Performance Characterization and Optimization of Atomic Operations on AMD GPUs

Marwa Elteir, Heshan Lin, and Wu-chun Feng
Department of Computer Science
Virginia Tech
{*maelteir, hlin2, feng*}*@cs.vt.edu*

***Abstract*—Atomic operations are important building blocks in supporting general-purpose computing on graphics processing units (GPUs). For instance, they can be used to coordinate execution between concurrent threads, and in turn, assist in constructing complex data structures such as hash tables or implementing GPU-wide barrier synchronization.**

While the performance of atomic operations has improved substantially on the latest NVIDIA Fermi-based GPUs, system-provided atomic operations still incur significant performance penalties on AMD GPUs. A memory-bound kernel on an AMD GPU, for example, can suffer severe performance degradation when including an atomic operation, even if the atomic operation is never executed.

In this paper, we first quantify the performance impact of atomic instructions to application kernels on AMD GPUs. We then propose a novel software-based implementation of atomic operations that can significantly improve the overall kernel performance. We evaluate its performance against the system-provided atomic using two micro-benchmarks and four real applications. The results show that using our software-based atomic operations on an AMD GPU can speedup an application kernel by 67-fold over the same application kernel but with the (default) system-provided atomic operations.

***Keywords*-atomic operations, GPU, GPGPU, heterogeneous computing, MapReduce**

I. Introduction

While graphics processing units (GPUs) were originally designed to accelerate data-parallel, graphics-based applications, the introduction of programming models such as such as CUDA [13], Brook+ [1] and OpenCL [10] has made general-purpose computing on the GPU (i.e., GPGPU) a reality. One critical mechanism to support general-purpose computing on GPUs is atomic operations. Atomic operations allow different threads to safely manipulate shared variables, and in turn, enable synchronization and work sharing between threads on the GPU. For instance, atomic operations have been used to implement barrier synchronization on GPUs within a kernel [19]. These operations are also useful in constructing complex data structures such as hash tables [12] or building high-level programming frameworks such as MapReduce [8], [9].

Although the performance of atomic operations has been greatly improved on NVIDIA GPUs [16], using the atomic operations on AMD GPUs e.g., AMD/ATI Radeon HD 5000 series, can lead to significant performance penalties. As noted in the AMD OpenCL programming guide [3], including an atomic operation in a kernel may force *all* memory accesses to follow a slow *CompletePath* as opposed to a much more efficient *FastPath* even if the atomic operation is not executed at all.[1] For example, the bandwidth of a simple copy kernel, as shown in Figure 1, drops from 96 GB/s to 18 GB/s when an atomic add operation is included.

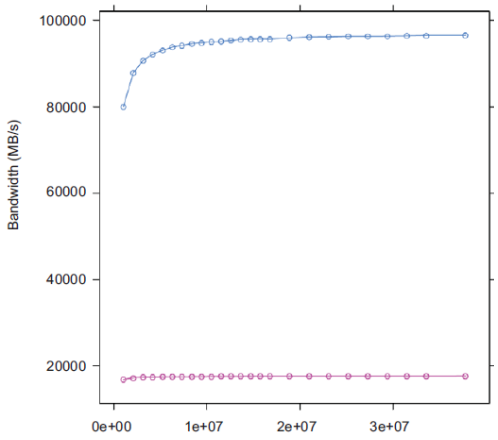

Figure 1. FastPath (blue) vs CompetePath (red) using float1 [3]

In this paper, we first seek to quantify the performance impact of using atomic operations on AMD GPUs. We then propose a novel software-based implementation of the *atomic add* operation, which can significantly improve application performance by preventing a kernel from using the degraded memory-access path. Together with a detailed discussion on the design of our software-based atomic add, we present performance evaluation of our approach with two micro-benchmarks and four representative applications. The results show that our software-based atomic add operations on an AMD/ATI Radeon HD 5870 GPU can speedup an application kernel by 67-fold over that same kernel but with the (native) system-provided atomic add.

II. Background

In this section, we give some background on the AMD GPU architecture with a focus on its memory access design.

A. AMD GPU Architecture

An AMD GPU comprises groups of compute units. Each compute unit has several cores named *stream cores*, and each stream core contains several processing elements.

[1]Details of the CompletePath and FastPath will be discussed in Section II.

These processing elements are the main computational units that perform integer and floating-point operations. All stream cores in a single compute unit execute the same sequence of instructions; however, different compute units can execute different instructions independently. A stream core is a five-way VLIW processor that contains five processing elements and one branch execution unit. As such, up to five scalar operations can be issued in a single VLIW instruction. Double-precision, floating-point operations are executed by connecting two or four processing elements.

The AMD GPU has off-chip device memory called global memory; data from the host is transferred to this memory via PCIe. The global memory consists of several memory channels, each having its own memory controller. Each memory channel has a L2 cache shared between different compute units. Moreover, all stream cores in a single compute unit share an on-chip memory named "Local Data Store" (LDS), whose bandwidth is significantly higher, i.e., more than 14 times higher on the AMD/ATI Radeon HD 5870, than global memory. LDS is connected to the L1 cache, as shown in Figure 2. In addition, there are two other on-chip memories called texture memory and constant memory that are shared by all compute units.

The Radeon HD 5870 GPU consists of 20 compute units, where each compute unit has 16 stream cores. Each stream core in turn has five processing elements. The global memory is divided into 8 memory channels that have an interleave of 256 bytes, so a linear burst switches channels every 256 bytes. The LDS per compute unit is 32KB; L1 cache is 8KB per compute unit; and L2 cache is 512KB.

B. FastPath vs. CompletePath

On AMD GPUs, specifically the Radeon HD 5000 series [3], two independent memory paths connect the compute units to global memory: CompletePath and FastPath, as shown in Figure 2. The effective bandwidth of the FastPath is significantly higher than that of the CompletePath. In addition, each path is responsible for different memory operations; FastPath performs loads and stores to data whose sizes are multiples of 32 bits, whereas the CompletePath performs advanced memory operations, e.g., atomic operations and accesses to data types that are not 32 bits in length.

Executing a memory load through the FastPath is done directly by issuing a single machine-level instruction. However, a memory load through the CompletePath is done through a split-phase operation. First, the old value is atomically copied into a special buffer. The executing thread is then suspended until the read is done, which may take hundreds of cycles. Finally, a normal load is executed to read the value from the special buffer.

The selection of memory path is statically done by the compiler on AMD GPUs. The strategy used by the compiler maps all data accesses by a kernel to a single "Unordered Access View" (UAV) to enable the compute units to store results in any arbitrary location. As a result, including one atomic operation in the kernel may force the compiler to use the CompletePath for all loads and stores in the kernel.

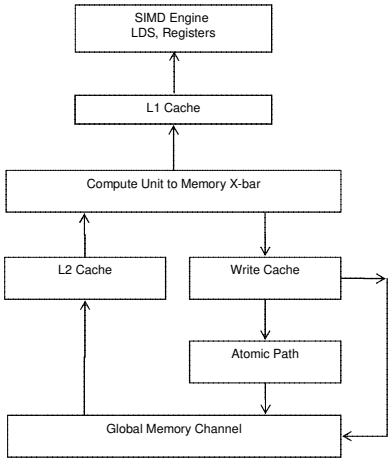

Figure 2. AMD GPU Memory Architecture

C. GPGPU with AMD

Currently, OpenCL [10] is the main programming language for AMD GPUs. Consequently, we choose OpenCL as our implementation tool in this paper. Using OpenCL terminology, each instance of a kernel running on a compute unit is called a *workitem*. All workitems are grouped into several groups named *workgroups*. Workitems in a workgroup are executed in *wavefronts*. Several wavefronts are executed concurrently on each compute unit to hide memory latency. Specifically, the resource scheduler switches the executing wavefront whenever the active wavefront is waiting for a memory access to complete.

III. Quantifying the Impact of Atomic Operations

To quantify the performance impact of atomic operations on memory access time, we run the simple kernel code, shown in Figure 3, on the Radeon HD 5870 GPU. The code includes only two instructions; the first is an atomic add operation to a global variable, and the second is a memory transaction that reads the value of the global variable and writes it to an element of an array.

```
__kernel void Benchmark (__global uint *out,
                         __global uint *outArray)
{
    int tid = get_global_id(0);

    // Safely incrementing a global variable
    atom_add(out,tid);

    /* Writing the value of the global variable
       to an array element */
    outarray[tid]=*out;
}
```

Figure 3. A simple copy kernel with atomic add operation

We measure the kernel execution time of three versions of the aforementioned kernel, as shown in Figure 4. The first version contains only the atomic operation. The second contains only the memory transaction. The third contains

both. Ideal represents the sum of the execution times of the atomic-only and the memory transaction-only versions.

By analyzing the ISA code, we found that the number of CompletePath memory accesses is 0 and 3 for the second and third versions, respectively. As a result, the memory access time increases significantly by 2.9-fold and 69.4-fold for 8 and 256 workgroups, respectively, when including the atomic operation. Note that, as the number of memory transactions in the kernel increases, the impact of accessing the memory through the CompletePath is exacerbated, as discussed in Section VI.

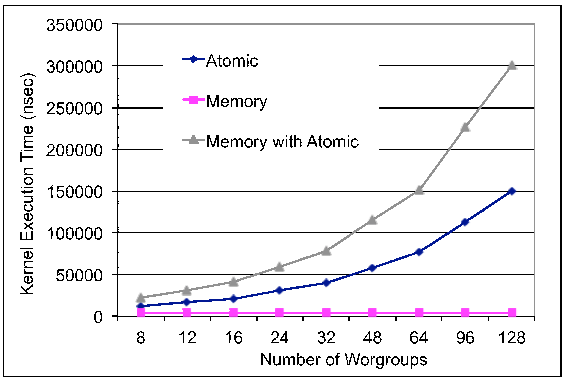

Figure 4. Kernel execution time of the simple copy kernel

Based on the above results, our goal in this paper is to develop an efficient software-based atomic operation that can efficiently and safely update a shared variable, and at the same time, does not affect the performance of other memory transactions.

IV. Software-Based Atomic Add

Here we present the design details of our software-based atomic add operation.

A. Overview

In Khronos's OpenCL specification [10], atomic operations have six types: add, subtract, increment, decrement, exchange, and compare then exchange. Two atomic operations, add and compare then exchange, can be viewed as the core for the other operations.

In this paper, we only propose a software-based version of atomic add and leave the rest for the future work. However, the subtract, increment, and decrement atomic operations can be implemented using an approach similar to our proposed method.

Implementing atomic add on GPUs is tricky because of the lack of efficient synchronization primitives on GPUs. One straightforward approach uses a master-slave model to coordinate concurrent updates at the granularity of threads. As shown in Figure 5, three arrays, i.e., *address array*, *increment array*, and *shared variable array*, are maintained in global memory. Each thread executing the software atomic add operation writes the increment values to a shared variable to the increment array and the address of the shared variable to the address array. Note that storing the address of a shared variable enables support for multiple shared variables in a kernel. A dedicated master/coordinator thread, which can be run in a separate workgroup, continuously spins on the address array. Once the master thread detects any thread executing the atomic operation, it updates the corresponding shared variable using the address and the increment value stored. Once the update is finished, the master thread resets the corresponding element of the address array to 0, signaling the waiting thread, busy waits on its corresponding element until the update is finished. Since only one thread is doing the update, the atomicity is guaranteed.

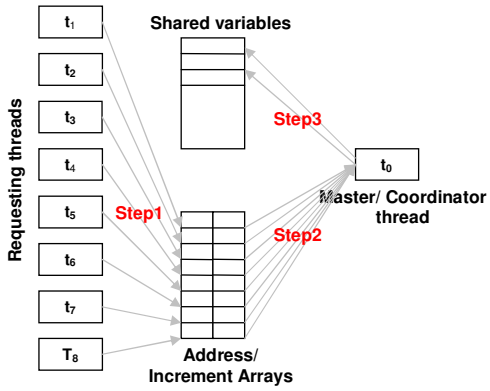

Figure 5. High level illustration of handling the software atomic operation

However, in this basic implementation described above, the master thread can easily become a performance bottleneck because of the serialization of update calculation as well as the excess number of global memory accesses. In addition, maintaining one element per thread in the address and increment arrays can incur space overhead for a large number of threads. To address these issues, we introduce a hierarchal design that performs coordination at the level of wavefronts and parallelizes the update calculation across the current threads executing the software atomic add. Specifically, the increment array maintains one element per wavefront, so does the address array. Each wavefront first calculates a local sum of the increment values requested by the participant threads in the fast local memory,[2] then it stores the local sum to the increment array in the global memory. The first workgroup is reserved as the coordinator workgroup. Threads in the coordinator workgroup read the address and increment arrays in parallel and collaboratively calculate the update value. Note that the coordinator workgroup does not participate in the kernel computation, otherwise deadlocks may occur when threads diverge in the coordinator group. Such a hierarchical design can greatly reduce global memory transactions as well as parallelize the update computation.

[2]Local memory in OpenCL is equivalent to shared memory in CUDA.

One challenge in the hierarchal design is to support divergent kernels, in which case not all threads participate in the software atomic add. In this case, care must be taken to avoid potential deadlocks and race conditions. As we will explain in Section IV-B, we use system-provided atomic operations on local variables to coordinate between threads within a wavefront, leveraging the fact that atomic operations on local variables will not force memory access to take the CompletePath.

To guarantee that the coordinator will always be executed, our current implementation assumes that the number of workgroups used in the kernel does not exceed the maximum number of concurrently running workgroups. For the Radeon HD 5870, we have found that for a simple kernel, each compute unit (of the 20 compute units) can run up to seven workgroups, so the maximum number of workgroups supported by our implementation in this case is 140. This value can be easily calculated following a similar methodology to the one proposed by the CUDA occupancy calculator [14]. While we leave support for an arbitrary number of workgroups for future work, the current design is useful in practice by adopting a large number of threads.

B. Implementation Details

By default, the atomic add operation returns the old value of the global variable just before executing the atomic operation. To support this feature, in addition to the hierarchical design described in Section IV-A, an old value of the shared variable is returned to each wavefront, which then calculates a return value for each participating thread with backtracking.

1) Data Structures: Four global arrays are used in our implementation. The number of elements of each array equals the number of wavefronts of the kernel, so each wavefront reads or writes to its corresponding element of these arrays. The first array is the *WavefrontsAddresses* array; whenever a wavefront executes an atomic operation to a shared variable, it writes the address of this variable to its corresponding element in this array. The second array is the *WavefrontsSums* array, which holds the increment of every wavefront to the shared variable. The third array is the *WavefrontsPrefixsums* array, which contains the old value of the global variable just before executing the atomic operation and is used by the requesting wavefront to generate the return value from the atomic add operation i.e., to mimic the system-provided atomic. The final array is the *Finished* array. Whenever a wavefront finishes its execution, it sets its corresponding element of this array to one.

2) Requesting Wavefront: Any thread executing our software-based atomic add operation passes through four steps, as shown in Figure 6. In the first step, the thread collaborates with other threads concurrently executing the atomic add operation to safely increment the wavefront's increment using local atomic add (line 13, 15, 16, and 18). In the second step, only one thread called the *dominant thread* writes the increment and address of the shared variable to the global memory (lines 22-26), i.e., *WavefrontsSums*, and *WavefrontsAddresses*, respectively. Since threads of any wavefront may diverge, the atomic operation may not be executed by all threads in the wavefront. Consequently, instead of fixing the first thread of the wavefront to write to the global memory, the first thread executing the local atomic add operation is chosen to be the *dominant thread* (line 14, 15, 17, and 18). In the third step, the thread waits until the coordinator workgroup handles the atomic operation and resets the corresponding element of the *WavefrontssAddresses* array (lines 29-32). Once this is done, the *WavefrontsPrefixsums* array contains the prefix sum of this wavefront, and every thread in the wavefront then generates its prefix sum and returns (line 36).

```
1  int  software_atom_add(__global int *X,  int Y,
2  __local int *LocalSum, __local int *ThreadsNum,
3  __global int *WavefrontsAddresses,
4  __global int *WavefrontsSum,
5  __global int *WavefrontsPrefixsum)
6  {
7     //Get the wavefront global and local ID
8    int wid = get_global_id(0) >> 6;
9    int localwid = get_local_id(0) >> 6;
10
11    /* Safely incrementing the wavefront increment and
12       threads number */
13    LocalSum[localwid]  =  0;
14    ThreadsNum[localwid]  =  0;
15    mem_fence (CLK_LOCAL_MEM_FENCE);
16    int threadSum  =  atom_add(&LocalSum[localwid],Y);
17    int virtualLid  =  atom_inc(&ThreadsNum[localwid]);
18    mem_fence(CLK_LOCAL_MEM_FENCE);
19
20    /* The first thread only writes the sum back to the
21       global memory */
22    if (virtualLid  ==  0)      {
23         WavefrontsSum[wid]  =  LocalSum[localwid];
24         WavefrontsAddresses [wid]  =  X;
25     }
26     mem_fence(CLK_GLOBAL_MEM_FENCE);
27
28   //Wait until the coordinator handles this wavefront
29     while(1)  {
30         mem_fence(CLK_GLOBAL_MEM_FENCE);
31         if (WavefrontsAddresses [wid]  ==  0) break;
32     }
33
34    /* Generate the retrun value and re-initialize the
35       variables */
36    int  ret =  WavefrontsPrefixSum[wid]  +  threadSum;
37    if (virtualLid  ==  0) {
38       LocalSum[localwid]  =  0;
39       ThreadsNum[localwid]  =  0;
40       mem_fence(CLK_LOCAL_MEM_FENCE);
41     }
42     return ret;
43 }
```

Figure 6. Code snapshot of software atomic add operation

3) Coordinator Workgroup: For convenience, the functionality of the coordinator workgroup is described assuming the number of wavefronts of the kernel equals to the number of threads of the coordinator workgroup. However, the proposed atomic operation handles any number of wavefronts that is less than or equal to the maximum number of concurrent wavefronts. Each thread of the coordinator workgroup is responsible for handling atomic operations executed by a specific wavefront. All threads in the coordinator group keep executing four consequent steps until all other wavefronts are done.

As shown in Figure 7, in the first step (lines 16-19), each thread loads the status of its wavefront into the local memory. The thread *lid* reads the status of the wavefront *lid*. More specifically, it reads *WavefrontsAddresses[lid]*, and *WavefrontsSums[lid]* and stores these variables into

the local memory, i.e., *Address[lid]* and *LocalSum[lid]*, respectively, as shown in lines 16, and 17. All threads are then synchronized (line 19) before the next step to ensure that the status of all wavefronts have been loaded.

In the second step (lines 23-36), the prefix sum of each wavefront and the increment of each unique address are generated. Each thread *lid* checks whether the wavefront *lid* executes the atomic operation or not by examining the address *Address[lid]* (line 23). If it is the only wavefront executing atomic operation to this address, the prefix sum is simply the value of this address (line 34), and the increment is the wavefront's increment represented by *LocalSum[lid]*. If there are several wavefronts concurrently executing atomic add for this address, the prefix sum of each wavefront and the increment of this address are generated using local atomic add operation i.e., atomic add to a local memory variable (lines 25-33). Note that the increment of the first of these wavefronts called *dominant wavefront* holds the increment of this address and the other wavefronts increments are set to zero (line 29) to ensure correctly incrementing the shared variable. All threads are again synchronized (line 36) to ensure that the increments of all wavefronts are used to calculate the increments of the global variables.

In the third step (lines 40-46), the global variables are safely updated and the blocked wavefronts are released. Specifically, each thread *lid* checks whether the wavefront *lid* executes the atomic operation or not by examining the address *Address[lid]* again (line 40). If it is a requesting wavefront, the thread *lid* sets *WavefrontsAddresses[lid]* to zero (line 44) to release this wavefront. If it is a *dominant wavefront*, its global variable is safely updated (line 41). Also, the local address and increment of this workgroup are reinitialized (line 42, and 43).

Finally, each thread re-evaluates the termination condition by calculating the number of the finished wavefronts (lines 50-54). If all wavefronts are done, the thread terminates.

```
1  void AtomicCoordinator(__local int *Address,
2  __local int *LocalSums,
3  __global int *WavefrontsAddresses,
4  __global int *WavefrontsSums,
5  __global int *WavefrontsPrefixsums,
6  __global int *Finished)
7  {
8   //Get thread ID in workgroup, and number of wavefronts
9    int lid = get_local_id(0);
10   int wavefrontsPerWorkgroup = get_local_size(0) >> 6;
11   int wavefrontsNum = get_num_groups(0) *
12                       wavefrontsPerWorkgroup;
13
14   while (1)    {
15     //1- Read the status of the wavefronts
16     Address[lid]   =  WavefrontsAddresses [lid];
17     LocalSum[lid]  =  WavefrontsSums[lid];
18     __global int * X  =  (__global int*)Address[lid];
19     barrier(CLK_LOCAL_MEM_FENCE);
20
21     /* 2- Safely generate the wavefronts prefixsums and
22          the increment of each unique variable     */
23    if ((lid < wavefrontsNum) && (Address[lid] > 0 )){
24     int  replaced   =   0;
25     for (int  k = 1;  k < lid ;  k++){
26      if (Address[lid]   ==   Address[k]) {
27       int temp = atom_add(&LocalSum[k], LocalSum[lid]);
28       WavefrontsPrefixSum[lid]  =  *X +  temp;
29       LocalSum[lid]  =  0;
30       Replaced  =  1;
31       break;
32      }
33     }
34     if (replaced == 0) WavefrontsPrefixsum[lid]  = *X;
35    }
36    barrier(CLK_LOCAL_MEM_FENCE);
37
38    /* 3- Safely increment the global variable and
39         release the blocked wavefronts */
40    if ( Address[lid]  >  0) {
41       if ( LocalSum[lid] > 0) *X  +=  LocalSum[lid];
42       Address[lid]   =   0;
43       LocalSum[lid]   =   0;
44       WavefrontsAddresses [lid]   =   0;
45    }
46    mem_fence(CLK_GLOBAL_MEM_FENCE);
47
48    //4- Check for exiting
49    int count  =  0;
50    for(int i = wavefrontsPerWorkgroup; i <
51           wavefrontsNum;  i++)
52              if (Finished[i]  ==  1) count++;
53    if (count == wavefrontsNum - wavefrontsPerWorkgroup)
54              break; //All wavefronts are done
55   }
56 }
```

Figure 7. Code snapshot of coordinator workgroup function

C. Discussion

We have taken great care in our design to ensure its correctness. Within a requesting wavefront (Figure 6), one design challenge is to select the dominant thread in divergent kernels. Since all threads within a wavefront are executed in a lock-step manner, using atom_inc on a variable in local memory can guarantee only one thread is chosen as the dominant thread. Our implementation also maintains separate local sums for different wavefronts; if a local sum is shared between wavefronts, a race condition can occur when threads from different wavefronts try to update the same local sum.

Another design challenge is to ensure that data is correctly exchanged between different workgroups. According to [17] and [19], the correctness of implementing a GPU primitive that requires inter-workgroup communication cannot be guaranteed until a consistency model is assumed. Xiao et al. [19] solved that by using __threadfence() function that ensures the writes to global memory by any thread is visible to threads in other blocks (i.e., workgroup in OpenCL). OpenCL does not have an equivalent to the __threadfence function. The mem_fence function in OpenCL only ensures that the write of a thread is visible to threads within the same workgroup. Fortunately, mem_fence guarantees the order that the memory operations are committed [10]. That means, for two consecutive memory operations A and B issued by a thread to a variable in the global memory, if mem_fence is called between them, once B is visible to threads in other workgroups, A will be visible as well because A is committed to the global memory before B. The correctness of our implementation in data exchange between different workgroups is achieved by the memory consistency provided by mem_fence.

Finally, although our implementation allows different wavefronts to concurrently execute atomic operation to different variables, threads within the same wavefront should concurrently execute the atomic operation to the same variable, since the status of each wavefront is represented by only one element in the global arrays. We believe that this requirement can be satisfied by restructuring the code and utilizing the shared memory.

V. Model for Speedup

In this section, we derive a model representing the speedup of our software-based atomic over the system-provided atomic for both divergent and non-divergent kernels. For simplicity, this model assumes that there is only one wavefront per workgroup.

In general, any GPU kernel involves three main steps; reading the input data from the global memory, doing some computations, and writing the results back to the global memory. The first and third steps are memory accesses, the second step can be divided into general computations and atomic-based computations. So the total execution time of atomic-based kernels is composed mainly of three components: memory access time, atomic execution time, and computation time. The software-based atomic operation affect only the first and second terms. Due to space limitations, we discuss only the atomic operation speedup that affect the second term, because the speedup of the memory access can be easily derived.

To derive the speedup of atomic operations, we need to consider the details of handling atomic operations using system-based and software-based approaches. Executing one system-provided atomic operation concurrently by several threads is done serially, and hence requires $N \cdot t_1$, where N is the number of threads concurrently executing the atomic operation. For non-divergent kernels, N equals the total number of threads in the kernel. Moreover, t_1 is the time to modify a global variable through the CompletePath. By neglecting the computations embedded within the atomic operation, t_1 can be replaced with t_{comp}, where t_{comp} is the time to execute a memory transaction through the CompletePath. So, the time required to execute the system-provided atomic, $t_{a_{system}}$, can be represented as:

$$t_{a_{system}} = N \cdot t_{comp} \tag{1}$$

Executing a software-based atomic operation can be represented by:

$$t_{a_{software}} = t_{RWGI} + t_{CWG} + t_{RWGP} \tag{2}$$

Where t_{RWGI} is the time needed for the requesting workgroup to generate its increment and updates the global arrays (section IV-B2), T_{CWG} is the time required by the coordinator workgroup to generate the prefix sums and update the shared variables (section IV-B3), and finally t_{RWGP} is the time needed by the requesting workgroup to generate the prefix sum and return from the atomic operation (section IV-B2).

Since the wavefront's increment is calculated using atomic add operation to shared memory (lines 13-18 in Figure 6), then t_{RWGI} can be represented by $2 \cdot N_c \cdot t_l + 2 \cdot t_{fast}$, where N_c is the number of threads per workgroup concurrently executing the atomic; t_l is the time to modify a variable in the shared memory; and t_{fast} is the time to execute memory transaction through the FastPath. And $2 \cdot t_{fast}$ is the time for writing the address and the increment to the global arrays (line 22-26 in Figure 6). Moreover, t_{CWG} can be represented by $5 \cdot t_{fast} + N_{cwg} \cdot t_l + \frac{N_{wg}}{2} \cdot t_l$, the first term corresponds to reading the workgroups increments, and addresses, writing the prefix sums to the global memory, updating the workgroup's address and shared variable. The second term corresponds to the time needed to generate the global increment using atomic add operation to the shared memory (line 27 in Figure 7), where N_{cwg} is the number of concurrent workgroups executing the atomic operation. The third term is time needed to check the value of local addresses (lines 25 and 26 in Figure 7), where $\frac{N_{wg}}{2}$ is the average number of comparisons until reaching the *dominant wavefront*. Finally, t_{RWGP} equals $2 \cdot t_{fast}$ because it requires only reading the address and the prefix sum from the global memory.

From the above discussion:

$$t_{a_{software}} = (\ (2 \cdot N_c + N_{cwg} + \tfrac{N_{wg}}{2}) \cdot x_l + 9) \cdot t_{fast} \tag{3}$$

where $x_l = \frac{t_l}{t_{fast}}$ and it is less than one by definition. For non-divergent kernels, we can substitute N in equation 1 with $N_c \cdot N_{cwg}$ and t_{comp} by $x \cdot t_{fast}$,where x = $\frac{t_{comp}}{t_{fast}}$ is the speedup of a single memory access when using FastPath relative to the CompletePath. Then $t_{a_{system}}$ can be represented by:

$$t_{a_{system}} = N_c \cdot N_{cwg} \cdot x \cdot t_{fast} \tag{4}$$

By comparing equation 3 by equation 4, we can see that the atomic operations speedup $\frac{t_{a_{system}}}{t_{a_{software}}}$ increases significantly as the number of workgroups increases. Furthermore, for divergent kernels, the speedup is smaller than that of non-divergent kernels, because $t_{a_{system}}$ is proportion to the number of threads concurrently executing the atomic operation, but $t_{a_{software}}$ remains almost the same.

VI. Performance Evaluation

All of the experiments are conducted on a 64-bit server with Intel Xeon e5405 x2 CPU and 3GB RAM. The attached GPU device is ATI Radeon HD 5870(Cypress) with 512MB of device memory. The server is running the GNU/Linux operating system with kernel version 2.6.28-19. The test applications are implemented using OpenCL 1.1 and built with AMD APP SDK v2.4.

In all experiments, three performance measures are collected. The first is the total execution time in nano-seconds. The second is the ratio of FastPath to CompletePath memory transactions, and the third is the ALU:Fetch ratio that indicates whether the kernel is memory-bound or compute-bound. Stream kernel analyzer 1.7 is used to get the second and third metrics. For the second metric, the equivalent ISA code of the OpenCL kernel is generated, then all memory transaction are counted. MEM_RAT, and MEM_RAT_CACHELESS transactions are considered as CompletePath and FastPath transactions respectively [3]. Note that these metrics do not capture runtime information. For instance, the absolute numbers of memory transactions following different paths are not revealed by these metrics.

Each run is conducted using 64, 128, and 256 threads per workgroup, and the best performance is used to generate the graphs.

Applications	Divergence	Atomic granularity
$MatrixMultiplication$ $(MapReduce)$	No	thread, wavefront
$KMeans(MapReduce)$	No	thread, wavefront
$ScalarProduct$	No	thread
$StringMatch$	Yes	thread

Table I
APPLICATIONS SUITE

A. Micro Benchmarks

The first micro benchmark aims at identifying the overhead of executing the system-provided atomic operation. The code of this microbenchmark is simple. Each thread only executes the atomic operation to increment a global variable by the global index of the thread. The kernel does not include any memory transaction, for our goal is to measure the overhead of executing the atomic operation by itself.

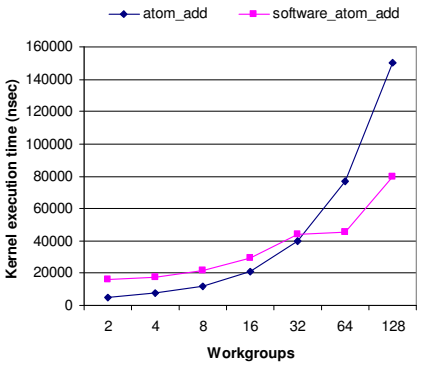

Figure 8. The execution time of system and software-based atomic

As shown in Figure 8, for small numbers of workgroups, (e.g., less than 32 workgroups), the performance of our software-based atomic is slower than the system-provided atomic by 0.5 fold on the average. As the number of workgroups increases, the speedup of our atomic increases until reaching 1.9 folds for 128 workgroups. This can be explained by the model discussed in Section V. As indicated in equations 3 and 4, the execution time of the system atomic operation increases linearly with the number of concurrent threads. However, the execution time of the software-based atomic is proportional to the number of concurrent wavefronts. Consequently, as the number of workgroups increases, our atomic add implementation can significantly outperforms the system one.

The second micro benchmark aims at studying the effect the impact of atomic operations on the performance of the memory transactions. The code of this micro benchmark looks very similar to the previous one, with another memory instruction being added.

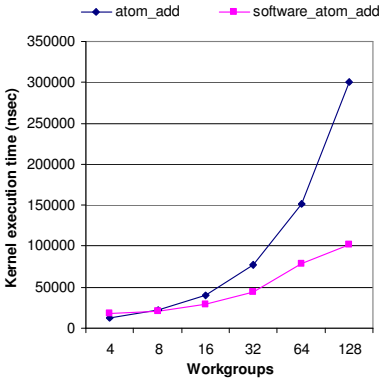

Figure 9. The execution time of system and software-based atomic when associated with memory transactions

As shown in Figure 9, the speedup of our atomic add implementation with regard to the system-provided atomic add operation increases significantly as as the number workgroups grows. This is due to that the performance of CompletePath is much worse than FastPath. Although our atomic add implementaiton performs more loads and stores to global memory compared to the system atomic add. Also, the ratio of complete to FastPath transactions is 3:0 and 0:10 for the system-provided atomic add and our software-based atomic add, respectively.

B. Applications Suite

1) MapReduce: MapReduce is a programming model that is originally proposed by Google to simplify parallel data processing on commodity clusters [5]. With MapReduce, programmers need to write only two functions, i.e., map and reduce. The parallel execution and fault tolerance will be handled by the MapReduce framework. MapReduce has been ported to GPUs in 2008 [7] by B. He et al. Since atomic operations were not available on GPUs when Mars was introduced, Mars uses a two-pass algorithm to coordinate output from different threads. In the first pass, the map/reduce function is executed to calculate the sizes of output records. Based on the size information, Mars can decide the write location of each output record using prefix summing. In the second pass, the map/reduce function is executed again to write the output records to global memory based on the calculated write locations. Such a two-pass design is not efficient because it introduces redundant computation. Consequently, state-of-the-arts implementations of MapReduce on GPUs [8], [9] rely on atomic add operations to safely write the intermediate and final output to global buffers.

In order to evaluate the performance impact of using atomic operations in MapReduce design, we first implemented a baseline MapReduce framework based on Mars. We then implement a single-pass output writing design using atomic operations. The atomic operations are applied in both the thread level and the wavefront level. In addition, we have implemented two applications over this framework: matrix multiplication and KMeans.

Matrix multiplication accepts two matrices X and Y as input, and produces matrix Z as output. Every element $z_{i,j}$ in Z is calculated by multiplying every element in row i of X with every element in column j of Y and summing these products. Implementing matrix multiplication over MapReduce framework requires implementing only the map function. Each map task corresponds to one thread, and it is responsible for emitting one element of the Z matrix. Since all map threads access the same number of elements of X and Y and executes the same number of operations, then matrix multiplication is an example of non-divergent kernels whose threads execute the atomic operation at the same time.

KMeans iteratively clusters a set of points. It accepts an initial set of clusters and a set of points. At each iteration, the distance between each point and the centroid of each cluster is calculated. The points are then assigned to the closest centroid, and a set of new centroids are calculated based on the point assignments. The above process is repeated until the results are converged. Implementing KMeans over MapReduce framework requires implementing both the map and reduce functions. Each map and reduce task corresponds to one thread. The map task determines the best cluster for one point, and the reduce task calculates the new centroid of each cluster based on the points attached to it. The Map phase consumes more than 50% of the total execution time, so in this experiment we only measure the execution time of the map phase. Note that, KMeans also is an example of non-divergent kernels whose threads execute the atomic operation at the same time.

2) Scalar Product: Scalar product is a simple vector arithmetic operation. Given two vectors X and Y, the scalar product X·Y is computed by multiplying every component of X with every component of Y and summing those products. In our implementation, every thread multiplies one component of X with one component of Y, then it executes atomic add operation to safely add the product to a global variable.

3) String Match: String match is an application that is extensively used in building search tools and search engines. Given a document containing text of any format and a keyword, string match searches for this keyword in the whole document and returns the occurring locations. In our implementation, each thread processes a trunk of the input file. Once a match is found between the keyword and the input, the thread writes the location of this word to a global output buffer shared among all threads. The global buffer is guarded by atomic add operations to avoid write conflicts from different threads. String match is an example of divergent kernels.

C. Application Performance

1) MapReduce: Matrix multiplication performance is shown in Figure 10. As we can see, the speedup of using software-based atomic add over the system atomic add increases as the input matrices get larger. Specifically, the speedup is improves from 0.62 folds for a 8X8 input 13.55 folds for a 256X256 input. The main reason is that for larger inputs, there will be more memory access, exacerbating the memory performance of using CompletePath. By analyzing the ISA, we realize that the ratio of the FastPath to CompletePath memory accesses is 30:0 and 3:28 for software-based atomic and system-provided atomic implementations, respectively.

Note that, since the number of workgroups is constrained by the maximum number of concurrent workgroups, for matrices of dimensions greater than 64X64, every thread manipulates several elements in the output matrix instead of one element.

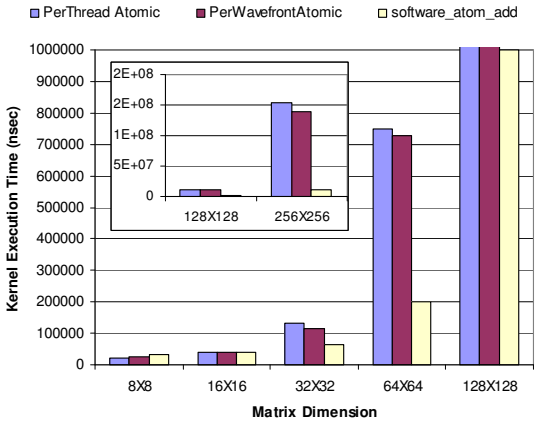

Figure 10. The execution time of Matrix multiplication using system and software-based atomic operation

For KMeans, we run it for different number of points ranging from 512 to 8192. As shown in Figure 11, the speedup gets improved from 15.52 folds for 512 points to 67.3 folds for 8192 points. Again, this is because of there are more memory accesses for larger inputs, amortizing the overhead of the software atomic add.

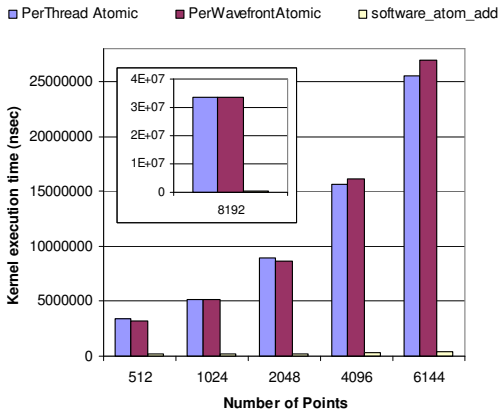

Figure 11. The execution time of map phase of KMeans using system and software-based atomic operation

2) Scalar Product: As shown in Figure 12, this application does not benefit from the use of software-based atomic. The average speedup of the software-based atomic compared to system-provided atomic is 0.61 fold. This is because of two main reasons. First, the number of memory transactions

is very small, i.e., only 3 according to the ISA. Second, the ALU:Fetch ration of this application is 2.88, suggesting that this application is not memory bound. As a result, the overhead of the software atomic add is not able to be compensated by improvement of memory access efficiency. Nonetheless, the speedup increases as the number of work-groups increases. Specifically, the speedup increases from 0.4 fold for 2 workgroups to 0.92 fold for 128 workgroups.

Scalar product can be viewed as a core operation that is used as a primitive to build more complex kernels. As the number of memory accesses in the kernel increases, the benefits of using the software-based atomic can be greater.

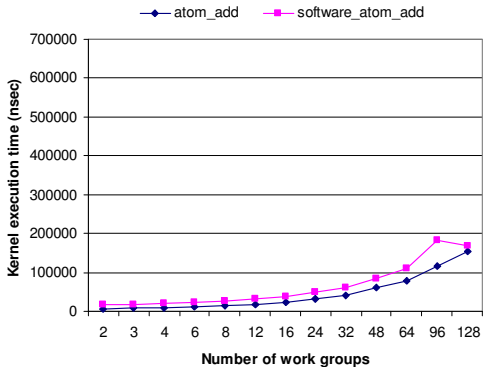

Figure 12. The execution time of scalar product using system and software-based atomic

3) String Match: We run String match using a dataset of size 4 MB [18] to search different keywords. For each keyword, we vary the number of workgroups from 32 to 128. As shown in Figure 13, the performance of the software-based atomic is better than that of the system-provided atomic in almost all cases for the first three queries. More specifically, the average speedup is 1.48 folds.

However, for the fourth query, the performance of our atomic is significantly worse than the system-provided atomic. This query returns significantly higher number of matches compared to the other queries. Specifically, the number of matches is 7, 87, 1413, and 20234 for first, second, third, and fourth query respectively. A larger number of matches requires more memory transactions to write the matches as well as more computations. We realize that writing the matches are done through the FastPath even when system-provided atomic is used, so increasing the number of matches only contributes to increase of the compute-boundness of the application. Note that the number of read operations are the same for four queries. In other words, the software atomic approach does help improve the memory read performance, thus we observe performance improvements for the first and second queries with less computation. For the fourth query, with more amounts of computation, the overhead incurred by the software atomic approach for writing results start to offset the benefit of using FastPath for read accesses.

By analyzing the ISA of both kernels using the software-based atomic and the system-driven atomic, we realize that the ratio of FastPath to CompletePath memory accesses is 12:0 and 1:19 for the software-based atomic and the system-provided atomic respectively. This result also reveals one important fact that is not explicitly mentioned in the AMD OpenCL Guide [3]; although in [3], they mentioned that non-32 bits memory transaction are executed through the CompletePath, in the kernel that uses the software-based atomic, all transactions are executed through the FastPath although input file is read character by character. In-depth mapping of OpenCL kernel instructions to ISA instructions have shown that only stores of char are executed through the CompletePath (loads of char are executed through the FastPath).

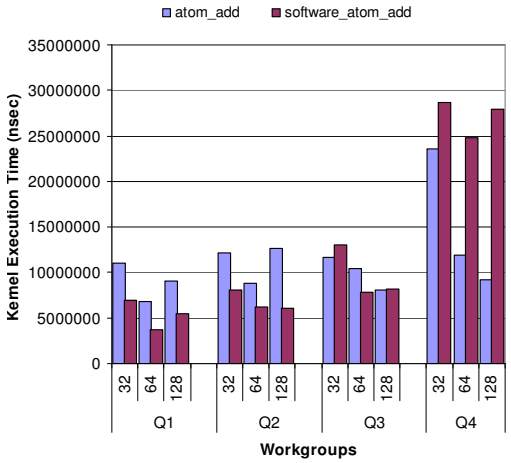

Figure 13. The execution time of string match using system and software-based atomic

D. Discussion

As we can see previously, the speedup of the non-divergent applications differs significantly from scalar product to matrix multiplication and KMeans. We realized that the speedup of these applications is directly proportional to two metrics. The first one is the number of memory accesses through the CompletePath. The CompletePath memory accesses are 3, 28, and 37 for scalar product, matrix multiplication, and KMeans, respectively. The second one is the ALU:Fetch ratio that reflects the contribution of memory accesses on the total execution time. As shown in Figure 14, the more memory-bound is an application, the better is the speedup of using software atomic add compared to the system atomic add.

Based on the above results, we believe that the application that can benefit from our software-based atomic should satisfy two conditions. First, the application's ALU:Fetch ratio should be relatively low. Second the number of memory transactions executed through the CompletePath should be large to compensate the overhead of the software-based atomic. We plan to investigate this more in our future work.

VII. Related Work

Our work is closely related to research studying the atomic behavior on GPUs and CPUs. In [17], Volkov et

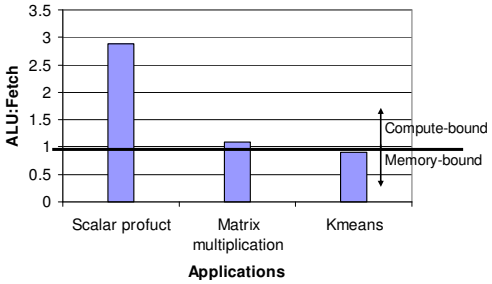

Figure 14. Ratio of CompletePath to FastPath memory accesses

al. proposed a global barrier synchronization on GPUs in order to accelerate dense linear algebra computations. Their approach does not use atomic operations. Instead, each thread manipulates one private variable, *arrival*, to indicate its arrival to the synchronization point. The first thread acts as the coordinator thread that spins on the *arrival* variables of all threads and announces the completion of the global synchronization. In [19], Xiao et al. improved the performance of Volkov's approach by using all threads in the first workgroup to simultaneously spin on the *arrival* variables. Furthermore, they proposed another lock-based synchronization that uses global atomic operations.

Suryakant et al. [15] proposed an efficient and scalable implementation of the split primitive for GPUs that use atomic operations to shared memory instead of using atomic operations to the global memory. To guarantee the correctness of their implementation, the ordering of threads executing atomic operations should be preserved. Since this is not supported by the system-provided atomics, they simulate ordered atomic operation by serializing the threads of a warp. However, its performance is 5 to 10 times slower than the hardware atomics in NVIDIA GPUs.

In addition to the above studies, another related research area is supporting mutual exclusive access to a critical section on CPUs. The classical problem of guaranteeing mutually exclusive access among a number of competing processes is originally presented in 1970 by Dijkstra [6], which depends on using system-provided atomic operations. There are also several studies that enable mutual exclusion on processors that do not have atomic test and set operations [11], [2], [4].

VIII. Conclusion and Future Work

In this paper, we first quantify the effects of using the system-provided atomic operations on the performance of kernels designed for AMD GPUs. Then we propose a novel software-based atomic operation that can significantly improve the performance of memory-bound kernels. We evaluate the proposed design of atomic add using four representative applications that follow different divergence patterns and ALU:Fetch ratio. The experimental results show that for memory-bound kernels, our software-based atomic add can deliver an application kernel speedup of 67-fold compared to one with a system-provided atomic add.

As for future work, we will further investigate a set of guidelines to decide when to use our software-based atomic operation. Second, we will study other approaches for implementing software-based atomic operations that support any number of workgroups.

Acknowledgment

This work was supported in part by NSF grant IIP-0804155 and an AMD Research Faculty Fellowship.

References

[1] AMD. Stream Computing User Guide. Website, April 2009. http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf.

[2] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. *The Interaction of Architecture and Operating System Design*, volume 26. ACM, 1991.

[3] ATI Stream Computing. OpenCL Programming Guide. Website, August 2010. http://developer.amd.com/gpu_assets/.

[4] B. Bershad, D. Redell, and J. Ellis. Fast Mutual Exclusion for Uniprocessors. In *5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 223–233. ACM, 1992.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems, Design & Implementation, OSDI*, 2004.

[6] E. Dijkstra. Solutions of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, 1965.

[7] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 260–269. ACM, 2008.

[8] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: Writing Parallel Program Portable between CPU and GPU. In *19th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 217–226. ACM, 2010.

[9] F. Ji and X. Ma. Using Shared Memory to Accelerate MapReduce on Graphics Processing Units. In *25th IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, 2011.

[10] Khronos Group. The Khronos Group Releases OpenCL 1.0 Specification. *http://www.khronos.org/news/press/releases*, 2008.

[11] L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems (TOCS)*, 5(1):1–11, 1987.

[12] M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *14th ACM Symp. on Parallel algorithms and Architectures*, pages 73–82. ACM, 2002.

[13] NVIDIA. NVIDIA CUDA Programming Guide-2.2. Website, 2009. http://developer.download.nvidia.com/compute/cuda/.

[14] NVIDIA CUDA. CUDA Occupancy Calculator. Website, 2007. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[15] S. Patidar and P. Narayanan. Scalable Split and Gather Primitives for the Gpu. Technical Report Tech. Rep. IIIT/TR/2009/99, 2009.

[16] D. Patterson. The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges. *NVIDIA Whitepaper*, 2009.

[17] V. Volkov and J. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *ACM/IEEE SC 2008*, 2008.

[18] Wikimedia Foundation Project. English-LanguageWikipedia. *http://download.wikimedia.org/*, 2010.

[19] S. Xiao and W. Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. In *24th IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, 2010.