Efficient Simulation of Nested Hollow Sphere Intersections

for Dynamically Nested Compartmental Models in Cell Biology

Till Köster
University of Rostock
Institut für Informatik
Albert-Einstein-Straße 22
Rostock, Germany 18059
till.koester@uni-rostock.de

Kalyan Perumalla
Oak Ridge National Laboratory
One Bethel Valley Rd
Oak Ridge, Tennessee, 37831-6085, USA
perumallaks@ornl.gov

Adelinde Uhrmacher
University of Rostock
Institut für Informatik
Albert-Einstein-Straße 22
Rostock, Germany 18059
adelinde.uhrmacher@uni-rostock.de

ABSTRACT

In the particle-based simulation of cell-biological systems in continuous space, a key performance bottleneck is the computation of all possible intersections between particles. These typically rely for collision detection on solid sphere approaches. The behavior of cell biological systems is influenced by dynamic hierarchical nesting, such as the forming of, the transport within, and the merging of vesicles. Existing collision detection algorithms are found not to be designed for these types of spatial cell-biological models, because nearly all existing high performance parallel algorithms are focusing on solid sphere interactions. The known algorithms for solid sphere intersections return more intersections than actually occur with nested hollow spheres. Here we define a new problem of computing the intersections among arbitrarily nested hollow spheres of possibly different sizes, thicknesses, positions, and nesting levels. We describe a new algorithm designed to solve this nested hollow sphere intersection problem and implement it for parallel execution on graphical processing units (GPUs). We present first results about the runtime performance and scaling to hundreds of thousands of spheres, and compare the performance with that from a leading solid object intersection package also running on GPUs.

Keywords

Neighbor Lists, Nearest Neighbors, Cellular Biology, Compartmental Models, Nested Shells, Smooth Particles, GPU, CUDA, Prefix Sum

1. INTRODUCTION

1.1 Background and Motivation

Particle-based reaction-diffusion models in continuous space are computationally very expensive. One possibility to speed up simulation is parallelization [9]. Over the past decade, general purpose graphical processing units (GPU)-based computation has been effectively exploited for simulation [28, 22, 25], and for simulating cell biological systems in particular [5]. Models that are executed on the GPU are typically composed of many individual entities with similar behavior patterns, where the simulation proceeds stepwise and the calculations per simulation step are quite similar [24]. Thus, individual particles moving stepwise [8, 4, 26] or stochastic reaction-diffusion algorithms based on operator-splitting, lend themselves very well to GPU-based execution [10].

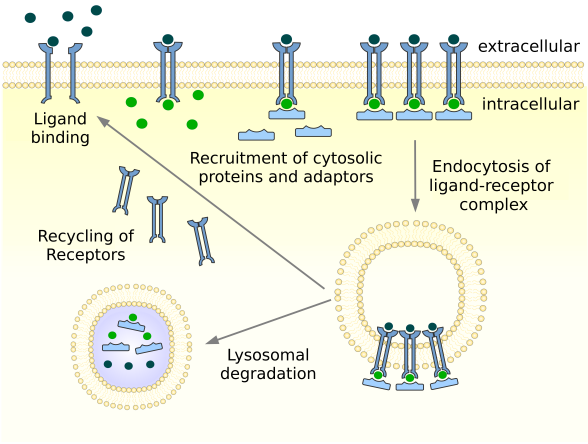
To take excluded volumes and crowding into account in particle-based reaction-diffusion models [30], sub-cellular components may be abstracted as spherical in shape of different radii, with a diffusion-excluded interior [32]. The core computational problem concerns the proximity-based interaction of many sub-cellular components [30]. If a particle attempts a move, it is checked whether the particle will as a result of the move overlap with another particle and, if so, the move will be rejected. Other tools rely on energy landscapes to model the interaction of particles. Independently, whether the particles' interactions are modeled by move rejection [14, 27] or by potentials (energy landscape) [29], typically they are assumed to be *solid* (hard-core or soft-core) spheres.

However, abstracting them as solid spheres ignores the dynamic nesting of cell-biological systems that plays a key role in the functioning of the cell. In addition to static compartments, such as the nucleus in eucaryotes, we find dynamic compartments and vesicles, like mitochondria and lysosomes, whose number, volume, content, and interconnectivity changes over time. This is illustrated in Figure 1 [23], which shows the process of receptor recycling in the cell schematically. Vesicles form at the membrane and engulf protein receptor complexes, those are transported to a lysosome. The vesicle merges with the lysosome. Part of the receptor complex is freed and returns to the membrane (being recycled) whereas the other part of the receptor complex is degradated within the lysosome.

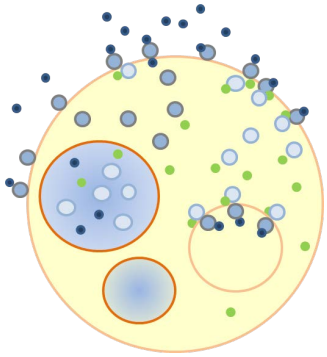
The forming, merging, and destruction of vesicles lead to dynamically and arbitrarily nested compartmental structures. These require the models to treat cells, compartments, and vesicles as *hollow* spheres that allow an arbitrary dynamic nesting of spheres within each other [3]. In cell biological systems of interest to us, the nesting is such that there are a small number of large-sized compartments

SIGSIM-PADS'17 May 24-26, 2017, Singapore, Singapore

DOI: http://dx.doi.org/10.1145/3064911.3064920



(a) Biological view



(b) Hollow spheres view

Figure 1: Illustration of a cellular subsystem model whose components can be specified as nested arrangements of objects within objects that interact at intersection points and can be approximated to spherical objects/shells of various sizes and thicknesses moving within and through each other

such as nucleus, cytosol, or membrane, and a larger number of smaller-sized compartments, such as mitochondria or lysosomes interacting with many small objects like proteins.

Additional applications in which nested containment in hollow spheres is used include simulations of friction-based contacts among many objects ("intersection" of a range) for estimating the dissipated energy in rotating containers [31]. To meet desired configurations containing 200,000 particles, parallel simulation is warranted. Other peripherally related applications include simulation of foams and foam-packed materials.

1.2 Contributions

In this paper, we identify core computational problem in such cellular simulations in terms of a new, formally-defined and generalized problem of arbitrarily nested configurations of hollow spheres or shells. The determination of intersections is a crucial step in such simulations to model the interactions. The determination of interaction activity due to intersections forms a major, time-intensive part of cellular simulations.

We analyze the nested sphere problem in detail with the range of possible scenarios of nesting levels, sphere densities, sphere radii, thicknesses, and position distributions. Based on this problem specification, we propose a new, efficient solution for fully arbitrary nested configurations. We present a novel algorithm that is efficient and scalable to large numbers of nested hollow spheres with widely varying radii and thicknesses. We have completed an efficient implementation of the algorithm on the GPU (implemented in CUDA and run on a NVIDIA K80 card) and have tested correctness. The implementation addresses multiple GPU-specific issues, the most challenging of them being the variable loading per GPU thread. We present runtime performance data of the new algorithm on multiple benchmarks exercising a range of scenarios with varying sizes and positions of large numbers of spheres.

The algorithm and results presented here are among the first in the literature on the definition of the novel problem in nested spherical body intersections, along with algorithm, implementation, and performance data tested in a experimental design with variation of many scenario parameters. The algorithm and implementation have also been integrated into a cell biological simulator that has been sped up with our algorithm to simulate scenarios with higher scale. A separate experimental study is underway in analyzing the runtime gains of the overall application, which are planned for a subsequent publication. In this paper, we focus on the computation of the underlying general problem of computing nested hollow sphere intersections.

1.3 Related Work

The problem of collision detection (also called intersection detection) is well explored in the literature [15, 19, 7], predominantly studied in the context of solid, non-nested bodies. When interested in the collisions of n objects, the naïve approach would be to simply perform a brute force pass over all the objects, checking each object for potential intersection with all other objects. However, this leads to an execution time of $O(n^2)$, which is impractical for large systems. At this point, we can exploit the fact that all potential object-object intersections exist only at the scales of the particle sizes rather than at the full domain length scales. As identified by Hubbard [12], most state of the art collision detection algorithms consist of two phases, a broad phase and a narrow phase, as follows.

Broad Phase In this phase, a rough measure is used to separate *potential* collisions (of objects that are in some vicinity of one another) from *non-potential* collisions (of object pairs that are too far from one another to intersect).

Narrow Phase In this phase, the *potential* pairs of objects identified in the Broad Phase are more closely investigated. In computer graphics, this phase often involves computationally intensive mesh calculations, as the objects tend to have complex three dimensional geometrical structures defined by triangle meshes.

In this paper, we focus on the broad phase due to the simplicity of geometries typically assumed in our immediate application of interest, namely, particle-based reaction-diffusion simulation. We are interested in a rough pruning of the n^2 set of pairs for large number of objects. After the broad phase, the pruning of this set during the narrow phase can be rather easily done with our type of objects (see Section 2.1), compared to the sometimes very complex

triangle meshes as found in other applications such as visualization, computer gaming, and computer graphics. Consequently, we describe different broad phase algorithms here. Specifically, we will be examining algorithms that are suited for execution on the GPU, which is a power efficient parallel processing hardware architecture for simulating systems with large numbers of particles.

Fixed radius on grid

There exists a very fast and refined solution for a particular sub-problem, that of equal-sized (or very similarly-sized) spheres. This fixed radius collision problem has been investigated and well understood, because a consistent spherical cutoff is commonly encountered in many simulation contexts, such as smooth particle hydrodynamics and molecular dynamics simulations [6].

In the state of the art algorithm [11] called *fast Fixed Radius Nearest Neighbor* (fFRNN), the system is organized into bins such that each particle, after being assigned to its own bin, only needs to check the bins in its Moore neighborhood in order to find all possible neighbors. This works only when the interaction cutoff between two particles is ensured to be smaller than the bin width. Moreover, solidity of objects is implicit.

Grids for variable sizes

One way to approach the problem of finding intersections among a number of complex shaped particles is to allow one particle to span across multiple bins in the grid. This is achieved in some approaches [20] by approximating all particles as boxes spanning over multiple bins (easy to calculate for the axis aligned boundary box (AABBs)). AABB means a rectangle or cuboid whose edges are aligned with the x, y and z axis of the underlying coordinate system. Object identifiers are added to all the bins. For each bin, all possible collisions are calculated. Finally, the redundantly counted collisions need to be removed.

Sweep and prune

The sweep-and-prune (SAP), also called sort-and-sweep, algorithm originates in works by Baraff [2]. The idea is to project the objects onto one linear axis. All objects then calculate the starting and the ending points of their projection onto that axis. The objects are then sorted by both these values. If one object's starting point or ending point lies within the starting and ending interval of another object, these two objects potentially have a collision. Projection is then repeated for all axes (for example, the three axes for 3D space). If two objects have overlapping intervals for all tested projections, they are considered a possible collision pair.

An advantage of this algorithm is that the axis projections do not completely change, if the system is only minutely altered. Specific sorting algorithms are employed, which are efficient for sorted lists (such as insertion sort). Numerous optimizations exist, such as using Principal Component Analysis (PCA) for determining the axes. The SAP method cannot immediately be ported to the GPU, as there are some naturally sequential parts (mostly contained in the sweep phase). After some modifications, an efficient GPU variant has been developed [18]. This however only works for small (< 64K) system sizes, as SAP does not cope well with very large systems. Scaling for large systems has been achieved in their implementation by introducing coarse bins for the system that are independently "swept and pruned."

Bounding volume hierarchy trees

A more complex approach to the collision problem is that of a bounding volume hierarchy. The underlying data structure of this approach is a tree (usually a binary tree), whose leaves are the objects of the system. Every other node of the system contains a volume. The tree is built in such a manner that all child node's volume is contained in their parent node's volume. These trees can be built with varying quality (e.g. how balanced they are). For applications in which the system is static and the tree is frequently traversed, such as in ray tracing, higher quality trees are used. For our application, we need to frequently rebuild the tree. This can be done efficiently using the algorithm proposed by Karras [13] which is based on the M-Code [16]. Lauterbach also presents an efficient way to traverse the tree on the GPU [17].

1.4 Organization

The rest of the paper is organized as follows. We describe the hollow sphere/shell intersection problem in Section 2, followed by a presentation of our new hollow sphere/shell intersection algorithm in Section 3. Section 4 describes the parallel version of the algorithm and presents the details of its implementation on GPUs. A detailed performance study is presented in Section 5. Finally, Section 6 concludes the paper with a summary of the results and findings, and outlines future work.

2. HOLLOW SPHERE INTERSECTION

In what follows, we will refer to the solid sphere intersection problem as SSX and the nested hollow sphere intersection problem as HSX. Also, in the context of HSX, we will use the terms spheres and shells interchangeably.

2.1 Definition and Description

The HSX problem is concerned with finding all pairs of spheres that are overlapping and can be formally defined as follows (vectors are denoted in bold font).

A single hollow sphere object h_i is defined by its position $\boldsymbol{x}_i$, its radius r_i, and its shell thickness q_i ($q_i = 0$ is a valid choice). Note, that an sphere's radius marks the outer perimeter, while a sphere around $\boldsymbol{x_i}$ with radius $r_i - q_i$ marks the inner perimeter of the hollow sphere.

This allows for the definition of the following three relations.

- Non-commutative *contains* relation $C(j, i)$ that specifies whether shell h_j is completely contained within shell h_i:

$$C(j,i) = \begin{cases} true & \text{if } |\boldsymbol{x}_i - \boldsymbol{x}_j| < r_j - q_j - r_i \\ false & \text{otherwise} \end{cases} \quad (1)$$

- Commutative *contains* relation $\hat{C}(j, i)$ that specifies whether one of the shells h_j and h_i completely contains the other:

$$\hat{C}(j,i) = C(j,i) \vee C(i,j), \text{ and} \quad (2)$$

- Commutative *intersection* relation $I_{\text{HSX}}(j, i) = I(j, i)$

that specifies if shells h_i and h_j intersect:

$$I(j,i) = \begin{cases} true & \text{if } \neg\hat{C}(j,i) \wedge |\boldsymbol{x}_i - \boldsymbol{x}_j| \leq r_j + r_i \\ false & \text{otherwise} \end{cases} . \tag{3}$$

The HSX problem can now be defined as: given a set of n hollow spheres, compute the set of unique pairs of intersecting spheres $\{\langle j,i\rangle \,|I(j,i) = true\}$.

2.2 HSX versus SSX

The SSX problem only differs from HSX in the definition of the intersection relation. In SSX, the commutative intersection relation $I_{\text{SSX}}(j,i)$ is defined as

$$I_{\text{SSX}}(j,i) = \begin{cases} true & \text{if } |\boldsymbol{x}_i - \boldsymbol{x}_j| \leq r_j + r_i \\ false & \text{otherwise} \end{cases} . \tag{4}$$

Since the HSX intersection criterion is stricter than that of SSX, it follows that an HSX intersection implies an SSX intersection, that is,

$$I_{\text{HSX}}(j,i) \Rightarrow I_{\text{SSX}}(j,i) \ , \tag{5}$$

but not the other way around. This means that any result of SSX can be converted to an HSX result by means of a linear pass over the results to cull spurious pairs that do not intersect in the hollow sense. However, such an approach of generate-and-cull-pairs can be prohibitively expensive. Consider a simple scenario in which all n spheres share the same center, but have varying radii and zero thickness. Any SSX algorithm would return $\frac{n(n-1)}{2} = \mathcal{O}(n^2)$ pairs, however the correct HSX result would be 0 pairs. The difference is further illustrated in Figure 2.

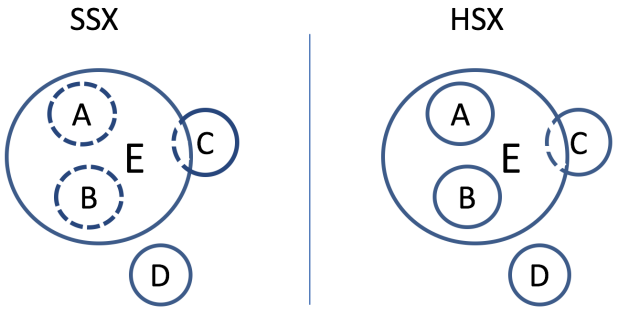


Figure 2: Here the difference between HSX and SSX is ilustrated. Collisions are indicated by dashed lines. On the right, the HSX case illustrates a collision occurring between E and C. On the left, in the SSX case, the inclusion of A and B in E is also counted as a collision, which are not valid collisions in HSX

2.3 Computational Problem

The computation of the list of pairs of intersecting pairs is a major computational burden in the simulation of cell biology systems [3]. Hence, efficiency improvements in the computation of intersections directly will improve the simulator speed and the size and number of models that can be simulated.

Efficient intersection algorithms exist for calculating collisions of solid spheres, which could be applied to hollow sphere models as a rough approximation. However, for dynamically nested models as in cell biological systems, the use of solid sphere intersection algorithms is fundamentally mismatched. The computationally intensive core problem is in fact an HSX problem, rather than SSX.

Although it is possible to apply SSX algorithms for solving the HSX problem, (a) the number of pairs generated by SSX algorithms is potentially much larger than what HSX defines, and (b) more computational work/runtime may be wasted by SSX algorithms when applied to the HSX problem.

In many systems spheres only move very little between steps. Instead of recalculating all intersections every step, it could be useful to calculate all neighbors in some vicinity and update that data only occasionally. This can easily be done by enlarging the radius and thickness of the spheres during the neighbor calculation. A simple upper bound for the size/thickness increase may be computed by considering the maximum velocity in the system and then calculating the maximal distance covered by any object within a given time step.

3. NEW HSX ALGORITHM

In this section, we present our new algorithmic approach to solve HSX problems. An overview of the algorithm is provided, followed by the details of the algorithmic steps, and a description of how large and small spheres are handled in the algorithm.

3.1 Algorithm Description

The algorithmic framework for HSX we propose here is inspired by the fast Fixed Radius Nearest Neighbor (fast FRNN, or fFRNN) algorithm proposed by Hoetzlein [11]. The FRNN problem can be seen as a special case of SSX in which all spheres have the same radius. The fFRNN algorithm provides a very efficient way to solve such problems on the GPU, by binning all spheres into a grid and then iterating over that grid. In fFRNN, each sphere is associated with exactly one bin, and the bins are large enough such that each sphere needs to only consider the neighboring bins for possible collisions.

Input: Set M of shells (hollow spheres)
Data: The initial grid size d_0
Result: Set R of possible HSX collisions
1 $d \leftarrow d_0$ // typically, $d_0 = \min_i(r_i)$
2 **while** $M \neq \emptyset$ **do**
3 $\quad S \leftarrow \{s_i | s_i \in M \wedge r_i * 2 < d\}$
4 $\quad L \leftarrow \{s_i | s_i \in M \wedge r_i * 2 \geq d\}$
5 $\quad G_{\text{FRNN}} \leftarrow grid(S,d)$
6 $\quad R_S \leftarrow \{\langle s_j, s_i\rangle \,|\, \text{inMoore}(G, s_j, s_i)\}$
7 $\quad B_{l_i} \leftarrow$
$\qquad \{b | b \in G \wedge \exists s_k \{b = \text{Bin}(s_k, G) \wedge I(s_k, l_i) = true\}\}$
8 $\quad R_L \leftarrow \{\langle s_j, l_i\rangle \,|s_j \in S \wedge s_j \in b \wedge b \in B_{l_i}\}$
9 $\quad R \leftarrow R \cup R_S \cup R_L$
10 $\quad M \leftarrow M \setminus S$
11 $\quad d \leftarrow d \cdot 2$
12 **end**
13 **return** R

Algorithm 1: The new HSX algorithmic framework

Our algorithm builds on the concept of grid-based binning,

but introduces different phases for different sizes of shells. This will be covered in the following step-by-step description of Algorithm 1.

Initially, a grid size d is chosen that is able to handle the smallest sphere in an fFRNN fashion. Grid size here refers to the width of the spatial bins. Thus, as a reasonable choice, d needs to be larger than the diameter of the smallest sphere. Furthermore, at the beginning, all the spheres are considered to be *unprocessed* and are therefore part of the set M of spheres that have not yet been fully processed.

In the main loop (**while** starting on line 2), the number of unprocessed spheres M is iteratively decreased until all spheres are fully processed and the problem is completely solved.

First, spheres are partitioned into two sets: L being those spheres with a diameter larger than d, and S being those with a diameter less than or equal to d. Next, these two sets are processed in relation to each other.

3.2 Processing Small Spheres

The grid G_{FRNN} is used to determine, in constant time, all small spheres in a certain area of space. This is shown in Algorithm 1 on line 5. Each sphere in S is assigned to a grid cell in G_{FRNN} as determined by the position of its center. Efficient creation of the grid and its associated look-up structure is discussed in Section 4.1.

On line 3, all collisions among the spheres within S are determined, that is, one small sphere overlapping with another small sphere. The notion of "small" is relative to the current value of d: at any given iteration, all spheres in S are of size no larger than d. Since all spheres in S are smaller than the grid width in terms of diameter, it is sufficient to examine the 9 cells (in 2D models) or 27 cells (in 3D models) within the Moore neighborhood of each sphere s_i (including itself). This is accomplished in line 6, where inMoore(G, s_j, s_i) denotes that the spheres s_j and s_i are contained in the Moore neighborhood of each other within the grid G.

It is also useful to make sure that redundancies are avoided, such as in the pairs $\langle a, b\rangle$ and $\langle b, a\rangle$. This can be easily achieved by imposing an ordering criteria (for example, using the sphere identifiers).

3.3 Processing Large Spheres

The goal when processing the large spheres is to find all grid cells for each large sphere that contain smaller spheres potentially colliding with the larger spheres. In other words, for each large sphere, this part of the processing considers all smaller spheres whose centers are located in any of the grid cells spanned by the large sphere.

This is accomplished in two steps. First, in line 7, for each large sphere l_i, a list B_{l_i} is created that contains all the grid cells that could possibly contain a small sphere s with a potential collision with l_i. This list of grid cells is designed to only consider the rim of the large sphere and avoid the potentially empty space in the middle of the sphere where no intersections can occur with that sphere. This computation would take into account the discretization of the sphere into d-sized grid cells enclosing the shell walls. In line 8, the grid cells B_{l_i} of each l_i are consulted to determine each small sphere s_j that actually lies in those grid cells and, thereby, mark $\langle s_j, l_i\rangle$ as a colliding pair.

In the next step, in line 9, all the cells that were found to contain potentially colliding small spheres are added to the resulting list R. This is the combination of R_S, which contains small-small sphere collisions, and R_L, which contains small-large sphere collisions.

All the small spheres of this level (corresponding to grid size d) can now be removed from the M, as they no longer need to be processed. This is done in line 10. The loop is then repeated with an increased value of grid size d, as shown in line 11.

4. PARALLEL EXECUTION ON GPU

The HSX algorithm in Algorithm 1 has been designed with the objective of applying it to parallel computing hardware such as GPUs. Therefore, each step of the algorithm has been designed to allow implementation as parallel computing operations.

Note that, as a general rule, all spheres are stored in contiguous memory arrays to enable efficient memory access throughout the algorithm. To implement line 3 and line 4 of Algorithm 1, the array containing the sphere data is partitioned such that all smaller spheres are in front and all larger spheres are behind a certain indexed position. This is parallelized on the GPU using techniques such as indicator sequences and prefix sums [21].

4.1 Creating the Grid

To implement line 5 of Algorithm 1, the process of creating the grid for fast look-up via counting sort on the GPU (similar to the method used by fFRNN) is as follows.

1. Generate a vector V of integers initialized to zero, with each entry in the vector corresponding to one grid cell.

2. Iterate (in parallel) over all spheres, calculating their grid cell identifier to which they belong, (atomically) incrementing the corresponding counter (of that grid cell) in V by 1, storing the old value of V at that position with each sphere. This value will indicate for each sphere its rank within a grid cell. The global atomic operations on modern GPUs permit this step to be accomplished very efficiently.

3. Compute the prefix sums[1] of V. This is a well studied operation that is efficiently parallelized on the GPU.

4. Create a new vector K storing all spheres in a different order – the new position of each element is computed as the index i, where i is equal to the value of its grid cell in the prefix-summed V plus its rank stored from the earlier access to V. This guarantees that each sphere has a different position in K, and all spheres can be moved to K independently, which allows this operation to be performed in parallel.

5. As a result of the previous steps, the data structures V and K are created that, when used together, allow constant-time access to all spheres in a certain grid cell. For example, queries to all spheres in any given i-th cell can be found in constant time. These are conveniently stored as contiguous elements from $K[V[i]]$ to $K[V[i+1]-1]$.

[1] An exclusive prefix sum is an operation on a list of numbers in which each element is the sum of all its preceding elements

4.2 Assembling to the Result Set

In principle, it is possible to parallelize the operation of looping over all data in order to implement line 9 of Algorithm 1. However, parallel execution of operations on data structures such as R might lead to race conditions. After all the pairs are determined, the pairs are scattered in multiple segments across the memory. Therefore, they need to be collected into a single contiguous array before they are returned as the result. This is done in the following way in our implementation when processing the small-small sphere interaction.

1. Create an array with an integer entry for each grid cell.
2. Iterate in parallel over each grid cell to determine the maximum number of collisions found by looking at the number of spheres in the *Moore Neighborhood* of that grid cell and store that count in the array.
3. Perform a prefix sum on the array and allocate sufficient storage for R such that all possible collisions would fit.
4. Using the array as an index structure, iterate in parallel over each grid cell and write all possible collisions to R.
5. Prune the list R under the criterion of redundancy ($\langle a, b\rangle \leftrightarrow \langle b, a\rangle$). Since the previous steps iterate through the result anyway, the precise intersection condition Equation 3 can also compute and remove the pairs that were erroneously added due to the grid being too coarse (that is, they were incorrectly flagged as intersecting).
6. Removing spheres from an array A in parallel is a simple, four-step process, outlined here for illustration, similar to the array partitioning operation previously indicated.
   (a) Create a second array C that has an integer corresponding to each entry in the array.
   (b) In parallel, set each entry to 0 if the corresponding entry in A is to be removed, or set it to 1 if the entry should stay.
   (c) Perform a prefix sum on C.
   (d) Re-position all entries to be kept in A to their new position as given by C.

The same process of creation followed by removal can also be used for adding the small-large collisions to R. However, pruning is not necessary for R as there cannot be redundant entries.

4.3 Finding Grid Cells for Large Spheres

In implementing line 7 of Algorithm 1, the task of finding the grid cells that enclose each large sphere's shell walls is not an easy operation to accomplish on a GPU. This is because different sphere sizes lead to different workloads per sphere. To address this problem, we have chosen the following approach.

1. For each sphere, using a rectangular bounding box, estimate the maximum number of grid cells that it overlaps.
2. Instantiate each of these grid cells in an array B and check if the cells actually overlap the sphere, or, if the cell is surrounded by l_i, but, due to its finite thickness, there is no possibility of collision. If not, mark them and remove them from the array.
3. Now, for each of these grid cells b in B, find the corresponding grid cell g_{FRNN} in G_{FRNN}.
4. For all b, determine the collisions of all small spheres in g_{FRNN} with the large sphere that spans the grid cell b. Add these collision pairs to the result set R.

4.4 Implementation Details

We have implemented our algorithm using NVIDIA CUDA v8.0 and the `thrust` package. The `thrust` package is a library that provides numerous standard algorithms, such as the aforementioned *partition* or *prefix sum* operations, with a consistent iterator interface that is inspired by the C++ standard template libraries. This offers us several advantages. First, the implementation effort is simplified because basic algorithms such as prefix sums do not need to be reimplemented in CUDA. Secondly, the code remains resilient to hardware changes as the `thrust` library is continuously updated and optimized to fit the latest parallel hardware. The benefits of the approach are evident from the fact that it was not necessary to specify low-level CUDA configuration such as block and thread counts to obtain optimal performance in our current hardware setup. Our HSX algorithm required several complex functions or CUDA-specific expressions such as atomics, but even those were utilized via the `thrust` interface (such as *for each*) available in the form of C++11 lambda functions. In the form of zip iterators, `thrust` also provides a convenient and consistent way to use the structure-of-arrays idiom, that is preferred for performance considerations over the, array-of-structures idiom, that is commonly used in object oriented programming. Finally. `thrust` also provides a back-end for CPU parallelism. This would allow us to create a parallel CPU version of the code with little extra work.

Our prototype can handle both two-dimensional (2D) and three-dimensional (3D) scenarios. Our code supports both single and double precision floating point arithmetic, but performance results presented here are with single precision.

5. PERFORMANCE STUDY

In order to empirically understand the performance of the algorithm and its implementation (called GHSX), we undertake a scalability study and run time evaluation of the system operating on a range of scenarios.

5.1 Experimental Setup

For each number of spheres investigated, we generated multiple system configurations, with different random initialization. Each of the generated scenarios was processed using different intersection algorithms. For each system-algorithm pair, the average run time is recorded from ten runs (five runs in the case of less sensitive configurations). In order to compare our algorithm to some baseline, we chose the `chrono::collide` package from the *Project Chrono* suite of physical simulation tools [33]. The source code for `chrono::collide` was published and documented under BSD-3 licence on github [1]. The `chrono::collide` package

is a CUDA-based GPU implementation of a uniform grid collision detection algorithm. The experimental study executed all scenarios with each intersection algorithm. Note that our algorithm as well as the `chrono::collide` package return a set of pairs that are *potential* collisions rather than *actual* collisions. This is because generally broad phase collision detection algorithms don't return a *perfect* result containing only the actual collisions.

Hence, for a fair comparison, we prune the returned results from all algorithms the same way (to determine *actual* collisions from the computed *potential* collisions set). This pruned (tighter) set is the final result transferred back from the GPU. Operationally, this is achieved by means of a `thrust::remove_if` call. Note that `chrono` takes the initial grid size as a parameter, just as our algorithm (d_0 in Algorithm 1 line 1). For both algorithms we always chose the optimal grid size.

For simplicity and scenario-independence, we increase the grid size d by a factor of 2 in each iteration (Algorithm 1 line 11). In future work, we plan to investigate the impact of this growth function of grid size on the overall performance of our algorithm depending on the type of system.

In order to make a reasonable comparison, the tool should run on the GPU and be able to handle arbitrary sphere sizes. Many current tools that originate in the areas of molecular dynamics or fluid dynamics are unsuitable, because they do not support the combination of GPU-based execution and arbitrary sphere sizes. The gSAP system for simulating collisions among many rigid bodies [18] (available on-line) is an exception, but it does not offer documentation and is restricted to Windows platform (communication with the authors also was not fruitful).

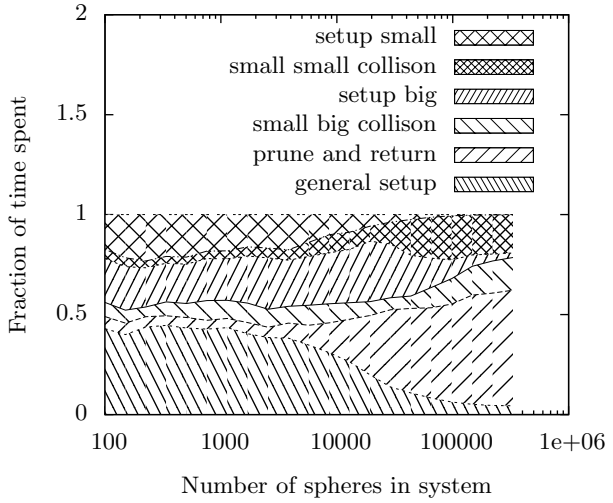


Figure 3: Run time spent in different parts of the algorithm, as a function of the number of spheres

5.2 Benchmarks

The generalized HSX problem can be instantiated in a variety of system configurations. To exercise the algorithm and its implementation, multiple parameters are varied: the total number of spheres in the system, the distribution of the spheres' radii, the density of the system measured in spheres per unit volume, and the thickness of the shell walls. All systems are investigated in 3D, except where explicitly mentioned otherwise.

Generally, in cell biological systems of interest to us, there are a relatively smaller number of large-sized compartments such as nucleus, cytosol, or membrane, and a larger number of smaller-sized compartments, such as mitochondria or lysosomes interacting with many small objects like proteins. In our test cases we model this variance by means of an exponential distribution

$$\mathcal{P}(r) = \mathrm{e}^{-r} \ , \tag{6}$$

as it offers a representative distribution of sizes.

In order to avoid system boundary overlap (or periodic boundary conditions), that are supported by GHSX, but not by `chrono`, the systems are initialized as follows.

1. Generate an sphere with uniform random position inside the volume and a radius from the exponential distribution (Equation 6)

2. If the sphere does not overlap with the systems border, add it to the system. Otherwise, discard it.

3. Repeat the preceding steps until the system has the desired density for the scenario.

When applied to systems with a small number of spheres, this process results in a slight bias towards smaller spheres, as they are less likely to intersect with the outline of the volume, when placed randomly. For larger systems, this bias asymptotically approaches zero.

For specific sections of the performance evaluation we also study other special cases, such as uniform sphere size. Some of the system scenarios are illustrated in 2D in Figure 4.

5.3 Software and Hardware Enviroment

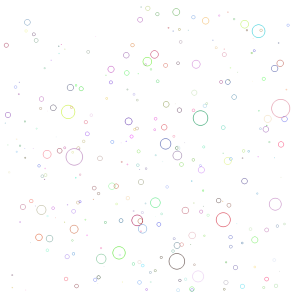
All the experiments were executed on a headless server, DELL PowerEdge R730, containing two Intel Xeon Processor E5-2683v4 (40M Cache, 2.10 GHz), 256GB RDIMM Memory (2400MT/s) and running CentOS version 7. GPU acceleration was provided by an Nvidia Tesla K80 card using a driver version 367.48 under the CUDA toolkit version 8.0.44.

5.4 Performance Results

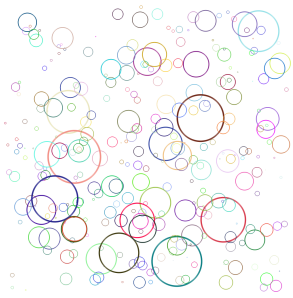
Our first key result (as observed from Figure 5) is that for a dense large (10000) system the performance of our algorithm performs well, running faster than `chrono` by about 50%. However, for sparse systems, it does not perform as well, as shown in Figure 6. This is expected because our algorithm is specially designed to handle nesting, while sparse systems exhibit limited nesting. All these runs used the same exponential distribution for the sphere sizes. The density was varied between 10 and 0.1 spheres per unit volume.

The density relation can also be seen in Figure 7. Clearly our implementation benefits from having a more tightly packed system. The denser a system is, the higher the number of collisions between large and small spheres, and, more importantly, the more likely it is that non-HSX tools will spend extra work detecting spurious collisions that do not exist when nesting is considered.

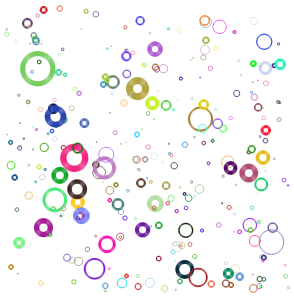
The observation that large spheres make the difference can be made from Figure 8. In highly dense scenarios containing



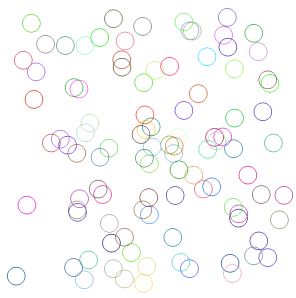
(a) Exponentially distributed radii, sparsely distributed positions



(b) Exponentially distributed radii, densely distributed positions



(c) Varying thicknesses, exponentially distributed radii, sparsely distributed positions



(d) Uniformly distributed radii, sparsely distributed positions

Figure 4: Illustrations of some of the benchmark cases

large spheres, both tools perform fairly similarly. This is because both of them perform roughly the same steps.

When the system contains spheres of only one size, there is no need for our algorithm to iterate through multiple phases, as there are only small spheres. The small-small intersection is already handled in an optimal way. Nonetheless, it can be seen that `chrono` performs better, at least at a large number of spheres. This is due to differences in implementation between the two software tools, to be able to account for nesting or not. Interestingly, `chrono` performs poorly at 5000 spheres (actually slower than with 10000 spheres). In this case, our implementation was 20% faster. This performance persists even when the positional configuration of the spheres is changed; also,it appears in sparse systems as in Figure 6, although not as pronounced. We hypothesize it as attributable to some thread allocation or buffer size which is perhaps sub-optimally set.

5.5 Number of Collisions

Further motivation for our multiphase algorithm can be found by looking at the theoretical number of HSX collisions present in the system compared to what other variants find. Here we focus on the theoretical minimum that could possibly be found by an ideal algorithm. In reality, most algorithms (including ours) return a larger set. In Figure 10, the difference between, the collisions found by using HSX, SSX and axis aligned boundary boxes (AABBs), as used by `chrono` is laid out. It is clear that in this test case the difference between HSX and SSX in marginal. This can be further seen in Figure 9, where varying the thickness of the spheres only very slightly changed the number of collisions. The run time was also hardly influenced by the thickness.

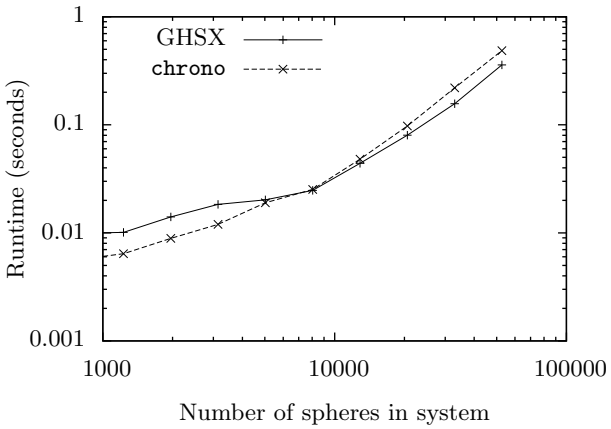


Figure 5: Run time for very dense system with exponentially distributed radii. At large system sizes, the performance gain of GHSX is about 50%

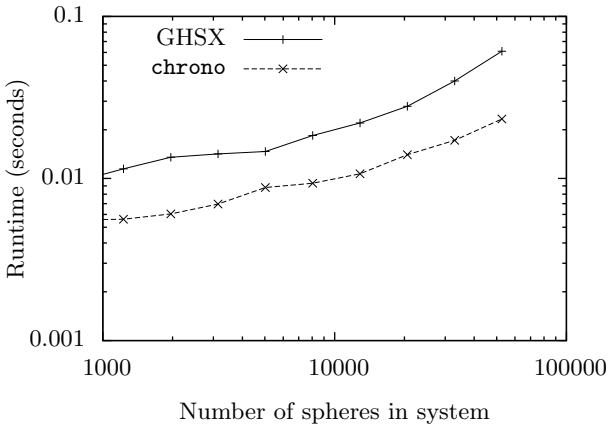


Figure 6: Run time for sparse system with exponentially distributed radii. `chrono` is significantly faster than GHSX for this sparse scenario

However, what makes a big difference in both plots is the difference between AABB and HSX/SSX. This is also where the performance difference seen in Figure 5 stems from. Not only does our algorithm handle nesting well, but also, as a byproduct, offers an efficient, generalized way of handling large spheres in close proximity to smaller ones. This is due to the finer grid used to sample the surroundings of big spheres, whereas `chrono` uses an AABB approach.

5.6 Scaling with System Size

In Figure 11, we investigate the scaling characteristics of the algorithm in 2D and 3D. It is seen that, with fewer than 10000 spheres, the resources are not used very well, as time per sphere or collision is seen to increase. The 2D version is observed to be relatively less efficient than the 3D simulations. This is possibly because the program, designed for 3D, has unused memory when only doing 2D. It could be customized to perform well specifically for 2D. *Unlike ours, few other tools offer the useful capability to handle both 2D and 3D scenarios in the same implementation.*

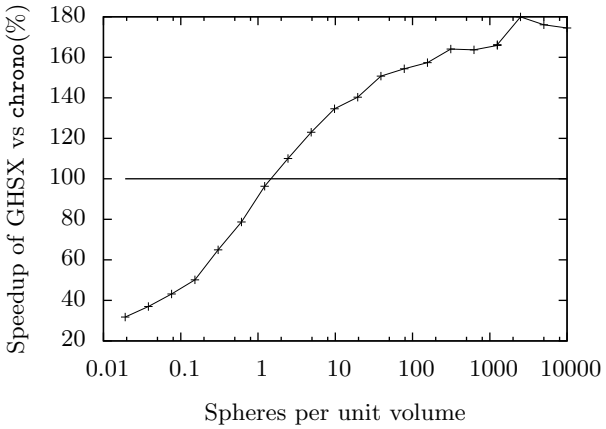


Figure 7: Performance when density is varied in a system of 30000 spheres. GHSX is significantly faster at higher densities

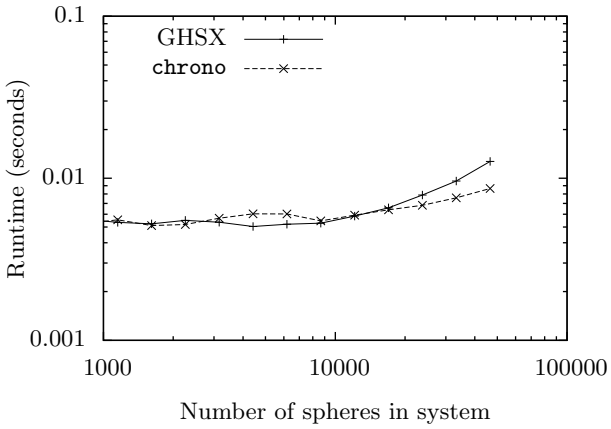


Figure 8: Run time for a system with uniformly distributed radii, corresponding to the configuration illustrated in Figure 4d

5.7 Highly Nested Case

The preceding experiments using random initial positions offer only a low degree of nesting. In physically-based structures, as in cell biology, the degree of nesting is higher. Spheres are more likely to be fully nested than to be intersecting. Therefore, to experiment with an extreme case, a scenario is created in which all spheres are perfectly nested within one another (concentric spheres).

As expected, Figure 12 shows that increased amount of nesting leads to a significant speedup of our implementation over `chrono`, because `chrono`, by design, only handles non-nested (solid) spheres, and hence does not account for the nesting.

5.8 Performance Improvements

There are a few aspects of the implementation that remain to be improved. Figure 3, shows that *pruning and returning* takes up a significant fraction of time towards the end of the execution. This is mainly due to design choices made for ease of implementation. In some places, in order to simplify memory access, the used amount of memory needs to be overestimated, leading to data structures that turn out to be smaller than estimate and hence need to be pruned. Furthermore, data layout could be improved after pruning which makes most memory access not coalesced.

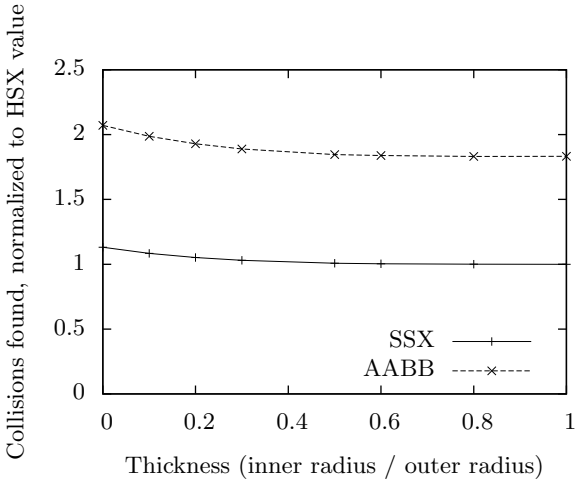


Figure 9: Variation of the number of estimated collisions with the thickness of the spheres in the case of exponentially distributed radii. Thickness is observed to have little bearing for a given algorithm, but the difference between the AABB and SSX algorithms is observed to persist even with increased thicknesses

Another potential improvement is in the work balance among the computational threads. When threads are assigned per spheres, large spheres induce higher work loads, leaving threads with smaller spheres idle.

The current implementation is also limited in the size of the problem due to memory concerns. The current data structures limit the number of spheres to 50000-200000, depending on problem structure. This is planned to be fixed in the future by partitioning larger problems into manageable sizes. Sometimes in the later iterations of the algorithm, only a small number of large spheres remain. At this stage, a simple brute force approach of checking all pairs for potential intersection would be faster, compared to the cost of building up a complex grid data structure as is done currently.

Additionally, there are potentially *algorithmic* improvements. The first concerns the determination of the initial grid size and, related to that, the growth factor of the grid size, or, more generally, a non-linear growth function. At the moment, in each iteration we simply double the grid width, but other more efficient ways can be imagined, dependent on the properties of the system.

6. SUMMARY AND FUTURE WORK

Motivated by the problem to introduce dynamic nesting into particle-based reaction diffusion simulation in continuous space, we identified a core computational problem, namely, the efficient determination of all pairs of spatial intersections of the spheres.

Detecting whether or not two spheres intersect (and thus collide) is a crucial step in such simulations, as it forms the basis for determining various types of biochemical reactions. Most collision detection algorithms focus on solid

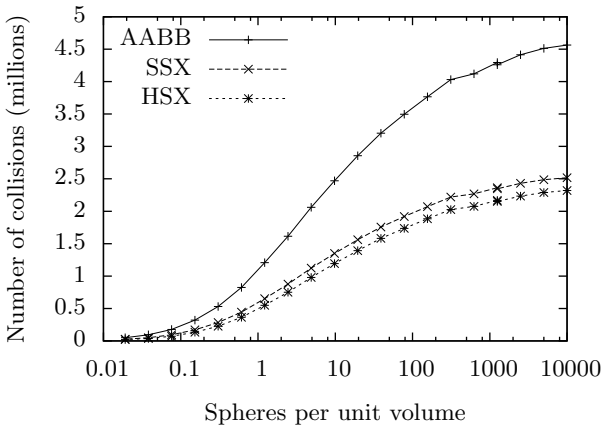


Figure 10: Variation of the number of estimated collisions in a system with exponentially distributed radii, counted using HSX, SSX, and AABB

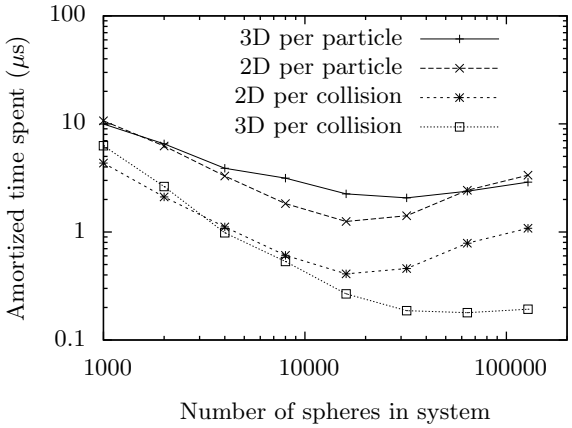


Figure 11: Scaling for 2D/3D variants of semi-dense system with exponentially distributed sphere radii. The number of collisions was kept similar for both 2D and 3D

sphere interactions (SSX) and thus tend to identify spurious collisions.

As a new and more suitable formalization, we defined the problem of hollow sphere intersections (HSX), which is an elegant generalization of dynamically nested compartmental structures. The abstraction is based on spherical bounding boxes for the spheres and inter-sphere interactions at multiple levels of nesting. We addressed the computational problem in determining all pairs of intersecting spheres under the special considerations of nested containment.

We designed a novel algorithm for the problem and implemented the algorithm on the GPU platform. We executed experiments with several scenarios of hollow spheres of different sizes, thicknesses, positions and nesting levels in systems of varying density. A comparison with another GPU-based algorithm from a leading solid sphere intersection package demonstrated the value of our new approach, not only with regard to handling highly nested systems but also for handling systems with spheres of varying sizes, whether they are solid or hollow.

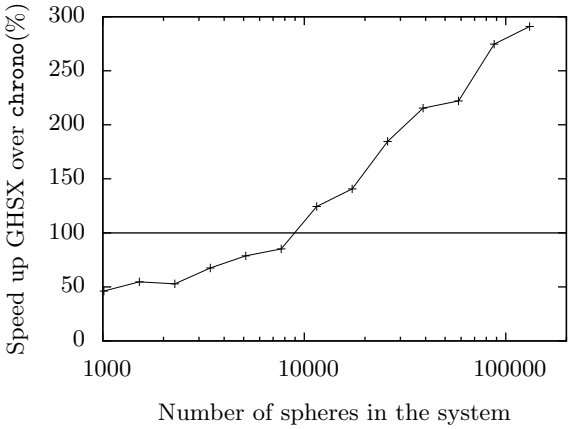


Figure 12: Speed up observed in a highly nested case

The quality of collision detection results and the run time performance of the algorithm are very encouraging. Our implementation presents a proof of concept with many possibilities for additional improvement. The algorithm and implementation have also been integrated into a cell biological simulator to simulate scenarios at larger scale and higher speed. A separate study is underway in analyzing the performance of the overall application with cell-biology configurations.

Acknowledgements

This work was partly supported by a fellowship of the German Academic Exchange Service (DAAD) and the German Research Foundation (DFG) via the research grant ESCEMMO (UH66-14).

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

7. REFERENCES

[1] https://github.com/uwsbel/collision-detection, 2014.

[2] D. Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. Ph. D thesis, Computer Science Department, Cornell University, 1992.

[3] A. Bittig and A. M. Uhrmacher. ML-Space: Hybrid spatial gillespie and particle simulation of multi-level rule-based models in cell biology. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, online first: August, 2016.

[4] L. Dematte. Smoldyn on Graphics Processing Units: Massively Parallel Brownian Dynamics Simulations. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(3):655–667, May 2012.

[5] L. Dematté and D. Prandi. GPU computing for systems biology. *Briefings in Bioinformatics*, 11(3):323–333, May 2010.

[6] P. Eastman, M. S. Friedrichs, J. D. Chodera, R. J. Radmer, C. M. Bruns, J. P. Ku, K. A. Beauchamp, T. J. Lane, L.-P. Wang, D. Shukla, T. Tye, M. Houston, T. Stich, C. Klein, M. R. Shirts, and V. S. Pande. OpenMM 4: A Reusable, Extensible, Hardware Independent Library for High Performance Molecular Simulation. *Journal of Chemical Theory and Computation*, 9(1):461–469, Jan. 2013.

[7] C. Ericson. *Real-time collision detection*. Morgan Kaufmann series in interactive 3D technology. Elsevier, Morgan Kaufmann, Amsterdam, nachdr. edition, 2008. OCLC: 551935024.

[8] M. Falk, M. Ott, T. Ertl, M. Klann, and H. Koeppl. Parallelized agent-based simulation on CPU and graphics hardware for spatial and stochastic models in biology. In *Proceedings of the 9th International Conference on Computational Methods in Systems Biology*, pages 73–82. ACM, 2011.

[9] R. M. Fujimoto. Research challenges in parallel and distributed simulation. *ACM Trans. Model. Comput. Simul.*, 26(4):22:1–22:29, May 2016.

[10] M. J. Hallock, J. E. Stone, E. Roberts, C. Fry, and Z. Luthey-Schulten. Simulation of reaction diffusion processes over biologically relevant size and time scales using multi-GPU workstations. *Parallel Computing*, 40(5-6):86–99, May 2014.

[11] R. C. Hoetzlein. Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids, 2014.

[12] P. M. Hubbard. Interactive collision detection. In *Virtual Reality, Proceedings., IEEE Symposium on Research Frontiers in*, pages 24–31, 1993.

[13] T. Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37. Eurographics Association, 2012.

[14] M. T. Klann, A. Lapin, and M. Reuss. Agent-based simulation of reactions in crowded and structured intracellular environment: Influence of mobility and location of reactants. *BMC systems biology*, 5(1), 2011.

[15] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and R. Rowe. Collision detection: A survey. In *2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 4046–4051, Oct. 2007.

[16] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, Apr. 2009.

[17] C. Lauterbach, Q. Mo, and D. Manocha. gProximity: hierarchical GPU-based operations for collision and distance queries. In *Computer Graphics Forum*, volume 29, pages 419–428. Wiley Online Library, 2010.

[18] F. Liu, T. Harada, Y. Lee, and Y. J. Kim. Real-time collision culling of a million bodies on graphics processing units. In *ACM Transactions on Graphics (TOG)*, volume 29, page 154. ACM, 2010.

[19] L. Loi. Fast gpu-based collision detection. 2010.

[20] H. Mazhar, T. Heyn, and D. Negrut. A scalable parallel method for large collision detection problems. *Multibody System Dynamics*, 26(1):37–55, June 2011.

[21] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 1 edition edition, Aug. 2007.

[22] A. J. Park and K. S. Perumalla. Efficient heterogeneous execution on large multicore and accelerator platforms: Case study using a block tridiagonal solver. *Journal of Parallel and Distributed Computing*, 73(12):1578–1591, Dec. 2013.

[23] D. Peng, T. Warnke, F. Haack, and A. M. Uhrmacher. Reusing simulation experiment specifications to support developing models by successive extension. *Simulation Modelling Practice and Theory*, 68:33–53, 2016.

[24] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on gpus. In *Proceedings of the 2008 Spring simulation multiconference*, pages 116–123. Society for Computer Simulation International, 2008.

[25] K. S. Perumalla, B. G. Aaby, S. B. Yoginath, and S. K. Seal. Interactive, graphical processing unitbased evaluation of evacuation scenarios at the state scale. *SIMULATION*, page 0037549711425236, Oct. 2011.

[26] K. S. Perumalla and V. A. Protopopescu. Reversible simulations of elastic collisions. *ACM Transactions on Modeling and Computer Simulation*, 23(2):12, 2013.

[27] D. Ridgway, G. Broderick, A. Lopez-Campistrous, M. Ru'aini, P. Winter, M. Hamilton, P. Boulanger, A. Kovalenko, and M. J. Ellison. Coarse-grained molecular simulation of diffusion and reaction kinetics in a crowded virtual cytoplasm. *Biophysical journal*, 94(10):3748–3759, 2008.

[28] S. Rybacki, J. Himmelspach, and A. M. Uhrmacher. Experiments with Single Core, Multi-core, and GPU Based Computation of Cellular Automata. In *International Conference on Advances in System Simulation. SIMUL '09*, pages 62–67, Sept. 2009.

[29] J. Schöneberg and F. Noé. Readdy-a software for particle-based reaction-diffusion dynamics in crowded cellular environments. *PLoS One*, 8(9):e74261, 2013.

[30] J. Schöneberg, A. Ullrich, and F. Noé. Simulation tools for particle-based reaction-diffusion dynamics in continuous space. *BMC Biophysics*, 7(1):1–10, 2014.

[31] T. Steinle, J. Vrabec, and A. Walther. Dynamic simulation of particle-filled hollow spheres. *Proceedings of the Applied and Mathematical Mechanics (PAMM)*, 11:881–882, 2011.

[32] K. Takahashi, S. Tănase-Nicola, and P. R. ten Wolde. Spatio-temporal correlations can drastically change the response of a mapk pathway. *Proc. of the National Academy of Sciences*, 107(6):2473–2478, 2010.

[33] A. Tasora, R. Serban, H. Mazhar, A. Pazouki, D. Melanz, J. Fleischmann, M. Taylor, H. Sugiyama, and D. Negrut. Chrono: An open source multi-physics dynamics engine. In *International Conference on High Performance Computing in Science and Engineering*, pages 19–49. Springer International Publishing, 2015.