Multibody Syst Dyn (2011) 26: 37–55
DOI 10.1007/s11044-011-9246-y

A scalable parallel method for large collision detection problems

Hammad Mazhar · Toby Heyn · Dan Negrut

Received: 6 July 2010 / Accepted: 14 January 2011 / Published online: 5 February 2011

Abstract This paper discusses a parallel collision detection algorithm. Implemented using software executed on ubiquitous Graphics Processing Unit (GPU) cards, the algorithm demonstrates two orders of magnitude speedup over a state-of-the art sequential implementation when handling multimillion object collision detection tasks. GPUs are composed of many (on the order of hundreds) scalar processors that can simultaneously execute an operation; this strength is leveraged in the proposed algorithm, which combines the use of multiple CPU cores with multiple GPUs. The software implementation of the algorithm can be used to detect collisions between five million objects in less than two seconds and was used to detect 1.4 billion contact events in less than 40 seconds. A spherical padding approach is used to represent surface geometries as large collections of spheres when dealing with collision detection between bodies with complex geometries. The proposed methodology is expected to be relevant in computational mechanics with applications in granular flow dynamics and smoothed particle hydrodynamics (SPH), where the number of contact events ranges from millions to billions.

Keywords Collision detection · GPU computing · Multibody dynamics · Friction · Contact

1 Introduction

Collision Detection (CD) is a ubiquitous task performed in computer simulations in a variety of fields such as computer games, nonlinear finite element analysis, smoothed particle hydrodynamics (SPH), multibody dynamics analysis, and granular dynamics. An example application, which draws on the last two fields and aims at gauging the mobility of tracked vehicles on granular terrain [1], is shown in Fig. 1. The purpose of the collision detection task is to understand whether two geometries are in contact, and if so, to quantify the nature

H. Mazhar · T. Heyn · D. Negrut (✉)
Simulation Based Engineering Lab, Department of Mechanical Engineering,
University of Wisconsin–Madison, 1513 University Avenue, Madison, WI 53706, USA
e-mail: negrut@engr.wisc.edu

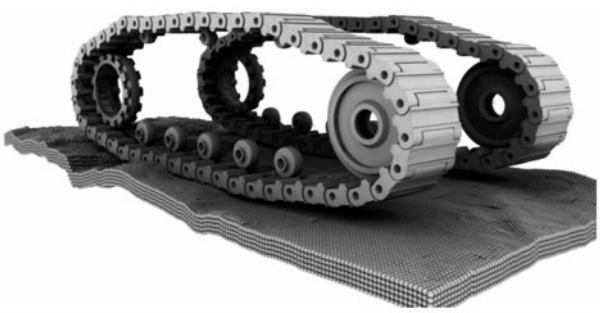


Fig. 1 The model of a tracked vehicle operating on granular material has close to 300,000 bodies leading at each simulation time step to a collision detection task that found on average more than 1 million collision events [1]

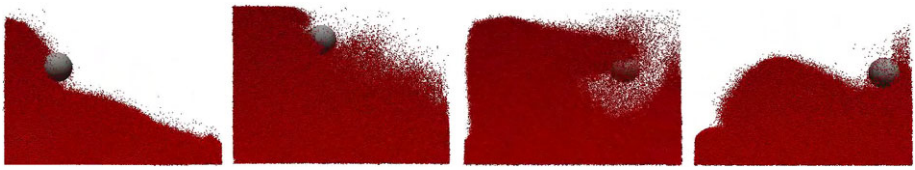
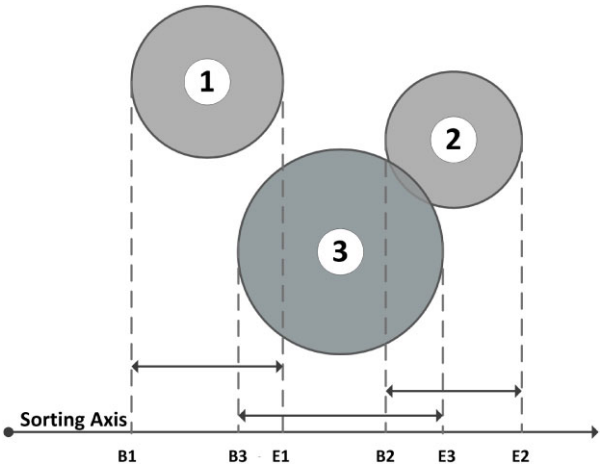


Fig. 2 Light ball floating on 1 million rigid bodies moving around in a tank while interacting through friction and contact. The simulation setup is discussed in [3]; the parallel implementation of the dynamics solution is presented in [2]. The collision detection stage found on average more than 3 million contacts at each time step of the simulation

of the contact; i.e., depth of penetration, volume of penetration, center of mass, and normal vectors to the contact surface (if applicable). For the dynamics analysis of the double tracked system in Fig. 1, a problem with 2×10^5 to 4×10^5 elements, the collision detection can become a significant computational bottleneck. For instance, when the time-stepping (numerical integration) component of the simulation was solved using a GPU parallel implementation in [2], the sequential implementation of the collision detection stage prevented any significant speedup, a consequence of Amdahl's law. A similar scenario occurred when simulating the dynamics of granular material; see Fig. 2 [3]. In order to fully leverage the potential of parallel computing in many-body dynamics problems, it becomes apparent that parallelizing the collision detection task is mandatory. This paper concentrates on the details of the parallelization of the collision detection. A discussion of the overall benefit of having both the dynamics solution and the collision detection implemented in parallel is discussed elsewhere [4].

In the multibody dynamics community, handling frictional contact dynamics and/or the task of carrying out collision detection to that end have been topics of intense research; see for instance [5–8] and references therein. These papers handle problems where the numbers of contacts at each time step is relatively small, that is, less than thousands of events. Large collision detection tasks are encountered in granular dynamics; see for instance [9] and references therein. However, this and similar many-body dynamics contributions either adopt sequential computing approaches, or are very focused in nature and fall short of presenting a solution that can address industrial applications of general interest such as the one in Fig. 1. A large body of literature addressing the collision detection topic has been compiled by the computer science community. One of the most widely used collision detection algorithms draws on a "Sort and Sweep" approach that leverages the properties of Axis Aligned Bounding Boxes (AABBs) in order to detect collisions between objects of varying size [10, 11]. An AABB is a special case of a bounding box that is always aligned to the global reference frame, simplifying collision detection as the bounding box (box that completely encloses

Fig. 3 Axis for sorting in CD



the space around an object) cannot rotate. The basic premise of this algorithm is that if two objects are colliding, their projections/shadows will overlap on all three principal axes (see Fig. 3). To this end, bounding boxes are generated for all objects and projected onto the global X, Y, and Z-axes. Each axis is then sorted by position and iterated over in linear time. A stack-like data structure is used to keep track of collision pairs; a collision will be recorded only if two bodies overlap on all three axes. The approach is simple to implement and ideally suited to execute sequentially on the Central Processing Unit (CPU). Three parallel threads are used to sort and traverse the axes sequentially and only one thread can be used to compare the three lists to detect collisions thus limiting this method's effectiveness on parallel architectures. One salient feature of this algorithm is its ability to take advantage of temporal coherence. Since objects in a dynamics simulation rarely move large distances in one time step (the positions are temporally coherent), updates can be made to the sorted axes easily and with relatively little cost. This characteristic was exploited in I-Collide [12].

Parallel collision detection approaches became tractable with the introduction of easily programmable Graphics Processing Units (GPUs). Many algorithms that use GPUs rely on shaders to do computations. Shaders, which are instruction sets designed to perform calculations on polygons and textures for graphics processing, can also be used to perform basic scientific computations. CULLIDE [13], R-CULLIDE [14], and Q-CULLIDE [15] are three CD algorithms that take advantage of shaders. These algorithms could resolve contacts between tens of thousands of polygons at near real-time speed and were faster than their CPU counterparts. More recently, work on large dynamics problems has been reported in [16], where spheres of constant radius are used to decompose complex geometries in order to decrease simulation times. Spherical decomposition removed the overhead associated with the tasks of handling complex geometries represented as triangle meshes and the ensuing triangle-triangle collision detection. This in turn allowed for real-time collision detection between thousands of objects composed of spheres.

Figure 4 shows an example of the spherical decomposition of a track shoe from a tank tread. The previously mentioned decomposition algorithm was revisited in [17]. Originally, when generating the three-dimensional grid, textures were allocated to store the complete space, and empty bins (bins that contained no bodies at all) used the same amount of memory as filled ones. In the updated version of this algorithm, bins that were not in use were trimmed, forming a tighter dynamic grid that saved memory. While optimal for spherical

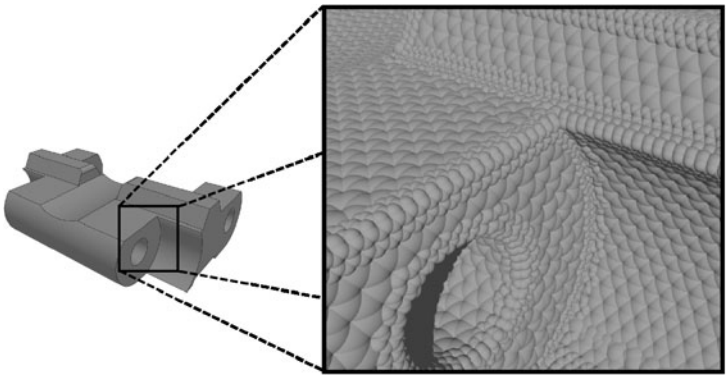


Fig. 4 Decomposition of a track shoe for application in Fig. 1

decomposition, this method could not support any additional object types that were not decomposed into spheres, limiting its usefulness in simulations containing complex shapes made with boxes, planes, ellipsoids, etc.

With the introduction of NVIDIA's Compute Unified Device Architecture (CUDA) [18], GPUs are now able to execute C code instead of relying on shader code. A parallel spatial subdivision algorithm that uses CUDA to compute interactions between spheres was introduced in [19]. This approach relies on the observation that the objects in a given bin in the spatially subdivided space can only interact with the 26 surrounding bins in a $3 \times 3 \times 3$ grid. Unlike the method introduced in [16], the CUDA based approach in [19] does not support spherical decomposition of objects. Moreover, it does not compute collisions; rather interaction forces between spheres are calculated using equations based on fluid dynamics principles. One unique feature, discussed at great length in [19], is the radix sort algorithm. Most collision detection algorithms require a fast sorting algorithm to arrange collision information into specific data structures. Commonly used sorting methods such as quick sort, while fast on CPUs, are difficult to parallelize. Radix sort is not hindered by this drawback, as it is able to sort key-value pairs in parallel extremely efficiently. Additional work in [20] was done to improve the performance and scalability of the sorting algorithm.

The collision detection approach proposed herein is geared at solving many-body dynamics problems. Examples include the dynamics of sand flowing inside an hourglass, a rover running over sandy terrain, an excavator/frontloader digging/loading granular material, etc. In this context, the collision detection task is performed on a rather small collection of rigid and/or deformable bodies of complex geometry (hourglass wall, wheel, track shoe, excavator blade, dipper), and a very large number of bodies (millions to billions) that make up the granular material. On this scale, the collision detection task, particularly when dealing with the granular material, fits perfectly the Single Instruction Multiple Data (SIMD) computation paradigm. Specifically, the same sequence of instructions needs to be applied to every individual body and/or contact in the granular material. NVIDIA's Tesla family of GPUs was selected as the target hardware for the proposed CD algorithm due to affordability and software support. Tesla C1060 has a set of 240 scalar processors. They are organized in groups of eight, each group, along with associated shared memory (16KB), a register file that can store 16,384 floats/integers, and a transcendental function unit, making up a stream multiprocessor (SM) [21]. There are 30 SMs in each GPU card, sharing 4GB of high bandwidth (140 GB/s) memory, which is the analog of the RAM in traditional CPU computing. The hardware is managed by the CUDA runtime Application Programming Interface (API)

that provides a simple interface against which users can program the GPU. Data exchange between the CPU (the host) and GPU (the device) takes place over a PCIe X16 2.0 connection with an 8 GB/s bandwidth. The CUDA driver can simultaneously manage up to 30,720 computational threads running on the C1060 hardware. While some of these threads are actively executing instructions, others idle waiting for memory transactions or queue for access to the 240 scalar processors. This thread-level time slicing, or context switching, is extremely lightweight in CUDA and hides very effectively the latency associated with global, texture, and constant memory transactions.

This paper is organized as follows. Section 2 introduces the parallel CD algorithm that draws on the GPU hardware and CUDA software setup. The section includes a discussion of the spherical padding approach invoked to perform CD between bodies with complex geometries. The outcome of several numerical experiments carried out with the proposed CD algorithm is summarized in Sect. 3. Two tests in Sect. 3.1 validate against and compare the proposed algorithm with a sequential state-of-the-art CD engine popular in the computer gaming community. Section 3.2 discusses a many-body dynamics problem whose solution uses the proposed CD method. The document concludes with final remarks and directions of future work.

2 Proposed algorithm

The parallel CD algorithm proposed performs a two level spatial subdivision. The first partitioning occurs at CPU level, which leads to a relatively small number of large *boxes*. A second partitioning of each of these boxes occurs at the GPU level, which leads to a large number of small *bins*. The collision detection occurs in parallel at the bin level. Specifically, an exhaustive collision detection process is carried out by one GPU thread to check for collisions between all the bodies that happen to intersect the respective bin. Since the bin size can be made arbitrarily small, the number of possible collisions inside the bin is kept small. Figure 5 outlines the software and hardware stack associated with this methodology.

Four OpenMP threads control the four GPUs available on the computer. The coarse grain partitioning at the CPU level is straightforward: the volume occupied by the objects is partitioned into boxes whose edges are aligned with a global Cartesian reference frame. Typically, this operation results in hundreds of boxes, which are subsequently assigned in a round

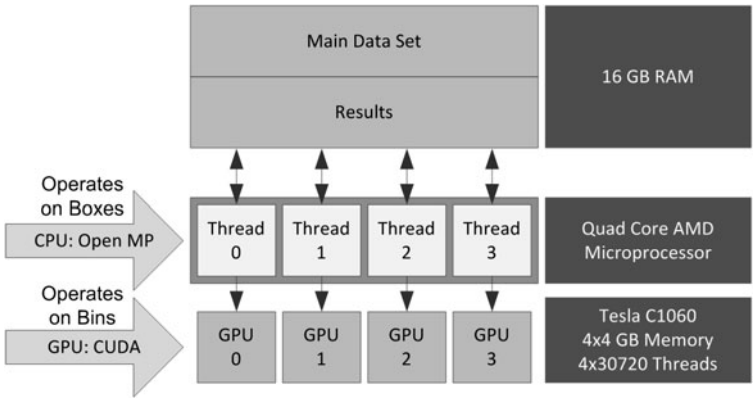


Fig. 5 Software and hardware stack showing two levels of parallelism in the proposed CD algorithm: one takes place at the CPU level, the second one, at the GPU level

robin fashion to each of the four GPUs. For instance, if there are 125 boxes it is expected that on average each of the four GPUs will have to process about 31 or 32 boxes. Objects that span two or more boxes are automatically assigned to each box when the data is sent down for CD on the GPU. A mechanism has been put in place on the GPU side to avoid double counting of potential collisions in this case. The specifics of the GPU collision detection are discussed in detail in the following subsections.

2.1 Stages of GPU collision detection algorithm

A high-level overview of the GPU-based collision detection is as follows. The collision detection process starts by identifying the intersections between objects and bins. The object-bin pairs are subsequently sorted by bin id. Next, each bin's starting index is determined so that the bins' objects can be traversed sequentially. All objects inside of a bin are subsequently checked against each other for collisions by one GPU thread. This high level process is implemented in a sequence of nine stages, each of which is discussed next. Figure 6 shows what a typical set of data used for collision detection looks like and will be used in what follows to explain the proposed approach.

Stage 1. The collision detection process begins by identifying all object-to-bin intersections. As shown in Fig. 7, an object (body) can "touch" more than one bin; there is no limit to how many intersections take place. The stage commences by determining the bounds of the simulation space. Both the largest and outermost objects are determined, allowing the required bin size to be calculated. In order for the grid and bins to remain uniform, each side of the grid, like a cube, has equal length. The bin size is set to be twice as large as the radius of the largest object, which ensures that each sphere can touch a maximum of eight bins. Optimal bin size will be further discussed in Sect. 2.2.1 as it relates to efficient use of the GPU. Finally, the number of bins used in the collision detection process is determined based on the bin size.

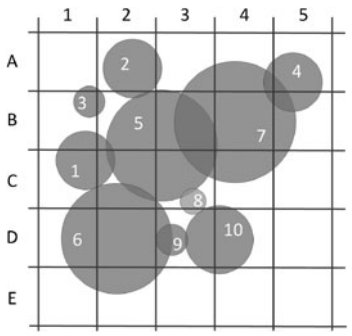


Fig. 6 Two-dimensional example used to introduce the nine stages of the collision detection process. The grid is aligned to a global Cartesian reference frame

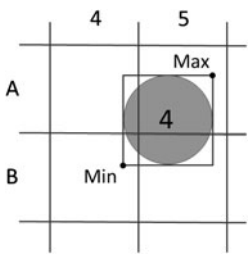


Fig. 7 Minimum and maximum bounds of object. Based on spatial subdivision in Fig. 6

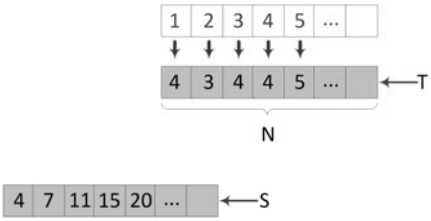


Fig. 8 Array **T** with N entries. Based on spatial subdivision in Fig. 6

Fig. 9 Result of prefix sum operation on **T**, based on spatial subdivision in Fig. 6. Each entry represents an object's offset based on the number of bins it touches

Next, the minimum and maximum bounding points of each object are determined and placed in their respective bins. For example, Fig. 7 shows that object 4's minimum point lies in B4 and its maximum point lies in A5. The entire object must fit between the minimum and maximum points, therefore, the number of bins that the object intersects can be determined quickly by counting the number of bins between the two points in each axis and multiplying them, in this case the number is 4. This number is then saved into an array, **T** (see Fig. 8), which is the size of the number of objects N. As a result of this stage, array **T** contains at index i the number of bins that object i touches.

Stage 2. An inclusive parallel prefix sum is next performed on **T**, which was created in Stage 1. A parallel prefix sum (scan) operation takes an array of N elements and returns a second array in which element i is the sum of the first i entries of the original array [22].

$$[a_0, a_1, a_2, \ldots, a_{N-1}] \longrightarrow \text{Prefix Scan} \longrightarrow \left[a_0, (a_0 + a_1), \ldots, (a_0 + a_1 + a_2 + \ldots + a_{N-1})\right] \tag{1}$$

The CUDA-based Thrust library implementation of the scan algorithm was used in this stage [23]. The inclusive scan performed on **T** returns an array **S** (see Fig. 9) where each entry is the memory offset for each object in **T**. Therefore, if one needs to determine what bins object b intersects, one would first go to $\mathbf{S}[b-1]$ (in the C language indexing begins at zero) and get the memory offset a which can be used in Stage 3. The total number of bins that object b intersects can be determined as $\mathbf{S}[b] - \mathbf{S}[b-1]$ or as $\mathbf{T}[b]$.

Stage 3. In this stage, an array **B** (see Fig. 12) is allocated of size equal to the value of the last element in **S**. This value is equal to the total number of object-bin intersections in the uniform grid. Each element in **B** is a key-value pair of two unsigned integers. The key in this pair is the bin number and the value is the object number. The bin number is calculated as described in the pseudocode of Fig. 11, where $[i, j, k]$, in 3D, are the coordinates of the bin in the uniform grid. This process is equivalent to a 3D geometric hash function and ensures that each bin number is unique. Additional checks make sure that the bin number is within the valid bounds of the uniform grid. As Fig. 10 shows, objects not fully contained within the outer edge of the grid are restricted so that their maximum bound cannot be greater than the bounds of the uniform grid. The process used to determine the intersections is essentially the same as in Stage 1 with the caveat that intersections are written rather than just being counted. In this stage, the memory offsets contained in **S** are used so that the thread associated with each body can write data in parallel to the correct location in **B**.

Stage 4. In this stage, the key-value array **B** is sorted by its key, that is, by bin id. This stage utilizes a GPU based radix sort from the Thrust library [23]. Radix sort is an efficient

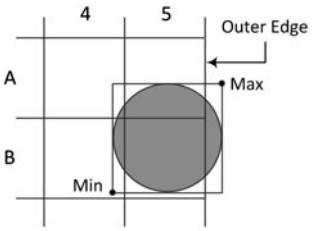


Fig. 10 Max bound is constrained to bin A5

SIDE: number of bins on side of uniform grid (stages 1 and 3)

$$BinNumber = i + j * SIDE.x + k * SIDE.x * SIDE.y$$

Fig. 11 Pseudocode: Bin number computation

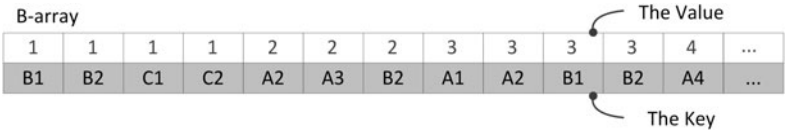


Fig. 12 Array **B**, based on spatial subdivision in Fig. 6

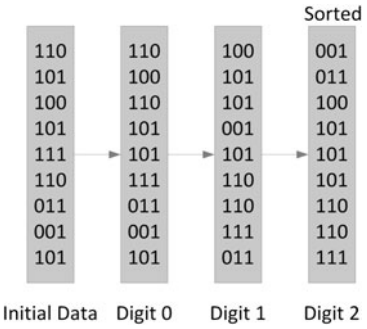


Fig. 13 Radix sort example

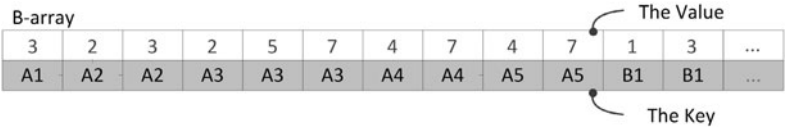


Fig. 14 Sorted array **B**, based on spatial subdivision in Fig. 6

parallel sorting algorithm that relies on the individual bits of the numbers it is sorting rather than on the actual values. Figure 13 shows the general procedure used during radix sort. Bit zero-based sorting is carried out first, followed by bit one, and so on, until the numbers are completely sorted. Figure 14 shows array **B** after sorting. In this radix sort, 32 bit keys are used, and lower order bits are sorted before higher order ones. This means that the sort is stable and values that have already been sorted will not lose their order. This stage effectively inverts the body-to-bin mapping to a bin-to-body mapping by grouping together all bodies in a given bin for further processing.

Fig. 15 Pseudocode: Bin starting index computation

```
For each thread index:
  If index<number of active Bins:
    if index > 0:
      if Current bin number != Previous bin number
        Bin start = index
          else if index=0:
            Bin start = 0
```

Fig. 16 Array **C**, based on spatial subdivision in Fig. 6

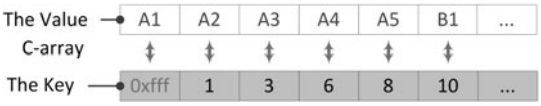
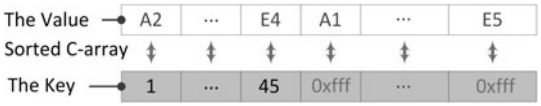


Fig. 17 Sorted array **C**, based on spatial subdivision in Fig. 6



Stage 5. Once the key value pairs are sorted by bin id, stage 5 determines the start of each bin, that is, the offset of the bin in the **B** array. The total number of elements in this array is known and equal to the total number of object-bin intersections. The process for this stage is outlined in the pseudocode of Fig. 15. Each element in the array is processed in parallel by one thread. Each of these threads reads the current and previous bin value. If these values differ, then the start of a bin has been detected. The first thread only reads the first element and records it as the initial value. The starting positions for each bin are written into an array **C** of key-value pairs of size equal to the number of bins in the 3D grid. When the start of a bin is found in array **B**, the thread and bin id are saved as the key and value respectively. This pair is written to the element in **C** indexed by the bin id. Not all bins are active. Inactive bins, i.e., bins touched by zero or one bodies, are set to 0xffffffff, the largest possible value for an unsigned integer on a 32bit, X86 architecture. This simplifies the sorting process in the next stage as such bins cannot host any contacts. Figure 16 shows the outcome of this stage.

Stage 6. In this stage, the inactive bins are pushed to the end of the array. To accomplish this, array **C** is sorted by key, a process that concentrates all inactive bins (the 0xffffffff entries, represented for brevity as 0xfff in the figure) to the end of the array; see Fig. 17. This stage allows **C** to be traversed sequentially, so that the number of active bins can be determined for the next stage. To this end, a second radix sort is utilized. Once sorted, the array is processed in parallel and the index of the last valid entry in the array is determined. No bins after this index will be processed.

Stage 7. One GPU thread is assigned to each active bin in Stage 7 to perform an exhaustive, brute-force, bin-parallel collision detection. This is effective since the number of objects being tested for collisions has become relatively small. First, the total number of active bins is determined by finding the index in the sorted **C** array where the bin value is a valid number and the next value is an invalid 0xffffffff. Because the values were sorted in the previous stage, there is only one place in the array where this can occur. Determining this value allows memory and thread usage to be allocated accurately. In this stage, each thread computes the total number of collisions in its bin and writes that number to an array **D** of unsigned integers with a size equal to the number of active bins. The bin starting number

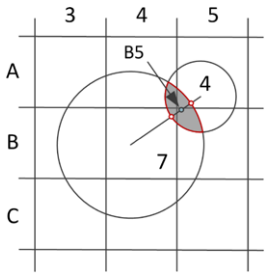


Fig. 18 Center of collision volume. Based on spatial subdivision in Fig. 6

```
For each thread index:
  If index<last:
    For posA=bin start && posA<bin end:
      For posB=posA+1 && posB<bin end:
        centerDist = distance between center of A and B
        rAB =Radius of A plus Radius of B
        if centerDist<=rAB:
          if centerDist+radius of A)<radius of B):
            collision center=bin of object A
          if centerDist+radius of B)<radius of A):
            collision center=bin of object B
          if(current bin=collision center)
            D[index]++;
```

Fig. 19 Pseudocode: Determine number of collisions

(from Stage 5) is read for the current and next bins; the starting value for the next bin being the ending value for the current one, allowing the list of objects to be iterated through.

The algorithm used to check for collisions between spheres does so by calculating the distance between both objects. Because all objects are spheres, contacts can only occur when the distance between each objects' centroid is less than or equal to the sum of their radii. Because one object could be contained within more than one bin, caution was required to prevent repeated detection of the same collision. For example, if two objects intersect within two separate bins, each thread processing its respective bin should not find the exact same collision pair. Therefore, several conditions need to be satisfied in order to guarantee unique collisions. The principle used is simple; the midpoint of a collision volume can only be contained within one bin. Therefore, only one thread would find a collision pair. For example, in order to obtain the midpoint of the collision volume, the vector going from centroid of object 4 to the centroid of object 7 is determined; see Fig. 18. Then the point where this vector intersects each object is obtained. The midpoint between these two points is used as the midpoint of the collision volume. If one object is completely inside of the other, the midpoint of the collision volume is the centroid of the smaller object. Using this process, the number of collisions are counted for each bin and written to **D**. This stage is outlined in the pseudocode of Fig. 19.

Stage 8. Once the number of collisions per bin is returned, an inclusive prefix scan operation is performed upon it [23]. This stage returns an array **E** whose last element is the total number of collisions in the uniform grid, which allows the right amount of memory to be allocated in the next and final stage of the algorithm.

Stage 9. The final step of the collision detection algorithm is to write the contact information to the contact pair array. Concretely, an array of contact information structures **F** is allocated with a size equal to the value of the last element in **E**. The collision pairs are then found using the algorithm outlined in Stage 7. At this point, instead of simply counting the

Fig. 20 Pseudocode: Computing collision data

```
ObjectA=A
 ObjectB=B
 Normal=-midpoint/centerDist
 Collision point on B(x)= B.x+(B.w/centerDist)*(A.x-B.x)
 (repeat for y and z)…
 Collision point on A(x)= A.x+(A.w/centerDist)*(B.x-A.x)
 (repeat for y and z)…
```

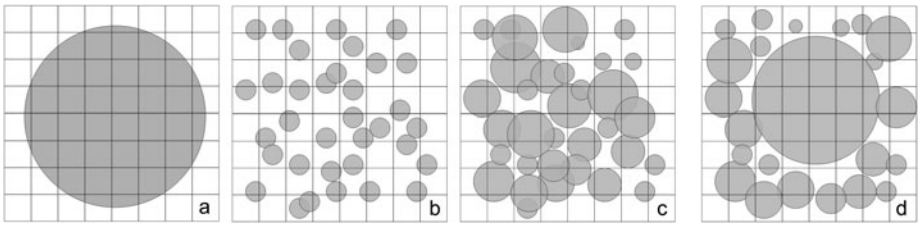


Fig. 21 Cases related to optimal bin size. (**a**) Grid is too fine. (**b**) Grid is adequately sized to the radius of the bodies within. (**c**) Bins could be optimized to be slightly larger. (**d**) Worst case scenario, difficult to optimize the bin size in this scenario

number of collisions, actual contact information is written to its respective place in **F**; see pseudocode of Fig. 20. Additional contact information can be computed if necessary in this stage.

2.2 Optimizations

Optimizations can be performed in several stages of the parallel algorithm to increase speed and reduce memory usage. They range from tuning the collision criterion to computing the optimal occupancy of the GPU and the number of threads associated with it. With the multi-GPU version of the collision detection algorithm, there are additional criteria that can decrease time taken to determine collisions.

2.2.1 Bin size

This collision detection algorithm is currently based on a uniform three dimensional grid structure, therefore, all bins are of equal physical size. As the size of objects in the algorithm is allowed to vary, it is important that an optimal bin size be chosen so that one object does not intersect many bins, thus wasting memory and computation time. However, it is important that the number of bins remains relatively large so none of the processors on the GPU become idle as there would not be enough work to fully saturate the GPU. Figure 21 shows the various scenarios related to bin and object size. For scenarios (b) and (c), the optimal value for bin size was found to be two to three times the radius of the largest sphere in the grid. Scenario (a) is not optimal as one sphere intersects many bins, and the bin size should be increased in this case. However, as in scenario (d), if the differences in radii are too large, it is difficult to determine an optimal size. As the amount of GPU memory available to process each bin is limited, if the size of a bin is too large there will be too many spheres in that bin leading to severe load imbalance. For cases like (d), the solution is to pick a bin size that is two or three times larger than the average size of all spheres in the grid. Consequently, as bins with only one object are trimmed during Stage 6, larger objects will not have a major effect on collision detection time.

2.2.2 *Efficient memory usage*

One technique considered to improve performance was the efficient use and reuse of memory available on the GPU. Two areas were looked at, namely, GPU constant memory use and reduction in unnecessary computations. There are many constants used in the proposed collision detection algorithm. Variables such as the bin dimension, dimensions of the grid, and number of objects are guaranteed not to change during the collision detection process. Rather than storing the variables in global memory or as shared memory in a kernel function, it is much more efficient to store them in the GPU's constant memory. Unlike the global memory, constant memory on the GPU is cached. Additionally, unlike the shared memory, it does not need to be copied for each kernel call. Constant memory is copied once to the GPU and then can be read by any kernel with very low latency. Second, reducing the amount of repeated computations is a basic step that can be used to reduce computation time. For example, the distance between two bodies is repeatedly used in the contact determination kernel: rather than recomputing it, it is stored in a local variable. The downside is that additional registers need to be used. However, no more than 32 registers per thread were required during any kernel call used to implement the proposed collision detection algorithm.

2.2.3 *Register and shared memory usage*

The binary file generated during CUDA compilation in combination with the CUDA occupancy calculator [24] were used to gauge and then fine tune for improved flop rate the register and shared memory usage. Specifically, in addition to containing the binary code for each GPU kernel/function, the .cubin file generated by the nvcc CUDA compiler [25] provides the number of registers used and the amount of shared memory allocated by each kernel call. The CUDA occupancy calculator uses this information to determine the maximum occupancy (percent use) that the GPU will be able to achieve. Occupancy at 100% means that the stream multiprocessors that make up the GPU are fully utilized. This level of occupancy is rarely attained, and occupancy hovering in the neighborhood of 50% is typically what real world applications manage to achieve.

Figure 22 shows occupancy as a function of threads per block, register count per thread, and shared memory usage per thread. Note that by tuning the amount of threads, it is possible to increase the occupancy for a given configuration. Finally, Table 1 lists the configuration of each kernel launched in the algorithm.

2.2.4 *Multi-GPU tuning*

When using multiple GPUs to compute collision data it was observed that certain configurations are faster than others. First, if the number of objects is relatively small, around one to five million, multi-GPU collision detection led to only modest efficiency gains, compared to a single GPU, due to the inherent overhead associated with the implementation. For larger

Table 1 Register usage, shared memory usage, and occupancy for each function in the collision detection algorithm. Note that the Tesla C1060 was used to determine the optimal values reported

Stage	Registers	Shared memory	Threads used	Occupancy
1,3	17	96	448	87.5%
5	6	48	512	100%
6	6	48	512	100%
7,9	32	96	512	50%

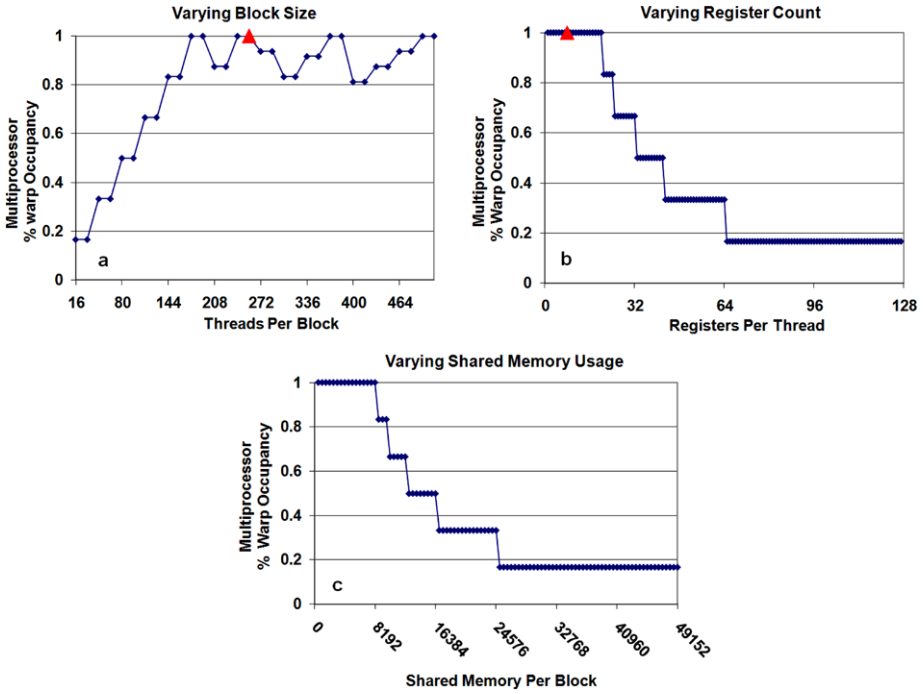


Fig. 22 Variation of occupancy with changes in threads per block (*upper left*), register usage per thread (*upper right*), and shared memory usage per thread (*lower image*)

numbers of objects, optimal speed is achieved when data is shared equally between GPUs. Finally, the size of each box that the grid was separated into had a large effect on overall speed. For a small number of boxes, and thus a large box size, each GPU will take longer to process a box and the chances of an uneven distribution of objects between boxes is greater. As the box size is decreased, the objects become more evenly distributed between GPUs; however, each box has its own overhead. Having a large number of boxes will therefore decrease the performance of the algorithm. In order to achieve optimal work distribution, the number of boxes that the physical space is divided into needs to be increased as the physical space gets larger. In this manner, the relative amount of work performed per box remains constant, only the number of boxes changes.

2.3 Spherical padding

The proposed CD algorithm belongs to the general class of multilevel spatial subdivision approaches. Rather than optimizing for spheres of equal radius like in [16], a broader view was adopted in the algorithmic design. As such, it can be modified to accommodate several other primitive geometries such as cylinders, ellipsoids [26], cones, and boxes. When dealing with complex geometries, such as the case for the tracked vehicle in Fig. 1, one of two possible courses can be adopted. The simple solution is to perform a decomposition of the surface of the bodies with complex geometry by using simple primitives. The alternative solution is to perform collision detection between the exact collision geometries in their native form or in a high-accuracy mesh form, which is much more challenging. The primitive decomposition approach was adopted in this work and further simplified to include only sphere-sphere collisions. Herein, it is referred to as spherical padding [1].

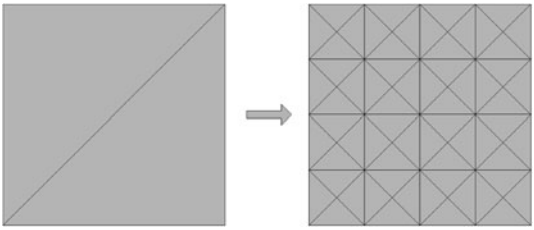


Fig. 23 Example of output from re-meshing step

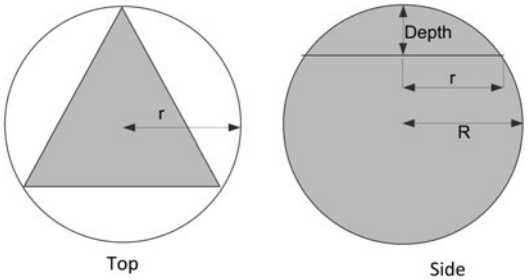


Fig. 24 Top and side views of how a sphere fits a triangle on a meshed surface

The spherical padding algorithm can be divided into several steps: meshing, sphere-fitting, and refining. In the meshing step, the triangulated three-dimensional mesh is subdivided so that it becomes finer and has a higher density of triangles per unit area. Meshing followed by remeshing generates a Delaunay triangulation of each surface in the object. It guarantees that triangles will, on average, have equal aspect ratio, and long skinny triangles will not be generated. Figure 23 shows an example of such a triangulation. Once the object has been re-meshed, the next step is to use the new triangles to fit spheres.

A sphere is fit to a triangle by using its three vertices as shown in the top view of Fig. 24. However, a unique sphere cannot be generated using this method, as the radius of the sphere is not constrained. Therefore, a depth ratio is used to specify how far above the surface of the triangle the sphere protrudes. This allows more control over the radii of the spheres while also allowing the smoothness of the surface to be controlled. Once the tessellated surfaces are covered with spheres, refining needs to be done close to the edges so that spheres generated during the previous step do not protrude out on edges where two surfaces meet. A special edge-finding algorithm was implemented to refit spheres along every edge in order to create a better approximation in [1].

To use the new spherically padded surface with the proposed collision detection algorithm, one modification was made. Because spheres making up one object should not collide with each other, each sphere was assigned two specific identifiers. The first specifies the family, or object, that a group of spheres represents. This number can be specific to each object or several similar objects. If two different colliding spheres have the same family identifier, the collision will be ignored. The second number specifies a collision mask; this mask identifies what family identifiers should not collide. In this manner, different family identifiers can be grouped together to allow more control over how objects collide.

3 Numerical experiments

3.1 Validation against and comparison with state of the art sequential collision detection

A first set of experiments was carried out to validate the implementation of the algorithm using various collections of spheres that display a wide spectrum of collision scenarios: dis-

Table 2 Error is computed by taking the Euclidean norm of the difference between the collision data from Bullet Physics Library and the algorithm presented above. We defined contact point error for each contact as the maximum location error of the two points associated with that contact

Spheres	Contacts	Contact dist. error [m]		Contact normal error [m]		Contact point error [m]	
		Avg Error	Std Dev	Avg Error	Std Dev	Avg Error	Std Dev
1,000,000	462108	1.46E-7	2.48E-4	8.24E-11	2.21E-7	2.73E-6	2.98E-3
2,000,000	1015556	7.40E-8	2.91E-4	1.91E-10	2.15E-7	2.37E-6	3.35E-3
3,000,000	1379397	1.69E-7	3.52E-4	2.75E-10	2.26E-7	3.58E-6	4.09E-3
4,000,000	1530309	5.49E-7	4.14E-4	2.33E-10	2.24E-7	1.94E-6	4.78E-3
5,000,000	1995548	6.35E-7	4.38E-4	1.09E-10	2.23E-7	3.10E-6	5.09E-3

joint spheres, spheres fully containing other spheres, spheres barely touching each other, spheres that are in contact but not full containment. In the first column, Table 2 reports the number of objects in the test for five scenarios. For each test, the error between the reference algorithm and the implemented algorithm is reported for the total number of contacts identified, the average error and standard deviation of the contact distance, contact unit normal, and point of contact. The reference algorithm used for validation was the sequential (non-parallel) collision detection implementation available in the open source state-of-the-art Bullet Physics Engine [27]. The Bullet collision detection solution was designed for speed, making it ideally suited to compare against when gauging the performance of the proposed parallel algorithm. It also relies on spatial subdivision but, unlike in [19], it is not limited to spheres.

Results showed in Table 2 indicate that the error in the proposed algorithm, when compared to the CPU implementation, is minimal and is due to floating point error. The CPU-based algorithm relies on double precision while the GPU algorithm relies on single precision. While this had an effect on the overall contact data, the number of contacts was the same. Furthermore, the small errors reported above show that no collisions were missed by the algorithm.

A second set of numerical experiments was carried out to gauge the efficiency of the parallel CD algorithm developed. The reference used was the same sequential CD implementation from Bullet Physics Engine. The CPU used in this experiment (relevant for the Bullet implementation) was AMD Phenom II Black X4 940, a quad core 3.0 GHz processor that drew on 16GB of RAM. The GPU used was NVIDIA's Tesla C1060. The operating system used was the 64 bit version of Windows 7. Three scenarios were considered. The first one measured the relative speedup gained with respect to the serial implementation. This test stopped when dealing with about six million contacts (see horizontal axis of Fig. 25), when Bullet ran into memory management issues. The plot illustrates that the relative speedup is up to 180. The second scenario determined how many contacts a single GPU could determine with this algorithm before running short on memory. As Fig. 26 shows, approximately 22 million contacts were determined in less than 4 seconds. This was followed by a third scenario, where the problem size was increased up to 1.6 billion contacts; see Fig. 27. This experiment relied on the software/hardware stack outlined in Fig. 5. Specifically, the test combined the use of OpenMP, for multiple GPU management, with CUDA, for GPU-level computation management.

3.2 Collision detection scaling for relevant dynamics application

This numerical experiment illustrates how the proposed parallel algorithm performs when interfaced with a physics based dynamics simulation package. The simulation consisted in

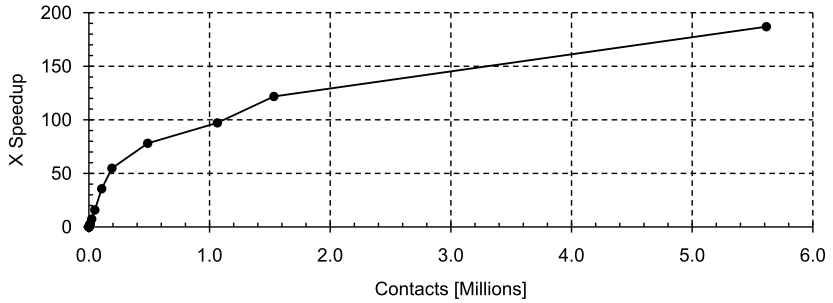


Fig. 25 Overall speedup when comparing the CPU algorithm to the GPU algorithm. The maximum speedup achieved was approximately 180 times

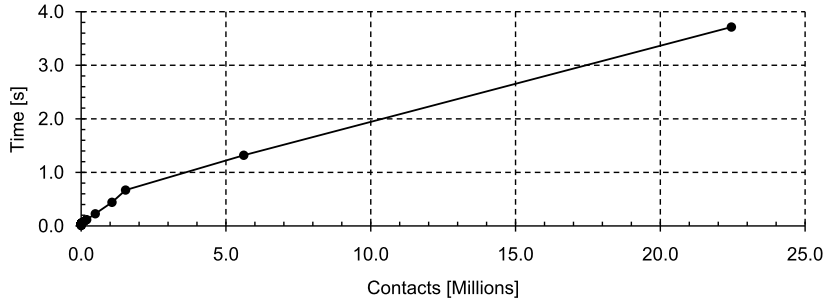


Fig. 26 Collision time vs. contacts detected, this graph shows that when the algorithm is executed on a single GPU it scales linearly

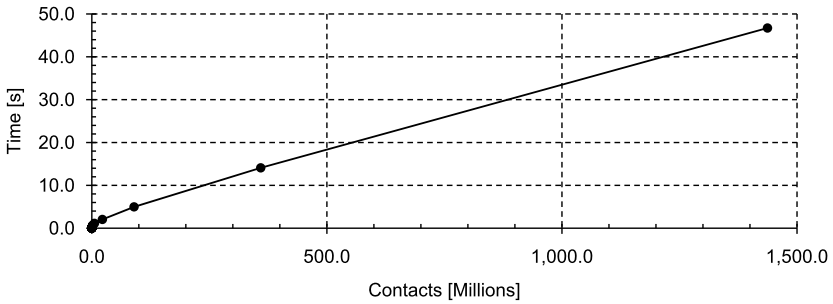


Fig. 27 Collision time vs. contacts detected, this graph shows that the multi-GPU algorithm scales linearly and can detect more than a billion contacts in less than a minute

the filling of a cylindrical tank (silo) that had a constant height and a radius varying with the number of spheres in the tank; see Fig. 28. The number of spheres in the tank was increased with each simulation from 100,000 to 1,000,000, without increasing the height of the tank. Instead, the radius of the cylinder, which had to increase, was determined for each simulation using the number of spheres and their packing factor. Each test was run using an NVIDIA Tesla C1060 until the number of collisions and thus the simulation time per time step reached steady state; i.e., the dynamics of the pile of spheres settled and the

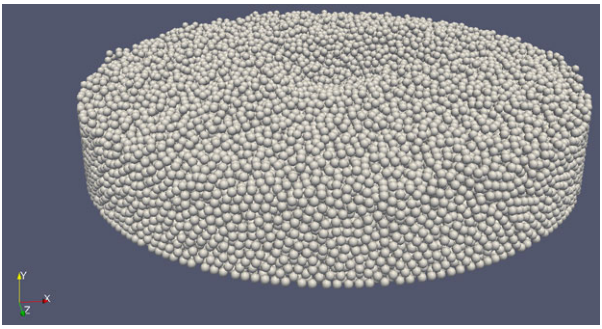


Fig. 28 Example illustrating the use of the parallel CD algorithm in conjunction with a dynamics simulation engine. The simulation captures the dynamics of a granular material with up to 1 million bodies as it settles inside a silo

Table 3 Total time taken per time step at steady state and the number of contacts associated with it

Objects	Total time [s]	GPU collision detection [s]	GPU solver [s]	Contacts
100,000	6.1972	0.5436	5.4243	361,440
200,000	12.1190	1.0758	10.5881	718,377
300,000	18.2708	1.6183	15.9482	1,080,069
400,000	23.2806	1.9746	20.4606	1,403,784
500,000	29.2565	2.4568	25.7773	1,765,772
600,000	35.0433	2.9785	30.7971	2,124,639
700,000	40.5938	3.4695	35.6405	2,439,241
800,000	46.9516	4.0234	41.2297	2,838,832
900,000	52.6227	4.5272	46.1909	3,178,228
1,000,000	58.1518	4.9473	51.1686	3,548,594

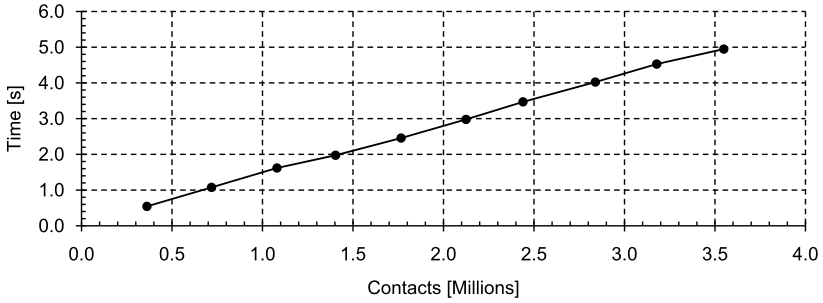


Fig. 29 Collision time as the number of contacts increases

time required to advance the simulation by one time step was practically constant. The open source dynamics engine, Chrono::Engine, was used for this dynamics simulation [28] in conjunction with an inelastic contact model.

Given the significant reduction in the collision detection time reported in the previous subsection, it can be expected that collision detection will stop representing a computational bottleneck of the overall parallel simulation approach. The results presented in Table 3 and graphed in Figs. 29 and 30 confirm this expectation. Moreover, they indicate that even in a dynamics application, the collision detection algorithm scales linearly. The results show that

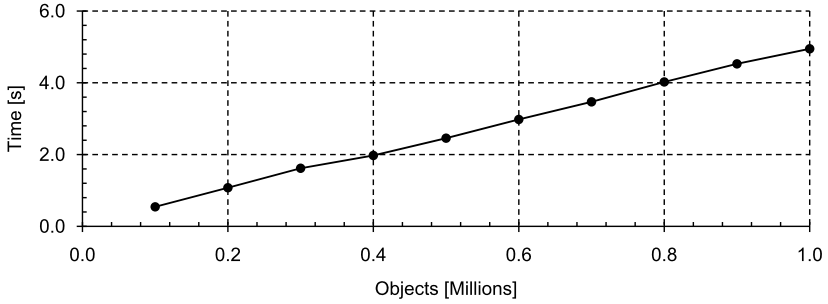


Fig. 30 Collision time as the number of objects increases

the amount of time required to advance the simulation by one time step was spent primarily on the GPU dynamics solver portion of the simulation, with a small amount of time taken up by the collision detection step. These times are larger than the raw collision detection times presented earlier due to the pre and post processing required by the physics engine as it transfers and organizes data for use between the solver and collision detection.

4 Conclusions

This paper proposes a spatial subdivision based approach for parallel collision detection on the GPU. The algorithm combines two paradigms, CPU computing and GPU computing, to enable in an open source code effective and scalable parallel collision detection for scenarios with billions of collision events. On the CPU side, the proposed approach draws on OpenMP software support. On the GPU, it relies on the CUDA toolkit. The proposed algorithm significantly enlarges the solvable problem size: from 6 million collision events with the Bullet Physics Engine, to either billions of events in a multi-GPU configuration, or to approximately 23 million collision events when using only one GPU card. Moreover, when compared to Bullet, the proposed algorithm was two orders of magnitude faster and eliminated the collision detection as a computational bottleneck in the physics based simulation of granular material dynamics. Ongoing work aims at streamlining the implementation to leverage the Message Passage Interface (MPI) standard for cluster simulation setups, which represents a more challenging undertaking due to load balancing issues. In terms of geometries handled, an ongoing effort concentrates on collision detection for ellipsoids and cylinders. The collision detection framework outlined herein should accommodate these geometries with minimal implementation changes. More complex in scope and methodology, a second investigation/implementation effort should aim at large scale collision detection of geometries defined through Delaunay triangulations.

Acknowledgements Financial support that enabled this research comes from NSF grant CMMI-0840442, NVIDIA Corporation, and Function Bay, Inc.

References

1. Heyn, T.: Simulation of tracked vehicles on granular terrain leveraging GPU computing. M.S. Thesis, Department of Mechanical Engineering, University of Wisconsin–Madison. http://sbel.wisc.edu/documents/TobyHeynThesis_final.pdf (2009)

2. Tasora, A., Negrut, D., Anitescu, M.: Large-scale parallel multi-body dynamics with frictional contact on the graphical processing unit. Proc. Inst. Mech. Eng., Proc. Part K, J. Multi-Body Dyn. **222**(4), 315–326 (2008)
3. Tasora, A., Negrut, D., Anitescu, M.: GPU-based parallel computing for the simulation of complex multibody systems with unilateral and bilateral constraints: An overview. In: Blajer, W., Arczewski, K., Fraczek, J., Wojtyra, M. (eds.) Multibody Dynamics: Computational Methods and Applications, pp. 45–55. Springer, Berlin (2010)
4. Negrut, D., Tasora, A., Mazhar, H., Heyn, T., Hahn, P.: Leveraging parallel computing in multibody dynamics. Multibody system dynamics. Under review (2011). This paper was submitted in December with manuscript number MUBO-10-79
5. Gonthier, Y., McPhee, J., Lange, C., Piedbuf, J.-C.: A regularized contact model with asymmetric damping and dwell-time dependent friction. Multibody Syst. Dyn. **11**, 209–233 (2004). 10.1023/B:MUBO.0000029392.21648.bc
6. Förg, M., Pfeiffer, F., Ulbrich, H.: Simulation of unilateral constrained systems with many bodies. Multibody Syst. Dyn. **14**, 137–154 (2005). 10.1007/s11044-005-0725-x
7. Najafabadi, S.M., Kövecses, J., Angeles, J.: Impacts in multibody systems: modeling and experiments. Multibody Syst. Dyn. **20**, 163–176 (2008). 10.1007/s11044-008-9117-3
8. Choi, J., Ryu, H., Kim, C., Choi, J.: An efficient and robust contact algorithm for a compliant contact force model between bodies of complex geometry. Multibody Syst. Dyn. **23**, 99–120 (2010). 10.1007/s11044-009-9173-3
9. Fleissner, F., Gaugele, T., Eberhard, P.: Applications of the discrete element method in mechanical engineering. Multibody Syst. Dyn. **18**, 81–94 (2007). 10.1007/s11044-007-9066-2
10. Baraff, D.: Dynamic simulation of non-penetrating rigid bodies. Ph.D. Thesis, Cornell University (1992)
11. Baraff, D.: An introduction to physically based modeling: rigid body simulation II6Nonpenetration constraints. In: An Introduction to Physically Based Modelling, SIGGRAPH'97 Course Notes (1997)
12. Cohen, J.D., Lin, M.C., Manocha, D., Ponamgi, M.: I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In: Proceedings of the 1995 Symposium on Interactive 3D Graphics, p. 189. ACM, New York (1995)
13. Govindaraju, N.K., Redon, S., Lin, M.C., Manocha, D.: CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 25–32. Eurographics Association, Geneve (2003)
14. Govindaraju, N.K., Lin, M.C., Manocha, D.: Fast and reliable collision culling using graphics hardware. In: IEEE Transactions on Visualization and Computer Graphics, pp. 143–154 (2006)
15. Govindaraju, N.K., Lin, M.C., Manocha, D.: Quick-CULLIDE: fast inter-and intra-object collision culling using graphics hardware. In: ACM SIGGRAPH 2005 Courses, p. 218. ACM, New York (2005)
16. Harada, T.: Real-time rigid body simulation on GPUs. GPU Gems **3**, 611–632 (2007)
17. Harada, T., Koshizuka, S., Kawaguchi, Y.: Sliced data structure for particle-based simulations on gpus. In: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia, pp. 55–62. ACM, New York (2007)
18. NVIDIA. CUDA Programming Guide. Available online at http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf (2009)
19. Le Grand, S.: Broad-phase collision detection with cuda. GPU Gems **3**, 697–721 (2007)
20. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium, pp. 1–10. IEEE Press, New York (2009)
21. NVIDIA Corporation. Tesla c1060 datasheet. Available online at http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C1060_US_Jun08_FINAL_LowRes.pdf (2008)
22. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. GPU Gems **3**(39), 851–876 (2007)
23. Hoberock, J., Bell, N.: Thrust: a parallel template library. Available online at http://code.google.com/p/thrust/ (2009)
24. NVIDIA Corporation. Compute unified device architecture occupancy calculator (2009)
25. NVIDIA Corporation. Compute unified device architecture software development kit (2009)
26. Pazouki, A., Mazhar, H., Negrut, D.: Parallel ellipsoid collision detection with application in contact dynamics-DETC2010-29073. In: Fukuda, S., Michopoulos, J.G. (eds.) Proceedings to the 30th Computers and Information in Engineering Conference. ASME International Design Engineering Technical Conferences (IDETC) and Computers and Information in Engineering Conference (CIE) (2010)
27. BULLET: Physics simulation forum. Bullet physics library. Available online at http://www.bulletphysics.com/Bullet/wordpress/ (2010)
28. Tasora, A.: Chrono::Engine, an open source physics–based dynamics simulation engine. Available online at www.deltaknowledge.com/chronoengine (2006)