Fast Uniform Grid Construction on GPGPUs Using Atomic Operations

Davide BARBIERI [a], Valeria CARDELLINI [a] and Salvatore FILIPPONE [b]
[a] *Dipartimento di Ingegneria Civile e Ingegneria Informatica*
Università di Roma "Tor Vergata", Roma, Italy
[b] *Dipartimento di Ingegneria Industriale*
Università di Roma "Tor Vergata", Roma, Italy

Abstract. Domain decomposition based on spatial locality is a classical data-parallel problem whose solution may improve by orders of magnitude when implemented on a GPU. Among the data structures involved in domain decomposition, uniform grids are widely used to speed up simulations in a number of fields, including computational physics and graphics. In this work, we present two commonly used approaches to generate uniform grids on GPUs and propose a new single-pass method that has several advantages over the previous ones. We also present some performance results of our CUDA implementation of a broad-phase collision detection algorithm for particles simulation, comparing the different methods. In some tests our method achieves a speedup of 2 compared to the fastest known method supporting a fixed maximum number of elements per cell, and a speedup of 7 compared with the fastest method without such a constraint.

Keywords. Uniform grid, Space partitioning, Collision detection, GPU, CUDA

Introduction

Many real world problems naturally exhibit spatial locality, especially for particle-based simulations; this occurs in many physical models where we consider just short-range interactions while ignoring weaker long-range interactions, as well as in those cases where a certain phenomenon has effects only through some paths in space. When implementing a computational simulation of these models, we always take locality into consideration, not the least because of its performance implications.

For spatial locality problems, the regular nature of uniform grids enables a straightforward partitioning of space that can be easily accessed by data-parallel kernels running on the GPU, thus allowing a concurrent access to the elements of the simulation and avoiding the need to deal with unnecessary portions of large volumes of data. Uniform grids are widely used to speed up simulations for a multitude of topics, including computational physics [1] and graphics [2].

The many-core architecture of the GPU is relatively recent and its set of features is rapidly increasing over the years. Sometimes these changes improve the performance of some classes of instructions by order of magnitudes with respect to previous versions of the same hardware. Several research efforts have been devoted to obtaining the most

efficient algorithm(s) for uniform grid construction and traversal on the latest GPU hardware, e.g. [3,4,5]. In this paper, we review the state of the art of this research field, then we identify a specific performance improvement provided by the new NVIDIA Kepler architecture that allows the development of a faster and simpler method to build up a uniform grid when compared to the existing solutions in literature.

Our method exploits the Kepler's fast atomic operations in global memory to create a uniform grid made by a set of linked lists, using a single-pass kernel. We apply our method to a well known problem, the broad-phase collision detection method, showing the tradeoffs of our method compared to the state of the art methods and detailing their bottlenecks and performance issues.

The results we present are related to uniform grid construction. However, our considerations can be applied to speed up the generic problem of the parallel insertion of a set of elements inside a set of buckets, provided that each element is inserted in a certain bucket in a way that is independent of the other elements' target and that the peak number of elements per bucket is not too large compared to the average value. An example of such an application is the parallel insertion of a large number of elements in a hash table using the *chaining* method [6], in which each bucket is implemented as a linked list of elements that share the same hashcode. The implementations in [6] can achieve hash insertion rates of around 250 million pairs per second, while our best implementation can achieve around 2 billion pairs per second considering an input size of 1 million elements.

The rest of this paper is organized as follows. In Section 1 we describe how uniform grids are implemented on GPU by the existing methods. In Section 2 we present our new single-pass method. In Section 3 we then discuss the performance results of a well known application, i.e., the broad-phase collision detection, which has been implemented using all the uniform grid construction methods we consider in this paper. Finally, we summarize in Section 4 and discuss future work.

1. Uniform Grids on GPU

A uniform grid is the simplest form of space partitioning, since it splits the Euclidean space into equally sized volumes (called *cells*), for instance cubes. The size of each cell is usually chosen so that a single cell can contain the axis-aligned bounding box (AABB) of the largest element of the target simulation. This means that each element touches at most 8 adjacent cells. The advantage of this partitioning technique is that given an element and its AABB, locating the cells it touches is completely independent of any other inserted element. This simplicity makes the creation of the entire grid very fast on a parallel architecture; indeed a grid can be recreated on a GPU at each time step without significantly affecting the simulation performance.

In theory, it should be possible to perform incremental updates to the grid but, when dealing with dynamic environments on GPU, it is better to generate the grid data structure from scratch at each time step since this consumes a small fraction of the time step. There are two kinds of uniform grid, namely *tight* and *loose* grids, differing in the number of entries of each element within the data structure.

Tight Uniform Grids In the tight version of a uniform grid, each element is inserted inside every cell touched by its AABB, at most 8 adjacent cells. Therefore, the resulting grid will have more insertions per element (up to 8), but a search for colliding AABBs

of a single element will be limited to maximum of 8 adjacent cells, the cells touched by the given element.

Since there are multiple entries for each element in the grid, pairs of elements may be present in different cells' buckets. For this reason we need to find a way to enable the threads to process each pair exactly one time, thus avoiding repetitions. To achieve this goal, we associate a bitmask of three bits to each entry in a cell. In this bitmask ($b_x b_y b_z$), b_i is equal to 0 if the AABB touches that cell with its negative side along the i coordinate, otherwise it is equal to 1. Actually, b_i is 1 if the same element is also inserted in the previous cell along the i coordinate. Then, a pair of elements should be processed only when the bitwise operation ($b_x b_y b_z$ *AND* $b'_x b'_y b'_z$) is equal to 0. After the insertion, this test becomes true exactly in one instance, because every time two AABBs are inserted in the same cell with b_i *AND* b'_i equal to 1, there is also an insertion with b_i *AND* b'_i equal to 0 in the previous cell along the i direction.

Loose Uniform Grids In the loose version of a uniform grid, each element is inserted just once, usually in the cell containing the center or a particular vertex of the AABB. In our implementation we have chosen the vertex whose local coordinates (relative to the AABB) are equal to 0. In this case, we have just one insertion, but searches for colliding AABBs are extended to a minimum of 8 and a maximum of 27 cells; on the other hand, there is a smaller number of entries in each cell.

Related Work We now review two methods that have been proposed to build a uniform grid with the Compute Unified Device Architecture (CUDA) C language on NVIDIA GPUs. The first method is the atomic increment construction algorithm, while the second is the sorting construction algorithm [3,4]. Both methods are based on the hash id of the destination cell, which is computed directly from its position according to two possible options:

- Using a function that joins the three coordinates (x, y and z), each comprising k bits, in a single integer $z_k \ldots z_2 z_1 z_0\ y_k \ldots y_2 y_1 y_0\ x_k \ldots x_2 x_1 x_0$;
- using a function that maps cell coordinates to integers following a space filling curve, like the Z-order curve (also called Morton curve), Peano curve or Hilbert curve [7,8].

In many applications of uniform grids, including those described here, each thread accesses the data of elements that belong to different cells mapping adjacent space areas. The space filling curve option is used by some implementations based on the sorting method [3,5] to place the elements' data of neighboring cells in adjacent linear memory addresses. In this way, elements from multiple adjacent cells can be accessed with an improved cache utilization and therefore higher performance. On the other hand, when using a method that does not provide a sorting pass of elements' data, the use of a space filling curve is quite non influential.

Building Uniform Grids Using `atomicAdd` An atomic operation is guaranteed to read, modify and write back a value in memory, both global and shared, without generating race conditions with other threads. This property is essential in the concurrent construction of data structures such as hash tables, linked lists, and trees. Green in [3] presents a method that uses the CUDA `atomicAdd` function to construct a uniform grid in a single pass. The method is very simple, but it also has some limitations.

The grid is implemented by two fixed-sized allocations:

- `cellCount[numberOfCells]` is a vector storing the current number of elements inserted in a cell (initialized to the zero vector at each time step start);
- `cellElements[numberOfCells*maxElementsPerCell]` is a matrix storing for each cell the vector of inserted elements ids.

We observe that since the `cellElements` matrix is fixed-sized, there is a hard limit on the number of elements contained in a cell.

A single kernel can be used to concurrently insert all the elements in the data structure. To achieve this, each thread inserts a single element in three steps:

- First, it computes the index `cellId` of the cell in which the element should be inserted;
- then, it calls `atomicAdd` with argument 1 on the address of `cellCount[cellId]`;
- finally, the thread uses the returned integer (i.e., the value previously stored in the operation's destination address) as a unique index in the cell vector `cellElements[cellId]` to write the element's id.

The same can be expressed in CUDA with:

```
int old = atomicAdd(&cellCount[cellId], 1);
cellElements[cellId*maxElementsPerCell + old] = elementId;
```

In our final implementation of the grid construction method based on `atomicAdd`, we have chosen a column major memory layout for `cellElements`. Therefore, we store `elementId` to `cellElements[cellId + old*numberOfCells]` because we have found that the column major memory layout is faster (up to 15%) than the row-major order in our performance tests.

This method can be extended to avoid the limitation of the maximum entries count per cell, executing these passes:

- counting in a first pass the number of entries to be stored in each grid cell, using the atomic increment code shown above;
- performing a *parallel prefix sum* on the entries' count array;
- in the final pass, evaluating again all the entries to be inserted, but this time writing them to contiguous locations in a packed grid array; this can be done using the results of the prefix sums pass.

Building Uniform Grids Using Sorting The method based on `atomicAdd` has two main drawbacks. The first one is that atomic operations are slow in many GPU architectures, even when there are no collisions during atomic accesses; moreover, significant slowdowns may occur with collisions. The second drawback is that if we want to extend the method to avoid the fixed length limitation, a parallel prefix sum on the whole grid is needed. This is a limiting condition, since the performance of this kernel does not scale with the domain size (i.e., the number of elements) but with the grid size, incurring severe slowdowns when using refined grids.

Previous efforts in [3,4] propose an alternative approach which addresses both issues by means of a sorting algorithm. The method consists of the following passes:

1. the first pass evaluates and stores each (cell id, element id) pair in a temporary array;
2. the second pass sorts these pairs on the basis of their cell id value so that elements of the same cell are stored sequentially in memory;
3. the last pass finds the start index and end index of every cell in the sorted array.

The sorting can be performed using several different algorithms, but *radix sorting* is the fastest open source method on GPU reported in literature. It was first implemented on GPU by Satish et al. in [9], then improved by Merill and Grimshaw in [10], and finally included in the highly-tuned GPU Thrust library [11]. This fast radix sort implementation is the same used in [3] and is based on the parallel prefix sum kernel.

The last pass is performed by comparing each element in the sorted array with the previous and the next entries: if the cell id of these entries differ, then that element is positioned at the start or the end index for that cell. Then, it is possible to find every grid entry of a given cell just reading the pairs stored on the sorted array from the cell start index to the cell end index found in the previous passes.

Since the method avoids the use of atomic functions, its performance is not affected by the distribution of the elements in the grid. Moreover, it scales with the problem size rather than the grid size, since no empty cell is considered in the sorting process.

As a further optimization, Green for his particle simulation in [3] also sorts the elements data along with the elements entries in the grid. We observe that this sorting is convenient for a particle simulation that has a low quantity of data for each particle, but it may cause slowdowns in simulations with a high amount of data per element.

2. A New Uniform Grid Building Method

In this section we present a new method which relies on the `atomicExch` function to build up a uniform grid in the form of a set of linked lists.

NVIDIA has introduced in Kepler, its latest GPU architecture, some performance enhancements concerning atomic operations. As reported in [12], the atomic operations speedups range from 2.5 to 11 compared to the Fermi architecture, especially in the presence of collisions. NVIDIA also states that Kepler atomics are fast enough to be inserted in inner loops. These significant performance gains suggested to us that a grid construction method based on atomic operations can still be used on Kepler, avoiding the drawbacks stated in [3,4].

In our method, the grid is represented by two arrays:

- `cellHead`, which stores for each cell the first inserted element id;
- `cellNext`, which stores for each entry the id of the next inserted element in the list.

With this configuration, the basic step to achieve a concurrent insert of an element with unique ID `elementID` in a cell whose unique ID is `cellID` is:

```
int lastStartElement = atomicExch(&cellHead[cellID], elementID);
cellNext[elementID] = lastStartElement;
```

The `atomicExch` function reads `cellHead[cellID]` in `lastStartElement` and writes `elementID` in `cellHead[cellID]` in one atomic transaction. This permits each thread to insert its element in the list head `cellHead[cellID]`, taking the previous head element and storing it as the *next* pointer for the inserted element. We observe that `cellNext` will be accessed with coalesced writes, since `elementID` is generated from the linear thread id. In the tight uniform grid version, the `cellNext` allocation is 8 times larger than the loose version, since every element can have up to 8 entries in the grid, and each entry ID is stored using the last three bits to keep track of which entry belongs to the linked list. Clearly, this method can be applied to simple uniform grids or any kind of linked lists group (in such cases `cellID` would be a generic `listID`).

Using these basic steps, we can build an entire uniform grid with no significant impact on the whole time step performance. At every time step start, the entire grid is initialized and rebuilt, as shown in Figure 1. The memory footprint of our method with respect to the `atomicAdd`-based approach is minimal, since all elements can fit in the same cell with no padding; moreover, all non-atomic operations for insertion are coalesced. Compared to the approaches known in the literature, our new method is actually simpler, since no additional passes are needed to support independence from elements distribution for the memory allocation. Note however that the elements distribution still influences the atomics operations performance.

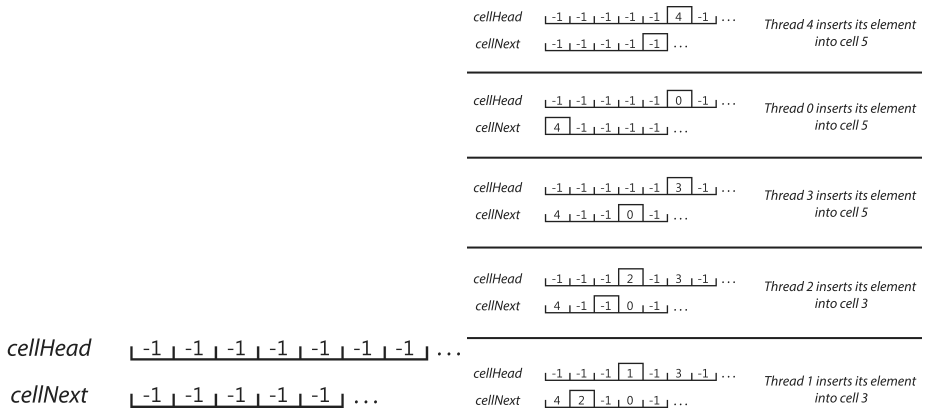

Figure 1. Grid initialization (left) and construction (right)

The drawback of our method is that the identifiers of the elements in the same linked list, stored via the *next* pointers array, are not sequential in memory, so an entire list of ids may not be read efficiently. However, in real world simulations to each element there is a certain amount of data, so it can be convenient to pack the *next* pointer directly within the allocation for the element's data, e.g. directly after three float coordinates values, so that it is possible to read both the position and the next pointer with a single CUDA's machine instruction implementing `float4` 16-bytes read operation. This change makes for a slower insertion throughput due to the loss of coalescence of non-atomic operations, but it leads to remarkable speedups in the complete simulation, given that insertion requires a comparatively small amount of time, since both grid entries identifiers and simulation data can be read in a single memory access. Moreover, these scattered read operations are implemented in our kernels using *texture cache* lookups, allowing us to reach up to 45% of additional performance improvement compared to using global memory reads.

3. Experimental Results

To evaluate the performance of the above space partitioning methods, we consider a classic application of uniform grids: the broad-phase collision detection for interacting particles simulation. We first briefly describe the benchmark and then discuss some performance results.

The collision detection algorithm is used in the simulation of interacting particles: it finds which pairs of particles are sufficiently close. In the simplest case the interaction is a collision, hence the name. The algorithm is usually implemented with a two-phase approach, a *broad phase* followed by a *narrow phase*, to find intersections between simulation's elements. The broad phase usually has a fast approximate implementation and it is used to minimize in a conservative way the number of tests that have to be performed in the narrow phase. During the narrow phase we perform exact intersection tests which are more complex and costly, but every element is tested against a smaller set of possible collisions. The broad-phase collision detection is the foundation of the so called discrete element method (DEM), which is an appealing simulation technique for computing the motion of a large number of particles in granular media and solving engineering problems in granular and discontinuous materials, like granular flows, powder mechanics, and rock mechanics [13].

Our benchmark implements the collision model described in [3], which consists of a spring force which spreads the particles apart, a dashpot force which causes damping, and a shear force which causes viscosity. Its time step is composed by three stages:

1. first, each particle's id is inserted in the target cell of the uniform grid;
2. then, for the narrow phase, a kernel checks the possible collisions of each particle with all the other particles that were inserted in the same cell; the kernel updates also the velocity of each collided particle;
3. finally, an integration step updates the particle position along the velocity vector and simulates collisions with fixed boundaries.

The broad phase is implemented in the first stage, reducing significantly the number of collision tests to be executed in the second stage.

Our benchmark involves different test cases with an increasing number of interacting particles left free to fall in an empty closed box, using the space partitioning scheme described above to limit collisions tests just to those elements inserted in the same or in the adjacent cells. The empty space in which the particles can move is partitioned with a uniform grid made by $64 \times 64 \times 64$ cells. In our tests all particles have the same diameter, which is also equal to the cell length. When the number of elements increases, the box becomes full and so the concentration of particles per cell increases. The expected behavior for each method is an increasing performance (in particle updates per second) for simulations with higher number of particles, due to the better utilization of the GPU, until a certain threshold, after which we should experience a performance decrease due to a greater number of intersection tests (which scales with the square of the number of particles per cell).

3.1. Performance Analysis

We now compare the performance results of the above described benchmark, implemented using all the uniform grid construction methods in Sections 1 and 2. All the ex-

periments have been executed on the NVIDIA GeForce GTX 660 card, which is based on the Kepler architecture, by running a particle simulation on a $64 \times 64 \times 64$ uniform grid.

Each experiment starts with all the particles uniformly distributed in the left half ($x < 32$) of the free space of the box, with a little additional random offset in their positions. Due to gravity, they fall and when they hit ground, they tend to distribute themselves uniformly on the bottom of the box, generating a wave. The simulation proceeds for 2000 time steps, which is enough to damp the wave completely.

Figure 2 shows the average performance of a single time step over the whole simulation. To point out how the uniform grid construction phase affects the performance of each time step, we also show in Figure 3 the performance of the insertion kernel in the slowest time step of the simulation. On the x-axis of all the figures, we report the number of particles. In Figures 2 and 3, the curves labeled as **Fixed Grid** and **Sorting (Thrust)** correspond, respectively, to the implementation based on the *atomicAdd* function and that by Green based on sorting (which calls the routines of the Thrust library, included in the CUDA toolkit). The remaining performance curves refer to our three implementations based on linked lists. Specifically, **Linked List Loose** and **Linked List Tight** store, for each node, the index of the next node in dedicated arrays, while **Linked List Loose Packed** stores this index in the same memory area used to store the domain data, that is, replacing the 4 bytes of padding after the particle velocity.

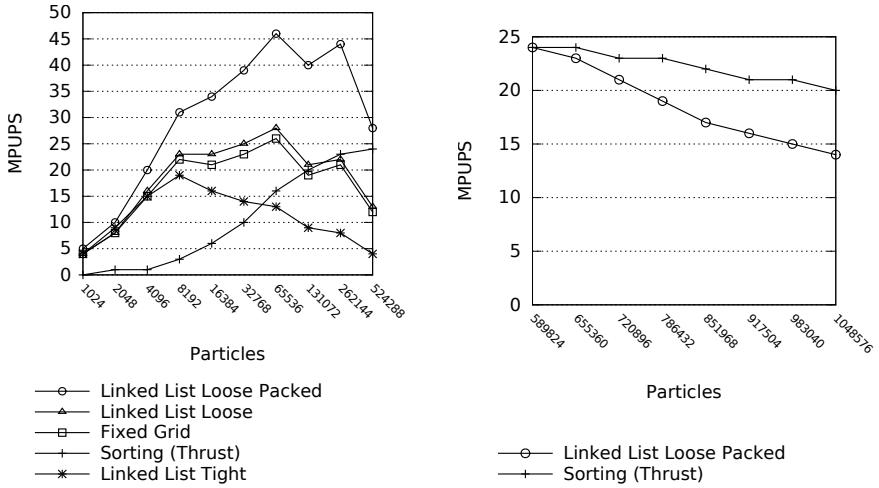

Figure 2. Comparison of the broad-phase simulation performance using different uniform grid methods. Performance results are provided in terms of MPUPS (million particle updates per second).

Figure 2 confirms our expectations: the *Linked List Loose Packed*, *Linked List Loose* and *Fixed Grid* methods have their performance peak at 64K particles, the *Sorting* curve reaches its maximum at 512K particles, while *Linked List Tight* has its peak at just 8K particles. After these thresholds, all the curves decrease. Indeed, when the concentration of particles per cell is high, the same elements must be read over and over again, the number of collision tests per particle increases with the square of the elements' count, and so the computation of forces acting on a particle takes more time. When the space

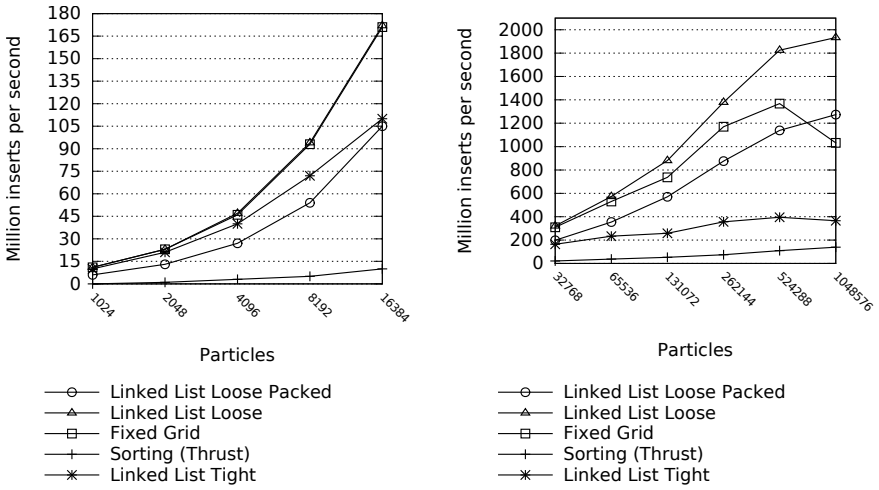

Figure 3. Performance of elements insertion in the uniform grid (small and large size problems)

is full of particles, the collision model allows every particle to push the particles below due to gravity acceleration, increasing the concentration of particles per cell on the lower layers. In our benchmark, at 512K elements of simulation, every particle must be tested on average against 3.5×27 elements. All curves except *Sorting* show a similar trend, and we can observe that the *Linked List Loose Packed* method is faster than all other methods based on atomics operations in our test cases.

Figure 2 also shows that loose grid kernels are better than tight ones at all sizes. As previously stated, in our benchmark the particle's diameter is equal to the cell size, so particles often touch 8 cells. In the loose version, the particle's id will be inserted in only one of these cells, and afterwards then the thread which checks collisions for that particle will traverse 27 linked lists of neighbouring cells. In the tight version instead, each particle's id will be inserted in 8 linked lists; each collision testing thread will traverse only 8 linked lists, but at high occupancy each of these lists tends to have [1] an average number of elements that is 8 times larger than that in the loose version. Therefore, in the tight version, each thread will perform up to 8 times the number of insertions made in the loose version, and will traverse up to $8 \times 8/27 = 2.37$ times the number of linked list's nodes traversed in the loose version.

The *Sorting* method is quite different. When considering small size simulations, any method based on atomic operations is faster than *Sorting* (see Section 2); in particular, our packed implementation reaches a speedup of almost 7. As explained in Section 3, the *Linked List Loose Packed* has a slower insertion kernel than the *Linked List Loose* method, which has coalesced accesses on all non atomic writes, but provides a faster overall timed, achieving a speedup greater than 2. On very large problems however (over 512K particles) *Sorting* outperforms all other methods.

To understand this behavior we should examine Figure 3; we can see that *Sorting* has a much slower insertion kernel, up to 17 times slower than the best, and this is the main bottleneck for many small test cases. Whereas the worst among the other curves reaches

[1] Note that the bottom of the box has a size of $64 \times 64 = 4096$ cells

105 million particle inserts per second at 16K particles, making insertion time negligible for small problems, *Sorting* reaches a comparable level only near 512K particles. In the sorting method, however, the particles' velocities and positions are sorted in memory according to the containing cell id, following a space filling curve; this allows for a better cache pattern when accessing elements' data, so that when the bottleneck shifts from entries' insertion to elements' data reading the sorting method becomes the best.

A further drawback of atomics-based methods is that they are nondeterministic, because the order of the elements in the same linked list depends on the chronological order of atomics operations performed on the same address. This could be a problem, since the impossibility to reproduce the numerical result of a particular experiment makes both debugging and numerical stability analysis noticeably harder.

4. Conclusions

The performance improvements of modern GPUs on atomic operations make it possible to build uniform grids using a single-pass kernel that is simpler and faster for many application cases with respect to state of the art approaches. In future work we will focus on using atomic operations also for fast construction of tree-based spatial partitioning structures, like octrees, which are mainly used in environments that need non-uniformly refining.

References

[1] J. Zheng, X. An, and M. Huang. GPU-based parallel algorithm for particle contact detection and its application in self-compacting concrete flow simulations. *Comput. Struct.*, 112-113:193–204, 2012.

[2] S. Guntury and P. J. Narayanan. Raytracing dynamic scenes on the GPU using grids. *IEEE Trans. on Visualization and Computer Graphics*, 18(1):5–16, 2012.

[3] S. Green. CUDA particles. Technical report, NVIDIA Corporation, 2008.

[4] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *Proc. of 1st ACM Conf. on High Performance Graphics*, HPG '09, pages 23–28, 2009.

[5] D. Madeira, A. Montenegro, E. Clua, and T. Lewiner. GPU octrees and optimized search. In *Proc. of VIII Brazilian Symposium on Games and Digital Entertainment*, pages 73–76, 2010.

[6] D. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta. Building an efficient hash table on the gpu. In W. W. Hwu, editor, *GPU Computing Gems Jade Edition*, chapter 4. Morgan Kaufmann, 2011.

[7] A. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, 1966.

[8] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[9] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proc. of 2009 IEEE Int'l Symp. on Parallel & Distributed Processing*, IPDPS '09, May 2009.

[10] D. G. Merrill and A. S. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *Proc. of 19th Int'l Conf. on Parallel Architectures and Compilation Techniques*, PACT '10, pages 545–546, 2010.

[11] N. Bell and J. Hoberock. Thrust: a productivity-oriented library for CUDA. In W. W. Hwu, editor, *GPU Computing Gems, Jade Edition*, chapter 16. Morgan Kaufmann, October 2011.

[12] NVIDIA Corp. NVIDIA GeForce GTX 680. Technical report, NVIDIA Corporation, 2012. `http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf`.

[13] C.-Y. Wu, editor. *Discrete Element Modelling of Particulate Media*. Royal Society of Chemistry, 2012.