Int J Parallel Prog (2012) 40:313–330
DOI 10.1007/s10766-011-0184-3

Nearest Neighbor Searches on the GPU

A Massively Parallel Approach for Dynamic Point Clouds

Pedro Leite · João Marcelo Teixeira · Thiago Farias · Bernardo Reis · Veronica Teichrieb · Judith Kelner

Received: 6 February 2011 / Accepted: 20 August 2011 / Published online: 12 October 2011

Abstract We introduce a GPU grid-based data structure for massively parallel nearest neighbor searches for dynamic point clouds. The implementation provides real-time performance and it is executed on GPU, both grid construction and nearest neighbors (approximate or exact) searches. This minimizes the memory transfer between device and system memories, improving overall performance. The proposed algorithm may be used across different applications with static and dynamic scenarios. Moreover, our data structure supports three-dimensional point clouds and given its dynamic nature, the user can change the data structure's parameters at runtime. The same applies to the number of neighbors to be found. Performance comparisons were made against previous works, endorsing the benefits of our solution. Finally, we were able to develop a real-time Point-Based Rendering application for validation of the data structure. Its drawbacks and data distribution's impact on performance were analysed and some directions for further investigation are given.

P. Leite · J. M. Teixeira · T. Farias (✉) · B. Reis · V. Teichrieb · J. Kelner
Computer Science Center, Federal University of Pernambuco, Av. Professor Moraes Rego S/N, DINE, 1st floor, Cidade Universitaria, Recife, Pernambuco, CEP 50670-901, Brazil
e-mail: tsmcf@cin.ufpe.br

P. Leite
e-mail: pjsl@cin.ufpe.br

J. M. Teixeira
e-mail: jmxnt@cin.ufpe.br

B. Reis
e-mail: bfrs@cin.ufpe.br

V. Teichrieb
e-mail: vt@cin.ufpe.br

J. Kelner
e-mail: jk@cin.ufpe.br

Keywords Nearest neighbor query · Massive parallel programming · KNN · ANN · Point-Based Rendering

1 Introduction

Spatial subdivision is a well-known technique for improving performance used in a variety of applications. There are many data structures that handle spatial subdivision efficiently. However, some data structures are well suited for specific problems. As an example, ray tracers need to do a lot of ray-triangle intersection tests in order to perform triangle culling. And kd-trees can provide a fast approach to solve such tests as shown in [1]. In addition, kd-trees can also be used for nearest neighbor search in photon mapping [2] or in point cloud modeling [3]. Other data structures such as octrees have been used for smoke and water simulation [4] and appearance preserving [5], while collision detection of deformable objects can be implemented with spatial hashes [6] or representative triangles [7]. Finally, different types of spatial subdivision data structures ease the task of culling non visible objects from a scene [8].

Another common problem solved with an efficient spatial subdivision is the nearest neighbors, which consists on finding the closest neighbors to an input query. The neighbors and the input query can be described as a location, a car, a restaurant, etc. Nearest neighbor searches have their roots on the post-office problem, in which residences (input query) are assigned to their nearest post office (neighbor) and were first described by Donald Knuth [9]. Nowadays, a vast number of problems rely on nearest neighbor searches, including pattern classification [10], mobile information systems [11], implicit surfaces definition [12,13], simplification of point-sampled surfaces [14], nearest photon queries in photon mapping [2], nearest neighbor search in point cloud modeling and particle-based fluid simulation [3,15], normal estimation [16] and finite element modeling [17], among others. In this context, solving problems that rely on nearest neighbors searches implies in improving spatial subdivision techniques.

Nearest neighbor searches in dynamic point clouds comprise constructing a data structure (at each given timestamp) to hold the input data set as well as realize a lot of sorting, when searching for neighbors of a given set of query points. If the data structure does not intelligently subdivide the input data set in order to minimize such sorting, searches become slow and real-time criteria (a maximum processing time of 33 ms, including data structure construction) cannot be met. In other words, the main concern of this work is to address real-time massively parallel nearest neighbor searches in dynamic point clouds. This can be achieved through a grid data structure for both KNN (K nearest neighbors) and ANN (approximate nearest neighbors) searches and by exploiting the inherent parallel processing power of modern GPUs. In addition, a Point-Based Rendering (PBR) application is proposed, so the data structure can be tested and evaluated.

The remainder of this paper is organized as follows. In the next section we discuss some related work regarding KNN and ANN searches on CPU and GPU. Section 3 introduces the proposed solution to massively parallel nearest neighbor searches in dynamic point clouds. Both CPU and GPU implementations are highlighted. The obtained results are compared with each other as well as with previous works. Finally,

a discussion justifies the advantages of a GPU approach. Section 4 introduces concepts related to PBR and evaluates the proposed solution using PBR as case study. Section 5 draws a conclusion for the presented work, exploiting the contributions made and pointing some future works to it.

2 Related Work

This section discusses KNN and ANN searches. The former is a generalization of the post-office problem, where the solution to the problem involves the K nearest post-offices instead of the nearest one only, while the later can tolerate minimum errors, i.e. there can be post-offices that are farther than the KNN, but can be found faster and are as good as the others for the problem solution.

Previous works on nearest neighbor searches are mainly implemented in CPU and are optimized for cache efficiency, minimal disk access, among others. In large point clouds data sets, Sankaranarayanan et al. [18] proposed a search by identifying a region in space (called locality) that contains all of the KNNs of a collection of points. Once the best possible locality is built, each point searches only the locality for the correct set of K nearest neighbors. The data structure used is a disk-based quadtree variant (e.g., see [19]) and can handle large data sets. Even though their algorithm improves previous techniques, it does not deliver real-time performance, since its results show that 6.22 s are needed for computing the neighborhood of size K = 8 for each point in a point cloud with 37K points, on a Quad Intel Xeon with 1 GB of RAM and SCSI hard disks.

Aiming to speed up Sankaranarayanan's work, Connor and Kumar [20] solve the construction of KNN graphs for point clouds in parallel, benefiting from multiple processors CPUs and handling large data sets. The KNN graph construction is done in three phases. First, a parallel distribution sort is used, then the sorted array is split into p chunks (p is the number of processors to be used) in a way that each processor can compute the initial approximate nearest neighbors for one chunk independently. Finally, a recursive function refines the approximate nearest neighbor solution to an exact answer. Albeit being an efficient algorithm for KNN searches by taking advantage of multiple processors, their results are impractical for real-time applications. Exemplifying, on Dual Quad-core 2.66 GHz Intel Xeon CPUs, with 4 GB of DDR RAM, for each point in a data set with about 27K points, the KNN graph construction for K = 1 takes 40 ms for points stored as 64-bit integers, which delivers a maximum frame rate of 25 fps.

Some problems that rely on nearest neighbors to be solved do not require the exact closest neighbors. An approximation can be used and as a consequence, ANN searches are performed instead of KNN ones. Such approximation carries an inherent error, however ANN generally improves performance. The ANN effort is justified when the error can be minimized without compromising the problem's solution. Lin and Yang [21] proposed an index structure (ANN-tree) to solve ANN searches with a high accuracy. The ANN-tree greatly improves the accuracy of searches, but the results shown do not provide time measures and also no more than one neighbor is computed for the data sets. Furthermore, the normal estimation proposed by

Mitra and Nguyen [16], for example, uses Arya et al.'s [22] ANN library for neighborhood evaluation.

Recently, GPU implementations have improved performance by utilizing the massive parallel architecture of graphics cards. In [23], the authors propose a brute force KNN GPU implementation that overcomes the ANN CPU implementation by over 100 times (Pentium 43.4 GHz with 2 GB of DDR2 memory versus NVIDIA GeForce 8800 GTX with 768 MB). The brute force KNN algorithm sorts the elements by distance to the query point and takes the first k elements as solution. Although this solution surpasses the ones in which query oriented data structures were used, the brute force method is still a naive implementation. For example, NN searches have been used in GPU data structures for dynamic sampling and rendering of algebraic point set surfaces [24]. The results reported by Guennebaud et al. show a frame rate of 45 fps for dynamic sampling and rendering of a model with approximately 23K vertices (including NN searches), using a Core2 Duo 2.4 GHz CPU and an NVIDIA GeForce 8800 GTX. Furthermore, kd-trees have been constructed in real-time on the GPU for ray tracing, point cloud modeling and photon mapping [25]. The implemented point cloud modeling in [25] performs 127K KNN searches for a neighborhood of $K = 10$ in 14 ms, on an Intel Xeon 3.7 GHz CPU with an NVIDIA GeForce 8800 ULTRA, which proves that a GPU approach is not only feasible for KNN searches, but practically mandatory.

To overcome the necessity of fast nearest neighbor searches, our data structure provides a real-time performance for its construction and both KNN (except for large data sets) and ANN searches, while supporting dynamic input data.

3 Methodology

In this section we propose a grid-based GPU data structure for dynamic point cloud data sets capable of achieving real-time performance for both approximate and exact nearest neighbor searches [26]. The grid-based data structure is similar to Fig. 1 and it will be further used in a PBR application. It handles three-dimensional points and is subdivided in NNN parts. For instance, Fig. 1 is subdivided in $3 \times 3 \times 3$ parts.

In sequence, the construction of the data structure and the algorithm for both ANN and KNN searches are highlighted, some implementation details are given and results are discussed.

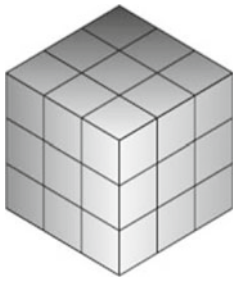

Fig. 1 Grid-based cube, subdivided in a $3 \times 3 \times 3$ manner

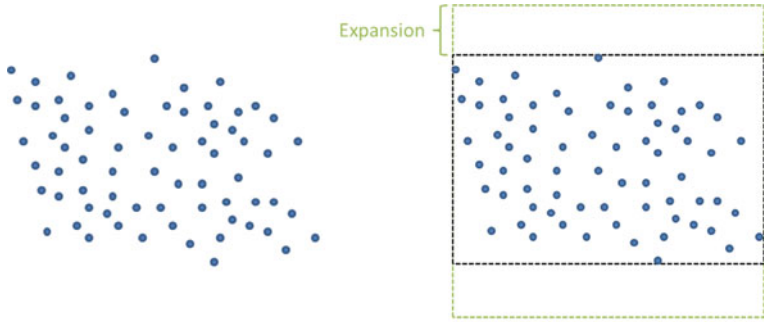

Fig. 2 Given an input point cloud, an AABB expansion is required for an even grid subdivision

3.1 GPU Grid Construction

The GPU data structure construction algorithm proceeds as follows. For a given point set P (Fig. 2a), the minimum and maximum coordinates are found through a parallel reduction [27]. The reduction algorithm basically elects a candidate (minimum or maximum) from a given set (in this case the point set P). This way, the AABB (axis aligned bounding box) of P is computed. The AABB sides are then set to its maximum size, i.e. it is "rounded up" to a cube (our data structure currently handles three dimensions), while keeping its center (see Fig. 2b).

From now on, we have set our "working" space and it will be subdivided into cells (the number of subdivisions can be defined by the user interactively). In sequence, we guarantee that each point $p \in P$ remains within the cube and can be indexed into its cells following a simple injective hashing function,

$$hash(p) = c_p.x \times s^2 + c_p.y \times s + c_p.z, \tag{1}$$

executed entirely on the GPU. The variable c_p has three dimensions and holds the spatial location of the cell, with regard to the cube's number of subdivisions (represented by the variable s). It is computed as

$$c_p \leftarrow cell(p) = \frac{(p - lowerBound)}{(upperBound - lowerBound)} \times s, \tag{2}$$

with the variables $upperBound$ and $lowerBound$ representing the maximum and minimum values of the "rounded up" AABB, respectively, and s being the cube's number of subdivisions. Figure 3 presents a given point set hashed into a grid with four subdivisions per dimension.

Since we have all the points $p \in P$ hashed through the $hash(p)$ function, we sort them in a way that points from the same cell will be placed sequentially in memory. Finally, we reorder the input point set accordingly to the point's hash value and in sequence we find each cell size, i.e. the amount of points that are within that cell, as at the bottom of Fig. 4. It is noteworthy that except for the reduction, all stages of the data

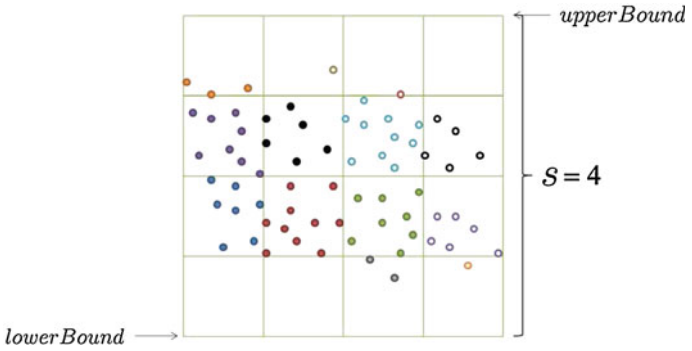

Fig. 3 Point set hashed into a grid with four subdivisions per dimension

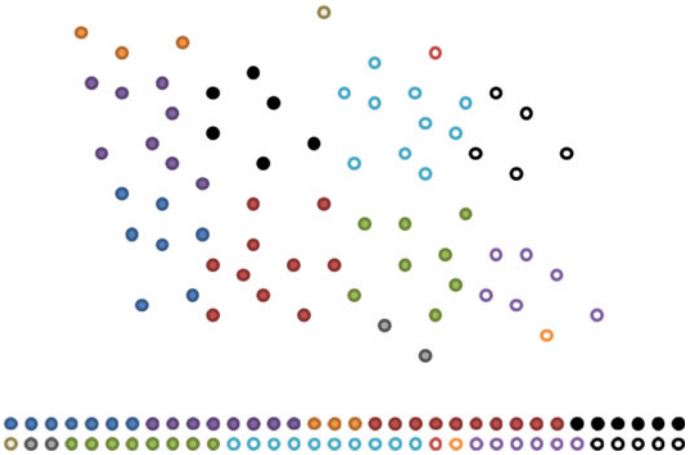

Fig. 4 Point set aligned sequentially in memory, regarding cell position

structure construction are done on the GPU. The reduction is an exception because part of it is done on the CPU. Section 3.3 will further discuss the implementation details.

3.2 NN Search

In this section, we describe how our nearest neighbor searches (both ANN and KNN) are performed. The ANN search is implemented using range search. Initially, for each point $p_i \in P$, with i being the point index, their hash is computed and a search radius $r_i = 1$ is dictated in parallel. Starting from the cell that contains p_i, we increment r_i until we find the radius value r_k which guarantees that at least K points are present. This search radius is measured in cells, i.e. if a point is within a cell $c_k = \{5, 4, 6\}$ and we find a radius $r_k = 3$, then it is necessary to visit all the cells within the cube with edges $c_1 = \{2, 1, 9\}$, $c_2 = \{2, 7, 9\}$, $c_3 = \{2, 7, 3\}$, $c_4 = \{2, 1, 3\}$, $c_5 = \{8, 1, 9\}$, $c_6 = \{8, 7, 9\}$, $c_7 = \{8, 1, 3\}$, $c_8 = \{8, 7, 3\}$.

As shown in Algorithm 1, which is executed in parallel, we visit at least 27 cells, because $r_i = 1$, and we will end up having a cube with side equals to 3 cells. For all the cells visited, a point histogram is computed. When the size of the histogram is greater than K, we have the approximately K closest points, so we stop incrementing the radius. Finally we get the K closest points, based on their distance from p_i.

Algorithm 1 Approximate Nearest Neighbor Search Algorithm

Require: $k \geq 1$
Ensure: k approximate points to p are returned
1: $r_k \leftarrow 0$ {The initial range radius value is set to zero}
2: $c_p \leftarrow hash(p)$ {The cell position is computed based on the point position}
3: $p_{hist} \leftarrow size(c_p)$
4: **repeat**
5: $r_k \leftarrow r_k + 1$
6: **for** each cell c, c is r_k cells away from c_p **do**
7: $p_{hist} \leftarrow p_{hist} + size(c)$
8: **end for**
9: **until** $p_{hist} < k$
10: **return** k closest points to p within radius r_k

A similar approach is used for KNN searches (Algorithm 2). As with ANN, we find the value of r_k, but we continue expanding two more times. Since we consider cells as the radius magnitude, in the ANN algorithm we take the cell center as reference. In KNN, the point p_i can be near to an edge, so there could be points closer to it that are in cells not covered by r_k found by the ANN algorithm. This way we avoid false neighbors, so the expansion is straightforward.

Algorithm 2 Exact Nearest Neighbor Search Algorithm

Require: $k \geq 1$
Ensure: k closest points to p are returned
1: $r_k \leftarrow 0$ {The initial range radius value is set to zero}
2: $c_p \leftarrow hash(p)$ {The cell position is computed based on the point position}
3: $p_{hist} \leftarrow size(c_p)$
4: $t \leftarrow 0$
5: **repeat**
6: $r_k \leftarrow r_k + 1$
7: **for** each cell c, c is r_k cells away from c_p **do**
8: $p_{hist} \leftarrow p_{hist} + size(c)$
9: **end for**
10: **if** $p_{hist} \geq r_k$ **then**
11: $t \leftarrow t + 1$
12: **end if**
13: **until** $p_{hist} < k$ **or** $t \leq 2$
14: **return** k closest points to p within radius r_k

The difference between ANN and KNN is demonstrated in Fig. 5 using a two-dimensional problem as an example. Given a query point (in red) and a number of K = 10 neighbors to find, there are two point sets representing its neighbors. The approximate

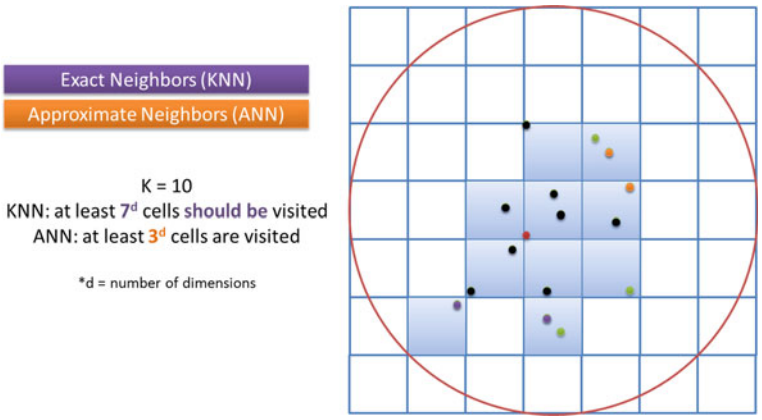

Fig. 5 Difference between ANN and KNN searches. More cells are visited with KNN searches

neighbors can be found using at least one expansion. They are represented by the black points plus the other two orange ones, and at least $3^2 = 9$ cells are visited. Moreover, some error is introduced with this approximation. The purple points are closer than the orange ones and thus are found only when executing the KNN algorithm. In this case, at least $7^2 = 49$ cells (in two-dimensional problems) should be visited. The green points represent points that are neither approximate nor exact neighbors of the query point.

3.3 Implementation Details

We implemented the grid data structure and NN searches previously explained using NVIDIA's Compute Unified Device Architecture (CUDA) framework [28,29]. Previous to CUDA, researchers have used shading languages for exploiting the GPU. The main disadvantages were knowledge requirement about graphics pipeline and the lack of arbitrary memory access (gather and scatter). CUDA solves such problems by providing a high level C-like programming language to abstract NVIDIA GPUs from series 8 or higher.

CUDA programming model is overviewed as follows. The code that runs on the GPU is called kernel. Each kernel has a grid configuration up to two dimensions defining how many blocks will logically run in parallel. A block is composed by threads and has up to three dimensions. Threads run physically in parallel and execute the kernel code.

We store several memory buffers for the data structure construction besides the input data set. For each point, we need to store four 32-bit integers representing the hash value, its index on the original buffer and temporary values for the sorting. In addition, we need two buffers with size equal to $s \times s \times s$ unsigned 32-bit integers to hold information about each cell start index and cell size (how many points the cell contains), with s being the number of grid subdivisions defined by the user. Currently, the value of s can range from 1 to 197, due to memory (for the additional buffers) and

performance (regarding time to compute cell start index and size) restrictions. Also, it has a fixed value for all the dimensions, so cells have the same spatial distribution, facilitating the nearest neighbor searches. After allocating these buffers, the grid is constructed.

As noted on Sect. 3.1, before constructing the grid, a parallel reduction is performed. We modified the reduction code provided by the CUDA SDK to support non power of two sized data sets and to compute both minimum and maximum values from the input data set. The reduction done on GPU generates per-block minimum and maximum values. These values are copied to host memory, i.e. the memory available to the system, and a CPU reduction computes the AABB bounds. The AABB is then "rounded up" to a cube similarly to the illustration presented in Fig. 2b.

Points are hashed into cells and a parallel key-value radix sort is applied; the key and value are the hash and the point index, respectively. This way points from a given cell will be placed sequentially in memory (as in Fig. 4). Finally, each cell size is computed and the original input data set buffer is reordered based on the index information after the parallel sorting. Since reads from contiguous memory addresses speed up performance due to CUDA's memory management model, the reordering is mandatory.

Nearest neighbor searches are performed as follows. For each query point, its hash is computed (faster than reading from global memory), so we know which cell it is within, and then we discover through range search the radius value r_k which guarantees that at least K points are present in surrounding cells. In addition, we count the number of non-empty cells on that range. The total number of non-empty cells counted is computed by using a parallel scan [30] and a cell buffer is allocated to hold their indices. In sequence, all the cells within the radius value r_k are re-visited, except that instead of counting cells or computing a search radius, non-empty cells have their indices stored in the cell buffer. This will be helpful in the neighbor search's sorting stage, since a reduced number of cells will be visited. As an example, for the KNN search in Fig. 5, instead of 49 cells only 10 are visited (the ones marked in blue).

In order to exploit the memory model of the GPU, when performing the neighbor search's sorting phase we declare ten variables that will hold the closest distances from the query point, although the algorithm's interest is the K-th distance. More than 10 variables, for our kernel code, would compromise the general availability of registers, thus limiting the amount of threads executing physically in parallel. On one hand, if $K \leq 10$, then all the points will be visited exactly once, and the K-th distance will be found. On the other hand, if $K > 10$, points will be visited more than once. For example, if $K = 17$, two iterations are performed, selecting the first ten closest points, then the seven remaining ones. This redundancy often pays off avoiding the use of shared memory or local memory, because we are declaring variables that are used as registers, and thus no latency or shared memory bank conflicts would happen. Finally, after computing the distance d_k of the K-th neighbor, we visit once more all the points and store those which have a distance to the query point smaller or equal to d_k. Points are stored in global memory in a way that threads will write in consecutive memory locations.

Table 1 Data sets used in tests

Data set	Size (points)
E.T.	17,345
Bunny	35,947
Dinosaur	53,504
Sphere	163,842
Hand	327,323
Dragon	423,565

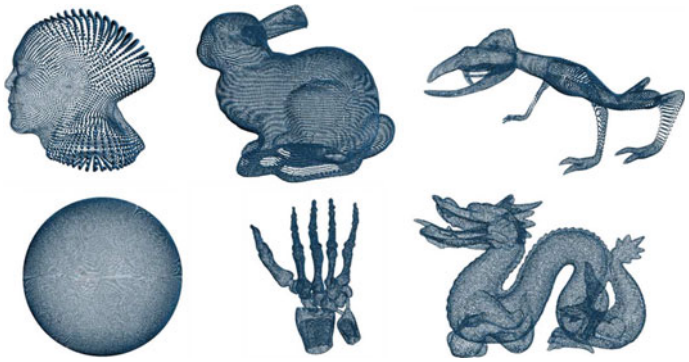

Fig. 6 Visual representation of the E.T. (*top left*), Bunny (*top center*), Dinosaur (*top right*), Sphere (*bottom left*), Hand (*bottom center*) and Dragon (*bottom right*) data sets

3.4 Results and Discussion

The described algorithm has been tested on an Intel Core i7 920 2.67 GHz with 4 GB of RAM and an NVIDIA GTX 480 graphics card. The system was running Windows 7 Professional 64-bit. Since the GTX 480 consists of two GPUs, the results shown in this section are related to the use of both cards in tandem.

Table 1 shows the different data sets used in the tests. It is noteworthy that the sphere data set is the only one with its points evenly distributed in space. The remaining data sets mostly have some areas with high density of points. A visual representation of the data sets is shown in Fig. 6. In addition, the E.T. data set is from a 3D OBJ online repository (no longer available), the sphere data set was generated with a 3D modeling tool, the Hand data set is from the Clemson University, while the remaining ones are from Stanford University.

For all data sets depicted in Table 1, grid construction, ANN searches and KNN searches numeric results were gathered and are shown in Table 2. The grid construction follows the number of subdivisions in the same table, and searches consist in finding, for each point in the data set, its $K = 10$ neighbors (approximate or exact). The standard deviation of the measured times shows that the values do not varies so much based on the mean, i.e. the queries perform almost in constant time. Regarding the number of subdivisions, it is important to note that they were carefully chosen (empirically), not for the grid construction, but to improve performance on the searches. The time results

Table 2 Timing results for grid construction, and nearest neighbor searches on GPU

Data set	# of subdiv.	Grid construction (avg/stdev) in ms	ANN (avg/stdev) in ms	KNN (avg/stdev) in ms
E.T.	35	0.72/0.08	3.54/0.26	7.62/0.21
Bunny	69	1.15/0.11	3.28/0.71	6.01/0.52
Dinosaur	127	1.39/0.06	4.96/1.28	12.35/1.66
Sphere	127	2.79/0.03	6.65/0.20	20.21/0.23
Hand	192	5.20/0.04	17.92/0.22	58.19/0.41
Dragon	192	6.36/0.04	21.15/0.32	63.44/0.34

Table 3 Performance comparison with previous CPU implementations

Work	Data set size	K	Search time	CPU	GPU
Sankaranarayanan et al.	37,000	8	6.220 s	Quad Intel Xeon	–
Connor and Kumar	27,000	1	0.040 s	Dual Quad-core 2.66 GHz Intel Xeon	–
This work	54,000	8	0.012 s (SD = 0.0004 s)	Intel Core i7 920 2.67 GHz	NVIDIA GTX 480

are shown in milliseconds and were gathered according to the following criteria: each grid construction is repeated N times according to equation 3 [31].

$$N = \left(\frac{2Z_\alpha \sigma}{\varepsilon}\right)^2 \tag{3}$$

The number of samples (N) is determined by the confidence level (α) using a normal distribution with parameters $\mu = 0$ and $\sigma = 1$(Z_α in Eq. 3), the standard deviation of the samples (σ) and the expected error (ε). The confidence level used in ours experiments was 95%, and the expected error was fixed in 0.1μs. The same rule applies to searches.

Previous results [20] have shown that performing similar KNN searches (K = 1) in CPU (using 8 cores in parallel) on a point cloud of about 56K points yields a result of 70 ms. The proposed algorithm performs both KNN search (K = 10) and grid construction in 23 ms for a point cloud of 163K points in GPU (almost three times more points). Furthering the comparisons, Table 3 presents some interesting results.

For all points in the data sets, one KNN search is performed. The number of points is depicted in the column "Data set size" and the number of neighbors found is represented by the column "K". Search time is given in seconds and the two first works

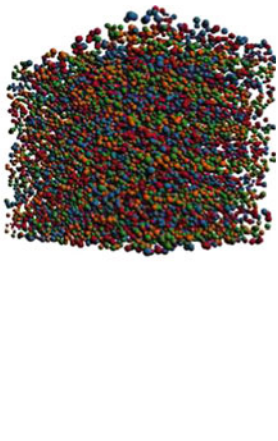

Fig. 7 Two clusters of points within the same grid data structure

are implemented in CPU. In addition, the standard deviation of the search time was provided in order to show that the search is performed almost in constant time.

The proposed algorithm surpasses both previous CPU implementations. When compared to [18], it presents a speedup of approximately 518x, while performing 16,504 more searches. In addition, when compared to [20], it performs 26,504 more searches with a speedup of approximately 3x, while finding 7 more neighbors. In summary, this work performs more searches and finds more neighbors in less time.

An example of a GPU implementation for range search is the Anderson et al.'s [32] neighbor list generation for molecular dynamics simulation. It reports that generating a list of all N neighbors (approximately 30 neighbors are found) on a data set with 50K points takes 25 ms. The proposed algorithm also obtains a result of 25 ms for the same conditions. However it is guaranteed that exactly 30 neighbors (exact or approximate ones) are found for each point, differently from Anderson et al.'s work.

However, although being fast, the data structure has its drawbacks. For example, higher performance is achieved with an even point distribution. The ANN search time for the Bunny data set is faster than for the E.T. one due to the fact that the points in the Bunny data set are evenly distributed, while in the E.T. one there are many high density areas, decreasing performance in the sorting stage.

Another impact of point distribution can be exemplified with clusters. Figure 7 shows two clusters of points, one with 9 and other with 15,000. If it is required for each point an ANN or KNN search of up to 8 neighbors, the searches will be as fast as they should (according to the algorithm presented). However, if $K \geq 9$ neighbors are required, each point in the smaller cluster will need to visit too many cells (most of them are empty) in order to reach the larger cluster, because locally there are not enough neighbors. Moreover, in such cases a careful choice for the subdivision number must be taken.

Figure 8 shows a performance comparison regarding the subdivision choice and ANN searches for the aforementioned data set on the Core i7/GTX 480 machine. The blue line represents timing results for ANN searches when both clusters are within the same data structure, while the green line represents timing results for ANN searches

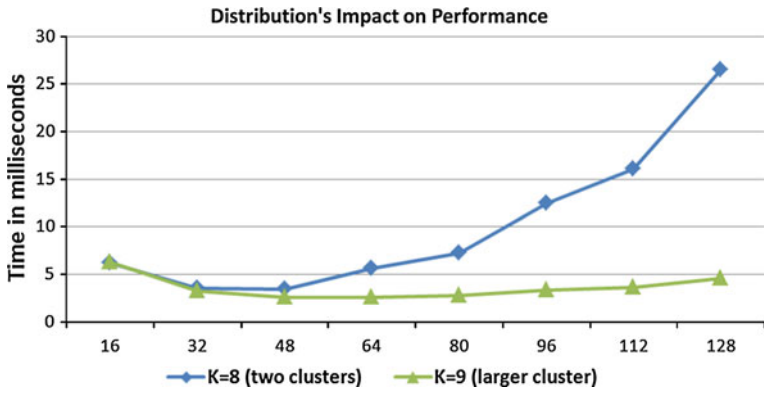

Fig. 8 Time necessary to find K neighbors with one and two clusters. With two clusters, memory access is incoherent and a sudden decrease in performance happens with the growth of subdivisions on the grid

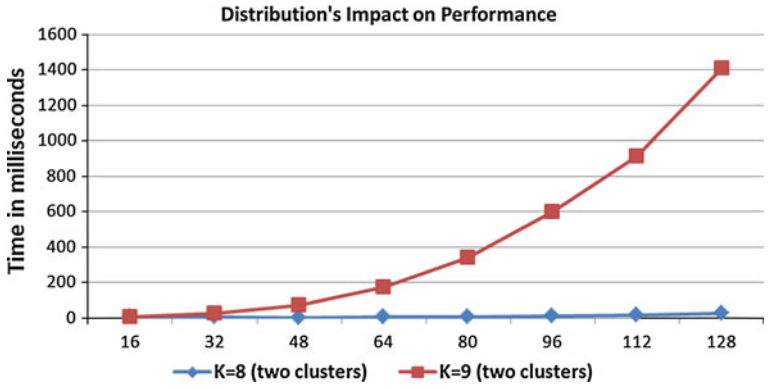

Fig. 9 Impact on ANN search performance in a clustered data set

only in the larger cluster, i.e. the cluster with 9 points is not present on the data structure, and thus is ignored.

With K = 9, the performance is drastically affected, as shown in Fig. 9. This is due to incoherent memory access, as well as a high number of cells visited by the smaller cluster.

Finally, the proposed algorithm needs the user to specify the number of subdivisions. Figure 10 reports timing results for different subdivision choices for the Dinosaur model. The red line indicates the time spent on grid construction, the green line the time spent with searches (K = 8), and the blue line represents the total time. From Fig. 7 to Fig. 10 the y-axis means execution time in milliseconds and the x-axis means the number of subdivisions of the data structure.

If a poor choice is made, for instance, the number of subdivisions is equal to one, grid construction will be as fast as possible, but NN searches will require $O(N^2)$ time, which is unacceptable in dynamic scenarios. The input data set distribution and its size are the main concerns for a better subdivision choice. Finally, the presented exact

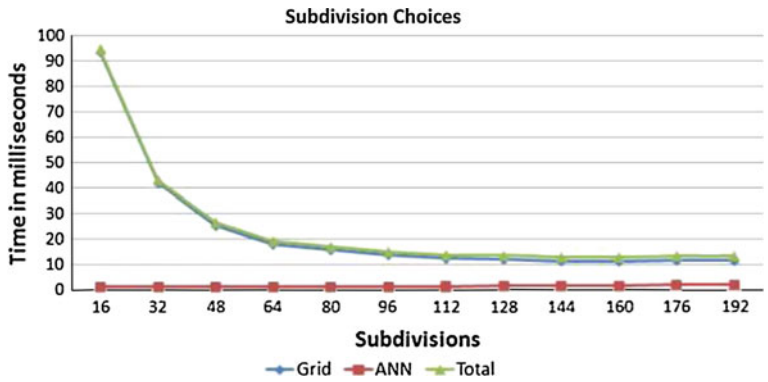

Fig. 10 Timing results for different subdivision choices on the Dinosaur data set

nearest neighbor search approach is able to deliver real-time performance for small data sets. If large data sets are required, some improvements are needed.

4 Point-Based Rendering

Regarding three-dimensional (3D) models, their representation is often based on triangle meshes, since they are easy to manipulate and render. However, a reliable surface representation is only achieved with a great number of triangles, and when this number surpasses the number of pixels on the screen, the rendering process becomes slow [33]. Furthermore, representation of volumetric objects, wave simulation, smoke simulation, etc. is commonly done through a particle system. The correct rendering of objects represented as particle systems can be done through a high-quality GPU PBR framework [34]. In short, PBR takes a point cloud with no explicit connectivity and reconstructs the underlying surface defined by such cloud.

In order to evaluate the performance of the proposed data structure and approximate nearest neighbor search, we developed a PBR application [26]. Regarding the data structure, its implementation was slightly modified to support reordering of more than one buffer. This is necessary, because the rendering framework needs to access color and normal information of each point. In addition, the PBR framework does not require information of a point's neighbors, just the distances to the farthest one. This avoids the need for storing neighbors in global memory and increases overall performance.

Rendering follows the three-pass algorithm proposed by Botsch and Kobbelt [35] and Guennebaud and Paulin [36] and is entirely implemented on GPU, using OpenGL and GLSL. Basically, each point is rendered as a circular disc (splat) in three passes: visibility, accumulation and shading. The visibility pass consists in rendering all splats perspectively [37] in order to store their depth values on the depth buffer. In addition, back face culling is performed. This way, each pixel will determine the z value of a visible splat. The accumulation pass is similar to the visibility pass regarding the perspective projection. However, it does not write to the depth buffer. Instead, each splat

Fig. 11 Bunny data set rendered using the Point-Based Rendering technique. The visibility pass (*left*) stores the depth value, the accumulation pass (*center*) interpolates weighted normals from neighboring splats and the shading pass (*right*) computes the average normal vector and performs lighting

is projected with a depth offset towards the viewer. This guarantees that overlapping splats will have fragments with different z values accepted on the depth-test. Moreover, this pass accumulates interpolated normal vectors for overlapping splats by using a blending function. The accumulated normal vectors will then be normalized on the following pass. Finally, the remaining pass (shading pass) will compute the average normal vector for each pixel and perform lighting. The passes are illustrated in Fig. 11.

Although both PBR and ANN searches are performed by the same GPU, thus limiting its processing availability, the results obtained show that the proposed algorithm and rendering framework are feasible for real-time applications.

4.1 Experimental Results

The PBR application has been tested on the same machine presented in Sect. 3.4 and only one GPU was exploited.

Regarding the data sets discussed in Table 1 and Table 2, they were rendered with the aforementioned PBR framework. The simulation of a dynamic point cloud was introduced by a wave function (implemented in CUDA), so that at each frame points' position changes. The number of subdivisions used is described in Table 4 and $\mathrm{K} = 6$ approximate neighbors are found, since this is a good (empirical) amount in order to reconstruct the surface of the aforementioned data sets. Finally, the window size was set to a high definition resolution of $720p$, i.e. 1280×720 pixels. The results presented in Table 4 comprise both dynamic and static point clouds and consist of the average frame rate for rendering, the time for constructing the data structure, performing searches and rendering the point cloud. Also the time spent only with rendering is stated.

Analyzing Table 4, most data sets presented real-time performance for static and dynamic point clouds, exempting Hand and Dragon data sets. This occurs due to the fact that to only one GPU is responsible for rendering, data structure construction and searches, with a high amount of points. When all these times are added, real-time is almost impossible. One probable solution could be splitting the workload among more than one GPU.

Figure 12 shows the results obtained with static and dynamic point clouds for all data sets previously discussed. All data sets (in this figure) are presented in a static-dynamic order.

Table 4 Point-Based Rendering execution times and frame rate for both static and dynamic scenes

Data set	# of subdivisions	Frame rate (FPS)		Data structure (ms)		Rendering (ms)	
		Static	Dynamic	Static	Dynamic	Static	Dynamic
E.T.	50	85	78	5.05	6.84	6.71	5.98
Bunny	69	75	65	5.02	7.04	8.31	8.34
Dinosaur	127	63	58	9.09	10.86	6.78	6.38
Sphere	127	47	44	11.11	13.15	10.16	9.57
Hand	192	26	20	25.64	35.71	12.82	14.29
Dragon	192	21	18	32.25	38.46	15.36	17.09

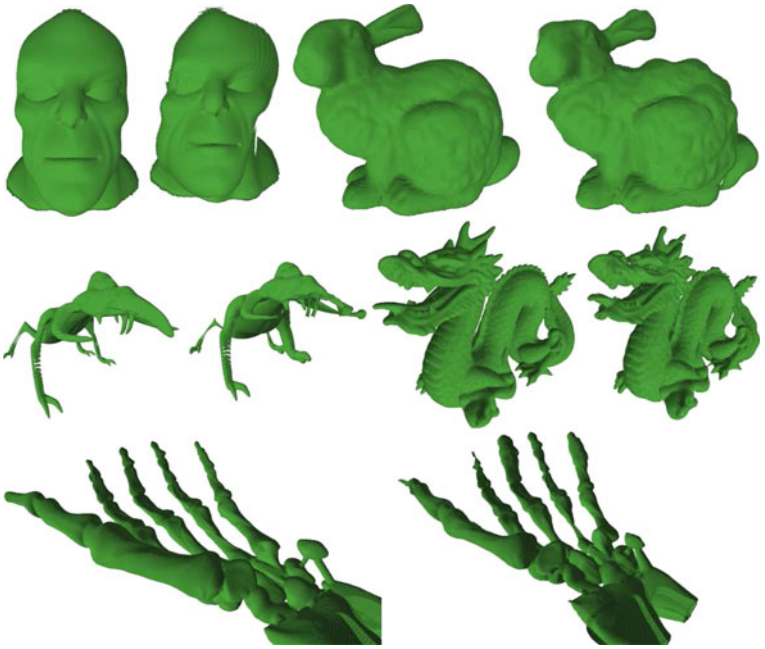

Fig. 12 E.T. (*top left*), Bunny (*top right*), Dinosaur (*middle left*), Dragon (*middle right*) and Hand (*bottom*) data sets rendered using the Point-Based Rendering technique

5 Conclusion

A GPU-based massively parallel nearest neighbor search algorithm for dynamic point cloud data sets has been developed. It achieves real-time performance for both approximate and exact nearest neighbor search.

The algorithm builds a spatial grid and exploits GPU features to improve performance in the search stage. The adopted solution outperforms many current off-line CPU implementations as well as being among the fastest existing GPU based ones, mainly when dealing with dynamic scenarios.

The achieved performance is highly dependent on factors such as the number of clusters as well as the selected number of subdivisions. The latter must be chosen according to the data set distribution and its size.

A PBR application has been evaluated to demonstrate the proposed strategies in handling real-time massively parallel nearest neighbor searches [38]. Note nonetheless, that the data structure used relies on the user specifying the subdivision number. Despite the lack of capability for handling large data sets (millions of points), due to memory availability of current GPUs, this may be overcome through the use of stream computing.

In addition, it is desirable for the grid to adapt according to the input data set distribution. In this context, a tradeoff between the time necessary for its construction and the time for performing the NN search may be established. As a result, user intervention is no longer needed. Similarly, KNN search performance may also benefit from this grid adaptation. Finally, the proposed approach will be evaluated in different contexts such as real-time physics simulation, point cloud modeling, among others.

Acknowledgments The authors would like to thank CNPq for the financial support received and the referees for their relevant contributions.

References

1. Pharr, M., Humphreys, G.: Physically Based Rendering: From Theory to Implementation. Morgan Kaufmann Publishers Inc., San Francisco (2004)
2. Jensen, H.W.: Realistic Image Synthesis Using Photon Mapping. A. K. Peters, Natick (2001)
3. Alexa, M., Gross, M., Pauly, M., Pfister, H.: Stamminger, Marc., Zwicker, Matthias.: Point-based computer graphics. In: SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes, p. 7. ACM, New York (2004)
4. Losasso, F., Gibou, F., Fedkiw, R.: Simulating water and smoke with an octree data structure. In: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, pp. 457–462. ACM, New York (2004)
5. Lacoste, J., Boubekeur, T., Jobard, B., Schlick, C.: Appearance preserving octree-textures. In: GRAPHITE '07: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia, pp. 87–93. ACM, New York (2007)
6. Teschner M., Heidelberger B., Müller, M., Pomeranets, Danat., Gross M.: Optimized spatial hashing for collision detection of deformable objects. In: Vision, Modeling, Visualization (VMV), pp. 47–54 (2003)
7. Curtis, S., Tamstorf, R., Manocha, D.: Fast collision detection for deformable models using representative-triangles. In: I3D '08: Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games, pp. 61–69. ACM, New York (2008)
8. Pantazopoulos, I., Tzafestas, S.: Occlusion culling algorithms: a comprehensive survey. J. Intell. Robot. Syst. **35**(2), 123–156 (2002)
9. Knuth, D.: Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, Massachusetts (1998)
10. Cover, T., Hart, P.: Nearest neighbor pattern classification. Inf. Theory IEEE Trans. **13**(1), 21–27 (1967)
11. Safar, M.: K nearest neighbor search in navigation systems. Mob. Inf. Syst. **1**(3), 207–224 (2005)
12. Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, C.T.: Point set surfaces. In: Visualization Conference, pp. 21–28. IEEE (2001)
13. Levin, D.: Mesh-independent surface interpolation. In: Brunnett, G., Hamann, B., Müller, H. (eds.) Geometric Modeling for Scientific Visualization, pp. 37–49. Springer (2003)
14. Pauly, M., Gross, M., Kobbelt, L.P.: Efficient simplification of point-sampled surfaces. In: VIS '02: Proceedings of the Conference on Visualization '02, pp. 163–170. IEEE Computer Society, Washington (2002)

15. Adams, B., Pauly, M., Keiser, R., Guibas, L.J.: Adaptively sampled particle fluids. In: ACM Transactions on Graphics (SIGGRAPH '07 papers), San Diego, CA, vol. 26, issue 3, art. no. 48. ACM Press, New York (2007)
16. Mitra, N.J., Nguyen, A.: Estimating surface normals in noisy point cloud data. In: SCG '03: Proceedings of the Nineteenth Annual Symposium on Computational Geometry, pp. 322–328. ACM, New York (2003)
17. Clarenz, U., Rumpf, M., Telea, A.: Finite elements on point based surfaces. In: Symposium of Point Based Graphics 2004 (2004)
18. Sankaranarayanan, J., Samet, H., Varshney, A.: A fast all nearest neighbor algorithm for applications involving large point-clouds. Comput. Graph. **31**(2), 157–174 (2007)
19. Samet, H.: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). Morgan Kaufmann, San Francisco (2005)
20. Connor, M., Kumar, P.: Parallel construction of k-nearest neighbor graphs for point clouds. In: Proceedings of Volume and Point-Based Graphics, pp. 25–32. IEEE VGTC (2008)
21. Lin, K.-I., Yang, C.: The ANN-tree: an index for efficient approximate nearest neighbor search. In: DASFAA '01: Proceedings of the 7th International Conference on Database Systems for Advanced Applications, pp. 174–181. IEEE Computer Society, Washington (2001)
22. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Angela, Y.W.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. J. ACM **45**(6), 891–923 (1998)
23. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using gpu. In: CVPR Workshop on Computer Vision on GPU, Anchorage (2008)
24. Guennebaud, G., Germann, M., Gross, M.: Dynamic sampling and rendering of algebraic point set surfaces. Comput. Graph. Forum **27**(2), 653–662 (2008)
25. Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time KD-tree construction on graphics hardware. In: SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers, pp. 1–11. ACM, New York (2008)
26. Leite, P.J.S., Teixeira, J.M.X.N., de Farias, T.S.M.C., Teichrieb, V., Kelner, J.: Massively parallel nearest neighbor queries for dynamic point clouds on the GPU. In: Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD '09, pp. 19–25. IEEE Computer Society, Washington (2009)
27. Mark H.: Optimizing parallel reduction in CUDA. http://www.nvidia.com/content/cudazone/cuda_sdk/Data-Parallel_Algorithms.html (2009)
28. NVIDIA.: Compute unified device architecture programming guide. http://www.nvidia.com/cuda (2009)
29. Farias, T.S.M.C., TeixeiraJoao Marcelo, N.X., Leite Pedro, J.S., Almeida, G.F., Almeida Mozart, W.S., Teichrieb, V., Kelner, J.: High performance computing: cuda as a supporting technology for next generation augmented reality applications. RITA **16**(1), 26 (2009)
30. Shubhabrata S., Mark H., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pp. 97–106. Eurographics Association, Aire-la-Ville (2007)
31. Eng, J.: Sample size estimation: how many individuals should be studied? Radiology **227**(2), 309–313 (2003)
32. Anderson, J.A., Lorenz, C.D., Travesset, A.: General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Comput. Phys. **227**(10), 5342–5359 (2008)
33. Levoy, M., Whitted, T.: The use of points as a display primitive. Technical Report 85-022, University of North Carolina at Chapel Hill (1985)
34. Botsch, M., Hornung, A., Zwicker, M., Kobbelt, L.: High-quality surface splatting on today's GPUs. In: Proceedings of the Eurographics Symposium on Point-Based Graphics, pp. 17–24 (2005)
35. Botsch, M., Kobbelt, L.: High-quality point-based rendering on modern gpus. In: Pac. Conf. Comput. Graph. Appl., pp. 335–343 (2003)
36. Guennebaud, G., Paulin, M.: Efficient screen space approach for hardware accelerated surfel rendering. In: Vision, Modeling and Visualization, pp. 485–495. IEEE Signal Processing Society (2003)
37. Botsch, M., Spernat, M., Kobbelt, L.: Phong splatting. In: Proceedings of Symposium on Point-based Graphics, pp. 25–32 (2004)
38. de Farias, T.S.M.C., Almeida, M.W.S., Teixeira Joao, M.X.N., Teichrieb, V., Kelner, J.: A high performance massively parallel approach for real time deformable body physics simulation. In: Comput. Archit. High Perform. Comput. Symp., pp. 45–52 (2008)