High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

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

Duksu Kim [1] Myung-Bae Son [1] Young J. Kim [2] Jeong-Mo Hong [3] Sung-eui Yoon [1]

[1] KAIST (Korea Advanced Institute of Science and Technology) [2] Ewha Womans University, Korea [3] Dongguk University, Korea
Project web page: http://sglab.kaist.ac.kr/OOCNNS/

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. Specifically, we identify a maximal work set that a GPU can process efficiently in an in-core mode. As a main technical component, we compute a memory footprint for processing a given work set 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 have demonstrated that our method handles up to 65 M particles and processes up to 15 M ε-NN queries per second by using two CPUs and a GPU, which has only 3 GB video memory. This result is up to 51× higher performance than a single CPU-core version for the out-of-core case. 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.

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 is a performance bottleneck of the simulation in practice. In our SPH simulation based on the method of Becker and Teschner [BT07], we found that neighbor search can take more than 50% of the overall simulation time, when we use a single CPU core. 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 handle such a high computational cost of ε-NN, many parallel techniques based on multi-core CPUs [IABT11] or GPUs [HKK07,GSSP10] have been proposed. Thanks to these recent works, we can achieve high performance improvement (e.g., 20× to 40×) by using a GPU for processing ε-NN queries over a single CPU-core based serial method.

Unfortunately, it has not been actively studied to handle massive-scale ε-NNs for data sets that do not fit in the GPU memory (video memory) for particle-based fluid simulations. For example, Harada et al. [HKK07] reported that GPU can handle about 5 M particles per 1 GB video memory and most commodity-level GPUs have from 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 for the out-of-core case nor directly applicable to such cases.

Contributions. We propose an out-of-core technique utilizing heterogeneous computing resources for processing ε-NNs used in a large-scale 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.

DOI: 10.2312/hpg.20141096

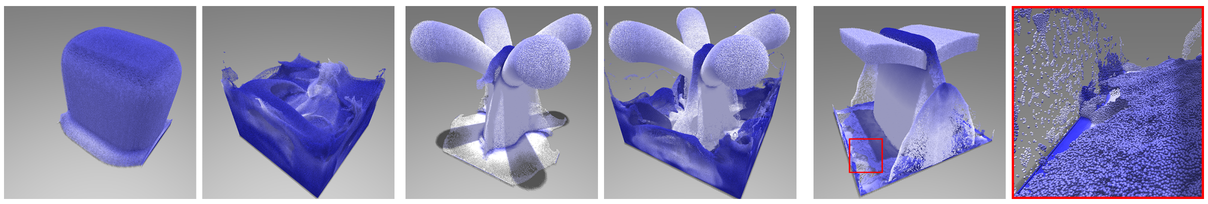

Figure 1: *These figures show three different benchmarks. By using two hexa-core CPUs and a GPU, our ε-NN method takes 1.1 s for the Dam., 2.4 s for the four sources, and 3.6 s for two sources benchmarks on average, respectively.*

We use a uniform grid, a commonly employed acceleration data structure for particle-based 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 transfers 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 identifying 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 (Fig. 1). These benchmarks require up to 16 GB memory space for processing ε-NNs. Our out-of-core method for ε-NNs can process these benchmarks with a GPU that has only 3 GB video memory. Overall, our method can perform up to 15 M ε-NNs per second. We have also implemented an alternative, GPU-based out-of-core approach based on an Nvidia's mapped memory method [NVI13]. Compared to this alternative, our method shows up to 26 × 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 up to 6.3 × improvement using an additional GPU. This result is 51× higher performance compared to using a single CPU core.

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 [MCG03, BT07, SP09, ICS*13, CIPT14, IOS*14].

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 [APKG07, SG11, HS13], but the number can be still high, requiring to run the simulation in an out-of-core manner, to present many details.

2.2. Near Neighbor Search (NNS) and Parallel NNS

NNS is one of the widely used proximity queries and finds points closely placed to a given query point in a metric space [Sam06]. 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.

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 [PDC*03, ZHWG08], 3D registration [QMN09], etc. Unfortunately, these methods are neither directly applicable nor effective to our problem, since our application uses 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 [GSSP10]. 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 the number of particles is limited to the video memory size, e.g., about 5 M for 1 GB video memory [HKK07].

Ihmsen et al. [IABT11] used multi-core CPUs in the whole process of SPH. They showed that a CPU-based parallel approach can handle a larger number of particle (e.g. 12 M) thanks to the large memory space (e.g., 128 GB) in the CPU side. Different with those methods, our method utilize both multi-core CPUs and GPUs. To handle a large number of particles with GPU, a multi-GPU approach was also proposed [DCVB*13]. This approach partitioned the simulation space into multiple GPUs and adopted a MPI based distributed computing among those GPUs. For example, they used four GPUs (GTX480 with 1.5 GB) to simulate 40 M particles. On the other hand, we can handle a large number particles even with a single GPU in an out-of-core manner as long as main memory in CPU can hold all the data. Furthermore, our method runs much faster than the MPI-based distributed computing approach, more suitable for graphics simulations.

2.3. Out-of-Core GPU Algorithms

The limited video memory space raises various challenges for handling a large data set in GPU. The out-of-core issue has been well studied for rendering [YGKM08]. Nonetheless, it has not been actively studied for different parts of particle-based fluid simulations.

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 [NVI13]. 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. [BBS*09] 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. [KSY14] 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.

3. Out-of-Core, Parallel ε-NN

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 [BGS10] on the shared memory model with the same parallel cores. Instead, we propose a simple, hierarchical approach, tailored to our particular problem, that simultaneously computes a job unit that can fit into the video memory of a GPU, while utilizing heterogeneous parallel computing resources.

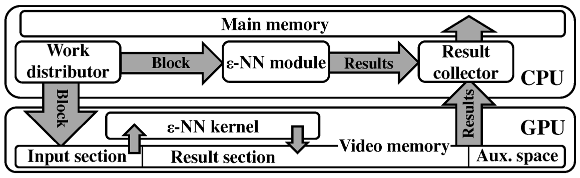

Figure 2: *This figure shows an overall framework for processing ε-NNs in an out-of-core manner using heterogeneous computing resources.*

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.

Fig. 2 shows an overview of our system. We use a uniform grid that are commonly used for accelerating ε-NN, while determining cell indexes with Z-curve to exploit spatial locality [IABT11]. The simulation space is split uniformly into *cells* so that the length of each cell is equal to ε or 2ε. Then, neighboring particles for a query particle with the ε-NN are located in the cell enclosing the query particle or its adjacent cells. Initially, the uniform grid is stored in main memory. As a result, we need to send those cells and their particles from CPU to GPU to perform the ε-NN in the GPU side. We use the term *processing cells* to denote the process of performing ε-NN for particles in the cells.

In the CPU side, *work distributor* divides the uniform grid into sub-grids dynamically and assigns them to available computing resources based on our hierarchical work distribution method (Sec. 3.2). To process data in a cache-coherent mammer, we divide the uniform grid in the form of a cubic sub-grid. A *block* represents a sub-grid and the term *processing block* denotes processing cells in the block.

The *ε-NN module* in CPU and *ε-NN kernel* in GPU receive a block from the work distributor when they are idle. Once ε-NN module or ε-NN kernel finishes to process the block, it pushes the results back to the result collector. Finally, the *result collector* takes the results, stores them in main memory, and returns them to the particle-based fluid simulator.

3.2. Work Distribution

To fully utilize high performance GPUs in an out-of-core manner, we divide the grid such that the size of the working set of each block should be smaller than the size of video memory.

Processing a block requires to access particles and to write information of their identified neighbor particles. 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:

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

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. s_p requires 12 bytes to encode the particles position. In addition we need a minor space to store sub-grid data, 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. A naive approach is a two-pass algorithm: it measures the required memory space without writing result at the first pass and re-processes the queries while storing the result in the second pass. Although we can exactly measure n_{p_i}, but this approach has redundant operations. Another alternative is to use a general vector-like data structure that adaptively grows according to identified neighbors [YHGT10]. This data structure, unfortunately, is not designed for the out-of-core case and thus fails, 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 (Sec. 4). When n'_{p_i} is the expected number of neighbors for the i-th particle, $s(B)$ becomes as following:

$$s(B) = n_B s_p + s_n \sum_{p_i \in B} n'_{p_i} + s_{Aux}. \quad (2)$$

In the above equation, we introduce a new term, s_{Aux}, the size of an auxiliary space, which is an additional space to handle the error related to our estimation process, and its size is computed depending on n_B and the estimation error; its details are given in Sec. 4. For example, a block of 32^3 has up to about 1.3 M particles, and 18 expected neighbors per particle on average across our tested benchmarks. $n_B s_p$, $s_n \sum_{p_i \in B} n'_{p_i}$, and s_{Aux} then take 15.6 MB, 187 MB, and 38.5 MB, respectively. Thanks to this memory estimation process, we can efficiently find a block that fits in the video memory and perform ε-NNs for the block in GPU without any memory I/O thrashing.

Once we know the required memory space for processing a block, the next task is to divide the uniform grid into blocks. Simply, we can use a small fixed size (e.g., 8^3) of blocks so that each block requires less memory space than the size of the video memory. We found that a larger block size (e.g., 64^3) shows better performance, but it is hard to find a maximal block size for each simulation in practice, since the maximal block size that GPU can handle varies depending on the simulation state and regions of the simulation space (Sec. 5.3).

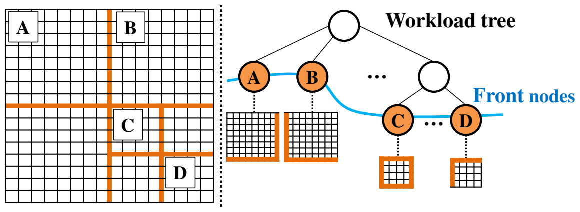

Figure 3: *The left figure shows a uniform grid with a few sub-grids; boundaries of these sub-grids, i.e., blocks, are shown in orange lines. The right figure shows an example of the workload tree with these blocks.*

Hierarchical work distribution. To efficiently construct such a maximal block, we use a *workload tree*, which is an octree built on 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.

The work distributor running on a CPU thread finds the maximal blocks that the GPU video memory can hold by traversing the workload tree, and give those blocks to GPU. The detail work flow of our hierarchical work distribution is summarized in Algorithm 1. The work distributor maintains a *front node queue* containing blocks that are candidates for maximal blocks. Initially, the front node queue contains the root node of the workload tree. When a GPU has available video memory space, the work distributor takes the front node and checks whether the available space is larger than the required memory size for processing the block in the node. If we have enough available space in the GPU, the work distributor assigns the block to the GPU. Otherwise, we also check whether the block size is bigger than the maximum size of the GPU video memory. If the required memory size is larger than the maximum size of video memory, we have to process its child blocks and thus the distributor enqueues its eight child nodes to the queue. If the required memory size of the block is smaller than the maximal video memory size, we decide not to process it at the current time, and thus push the node back to the queue, which will be processed later when the GPU has enough space for the block. Based on this simple, hierarchical process, we efficiently identify maximal blocks that can be processed simultaneously in the GPU, while utilizing the available video memory.

Boundary handling. There are two types of cells in a subgrid; the cells at the boundary of a block (*boundary cells*)

Algorithm 1 Our hierarchical work distribution algorithm

```
$Queue_{node}$ ← push the root node of the workload tree
Repeat until $Queue_{node}$ is not empty
if ($G_{remain}$ ← the remaining size of video memory) $\geq 0$ then
  $B$ ← pop a node from $Queue_{node}$
  if ($s(B)$ ← required memory size to process $B$) $\leq G_{remain}$ then
    Give B to the GPU
  else
    if $s(B) > (G_{max}$ ← the maximal video memory size$)$ then
      $Queue_{node}$ ← push the eight children of $B$
    else
      $Queue_{node}$ ← push $B$
    end if
  end if
end if
```

and other cells are *inner cells* (Fig. 3). Usually, the number of boundary cells is much smaller than that of inner cells, because boundary and inner cells exist in 2D and 3D space, respectively. To find neighbors of boundary cells of a block we need to access cells in its adjacency block and it requires a larger working set over handling inner cells, resulting in a lower locality. As a result, we let the CPU cores to process those boundary cells, since main memory already has all data and CPU can efficiently handle random memory access thanks to the well-organized caches. On the other hand, we let a high-performance GPU to focus on processing a large number of inner cells.

3.3. Processing a Block in GPU

When a block is given to a GPU, we create a new stream to perform the data transfer for concurrently processing other blocks and hiding the data transfer overhead. We then configure work space in the video memory. The work space is decomposed into input, result, and auxiliary space sections (Fig. 2). Each section reserves space based on each term of Eq. 2: $n_B s_p$ for the input section, $s_n \sum_{p_i \in B} n'_{p_i}$ for the result section, and s_{Aux} for the auxiliary space, respectively. Each query particle then receives the estimated, fixed amount memory space, and each GPU thread processes an ε-NN for the query particle in the block. When a GPU thread identifies neighbors for the query particle, it stores them in its pre-defined, corresponding space in the result section without any synchronization.

We may need further memory spaces than the pre-defined result section, due to the inaccuracy of our estimation process. In this overflow 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 with the tested benchmarks. Our approach of 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.

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. 2) for processing blocks is to compute the expected number of neighbors, n'_{p_i}, of a particle. We estimate it 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, $S(p,\varepsilon)$, 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 [SP09, BT07] 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 $S(p,\varepsilon)$ 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:

$$E(p_{x,y,z}) = \sum_i n(C_i) \frac{Overlap(S(p_{x,y,z},\varepsilon),C_i)}{V(C_i)}, \qquad (3)$$

where C_i indicates the cell containing $p_{x,y,z}$ and its adjacent cells that have any overlap 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. $Overlap(S(p_{x,y,z},\varepsilon),C_i)$ represents the overlap volume between $S(p_{x,y,z})$ and C_i.

This equation requires us to compute $Overlap(S(p_{x,y,z},\varepsilon),C_i)$ 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

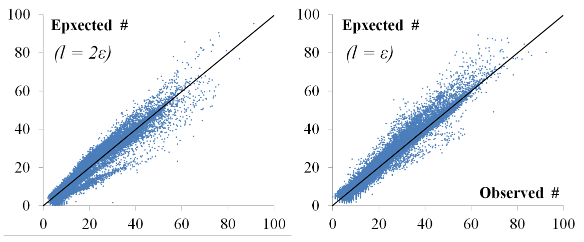

Figure 4: *This plot shows the expected and observed number of neighbors in two configurations, $l = 2\varepsilon$ (the left) and $l = \varepsilon$ (the right), for the dam breaking benchmark. The linear (Pearson product-moment) correlation coefficients are 0.97 and 0.96, respectively.*

of neighbors. The average, expected number of neighbors of particles $E(C_q)$ in a cell C_q is then defined as:

$$\begin{aligned} E(C_q) &= \frac{1}{V(C_q)} \int_0^l \int_0^l \int_0^l E(p_{u,v,w})\, \mathrm{d}u\mathrm{d}v\mathrm{d}w \\ &= \frac{1}{V(C_q)} \sum_i n(C_i) \frac{D(C_q,C_i)}{V(C_i)}, \end{aligned} \tag{4}$$

where $D(C_q,C_i) = \int_0^l \int_0^l \int_0^l Overlap(S(p_{u,v,w},\varepsilon),C_i)\mathrm{d}u\mathrm{d}v\mathrm{d}w$, l is the length of a cell along each dimension, and $p_{u,v,w}$ is a particle p positioned at (u,v,w) on a local coordinate space in C_q. 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. As a result, we pre-compute these values in an offline manner (taking a few seconds). Specially we use 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. 4 by considering $n(C_i)$ and looking up pre-computed $D(C_q,C_i)$ values, which are 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 from 100 to 500 ms at each frame on average with our tested benchmarks consisting of up to 65 M particles.

4.2. Validation and Error Handling

We have measured the accuracy of our expectation model with our tested benchmarks. Fig. 4 shows the scatter plots between the expected number of neighbors $E(C_q)$ and the actual number of neighbors that are computed after finishing ε-NN for each particle. We use two most common configuration for computing the uniform grid and they have strong linear correlation (e.g., 0.97). We achieve such a high accuracy by considering the number of particles in cells, while assuming the local uniform distribution of particles. We have observed similar results with other tested models.

Additionally, we have measured the root mean square error (RMSE) between the estimated and observed ones. In our tested benchmark the RMSE is measured up to 3.7 (for four source benchmark), and indicates that our estimated number of neighbors can be higher or lower by 3.7 on average to the actual number of neighbors. This information is useful for estimating the required space for the auxiliary space. We need to access the auxiliary space to accommodate underestimation and the underestimation error is less than 3.7 in most case. Based on this analysis, we set the size of the auxiliary space s_{Aux} in Eq. 2 as $3.7 * (n_B s_n)$.

5. Results and Analysis

We have implemented and tested our out-of-core parallel ε-NN method in a machine consisting of a GPU (Nvidia Geforce GTX 780, 3 GB) and two Intel Xeon hexa-core CPUs (2.93GHz) with 192 GB main memory. We use 2.8 GB of 3 GB video memory for all 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 ε-NN module on multi-core CPU based on a prior method [IABT11] and use 12 threads for the module. For ε-NN kernel in GPU, we have implemented a locality-aware GPU algorithm based on a prior in-core GPU algorithm [GSSP10]. We have implemented a vector data structure for the auxiliary space in GPU by using the atomic operation [YHGT10].

Our simulation method [BT07] has been implemented on multi-core CPUs based on Ihmsen et al. [IABT11]. We first perform ε-NNs and pass their results to the simulation solver, which moves particles based on the computed neighbor search results. We set the cell length of the uniform grid as the two time of search radius, i.e., $l = 2\varepsilon$, since it is one of commonly adopted choices in practice [Hoe09]. The grid resolutions of our benchmarks are then 128^3 for dam breaking and four source benchmarks and 256^3 for the two source benchmark.

To compare the efficiency and robustness of our out-of-core system, we have implemented two alternative methods:

- *IC-GPU* has been implemented by removing all out-of-core features from our method. This method stores all results in a vector structure designed for GPU [YHGT10] like the auxiliary space in our system. All available GPU memory is used as a memory pool for the vector structure.
- *Map-GPU* method uses Nvidia's mapped memory techniques. 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 ε-NN search results.

We have also implemented a simple workload distribution method to measure the benefit of our hierarchical approach:

- *Fixed-Block* method divides the uniform grid into the fixed size (e.g., 16^3) of blocks and assigns blocks to GPU, while using multiple streams to hide the data transfer overhead. This method is also tested by reserving a maximal memory space (*Fixed-Block(Max)*), or by reserv-

	Dam (Fig. 1(a))	Four src. (Fig. 1(b))	Two src. (Fig. 1(c))
Max. # of pts.	15.8 M	32.7 M	65.6 M
Max. data size	5.7 GB	15.5 GB	13.0 GB
Avg. n_{p_i} / Max. n_{p_i}	15.4 / 184	26.1 / 489	11.0 / 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. The max. data size is the maximum $s(B)$ among all frames, where B is the whole grid.*

ing the memory space based on our memory estimation method (*Fixed-Block(Exp)*).

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. 1(a)). 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. 1(b) and Fig. 1(c)). These benchmarks are available at our project webpage 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. When we attempt to process the whole uniform grid in a single block, these three benchmarks require 6 GB to 16 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.

5.1. Results

Fig. 5 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 alternative methods.

As long as all the data fit into the video memory of a GPU, we can use the in-core method to perform all ε-NN queries within the GPU. Based on our memory estimation model, our approach determines maximal blocks that fit into the GPU video memory and thus uses the in-core ε-NN processing algorithm with those blocks. When we have the small number of particles (e.g., 5 M), our method shows 60% lower performance on average than *IC-GPU*, since our method generates the workload tree to estimate the memory footprint; it takes from 100 ms to 500 ms in CPU and it is relatively a high overhead in the case with the small number of particles. 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 required memory size exceeds the size of video memory. Our method continues to process larger particles in an out-of-core manner, while *IC-GPU* fails to perform ε-NNs.

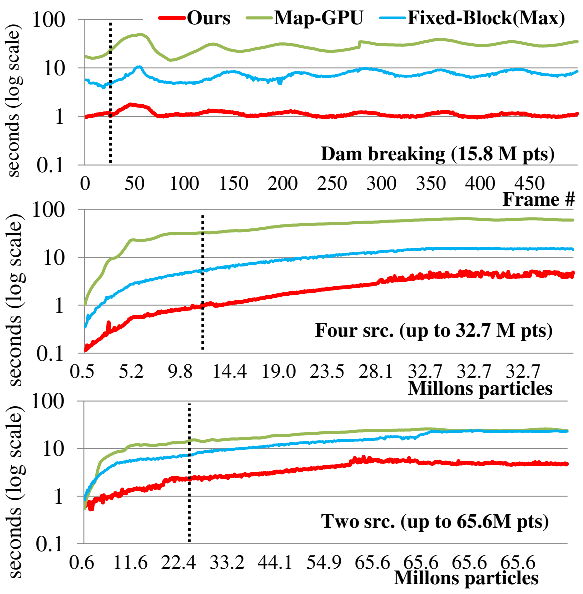

Figure 5: *These graphs show the processing time in the log scale for ε-NNs based on different out-of-core methods including ours. The measured time includes data communication time for sending input data and copying the results to the main memory. Starting from the dotted lines, our method runs in an out-of-core manner.*

We have compared our method with *Map-GPU* to see the efficiency of our out-of-core approach. Our method achieves higher performances: 26× for the Dam, 18× 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 available, 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*.

Fig. 5 also compares the performance of ours and *Fixed-Block(Max)*, which reserves a maximal memory space, i.e., 250 neighbors, for each particle, and uses 16^3 blocks for all the benchmarks. When the number of neighbors exceed the maximum, it uses the auxiliary space in the video memory. Our method achieves 6× for the Dam., 4× for the four sources, and 4× for two sources benchmark higher performance over *Fixed-Block(Max)* on average. This result demonstrates the benefits of our hierarchical work distribution method based on our memory estimation model.

Comparison with CPU-based ε-NN. One could simply use

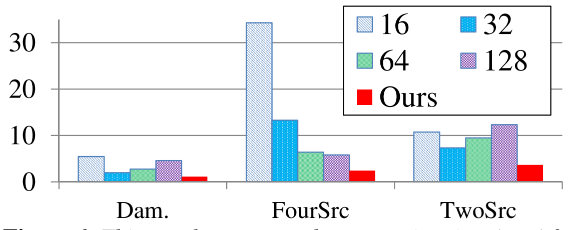

Figure 6: *This graph compares the processing time (sec.) for ε-NNs with various space sizes and our estimation model for recording results within our method.*

the CPU with a large main memory space to handle our tested benchmarks. We have implemented the state-of-the-art parallel ε-NN for particle-based simulation [IABT11] and compared the performance with ours. Compared with using a single CPU-core, the parallel CPU algorithm using 12 cores shows 8.4× higher performance on average. With an additional GPU, our method achieved 5.4× on average and up to 6.3× performance improvement over the parallel CPU algorithm. This is 46× on average and up to 51× higher performance compared with using a single CPU core. Please note that the performance improvement we report in the paper is computed by including the data transfer overhead between CPU and GPU.

5.2. Benefits of Our Memory Estimation Model

To measure benefits from our estimation model, 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. Fig. 6 shows the average processing time with various fixed space sizes within our method and our method with the estimation model in the benchmarks. Overall, our system equipped with our estimation method achieves much higher performance, 1.4 × to 14.2 × over the tested fixed neighbors.

For a small fixed space (e.g., 16 or 32), we found that some of the identified neighbors have to be recorded in main memory through expensive PCI-Express communication, and thus it can drop down the performance significantly, especially for the four sources benchmark. For a large space (e.g., 64 or 128) the transaction to main memory happens rarely, but we observe low GPU utilizations, especially in the two sources benchmark, since we cannot send many particles to GPU. 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 achieving high GPU utilizations. These results demonstrate the effectiveness and robustness of our approach.

5.3. Benefits of Hierarchical Workload Distribution

To measure benefits brought by our hierarchical work distribution approach, we have tested the performance of *Fixed-Block(Exp)* that reserves spaces according to our memory estimation method, not to the maximum number of neighbors (e.g., 250). We have tested 16^3, 32^3, and 64^3 as the fixed block size and found that a larger block size shows a better performance, as long as these blocks can fit into the video memory. For example, 32^3 and 64^3 block sizes show 1.5 and 1.8 × higher performance compared with using 16^3 blocks in the four source benchmarks, respectively. When we measure the GPU processing time only, 32^3 and 64^3 block sizes take 22% and 30% less times than using 16^3 blocks on average. This is mainly because a large block can better utilize the massive parallel nature of GPU architecture.

The maximal block size, unfortunately, varies depending on the benchmarks. For achieving the best performance in each benchmark, we have to manually set 32^3 for Dam. and two sources, and 64^3 for two source benchmarks. On the other hand, our hierarchical work distribution with our memory estimation method finds the optimal block size dynamically without those manual tunning. Furthermore, our hierarchical workload distribution method achieved 33% higher performance on average across all the tested benchmarks compared with *Fixed-Block(Exp)*, which reserves memory with our memory estimation method, but uses the manually chosen, best block size for each benchmark.

6. Conclusion and Discussion

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. Specifically, we have proposed a novel estimation 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.

Implicit and incompressible SPHs. In this paper, we have used an explicit method for the particle-based simulation, i.e., WCSPH [BT07]. In implicit methods like PCISPH and IISPH [SP09, ICS*13], the relative computational overhead of the neighbor search step can be reduced, since implicit methods can tolerate larger time steps. Nonetheless, ε-NN is still a basic operation even in these implicit methods, and the neighbor search step is also recommended to be used in the convergence iterations for more accurate simulations. As a result, the proposed approach can improve the performance of these implicit methods. We leave to verify this as one of future directions.

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 prevent such overflows in general. We have 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 [KLL*13]. 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, we have not shown other applications of our method in this paper, but our work can be applied to other applications such as photon mapping [KSY14]) using a volumetric grid.

Acknowledgements

This work was supported in part by MCST/KOCCA/CT/R&D 2011, NRF-2013R1A1A2058052, DAPA/ADD (UD110006MD), MEST/NRF (2013-067321), and IT R&D program of MOTIE/KEIT [10044970]. Young J. Kim was supported in part by NRF in Korea (No. 2012R1A2A2A01046246, No. 2012R1A2A2A06047007). Prof. Yoon is a corresponding author.

References

[APKG07] Adams B., Pauly M., Keiser R., Guibas L. J.: Adaptively sampled particle fluids. In *ACM Transactions on Graphics* (2007), vol. 26, p. 48. 2

[BBS*09] Budge B., Bernardin T., Stuart J. A., Sengupta S., Joy K. I., Owens J. D.: Out-of-core data management for path tracing on hybrid resources. *Comput. Graph. Forum (EG) 28*, 2 (2009), 385–396. 3

[BGS10] Blelloch G. E., Gibbons P. B., Simhadri H. V.: Low depth cache-oblivious algorithms. In *ACM Symp. on Parallelism in Algorithms and Architectures* (2010), pp. 189–199. 3

[BT07] Becker M., Teschner M.: Weakly compressible SPH for free surface flows. In *Proc. of ACM SIGGRAPH/EG Symp. on Computer Animation* (2007), pp. 209–217. 1, 2, 5, 6, 8

[CIPT14] Cornelis J., Ihmsen M., Peer A., Teschner M.: IISPH-FLIP for incompressible fluids. *Computer Graphics Forum (Proc. Eurographics)* (2014). 2

[DCVB*13] Domínguez J. M., Crespo A. J., Valdez-Balderas D., Rogers B., Gómez-Gesteira M.: New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters. *Computer Physics Communications 184*, 8 (2013), 1848–1860. 3

[GSSP10] Goswami P., Schlegel P., Solenthaler B., Pajarola R.: Interactive SPH simulation and rendering on the GPU. In *Proc. of ACM SIGGRAPH/EG Symposium on Computer Animation* (2010), pp. 55–64. 1, 2, 6

[HKK07] Harada T., Koshizuka S., Kawaguchi Y.: Smoothed particle hydrodynamics on GPUs. *In Proc. of Computer Graphics International* (2007), 63–70. 1, 2

[Hoe09] Hoetzlein R.: FLUIDS v.2 - a fast, open source, fluid simulator. http://www.rchoetzlein.com/eng/graphics/fluids.htm, 2009. 6

[HS13] Horvath C. J., Solenthaler B.: Mass preserving multi-scale SPH. *Pixar Technical Memo 13-04* (2013). 2

[IABT11] Ihmsen M., Akinci N., Becker M., Teschner M.: A parallel SPH implementation on multi-core CPUs. *Computer Graphics Forum 30*, 1 (2011), 99–112. 1, 2, 3, 6, 8

[ICS*13] Ihmsen M., Cornelis J., Solenthaler B., Horvath C., Teschner M.: Implicit incompressible SPH. *Visualization and Computer Graphics, IEEE Transactions on* (2013). 2, 8

[IOS*14] Ihmsen M., Orthmann J., Solenthaler B., Kolb A., Teschner M.: SPH fluids in computer graphics. *State-of-the-Art Report, Eurographics* (2014). 2

[KLL*13] Kim D., Lee J., Lee J., Shin I., Kim J., Yoon S.-E.: Scheduling in heterogeneous computing environments for proximity queries. *IEEE Transactions on Visualization and Computer Graphics 19*, 9 (2013), 1513–1525. 9

[KSY14] Kim T.-J., Sun X., Yoon S.-E.: T-ReX: Interactive global illumination of massive models on heterogeneous computing resources. *IEEE Transactions on Visualization and Computer Graphics 20*, 3 (2014), 481–494. 3, 9

[MCG03] Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. In *Proc. of ACM SIGGRAPH/EG Symposium on Computer animation* (2003), pp. 154–159. 2

[NVI13] NVIDIA: CUDA programming guide 5.0, 2013. 2, 3

[PDC*03] Purcell T. J., Donner C., Cammarano M., Jensen H. W., Hanrahan P.: Photon mapping on programmable graphics hardware. In *Proc. of the ACM SIGGRAPH/EG conf. on Graphics hardware* (2003), pp. 41–50. 2

[QMN09] Qiu D., May S., Nüchter A.: Gpu-accelerated nearest neighbor search for 3d registration. In *Computer Vision Systems*. Springer, 2009, pp. 194–203. 2

[Sam06] Samet H.: *Foundations of MultiDimensional and Metric Data Structures*. Morgan Kaufmann, 2006. 2

[SG11] Solenthaler B., Gross M.: Two-scale particle simulation. *ACM Transactions on Graphics 30*, 4 (2011), 81. 2

[SP09] Solenthaler B., Pajarola R.: Predictive-corrective incompressible SPH. *ACM Transactions on Graphics 28*, 3 (2009), 40. 2, 5, 8

[YGKM08] Yoon S.-E., Gobbetti E., Kasik D., Manocha D.: *Real-Time Massive Model Rendering*. Morgan & Claypool Publisher, 2008. 3

[YHGT10] Yang J. C., Hensley J., Grün H., Thibieroz N.: Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. 4, 6

[ZHWG08] Zhou K., Hou Q., Wang R., Guo B.: Real-time kd-tree construction on graphics hardware. In *SIGGRAPH Asia* (2008), ACM, pp. 1–11. 2