Out-of-Core Proximity Computation for Particle-based Fluid Simulations

Duksu Kim [1] Myung-Bae Son [1] Young-Jun Kim [2] Jeong-Mo Hong [3] Sung-eui Yoon [1]
Dept. of CS, KAIST, Technical Report CS-TR-2014-385
[1] KAIST (Korea Advanced Institute of Science and Technology)
[2] Ewha Womans University, Seoul, Korea [3] Dongguk University, Seoul, Korea

Abstract

To meet the demand of higher realism, a high number of particles are used for particle-based fluid simulations, resulting in various out-of-core issues. In this paper, we present an out-of-core proximity computation, especially, ϵ-nearest neighbor (ϵ-NN) search, commonly used for particle-based fluid simulations, to handle such big data sets consisting of tens of millions of particles. We use a uniform grid and perform ϵ-NN in the granularity of sub-grids, called blocks. Specifically, we identify a maximal block that a GPU can process efficiently in an in-core mode based on a workload tree. As a main technical component, we compute the memory footprint required for processing blocks based on our expectation model of the number of neighbors of particles. Our method can naturally utilize heterogeneous computing resources such as CPUs and GPUs, and has been applied to large-scale fluid simulations based on smoothed particle hydrodynamics. We demonstrate that our method handles up to 65 M particles and processes 21 M ϵ-NN queries per second by using two CPUs and two GPU, each of which has only 3 GB video memory. This high performance for large-scale data given a limited video memory space is achieved mainly thanks to the high accuracy of our memory estimation method and efficiency of our out-of-core ϵ-NN system.

1 Introduction

Thanks to ever growing demands for higher realism and the advances of particle-based fluid simulation techniques, large scale simulations are getting increasingly popular across different graphics applications including movie special effects and computer games. This trend poses numerous technical challenges related to an excessive amount of computations and memory requirements.

In this paper we are mainly interested in handling nearest neighbor search used for particle-based fluid simulations. Nearest neighbor search is performed for each particle in the simulation and dominates the overall computation cost of the simulation in practice [Solenthaler and Gross 2011]. Most particle-based fluid simulations use ϵ-Nearest Neighbor, ϵ-NN, for a query particle, which identifies all the particles that are located within a search sphere, whose center is at the query particle and radius is set to ϵ.

To achieve a higher performance for large-scale particle-based fluid simulations, many parallel techniques have been proposed [Goswami et al. 2010; Ihmsen et al. 2011]. These approaches utilize many cores of GPU and achieve much higher performance over using a CPU thread (e.g., about 20× to 40× higher performance according to our own test and the work of Harada et al. [2007]). Unfortunately, it has not been actively studied to handle massive-scale ϵ-NNs for data sets that do not fit in the video memory of GPU for particle-based fluid simulations. For example, we can handle about up to 5 M particles per 1 GB video memory in the GPU side for the simulation [Harada et al. 2007], and most commodity-level GPUs have one to three GBs of the video memory. As a result, large-scale particle-based fluid simulations consisting of more than 10 M or more particles have to be processed in a much less performance in the CPU side that can have much larger memory space than GPU. This is mainly because prior GPU-based parallel techniques were neither designed to out-of-core cases nor directly applicable to such cases.

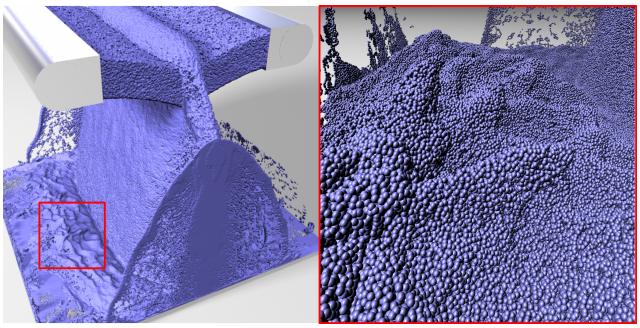

Figure 1: *These figures show a particle-based fluid simulation frame of our two sources benchmark consisting of up to 65 M particles. The right image zooms in simulated particles within a box shown in the left image. Our ϵ-NN method takes 3.1 s on average per frame by using two hexa-core CPUs and two Geforce GTX 780.*

Contributions. In this paper we propose an out-of-core technique utilizing heterogeneous computing resources for processing ϵ-NNs used in particle-based fluid simulation consisting of tens of millions of particles. In particular, we handle the out-of-core problem where the video memory of GPUs cannot hold all the necessary data of ϵ-NN, while main memory of CPU is large enough to hold such data. As an acceleration data structure for ϵ-NN, we use a uniform grid that is commonly used for particle-based fluid simulations.

Given this context, we use the granularity of a block containing a sub-grid of the uniform grid as a main work unit, to streamline various computation and memory transfer between CPU and GPU. Once GPU receives a block from CPU, the GPU performs ϵ-NNs with the particles contained in the block (Sec. 3.1). Our main problem is then reduced to identify a maximal work unit that can fit into the video memory. To estimate the memory requirement of processing a block, we present a novel, memory estimation method based on the expected number of neighbors for a query particle (Sec. 4). To efficiently compute a maximal block for each GPU, we also propose a simple, hierarchical work distribution method (Sec. 3.2).

To demonstrate the benefits of our method, we have tested our method with three large-scale particle-based fluid simulation benchmarks consisting of up to 65 M particles. These benchmarks require up to 21 GB memory space for processing ϵ-NNs. Our out-of-core method for ϵ-NNs can process these benchmarks with two GPUs, each of which has

only 3 GB video memory. Overall, our method can perform 21 M ϵ-NNs per second in this configuration consisting of two GPUs and two CPUs. We have also implemented an alternative, GPU-based out-of-core approach based on an Nvidia's mapped memory method [NVIDIA 2013]. Compared to this alternative, our method shows up to 22 $\times$ performance improvement. These results are mainly thanks to the efficiency of our out-of-core method and the high accuracy of our memory estimation model that shows up to 0.97 linear correlation with respect to the observed number of neighbors. Also, compared to our base method, an in-core CPU version using only those two hexa-core CPUs and the large main memory space holding all the data, our method achieves 5.7 $\times$ improvement using the additional two GPUs.

2 Related Work

In this section we review prior neighbor search techniques and their applications to particle-based fluid simulations.

2.1 Particle-based Fluid Simulation

In the Lagrangian context, fluid is discretized by particles. Smoothed Particle Hydrodynamics (SPH) is a well-known particle-based solver, and a series of extensions for SPH has been proposed to improve the simulation quality and performance [Müller et al. 2003; Becker and Teschner 2007; Solenthaler and Pajarola 2009; Ihmsen et al. 2013].

For particle-based solvers, the physical and visual quality of the simulation strongly depends on the number of particles. Generally, many particles are needed to catch small-scale details like splashes, spray, and surface waves in large-scale scenes. To meet the increasing demands of high quality simulations, the number of required particles continues to increase. There have been techniques to reduce the number of particles [Solenthaler and Gross 2011], but the number can be still high, requiring to run the simulation in an out-of-core manner, especially when many details need to be presented in simulations.

In particle-based methods, the simulation is performed based on the neighborhood relationship among particles. Neighbor search is commonly performed for each particle and thus dominates the overall simulation time [Solenthaler and Gross 2011]. In our SPH simulation based on the method of Becker and Teschner [Becker and Teschner 2007], we found that neighbor search can take up to 90% of the overall simulation time, when we use a single CPU core.

2.2 Near Neighbor Search (NNS)

NNS is one of the widely used proximity queries and finds points closely placed to given a query point in a metric space [Samet 2006]. There are two variations of NNS: k-Nearest Neighbor (k-NN) search that finds top k nearest neighbors to a query point, and ϵ-NN.

NNS has been widely employed in various applications such as similarity searches for image retrieval [Heo et al. 2012], robotics [Pan et al. 2010], and particle-based simulations [Ihmsen et al. 2011]. Because of its high computation cost, NNS has been accelerated in various ways based on finding approximated results [Li et al. 2012] and reducing high dimensional space [Heo et al. 2012]. Spatial partitioning is also a commonly used method (e.g., kd-trees and grids) that narrows down the search space [Lin and Manocha 2003]. In our method, we use a uniform grid as an acceleration data structure that is usually employed for ϵ-NNs in particle-based fluid simulations.

2.3 Parallel NNS

Recently, parallel computing resources have been actively used to improve the performance of NNS queries. Many prior parallel methods are designed for k-NN used for photon mapping [Purcell et al. 2003; Zhou et al. 2008], 3D registration [Qiu et al. 2009], etc. Unfortunately, these methods are neither directly applicable nor effective to our problem, since our application uses ϵ-NN, and using algorithms designed for k-NN shows inferior performance over techniques specialized for ϵ-NN.

Parallel algorithms for ϵ-NN have been actively studied in the particle-based fluid simulation field. By utilizing the inherent parallel nature of many ϵ-NNs, efficient GPU-based SPH implementations have been proposed [Harada et al. 2007; Goswami et al. 2010]. These methods distribute particles to threads and each thread finds the neighbors of the given particle. While these approaches are simple, they are not designed for out-of-core cases, and Harada et al. [2007] reported that about 5 M is the maximum number of particles that the GPU-based method can handle with 1 GB video memory. Ihmsen et al. [2011] used multi-core CPUs in the whole process of SPH. They showed that a CPU-based parallel approach can handle a larger number, 12 M, of particles, thanks to the large memory space, 128 GB, in the CPU side.

In this paper we propose an efficient, parallel ϵ-NN algorithm that can handle a large number of particles as much as CPU memory allows, while utilizing GPU's high computing power in an out-of-core manner.

2.4 Out-of-Core GPU Algorithms

The limited memory space in the GPU size raises various challenges for handling a large data set in GPU. The out-of-core issue has been well studied for rendering [Yoon et al. 2008]. Nonetheless, it has not been actively studied for different parts of particle-based fluid simulation.

Abstracting distributed memory space of CPU and GPU into a logical memory is a general approach for handling massive data with GPU. Nvidia's CUDA supports a memory space mapping method that maps pinned-memory space into the address space of GPU [NVIDIA 2013]. While it is convenient to use, it can be inefficient, unless minimizing expensive I/O operations effectively. We compare this mapped memory based out-of-core approach with ours in Sec. 5.

Different out-of-core techniques have been proposed for k-NN used in ray tracing and photon mapping. In particular, Budge et al. [2009] designed an out-of-core data management system for path tracing with kd-trees constructed over polygonal meshes. This approach adopted a pre-defined task assignment policy to distribute different jobs to CPU or GPU. Recently, Kim et al. [2013b] used separate, decoupled data representations designed for meshes to fit large-scale data in the video memory. Unfortunately, it is unclear how these techniques designed for k-NNs can be applied to our problem using ϵ-NN and particles. Furthermore, an out-of-core GPU approach tailed to ϵ-NN used for particle-based fluid simulation has not been proposed yet, to the best of our knowledge.

3 Out-of-Core, Parallel ϵ-NN

In this section we give a system overview of our method, followed by our hierarchical work distribution method. We target mainly for handling large-scale ϵ-NN used for particle-based fluid simulation both in out-of-core and parallel manners.

Theoretically, achieving the optimal performance in this context is non-trivial and thus has been studied only for particular problems such as sorting and FFTs [Blelloch et al. 2010] on the shared memory model with the same parallel cores. As a result, we propose a hierarchical approach, tailored to our particular problem, that simultaneously com-

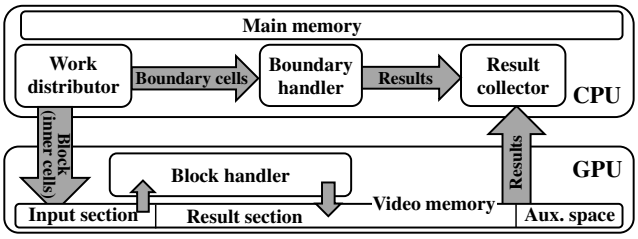

Figure 2: *This figure shows an overall framework for processing ϵ-NNs in an out-of-core manner using heterogeneous computing resources.*

putes a job unit that can fit into the video memory of a GPU, while utilizing heterogeneous parallel computing resources.

3.1 System Overview

The main goal of our system is to efficiently find and store the neighborhood information for a massive amount of particles that cannot be handled at once by a GPU. We assume that the CPU memory is large enough to hold all those information. This assumption is valid for tens or hundreds of millions of particles, since current PCs can have hundreds of gigabytes up to 4 TB memory.

For particle-based fluid simulations, a uniform grid is commonly used for accelerating ϵ-NN and we also use a uniform grid, while determining cell indexes with Z-curve to exploit spatial locality [Ihmsen et al. 2011]. To construct a uniform grid, we split the simulation space uniformly with a given grid resolution and each divided space is called a *cell*. The grid resolution, i.e., the dimension of each cell, is usually set to the search radius ϵ or 2ϵ. In either case, neighboring particles for a query particle with the ϵ-NN are located in the cell of the query particle or its adjacent cells.

Fig. 2 shows an overview of our system, which consists of CPU and GPU. Initially we construct the uniform grid with the multi-core CPU and then perform ϵ-NN by using the grid in GPU. To perform an ϵ-NN for a query particle, we need to access the cell containing the query particle and its adjacent cells. As a result, we need to send those cells and their particles from CPU to GPU to perform the ϵ-NN. We use the term *processing cells* to denote the process of performing ϵ-NN for particles in the cells.

There are three components in the CPU side: *work distributor*, *boundary handler*, and *result collector*. We use the work distributor to identify an appropriate amount of data including cells and their particles that can fit into the video memory and to distribute them to the GPU. To efficiently identify such data, we use a *workload tree*, whose node indicates how many particles are located in the sub-tree rooted at the node and how much memory space is needed to perform ϵ-NN for those particles in the sub-tree (Sec. 3.2).

We use the term of a *block* to denote a sub-grid defined by a node of the workload tree. The work distributor allocates work as the form of blocks to GPU dynamically based on the workload tree. There are two types of cells in a block; the cells at the boundary of a block are denoted as *boundary cells* and other cells are *inner cells*. Boundary cells are chosen to be processed in the CPU-side boundary handler, since handling these cells require a larger working set over handling inner cells. Once the boundary handler finishes to process the boundary cells, it then sends the results of ϵ-NN to the result collector.

The GPU side has a single component, *block handler*, that processes inner cells contained in a given block. The block handler in the GPU side maintains a work queue to receive blocks from the work distributor. When a work queue is not full, the work distributor can push a block to the work queue. When GPU is idle, the block handler dequeues a block from the work queue and loads required data from main memory into the video memory and processes the block. Once the GPU finishes to process the block, it pushes the results back to the result collector (Sec. 3.3). Finally, the result collector takes the results, stores them in main memory, and returns them to the particle-based fluid simulator.

3.2 Hierarchical Work Distribution

We process a block containing a cubic sub-grid and its contained particles as the main work unit for CPUs and GPUs, to exploit spatial locality and thus to achieve a high utilization efficiency. The smallest block contains only a single cell (e.g., 1^3 cube) and a block size can increase exponentially (e.g., 2^3 and 4^3). Among possible block sizes, we use a workload tree to identify a proper block size that fits to the video memory of the target GPU and to efficiently process the block.

It is very important to accurately estimate the required amount of memory space for processing a block, in order to efficiently perform out-of-core computation. Processing a block requires to access particles and to write indices of their identified neighbor particles to GPU. Therefore, the required memory size, $s(B)$, for processing a block, B, can be determined mainly by the number of particles, n_B, stored in the block and the number of neighbors for each particle, n_{p_i}, as the following:

$$s(B) = n_B s_p + s_n \sum_{p_i \in B} n_{p_i}, \tag{1}$$

where s_p and s_n are the data sizes of storing a particle and a neighbor particle, respectively. i indicates the i-th particle stored in the block B. Typically, s_n is 8 bytes required for encoding an index of a particle and the distance between the query and its neighbor particle that is used for computing forces in the simulation part, while s_p is much larger since we need to encode the particles' positions, etc.

Evaluating the required size of processing a block is straightforward except the number of neighbors, n_{p_i}. Unfortunately, we cannot know the exact number of neighbors until we actually perform the query, a common chicken-and-egg problem. One can pre-define a maximum number of neighbors (e.g., 500) for a simulation and reserve the maximum space, but can significantly degrade the memory utilization for the out-of-core computation due to the overestimated allocation. Another alternative is to use a general vector-like data structure that adaptively grows according to identified neighbors. Along this direction, efficient GPU implementations are available for this kind of data structure [Yang et al. 2010]. Nonetheless, these data structures are not designed for the out-of-core case and thus fail, when all the sizes of these vectors grow even bigger than the available video memory size.

Instead of these approaches, we estimate the number of neighbors as the expected number of neighbors, and reserve the memory space based on the estimation result. Specifically, we compute the expected number of neighbors based on the distribution of particles in the simulation space; its details are given in Sec. 4. Thanks to this estimation process, we can efficiently perform ϵ-NN with GPU in an out-of-core manner.

Once we know the required memory space for processing a block, the next task is to compute a maximal block that fits into the target GPU. To efficiently construct such a maximal block, we use the workload tree, which is an octree built on

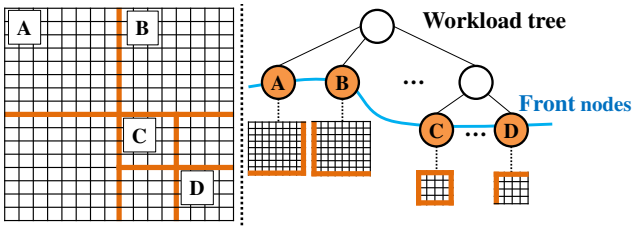

Figure 3: *The left figure shows a uniform grid with a few sub-grids; boundaries of these sub-grids, i.e., blocks, are shown in orange lines. The right figure shows an example of the workload tree with these blocks.*

top of the grid (Fig. 3). Each node of the workload tree represents a block, and also contains the number of particles included in the block and the expected number of neighbors of those particles. Its child nodes are computed by dividing the sub-grid of the parent node in all the dimensions. As a result, each leaf node of the tree represents a cell, while the root includes the whole uniform grid.

Work distribution. The distributor running on a CPU thread maintains a *front node queue* containing blocks that will be processed by GPUs. Initially the front node queue contains the root node of the workload tree. When a GPU is available, the work distributor takes the front node and checks whether the GPU has enough memory space for processing the block contained in the node. If not, the distributor enqueues its eight child nodes to the queue. Otherwise, we assign the block to the GPU. Based on this simple, hierarchical process, we efficiently identify and process a maximal block for the GPU. An example of front nodes in the workload tree is shown in Fig. 3.

Note that in this process we may not fill the video memory of the GPU with a single maximal block. One may concern that this approach may have many GPU kernel launches and their overhead may be high. As an alternative to our current approach, we have also tried to add more blocks to almost fill the video memory, but this alternative results in more complex implementations with a negligible performance improvement. This is mainly because multiple GPU kernel launches have relatively small overheads (e.g., a few ms) compared to the most common processing time (100 ms on average for a 32^3 sub-grid) of a block in our tested benchmarks. As a result, we have chosen our current, simple approach.

3.3 Processing a Block in GPU

When a block is given to a GPU, we configure its available memory space into work and auxiliary spaces. The work space is decomposed into input and result sections (Fig. 2), and the *input section* is reserved for holding input data such as particles positions and other required data for processing the given block. The *result section* is reserved for collecting results of ϵ-NNs. The size of the result section is computed according to the expected number of neighbors of query particles stored in the block, and then each query particle receives the estimated, fixed amount memory space.

Once we reserve the memory space in the GPU side, we copy the required input data from main memory to the work space of the GPU, and then generate GPU threads, each of which processes an ϵ-NN for a query particle in the block. As a GPU thread identifies neighbors for the query particle, it stores them in its pre-defined, corresponding space in the result section. Since we have already reserved memory space for maintaining results of each query particle, each GPU thread writes their results to the result section without any locking operations.

Nonetheless, we may need further memory spaces than the pre-defined memory space in the result section, due to inaccuracy of our estimation process. In this case, we write such results into the auxiliary space. Multiple GPU threads can access the auxiliary space simultaneously and thus we need to perform synchronization. Fortunately, we have found that this happens rarely (i.e., 3% of all identified neighbors on average), thanks to the high accuracy of our estimation model. Even when the auxiliary space becomes full, we can also use an additional space in main memory in the CPU side. This operation accessing main memory from the GPU side is very expensive, and never happened in our method, when we allocate 250 MB for the auxiliary space with our tested benchmarks. In Sec. 4.2 we explain how we set the size, 250 MB, of auxiliary to avoid such expensive overflows of the auxiliary space. Our approach handling these overflows can be seen as designing an effective out-of-core vector data structure, whose initial size is determined by our memory estimation model, while reducing the expensive synchronizations.

After finishing to process the given block in the GPU side, we notify the results collector in the CPU side and transfer results to main memory.

3.4 Boundary Cells Handling

When a block B is given to a GPU, we send particles contained only in the block B. As a result, to find neighbors of particles in the boundary cells, we need information of particles in their adjacent cells, some of which are stored in adjacent blocks. We could send those adjacent cells/blocks together with the block B, but we have found that this approach requires a higher memory footprint and lowers down the locality.

Instead, we let CPU cores that are mostly idle to process those boundary cells. CPU can efficiently process the boundary cells, since main memory already has a well-organized cache hierarchy that can efficiently handle random memory accesses in the CPU side. Note that the number of boundary cells are much the smaller than that of inner cells, because boundary and inner cells exist in 2D and 3D spaces of the uniform grid, respectively. As a result, we let a high-performance GPU to process those many inner cells, while allocating CPU to process a lower number of boundary cells that cause frequent cache misses. Nonetheless, one could use either one of CPU or GPU for handling boundary and even inner cells for achieving the optimal performance. We, however, have not explored this optimal scheduling problem of heterogeneous computing resources [Kim et al. 2013a], and left it as a future work.

When the work distributor decides a block for processing, it identifies the boundary cells of the block and passes them to the boundary handler in the CPU side. The boundary handler processes those cells by using CPU threads and then sends results back to the result collector.

4 Expected Number of Neighbors

We have described so far that it is critical to compute and reserve an appropriate amount of memory space for processing blocks in an out-of-core manner. The main unknown factor for computing the required memory space (Eq. 1) for processing blocks is to compute the number of neighbors, n_{p_i}, of a particle. Computing them in an exact way is the chicken-and-egg problem and instead we estimate it by computing the expected number of neighbors based on the particle distribution in the simulation space, while considering the relationship between the search radius and cell size.

4.1 Problem Formulation

ϵ-NN for a particle, p, is to find neighbor particles, which are located within a search sphere, $S(p, \epsilon)$, whose center is at p and radius is ϵ. In general, particle distributions over the uniform grid covering the simulation space is not uniform. In many cells, however, particle distributions tend to show local uniformity around each cell in particle-based fluid simulations. This is mainly because designing high-quality SPH techniques is related to reduce the density variation over time [Solenthaler and Pajarola 2009; Becker and Teschner 2007] and therefore particles tend to have a similar movement with nearby particles, while maintaining a specific distance with them. Based on this observation, we assume a local uniform distribution, i.e., particles are uniformly distributed in each cell.

Assuming the local uniform distribution, the number of neighbors is then proportional to the overlap volume between the search sphere $S(p, \epsilon)$ and cells weighted by their associated particles. Specifically, the expected number of neighbors, $E(p_{x,y,z})$, for a particle p located at (x, y, z) is defined as the following:

$$E(p_{x,y,z}) = \sum_i n(C_i) * \frac{Overlap(S(p_{x,y,z}, \epsilon), C_i)}{V(C_i)}, \quad (2)$$

where C_i indicates the cell of $p_{x,y,z}$ and its adjacent cells that have any overlap, $Overlap(S(p_{x,y,z}, \epsilon), C_i)$, between the search sphere and the bounding box of the cell C_i. $n(C_i)$ is the number of particles contained in the cell C_i, and $V(C_i)$ represents the volume of the cell.

This equation requires us to compute $Overlap(S(p_{x,y,z}, \epsilon), C_i)$ for each query particle in a cell, and thus causes a high computational overhead overall, since many particles can exist in each cell (e.g., 10 to 30 on average). Instead, we compute the average, expected number of neighbors for particles of a cell, C_q, and use the value, $E(C_q)$, for all the particles, as their expected number of neighbors. The average, expected number of neighbors of particles $E(C_q)$ in a cell C_q is then defined as:

$$\begin{aligned} E(C_q) &= \frac{1}{V(C_q)} * \int_0^l \int_0^l \int_0^l E(p_{x,y,z})\ \mathrm{d}x\mathrm{d}y\mathrm{d}z \\ &= \frac{1}{V(C_q)} * \sum_i n(C_i) * \frac{D(C_q, C_i)}{V(C_i)}, \end{aligned} \quad (3)$$

where l is the length of a cell along each dimension, $p_{x,y,z}$ is a particle p positioned at (x, y, z) on a local coordinate space in C_q, and $D(C_q, C_i) = \int_0^l \int_0^l \int_0^l Overlap(S(p_{x,y,z}, \epsilon), C_i)\mathrm{d}x\mathrm{d}y\mathrm{d}z$.

Given the uniform grid with l and ϵ values, which are not frequently changed by users, $D(C_q, C_i)$ can be pre-computed depending on the relative configurations between C_q and C_i. As a result, we pre-compute these values in an offline manner (taking about 30 seconds), especially by using the Monte Carlo method, which achieves high accuracy as we generate many samples (e.g., 1 M).

At runtime, we evaluate $E(C_q)$ of Eq. 3 by considering $n(C_i)$ and looking up pre-computed $D(C_q, C_i)$ values, which is stored at a less than 1 KB sized look-up table. Overall, this runtime evaluation is done in a constant time. All the expectation computation combined with traversing the workload tree takes less than 10% of the overall computation performed at each frame on average with our tested benchmarks.

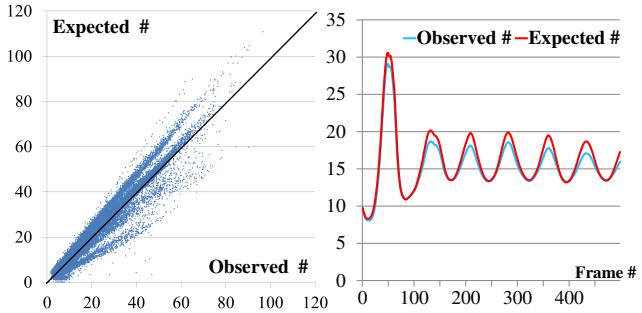

Figure 4: *The left plot shows a high correlations, 0.97, between the expected and observed numbers of neighbors in the dam breaking benchmark. The right graph shows the expected and actual numbers of neighbors averaged for all the particles at each simulation frame.*

4.2 Validation

We have measured the accuracy of our expectation model with our tested benchmarks. We set $l = 2\epsilon$ for computing the uniform grid, and compare the expected number of neighbors $E(C_q)$ with the actual number of neighbors that are computed after finishing ϵ-NN for each particle.

The left figure of Fig. 4 shows the scatter plot between the expected and observed results. As can be seen, our expectation model shows a high correlation (i.e. 0.97) between the observed and expected number of neighbors. We achieve such a high accuracy by considering the number of particles in cells, while assuming the local uniform distribution of particles. The right figure of Fig. 4 shows expected and observed numbers of neighbors averaged for all particles as a function of simulation frames. One interesting point is that we can observe the periodic overestimation of our method. In those periodic frames, the dam breaking benchmark undergoes strong compressions, resulting in higher pressures on deeper particles. This causes a subtle density variation along the depth and generates a case where our local uniform distribution does not apply well. Nonetheless, the overall trend of the right figure further verifies the high accuracy of our estimation method.

Additionally, we have measured the mean square error (MSE) between the estimated and observed ones. The MSE is computed as 3.8, indicating that our estimated number of neighbors can be higher or lower by 3.8 on average to the actual number of neighbors. This information is useful for estimating the minimum required space for the auxiliary space too. At the worse case where we underestimate 3.8 neighbors for each query particle in a block, we need to access the auxiliary space to accommodate such underestimation. For the most common block size (32^3 sub-grids) and the maximum (26.1) of the average numbers of neighbors across cells in our tested benchmarks, our auxiliary space that allocates 8 bytes for recording the neighbor ID and the distance between two particles requires 24.78 MB at least. Based on this we have assigned 250 MB, an ample space that can also accommodate overflows generated from 64^3 sub-grids, for the auxiliary space.

5 Results and Analysis

We have implemented and tested our out-of-core parallel ϵ-NN method for particle-based fluid simulation in a machine consisting of a GPU (Nvidia Geforce GTX 780) and two Intel Xeon hexa-core CPUs (2.93GHz) with 192 GB main memory. Unless mentioned otherwise, we use this machine

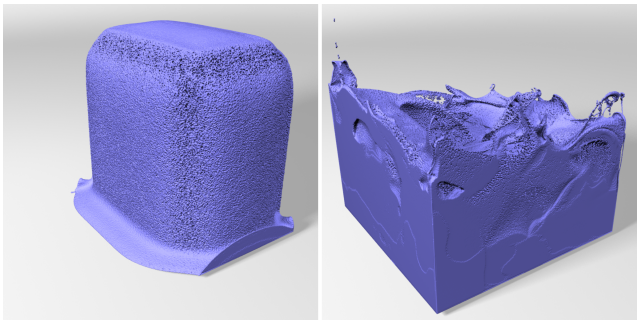

Figure 5: *This figure shows two sequences of the dam breaking benchmark consisting of 15.8 M particles.*

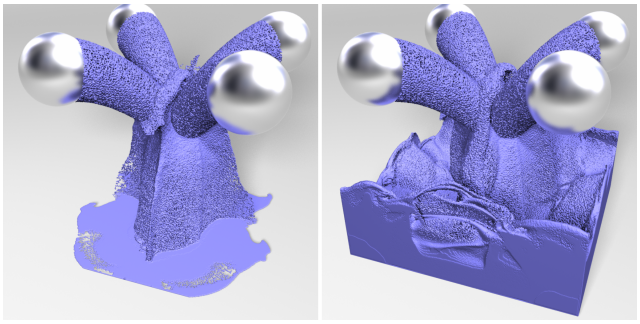

Figure 6: *This figure shows two sequences of the four source benchmark consisting of 32.7 M particles.*

configuration consisting of two CPUs and one GPU for various tests. The tested Nvidia Geforce GTX 780 has 3 GB video memory, and we use about 2.8 GB of those 3 GB GPU memory for all the tests, since some of GPU memory (e.g., 200 MB) is reserved by GPU drivers for display and running CUDA kernels (e.g., thread local memory).

We have implemented a CPU version of our parallel algorithm based on a prior CPU-based method [Ihmsen et al. 2011] and use 12 threads for the boundary handler in the CPU side. We have also implemented a locality-aware GPU ϵ-NN similarly based on a prior in-core GPU algorithm [Goswami et al. 2010]. To implement the auxiliary space in GPU, we treat the auxiliary space as a memory pool for the vector data structure, and a GPU thread gets a slot from the pool by using atomic operations [Yang et al. 2010]. Our simulation method [Becker and Teschner 2007] has been implemented on multi-core CPUs based on Ihmsen et al. [2011]. We first perform ϵ-NNs and pass their results to the simulation solver, which moves particles based on the computed neighbor search results.

Compared methods. To measure the overhead and benefits of our out-of-core approach, we have implemented an in-core GPU algorithm, *IC-GPU*, by removing all out-of-core features from our method. Also, the in-core GPU algorithm stores all results in a vector data structure designed for GPU [Yang et al. 2010], instead of reserving a specific space for each particle. As a result, it uses all available GPU memory as a memory pool like the auxiliary space used in our method.

We have also implemented an out-of-core ϵ-NN method using Nvidia's mapped memory method, *Map-GPU*, to see the efficiency of our method. In *Map-GPU*, a sufficiently large space (e.g., 50 GB) is reserved in main memory and is then mapped into the GPU memory address space for writing and accessing ϵ-NN search results.

Benchmarks. We have tested different methods against three different benchmarks (Table 1). The first benchmark, Dam, is a well-known dam breaking benchmark that has a fixed number of particles, 15.8 M particles, throughout the simulation (Fig. 5). The other two benchmarks, four and two sources benchmarks, have four or two sources emitting particles up to 32.7 M and 65.6 M particles, respectively (Fig. 6 and Fig. 1). These benchmarks are available at our project webpage, which is hidden for the anonymous submission. In these benchmarks, we use grid resolutions of 128^3 and 256^3 and set ϵ to be the half of the used cell size. In these settings, the average number of neighbors for each particle in three different benchmarks ranges 11 to 26, while their maximum reaches up to 489 neighbor particles.

	Dam (Fig. 5)	**Four src.** (Fig. 6)	**Two src.** (Fig. 1)
Max. # of pts.	15.8 M	32.7 M	65.6 M
Grid res.	128^3	128^3	256^3
Max. data size	7.8 GB	21.1 GB	17.9 GB
Avg. n_{p_i}	15.4	26.1	11.0
Max. n_{p_i}	184	489	327
Avg. $\sigma(n_{p_i})$	9.6	26.9	10.8

Table 1: *This table shows different statistics of each benchmark. We show the average and maximum numbers of neighbors computed for each simulation frame, with the standard deviation. It also shows the maximum data size required for processing ϵ-NNs for each benchmark.*

These three benchmarks require 8 GB to 21 GB memory space to contain all the required data given the configuration. The space is used for holding particle positions, grid structures, and recording neighbors identified for ϵ-NNs. The four sources benchmark has the biggest memory requirement up to 21 GB, although the two sources benchmark has more particles. This is because the four sources benchmark has a less grid resolution and thus has about 2.3× more neighbors than the two sources one.

5.1 Results

Fig. 7 shows the performance of different methods on each benchmark. For the four and two sources benchmarks, we draw the graph as a function of the number of particles, to see the overhead and benefits of our out-of-core approach over the in-core method.

As long as all the data fits into the video memory of a GPU, we can use the in-core method to perform all ϵ-NN query within the GPU. In this case, the in-core GPU method *IC-GPU* shows a higher (i.e., 60% on average) performance than our out-of-core method. This is mainly because our out-of-core method performs unnecessary out-of-core operations such as generating and traversing the workload tree by estimating the number of neighbors even for the in-core case. This overhead, however, is a small price to pay for handling the out-of-core case.

At a specific point (e.g., 12 M particles for the four sources benchmark), the in-core method fails to perform ϵ-NNs, since the required memory space is larger than available memory in the GPU. On the other hand, our method identifies maximal blocks that fit into the video memory based on the workload tree and processes them in an in-core mode. As a result, our method continues to process out-of-core data with a graceful performance degradation, while the in-core approach fails.

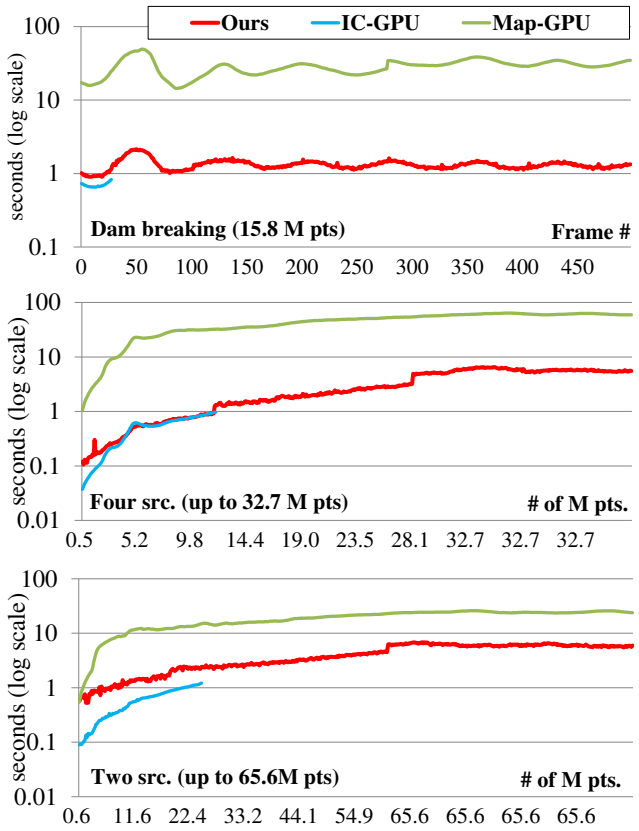

Figure 7: *These graphs show the processing time in the log scale for ϵ-NNs based on different in-core and out-of-core methods including ours.*

We have also compared our method with *Map-GPU* to see the efficiency of our out-of-core approach (Fig. 7). Our method achieves higher performances: 22 × for the Dam, 14 × for the four sources, and 5 × for the two sources benchmark over *Map-GPU* on average. Detailed implementation for the mapped memory feature used for *Map-GPU* in the GPU driver is not provided, but L2 cache in the GPU side is used for the mapped memory. On the other hand, we specifically use the global memory in GPU for caching data (i.e., particles and cells) and reserving the memory space with our memory estimation model. Thanks to them, our method achieves such high performance improvements over *Map-GPU*.

5.2 Benefits of Our Memory Estimation Model

To measure benefits caused by our expected number of neighbors for query particles, we have tested our method without using the estimation model. Instead we set a fixed space for recording results of ϵ-NNs; when we have the overflow, we also use the auxiliary space in the video memory and then use space in main memory, as used for our method. We use values, i.e., 16, 32, and 64, around the average number of neighbors observed in our tested benchmarks. Overall, our system equipped with our estimation method achieves much higher performance, 6 × to 13 × over the tested fixed neighbors.

For a small fixed space (e.g., 16), we found that some of the identified neighbors have to be recorded in main memory through expensive PCI-Express communication, and thus it drops down the performance significantly. For a large space (e.g., 64), the transaction to main memory is much reduced compared to the case with 16 fixed neighbors, but still many transactions occur; about 3% of the identified neighbors are recorded in main memory, while wasting other spaces. On the other hand, our memory estimation method results in a high space utilization, i.e., more than 90% of allocated spaces are used, while avoiding expensive main memory access during processing a block. These results demonstrate the efficiency and robustness of our approach.

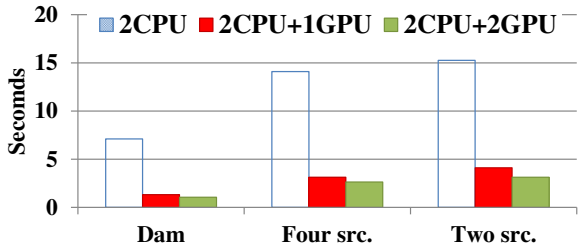

Figure 8: *This figure shows the average time for performing ϵ-NNs per frame (seconds) across the three benchmarks by using a combination of two hexa-core CPUs (2CPU) and one or two GPUs, denoted as 1GPU or 2GPU, respectively,*

5.3 Analysis

One could simply use the CPU with a large main memory space to handle our tested benchmarks. We have also measured performance of ϵ-NN queries by using only two hexa-core CPUs. Our method utilizing a single GPU and CPUs achieves 4.5 times on average and up to 5.4 times performance improvement by using the CPUs only (Fig. 8). While the GPU has a limited video memory, it can achieve high performance because of many parallel units, once GPU is assigned to handle in-core data.

Our method can use additional GPUs. We have added one more GPU, Geforce GTX 780, to our test machine, and let our workload tree to assign blocks to idle GPUs among two of them. Our method in this setting achieves up to 6.8 × (5.7 × on average) improvement over our method running only with two CPUs. While our method is not optimized for utilizing multiple GPUs and thus we do not achieve high scalability, we believe that we open up a direction for improving the large-scale particle based fluid simulations by using heterogeneous computing resources.

Our current SPH solver runs only in the CPU side. In terms of the whole simulation including the SPH solver and ϵ-NNs, we achieve 3.2 times higher performance by using the additional single GPU only for the ϵ-NN part over using only two CPUs for both parts in the dam breaking benchmark. Also, the computation time ratio between ϵ-NNs and the simulation solver becomes about 1:1.1 by using the additional single GPU for ϵ-NNs from about 1:5.8 achieved using only the two CPUs for both simulation and neighbor search. We have not implemented our SPH solver for GPU in this paper, but can be easily parallelized in an in-core GPU mode, since our method decomposes the work in blocks that fit into the video memory of GPU.

6 Conclusion

We have presented an out-of-core technique for ϵ-NN computing used in large-scale particle-based fluid simulation consisting of tens of millions of particles. Our method processes ϵ-NNs based on blocks (i.e., sub-grids) of the uniform grid associated with particles that can fit into the video memory of GPU. Specifically, we have proposed a novel esti-

mation model for the number of neighbors for particles and used the model for estimating the memory footprint required for processing a block based on the workload tree. We have applied our method to three different large-scale particle-based fluid simulations whose memory requirement is much bigger than the video memory of GPUs. Overall our method has shown higher performances over other out-of-core techniques.

Limitations and future work. Our memory estimation method has a high accuracy and did not cause overflows from the auxiliary space in our tested benchmarks. There is, however, no guarantee to avoid such overflows in general. Our workload tree is mainly designed for handling out-of-core particle data, but can be used well even for utilizing CPUs and GPUs. We have also tested our method by adding one more GPU to our currently tested machine configuration, but have observed about 30% improvement. Along this direction, we would like to extend it further for achieving the optimal performance and higher scalability even for parallelization efficiency based on optimization-based scheduling methods [Kim et al. 2013a]. Our current technique adopted a modular approach by decoupling ϵ-NN and simulation parts. As a result, when we have a better module on these two parts, we can achieve higher performance easily, by simply replacing one of existing modules with a better one. However, our current approach assumed that the simulation accesses the simulation grid block by block, which is common practice for accessing them. As a future work, we would like to extend our modular approach to allow random access from the simulation part. Finally, many issues have arisen for rendering and polygonization parts in terms of handling out-of-core data, and addressing them is one of near future research directions.

References

Becker, M., and Teschner, M. 2007. Weakly compressible sph for free surface flows. In *Proc. of ACM SIGGRAPH/EG Symp. on Computer Animation*, 209–217.

Blelloch, G. E., Gibbons, P. B., and Simhadri, H. V. 2010. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures*, 189–199.

Budge, B., Bernardin, T., Stuart, J. A., Sengupta, S., Joy, K. I., and Owens, J. D. 2009. Out-of-core data management for path tracing on hybrid resources. *Comput. Graph. Forum (EG) 28*, 2, 385–396.

Goswami, P., Schlegel, P., Solenthaler, B., and Pajarola, R. 2010. Interactive sph simulation and rendering on the gpu. In *Proc. of ACM SIGGRAPH/EG Symposium on Computer Animation*, 55–64.

Harada, T., Koshizuka, S., and Kawaguchi, Y. 2007. Smoothed particle hydrodynamics on gpus. *In Proc. of Computer Graphics International*, 63–70.

Heo, J.-P., Lee, Y., He, J., Chang, S.-F., and Yoon, S.-E. 2012. Spherical hashing. In *CVPR*.

Ihmsen, M., Akinci, N., Becker, M., and Teschner, M. 2011. A parallel sph implementation on multi-core cpus. *Computer Graphics Forum 30*, 1, 99–112.

Ihmsen, M., Cornelis, J., Solenthaler, B., Horvath, C., and Teschner, M. 2013. Implicit incompressible sph. *Visualization and Computer Graphics, IEEE Transactions on*.

Kim, D., Lee, J., Lee, J., Shin, I., Kim, J., and Yoon, S.-E. 2013. Scheduling in heterogeneous computing environments for proximity queries. *IEEE Transactions on Visualization and Computer Graphics 19*, 9, 1513–1525.

Kim, T.-J., Sun, X., and Yoon, S.-E. 2013. T-ReX: Interactive global illumination of massive models on heterogeneous computing resources. *IEEE Transactions on Visualization and Computer Graphics*.

Li, S., Simons, L., Pakaravoor, J. B., Abbasinejad, F., Owens, J. D., and Amenta, N. 2012. kann on the gpu with shifted sorting. In *Proc. of ACM SIGGRAPH/EG Conf. on High-Performance Graphics*, 39–47.

Lin, M., and Manocha, D. 2003. Collision and proximity queries. *Handbook of Discrete and Computational Geometry*.

Müller, M., Charypar, D., and Gross, M. 2003. Particle-based fluid simulation for interactive applications. In *Proc. of ACM SIGGRAPH/EG Symposium on Computer animation*, 154–159.

NVIDIA, 2013. CUDA programming guide 5.0.

Pan, J., Lauterbach, C., and Manocha, D. 2010. g-planner: Real-time motion planning and global navigation using gpus. In *AAAI*.

Purcell, T. J., Donner, C., Cammarano, M., Jensen, H. W., and Hanrahan, P. 2003. Photon mapping on programmable graphics hardware. In *Proc. of the ACM SIGGRAPH/EG conf. on Graphics hardware*, 41–50.

Qiu, D., May, S., and Nüchter, A. 2009. Gpu-accelerated nearest neighbor search for 3d registration. In *Computer Vision Systems*. Springer, 194–203.

Samet, H. 2006. *Foundations of MultiDimensional and Metric Data Structures*. Morgan Kaufmann.

Solenthaler, B., and Gross, M. 2011. Two-scale particle simulation. *ACM Transactions on Graphics (TOG) 30*, 4, 81.

Solenthaler, B., and Pajarola, R. 2009. Predictive-corrective incompressible sph. *ACM Transactions on Graphics (TOG) 28*, 3, 40.

Yang, J. C., Hensley, J., Grün, H., and Thibieroz, N. 2010. Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum 29*, 4, 1297–1304.

Yoon, S.-E., Gobbetti, E., Kasik, D., and Manocha, D. 2008. *Real-Time Massive Model Rendering*. Morgan & Claypool Publisher.

Zhou, K., Hou, Q., Wang, R., and Guo, B. 2008. Real-time kd-tree construction on graphics hardware. In *SIGGRAPH Asia*, ACM, 1–11.