Institut für Numerische Simulation
Rheinische Friedrich-Wilhelms-Universität Bonn
Wegelerstraße 6 • 53115 Bonn • Germany
phone +49 228 73-3427 • fax +49 228 73-7527
www.ins.uni-bonn.de

P. Diehl, M. A. Schweitzer

# Efficient Neighbor Search for Particle Methods on GPUs

INS Preprint No. 1405

June 2014

# Efficient Neighbor Search for Particle Methods on GPUs

Patrick Diehl[1] and Marc Alexander Schweitzer[2]

**Abstract**

In this paper we present an efficient and general sorting-based approach for the neighbor search on GPUs. Finding neighbors of a particle is a common task in particle methods and has a significant impact on the overall computational effort–especially in dynamics simulations. We extend a space-filling curve algorithm presented in [13] for its usage on GPUs with the parallel computing model Compute Unified Device Architecture (CUDA). To evaluate our implementation, we consider the respective execution time of our GPU search algorithm, for the most common assemblies of particles: a regular grid, uniformly distributed random points and cluster points in 2 and 3 dimensions. The measured computational time is compared with the theoretical time complexity of the extended algorithm and the computational time of its reference single-core implementation. The presented results show a speed up of factor of 4 comparing the GPU and CPU run times.

**Key words:** Neighbor search, GPU, meshfree methods and particle methods

## 1 Introduction

Particle methods are widely used today, especially in fluid dynamics. One of the earliest particle methods, smoothed-particle hydrodynamics (SPH) [9], was already developed in 1977 to simulate astrophysics problems and is applied today in many others areas of application. Other prominent examples of

Institute for Numerical Simulation, Wegelerstr. 6, 53115, Bonn, Germany diehl@ins.uni-bonn.de · Institute for Numerical Simulation, Wegelerstr. 6, 53115, Bonn, Germany schweitz@ins.uni-bonn.de

particle methods, are e.g. Molecular Dynamics (MD) [10] and Peridynamics (PD) [21], which are typically used in material science problems.

A common task in particle methods is the search for neighboring particles, since the discretization depends on the interactions of particles inside an interaction sphere or so-called horizon. Finding the neighborhood of each particle accounts for a large part of the overall computational time and due to the often weak convergence properties of particle methods, very large particle clouds need to be employed in high fidelity simulations. To overcome the computational complexity $\mathcal{O}(n^2)$ of the naïve neighbor search, where we compute distances $||x_i - x_j||$ of each particle $x_i$ to all other particles $x_j$, a number of approaches and numerical libraries exist.

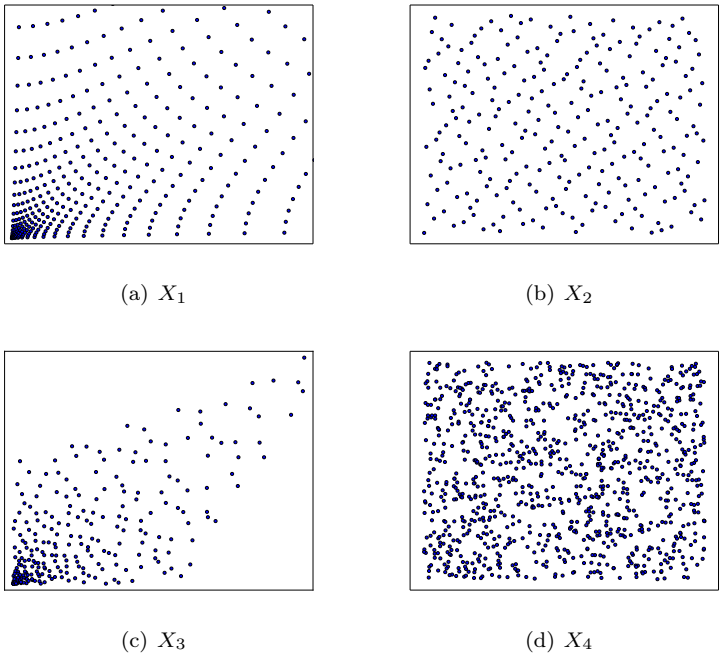
One of the most renowned single-core libraries is ANN [14]. However, to deal with very large particle clouds, necessary for accurate simulations results today, the particle neighborhoods need to be computed in parallel. Here, also a number of libraries exist for shared-memory [2, 15, 24] and distributed-memory [13, 17] parallel computers. On Graphics Processing Units (GPU), however, this task is essentially an open question. Some algorithms [7, 11] and the "knn Cuda" [8] library is available for General Purpose Computation on GPU(GGPU). This library supports GGPU using the Compute Unified Device Architecture (CUDA), but is limited to 65535 particles, which renders the library essentially useless in our setting, where we are concerned with particle numbers of $\mathcal{O}(10^6)$.

In this paper we present a generic GPU accelerated nearest neighbor search algorithm, which can be utilized in any particle method. Our algorithm rests upon the approach presented in [13]. According to [13] the algorithm is designed for better efficiency on multi-core machines, due to low memory usage and good cache efficiency. For GGPU low memory usage and good cache efficiency are important indicators of the performance on the GPU. These benefits of the algorithm and its complexity $\mathcal{O}(\lceil \frac{n}{p} \rceil m \log(m))$, where $n$ denotes the number of particles, $p$ the number of threads and $m$ the number of neighbors, offers a good initial situation for an implementation on the GPU.

The remainder of this paper is structured as follows: In section 2 we shortly introduce our reference particle method, peridynamics (PD), to describe the problem setting considered in this study. Note however that our approach is not restricted to PD simulations, but rather is applicable in general particle methods. In section 3, we introduce the fundamental algorithm and discuss challenges in its extension to GPUs. Then, we present the measured run time of the GPU accelerated algorithm and compare these with the run time of the CPU implementation of the algorithm and our GPU accelerated implementation in section 4. We conclude with the comparison of these implementations and a suggestion for use cases of the different implementations in section 5.

## 2 Neighbors in particle methods

Figure 1 shows sketches of reference particle clouds considered in this paper. Initial particle configurations in simulations are often uniformly distributed. Thus, we consider a regular particle arrangement and a uniformly distributed, but irregularly spaced particle cloud. These initial particle clouds are redistributed at later time steps of the simulation by the respective particle method. Thus, we may encounter particle clouds with large variations in the particle density. Throughout this paper we consider these reference cases, see also Figure 1, for the evaluation of our search algorithm.



(a) $X_1$ (b) $X_2$

(c) $X_3$ (d) $X_4$

**Fig. 1** Sketch of reference particle clouds: graded point cloud of a regular grid $X_1$ (a), points of the Halton sequence $X_2$ (b), graded Halton sequence $X_3$ (c) and uniformly distributed random points $X_4$ (d). Both point clouds were graded with $g(x) = ||x||^2 x$.

As a reference particle method we consider peridynamics (PD) [16,21,22], which is a non-local generalization of continuum mechanics, with a focus on discontinuous solutions as they arise in fracture mechanics. The principle of this theory is, that particles interact with other particles at a finite distance by exchanging forces – very similar to SPH and MD.

We present briefly the essential ingredients of the simple bond-based PD, which are important for the neighbor search algorithm. For further details see [16, 21, 22]. The PD equation of motion for the displacement field $u$ is given by the integral equation

$$\varrho(x)\ddot{u}(x,t) = \int_{B_\delta(x)} f(u(x',t) - u(x,t), x' - x)dx' + b(x,t), \qquad (1)$$

with mass density $\varrho(x)$, $f$ as the kernel function modeling the interaction of particles $x$ and $x'$ in the initial reference configuration and $b(x,t)$ denotes the external force. Here $B_\delta(x)$ denotes the radial interaction zone, with the cut-off radius $\delta$ for the internal forces, see Figure 2. As usual the deformed material configuration and instantaneous particle cloud $Y = \{y_i\}$ are obtained by $y_i(t) = x_i + u(x_i,t)$. Discretizing (1) in space by a collocation approach using

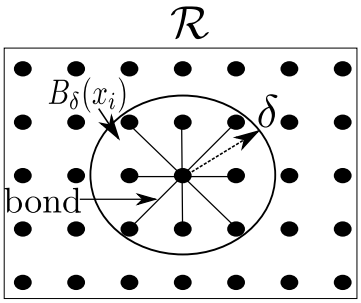



**Fig. 2** Shows the reference configuration $\mathcal{R}$ of the particle cloud $X := \{x_i | i = 1, \dots n\}$. All particles inside the interaction zone $B_\delta(x_i)$ of the particle $x_i$ are connect with bonds to exchange forces.

the particle cloud $X = \{x_i\}$ yields

$$\varrho(x^i)\ddot{u}(x_i,t) = \sum_{j \in \mathcal{F}_i} f(u(x_j,t) - u(x_i,t), x_j - x_i)\tilde{V}_j + b(x_i,t), \qquad (2)$$

with $\mathcal{F}_i = \{j \mid ||x_j - x_i|| \leq \delta,\, i \neq j\}$, $f$ as the kernel function and $V_j$ as the volume associated with the particle $x_j$. We determine $\mathcal{F}_i$ in the reference configuration $\mathcal{R}$, see Figure 2. All particles inside $B_\delta(x_i)$ are connected with an bond to $x_i$ and they exchange forces through the kernel function $f$, which depends on the stretch of these bonds. $\tilde{V}_j$ denotes the scaled volume of particle $x_j$ which is the intersection of $V_j \cap B_\delta(x_i)$. This set depends only on the initial positions of the point cloud and can be precomputed.

To prevent particle contact and overlap an additional force term $f_s$ is introduced in the discrete model [16]. An example for the force term $f_s$ is a "hard" potential, e.g. the repulsive part of the Lenard-Jones potential. The overall discrete PD model then reads as

$$\varrho(x_i)\ddot{u}(x_i,t) = \sum_{j\in\mathcal{F}_i} f_b(u(x_j,t)-u(x_i,t),x_j-x_i)\tilde{V}_j + \sum_{j\in\mathcal{F}^s_{i,t}} f_s(u(x_j,t)-u(x_i,t),x_j-x_i)V_j + b(x_i,t). \tag{3}$$

Thus, a second set of interacting particles

$$\mathcal{F}^s_{i,t} = \{j \mid ||y_j(t)-y_i(t)|| \leq \min(0.9||x_i-x_j||, 1.35r_i),\ i\neq j\}, \tag{4}$$

which employs that the instantaneous particle positions $y(t) = x + u(x,t)$ needs to be computed. This set denotes all particles which are around the particle $x_i$ with the radius $r_i$, due to deformation with respect to the instantaneous configuration and is at time $t = 0$ the empty set.

This set depends on the current time and must be updated in every time step during the simulation. Remember the computational complexity $\mathcal{O}(n^2)$ of the naïve approach, which would dominate the overall computational effort of most particle methods. Thus, this step must be realized in an extremely efficient way to allow for the simulation of large particle clouds.

## 3 Neighbor search

In this section we shortly review the fundamental algorithm presented in [13] which will serve as the basis for our massive parallel implementation using CUDA. The basic idea is to sort the points with respect to a space–filling curve (SFC). This is a very successful and widely used approach for sorting multidimensional data with respect to topological information [1, 3, 4, 12, 19, 20, 25]. Such a sorting strategy is essentially realized via the following four steps (see also Algorithm 1).

1. Generating keys: The multidimensional point cloud $X$ is transformed with $T : \mathbb{R}^d \to \mathbb{R}$, which is the inverse of a SFC. Thus, a one dimensional key $k_i = T(x_i)$ is assigned to every particle $x_i \in X$.
2. Sorting keys: These one-dimensional keys $k_i$ can easily be sorted and induce a respective ordering of the particles $x_i \in X$.
3. Range scan: Moreover it is easy to identify "neighbor keys" in one dimension to obtain a good initial guess $\tilde{N}(x_i)$ for the geometric neighbors in multidimensions of a particular particle $x_i$.
4. Geometrical validation: Finally, we need to check if all geometrical neighbors $x_j \in X$ of particle $x_i \in X$ are already found.

In the following we review this approach in more detail and discuss its implementation on a GPU. In a SFC-method, a multidimensional point $x \in \mathbb{R}^d$ is transformed to a (large) integer value $T(x) = k \in \mathbb{N}$, which can then be easily sorted. Thus, the transformation $T : \mathbb{R}^d \to \mathbb{N}$ is essentially influencing the

**Data**: Morton order compare operator $\leq_M$ and Point cloud $P$
**Result**: m-nearest neighbors $\forall p_i \in P$
$P \leftarrow \text{ParallelSort}(P, \leq_M)$;
**for** $p_i \in P$ **do**
  $A_i \leftarrow nn^m(p_i, \{p_{i-m}, \ldots, p_{i+m}\})$;
  **if** $p_i^{\lceil rad(A_i) \rceil} < p_{i+m}$ **then**
    $u \leftarrow i$
  **else**
    $I \leftarrow 1$; **while** $p_i^{\lceil rad(A_i) \rceil} < p_{i+2^i}$ **do** $++I$; $u \leftarrow \min(i + 2^I, n)$;
  **end**
  **if** $p_i^{-\lceil rad(A_i) \rceil} > p_{i-m}$ **then**
    $l \leftarrow i$;
  **else**
    $I \leftarrow 1$; **while** $p_i^{-\lceil rad(A_i) \rceil} > p_{i-2^i}$ **do** $++I$; $l \leftarrow \min(i - 2^I, n)$;
  **end**
  **if** $l \neq u$ **then**
    CSEARCH$(p_i, l, u)$;
  **end**
**end**

**Algorithm 1:** Origin algorithm described in [13]. The steps highlighted in blue are discussed in this section, because these need to be adapted for the implementation on the GPU.

quality of the resulting ordering of the data. There exists many different space filling curves, e.g. the Hilbert curve, the Peano curve, the Lebesgue curve, also referred to as Morton order, which can in principle be employed. One early application of the Morton order is efficient range searching of multidimensional data in dynamically balanced trees [24]. Using the Morton order, the transformation $T : \mathbb{R}^d \to \mathbb{N}$ is computationally cheap, but provides some sub–optimal locality. To clarify this let us consider the generation and comparison of keys for the Morton order in the following.

$$x = \begin{pmatrix} 0 \\ 3 \end{pmatrix}_{b=10} = \begin{pmatrix} 000 \\ 0111 \end{pmatrix}_{b=2} \xrightarrow{T(x):\mathbb{R}^2 \to \mathbb{N}} (000101)_{b=2} \tag{5}$$

Equation (5) shows the generation of the key $T(x)$ for the point $x = (0, 3)^T$. First the coordinates of the point in base 10 are converted to base 2 and then mapped to the scalar key $(x_0 y_0 x_1 y_1 x_2 y_2)_{b=2}$ in base 2 by bit-interleaving. Equation (6) shows the comparison of two points $x_1 = (0, 3)^T$ and $x_2 = (1, 2)^T$ using the Morton order compare operator $\leq_M$.

$$T(x_1) = T\begin{pmatrix} 0 \\ 3 \end{pmatrix} = (000101)_{b=2} \leq (000110)_{b=2} = T\begin{pmatrix} 1 \\ 2 \end{pmatrix} = T(x_2) \tag{6}$$

The computational effort of the comparison $T(x_1) \leq T(x_2)$ is small, since it involves only 2 operations: the exclusive-or operation and the most signifi-

cant bit (MSB) operation delivers the result. Thus, the largest cost of the computational work associated with the Morton order compare operator

$$x_1 \leq_M x_2 \Leftrightarrow T(x_1) \leq T(x_2) \tag{7}$$

is associated with the respective computation of the respective keys $T(x_1)$ and $T(x_2)$.

Applying the Morton order compare operator to the nodes of a uniform $4 \times 4$ grid yields to the ordering depicted in Figure 3. From this plot we find

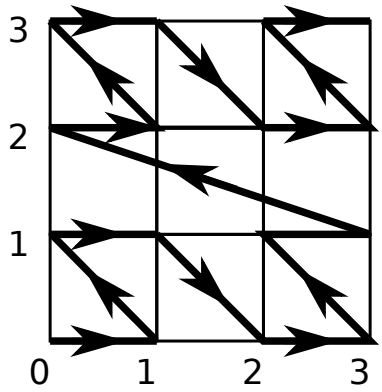



**Fig. 3** The resulting space–filling curve with the Morton Order compare operator $\leq_M$ on a uniform grid with $4 \times 4$ points. Because of its shape, the curve is also called Z–Curve.

that we assign "adjacent" keys to the points $(3,1)^T$ and $(0,2)^T$. Thus, these points are "neighbors" with respect to the Morton order, but not with respect to their euclidean distance. Other space-filling curves, e.g. the Hilbert curve, may provide a better data locality but the computational costs associated with the respective key generation or compare operator may be much larger.

We obtain an initial estimate of the m-nearest neighbors of a particle $x_i$ by selecting $m/2$ particles $x_j$ with the largest keys $k_j = T(x_j) \leq k_i = T(x_i)$ and $m/2$ particles with the smallest keys $k_j \geq k_i = T(x_i)$ and collect these keys in the set $\hat{N}(k_i)$. This initial guess $\tilde{N}(x_i) = \{x_j | T(x_j) = k_j, k_j \in \hat{N}(k_i)\}$ then needs to be validated, i.e. we need to check if $\hat{N}(k_i)$ in fact contains the particles $x_j$ which are closest to $x_i$.

To this end, the Morton order divides a $d$-dimensional unit cell recursively in $2^d$ sub cells. In each sub cell there exist two keys $q = T(x)$ and $p = T(x')$, so that all particles $x_l$ with $q \leq T(x_l) < p$ are included in this sub cell. To validate the initial guess $\tilde{N}(x_i)$ , we have to check if the smallest key $q$ and the largest key $p$ are included in $\hat{N}(k_i)$. Otherwise the range of the set $\hat{N}(k_i)$ needs to be extended. Algorithm 2 describes the extension of the set using some geometrical information about the sub cells. Note that we use $\upsilon = 4$ as

suggested in [13] and have not yet optimized this parameter for the GPU.

The described transformation $T$ is so far suitable for point clouds with integer values as coordinates. With the extensions, described in [5, 13], the Morton Order $\leq_M$ compare operator handles floating point values and multidimensional input data with $d > 3$.

Thus, in summary we need to provide the operations to generate the keys and handle recursiveness in Algorithm 2 on the GPU. To provide the exclusive-or operation and the most significant bit (MSB) operation, because on a GPU, these two operation are not available in the standard library, we need to define a new data type for the IEEE 754 representation of float and double, to access the exponent and the mantissa of the respective double or floating values of the particles.

Another issue for the implementation on a GPU are the recursive calls of Algorithm 2, because the programming model on the GPU does not support "real" recursiveness in kernel functions. The kernel function is launched on the device with a specified grid of blocks. Then, a device function is called, recursively, within a kernel function. Thus, some recursive calls need to be implemented with the help of device functions and the grid of blocks need to be adjusted, because of the Single Instruction Multiple Threads (SIMT) architecture, depending on the employed GPU.

```
CSearch(point p_i, int l, int h) ;
if h-l < υ then
    A_i = nn^m(p_i, {p_{i-m}, ..., p_{i-m}}) ;
    return
end
b = (h+l)/2 ;
A_i = nn^m(p_i, A_i ∪ p_b) ;
if dist(p_i,box(p_l,p_h)) ≥ rad(A_i) then
    return
end
if p_i < p_b then
    CSearch(p_i, l, b-1);
    if p_b < p_i^{⌈r(A_i)⌉} then
        CSearch(p_i, b+1, h);
    else
        CSearch(p_i, b+1, h);
    end
    if p_i^{-⌈r(A_i)⌉} < p_b then
        CSearch(p_i, l, b-1);
    end
end
```

**Algorithm 2:** Function *CSearch(...)* [13] extends the range of the set $\{x_{i-ck}, \ldots, x_{i+ck}\}$ with using some geometrical information about the generated sub cubes.

### *3.1 Parallel Sorting*

With the increasing popularity of GGPU many standard sorting algorithms are available for GPU [18, 23]. The library Thrust [26], a powerful library of parallel algorithms and data structures, was extended with CUDA support and is now integrated in the CUDA SDK. The Thrust library contains the data structure *vector*, to store the multidimensional input data, in our case the particle locations $x_j$, and an optimized parallel merge sort algorithm, which utilizes a user defined *StrictWeakOrdering comp* which we realize with the help of $\leq_M$, see Equation (7).

## 4 Results and Discussion

To evaluate the implementation of the extended algorithm the measured computational time is compared with its theoretical complexity and with the run time of the `C++` Library STANN [6], the reference implementation of our base algorithm [13].

In all presented experiments we choose $m = 168$ (if not stated otherwise) for 3D particle clouds which corresponds to the number of geometric neighbors $x_j$ of a particle $x_i$, i.e. $||x_j - x_i|| \leq \delta = 3a$, on a regular lattice with the lattice constant $a$. For 2D points clouds we choose $m = 24$, respectively. The number of threads $p$ per processor is determined by the available hardware. The Nvidia K20c we used throughout this paper contains a Kepler GK110 G chip set. The specification of the chip set describes 13 multiprocessors each with 2048 threads. In a optimal case all 13 multiprocessor with all threads can be executed in parallel. Thus, we use $p = 13 \cdot 2048 = 26624$.

First we compare the measured run times with the theoretical complexity $\mathcal{O}(\lceil \frac{n}{p} \rceil m \log(m))$ [13], where $n$ denotes the number of particles of the point cloud $X$, $p$ the number of threads and $m$ the number of neighbors. In Figure 4 we consider particle clouds $X$ from $n = 512$ up to $n = 2^{21}$, which corresponds to the memory limit of the GPU. From the depicted plots we can observe that our implementation shows the theoretical complexity with respect to $n$ for most of the considered point clouds $X$: nodes of a regular grid $X_0$, nodes of a graded grid $X_1$ (Figure 1(a)), Halton points $X_2$ (Figure 1(b)), graded Halton points $X_3$ (Figure 1(c)) and uniformly distributed random points $X_4$ (Figure 1(d)). For $X_0$-$X_3$ we find the expected linear asymptotic behavior. Only or the uniformly distributed random points $X_4$ we see, the influence of the recursive calls in Algorithm 2 in the serrations of the computational time up to $n = 2^{20}$. For small particle clouds $X_4$ the particle density around some particles is low and to find all $m$ neighbors the range of the initial guess $\tilde{N}(x_i)$ needs to be extended.

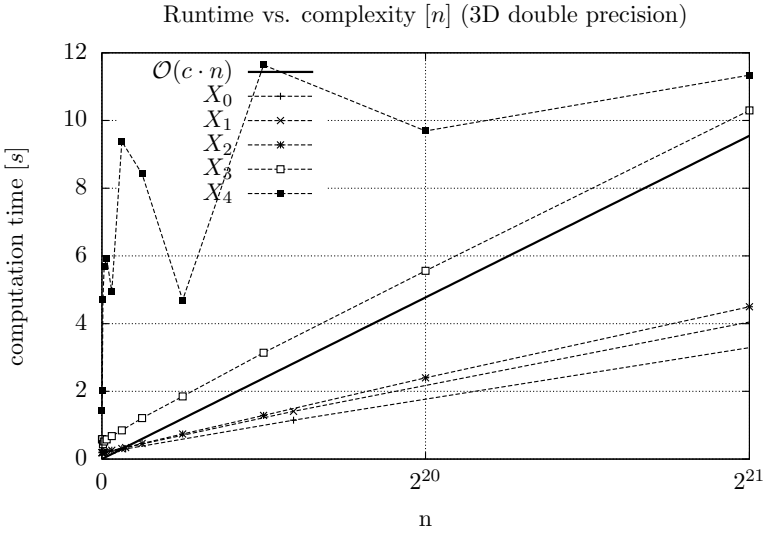



**Fig. 4** Comparison of the theoretical complexity $\mathcal{O}(c \cdot n)$ with the measured computational time on the GPU.

| | CPU | GPU |
|---|---|---|
| Compiler | g++ (4.6.3) | nvcc (release 5.5, V5.5.0) |
| Compiler options | -o3 | -arch=sm_35 |
| Hardware | Intel Xeon 5500 (1 Core) | NVIDIA Tesla K20c |

**Table 1** Build configuration of the libraries for the CPU and GPU.

To confirm this assertion we consider $m = 24$ up to $m = 830$ for the point clouds $X_1$-$X_4$ with $n = 9706$ and $n = 77672$ in three dimensions. From these plots given in Figure 5 we see for $X_1$-$X_3$ we attain the anticipated complexity with a very small constant. Only for $X_4$ we can again observe some oscillations. A more robust results is obtained with extending the initial estimate to the range $[-m, m]$ instead of $[-\frac{m}{2}, \frac{m}{2}]$. For the point cloud $X_4$ we see again that the range need to be extended.

Secondly the extended algorithm for the GPU is compared with the STANN library in the version 0.74 which does not support any parallel implementation for finding the $m$-nearest neighbors. The configurations shown in Table 1 were used to build the libraries. For the computation time on the GPU we measured the full work flow. This means that the measured computation time includes the copying process of the point cloud to the device and copying the resulting list of the $m$–nearest neighbors back to the host.
The neighborhoods for all point clouds $X_0$, $X_1$, $X_2$, $X_3$ and $X_4$ were computed in two dimensions ($m = 24$) and three dimensions ($m = 168$) with double precision (*double*) floating point values. The computational time for

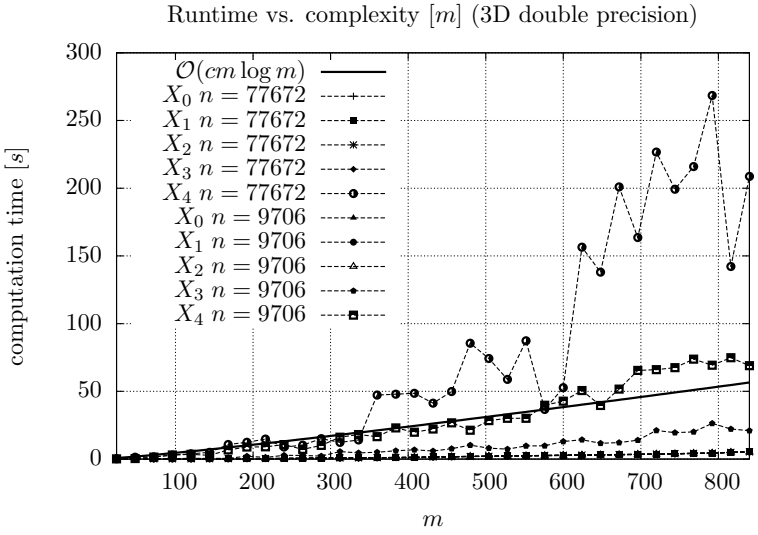



**Fig. 5** Comparison of the theoretical complexity $\mathcal{O}(cm \log m)$ with the measured computational time on the GPU.

the STANN library was measured on 1 core of an Intel Xeon 5500 CPU. The extended algorithm was executed on a Nvidia Tesla K20c GPU. The number of multidimensional input points is limited by the memory of the GPU, since our implementation of the presented algorithm does currently not support streaming techniques and thereby only single-GPU computing.

Figure 6 shows the measured computational time for the nodes of a regular grid $X_0$ with $n$ from 512 up to $n = 2^{21}$. The depicted plots show that the search on the GPU is substantially faster than on a single core of the CPU. Moreover, we find that the computational time in two and three dimensions on the GPU are essentially identical which indicates that the additional operations in three dimensions come for free.

Note however, that for this perfectly homogeneous node arrangement the results are not very representative for the general case since here essentially no (expensive) geometric validation of the initial guess is necessary. Thus the initial guess $\tilde{N}(x_i)$ contains all geometrical neighbors with respect to the euclidean distance.

The plots in Figure 7 shows that for the graded nodes of a regular grid $X$ the asymptotic behavior holds[1]. The difference between the computational time between the dimensions should be $\frac{24 \log 24}{168 \log 168} \approx 0.088$, excluding the

[1] Storing the graded nodes of the two dimensional regular grid in row-major order in a thrust vector leads to some issues in the costs of the Merge sort algorithm. To avoid this the nodes need to be stored randomized.

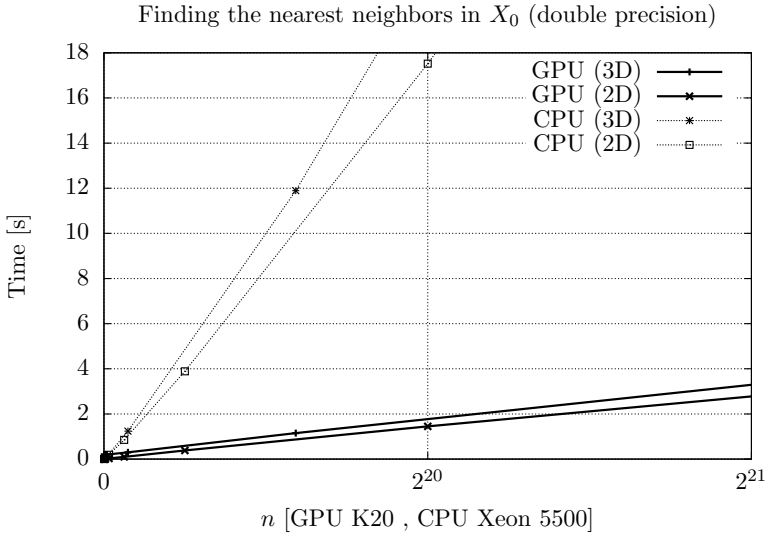



**Fig. 6** Computational time for finding the nearest neighbor in the nodes of regular point grid $X_0$.

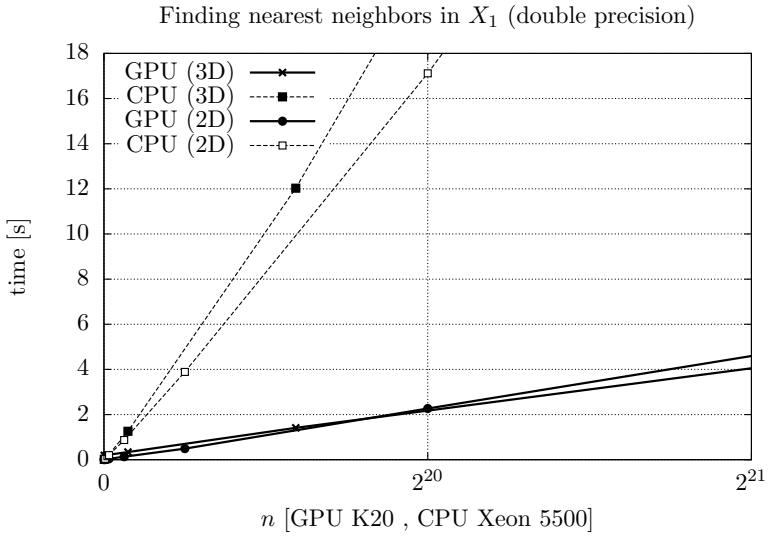



**Fig. 7** Computational time for finding the nearest neighbor in a graded point cloud of a regular grid $X_1$ 1(a).

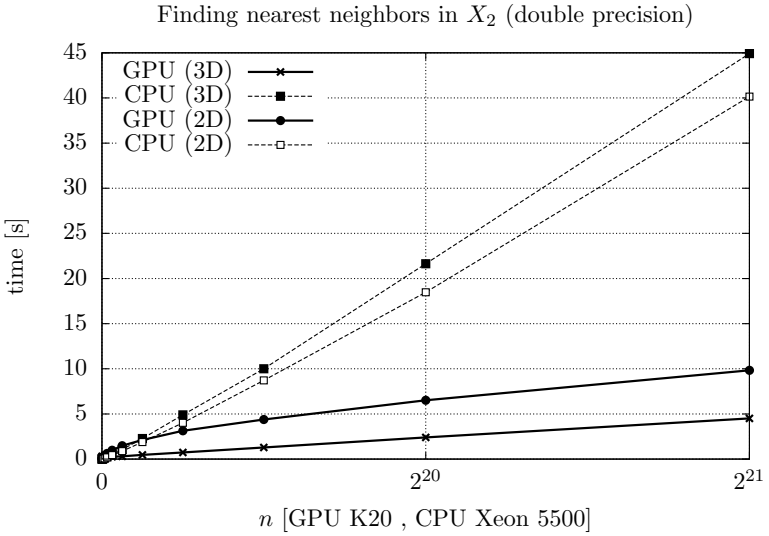



**Fig. 8** Computational time for finding the nearest neighbor in the Halton sequence $X_2$ 1(b).

computational costs for the euclidean distance in three dimensions. On the GPU we differ between the dimensions with the factor 0.01 in $X_2$.

Figure 8 shows the computational time for the Halton sequence $X_2$. Here we see a difference of 2.18 in the computational time. Here we see the sub-optimal locality of the curve for particles which are close in respect to the Morton order but not with respect to their euclidean distances.

Figure 9 shows the measured time for the graded Halton sequence $X_3$. Here we see a factor 0.77 between the computational time in two and three dimensions which is a factor of 10 away of the theoretical value.

Figure 10 shows the results for the point cloud $X_4$. Here we see artifacts of the differing particle density up to particle clouds with $n > 2^{21}$. Here the factor between the dimensions is 0.52. In this point cloud we see that the computational time depends on the particle density. Particle clouds with differing density has a enormous effect on the computational time.

## 5 Conclusion

We presented an efficient neighbor search for particle clouds on GPUs. As a reference particle method we considered peridynamics. We used the basic idea of sorting points with respect to a space-filling curve as a massive parallel implementation on a GPU and described briefly the four steps essentially

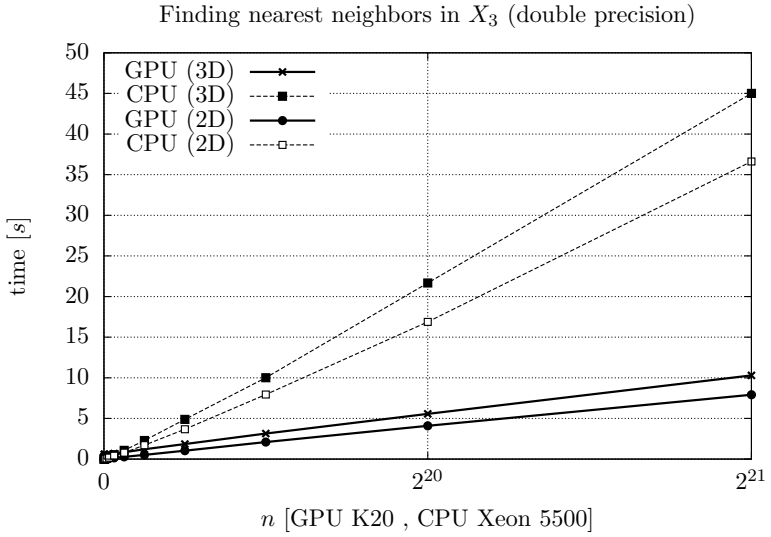



**Fig. 9** Computational time for finding the nearest neighbor graded Halton sequence $X_3$ 1(c).

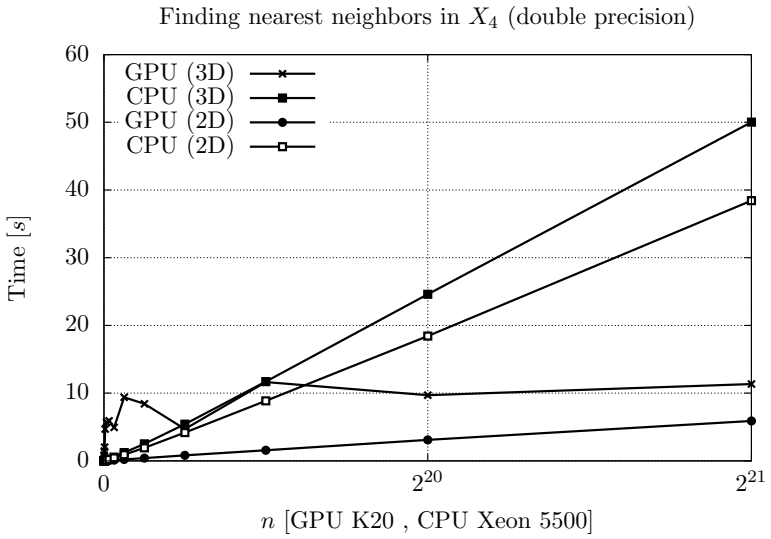



**Fig. 10** Computational time for finding the nearest neighbor in a uniformly distributed random points $X_4$ 1(d).

realizing such a sorting strategy. We validated our implementation against the theoretical complexity of the algorithm and compared the measured run times with the performance of the library STANN. We are in general faster by a factor of 4. Only for very small particle clouds $X_4$ we need more computational time. Using distributed points clouds, with a differing particle density, influence the computational time enormous, because of the extension of the range of the initial guess $\tilde{N}(x_i)$. The costs for copying the multidimensional point cloud $X$ to the device and copying the resulting $m$ neighbors back to the host does not influences the overall computational time. Thus, the library is suitable to update the neighborhood every time step in dynamic particle clouds.

To make the algorithm applicable in uncertainty quantification the dimension $d$ of the point should be enlarged to $d > 3$. Advanced techniques, like data streaming or Multi-GPU computation, bypass the restriction of the memory of the GPU.

## References


1. Srinivas Aluru and Fatih E. Sevilgen, *Parallel domain decomposition and load balancing using space-filling curves*, in Proceedings of the 4th IEEE Conference on High Performance Computing, 1997, pp. 230–235.
2. Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu, *An Optimal Algortihm for Approximate Nearest Neighbor Searching in Fixed Dimensions*, Fifth Annual ACM-SIAM Symposium on Discrete Algorithms **5** (1994), 573–582.
3. Michael Bader, *Space-filling curves – an introduction with applications in scientific computing*, Springer Berlin Heidelberg, 2013.
4. C. Böhm, S Berchtold, and A. D. Keim, *Searching in high-dimensional spaces: Index strucutres for improving the performance of multimedia databases*, ACM Computing Sruveys, 33, 2001, pp. 322–373.
5. Timothy M. Chan, *A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions*, https://cs.uwaterloo.ca/ tmchan/sss.ps, May 2006.
6. Michael Connor and Piyush Kumar, *Stann*, https://sites.google.com/a/compgeom.com /stann/.
7. Ali Dashti, Ivan Komarov, and Roshan M. D'Souza, *Efficient Computation of k-Nearest Neighbour Graphs for Large High-Dimensional Data Sets on GPU Clusters*, plosone.org, 2013.
8. Vincent Garcia, Eric Debreuve, and Michael Barlaud, *kNN CUDA*, http://vincentfpgarcia.github.io/kNN-CUDA/.
9. R.A. Gingold and J.J. Monaghan, *Smoothed particle hydrodynamics: theory and application to non-spherical stars*, Mon. Not. R. Astron. Soc. **181** (1977), 375–389.
10. M. Griebel, S. Knapek, and G. Zumbusch, *Numerical Simulation in Molecular Dynamics*, Springer, Berlin, Heidelberg, 2007.
11. Pedro Leite, JoãoMarcelo Teixeira, Thiago Farias, Bernardo Reis, Veronica Teichrieb, and Judith Kelner, *Nearest neighbor searches on the gpu*, International Journal of Parallel Programming **40** (2012), no. 3, 313–330 (English).

12. J. Mellor-Crummey, D. Whalley, and K Kennedy, *Improving memory hierarchy performance fir irregular applications using data and computation reorderings*, International Journal of Parallel Programming, vol. 29, June 2001, pp. 217–247.
13. Michael Connor and Piyush Kumar, *Fast construction of k-Nearest Neighbor Graphs for Point Clouds*, IEEE Transactions on Visualization and Computer Graphics, 2009.
14. David M. Mount and Sunil Arya, *ANN: A Library for Approximate Nearest Neighbor Searching*, http://www.cs.umd.edu/ mount/ANN/.
15. S.A. Nene and S.K Nayar, *A simple algorithm for nearest neighbor search in high dimensions*, IEEE Transactions on Pattern Analysis and Machine Intelligence **19** (1997), 989 – 1003.
16. Michael L. Parks, Richard B. Lehoucq, Steven J. Plimpton, and Stewart A. Silling, *Implementing peridynamics within a molecular dynamics code*, Computer Physics Communications (EL, ed.), vol. 179, June 2008, pp. 777–783.
17. S. Plimpton, *Fast Parallel Algorithms for Short-Range Molecular Dynamics*, Journal of Computational Physics, 117, 1995, pp. 1–19.
18. Nadathur Satish, Mark Harris, and Michael Garland, *Designing Efficient Sorting Algorithms for Manycore GPUs*, IEEE International Symposium in Parallel & Distributed Processing, 2009, pp. 1–10.
19. M. A. Schweitzer, *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*, Lecture Notes in Computational Science and Engineering, vol. 29, Springer, 2003.
20. Yaroslav D. Sergeyev, Roman G. Strongin, and Daniela Lera, *Introduction to global optimization exploiting space-filling curves*, Springer, 2013.
21. S. A. Silling, *Reformulation of Elasticity Theory for Discontinuties and Long-Range Forces*, Sandia Report SAND98-2176, Sandia National Laboratories, 1998.
22. S. A. Silling and E. Askari, *A meshfree method based on the peridynamic model of solid mechanics*, Computer & Structures, vol. 83, 2005, pp. 1526–1535.
23. Erik Sintorn and Ulf Assarsson, *Fast parallel GPU-sorting using a hybrid algorithm*, Journal of Parallel and Distributed Computing, vol. 68, October 2008.
24. H. Tropf and H. Herzog, *Multidimensional Range Search in Dynamically Balanced Trees*, Angewandte Informatik (Applied Informatics), Vieweg Verlag, February 1981, pp. 71–77.
25. M. S. Warren and J. K. Salmon, *A parallel hashed oct-tree n-body algorithm*, Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (New York, NY, USA), Supercomputing '93, ACM, 1993, pp. 12–21.
26. Wen-mei W., *GPU Computing Gems Emerald Edition (Applications of GPU Computing Series)*, 1 ed., Morgan Kaufmann, February 2011.