Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2015)
F. Jaillet, G. Zachmann, and F. Zara (Editors)

A More Efficient Parallel Method For Neighbour Search Using CUDA

Daniel Morillo, Ricardo Carmona, Juan J. Perea and Juan M. Cordero

Universidad de Sevilla, Sevilla, Spain

Abstract

In particle systems simulation, the procedure of neighbour searching is usually a bottleneck in terms of computational cost. Several techniques have been developed to solve this problem; one of particular interest is the cell–based spatial division, where each cell is tagged by a hash function. One of the most useful features of this technique is that it can be easily parallelized to reduce computational costs. However, the parallelizing process has some drawbacks associated to data memory management. Also, when parallelizing neighbour search, the location of neighbouring particles between adjacent cells is also costly. To solve these shortcomings we have developed a method that reduces the search space by considering the relative position of each particles in its own cell. This method, parallelized using CUDA, shows improvements in processing time and memory management over other "standard" spatial division techniques.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.6.8 [Computer Graphics]: Types of Simulation—Parallel

1. Introduction

In the field of Computer Graphics, the use of particle systems for dynamic environment simulation is widely applied. To obtain realistic simulations, particles must show a high level of cohesion. This is because each particle belonging to the system interacts with the closest particles or *neighbouring particles*. The amount of neighbouring particles is determined by a given distance, which is called *influence radius*. In terms of computing cost, the search of neighbouring particles is one of the main bottlenecks of simulation, and requires the correct techniques for the optimization of its processing. Otherwise, the computational cost is prohibitively increassed [GDNB10].

The most basic technique is the exhaustive search [GDB08]. It consists of calculating the distance between every particle and the rest of the system's particles, selecting only those that are at the same or at a lower distance than the influence radius. Although this technique allows to calculate the neighbouring particles, it does not seem appropriate due to its high computing cost, in the order of $\mathcal{O}(n^2)$.

There are several techniques that can reduce this high computing cost. These techniques operate mainly in stages: *spatial division* and *analysis–allocation*. In the spatial division stage the space is divided in cubicles or *cells*, so the particles are associated to the cells according to their location. In the analysis–allocation stage the distances between the particles of each cell and the adjacent particles are checked, in order to assign the neighbouring particles from the influence radius. For these techniques to be operational, it is necessary that the cells are organized in a sorted structure. Tree structure [Ben75] and hash function structure [IABT11] are the two most common structures. Tree structure basically consists in subdividing the space into decreasingly arranged hierarchies, allowing tracking each cell and their adjacent in a relatively short period of time. On the other hand, a hash function is used in hash sorting to obtain an integer, ideally a unique number, from the centre of each cell. Thanks to the hash codes obtained, an organization allowing the quick location of each cell and their adjacent is established. It should be noted that the techniques based on hash demand the inner particles of each cell to be associated to their own hash code.

Both techniques are comparatively appropriate to manage the neighbouring search, and both show satisfactory results. Nevertheless, two features in the hash sorting stand out over the tree structure. The first one is the static structure generated, which does not require to be recalculated at every step of the simulation. The second one is that the calculation of

DOI: 10.2312/vriphys.20151339

each particle's hash code is performed individually, making it suitable for its parallel processing [LH06].

There are different parallel programming architectures. Notable among these is CUDA technology, developed by NVIDIA, embedded in graphics processing units (GPUs). The cornerstone of this technology is that there are a high number of processing cores. Each processing core can execute many tasks in parallel at high speeds. Taking advantage of the computing power offered by CUDA in neighbouring particles search reduces the computing costs, making real time simulation feasible [RBG*12, Kno09] even when an exhaustive process is used [GDB08].

Despite the advantages of CUDA, the GPU memory management affects the efficiency of the neighbouring particles search, as the access to very dispersed data slows the process down. There are two ways to optimize the memory management: the first one is the use of techniques controlling the memory dispersion of information. The second one is reducing the information flow during processing as much as possible.

In this article we present a new technique that meets these two limitations, while we achieve to reduce the computing cost of neighbouring particles search. This technique is based on the division of space in cubic cells using a hash function to tag them. It uses the relative location of each particle inside the cell to reduce the amount of adjacent cells where neighbouring particles are searched for. Thus, the information consulted is as minimal as possible.

After this introduction, the article is structured as follows: In Section 2, we will describe the most relevant research in the field of neighbouring particles search. This research shapes the environment in which our technique is originated. In Section 3, we will describe the foundations of CUDA architecture, focusing on its capabilities and limitations, as they are important to the development of our technique. In Section 4, we will describe the foundations of the *standard* technique for cell space division, like the one used by Simon Green in [Gre10], and hash function tagging, while we prove the limitations of this technique. Thus, we will establish the conceptual environment in which our technique is based on, and we will discuss the progress it provides. In Section 5, we will describe the proposed technique, its capabilities and the bases for its implementation in CUDA. In Section 6, we will show the progress obtained thanks to our technique. To do so, we will compare our technique with the standard spatial division technique [Gre10]. Finally, in Section 7, we will talk about the conclusions obtained after analysing the collected results.

2. Related Works

In the dynamic simulation of particles systems, the most used techniques for neighbouring particles search are based on space division. Bentley et al. [Ben75] develops one of the pioneers techniques based on a tree structure covering the entire simulation space. Several improvements to the model proposed by Bentley have been developed to optimize the performance of this technique. Kumar et al. [KZN08] describe tree structures overcoming some limitations of the original proposal, as the node divisions used are always axis–aligned. Nevertheless, even though the tree structure has been widely used, it shows some disadvantages if parallel processing is to be implemented, specially when the tree has a large number of levels and it requires to be continuously recalculated in the simulation [PDC*03].

In parallel implementation, the most effective methods are those which discretize the simulation space into cells of equal size. Harada et al. [HSK07] relate each particle with the container cell by coordinate texture. Thus, each cell is coded by a pixel and is assigned to three-dimensional computational space. Ihmsen et al. [IABT11] propose a particles tagging technique by hash function. Garcia et al. [GDB08] study this technique and stress two main features: the rapid access to memory data and the ability to be parallelized.

Most current research is focused on formulating the optimal hash function to guarantee the uniqueness of each hash code with no excessive computing costs involved. Müller et al. [MPG03] or Teschner et al. [THM*03] present a function based on exclusive logic operations in which large prime numbers are involved and require to be modulated. Although it is very used, it may involve collision problems within the results due to the modulation of the values [Cay12]. A more appropriate formulation is proposed by Fan et al. [FWZS11] where the hash function reduces the possible hash collisions as it does not require modulation.

In models of cell–space discretization that use a constant influence radius, an appropriate size of the cells is needed so the efficiency of the process is not affected [WBK07]. However, Wroblewski et al. [WBK07] develop a detailed study on the optimal size of the cells for other techniques which do not necessarily use a constant radius, specifically using a constant number of neighbours for each particle. His conclusions show that the values for the cell side size fall between R and $2R$, where R is the influence radius. In a similar research approach, Viccione et al. [VBC08] sets an sorting criterion for neighbouring search, not just the cell containing the particle, but the 26 adjacent. This technique offers satisfactory results but it is insufficient, as it has to search for every adjacent cell. Ihmsen et al. [IABT11] discretizes the space in cells whose size is the same as the influence radius R. In his proposal, he assesses the value of each cell, suggesting that increasing the cell size produces better results.

Several research take advantage of the abilities of GPU parallel processing [Cud12, SGS10]. We will focus on CUDA technology, as it provides a higher performances, taking into account that this technology is linked to the use of NVIDIA hardware -which is not the case of OpenCL-. However, parallelization using CUDA may involve prob-

lems in memory management, specially errors related to cache [IABT11]. To overcome this barrier, Goswani et al. [GSSP10] develop a research describing the problem of the memory overhead. To reduce it, they propose using a z–indexing technique to sort the information into the memory. Likewise, Domínguez et al. [DAM13] carry out an exhaustive description of the bottlenecks that appear in the neighbouring search process. They set a relationship between the particles position and the cell that contains it. By this relationship, each particle is tagged with its container cell. To reduce the memory overhead, they propose using a sorting of the particle's tag. Several sorting algorithms have been developed together to those which group the information stored in memory. Relevant algorithms are Bitonic Sort [PSHL10], Sample Sort [LOS10], among others [Akl14]. The most recent version of Radix Sort [SHG09, Hwu11] is one of the most used due to its high speed, sorting the data without taking into account the previous sorting. This can decrease the efficiency, as it misses the previous sorting. Another way of facing problems related to memory management is using specific memory types which are able to access data in a more efficient way. Rozen et al. use texture memory, located in the processor (chip) cache offering a great spatial locality [RBA08].

3. CUDA

CUDA (Compute Unified Device Architecture) is a computing model developed by NVIDIA. It uses the GPU parallel processing capacity to obtain a great computing power, supporting different programming frameworks. It can be used in a wide range of graphic software by NVIDIA, starting from GeForce 8 series (Tesla architecture).

CUDA structure is hierarchically organized for both processing elements and memory. Processing elements are: *multiprocessors*, *cores* and *threads*, while memory types are: *global memory*, *shared memory* and *local memory*. Within the processing elements, each multiprocessor is composed of several CUDA cores containing groups of threads of execution. Each thread will access its own local memory, which cannot be accessed by any other thread of the same group. If threads of the same group need to share information during processing, the shared memory must be used. Comparatively, the writing and the access to the shared memory data is slower than the local memory, so it can slow the processing down.

The processing is performed in groups of 32 threads, called warps [Mic12]. A warp is the minimal data unit handled by a multiprocessor, optimizing the amount of operations and improving performance. In addition, not only threads of the same group can share information, the different groups can also exchange data using the global memory. The importance of global memory is essential, as the size of thread groups is limited and requires the synchronization between different groups to achieve an optimal and massive parallelism [Cud12]. This synchronization can only be carried out in the global memory, where access to information slows the processing down.

To optimize the performance in CUDA implementations, the limitations of its architecture regarding memory management and GPU execution flows must be borne in mind. For an optimal memory management, the data stored in the global memory must be very close [IABT11]. This limitation is related to cache, so as more information is transferred to the cache for every data request, the fewer requests and the quickest parallel processing [Ros13]. This is essential when all threads execute the same parallel request, as the accesses to the data stored in the shared memory need to be consecutively set, that is, they need to be *coalesced accesses*. The most usual method to avoid memory data disaggregation is the use of sorting algorithms. The process starts with a dispersed memory layout, in which every data must be linked to a tag to be sorted.

Considering a sorting criterion, tags -and so data- are sorted so as the information is consecutively set. There are several sorting algorithms: [SHG09, Hwu11], with Radix Sort [MG10] being the most efficient. On the other hand, CUDA execution flow depends on the processing of conditional instructions executed in the same warp, as they introduce different routes of execution. As CUDA goes over these routes in a consecutive way [Cud12], an excessive use of these instructions reduces the level of parallelism of the functions using them [FSYA07].

4. Neighbouring Search

Neighbouring particles search using standard spatial division is based on the fact that all particles have their neighbours inside the cell containing them and, eventually, in adjacent cells. The standard method considered will be the one used by NVIDIA [Gre10]. In this context, the particle's influence radius needs to be associated to the cell size.

For this technique to operate, it is necessary to establish a many–to–one connection between the particles and the cells containing them, as well as determine a sorting criterion among different cells. The usual way to meet this requirement is hash function tagging. This way, the cells and the particles are univocally tagged and related, as shown in Figure 1. Besides, the hash codes associated to cells allow us to set the sorting structure.

Descriptively, this technique can be divided into two stages: spatial division and analysis–allocation.

In the spatial division stage, the size for each cell is set. Wroblewski et al. [WBK07] develop a study on the optimal size of each cell, dividing the simulation space in non-overlapped connected cells. Then, the centre of each cell is located and an integer value is obtained using a hash function. This ideally unique value "tags" the cell; the same hash

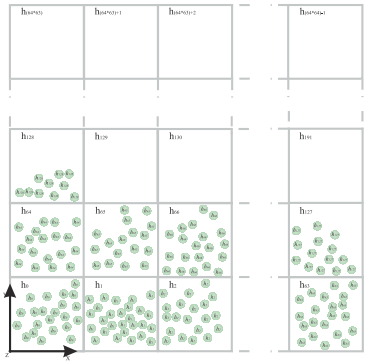


Figure 1: Representation of the tagged cells and particles

value is used to tag the particles contained within the cell. This way, a one–to–many relationship between the cell and the particles contained in it. To avoid a memory overflow, only hash codes of the cells containing particles must be stored in memory.

In the analysis–allocation stage, the occupied cells are checked. The distance inside every cell is calculated, that is, the particles associated to the same hash code. If the distance is the same or lower than the influence radius, the particles are neighbours, but not the only ones. As the influence radius defines a closed sphere centred on the particle and the cell is generally cubical, it is necessary to complete the neighbouring search in adjacent cells.

As tagging particles and neighbour allocation are independent processes for each particle, this technique is easily parallelized, making it possible to use CUDA. However, CUDA's architecture limitations regarding memory management must be kept in mind, specially those related to data dispersion in memory, which reduces the number of coalesced memory accesses. This limitation has an impact on the hash function selected, as the results obtained may cause data dispersion. This is enhanced by the necessity of searching neighbouring particles in adjacent cells, which is 27 in the three–dimensional case. The reason for this limitation is that the hash values in the neighbouring cells will be separated from the memory and the process will slow down.

This is why a hash function that induces the minimal dispersion possible is required, as well as developing a methodology reducing the amount of adjacent cells to a given minimum amount in which to search neighbouring particles. Our technique pursues that goal, as we explain in the following section.

5. Proposed Model

We present an efficient technique for neighbouring particles search, optimized for its implementation in CUDA. It is based on the spatial division of non-overlapped connected cells, using the relative location of each particle inside the cell to reduce the amount of adjacent cells where neighbouring particles are searched for.

As every technique based on spatial division, there are two different stages. In the spatial division stage we develop a standard spatial discretization in cubic cells of equal size. The cells are tagged using a hash function. We use the same hash function to tag particles in order to pair particles with the cells containing them. In the analysis–allocation stage, we determine the relative location of each particle inside the cell containing it. This relative position will allow us to significantly reduce the process of searching in adjacent cells. Thus, the cache miss rate is reduced, meaning a refinement of accesses to global memory, which is the slowest in processing [LCT14].

The algorithm 1 describes the main steps needed for the implementation of our proposed technique. In sections 5.1 and 5.2 we will carry out a more detailed description.

Algorithm 1 Neighbour search process.

Require: The space must be segmented in cells of length e. Each cell is tagged with hash function 1.
Ensure: Each particle has retrieved all its neighbour particles and knows their positions.
Input: Particles' positions, Influence Radius.

```
1:  for Each particle do
2:      Evaluate its hash code, which coincides with the hash
        code of its cell.
3:      Tag the particle with its cell's hash code.
4:  end for
5:  Sort hash codes
6:  for Each particle a do
7:      Determine which octant contains the particle.
8:      Calculate the subset of searchable cells N_Adj.
9:      for Each particle bcontainedinacell from N_Adj do
10:         Calculate the distance between a and b.
11:         if (distance <= influence Radius) then
12:             b is neighbour of a
13:         end if
14:     end for
15: end for
```

5.1. Spatial Division Stage

In spatial division, the cells size has an impact on the efficiency of the process [WBK07]. In our case, as we will show in section 6, the optimal value is obtained for cells with a $e = 2R$ edge length, that is, cells of $V = 8R^3$ size. Given this size, we discretize the space clockwise order, that is, we first discretize in X direction, then in Y direction, and finally in Z direction, see Figure 2. Considering that CUDA thread blocks reach the instructions in a similar way that single instructions do on linear code, the distance between data makes the nested $X \rightarrow Y \rightarrow Z$ loop the most useful structure

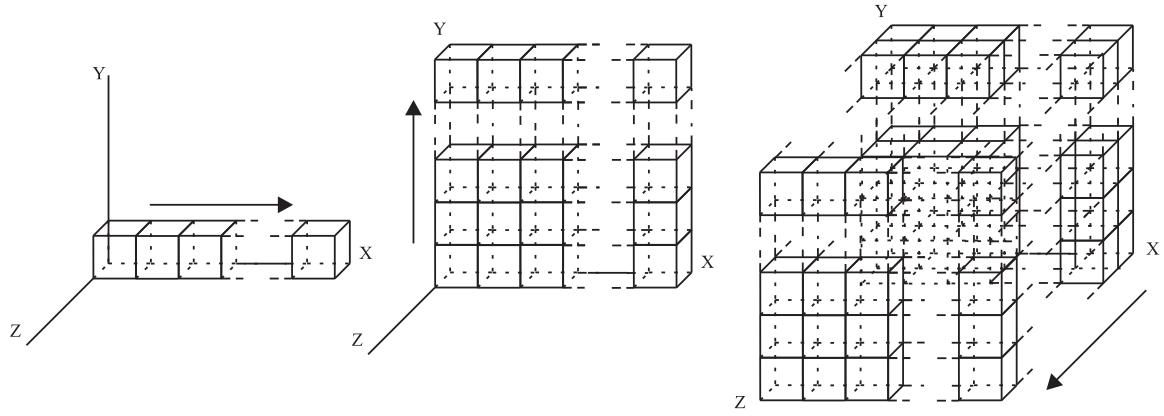


Figure 2: Segmentation order.

for our purpose. This fact is valid whenever a clockwise segmentation order is used, i.e., $X \to Y \to Z$, $Z \to X \to Y$ or $Y \to Z \to X$.

The next step is tagging each cell using a hash function. We use the function

$$Hash(x,y,z) = z_{trunc}\,\varepsilon_y\,\varepsilon_x + y_{trunc}\,\varepsilon_x + x_{trunc} \qquad (1)$$

where ε_x and ε_y are the number of subdivisions in x and y direction respectively, and x_{trunc}, y_{trunc} and z_{trunc} are the truncated coordinates of each particle that satisfy the equation 2.

$$x_{j\,trunc} = \left\lfloor \frac{x_j}{R} \right\rfloor \& (\varepsilon_{x_j} - 1) \quad \forall j = 1,2,3; \qquad (2)$$

where $x_{1\,trunc}, x_{2\,trunc}, x_{3\,trunc}$ refer to $x_{trunc}, y_{trunc}, z_{trunc}$ and $\varepsilon_{x_1}, \varepsilon_{x_2}$ and ε_{x_3} refer to ε_x, ε_y and ε_z respectively.

The expression 1, based on the hash function formulated by Fan et al. [FWZS11] guarantees the uniqueness of the hash codes obtained and favours the close cells to be associated with hash codes close to memory. Once the cells are tagged using hash codes, the next step is determining the amount of particles contained in each cell. In order to do so, we use the equation 1, so the cells and particles are paired. To avoid a memory overflow, only those cells containing particles must be stored in memory.

When updating the particles information, it must be borne in mind that the data are not generally stored in memory in a sorted way. This has a negative impact in the performance of subsequent accesses, as multiple memory requests are needed to obtain all the information. Even the particles contained in the same cell (and so with the same hash code) may be disaggregated.

One of most used methods to improve performance consists on sorting computed data according to their corresponding hash code. This is because CUDA architecture strongly depends on memory information transfers, obtaining significant improvements of performance by minimizing the amount of accesses.

There are several sorting algorithms which are appropriate for parallel implementations, but in this case we will use the Radix Sort algorithm from NVIDIA's library [HB08] to sort the hash codes list, as it is one of the fastest algorithms for the problem posed.

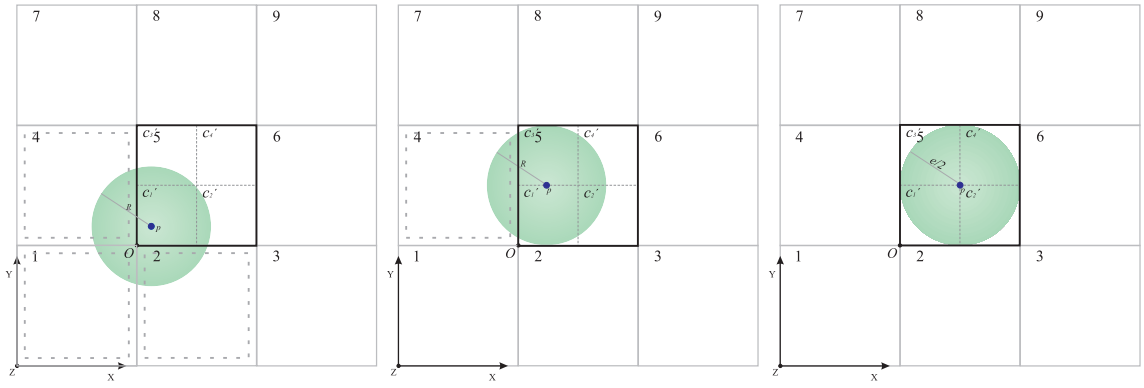
We have not selected the z–indexing technique because, in spite of giving better results in terms of proximity of the cells in memory, the cells are sorted using a different criterion which is costlier than ours in terms of computational costs. As the sorting process is a significant part of this algorithm, the advantages and disadvantages of z–indexing pretty much cancel each other out.

Now the first stage and the considerations related to optimizing performance are described, it is time to continue with the description of the second stage, in which we locate the particle inside each cell and the selection of adjacent cells where neighbouring particles are searched for.

5.2. Analysis–Allocation Stage

To locate the relative position of each particle inside the cell, called C, we use perpendicular secants located in the middle of each side, that is, $e/2$. Thus, eight octants of equal dimensions are obtained, called $C'_i \;\; \forall i = 1,\ldots,8$. We define a local origin on this breakdown, called O, occupying the minimum vertex of the cell, that is, $O = (x_{min}, y_{min}, z_{min})$, see Figure 3. It is remarkable that the original cell keeps its size for the rest of operations, the division made is "virtual" and it is only used to set the relative location of each analysed particle.

From each octant limit, we can distinguish the relative position of each particle. This process is carried out by compar-



(a) The general case. (b) The secant case. (c) The central case.

Figure 3: Relative particle positions inside the cell. The dashed contour indicates the adjacent cells where the neighbour particles must be searched.

ing the particle's coordinates and the secants position, that is

$$\left.\begin{array}{l} x_{jmin} \leq x_j < x_{jmin} + (e/2) \\ x_{jmin} + (e/2) \leq x_j \leq x_{jmin} + e \end{array}\right\} \forall j = 1,2,3$$

where $x_1 = x$, $x_2 = y$ and $x_3 = z$ are the particle coordinates.

Located the particle into the octant, we select the cell adjacent to search neighbour particles. From spatial division order, described in section 5.1, the cell's range to trace is fixed by:

$$\alpha_{x_j} R^{\delta(j-1)} \varepsilon_{x_j}^{(j-1)} \leq cell_{x_j} \leq \beta_{x_j} R^{\delta(j-1)} \varepsilon_{x_j}^{(j-1)}$$

where δ is the Dirac delta function, $cell_{x_j}$ is the cell increased in x_j direction and α_{x_j} and β_{x_j} are coefficients that satisfy:

$$\begin{array}{llll} \alpha_{x_j} = 0; & \beta_j = 1 & \text{if} & x_{jmin} \leq x_j < x_{jmin} + (e/2) \\ \alpha_{x_j} = -1; & \beta_j = 0 & \text{if} & x_{jmin} + (e/2) \leq x_j \leq x_{jmin} + e \end{array}$$

This way, the amount of adjacent cells to be tracked is significantly reduced, see Figure 3. Despite the extra calculations needed to locate the particles, this is balanced out with the reduction of cells where search is performed, as we will show in section 6.

The described process is applied to solve the most general case, see Figure 3a, nevertheless two more cases exist. In the first case, the particle is located in a secant plane, see Figure 3b Here, the adjacent cells to be searched are those in the intersection of the ones associated to adjacent octants. The second case has the particle located in the intersection of both secants planes, see Figure 3c. This is the simplest case, since we only need to search the neighbour particles into its own cell.

Once the subset of candidate cells is filtered, we continue to determine if the distance between the analysed particle and each particle contained in the subset is lower than the influence radius. If so, we allocate those particles as neighbours.

6. Results

Here we are going to develop a set of tests to show the improvements provided by our method. The first test will show the increase in time when looking for neighbouring particles in adjacent cells. In the second one we will measure execution times for different values of cell size, both for the standard implementation and our proposal. Both tests will be made taking into account increasing numbers of particles, from 1000 to 200000

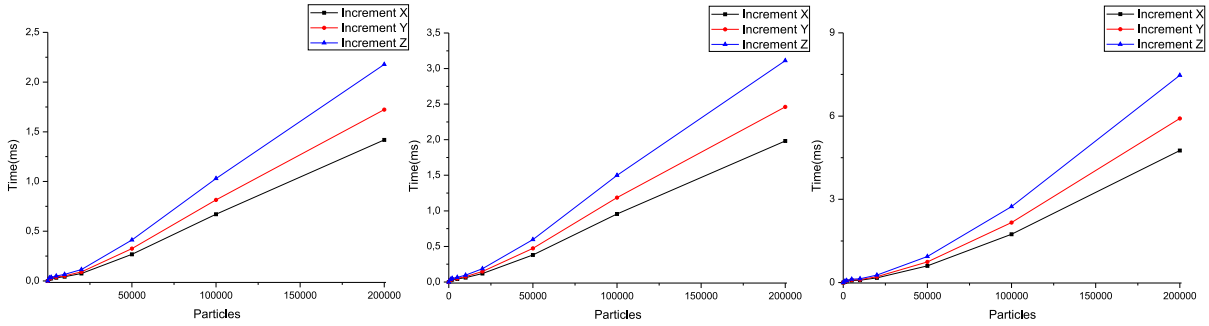
Using this range of the numbers of particles, the neighbouring particles that we are obtained are shown in Table 1. These values are related with the influence radius and it should be the same for all cell size.

To do these tests we have implemented a particle system simulation in CUDA, where particles interact with each other by means of a force limited by the influence radius that are inversely proportional to the distance between themselves. In addition, the particles are subject to the standard gravity force. For temporal integration, we use the second-order Euler Method. Our GPU is an NVIDIA GTX780. This method has similar characteristics to the one developed by Green [Gre10]. However, the CUDA Samples code is not designed for performance testing, so the results would be different. Nevertheless, it's possible to implement this method over the samples code in order to easily check the improvements in any system that supports CUDA.

The parameters' values used in our tests are the following: particle mass $m = 3 \cdot 10^{-4} kg$, force proportionality constant $k = 3600 N/m$ and time step $\Delta t = 2 \cdot 10^{-4} s$. We will work with three different edge lengths: $2R$, $2.52R$ and $4R$, with R

	1000	2000	5000	10000	20000	50000	100000	200000
Neighbouring Particles	10	13	17	21	30	41	50	62

Table 1: Neighbouring particles associated to each particules number that are used in our implementation.



(a) Edge length $e = 2R$ (b) Edge length $e = 2.52R$ (c) Edge length $e = 4R$

Figure 4: Graphs of processing time to each direction increment.

as the influence radius. While Wroblewski et al. [WBK07] work with cell edges of up to $2R$ length, other methods could need a larger cell size; this also allows us to work with a smaller array of cells, which makes the sorting process faster.

6.1. Temporal Dependency in Adjacent Cell Searching

In this test, we will show the increase in execution time when looking for neighbouring particles in adjacent cells. First we measure the time taken to look for neighbours in the original cell. Next, we search in cells which are only adjacent in a single coordinate; first X, then Y and Z. We enforce the adjacent cells to contain the same number of particles in order to ensure the tests' coherence. The results obtained are shown in Figure 4

This test justifies the order of division and the hash function used, as the increase in execution time is linear, no matter which position is occupied by the adjacent cells.

6.2. Temporal Dependency of Cell Size

This test will show the times obtained by varying the cell size. We will compare the execution times between our proposal and the standard division model. This comparison will show the improvements obtained over the standard implementation, as can be seen in Figure 5

7. Conclusions

In this paper we have analyzed the main features regarding neighbor search through spatial division by cells. We have highlighted the most relevant items that affect the computational costs when CUDA architecture is used. From these items, we have developed a method to improve the efficiency of standard spatial division techniques. We have carried out a set of tests to show the improvements by our technique. From these results we can conclude that:

- The use of orthogonal division by clockwise rotation, together with a non–normalized hash function, highly diminishes memory data dispersion. Consequently, the processing time is decreased. We have deducted this conclusion from the experiments whose results are shown in Figure 4.
- We have used the relative particle position to improve the search into adjacent cells. Despite the added operations, we have improved efficiency because the processing time is decreased. Comparatively, the improvement is in the range of 85% and 115%. These improvements are shown in Figure 5.
- From the implemented dynamical model, we have shown the best results are obtained when the regarded cells have a size whose edge is equal to double of the influence radius. This conclusion is backed up by the results shown in the Figure 5.

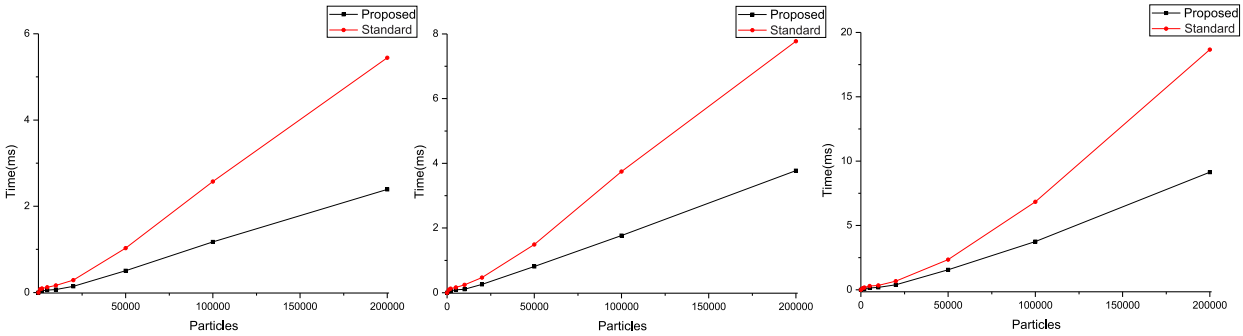
8. Acknowledgment

This research has been supported by the MeGUS project (TIN2013-46928-C3-3-R) of the Spanish Ministry of Science and Innovation.

References

[Akl14] AKL S. G.: *Parallel sorting algorithms*. Academic press, 2014. 3

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517. 1, 2



(a) Edge length $e = 2R$ (b) Edge length $e = 2.52R$ (c) Edge length $e = 4R$

Figure 5: Graphs of processing time of the standard division technique and the proposed method.

[Cay12] CAYTON L.: Accelerating nearest neighbor search on manycore systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International* (2012), IEEE, pp. 402–413. 2

[Cud12] CUDA C.: Programming guide. *NVIDIA Corporation, July* (2012). 2, 3

[DAM13] DOMÍNGUEZ J., A.J. C., M. G.-G.: Optimization strategies for {CPU} and {GPU} implementations of a smoothed particle hydrodynamics method. *Computer Physics Communications 184*, 3 (2013), 617–627. 3

[FSYA07] FUNG W. W., SHAM I., YUAN G., AAMODT T. M.: Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (2007), IEEE Computer Society, pp. 407–420. 3

[FWZS11] FAN W., WANG B., ZHOU J., SUN J.: Parallel spatial hashing for collision detection of deformable surfaces. In *Computer-Aided Design and Computer Graphics (CAD/Graphics), 2011 12th International Conference on* (2011), IEEE, pp. 288–295. 2, 5

[GDB08] GARCÍA V., DEBREUVE E., BARLAUD M.: Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on* (2008), IEEE, pp. 1–6. 1, 2

[GDNB10] GARCÍA V., DEBREUVE E., NIELSEN F., BARLAUD M.: K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *ICIP* (2010), pp. 3757–3760. 1

[Gre10] GREEN S.: Particle simulation using cuda. *NVIDIA whitepaper* (2010). 2, 3, 6

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2010), Eurographics Association, pp. 55–64. 3

[HB08] HOBEROCK J., BELL N.: Thrust, 2008. 5

[HSK07] HARADA T., SEIICHI K. Y. K.: Smoothed particle hydrodynamics on gpus. *Computer Graphics International* (May 2007), 63–70. 2

[Hwu11] HWU W.-M. W.: *GPU Computing Gems Jade Edition*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011. 3

[IABT11] IHMSEN M., AKINCI N., BECKER M., TESCHNER M.: A parallel SPH implementation on multi-core cpus. *Comput. Graph. Forum 30*, 1 (2011), 99–112. 1, 2, 3

[Kno09] KNOWLES P.: Gpgpu based particle system simulation. *School of Computer Science and Information Technology RMIT University Melbourne, Australia 12*, 04 (2009), 55–58. 2

[KZN08] KUMAR N., ZHANG L., NAYAR S.: What is a good nearest neighbors algorithm for finding similar patches in images? In *Computer Vision–ECCV 2008*. Springer, 2008, pp. 364–378. 2

[LCT14] LI X., CAI W., TURNER S. J.: Efficient neighbor searching for agent-based simulation on gpu. In *Distributed Simulation and Real Time Applications (DS-RT), 2014 IEEE/ACM 18th International Symposium on* (2014), IEEE, pp. 87–96. 4

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)* (2006), vol. 25, ACM, pp. 579–588. 2

[LOS10] LEISCHNER N., OSIPOV V., SANDERS P.: Gpu sample sort. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 1–10. 3

[MG10] MERRILL D. G., GRIMSHAW A. S.: Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (2010), ACM, pp. 545–546. 3

[Mic12] MICIKEVICIUS P.: Gpu performance analysis and optimization. In *GPU Technology Conference* (2012). 3

[MPG03] MÜLLER M. T. B. H. M., POMERANETS D., GROSS M.: *Optimized spatial hashing for collision detection of deformable objects*. Tech. rep., Technical report, Computer Graphics Laboratory, ETH Zurich, Switzerland, 2003. 2

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), Eurographics Association, pp. 41–50. 2

[PSHL10] PETERS H., SCHULZ-HILDEBRANDT O., LUTTENBERGER N.: Fast in-place sorting with cuda based on bitonic sort. In *Parallel Processing and Applied Mathematics*. Springer, 2010, pp. 403–410. 3

[RBA08] ROŻEN T., BORYCZKO K., ALDA W.: Gpu bucket sort algorithm with applications to nearest-neighbour search. *Journal of WSCG 16* (2008). 3

[RBG*12] RUSTICO E., BILOTTA G., GALLO G., HERAULT A., DEL NEGRO C.: Smoothed particle hydrodynamics simulations

on multi-gpu systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on* (2012), IEEE, pp. 384–391. 2

[Ros13] ROSEN P.: A visual approach to investigating shared and global memory behavior of CUDA kernels. *Comput. Graph. Forum 32*, 3 (2013), 161–170. 3

[SGS10] STONE J. E., GOHARA D., SHI G.: Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering 12*, 1-3 (2010), 66–73. 2

[SHG09] SATISH N., HARRIS M., GARLAND M.: Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), IEEE, pp. 1–10. 3

[THM*03] TESCHNER M., HEIDELBERGER B., MUELLER M., POMERANETS D., GROSS B.: Optimized spatial hashing for collision detection of deformable objects. In *Proceeding Vision, Modeling, Visualization VMV'03* (Nov 2003), pp. 47–54. 2

[VBC08] VICCIONE G., BOVOLIN V., CARRATELLI E. P.: Defining and optimizing algorithms for neighbouring particle identification in sph fluid simulations. *International Journal for Numerical Methods in Fluids 58*, 6 (2008), 625–638. 2

[WBK07] WRÓBLEWSKI P., BORYCZKO K., KOPEÆ M.: Sph-a comparison of neighbor search methods based on constant number of neighbors and constant cut-off radius. *TASK Quart 11* (2007), 275–285. 2, 3, 4, 7