ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/327540610

Performance study of atomic tally methods for GPU-accelerated Monte Carlo dose calculation

Conference Paper · August 2018

CITATIONS
0

READS
117

5 authors, including:

Tianyu Liu
Rensselaer Polytechnic Institute
49 PUBLICATIONS **128** CITATIONS

SEE PROFILE

Noah Z Wolfe
Rensselaer Polytechnic Institute
14 PUBLICATIONS **44** CITATIONS

SEE PROFILE

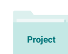
Hui Lin
Rensselaer Polytechnic Institute
20 PUBLICATIONS **5** CITATIONS

SEE PROFILE

George Xu
Rensselaer Polytechnic Institute
125 PUBLICATIONS **2,094** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Monte Carlo simulations for Interventional Radiologic organ dose estimates View project

Monte Carlo simulations for organ dose estimation of CT scans View project

All content following this page was uploaded by Tianyu Liu on 09 September 2018.

The user has requested enhancement of the downloaded file.

ANS RPSD 2018–20th Topical Meeting of the Radiation Protection & Shielding Division of ANS
Santa Fe, NM, August 26–31, 2018, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2018)

PERFORMANCE STUDY OF ATOMIC TALLY METHODS FOR GPU-ACCELERATED MONTE CARLO DOSE CALCULATION

Tianyu Liu, Noah Wolfe, Hui Lin, Christopher D. Carothers, and X. George Xu

Rensselaer Polytechnic Institute, 110 8th St, Troy, New York 12180, xug2@rpi.edu

Over the past several years, the graphics processing unit (GPU) technology has rapidly gained ground in scientific computing due to its outstanding performance and programmability. GPU implementation of Monte Carlo radiation transport for dose calculations has been reported by many investigators. The majority of these studies adopted single-precision floating point format because of the higher peak floating point operations per second (FLOPS) the GPUs can deliver than double-precision. It has been known that calculation using single-precision is more prone to numerical round-off errors, especially when a single tally data is accumulated "atomically" and repeatedly by thousands of GPU threads. To mitigate this problem, the least intrusive solution in theory is to replace the single-precision atomic-add tally function with a double-precision version. However, the complexity lies in the fact that some GPUs (Nvidia GPUs prior to the Pascal generation; all current AMD GPUs) do not readily offer such double-precision function at hardware level, and that software emulation is too slow to use if not optimized properly. This paper discusses several atomic-add tally methods with reduced numerical errors used throughout ARCHER development. The original software-based compare-and-swap method (CAS) was shown to be inefficient due to high intra-warp thread contention, whereas the improved software-based warp-aggregated method (WAG) and Kahan summation method (KAS) eliminated the thread contention and performed very well on Kepler and Maxwell GPUs, being more than 13 times faster than CAS in our tests. The hardware-based (HB) double-precision atomic-add feature available on Pascal and Volta GPUs exceeded software emulation and offered the best performance universally, so did WIB, a combination of WAG and HB methods. It was also shown that in a single-precision dose engine, KAS managed to maintain high reliability when the single-precision atomic-add tally underestimated the dose by 25%.

I. INTRODUCTION

I.A. Background

A graphics processing unit (GPU)-accelerated program needs to run with sufficient concurrent threads in order to make full use of the hardware. For Monte Carlo radiotherapy dose calculation, however, the GPU memory is usually not large enough for each individual thread to hold a local tally array. A common solution is to have a single array shared by all threads, accumulate the tally data via the *atomic-add operations*, and use the batch method for statistical uncertainty calculation. Here the atomic-add operations are intended to avoid race condition, a potential problem specific to multi-threaded programming. Race condition occurs when two threads try to update the same memory location, as illustrated in Figure 1. The atomic-add operation guarantees the correctness of the result, and in the given example, ensures the final value of $a[m]$ be $S + x + y$.

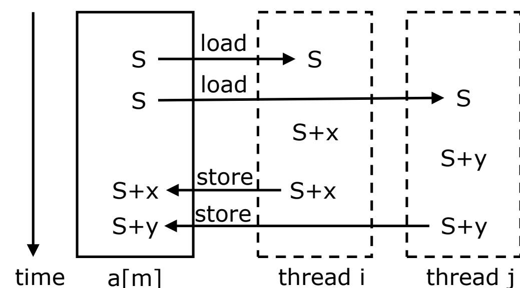


Fig. 1. A simple illustration of race condition. Thread i and j both update $a[m]$ whose original value is S. Thread i adds x to S, thread j adds y to S. The final value of $a[m]$ here should be $S + x + y$ but is instead $S + y$ because thread j stores its value to the global memory later than thread i. This problem can be avoided by atomic-add operations.

Previous research in GPU implementation of Monte Carlo method[1–4] usually adopted single-precision floating point format, with justification that its floating point operations per second (FLOPS) is 2~32 times higher than double precision's. A potential problem is that the numerical error resulting from accumulating single precision data may become remarkable when a few tally data are accumulated much more frequently than others. It has been observed that, in voxel-based dose calculation the single-precision result can deviate from double-precision by over 40%[5], and in CT scan organ dose calculation the lung dose can be underestimated by as much as 20%[6]. Reducing the numerical error in single-precision atomic-add operations is therefore worthy of extended study. In this paper, we compared several solutions we have used throughout ARCHER[3] development.

I.B. Related work

Recent work by Bossler[7] systematically compared the performance of five implementations of photon escape tally on GPUs. (1) global atomics — Having each thread atomic-

add to the global memory. (2) shared atomics — Having each thread atomic-add to the shared memory, and eventually having each block atomic-add to the global memory, (3) warp shuffle — Reducing data to a single thread in each warp, and having that single thread atomic-add to the global memory, (4) block reduction — Reducing data to a single thread in each warp, having that single thread atomic-add to the shared memory, and eventually having each block atomic-add to the global memory, (5) no atomics — Each thread holds and updates a local tally array. Three data types were tested on a Kepler GPU: 32-bit signed integer, 64-bit unsigned integer and 32-bit floating point (single-precision). The 64-bit floating point (double-precision) was also tested on a Pascal GPU. In addition, Bossler[7] compared CAS with HB for double-precision atomic-add. The findings of these test cases enabled a better understanding of the benefit of shared memory and warp shuffle functions for Monte Carlo tallies.

There are several key differences between Bossler's[7] work and ours. (1) In Bossler's study double-precision atomic-add tallies were deemed infeasible on GPUs prior to Pascal generation due to the very low performance of CAS, whereas the WAG and KAS tested in our study strive to solve the exact performance problem underlying CAS. (2) The WAG and KAS in our study can handle general situations on GPUs, including (a) thread divergence and (b) threads in a warp updating more than one memory locations. In Monte Carlo simulation, threads in a warp are likely to execute different branches, resulting in the existence of active and inactive threads. The inactive threads should not participate in the computation. More concretely, Ref. 8 pointed out that the warp shuffle functions would yield undefined value if data come from inactive threads. The major intricacy of WAG and KAS lies in the steps taken to skip those inactive threads when performing warp level reduction. (3) Bossler's methods that used the GPU shared memory, including "shared atomics" and "block reduction" were not considered because the tally arrays in dose calculation usually do not fit into the shared memory. (4) We focused only on the floating point data type (double-precision for CAS/WAG/HB/WIB, single-precision for KAS).

II. MATERIALS AND METHODS

II.A. Overview

Back in 2010 when the milestone Nvidia Fermi GPUs were first released, the only way of performing double-precision atomic-add operations was through a *software-based* approach with the compare-and-swap (CAS) algorithm implemented by Nvidia[8]. This algorithm may not be feasible in many cases due to heavy thread contention. In 2013 when the Kepler GPUs became available, a new feature "warp shuffle" was introduced, allowing threads in a warp to exchange data. It enabled a new algorithm "warp-aggregated method (WAG)" described in Ref. 9 and 10, which significantly reduced thread contention at the cost of additional intra-warp data manipulation. In ARCHER we implemented a GPU version of Kahan summation algorithm (KAS)[11], a classic algorithm originally used to reduce single-precision numerical errors for sequential code. WAG and KAS have remained the best atomic-add tally method with reduced numerical errors for Kepler and Maxwell generation of GPUs. It was not until recently that the latest Pascal and Volta GPUs provided *hardware-based* (HB) double-precision atomic-add operations, a major breakthrough that generally outperforms software-based solutions. The best approach for each generation of GPU is summarized in Table I.

TABLE I. Best tally method for atomic-add operations on different generations of GPUs, from the old Fermi to the latest Volta. "Best" here refers to small numerical error and decent speed. CAS is very slow, but is the only option for Fermi GPUs. WS: warp shuffle feature. HB: hardware-based double-precision atomic-add feature.

Generation	WS support	HB support	Best tally method
Fermi			CAS
Kepler	✓		WAG / KAS
Maxwell	✓		WAG / KAS
Pascal	✓	✓	HB / WIB
Volta	✓	✓	HB / WIB

II.B. Atomic-add tally methods with reduced numerical errors on GPUs

II.B.1. Compare-and-swap method (CAS)

Although Nvidia GPUs prior to Pascal generation do not directly support hardware-based atomic-add operations for double-precision data (64-bit), they do support atomic compare-and-swap operations for `unsigned long long int` data (64-bit as well). The compare-and-swap function[8] `old = atomicCAS(address, x, y)` reads the data `old` located at `address` and compares it with `x`. If they are equal, `y` is written to `address`, replacing `old`. Otherwise `old` is written back to `address` without any change. This function always returns the data `old`.

Ref. 8 exploited this function to realize double-precision atomic-add, shown in Listing 1 line 5. It adds `val`, the increment, to `old`, the tally, and checks if `old` at `address` is changed by other threads in the meantime. If not, the new value `(val + old)` is stored to `address`. Otherwise, the above steps are repeated.

GPUs implement the nominal single-instruction, multiple-thread (SIMT) model, where threads in a warp execute the same instruction at nearly the same time. This means that if some threads of the same warp update the same memory location, they will almost always result in

contention and have to make many repeated attempts before succeeding, thus reducing the overall performance. In fact, Ref. 5 considered CAS impractical because it tripled the computation time in their study.

II.B.2. Warp-aggregated method (WAG)

WAG[9,10] takes advantage of the GPU's warp shuffle intrinsic functions, and effectively eliminates the intra-warp thread contention in CAS. Initially, the warp shuffle functions `__shfl_sync()` in CUDA could only operate on 32-bit data. In order to handle double-precision atomic-add, we shuffled the 64-bit data `address` and `val` in two 32-bit chunks using the parallel thread execution (PTX) assembly language[12]. The recent CUDA API has started supporting warp shuffle on 64-bit data.

WAG is composed of three major steps. Step (1) (Listing 1 line 48), each thread in a warp finds their "peers", i.e., other threads in that warp updating the same memory location. Step (2) (Listing 1 line 101), by using intra-warp manipulations, dose values in peer threads are summed up and stored in a single leader thread. Step (3) (Listing 1 line 139), only these leader threads in a warp update memory locations using CAS.

In the worst case where all threads in a warp are active and update different memory locations, the first step of WAG will execute the code block in the `do-while` loop 32 times, making it slower than CAS. Such case rarely happens in our dose calculation tests.

Monte Carlo code typically consists of many conditional statements such as `if-else`. If threads in the same warp fall into different branches at runtime, the warp will execute all branches in sequence. During execution of a certain branch, threads that do not fall into it will be temporarily masked and set inactive, while other threads that do will remain active and execute instructions. It should be emphasized that having inactive threads participate in warp shuffle operations will lead to undefined values[8], and that WAG as well as KAS in the next section are very flexible such that they skip the masked, inactive threads and operate only on the active ones.

II.B.3. Kahan summation method (KAS)

KAS is a well-known algorithm initially used to reduce the numerical errors for sequential single-precision summation[11], shown in algorithm 1.

A GPU-specific KAS was implemented in this study. The initial version is shown in Listing 2. Two single-precision values `(tally, error)` (note: the "error" herein refers to the round-off error instead of the Monte Carlo statistical uncertainty) are put together to form a 64-bit data stored at `address`. Each time the tally is to be incremented by `val`, Kahan summation is performed such that (1) `val` is compensated using the previously calculated `error`, (2) `tally` is incremented, (3) `error` is recalculated. The new 64-bit

Algorithm 1: Original Kahan summation

input : P: an array with n elements
output: S: tally

```
1 S = 0 // tally
2 e = 0 // error
3 for i = 0; i < n; ++i do
4   y ← P[i] − e // compensate
5   t ← S + y // lower digits lost
6   e ← (t − S) − y // recover lower digits
7   S ← t // update tally
8 end
```

`(tally, error)` pair is then stored back to `address` using CAS.

This unoptimized KAS faces the same thread contention problem with CAS. We therefore used the three-step optimizations as in WAG to update the `(tally, error)` data. For completeness, the improved version is shown in Listing 3; for brevity, the comments have been stripped.

It is worth mentioning that Kahan summation requires the `error` term to be immediately applied to the next increment `val`. It is incorrect to accumulate `tally` and `error` separately without using the compare-and-swap function, and only sum `tally` and `error` at the end of the program.

II.B.4. Hardware-based method (HB)

The latest Nvidia Pascal and Volta GPUs provide hardware support for double-precision atomic-add operations. This effectively eliminates the need to emulate double-precision atomic operations via many single-precision operations. As an aside, growing trends in Deep Learning and Artificial Intelligence are placing stronger demands on lower precision operations and even half-precision hardware support[13]. For example, the latest Volta GPUs provide new "tensor cores" which leverage mixed FP16 (half-precision) and FP32 (single-precision) hardware support. This trend shows a potential shift in the GPU market back towards using lower precision hardware.

II.B.5. WAG in combination of HB (WIB)

If more than one threads in a warp perform atomic-add operations to the same memory location, the operations will be serialized[8]. It would be interesting to know whether this hardware overhead can be mitigated by WAG for further performance improvement. This gives birth to WIB, which follows the first two steps of WAG, as described in subsubsection II.B.2, to elect the leader threads and reduce data from peers, but performs a direct HB atomic-add in lieu of CAS in the third step. WIB is similar to Bossler's[7] "warp shuffle" method except that the direct warp shuffle operation is replaced by the intricate WAG method to handle thread divergence and update of multiple memory locations.

II.C. Test cases and hardware

Three test cases were considered in this study. Test 1 uses a mini application that only performs tally operations. It is assumed that each particle undergoes 10 collisions. The energy deposition in each collision fluctuates with a mean value of 0.1 MeV and is added to one of 8 tally data. Regardless of GPU models, the block per grid is set to 1024 and the thread per block is 64.

Test 2 uses the Monte Carlo photon-electron coupled transport code in ARCHER. A 20-MeV electron pencil beam is incident on a water phantom with a dimension of $40 \times 40 \times 40\ cm^3$ and a voxel count of $100 \times 100 \times 100$. The block and thread numbers are determined using the persistent thread method[14]. This test represents the scenario where atomic-add tallies are performed on many different voxels.

Test 3 is the same with test 2 except that the phantom voxel count is reduced to $10 \times 10 \times 10$ while the phantom dimension remains $40 \times 40 \times 40\ cm^3$. This test represents the scenario where atomic-add tallies are focused on fewer voxels. Test 4 futher reduces the voxel count to $2 \times 2 \times 2$.

A variety of Nvidia GPUs were used, including K40, M6000 and Titan X. We also ran the test on an experimental server hosted by Center for Computational Innovations (CCI) at Rensselaer Polytechnic Institute (RPI). This server has 4 state-of-the-art Nvidia Tesla V100 GPUs (SXM2 model featuring high-bandwith NVLink technology) and 2 IBM POWER9 processors. Each POWER9 processor has 20 cores with 4 hardware threads. Only 1 V100 GPU and 1 POWER9 were used in this study.

The performance was evaluated by measuring the wall time of the kernel functions.

III. RESULTS

III.A. Performance comparison of atomic-add methods

The computation time of the first three tests is shown in Table II. Double-precision format was chosen for the transport kernel. The CPU atomics by IBM POWER9 CPU simply refers to the OpenMP atomic operation.

From item (1) (2) and (3), the performance of WAG and KAS is superior to CAS in test 1. This benefit is not seen in test 2, where threads in a warp tend to update different memory locations. However, WAG and KAS are over 16 times faster than CAS in test 3, where the thread contention in CAS becomes more serious.

From item (5) and (6), HB provided by new-generation GPUs is superior to the software-based approach in test 1 and 3. As expected, HB exhibits no appreciable advantage in test 2 which is not bottlenecked by atomic-add operations.

From item (7) and (8), the latest Nvidia Volta GPU has shown outstanding performance in all tests. The latest IBM POWER9 CPU is second only to the Volta GPU in test 2. The cause of POWER9 CPU's serious underperformance in test 3 is currently being investigated by comparing the assembly of the OpenMP atomic operation with that on the x86 platform and GPUs.

To discern the difference between HB and WIB clearly, the above tests were run with increased number of particles. Table III shows that WIB is capable of improving the performance by a factor of 4 in the tally-only test (test 1), but this advantage is not carried to the photon-electron coupled transport tests (test 2, 3, 4). We verified that for both HB and WIB, the ARCHER code was run with the same execution configuration (i.e. the same number of threads and blocks) and occupancy. These interesting results indicate that the hardware support for double-precision atomic-add operations performs so well that the atomic tally no longer constitutes a bottleneck for particle transport simulation and that the hardware overhead of HB is rendered trivial. The tally-only test is able to greatly magnify this overhead, which can be effectively alleviated by WIB.

III.B. Performance and accuracy comparison of single-precision tally and KAS

To demonstrate the usefulness of KAS, test 3 was rerun on a K40 GPU using two tally methods, respectively: the native single-precision atomic-add operation `atomicAdd(address, val)` (where `address` is the address of the single-precision tally, and `val` is the single-precision increment) and KAS. In both cases, single-precision format was chosen for the transport kernel. After the simulation, the sum of all elements in the single-precision dose tally arrays was calculated. During this step, the single-precision data were promoted to double-precision in order to prevent further numerical errors. The sum was then compared with the double-precision simulation result, which provides the ground truth.

From Table IV, as the number of histories increases, the computation time of KAS increases proportionally, being about $40\% \sim 50\%$ slower than the single-precision atomic-add counterpart. However, KAS' effectiveness in numerical error reduction completely outweighs its drawback. While the single-precision atomic-add tally becomes increasingly inaccurate, KAS consistently yields reliable result.

Note that this does not mean the single-precision atomic-add tally is always unacceptable for Monte Carlo dose calculation. It all depends on the source, geometry, and tally conditions. KAS substantially reduces the risk with acceptable performance penalty.

IV. DISCUSSION AND CONCLUSION

There are several interesting directions for future research. (1) Ref. 9 was recently updated with a general comment that compiler-generated code can be faster than the manually-written warp-aggregation code. It is worth investigating on what conditions this happens and how

TABLE II. Performance comparison of different atomic-add methods with reduced numerical errors. Double-precision format was chosen for the transport kernel. T_1: time of test 1, which only performs tally accumulation. Test 1 only applies to GPUs, hence no data for the IBM POWER9 CPU. 10^7 particles were simulated. T_2: time of test 2, which simulates electron-photon transport in a water phantom with $100 \times 100 \times 100$ voxel count. 4×10^6 particles were simulated. T_3: time of test 3, which uses a coarser water phantom with $10 \times 10 \times 10$ voxel count.

Item	Tally method	Precision	Processor (generation)	T_1 [s]	T_2 [s]	T_3 [s]
1	CAS	double	K40 GPU (Kepler)	83	12	130
2	WAG	double	K40 GPU (Kepler)	6.3	12	8
3	KAS	single	K40 GPU (Kepler)	6.3	12	7.9
4	WAG	double	M6000 GPU (Maxwell)	13	6.7	7
5	WAG	double	Titan X GPU (Pascal)	6.9	4.5	4.2
6	HB	double	Titan X GPU (Pascal)	0.07	4.5	3.1
7	HB	double	V100 GPU (Volta)	0.035	1.9	1.3
8	CPU atomics	double	IBM POWER9 CPU (80 threads)	–	3.5	7.8

TABLE III. The test cases are the same with Table II but were run with increased numbers of particles. Test 1 was run with 3×10^9 particles, while test 2, 3, 4 with 4×10^7 particles. Test 4 used an even coarser water phantom ($2 \times 2 \times 2$ voxel count).

Tally method	Precision	Processor (generation)	T_1 [s]	T_2 [s]	T_3 [s]	T_4 [s]
HB	double	V100 GPU (Volta)	10	19	13	10
WIB	double	V100 GPU (Volta)	2.7	20	14	11

much faster the code becomes. (2) As the GPU hardware evolves over years, so does CUDA programmability. The recently introduced "cooperative groups" API may simplify the source code listed in this paper and enhance the readability. (3) For calculation requiring quadruple-precision format, Nvidia provided algorithms to perform basic arithmetic operations (multiply, divide, square root, etc)[15], but atomic-add algorithm for quadruple-precision has not been implemented yet. (4) Current AMD GPUs do not support HB double-precision atomic-add operations. It is necessary to evaluate the feasibility of porting WAG and KAS to OpenCL for AMD GPUs.

In conclusion, for simulation of photon-electron coupled transport, numerical errors may result from a few single-precision tally data being frequently updated by many threads via atomic-add operations. WAG and KAS are found efficient in reducing these numerical errors for Kepler and Maxwell GPUs. HB double-precision atomic-add on Pascal and Volta GPUs shows the highest performance. While WIB is equally fast in the transport simulation test, it is able to further alleviate the hardware overhead of HB in the tally-only test. ARCHER[3] is designed to adopt the fastest method on different GPUs and allows users to choose between single and double-precision dose engines.

ACKNOWLEDGMENTS

This research is funded by National Institute of Biomedical Imaging and Bioengineering (NIBIB) Grant R42-EB019265. We thank Nvidia for the generous GPU donation. We also thank Dr. Forrest Brown, LANL, for the helpful discussion in 2015 on the compensated summation algorithms.

REFERENCES

1. S. HISSOINY, B. OZELL, H. BOUCHARD, and P. DESPRÉS, "GPUMCD: A new GPU-oriented Monte Carlo dose calculation platform," *Medical physics*, **38**, *2*, 754–764 (2011).

2. X. JIA, X. GU, J. SEMPAU, D. CHOI, A. MAJUMDAR, and S. B. JIANG, "Development of a GPU-based Monte Carlo dose calculation code for coupled electron–photon transport," *Physics in Medicine & Biology*, **55**, *11*, 3077 (2010).

3. X. G. XU, T. LIU, L. SU, X. DU, M. RIBLETT, W. JI, D. GU, C. D. CAROTHERS, M. S. SHEPHARD, F. B. BROWN, ET AL., "ARCHER, a new Monte Carlo software tool for emerging heterogeneous computing environments," *Annals of Nuclear Energy*, **82**, 2–9 (2015).

4. R. M. BERGMANN and J. L. VUJIĆ, "Algorithmic choices in WARP–A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs," *Annals of Nuclear Energy*, **77**, 176–193 (2015).

5. V. MAGNOUX, B. OZELL, É. BONENFANT, and

TABLE IV. Performance and accuracy comparison of the single-precision atomic-add tally and KAS on a K40 GPU using test 3. Single-precision format was chosen for the transport kernel. The discrepancy refers to (tested method - double precision) / double precision.

Particles	Single-precision atomic-add tally		KAS	
	Discrepancy	Time [s]	Discrepancy	Time [s]
4×10^6	-0.19 %	6	-3.58×10^{-5} %	8
8×10^6	-0.88 %	11	1.67×10^{-5} %	16
1.6×10^7	-3.86 %	22	-1.98×10^{-5} %	31
3.2×10^7	-11.17 %	44	-1.40×10^{-5} %	63
6.4×10^7	-25.47 %	88	-9.73×10^{-6} %	121

P. DESPRÉS, "A study of potential numerical pitfalls in GPU-based Monte Carlo dose calculation," *Physics in Medicine & Biology*, **60**, *13*, 5007 (2015).

6. T. LIU, X. G. XU, and C. D. CAROTHERS, "Comparison of two accelerators for Monte Carlo radiation transport calculations, Nvidia Tesla M2090 GPU and Intel Xeon Phi 5110p coprocessor: A case study for X-ray CT imaging dose calculation," *Annals of Nuclear Energy*, **82**, 230–239 (2015).

7. K. L. BOSSLER, "Methods for computing Monte Carlo tallies on the GPU," in "PHYSOR 2018: Reactor Physics paving the way towards more efficient systems," American Nuclear Society (ANS) (2018).

8. NVIDIA, *CUDA C programming guide: design guide, v9.1*, Nvidia (2018).

9. A. ADINETS, "CUDA pro tip: optimized filtering with warp-aggregated atomics," (2014).

10. E. WESTPHAL, "Voting and shuffling to optimize atomic operations," (2015).

11. W. KAHAN, "Pracniques: further remarks on reducing truncation errors," *Communications of the ACM*, **8**, *1*, 40 (1965).

12. NVIDIA, *Parallel Thread Execution ISA Version 6.2*, Nvidia (2018).

13. A. KRIZHEVSKY, I. SUTSKEVER, and G. E. HINTON, "Imagenet classification with deep convolutional neural networks," in "Advances in neural information processing systems," (2012), pp. 1097–1105.

14. K. GUPTA, J. A. STUART, and J. D. OWENS, "A study of persistent threads style GPU programming for GPGPU workloads," in "Innovative Parallel Computing (InPar), 2012," IEEE (2012), pp. 1–14.

15. NVIDIA, "NVIDIA developer program – ComputeWorks exclusive downloads: CUDA double-double precision arithmetic," (2013).

APPENDIX

Listing 1. GPU implementation of fast double-precision atomic add based primarily on Ref. 8–10. The last function `UtilityGPU::atomicAddFast()` (line 161) is the interface to be used directly.

```
1   // CUDA-C++ code. Require nvcc 9.0 or newer.
2   namespace UtilityGPU
3   {
4   #if defined(__CUDACC__)
5   __device__ __forceinline__ double atomicAddFP64CAS(double* address, double val)
6   {
7       // nvidia's code
8       // one way of type punning
9       unsigned long long int* address_as_ull = (unsigned long long int*)address;
10      unsigned long long int old = *address_as_ull, assumed;
11      do
12      {
13          assumed = old;
14
15          // atomicCAS(address, compare, val):
16          // reads the 64-bit word old located at the address in global
17          // memory, evaluates (old == compare ? val : old), and stores the
18          // result back to memory at the same address. These three operations are
19          // performed in one atomic transaction. The function returns old.
20          old = atomicCAS(address_as_ull, assumed, __double_as_longlong(val + __longlong_as_double(assumed)));
21      }
22      // keep trying until the value at the address happens to be not changed by other threads
23      // use integer comparison to avoid hang in case of NaN (since NaN != NaN)
24      while (assumed != old);
25      return __longlong_as_double(old);
26  }
27
28  __device__ __forceinline__ double atomicAddFP64WAG(double* address, double val)
29  {
30      // for the dated Fermi GPUs
31  #if __CUDA_ARCH__ < 300
32      return atomicAddFP64CAS(address, val);
33
34  #else
35      // another way of type punning
36      union HelperUnionFP64
37      {
38          double* address;
39          double  helper;
40      };
41
42      union HelperUnion2FP64
43      {
44          double data;
45          unsigned long long int helper;
46      };
47
48      // step 1: find peers, i.e. threads in the same warp that are going to update
49      // the same address, which will cause CAS to be excruciatingly slow
50      unsigned int lane;
51       // %laneid is a predefined, read-only special register that returns
52       // the thread's lane within the warp. The lane identifier ranges from zero to WARP_SZ-1
53      asm volatile("mov.u32 %0, %laneid;" : "=r"(lane));
54
55      unsigned int peers = 0;
56      bool is_peer;
57
58      // --> __activemask(): Returns a 32-bit integer mask of all currently active threads in the calling warp.
59      //     The Nth bit is set if the Nth lane in the warp is active when __activemask()
60      //     is called. Inactive threads are represented by 0 bits in the returned mask.
61      //     Threads which have exited the program are always marked as inactive.
62      // --> __ballot(1) has been replaced by __activemask() since cuda 9.
63      unsigned unclaimed = __activemask();
64
65      do
66      {
67          // __ffs(): find the position of the least significant bit set to 1 in a 32 bit integer
68          // return 0 if no bit is set
69          // the position is 1-based rather than 0-based
70          int src = __ffs(unclaimed) - 1;
71
72          HelperUnionFP64 addressUnion = {address};
73          HelperUnionFP64 tempUnion;
74
75          // --> __shfl_sync(): returns the value of var held by the thread whose ID is given by srcLane.
76          //     If width is less than warpSize then each subsection of the warp behaves as
77          //     a separate entity with a starting logical lane ID of 0. If srcLane is
78          //     outside the range [0:width-1], the value returned corresponds to the value
79          //     of var held by the srcLane modulo width (i.e. within the same subsection).
80          //     since cuda 9, __shfl_sync() can operate on 64-bit data directly.
81          tempUnion.helper = __shfl_sync(__activemask(), addressUnion.helper, src);
82
83          // check if the addresses from the source lane is the same with self's
84          is_peer = (addressUnion.address == tempUnion.address);
85
86          // determine which lanes have a match with source
87          // essentially broadcast the results among active threads
88          // --> __ballot_sync(): evaluate predicate for all non-exited threads in mask and return
89          //     an integer whose Nth bit is set if and only if predicate evaluates to non-zero
90          //     for the Nth thread of the warp and the Nth thread is active.
91          peers = __ballot_sync(__activemask(), is_peer);
92
93          // remove lanes with the same address with source
94          // using xor (bit set if x and y is different)
95          unclaimed ^= peers;
96      }
97      // quit when the source lane has the same address to be updated with self's
98      // in the special case where all lanes have their unique address, peer will just be self
99      while(!is_peer);
100
101     // step 2: reduce peers. results are stored in the lowest peers
102     // find the peer with lowest lane index
103     int first = __ffs(peers) - 1;
104
105     // calculate relative index among peers
106     // __popc(): count the number of bits that are set to 1 in a 32 bit integer
107     int rel_pos = __popc(peers << (32 - lane));
108
109     // ignore peers with lower (or same) lane index
```

```
110	    peers &= (0xfffffffe << lane);
111
112	    while(__any_sync(__activemask(), peers))
113	    {
114	        // find next-highest remaining peer
115	        int next = __ffs(peers);
116
117	        // Threads may only read data from another
	             thread which is actively participating in
	             the __shfl_sync() command.
118	        // If the target thread is inactive, the
	             retrieved value is undefined.
119	        // This is why the code below is not placed in
	             the if(next) block
120	        double temp = __shfl_sync(__activemask(), val,
	             next - 1);
121
122	        // only add if next-highest remaining peer
	             exists
123	        if(next)
124	        {
125	            val += temp;
126	        }
127
128	        // results in lanes with an odd relative index
	             will be discarded
129	        // find if self has an odd index
130	        unsigned int done = rel_pos & 1;
131
132	        // remove all peers with an odd relative index
133	        peers &= ~__ballot_sync(__activemask(), done);
134
135	        // use relative index as iteration counter
136	        rel_pos >>= 1;
137	    }
138
139	    // step 3: Nvidia's compare-and-swap algorithm
140	    // only peers with the lowest index (defined as the
	         "leader threads") perform atomic operation
141	    unsigned long long int* address_as_ull = (unsigned
	         long long int*)address;
142	    HelperUnion2FP64 old, assumed, temp;
143	    if(lane == first)
144	    {
145	        // union version
146	        old.helper = *address_as_ull;
147	        do
148	        {
149	            assumed = old;
150	            temp = assumed;
151	            temp.data = val + assumed.data;
152	            old.helper = atomicCAS(address_as_ull,
	                 assumed.helper, temp.helper);
153	        }
154	        while (assumed.helper != old.helper);
155	    }
156
157	    return old.data;
158	#endif
159	}
160
161	__device__ __forceinline__ double atomicAddFast(double*
	     address, double val)
162	{
163	    // for the dated Fermi GPUs
164	#if __CUDA_ARCH__ < 300
165	    return atomicAddFP64CAS(address, val);
166
167	    // for the latest Pascal, Volta GPUs with hardware
	         support for FP64 atomic add
168	#elif __CUDA_ARCH__ >= 600
169	    return atomicAdd(address, val);
170
171	    // for the Kepler and Maxwell GPUs
172	#else
173	    return atomicAddFP64WAG(address, val);
174	#endif
175	}
176
177	#endif // __CUDACC__
178	} // end namespace UtilityGPU
```

Listing 2. Unoptimized GPU implementation of Kahan summation based on CAS.

```
1	// CUDA-C++ code. Require nvcc 9.0 or newer.
2	namespace UtilityGPU
3	{
4	#if defined(__CUDACC__)
5	__device__ __forceinline__ float2
	     atomicAddKahanUnoptimized(float2* address, float
	     val)
6	{
7	    union HelperUnionFP32
8	    {
9	        float2 data;
10	        unsigned long long int helper;
11	    };
12
13	    unsigned long long int* address_as_ull = (unsigned
	         long long int*)address;
14	    HelperUnionFP32 old, assumed, temp;
15
16	    old.helper = *address_as_ull;
17
18	    do
19	    {
20	        assumed = old;
21	        temp = assumed;
22
23	        float y = val - temp.data.y;
24	        float t = temp.data.x + y; // accumulate high-
	             order part
25	        temp.data.y = (t - temp.data.x) - y; // recover
	             lower-order part
26	        temp.data.x = t;
27
28	        old.helper = atomicCAS(address_as_ull, assumed.
	             helper, temp.helper);
29	    }
30	    while(assumed.helper != old.helper);
31
32	    return old.data;
33	}
34	#endif // __CUDACC__
35	} // end namespace UtilityGPU
```

Listing 3. Optimized GPU implementation of Kahan summation based on CAS. It is a combination of Listing 1 and Listing 2 such that only the “leader threads” in each warp update the tally and error term.

```
1	// CUDA-C++ code. Require nvcc 9.0 or newer.
2	namespace UtilityGPU
3	{
4	#if defined(__CUDACC__)
5	__device__ __forceinline__ float2 atomicAddKahan(float2*
	      address, float val)
6	{
7	    union HelperUnionFP32
8	    {
9	        float2* address;
10	        double helper;
11	    };
12
13	    union HelperUnion3FP32
14	    {
15	        float2 data;
16	        unsigned long long int helper;
17	    };
18
19	    unsigned int lane;
20	    asm volatile("mov.u32 %0, %laneid;" : "=r"(lane));
21	    unsigned int peers = 0;
22	    bool is_peer;
23	    unsigned unclaimed = __activemask();
24
```

```
25      do
26      {
27          int src = __ffs(unclaimed) - 1;
28          HelperUnionFP32 addressUnion = {address};
29          HelperUnionFP32 tempUnion;
30          tempUnion.helper = __shfl_sync(__activemask(),
                 addressUnion.helper, src);
31          is_peer = (addressUnion.address == tempUnion.
                 address);
32          peers = __ballot_sync(__activemask(), is_peer);
33          unclaimed ^= peers;
34      }
35      while(!is_peer);
36      int first = __ffs(peers)-1;
37      int rel_pos = __popc(peers << (32 - lane));
38      peers &= (0xfffffffe << lane);
39      while(__any_sync(__activemask(), peers))
40      {
41          int next = __ffs(peers);
42          float temp = __shfl_sync(__activemask(), val,
                 next - 1);
43          if(next)
44          {
45              val += temp;
46          }
47          unsigned int done = rel_pos & 1;
48          peers &= ~__ballot_sync(__activemask(), done);
49          rel_pos >>= 1;
50      }
51
52      unsigned long long int* address_as_ull = (unsigned
             long long int*)address;
53      HelperUnion3FP32 old, assumed, temp;
54      if(lane == first)
55      {
56          old.helper = *address_as_ull;
57          do
58          {
59              assumed = old;
60              temp = assumed;
61              float y = val - temp.data.y;
62              float t = temp.data.x + y; // accumulate
                     high-order part
63              temp.data.y = (t - temp.data.x) - y; //
                     recover lower-order part
64              temp.data.x = t;
65              old.helper = atomicCAS(address_as_ull,
                     assumed.helper, temp.helper);
66          }
67          while(assumed.helper != old.helper);
68      }
69
70      return old.data;
71  }
72  #endif // __CUDACC__
73  } // end namespace UtilityGPU
```


View publication stats