Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 721–728

DOI: 10.15439/2015F86
ACSIS, Vol. 5

Reproducible floating-point atomic addition in data-parallel environment

David Defour
Laboratoire DALI-LIRMM
52 avenue Paul Alduy
66860 Perpignan Cerdex - France
Email: david.defour@univ-perp.fr

Sylvain Collange
INRIA – Centre de recherche Rennes – Bretagne Atlantique
Campus de Beaulieu, F-35042 Rennes Cedex, France
Email: sylvain.collange@inria.fr

***Abstract*—Floating-point additions in concurrent execution environment are known to be hazardous, as the result depends on the order in which operations are performed. This problem is encountered in data parallel execution environments such as GPUs, where reproducibility involving floating-point atomic addition is challenging. This problem is due to the rounding error or cancellation that appears for each operation, combined with the lack of control over execution order. In this article we propose two solutions to address this problem: work reassignment and fixed-point accumulation. Work reassignment consists in enforcing an execution order that leads to weak reproducibility. Fixed-point accumulation consists in avoiding rounding errors altogether thanks to a long accumulator and enables strong reproducibility.**

I. Introduction

Efficient exploitation of modern multicore architectures relies on a hierarchical structuration of computation as well as execution concurrency. This affects determinism and numerical reproducibility making software development tedious. For example, GPUs manage several thousands of concurrent threads thanks to hardware resources such as warp and block schedulers. The design complexity of those processors is such that thread scheduling is mostly unknown and can be considered unpredictable. A common workaround is to enforce interaction between tasks using memory consistency with synchronization mechanisms such as locks, atomics or barriers.

Atomic operations are designed to perform a read-modify and write operation in one instruction. For example, atomicAdd() reads a word at a given address, 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. Although atomic operations can address the problem of memory consistency, they do not solve the problem of numerical reproducibility when dealing with floating-point numbers. This is a major issue as non-determinism of floating-point calculations in parallel programs causes validation and debugging issues, and may even lead to deadlocks [1].

The numerical non-reproducibility of floating-point atomic additions is due to the combination of two phenomena: rounding-error and the order in which operations are executed. This problem can be depicted with the simplified following CUDA kernel which computes the sum of N floating-point numbers stored in table *i_val* according to their address *i_adr* in a table *res* located in global memory.

```
__global__ void GlobalSum(float *i_val,
            int *i_adr, float *res, int N){
  int gid =  blockDim.x*blockIdx.x+threadIdx.x;

  for(uint i=0; i<N; i+=GridDim.x*blockDim.x)
     atomicAdd(&res[i_adr[i+gid]],
            i_val[i+gid]);
}
```

Listing 1. Floating-point atomic accumulation

The problem with this simple code, is that we do not have any information on the order in which threads will acquire access to the datum *Res*. For example, on a set of $N = 2^{16}$ values with a condition number[1] of 10^8 and a single output address, we measured that out of over 1000 runs with 1 block of 1024 threads on a GTX680 we obtain 1000 different results.

One can argue that in the case where there is a single accumulator, weak reproducibility could be achieved by replacing atomicadd with standard addition combined with a reduction algorithm. However this solution does not hold when some threads are not producing any value (not executing the atomic addition), or when there are multiple accumulator, as in the bin counting or histogram problem as encountered in Nbody [3],

[1]The condition number characterizes the numerical stability of a problem [2].

978-83-60810-66-8/$25.00©2015, IEEE

Real-time simulation [4], accurate reduction scheme [5], or SQL query [6]. In these applications, floating-point atomic addition is used to accumulate values while preserving memory consistency.

In this article, we will use two concepts regarding numerical reproducibility in the context of data parallelism. *Weak numerical reproducibility* consists in being able to reproduce the same result between two executions when input parameters are identical. *Strong numerical reproducibility* consists in being able to reproduce the same results between two executions independently of the execution parameters or the architecture. Strong numerical reproducibility can be further subdivided in two classes of algorithms. Those which are producing correctly rounded results such as the one based on long accumulators [7], and others which provide reproducible results without any guaranty on accuracy [8].

Following these two definitions, we propose two solutions to address reproducibility involving floating-point atomics addition in the context of GPU programming by attacking the problem at its two roots. The first solution, work reassignment, consists in enforcing an evaluation order by overloading the block index assignment. With this solution, results are identical among runs for identical execution parameters. It is considered weakly reproducible as the number of threads per block affects the evaluation scheme and therefore the result. The second solution, fixed-point accumulation, avoids the rounding errors that occur during floating-point addition by using a long accumulator. As the addition is now exact, thus associative, the result is independent on the evaluation order or the hardware. In addition, the result is as accurate as possible as the accumulation is performed exactly. This solution is considered strongly reproducible.

The rest of this article is organized as follow. Section II introduces the necessary background about floating-point arithmetic and model of execution of GPUs. Section III presents the first solution we propose, based on block reordering. Section IV presents the second solution based on long accumulators. Section V analyses the theoretical cost of both methods and Section VI presents performance measurements on Nvidia GPUs.

II. Background

A. Floating-point arithmetic

Floating-point (FP) numbers approximate real numbers with a significand, an exponent, and a sign. The IEEE-754 standard, which was revised in 2008, specifies floating-point formats and operations. In this paper, we consider the `binary32` or single precision format. The floating-point number system can represent a wide range of numbers with nearly-constant precision.

Floating-point addition is not associative, due to the rounding error that occur when adding numbers with different exponents. It leads to the absorption of the lower bits of the sum. For example the exact mathematical result of $(1 + 2^{100} - 2^{100})$ is equal to 1 whereas the computed result is either 0 or 1 depending on the order of operations. Thus, the final accuracy of a floating-point summation depends on the order of evaluation. More details can be found in the main references related to floating-point arithmetic [9], [10]. This problem of numerical reproducibility linked to the order of floating-point operations, is amplified when executed in massively parallel environments like GPUs.

B. GPU execution model

In this article we consider CUDA capable Nvidia GPUs used for the execution of tasks exhibiting data parallelism. These tasks are divided in threads operating in SIMT mode and executed by specific hardware. We must distinguish the software and hardware organization of threads. From the developer point of view, threads are divided into three hierarchical levels: a *grid* of *blocks* of *threads*. The same code, or kernel, is executed by multiple threads running in parallel on different data. *Threads* are grouped in set of *block_size* elements in order to make so-called *blocks*. Blocks are packed in set of *grid_size* elements in order to make a so-called *grid*. Threads in a block and blocks of a grid are uniquely identified by their coordinates in the blocks and the grid. In this model, threads in a block and the blocks of a grid are virtually launched in parallel, which implies that no assumption shall be made regarding the execution order.

In CUDA terminology, GPU hardware consists of CUDA cores organized hierarchically. These CUDA cores are grouped in streaming multiprocessors (SMs). The number of SMs varies depending on the architecture of the GPU and the CUDA compute capability. An additional and transparent level of grouping is introduced at the hardware level: the warp. These warps, corresponding to 32 threads, are created, managed, launched and executed by SIMT units. Multiprocessors share the instruction fetch, decoding and control logic across all the threads in a warp, so they run in lock-step [11].

Blocks are dispatched among the available multiprocessor by the block schedulers. This step consists in launching a new block with a unique identifier according to available resources. The number of concurrent blocks depends on the number and version of SMs and the resources such as registers and shared memory required by the executed kernel. This step impacts determinism as no assumption can be made on how indexes are generated and is subject to variations from one run to another [12].

Traditional thread synchronization primitives used in concurrent programming such as mutexes, barriers, and semaphores are limited on GPU to intra-block synchronizations. Atomic operations provide basic support for inter-block communication. Atomic instructions were first introduced on Nvidia hardware with compute capability 1.1, and atomic addition operating on 32-bit floating-point values in global and shared memory were introduced with compute capability 2.0. Atomic floating-point operations are necessary, first to provide the substrate for high performance floating-point operations, and second, to preserve the memory consistency necessary to deal with thousands of threads in flight.

The variety and performance of synchronization primitives available on GPUs have continuously increased over the years. This has lead to numerous works that have been focusing on efficient inter-block synchronization. For example, in [13] Volkov et al. propose a global software synchronization method that does not use atomic operations to accelerate dense linear-algebra constructs. In [14], [15], Xiao and Feng propose a mechanism for inter-block communication via global memory. In [16], Stuart and Owens are evaluating various implementations of barriers, mutexes and semaphores applied to Nvidia's GPU. Collange et al. use long accumulators to enable reproducible floating-point reductions [7]. However, none of those works have addressed the problem of reproducible atomic floating-point addition.

III. Enforcing execution order

As discussed in the introduction, atomic operations enforce memory consistency, but not execution order. This means that atomic additions ensure that all data will be considered, but do not guarantee the order of operations. This is problematic for floating-point addition. The first solution we propose is to enforce an execution order for atomic addition, similarly to what can be achieved using a reduction algorithm. Threads and blocks are spawned and scheduled by hardware, following unspecified and implementation-specific policies. Thus, no assumption can be made on their real execution order. Therefore, we cannot rely on CUDA thread and block identifiers used at software level. Such solution would leads to an inefficient execution scheme and most likely to a deadlock situation [15].

The proposed solution uses two key concepts. The first one is based on a new software assignment of the block index, in order to guarantee a fine control over blocks effectively executed by the hardware. The second concept is based on atomic locks in order to ensure uniqueness of accesses at block level. We extend the solution proposed in [17] in order to control the order of execution. The lock used is similar to a fetch-and-add mutex algorithm in which the block index corresponds to the token acquired by a block.

A. Block re-ordering

Block synchronization is challenging, as the CUDA programming model does not support it. The only safe solution consists on splitting kernels into subkernels, as a kernel launches involves an implicit synchronization barrier. Alternatively, resident kernel techniques take advantage of the fact that once launched, a block continue its execution until completion, freeing resources only at the end. For example, the block barrier proposed by Feng in [14] is working only when the number of launched blocks is less than the number of blocks that could be executed concurrently on the hardware. In case this assumption is not met, deadlocks may occur. For example, consider a GPU and a kernel such that only 8 blocks can run concurrently. If the kernel is launched on more than 9 blocks, then at least 1 block will not be scheduled. In that case running blocks will be waiting on the barrier for every block to complete, which will never happen as resources taken by those running block will never be released for others to complete.

In our case, we want to ensure that blocks are executed in a reproducible manner. This requires defining an execution order, which corresponds to a statically known order. This order can be, for example, the one corresponding to their block index. Besides requiring a known order, we need to prevent deadlock situation, which involves restrictions on the real scheduling of blocks.

The proposed solution consists in overriding the index generated by the block scheduler. This can be done by using a global variable *oBlkId* set to 0 at kernel launched which will be queried by one thread of each block at the beginning of the execution. The resulting index is stored in a shared variable *sBlkIdx* accessible by every thread of the block they belong to, and can be used as the new block index. An overview of the code is given in listing 2.

```
// Shared BlockIdx within a block
__shared__ uint sBlkIdx;

// Reindexing Block
struct BlkIdx{
  // New ordered index for BlockIdx
  uint *oBlkId;

  BlkIdx(void){
    cudaMalloc((void**) &oBlkId, sizeof(uint));
    cudaMemset(oBlkId, 0, sizeof(uint));
  }

  ~BlkIdx(void){
    cudaFree( ordBlockId );
  }
```

```
  __device__ uint get_ordered_blockId(){
    if (threadIdx.x == 0)
      sBlkIdx = atomicInc(oBlkId, gridDim.x-1);

    __syncthreads();
    return sBlockIdx;
  }
};
```

Listing 2. Block reindexing

Thanks to this new block index, we can ensure a known execution order for block, which will be deadlock-free. With this technique, we chain together running blocks following a well-defined order. Therefore the overhead is solely concentrated in the access to the global index *sBlockIdx*. We now have to set a floating-point atomic addition which is reproducible within each block, which we will describe next.

B. Atomic completion

Threads of blocks are scheduled as set of 32 consecutive threads, or warps. Again, as we do not have any control over the execution order of threads, we need to make sure that threads of a given block are always scheduled in a similar fashion when they are atomically accessing the destination address.

A solution is to encapsulate the atomic addition within a fetch-and-add mutex (FA) algorithm similarly to the one described in listing 3. With this solution, each thread of a given block is waiting for its turn on a variable shared at block level. These results ultimately to a serialization of each atomic addition at both thread-level and block-level as there are launched in an order corresponding to their global identifier. This solution is simple and provides strong numerical reproducibility. However accesses cannot happen concurrently and will perform poorly. We should mention that a syncthreads barrier is required at the end of the outer loop to ensure that no warp will go faster than any other warp with lower thread identifier.

```
struct Lock{
  uint *g_lock;

  Lock(void){
    cudaMalloc( (void**) &g_lock, sizeof(uint));
    cudaMemset( g_lock, 0, sizeof(uint));
  }

  ~Lock(void){
    cudaFree( g_lock );
  }

  __device__ void acquire_lock(int goalval){
    if (threadIdx.x == 0)
      while(atomicAdd(g_lock, 0) != goalval);
    __syncthreads();
  }

  __device__ void release_lock(){
    __syncthreads();
    if (threadIdx.x == 0){
      (*g_lock) = ((*g_lock)+1)%(gridDim.x);
    }
  }
};
```

Listing 3. Fetch-and-Add mutex

To improve performance of the previous solution, we can relax the constraint on the execution order in order to allow the atomic additions within a block to be replaced by simple additions executed in parallel. This results in a reduction scheme that only depends on the block size and is therefore considered as weakly reproducible. This solution requires 6 steps as described in listings 4. First, floating-point atomic data and addresses are stored in parallel in two tables located in shared memory. Then, both tables are sorted using the corresponding destination address as sorting key. In our implementation we used a bitonic sort as it preserve the order in case of equality. Once data are sorted according to their address, we perform a segmented sum for data with identical address. Then additions to the destination adresses are done in parallel for each different addresses within a block. We should mention that the number of additions is bounded by the number of threads per block. Those additions do not require atomic operations as atomic access is guaranteed with a FA lock on the new block identifier.

```
__device__ void ordered_concurrent_ AtomicAdd(
        float *dest, float val, Lock &lock){
  int tidx = threadIdx.x;
  // 1: Store in shared mem
  s_adr[tidx] = dest;
  s_val[tidx] = val;
  __syncthreads();

  // 2: Sort Element
  bitonicSort( s_adr, s_val);

  // 3: Segmented Sum
  segmented_sum_per_block(s_adr, s_val);

  // 4: Aquire the lock in order
  lock.acquire_lock(sBlockIdx);

  // 5: Final write in
  if (threadIdx.x < blockDim.x-1){
    if (s_adr[tidx] != s_adr[tidx+1]){
      *(s_adr[tidx]) += s_val[tidx];
    }
  }else{
      // For the last thread !
      *(s_adr[tidx]) += s_val[tidx];
  }
  // 6: Release the lock
  lock.release_lock();
}
```

Listing 4. Second solution: Serialization of atomic access at block level

IV. Making addition associative

To achieve numerical reproducibility of atomic addition, we have seen in section III a solution based on enforcing an execution order. This section describes an alternative solution that consists in avoiding rounding errors to compute the correctly-rounded result.

To enforce strong reproducibility, we replace floating-point additions by fixed-point additions, which are associative. To avoid losing any precision compared to the floating-point format, we use a fixed-point accumulator that covers the whole range of values representable in the considered floating-point format. For instance, an accumulator with 127 integral bits and 127 fractional bits covers the range of single-precision floating-point. Such counter is known as a long accumulator or superaccumulator [18].

While the use of such a wide accumulator may seem extremely costly at first sight, three considerations make its performance actually attractive in the context of atomic operations:

- First, the average complexity of adding one floating-point value to a superaccumulator does not depend on the width of the superaccumulator [7]. Assuming the accumulator is represented as a vector of machine words (32-bit in the single-precision case), an accumulation generally only affects two words in the vector. Although carries may need to be propagated, the probability that a carry propagates across multiple words quickly gets negligible assuming low-order digits follow a uniform distribution [19].
- Second, updates to individual words of the superaccumulator can happen in any order without bearing any impact on the result, as long as all carries are detected and eventually propagated (including carries that occur during carry propagation). This enable regular integer atomic operations to update each part of the accumulator independently, without the need to implement any locking or coarser-grain transactions. Carries are detected *a posteriori* from the older value returned by the atomic operation and the value that was accumulated. Carries that occur are iteratively propagated to the higher-order words of the superaccumulator. Atomic operations from other threads may modify the superaccumulator during carry propagation; however, it bears no impact on the final result as all carry are eventually propagated in an arbitrary order.
- Third, this solution only requires hardware support for atomic integer addition, and does not need support for atomic floating-point addition. Therefore, the long accumulator allows to design reproducible atomic addition for any floating-point representation format (e.i. half, single or double precision of the IEEE-754 standard). By contrast, current GPUs only support floating-point atomic operations on single-precision data. The only difference lies in the size of the long accumulator.

We propose a thread-safe generalization of the long accumulation algorithm [20]. This accumulation algorithm is illustrated on Figure 1. The exponent and mantissa are extracted from the input number. High-order bits e_h of the exponent are used to select the words that are affected in the superaccumulator. Lower-order bits e_l select a cutoff point to split the mantissa into (at most) two parts, m_h and m_l, whose weights are aligned with the words of the superaccumulators. This has the effect of shifting the mantissa to align it with the superaccumulator words. Words m_h and m_l are independently accumulated atomically to their respective accumulator words. Carries are detected and propagated using independent atomic operations.

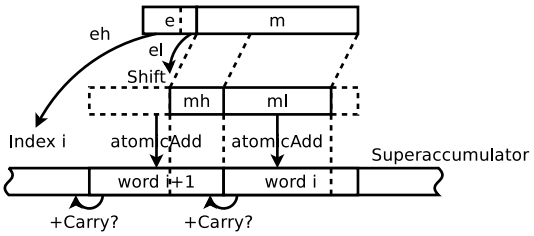


Fig. 1. Accumulation of a floating-point number to a superaccumulator.

V. Cost

In this section we details the memory and operation cost of the two proposed solutions.

A. Work reassignment

The solution based on enforcing an execution order requires one integer for the global index, one for the lock and one integer in shared memory for each block to store the shared index. In addition to these elements it requires a table of N integers and N floating-point numbers in shared memory per block to store temporary values, with N corresponding to the number of threads per block.

In terms of computational overhead, this solution first requires to get a new block index at the beginning of the kernel. This involves one atomic instruction. Then, the kernel is left unmodified except when floating-point atomic additions need to be performed. In that case,

each atomic addition is replaced by a write of both the address and the value to shared memory, plus a bitonic sort and a segmented sum per block over N elements achieved in $O(log(N))$.

Then, threads of each block are waiting for their turn by spinning over the lock index. Once this condition is met, threads with the last address to be written send their write to global memory along the normal store path. As mentioned before these writes do not require atomic addition and can be done in parallel with an increased probability that global memory accesses are coalesced as data are sorted according to their destination addresses. For these reasons, this solution will perform well when many threads of the same block attempt to atomically add a value to the same location.

B. Fixed-point accumulation

The fixed-point accumulator solution adds overhead in storage, computation, memory transfers and adds an extra rounding step. The most obvious cost of a long accumulator is its memory overhead. An accumulator able to represent exactly every single-precision floating-point value requires 280 bits of storage, as opposed to 32 bits using a floating-point accumulator. Likewise, a double-precision long accumulator needs about 2100 bits. Fortunately, long accumulators often contain sparse data: when accumulating numbers of the same order of magnitude, only a small part of the accumulator will be accessed, while the remaining part stays null. The hot parts can fit inside caches and benefit from fast cached atomic operations, while the cold part will remain in the large, off-chip memory.

Adding a floating-point value to a long accumulator also requires extra computations. The algorithm described in section IV extracts the exponent and mantissa of the floating-point number, then splits the mantissa by scaling and rounding it twice. However, binning algorithms are memory intensive and their performance is usually not limited by computations. This means the computational of long accumulation may be hidden by the memory access delays.

In terms of memory accesses, fixed-point accumulation requires two atomic operations per number in the common case, while a floating-point accumulation will only need one atomic add on platforms that support it in hardware. On the other hand, conflicts are reduced as atomic accesses span a larger memory area.

Finally, the fixed-point result has to be rounded to floating-point at the end of the accumulation. Rounding involves finding the first significant bit, then scanning the rest of the accumulator to compute the round, guard and sticky bits, that are necessary for IEEE-754 compliant rounding. The rounding phase can be performed in parallel between accumulators as they have no dependencies.

VI. Results

We tested both methods on a GTX480, a GTX560 and a GTX680 architectures with characteristics described in Table I. We made comparisons for a number of output addresses ranging from 1 (which corresponds to an accumulation) to 16384 different addresses randomly generated. Each thread was responsible for the accumulation of one floating-point value at a given address in global memory. Each kernel was launched with 100 blocks of 512 threads.

We compared the proposed methods against the *unordered solution* which consists in atomically adding the generated data at a given address as described in listing 1. For comparison purposes, we also included the fully sequential solution (block and warp sequential). Both block and warp sequential and block sequential methods relies on enforcing an execution order whereas superaccumulators avoid rounding errors altogether.

Figure 2(a), 2(b) and 2(c) describes the execution time on the GTX480, GTX560 and GTX680 respectively. One can observe that the execution time of the unordered solution quickly decreases as the number of output addresses increases. This is due to reduced contention : as the atomic accesses are spread across a wider range of accesses, the probability of conflicts decreases and the global performance improves. On the other hand, both sequential methods exhibit an execution time that is almost insensitive to the number of output addresses. The solution based on the long accumulator has an execution cost that decreases on the GTX480 and GTX560 and quickly increases after 64 to 128 different addresses. The decrease is due to concurrent accesses that can happen simultaneously as the number of different accumulators increases. However after a certain bound, the overhead of the accumulator do not compensate the gain obtained by the increase in concurrent accesses.

On the GTX 680, the unordered algorithm is always best according to figure 2(c), which is consistent with the fact that atomic operations were improved on this architecture compared to previous generations. Atomic performance is less sensitive to conflicts. The long accumulator also benefits from the improving atomic performance.

On GTX480 and GTX560, one can notice that the block sequential solution can even be faster than the unordered solution for small numbers of output addresses. This is because the small number of atomic access compensates the overhead of the segmented summation and sorting. By contrast, for the unordered solution

almost every atomic access generated by each thread is conflicting on the output address, leaving no room for concurrent execution.

Out of these results, we can observe that enforcing strong numerical reproducibility is between 1.3 and 21 times more expensive than the unordered summation, whereas weak numerical reproducibility corresponds to 0.4 to 10 times the execution time of the unordered summation.

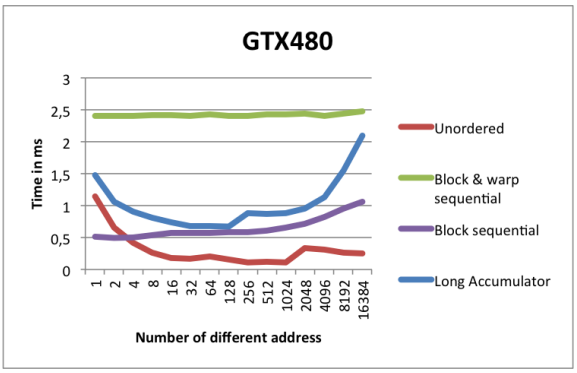
We should mention that the solution based on enforcing execution order uses block-wide synchronization barriers. This implies that atomics using the weakly reproducible solution need to be invoked from uniform control-flow regions. Every thread has to execute the instruction FPatomicAdd, otherwise the behavior is unspecified.

VII. Conclusion

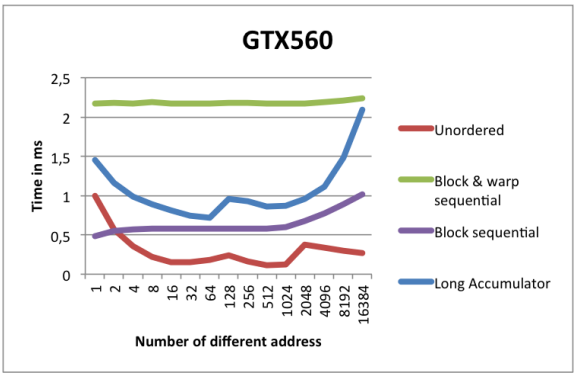
Atomic operations are very helpful in data parallel programming to enforce memory consistency. However when dealing with floating point values, this concurrency can lead to high variability, even when input data and execution parameters are identical. This behavior, due to rounding errors combined with the lack of control over execution order, is problematic as it causes validation and debugging issues, as well as producing hard-to-diagnose bugs in distributed environments.

In this article we propose two solutions to tackle this problem, each one addressing one source of the problem. We have described a first solution that enforces an execution order at block level thanks to new block synchronization primitive. We have shown that this solution is competitive in terms of performance (0.4-10 times) with the traditional hardware-based solution which is not reproducible. However, numerical reproducibility is ensured only for similar execution parameters as this solution depends on the number of threads per block and is therefore considered weakly reproducible.

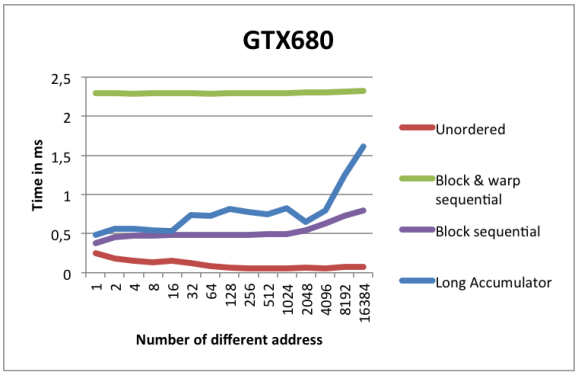
We have proposed another solution based on a very long accumulator, which eliminates every rounding error and makes floating-point addition associative. This solution avoids the need to enforce an execution order and provides strong numerical reproducibility. It provides a correctly-rounded result with optimal accuracy. We have shown that this solution is between 1.3 and 21 times more expensive than the unordered solution. On the other hand, this solution does not require hardware support for floating-point atomic addition and can be applied to any floating-point representation format.



(a) GTX480



(b) GTX560



(c) GTX680

Fig. 2. Execution time to atomically add 1 floating-point value per threads to a number of different addresses ranging from 1 to 16384. Kernel are launched with 100 blocks of 512 threads.

References

[1] K. Doertel, "Best known method: Avoid heterogeneous precision in control flow calculations," Intel, Tech. Rep., 2013.

[2] N. J. Higham, *Accuracy and stability of numerical algorithms*. SIAM, 2002, second edition. [Online]. Available: http://www.maths.manchester.ac.uk/~higham/asna

[3] (2014, july) N-body: Fp atomics v. recomputation. [Online]. Available: http://blog.cudahandbook.com/2012/11/02/n-body-fp-atomics-v-recomputation.aspx

[4] J. Allard, S. Cotin, F. Faure, P.-J. Bensoussan, F. Poyer, C. Duriez, H. Delingette, and L. Grisoni, "Sofa an open source framework for medical simulation," in *Medicine Meets Virtual*

TABLE I.
DESCRIPTION OF NVIDIA GPU' S OF DIFFERENT GENERATIONS .

GPU	Arch.	CUDA Cap.	#MP/ GPC	#MP	CUDA Core /MP	Warp. Scheduler /MP	GPU Clock (Mhz)	Memory Clock (Mnz)
GTX 480	GF100	2.0	4	15	32	2	1401	1848
GTX 560	GF114	2.1	4	7	48	2	1620	2004
GTX 680	K10	3.0	3	8	192	4	1059	3004

Reality (MMVR'15), Long Beach, USA, February 2007.

[5] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamari´c, D. H. Ahn, and G. L. Lee, "Determinism and reproducibility in large-scale HPC systems," in *Informal Proceedings of the 4th Workshop on Determinism and Correctness in Parallel Programming* (WoDet 2013), 2013.

[6] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units,* GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010, ser. ACM International Conference Proceeding Series, D. R. Kaeli and M. Leeser, Eds., vol. 425. ACM, 2010, pp. 94–103. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735706

[7] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk, "Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation on Multi- and Many-Core Architectures," INRIA, DALI–LIRMM, LIP6, ICS, Tech. Rep. HAL: hal-00949355, Feb. 2014.

[8] J. Demmel and H. D. Nguyen, "Parallel reproducible summation," *IEEE Trans. Computers,* vol. 64, no. 7, pp. 2060–2070, 2015. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TC.2014.2345391

[9] N. J. Higham, *Accuracy and stability of numerical algorithms,* 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2002.

[10] J.-M. Muller and al., *Handbook of floating-point arithmetic.* Birkhäuser, 2010.

[11] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro,* vol. 30, pp. 56–69, March 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.41

[12] D. Defour, "Impacting predictability of gpu's," *HAL-CCSD, Tech.* Rep. hal-00951920, 2013. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00951920

[13] V. Volkov and J. W. Demmel, "LU , QR and Cholesky factorizations using vector capabilities of GPUs," Department of Electrical Engineering and Computer Science, University of California, Berkeley, inst-UCB-EECS:adr, LAPACK Working Note 202, May 2008. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn202.pdf

[14] W. chun Feng and S. Xiao, "To gpu synchronize or not gpu synchronize?" in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on,* 2010, pp. 3801–3804.

[15] S. Xiao and W. chun Feng, "Inter-block GPU communication via fast barrier synchronization," in IPDPS. IEEE, 2010, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2010.5470477

[16] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for GPUs," CoRR, vol. Abs/1110.4623, 2011. [Online]. Available: http://arxiv.org/abs/1110.4623

[17] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming.* pub-AW:adr: Addison-Wesley, 2010.

[18] U. W. Kulisch, *Computer arithmetic and validity,* 2nd ed., ser. de Gruyter Studies in Mathematics. Berlin: Walter de Gruyter & Co., 2013, vol. 33, theory, implementation, and applications.

[19] T. Grandlund, "GNU MP: The GNU Multiple Precision Arithmetic Library," http://gmplib.org.

[20] G. Bohlender and U. Kulisch, "Comments on fast and exact accumulation of products," in *Applied Parallel and Scientific Computing.* Springer, 2012, pp. 148–156.