simple interface. The programmers can call call these libraries like calling regular native atomic functions, while achieving satisfactory performance at the same time.
- We applied atomic operations generally in several important kernels that traditionally non-atomic operations and achieved good speedup. These kernels include sorting, histogramming, sparse matrix operations and etc. We identified the potential of using atomic operations and exemplified them in these real important GPU kernels.

The rest of the paper is organized as follows. In Section 2 we describe the background of GPU execution model, and provide a motivation example. In Section 3, we discuss the performance aspect of using massive atomic operations in computation, detailed algorithms for eliminating both intra-warp collision and inter-warp collision, as well as the algorithm complexity and bound analysis. In Section 4, we discuss the programmability and the space efficiency aspect of applying massive atomic operations. In Section 5, we present evaluation results. We describe related work in Section 6 and conclude in Section 7.

2. Motivation

Background A GPU is a highly parallel many-core architecture. It normally consists of hundreds of or thousands of cores on one chip. We use NVIDIA GPU terminology to explain the architecture and programming model. A GPU is made up of multiple Streaming Multiprocessors (SM). A SM is a Single Instruction Multiple Data (SIMD) processor, in which a group of threads execute the same instruction on multiple data elements. A general purpose GPU program includes both CPU code and GPU code. The function that runs on GPU is called a kernel function. A kernel function may invoke millions of or more threads to run concurrently on a GPU. These threads are divided into many small groups to be scheduled on different SMs. Such a small group of threads is called a thread warp. A thread warp is the minimal scheduling and execution unit. The threads in a warp run in lock-step fashion, which means they have to execute the same instruction at the same time. There is also implicit synchronization within a thread warp. When one thread completes one instruction, all other threads in the warp finish as well.

There are two major types of memory on a GPU – on-chip memory and off-chip memory. On-chip memory includes cache and scratch-pad memory. The scratch-pad memory is almost as fast as cache, while it is also software manageable. NVIDIA terminology refers to scratch-pad memory as *shared memory*. As *shared memory* is a overloaded term, we still use the notion of scratch-pad memory in this paper. We denote the data objects that can be accessed by different threads as shared memory objects, whether they are in on-chip memory or off-chip memory. Off-chip memory

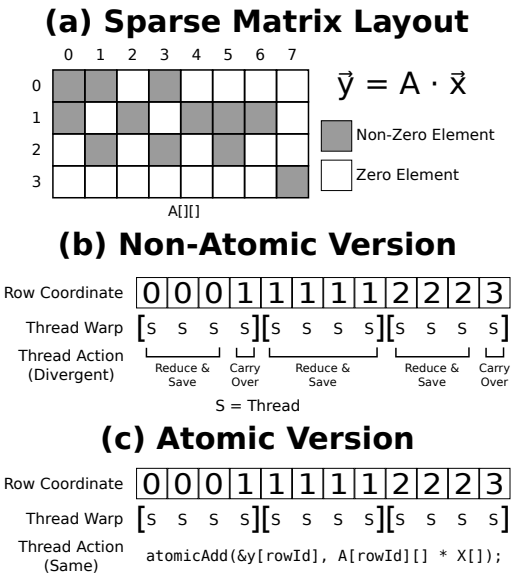

Figure 1. Sparse matrix vector multiplication kernel

is of a much larger size than on-chip memory, but has larger latency as well. The off-chip memory is partitioned into *global memory* for dynamic memory objects, *local memory* for static memory objects, *constant memory* for constance memory objects and *texture memory* for multi-dimensional data locality. We focus on *scratch-pad memory* and *global memory* for the study of atomic operations since they can both be explicitly managed by programmers and have atomic instruction support.

Atomic collisions refer to the scenarios when more than one concurrent thread try to read-modify-write the same data object at the same time. According to the atomicity property, these memory updates from different threads have to happen serially. We classify atomic collision into two categories - intra-warp collision and inter-warp collision. The first type of write collision exists only within warps, which means threads within the same warp atomically read-modify-write to the same memory location. The second type of write collision only exists across warps, which means different warps may have conflicting atomic memory destinations. The two types of write collisions can exist in both scratch-pad memory and global memory. The scratch-pad memory atomic instruction is implemented as a loop that atomically updates unique and unlocked memory locations at each iteration until the warp's requested atomic memory updates are completed. The number of iterations for a warp is then the maximal number of times a conflicting memory address exist in the whole thread warp's atomic memory request. For global memory atomics, the architecture support has been improving from the first generation of GPGPUs. In the latest Kepler GPU from NVIDIA, the performance of atomic transaction is similar to that of the non-atomic memory transaction when there is no conflicting read-modify-write operation [9]. Common atomic operations include atomicAdd, atomicOr, atomicAnd, atomicMin and etc. We focus on the atomic operations that have the commutative property.

A Motivation Example: Sparse Matrix Vector Multiplication In this section, we show the influence of atomic collisions on a real application. We compare the performance of a traditional non-atomics kernel and atomics implementation kernel. We use the sparse matrix vector multiplication kernel – a highly optimized version from CUSP [3], which is an open source C++ library of generic

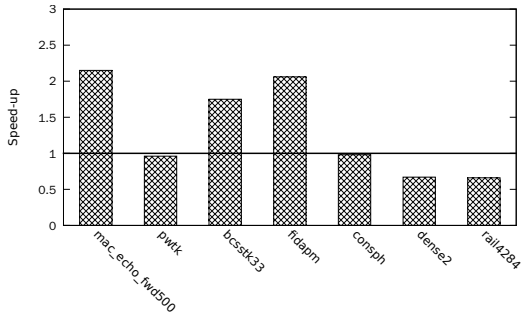

Figure 2. Using native atomics for sparse matrix vector multiplication

parallel algorithms for sparse linear algebra and graph computations on GPUs. In this implementation, a thread warp iterates over one or more consecutive rows in the sparse matrix, and update corresponding output vector entry. Depending on whether a thread is at the boundary of entries from different rows, different actions are taken and thread private objects are used to carry partially computed results. We explain this with an example in Fig. 1. In this example, we assume the format for the sparse matrix is coordinate format (COO). WIn Fig. 1 (a), we show a sparse matrix A with 4 rows, 32 columns and 12 non-zero elements in total. We have a column vector x of size 8. The output vector $y = A \times \vec{x}$. In Fig. 1 (b), we show the workload partition for the non-atomics involved implementation in *CUSP*. The row coordinate value of every non-zero element in the sparse matrix A is listed. We assume every thread warps consists of four threads. A thread warp iterates over different elements in the matrix, with each thread corresponding to a non-zero entry. Parallel reduction is performed for multiplication results on matrix entries on the same row. Every thread takes different actions from other threads in the same warp if it is operating on the data element at the boundary of different rows. This will cause thread divergence [15]. For instance, the third thread that is in charge of the last non-zero element in the first row, have to perform not only parallel reduction but also save the reduced value into the corresponding column vector *y[0]*. The fourth thread in the first warp needs to carry the multiplication result of *A[0][1]*x[1]* into the next iteration of the thread warp, which is different action from other threads too. Similarly, we have to perform *reduce &* save at the boundary between row 1 and row 2 at the fourth thread in the warp at the second iteration. We carry the multiplication value for *A[3][7]*x[7]* to the future iterations too. In Fig. 1 (c), we show the atomics implementation. The action for every thread is exactly the same – perform atomic add for the multiplication result of the corresponding matrix entry and vector entry. There is no thread divergence in this case. Note that the non-atomics version used scratch-pad memory to hold [tocheck] parallel reduction result before a final write back to memory to improve the performance.

In Fig. 2, we show the performance comparison of both traditional non-atomics version and atomics version. The experiment is performed on NVIDIA GTX 680 with latest NVIDIA Kepler technology. We plot the ratio between the time of non-atomic version and that of the atomic version as the y axis. If the ratio is greater than 1, then it means the atomics version is faster, otherwise the non-atomics version is faster. Every bar in this graph represents a matrix. We use the sparse matrix input from matrix market [tocite] – the standard pool of sparse matrix in real world applications which is also typically used as benchmarking for sparse matrix operations. In Fig. 2, we can see that, surprisingly, the use of *native* atomics even improved performance for some sparse matrices. For some dense matrices, the use of native atomic slows down

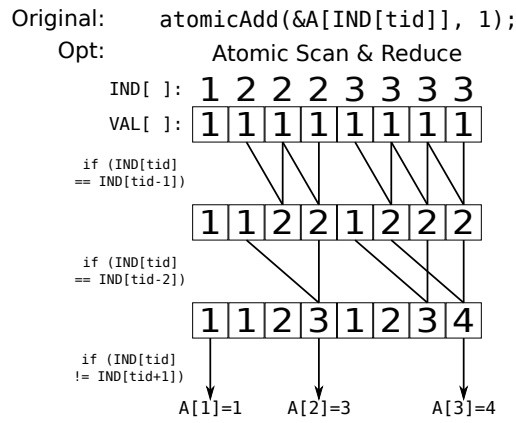

Figure 3. Atomic scan and reduce example

the program due to the increased extent of atomic collision, such as matrices *pwtk, consph, dense2, rail4284*. The reason for the performance improvement is that, there is a lot of thread divergence in the traditional non atomic implementation as mentioned above. When the matrix is very sparse, the thread warp might encounter row boundaries frequently and thus threads divergence in their execution, leading to a lot of time wasted on serialization of branch statements. The atomics implementation have every thread perform the same action and minimize the control divergence. While in dense matrices, the benefits of minimized control divergence start to be offset by the effect of atomic collisions. We start seeing slowdown in the dense matrices. We did not apply any generic atomic collision reduction techniques in this experiment, which means there is further improvement space for using atomics massively in computation. We discuss the techniques to eliminate atomic collision in Section 3.

Finally, it is noteworthy that the length of the kernel function implementation drops from 57 lines in the non-atomic version to 18 lines in the atomics implementation.

3. Performance

In this section, we discuss the performance aspect of using atomic operations extensively for computing purpose. We describe our algorithms for reducing atomic collisions. There are two major types of approaches. One approach is based on converting atomic collision into computation. We can not only decrease the amount of atomic collision, but also add the amount of computation and increase the computation/memory ratio, which helps increase the utilization of GPU cores significantly [6]. The other approach is based on re-scheduling atomic memory updates so that we can scatter potential atomically conflicted accesses over different time intervals. The intuition is that if the atomic updates of the same memory addresses are scheduled at the same time, they are atomic collisions. However, if they are scheduled at different times, they are not. For GPU programs, we typically run a lot more threads (than the number of cores) concurrently for instruction latency hiding purpose. Therefore, there is potentially good opportunity to reschedule threads for minimization of 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 3.1.

3.1 Atomic Collision To Computation

With the *atomic-collision-to-computation* approach, we perform parallel reduction for threads that have the same memory access addresses and perform atomic updates only on aggregated results to minimize the amount of atomic collisions. We present the application of *atomic-collision-to-computation* algorithm in two scenarios: (1). the threads that access the same address are placed next to each other, which happens frequently in kernels with regular and structured parallelism like matrix vector operation in scientific simulation applications [7]; (2) the threads that access the same memory locations are not necessarily placed next to each other, which is frequent in kernels with irregular and unstructured parallelism such as graph traversal and sampling. We refer to the first scenario as the *clustered-collision* case and the second one as the *non clustered-collision* case.

Clustered-collision Case: The difficulty of converting atomic collision to computation is that the number of threads that have one unique memory access may be different from the number of threads that have another unique memory access. The parallel reduction for these two groups of threads have to happen with different number of reduction steps. Further, the boundary for parallel reduction of different threads groups may be irregular and highly dependent on the program input. An example is the *sparse matrix vector multiplication* kernel mentioned in Section 2. It is not easy to write non-atomics code to handle these cases. Further, in the atomics implementation, how to detect the boundary and use only necessary number of reduction steps for every thread group is a challenge in this case. We propose algorithm for the *clustered-collision* case that can perform minimal number of reduction steps for every thread group at the same time and we can detect the boundary efficiently with this algorithm. We describe the algorithm by an example in Fig. 3. In this example, every threads tries to atomically add the value of 1 to a specific memory location in array *A* indicated by the indirection array *IND[ ]*, while IND[tid] points the location in *A* for thread with index *tid*. Every thread is in charge of one atomic add operation. As we can see from Fig. 3, we have 8 threads, and the first thread needs to increment *A[1]* by 1, while the second to the fourth threads need to atomically increment *A[2]* by 1, and the fifth to the eighth threads need to increment *A[3]*. The optimal case is to perform 2 reduction operations for threads *2-4* and threads *5-8*, while no reduction for thread *1*. We achieve the optimal number of computation steps by using a predicate to check if the threads before the current one holds the same memory access address. At *step 1* in Fig. 3, we mark the necessary addition of atomic increment values with arrows and we only perform addition when the thread before the current thread has the same memory access address. After that we get the summation of atomic update values for every thread and its precedent thread if and only if their destination addresses are the same. At *step 2*, we perform similar aggregation of atomic update values conditionally, except that we extend the distance between two threads to be checked and aggregated to 2. After this step, we have the summation of results from 3 threads before the current thread and itself if and only if they hold the same memory access address. For the threads that do not have the same memory access address, we do not include it in the computation. Finally, after step 3, we get the summation results for every thread and its precedent threads that have the same memory access address, and we save them correspondingly to the temporary *VAL[]* array. In the last step, we perform atomic addition with thread 1, thread 4 and thread 8 to *A[1], A[2] and A[3]* correspondingly. We achieved this by a predicate that checks if the thread has the same memory address as the one that follows it.

We describe the detailed algorithm in Alg. 1. In this algorithm, we present the version for atomicAdd – *atomicAddSR*, which can

Algorithm 1 Atomic scan and reduce algorithm

```
1: procedure ATOMICADDSR(A[], idx, val, N)
2:     alloc spmVal[]; ▷ allocated temporary scratch-pad memory
3:     alloc spmIdx[];
4:     spmIdx[tid] = idx;
5:     spmVal[tid] = val;
6:     sync();   ▷ only necessary if it is beyond thread warp level
7:     for i = 1 to log2 N do
8:         if (spmIdx[tid] == spmIdx[tid-2^i]) then
9:             spmVal[tid] += spmIdx[tid-2^i];
10:        end if
11:    end for
12:    if spmIdx[tid] ≠ spmIdx[tid+1] then
13:        atomicAdd(&A[idx], spmVal[tid]);
14:    end if
15:    sync();          ▷ only necessary if it is beyond warp level
16: end procedure
```

Algorithm 2 Atomic vote and reduce algorithm

```
1: procedure ATOMICADDVR1(A[], idx, val)
2:     spmIdx[tid] = idx;
3:     spmVal[tid] = val;
4:     sync();                          ▷ sync if beyond warp scope
5:     randTid = rng();                 ▷ randomly generate tid
6:     sampleIdx = spmIdx[randTid];
7:     voteMask = __ballot(idx == sampleIdx);
8:     if (freq(voteMask) ≥ threshold) then
9:         pSum = parallelReduce();     ▷ reduce sampled threads
10:        atomicAdd(&A[sampleIdx], pSum);
11:        if (idx ≠ sampleIdx ) then
12:            atomicAdd(&A[idx], val);
13:        end if
14:    else
15:        atomicAdd(&A[idx], val);
16:    end if
17: end procedure
```

be applied to other atomic functions that have the commutative property. For users who want to convert atomic collision to computation, they can simply replace their original atomic functions with this one and achieve better performance. The parameters for *atomicAddSR* include the array *A*, which is the array to be atomically updated, the index *idx*, which is the index of element in array *A*, the value *value*, the value to be atomically added to *A[idx]*, and the number of threads *N*, which threads we want to perform reduction before atomic updates. In this implementation, we have to sync after certain steps to make sure the memory updates are visible to every thread after every step of reduction (at line 11) or initialization (at line 6). However the synchronization is only necessary when we use more than one thread warp. The synchronization between threads within the same warp is implicit and very fast. Overall, the computation complexity of our algorithm is logarithmic with respect to the maximal number of threads that have the same memory access.

Non Clustered-collision Case: In this case, not only the amount of atomic collision is irregular, but also how they are distributed in different threads is irregular – they are not necessarily located next to each, which makes the detection of boundary more challenging. In certain cases, we may have no atomic collision at all and thus no reduction step is needed. Or we may have one memory address that is accessed by many threads and reduction is necessary. We propose an algorithm that identifies the most frequently accessed memory location with the maximum likelihood. Based on the frequency of this memory address, we know how much collision and we can decide how much reduction we will perform. We name this approach as *atomic vote and reduce* algorithm.

We describe the *atomic vote and reduce* algorithm in Alg. 2. We first randomly choose a thread index and the corresponding thread's memory access location (lines 5-6). Then we perform a ballot on all threads to find out which threads access the same memory address at line 7. Therefore we get the frequency of accessed to this address. If it is above a threshold, we perform partial reduction on the sampled threads and write the aggregated result back atomically (lines 8-14). Otherwise, we let the native atomic operations take care of the computation and memory updates (lines 15-16). Similarly, this approach not only applies to atomic addition but also any other atomic operation that has the commutative property. The synchronization instruction at line 4 is not necessary if we perform atomic vote and reduce at thread warp level.

We use the random sampling approach based on the following observation: the more a memory address appears in this thread group, the more likely it will be chosen. As the number of thread group increases, the most frequent memory address can be obtained with maximal probability according to the *large number theorem*. The number of threads invoked in a GPU kernel is typically large and thus our detection strategy is quite efficient in most cases confirmed by experiment results in Section 5.

We choose the *threshold* in a way that guarantees no performance degradation if compared to the naive atomic implementation case. Assume the total number of threads in this thread group is N, the maximal number of threads that access one memory location is x, and the extra setup overhead for the partial parallel reduction is s. Assume the threshold is *thresh*, we determine the value of *thresh* solving the following inequality:

$$(N - x) + log_2 x + s \geq N$$

The above formula considers the most conservative case. The left-hand side estimates the total number of computation instructions needed after we perform the parallel reduction, and the right-hand side represents total number of computation instructions if we use the naive atomic implementation. In the left-hand side, the $N - x$ component represents after partial parallel reduction how many extra computation iterations are needed, the $log_2 x$ component represents the computation steps needed for parallel reduction, and the s component represents other overhead such as the polling instruction before the partial reduction. If we follow this formula, we guarantee in the worst case, our transformed version at least does not run any slower than the original naive atomics version. It is because the component $N - x$ represents the case when all the other threads (the ones do not access the most frequent memory address) have conflicting memory access. However, this is the worst case scenario and it is rare in a lot of cases.

3.2 Atomic Collision to Scatter

In the *atomic-collision-to-computation* approach, we have to perform multiple computation iterations before we perform any atomic update to a memory location. Extra kernel is instrumented to help perform the parallel reduction and shared memory space is needed. Furthermore, during the process of parallel reduction or scan, there are always threads that stay idle and certain processor cores are under utilized. In this section, we present an approach that does not need to utilize extra computation steps or shared memory. This approach is based on the intuition that if two different threads that conflict at the same memory addresses are scheduled to run at two different time instants, then no computation needs to be serialized due to atomic collision.

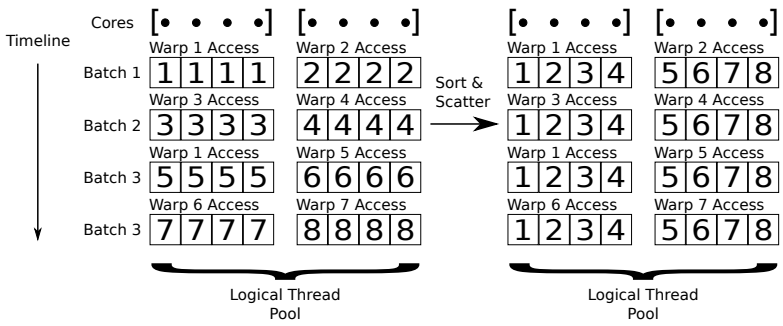

Figure 4. An example of the atomic-collision-to-scatter approach

We develop an algorithm that performs the scattering of different thread or thread warps to minimize atomic collision as much as possible. We first use an example in Fig. 4. In this example, we assume there are two sets of physical cores with four each, represented by the dots. We have a pool of 16 logical threads to run on these two set of cores. This is typically the case in GPUs, when we have a lot more threads than the total number of physical cores. These threads are divided to run in batches, each batch of threads run at all the cores simultaneously. Every batch has two thread warps, assuming that every thread warp is of size 4. Another batch of threads will be placed in these cores once the last batch of threads have finished the execution. We set a batch to be 8 threads here [1]. Fig. 4 (a) shows the original case, in which we have atomic collision within every thread warp. In the naive atomic implementation case, we might have to serialize every thread. Fig. 4 (b) we show the thread-core mapping after we perform a scattering of atomic memory threads by rescheduling the logical threads. For the *warp 1* in Fig. 4 (b), the threads are originally threads from *warp 1*, *warp 2*, *warp 3* and *warp 4*. After the re-scattering, we can see that within every thread batch, there is no atomic collision. If we count the total number of *addition* steps, we have one addition step for Fig. 4 (b) at every batch, while we have three *addition* steps using the atomic-collision-to-computation approach (2 for reduction + 1 for atomicAdd) and four *addition* steps using the naive atomic approach. Overall, we can have three to four times speedup with the *scatter* approach while avoiding extra code instrumentation and shared memory usage.

The essential idea of our scatter algorithm is to redistribute conflicted memory accesses as far as possible in terms of the distance between their host threads' logical indices. Logical threads that are placed together tend to run together, for instance, every thread in a thread warp group or a thread block group run at the same time one SM. Threads that are from different blocks may be scheduled to run at different batches and have no conflicting memory accesses. To enable this, we first group the threads that have the same memory access and construct many sets. We have the same number of sets as the number of unique memory addresses. We order these sets from low addresses to high addresses. Starting from the first set, we pick one thread and place it into the newly created empty group, representing the new thread layout. Then we pick one thread from the second set, and place it after the last thread in this new thread layout group. We keep doing this until we finish the last unique set of threads. We start from the first set again and keep appending threads to the group until we remove all threads from the sets. Finally, we obtain the new order of the logical threads for scattering purpose. We perform thread reordering with the new logical thread order with program transformation approaches as described in [15].

[1] In GPUs, there are typically more threads than the number of cores in every batch. We set it to be the same as the number of cores in this example to explain the idea of atomic-collisions-to-scatter

Algorithm 3 Atomic scatter algorithm

```
1: procedure ATOMIC2SCATTER(warpAddr[], newWarpLayout[], warpNum)
2:     swAddr = sort(warpAddr[]); ▷ may or may not be needed [2]
3:     Sets = unique(swAddr[]);          ▷ sets of unique addresses
4:     i = 0;
5:     while i < warpNum do
6:         for every set ∈ Sets do
7:             if |set| ≠ 0 then
8:                 wId = set.pop();
9:                 newWarpLayout[i] = wId;
10:                i++;
11:            end if
12:        end for
13:    end while
14:    return newWarpLayout[];
15: end procedure
```

We describe the whole process in Alg. 3. We can perform re-scattering at different granularity – individual threads, warps or blocks. Alg. 3 is described at the warp granularity. We choose to implement and perform re-scattering at warp granularity. The reason we choose warp level re-scattering is that (1) the original memory coalescing property is preserved, which refers to the memory access pattern for threads in the same warp – if all data objects are located near each other, then we can perform minimal number of memory transactions; (2) the conflicts within a warp can be easily resolved with intra-warp atomic-collision-to-computation approach describe in Section 3.1 since synchronization between threads is implicit and fast.

We present the results of the motivation example we used in Section 2 for sparse matrix multiplication after we further apply the optimization techniques in this section. In Fig. 5, we compare the results of using naive atomics and using atomic collision reduction techniques. The bar on the left in every group represents the performance of using the native atomics and the bar on the left represents the case after the optimizations are applied. Now we can see that for the matrices that are too dense to benefit from the benefit by atomics code version, we can improve their performance and make it almost the same as the non-atomics version. These matrices are respectively *consph*, *dense2* and *rail4284*.

We would like to point out that our re-scattering algorithm can potentially improve the cache performance for atomic update addresses. Since we we order these unique sets of threads based their memory access addresses and we pick threads from every

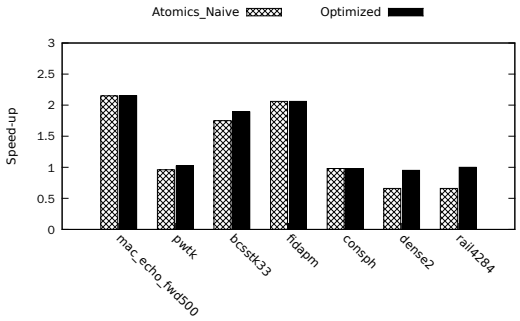

Figure 5. The results of sparse matrix vector multiplication after we reduce collision

Original:

```
void kernel() {
  int tid = TID;
  ...
  atomicAdd(&A[tid], val);
  ...
}
```

Intrawarp:

```
void kernel() {
  int tid = TID;
  ...
  atomicAddSR(&A, tid, val);
  ...
}
```

Interwarp:

```
void kernel() {
  int tid = TID;
  tid = remap[tid];
  ...
  atomicAdd(&A[tid], val);
  ...
}
```

Figure 6. Atomic collision removal function usage

set in such an order. One thread warp can potentially fetch data from memory that can be reused by another co-running thread warp already, since the data they access is near each other. This effect is confirmed by our experiment results in Section 5.

4. Programmability and Space Efficiency

In the last section, we studied the performance impact of atomic collision and proposed efficient atomic collision reduction techniques. We have demonstrated that for irregular sparse matrix vector multiplication kernel, compared to the non-atomics code that involves complicated implementation, we can achieve better performance for all the sparse matrix scenarios or at least the same performance for very dense matrix scenarios. In this section, we discuss the programmability and the space efficiency of using atomic operations.

programmability We have implemented the atomic-collision-to-computation and atomic-collision-to-scatter techniques as a library called *G-Atom*. We have implemented the functions that can be invoked within a kernel function or outside a kernel function. Our implementation for the atomic-collision-to-computation functions are mainly for thread-warp scopes. The programmers can directly call these functions as if they are calling the intrinsic atomic functions. For the atomic-collision-to-scatter case, the programmers can call the function with the interface like *atomic2scatter(...)* described in Section 3.2 before a kernel is invoked, and obtain a mapping array *remap* for the new layout of the logical threads. In the kernel function, we then replace the *thread index* with the one indicated in the layout array *remap*, in which *remap[0]* represents which old logical thread should be placed as the first logical thread in the current rescattere thread layout. The new thread index is used to derive the dataset and tasks (control flow) every thread is working on. We show the usage of *atomic-collision-to-computation* and *atomic-collision-to-scatter* with the following example in Fig. 6.

space efficiency As mentioned in the Section 1, the use of atomic operations can help reduce the total number thread private objects. We list a few cases in which the memory space can greatly reduced. For instance, in the parallel summation kernel [cuda sdk], we usually need an input and output buffer for partially reduced results at every level. With atomic implementation, we can reduce the output buffer size greatly. For the very important *sort* kernel, if we have same entries after every level of local sort (like every level of *merg-sort*, we can compute their frequency independently with atomic operations, and therefore reduce the number of elements to be sorted in the next round while only need to carry their frequency. The space efficiency can also be improved at different granularities, for instance, if we switch from using thread-private object to block-private object with atomics, we can also save the memory space used and perhaps utilize fast scratch-pad memory to store the block private objects.

Finally, the atomic implementation does not conflict with the non-atomics implementation. It can be combined with the non-atomic implement when the non-atomic implement is straightforward, and achieving best performance benefits, programmability and space efficiency with the appropriate level of mixing. We leave this as our future work.

5. Evaluation

We evaluate the performance of atomics implementation of various important kernels in this section. We perform our experiments on NVIDIA Kepler GPU card GTX680 with CUDA computing capability 3.0. It has 8 streaming multiprocessors, with 192 cores on each of them, and it has 1536 cores in total. Every streaming multi-processor is equipped with 65536 registers and 48KB shared memory. The host machine runs 64-bit Linux with kernel version 3.1.10 and the CPU is an Intel Core i7-3370 at 3.7GHz.

We implemented the techniques discussed in Section 3. For every benchmark program, we collect the results for the original non-atomics implementation, and for the atomics involved implementation with different collision reduction techniques. We denote the atomic-collision-to-computation functions as *atomicSR* and *atomicVR* respectively for "atomic scan and reduction", "atomic vote and reduction". They are mainly for intra-warp collision elimination. We denote the atomic-collision-to-scatter approach as *atomicSS*, representing "atomic sort and scatter". We also obtain the results for the program version with naive atomics implementation. We record the kernel running times and get performance counter information with CUDA profiler 5.0. As of benchmark programs, we use five kernels that are fundamental kernels used in many applications, including: histogramming [8], merge-sort, page-view count [5], parallel summation, sparse matrix vector multiplication [3]. All these benchmarks are implemented as highly optimized GPU code. They are either from CUDA linear algebra library for sparse matrices [3] or have utilized the techniques proposed in published papers [5]. For every input, we evaluated at least five different inputs for different collision density. We describe our results for every benchmark first and summarize all of them at the end of this section.

5.1 Image Histogramming

Image processing applications extensively use the *histogram* kernel, which counts the frequency of the color for every pixel taken from an input image. We used the image histogramming benchmark optimized by the authors in [8]. The authors provided warp-private histogram implementation, which uses warp-private histograms to store partial histogram results, and thread-private histogram implementation, which does not use any atomics. We extended this benchmark by adding block-private histogram implementation in order to test our techniques for removing both intra-warp and inter-warp atomic collision. In both block-private and warp-private implementation, atomic operation is used. We choose to present the results on block-private for the atomics version as it almost always out performs warp private implementation. For the non-atomics implementation, we use the thread-private implementation in [8]. We performed experiments on 37 different images with different

Input	Speed Up	CMR	VR/SR	Code %
Best case	14.01	0.98	VR	125
Worst case	0.78	0.99	VR	125

Table 1. Histogram Benchmark Results

Input	Speed Up	VR/SR %	Code %
Best case	1.63	SR	104
Worst case	0.8	SR	104
Avg. case	1.45	SR	104

Table 2. Merge Sort Benchmark Results

atomic collision distribution. The best case in Table 1 represents the input for which we get the best performance, and the worst case corresponds to the input that provides the worst performance. The *CMR* column represents the cache miss rates for the best case and the worst case. The *Code %* represents the total number of lines of code in the atomics kernel divided by the non-atomics kernel (here we used the thread private version, which in fact has less code because of less communication, and there is no control divergence). The column *VR/SR* denotes which intra-warp collision reduction technique is used for this benchmark. From the table we can see that the best speedup 14.01 and the worst speedup is 0.78, which means we can reduce the code size significantly while preserving satisfiable performance.

We first show the results in Table 1, we show the best case speed up, the worst case speed up and the average speedup as well as their cache miss rates, the percentage of collided threads and the change in code size after we use atomic implementation. The speedup is the speedup when we use atomicVR/SR + atomicSS.

5.2 Merge Sort

We use the merge sort from CUDA Thrust library (*Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL)*) as the baseline non-atomics implementation. Similarly, this kernel is highly optimized. It keeps performing local sort and then merge the partially sorted results at different levels until the whole array is sorted. The original implementation is non-atomics involved. In the atomic version, we use *atomicAdd* to count the frequency of every element, and associate the frequency with every element, and carries it to next level local sort and merge. We tested the atomic version with five different collision level for a large input array and we compare it against the original non-atomics version using the total amount of time for all the kernel invocations. Similarly, we show the best case, worst case and average case information in Table 2 and then we show the overall speedup in the summary section 5.6.

5.3 Page View Count

Page View Count is an *map-reduce* application to track the number of unique visitors to a given web page. It is from the GPU *map-reduce* benchmark suite Mars [5]. 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. We extended this benchmark by performing block-level bitonic sorting and atomic reduction between the mapping and reduce phases of the first iteration of Page View Count, thus eliminating all duplicate entries within each block prior to global sorting. Our block-level reduction invokes relatively very little overhead and vastly improves the performance of global sorting which is by far the most constricting bottleneck of Mars. We show the best case, worst case and average case in Table 3. We use more code here because we added extra code performed the reduction of number of elements to be sorted.

Input	Speed Up	CMR	VR/SR	Code %
Best case	7.62	0.03	SR	140
Worst case	0.94	0.28	SR	140

Table 3. Page View Count Benchmark Results

Input	Speed Up	VR/SR	Code %
Best case	2.81	SR	100
Worst case	0.93	SR	100

Table 4. Parallel Summation Benchmark Results

Input	Speed Up	CMR	VR/SR	Code %
Best case	2.63	0.75	SR	29
Worst case	0.86	0.97	SR	29

Table 5. Sparse Matrix Vector Multiplication Benchmark Results

5.4 Summation

The parallel sum kernel is from CUDA computing SDK 5.0 , which is also carefully optimized with respect to all different factors such as shared memory bank conflicts, loop unrolling and etc. The original version performs local sum at block level at every different iteration. It then output the partially sorted results for every block in the output array buffer, which becomes the input array for the next level of local sum for every block. We injected the atomics at the level when the partially summation are written back to the global memory output array. Instead of writing to the block private data objects, we let the threads atomically add it to a more compacted output array. We change the level of atomic collision by changing the size of the compacted output array. There we have tested seven different inputs at different collision level. We take the time of all the kernel invocations that are needed to finish the summation of a large array. We show the results in Table 4.

5.5 Sparse Matrix Vector Multiplication

Sparse matrix vector multiplication is from the CUSP [3], an open source C++ library of parallel algorithms for sparse linear algebra and graph computations on GPUs. This kernel uses scratch-pad memory to cache the frequently used element to improve the performance. Its implementation is much faster than its CPU counterpart. In the atomics implementation, we replace all occurrences of *atomicAdd* with our *atomicSR* version. We also performed collision to scattering experiment. We use 17 sparse matrices from matrix market [1]. We present the best case, worst case and the average case in Table 5.

Overall, when the matrix is very sparse, the naive atomic implementation outperforms the non-atomics version already, and our optimized version is similar to the naive implementation.

5.6 Summary

We present the overall speedup for all benchmarks with all different atomic collision reduction techniques in Fig. 7. We average the speedup for all different inputs we have tested for every benchmark. In this figure, we show five different versions for every

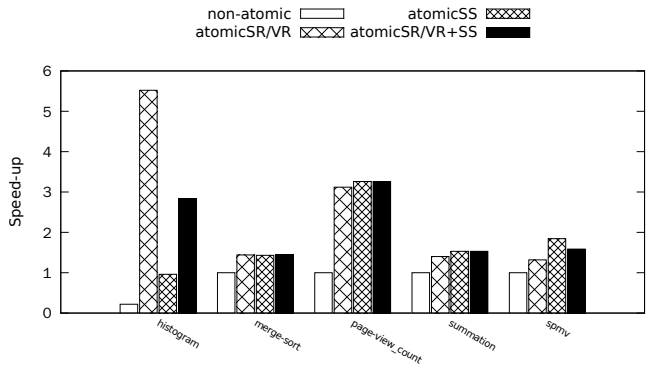

Figure 7. Summary of average speed up for all benchmarks

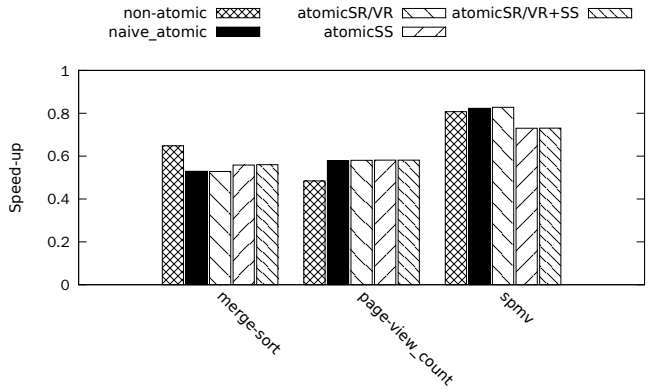

Figure 8. Summary of last level cache performance for benchmarks histogram, page view count and spmv

benchmark. They are respectively: (1) non-atomic version, which is the traditional implementation; (2) the naive atomic version, which is simply using intrinsic atomic instructions for every shared data object update; (3) the atomicSR or atomicVR version, which performs intra-warp atomic collision reduction with the *atomic scan and reduce* approach or the *atomic vote and reduce* approach; (4) the atomicSS version, which performs inter-warp atomic collision reduction with the *atomic sort and scatter* approach; (5) the atomicSR/VR + atomicSS approach, which first performs intra-warp atomic collision reduction and then inter-warp atomic collision reduction. We normalized the performance with the baseline of the traditional non-atomics implementation for every benchmark except *histogram*. For histogram, since the original thread private version takes too much scratch-pad memory with 256 bins per thread, it limits the number of active threads per SM to be 48, which is much less than the number of 2048 supported by the hardware. Therefore the naive atomics version runs much faster than non-atomics version. Further, there is already atomics implementation for histogram widely accepted by the developers. To ensure fairness of comparison, we use the *naive atomics* as baseline for different versions of histogram kernel. Henceforth, the non-atomic versions always have the speedup of 1 except for *histogram*.

In Fig. 7, we can see almost all the benchmarks have performance improvement on average. This is because even if there is serious atomic collision for some application inputs, with our atomic collision reduction approach, we can make the slowdown controllable. This is also shown in our worse case speed up for every individual benchmark earlier. Further, when the atomic collision is not that serious, for instance when the matrices are sparse, the speedup we can obtain from optimized atomic operation, is significant. Therefore, on average, we get performance improvement for every benchmark. It is also noteworthy that, for most applications the combination of *atomicSR/VR + atomicSS* performs the best among except for *spmv* – sparse matrix vector multiplication. This is because the sparsity of the matrices makes the *atomicSR* benefit less from the parallel reduction and with similar amount of the overhead. Therefore, the *atomicSS* version without *atomicSR* outperforms all other versions. In Fig. 8, we show the cache performance for three benchmarks, in which the re-scattering technique makes a difference in cache performance and therefore the overall performance, corresponding to the speedup number in Fig. 7.

Overall, we are able to achieve very good performance when we use massive atomic operations in computation when we use appropriate atomic collision reduction techniques. This demonstrates the potential of using atomic operations in computation for massively parallel GPUs.

6. Related Work

There are few studies that investigate the impact of massive atomics usage on GPGPUs. In conventional wisdom, atomic operations are known to be expensive compared to their non-atomic counterparts. Programmers have been trying their best to avoid using atomic functions in applications that need to deal with large amount of shared data structure such as parallel sorting [12].

One of the few relevant studies is on the hardware extension for efficient atomic vector support [7], where the authors study atomics for SIMD processors in Chip Multi-processors (CMP). The width of the SIMD lane is 4 or less. However, in most GPUs, the SIMD width is much than 4. Further, unlike the traditional SIMD machines, the speed of atomic operations have been greatly improved on general purpose GPUs. For instance, the performance of atomic memory operations, if no collisions, is almost the same as that of their non-atomic memory counterpart in NVIDIA's latest generation GPUs [9]. Other relevant software work includes application specific atomic usage, including GPU-MCML[2] – a highly optimized Monte Carlo (MC) code package for simulating light transport and GPU histogramming [10] [13] [11] [8].

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. We used statistical profiling to quickly identify the frequent atomic access that causes atomic collisions. The *atomic collision to scatter* approach is relevant to the job swapping idea used in control divergence and memory irregularity removal for GPU programs [15]. However, the ultimate goal is different. In this paper, we try to scatter same memory access threads as far from each as possible while [15] tries to gather same memory access threads as close as possible.

7. Conclusion

In this paper, we proposed the idea of using atomic operations extensively for computation purpose on many-core GPUs. We systematically studied the influence and solution of atomic collision in GPU programs. We proposed novel techniques to detect atomic collisions and eliminate them. We explored atomic collision reduction algorithms based on two fundamental techniques: the *atomic collision to computation* based and the *atomic collision to scatter* based. Overall, we have discovered that it not only provides good programmability but also good performance when we apply atomic operations for commutative operations in massively parallel programs.

References

[1] Matrix market. URL http://math.nist.gov/MatrixMarket/.

[2] E. Alerstam, W. C. Y. Lo, T. D. Han, J. Rose, S. Andersson-Engels, and L. Lilge. Next-generation acceleration and code optimization for light transport in turbid media using GPUs. *Biomedical Optics Express*, 1 (2):658–675, 2010.

[3] N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012. URL http://cusp-library.googlecode.com. Version 0.3.0.

[4] 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*, 29(12):84–89, Dec. 1996. ISSN 0018-9162.

[5] 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*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5.

[6] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. . URL http://doi.acm.org/10.1145/1555754.1555775.

[7] 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*, ISCA '08, pages 441–452, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. .

[8] 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*, GPGPU-4, pages 1:1–1:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0569-3.

[9] NVIDIA. Whitepaper - nvidia's next generation cuda compute architecture: Kepler gk110. NVIDIA. URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[10] NVIDIA. Cuda c programming guide 4.0. 2011.

[11] V. Podlozhnyuk. Histogram calculation in cuda. In *Technical Report*. NVIDIA, 2007.

[12] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. . URL http://dl.acm.org/citation.cfm?id=1586640.1587667.

[13] 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)*, pages 418–422, Gold Coast, Australia, Dec. 2007.

[14] 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.

[15] 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*, ASPLOS '11, pages 369–380, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1.