Analysis of CUDA atomic operations

Ashish Jindal, Rutgers University

Abstract

The atomic operations in the CUDA are crucial to some implementations but previous benchmarks reveal that, for an operation as simple as an addition of a single variable, performance of atomic operations can be thousands of times slower than a traditional read-modify-write cycle.

In some previous projects - sparse matrix vector multiplication (spmv), we found that under the certain circumstances, cuda atomic instructions can also perform *faster* than non-atomic instructions. Besides that, other factors like sparsity pattern, storage schemes etc also may impact the performance of the implementation.

The goal of this project to to build an understanding of the performance of cuda atomic operations while varying other factors like storage schemes, sparsity pattern etc. and determine why the results are the way they are, and then repeating the cycle until we're out of surprises.

1. Proposed Experiments

An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. For example, atomicAdd() reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete.

Atomic operations in a parallel environment present a real challenge because they serialize execution. Instead of seeing an n Processor parallel speedup, applications that perform an atomic operation on a single counter will only exhibit a sequential runtime. In other words, incrementing a single counter with atomicAdd() means that the counter has to be locked, thus forcing all the parallel threads to stop and wait so they can individually perform the increment operation — one after the other.

We want to perform numerous experiments in varied settings to determine the cause of performance in results, starting off with simple experiment to gauge performance of atomicAdd().

1.1. Analyze the performance of a parallel counter

In this experiment A counter is initialized on the host and the copied to GPU, where it is incremented using atomicAdd(), then the result is copied back to CPU. We also, spread the atomicAdd() operations on multiple memory locations and analyze how it impacts the performance. This helps to study the performance of atomic operations on continuous memory locations vs discontinuous memory locations. Following are the exact test cases we used -

- Coalesced atomic addition on global memory.
- Atomic addition on a restricted address space in global memory.
- Atomic addiction on same address in global memory.
- Atomic addition, with all threads in warp operating on a single memory location.

We also repeat the above experiment for shared memory and analyze the performance of atomic operation on shared memory.

1.2. Matrix copy and transpose test

In this experiment, we test the benefits of shared memory (if any) while working on matrices. We use a 4096 x 4096 matrix and perform matrix copy and matrix transpose in various conditions -

- Matrix copy in global memory
- Matrix copy via shared memory
- Matrix transpose in global memory
- Matrix transpose in Shared memory
- Matrix transpose via shared memory and banks conflicts resolution.

By this experiment we can gauge the performance improvement in matrix operation if we use shared memory and can calculate the expected speedup for the application.

1.3. Sparse matrix vector multiplication

The Sparse Matrix-Vector Multiplication (SpMV) kernel computes a vector y as the product of a n by m sparse matrix A and a dense vector x. The massive parallelism of graphics processing units (GPUs) offers tremendous performance which can be exploited for problems like sparse vector multiplication. In this report we discuss 4 data reordering techniques and show how reordering of data can affect performance of the kernel when we have an implementation based on CUDA atomics.

- Column Order
- Row Order
- Random
- Tiled

Since the access pattern is different in all 4 re-ordering techniques, we get different performance results and we will try to explain why one is different from other.

2. Hardware specifications

Device	GeForce GT 630
CUDA Driver Version / Runtime Version	7.5 / 7.5
CUDA Capability Major/Minor version number	3.0
Total amount of global memory	2046 MBytes (2145235968 bytes)
Multiprocessors x (192) CUDA Cores/MP	192 CUDA Cores
GPU Clock rate	876 MHz (0.88 GHz)
Memory Clock rate	891 Mhz
Memory Bus Width	128-bit
L2 Cache Size	262144 bytes

Max Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536,65536), 3D=(4096,4096,4096)
Max Layered Texture Size (dim) x layers	1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	49152 bytes
Total number of registers available per block	65536
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Maximum sizes of each dimension of a block	1024 x 1024 x 64
Maximum sizes of each dimension of a grid	2147483647 x 65535 x 65535
Maximum memory pitch	2147483647 bytes
Texture alignment	512 bytes
Concurrent copy and kernel execution	Yes with 1 copy engine(s)
Run time limit on kernels	Yes
Integrated GPU sharing Host Memory	No
Support host page-locked memory mapping	Yes
Alignment requirement for Surfaces	Yes

Device has ECC support	Disabled
Device supports Unified Addressing (UVA)	Yes
Device PCI Bus ID / PCI location ID	1 / 0
Compute Mode	Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)

3. Results and Analyses

3.1 Atomic operation on a parallel counter

Following example demonstrates different access patterns in case of a system with 8 threads - labeled 0-7 and an array of size 9 - labeled 0-8.

Coalesced memory access

Array [0 1 2 3 4 5 6 7 8]

Thread [0 1 2 3 4 5 6 7 0]

Address restricted memory access

Array [0 1 2 3 0 1 2 3 0]

Thread [0 1 2 3 4 5 6 7 0]

Warp restricted memory access

Array [0 1 2 3 0 1 2 3 0]

Warp [0 1 2 3 4 5 6 7 8]

Same address access

Array [0 0 0 0 0 0 0 0 0]

Thread [0 1 2 3 4 5 6 7 0]

Following results were obtained in following settings -

Block Count - 128

Block Size - 8

Array size - $2^{\wedge 24}$

Restriction size - 32 values from index 0

Atomic Add Pattern	Time (ms)	atomic_transactions	l2_atomic_transactions	shared_store_transactions
coalescedAtomicOnGlobalMem	95.872	4130685	7969177	0
addressRestrictedAtomicOnGlobalMem	92.341	4194303	8388608	0
warpRestrictedAtomicOnGlobalMem	386.52	28293794	67108864	0
sameAddressAtomicOnGlobalMem	461.55	27842801	67108864	0
coalescedAtomicOnSharedMem	2349.17	0	0	26932301
addressRestrictedAtomicOnSharedMem	1521.68	0	0	21956456
warpRestrictedAtomicOnSharedMem	9868..7	0	0	188295473
sameAddressAtomicOnSharedMem	10946.5	0	0	196152349

The Kepler architecture improved (vs. the previous Fermi architecture) global memory atomics performance by resolving conflicts in L2. On the other hand, Kepler emulates shared memory atomics in software, and high collision tests and real images suffer from serialization issues.

However, atomic operations to *shared* memory remained essentially unchanged: both architectures implemented shared memory atomics using a lock/update/unlock pattern that could be expensive in the case of high contention for updates to particular locations in shared memory.

3.2 Matrix copy and transpose

Following results were obtained in following settings -

Block Count - 128

Block Size - 8

Matrix dimensions - 4096 x 4096

Iterations - 100

Implementation	Time (ms)	dram_write_transactions	gst_transactions_per_request
Matrix copy (Naive)	660.74	2097150	1
Matrix copy via shared memory	550.74	2097095	1
Matrix Transpose (Naive)	2240.46	5369437	32
Matrix Transpose via Shared Memory	1317.20	2097126	1
Matrix Transpose via Shared Memory (No bank Conflicts)	620.70	2097138	1

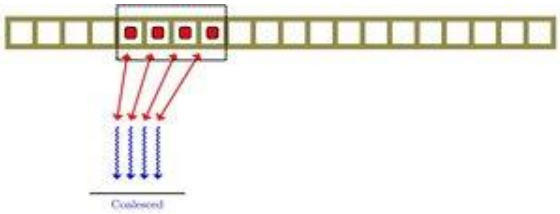
When accessing multidimensional arrays it is often necessary for threads to index the higher dimensions of the array, so strided access is simply unavoidable. We can handle these cases by using a type of CUDA memory called *shared memory*.

Shared memory is an on-chip memory shared by all threads in a thread block. One use of shared memory is to extract a 2D tile of a multidimensional array from global memory in a coalesced fashion into shared memory, and then have contiguous threads stride through the shared memory tile.

Unlike global memory, there is no penalty for strided access of shared memory.

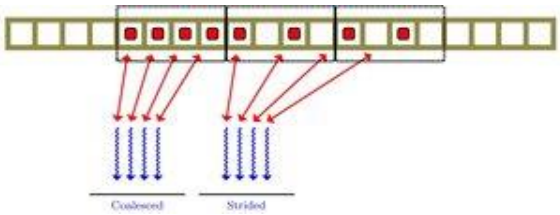
Memory is usually retrieved in large blocks from RAM.Some processing units will try to predict future memory accesses and cache ahead, while yet processing older parts of memory.Memory is cached in a hierarchy of successively larger-but-slower caches.

Therefore, making programs that can use predictable memory patterns is important. It is even more important with a threaded program, so that the memory requests do not jump all over; otherwise the processing unit will be waiting for memory requests to be fulfilled.



The memory accesses are close, and can be retrieved in one go/block (or the least number of requests).

However, if we increase the "*stride*" of the access between the threads, it will require many more memory accesses. Below: four more threads, with a stride of two.



Here you can see that these 4 threads require 2 memory block requests. The smaller the stride the better. The wider the stride, the more requests are potentially required.

3.3 Sparse Matrix Vector Multiplication

SPMV atomic (Column Sorted) -

Matrix	Time (256 x 8)	Time (384 x 5)	Time (512 x 4)	Time (768 x 2)	Time (32x32)
cant	19516 us	30608 us	37873 us	75165 us	5602 us
FullChip	262067 us	404004 us	496166 us	971418 us	82930 us
circuit5M_dc	192246 us	294600 us	363079 us	703680 us	57722 us
consph	28772 us	45173 us	56095 us	111257 us	8422 us
webbase-1M	29543 us	46118 us	57152 us	113311 us	8886 us

SPMV atomic (Row Sorted) -

Matrix	Time (256 x 8)	Time (384 x 5)	Time (512 x 4)	Time (768 x 2)	Time (32x32)
cant	22611 us	33534 us	40853 us	76583 us	8127 us
FullChip	288366 us	434661 us	526139 us	984783 us	91366 us
circuit5M_dc	204262 us	311806 us	381002 us	713856 us	59129 us
consph	33310 us	49680 us	60010 us	112975 us	12333 us
webbase-1M	31694 us	48993 us	60360 us	115927 us	9991 us

SPMV atomic (Random Order) -

Matrix	Time (256 x 8)	Time (384 x 5)	Time (512 x 4)	Time (768 x 2)	Time (32x32)
cant	27534 us	37776 us	43919 us	77948 us	14202 us
FullChip	361036 us	502275 us	577328 us	1022128 us	231784 us
circuit5M_dc	273203 us	376701 us	429312 us	747348 us	200228 us
consph	41290 us	56675 us	65962 us	115959 us	21726 us
webbase-1M	43115 us	58866 us	68549 us	120032 us	28855 us

SPMV atomic (Tiled) -

Using 32 x 32 config for block Number vs block size (Our pre-processing step to create tiled input takes lot of time, so data for very large matrices is missing!)

Matrix	Time (Tile Size = 16)	Time (Tile Size = 32)	Time (Tile Size = 128)	Time (Tile Size = 256)
cant	6197 us	6267 us	7047 us	7344 us
FullChip	-	-	-	-
circuit5M_dc	-	-	-	-
consph	9123 us	8995 us	10079 us	10490 us
webbase-1M	9206 us	9005 us	9105 us	9186 us

The performance is worst in case the data is randomly ordered and best when data is column sorted.

In case of column sorted data, all the column 1 values are ordered before column 2 and so on. This means that data reads on the vector ‘x’ will be clustered in column wise manner and hence would give better cache reads. When we are processing all column 1-2-3 values the processor, which are meant to be multiplied with corresponding 1st 2nd and 3rd value in the vector ‘x’ again and again, processor will automatically optimize the performance of reads by bringing the corresponding values of vector ‘x’ into texture cache, thus facilitating very good read performance on vector ‘x’.

For row ordered data, the problem is that the writes on output vector will not happen in parallel, as multiple threads will be processing data items for the same row which will end up simultaneously executing atomicAdd

on same memory locations multiple times in the output vector, thus making the processing sequential.

For tiled ordering, we get a decent performance. This ordering exploits data locality as well as address restricted atomic writes - which happen to be fast as shown in our previous experiments. The performance is close to column ordering and much better than row and randomly ordered data.

Also, if we increase the block size then performance improves, we can observe that performance for a 32x32 is much better than 512 x 4 configuration in all cases.

Refer run.sh for all the commands to run various configs.

3. References

[1] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[2] Nathan Bell and Michael Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004. December 2008.

[3] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[4] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In Proceedings of the 2Nd International Conference on Supercomputing, ICS '88, pages 140–152, New York, NY, USA, 1988. ACM.

[5] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, pages 521–532, New York, NY, USA, 2015. ACM. [13] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. Onthe-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 XVI, pages 369– 380, New York, NY, USA, 2011. ACM.

[6] http://www.drdobbs.com/parallel/atomic-operations-and-low-wait-algorithm/240160177.

[7] https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[8]http://www.nehalemlabs.net/prototype/blog/2014/06/23/parallel-programming-with-opencl -and-python-parallel-scan/.

[9] https://www.udacity.com/course/viewer#!/c-cs344/l-77202674/m-79993366.

[10] https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/

[11] http://docs.nvidia.com/cuda/maxwell-tuning-guide/#fast-shared-memory-atomics.

[12] http://www.strobe.cc/cuda_atomics/.

[13] https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/.

[14] Following paper also studies the performance of atomic operations in CUDA -
"Massive Atomics for Massive Parallelism on GPUs", Ian Egielski, Jesse Huang and Eddy Z. Zhang, ISMM'14