IADIS International Journal on Computer Science and Information Systems
Vol. 14, No. 2, pp. 25-40
ISSN: 1646-3692

HETEROGENEOUS DESIGN AND EFFICIENT CPU-GPU IMPLEMENTATION OF COLLISION DETECTION

Mohid Tayyub and Gul N. Khan.
Electrical, Computer and Biomedical Engineering, Ryerson University, 350 Victoria Street, Toronto ON M5B2K3 Canada

ABSTRACT

Collison detection is a wide-ranging real-world application. It is one of the key components used in gaming, simulation and animation. Efficient algorithms are required for collision detection as it is repeatedly executed throughout the course of an application. Moreover, due to its computationally intensive nature researchers are investigating ways to reduce its execution time. This paper furthers those research works by devising a parallel CPU-GPU implementation of both broad and narrow phase collision detection with heterogenous workload sharing. An important aspect of co-scheduling is to determine an optimal CPU-GPU partition ratio. We also showcase a successive approximation approach for CPU-GPU implementation of collision detection. The paper demonstrates that the framework is not only applicable to CPU/GPU systems but to other system configuration obtaining a peak performance improvement in the range of 18%.

KEYWORDS

CPU-GPU Systems, Efficient CPU-GPU Implementation, Fast Collision Detection, Gaming and Animation, Heterogeneous Computing

1. INTRODUCTION

Collision detection techniques allows various gaming and simulation applications to determine when some objects will or have collided. This is key for maintaining realism as ignoring object interaction leads to undesired scenarios and visual glitches. Collision detection allows for a response to occur due to an event such as when a game player is hit with projectile or when to recalculate the movement on a rigid body. Collision detection algorithms are computationally expensive and their complexity or cost increase with higher number of objects (in a game scene) and/or with an increase in the complexity of the shape of the objects. There have been various studies on collision detection methods as it is applicable to many fields. Moore and Wilhelms (1988) investigated collision detection and response for computer animation applications. Ng et al. (2012) focused on the optimization of collision detection method on mobile devices for a specific genre of games. Collision detection is a diverse field itself where the suitability of various algorithms can be tried depending on the application and the available computational resources. However, there is generally a commonality between all the implementations of collision detection. Efficient implementations of collision detection use a two-phase approach i.e. a broad phase followed by a narrow phase (Nguyen 2007). We have implemented a broad-phase collision detection on CPU-GPU based platforms (Tayyub and Khan 2019). In this paper, we extend that work by providing a complete collision detection and its CPU-GPU implementation by providing the details of both broad and narrow phase collision detection. Furthermore, we extend our workload partition approach to system configurations outside CPU/CPU such as a discrete GPU with an integrated GPU (IGPU).

Parallel implementations of various on-line algorithms are becoming increasingly popular as multicore processors including GPUs (Graphis Processing Units) are readily available in various devices and computing platforms. The traditional GPU model of executing a program considers the CPU as host and only interacts with parallel programs in terms of launching and synchronizing data transfers and kernel launches. Co-operative scheduling between GPU and CPU is the way to further increase performance. The use of this traditional model on a heterogeneous CPU/GPU platform is inefficient as it ignores the computational power of multi-core CPUs. The usage of cooperative GPU and CPU scheduling has been investigated and found to result in notable improvement when compared with GPU only execution (Pandit and Govindarajan 2014, Lee et al. 2015). A crucial information for cooperative execution is to get an optimal workload partition between the CPU and GPU. Our research makes the following contributions:

- Presents a successive approximation approach to estimate an optimal partition that is applicable to any application where offline profiles can be created or where the parallel kernels are executed multiple times over the course of application, and thus initial profiling cost can be amortized.
- Increasing the efficiency of a broad and narrow phase GPU parallel collision detection by porting it to a CPU/GPU cooperative workload sharing as well as some other CPU-GPU configurations.
- Proving the efficacy of partitioning the collision detection across multiple CPU-GPU platforms for a real-world application benchmark.

Collision detection is a well-studied technique (Jiménez et al. 2001) and our focus in this paper is on the CPU-GPU partitioning and implementation of a combined broad and narrow phase parts of the algorithm. As collision detection computation is to be divided between GPU and CPU, OpenCL is selected for implementation. OpenCL (Open Computing Language) is an open-source framework that enables parallel computing for various heterogeneous platforms involving GPU, CPU and FPGA (Stone et al. 2010). Application programs are written as OpenCL kernels, and compiled for any heterogeneous platform. Bullet (2019) is an open source physics library that provides an experimental OpenCL GPU support. Bullet also includes parallel collision detection in OpenCL, providing both narrow-phase and broad-phase kernels. In this paper, we employ these kernels as the base for collision detection workload sharing between a CPU and the GPU.

Broad-phase collision detection algorithm is fast, and it culls away most of the possible collision pairs using a simpler rejection test (Mirtich 1997). The narrow-phase takes a closer look at the pairs left after the culling process of broad-phase processing. It employs a more precise technique to determine the colliding pairs. These two phases act at different points and the algorithm choices are generally made independently. Broad-phase collision detection techniques generally share a commonality in calculation by employing a bounding volume method to check for object collision. A bounded volume check may be of the form of an axis-aligned bounding box (AABB), a bounding sphere or an oriented bounding box (OBB) (Coming and Staadt 2006). The broad-phase algorithm checks the occurrence of collision between two objects by first encompassing the objects in a simple silhouette or shape and then by identifying any overlap between the two silhouettes.

In the case of AABB, the bounding box is defined as the smallest cuboid which contains the object and its edges are parallel to the coordinate axes (Huang 2012). The condition of parallel to the coordinate axis is imposed to allow a static calculation to determine the overlap between bounded boxes (see equation 1). When the object rotates, the assigned axis aligned box must be large enough to handle all cases of the object or it must be re-calculated in real-time. Bounded spheres work similarly, however, it is a minimal size sphere around the object instead of a cuboid. It requires a slightly more complex algorithm for collision check. An OBB may use a cuboid as its silhouette but it is not under the condition of being axis-aligned. A very tight-fitting bounding box can be created, but with higher complexity that may result in a degraded performance as compared to AABB (Ng et al. 2012). The collision detection presented in this paper employs the AABB technique for collision check. The first step is to determine the axis aligned bounded box for each object. Axis aligned bounded boxes is applicable to both 3-D and 2-D objects. Figure 1(a) shows the bounded box for a 2-D car object. The vertices of the bounded box correspond to coordinates of (X_{min}, Y_{min}), (X_{max}, Y_{min}), (X_{min}, Y_{max}) and (X_{max}, Y_{max}). It is an axis aligned box as the edges that make the box are aligned with X and Y axis. The interval coordinates X_{min}, Y_{min}, X_{max} and Y_{max} create the condition test when determining the collision between two objects. Figure 1(b) demonstrates a possible collision detection of two objects. For this scenario, equation 1 will result in a *True* condition for the occurrence of a collision. This equation can be expanded to include Z_{min} and Z_{max} for 3-D objects. Equation (1) and Figure 1(b) demonstrate the simplicity of the AABB check. They also indicate the limitation, where a collision check will result in equation (1) being True. However, the two objects have not actually collided.

$$(X_{min1} \leq X_{max2}) \cap (X_{min2} \leq X_{max1}) \cap (Y_{min1}\ Y_{max2}) \cap (Y_{min2} \leq Y_{max1}) \qquad (1)$$

Moreover, one should not perform an AABB overlap test between all the objects in the world. In a scene with, N objects, the complexity of the algorithm will be $O(N^2)$ (Huang 2012). This is not a feasible implementation, and it may lead to numerous broad-phase collision checks. Various schemes have been proposed to reduce the number of overlap tests that may be required to create a more linear time complexity. Main schemes include sweep and prune (SAP), bounded volume hierarchy (BVH) and spatial subdivision (Coming and Staadt 2012, Ng et al. 2012, Zahmann 2002).

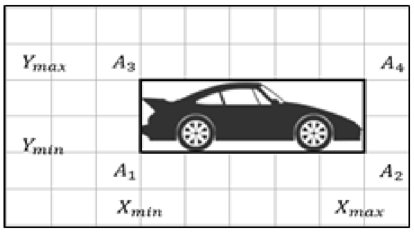


Figure 1. (a) Creating an AABB

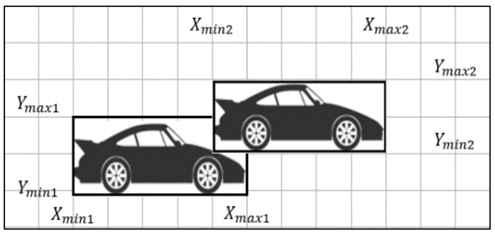


Figure 1. (b) Determining a collision occurrence

Sweep and prune is an algorithm that reduces the number of overlap checks by projecting the objects on a single axis. The objects AABB min/max values on that corresponding axis are obtained and sorted. Two data arrays (Sweep and Pairs) are populated as the sorted list is iterated over and shown in Figure 2. During iteration when an object's min value is reached it is pushed onto data structure Sweep and when the object's max value occurs it is removed from data structure Sweep. Objects that exist in data structure Sweep simultaneously form pairs that are pushed on data structure Pairs. At the end of iteration, the overlap test (1) is only performed on those pairs present in data structure Pairs. SAP takes advantage of temporal coherence (objects do not change drastically between frames). As such on the next iteration the list does not need to be created from scratch rather object values are updated quickly using insertion sort. (Tracy et al. 2009).

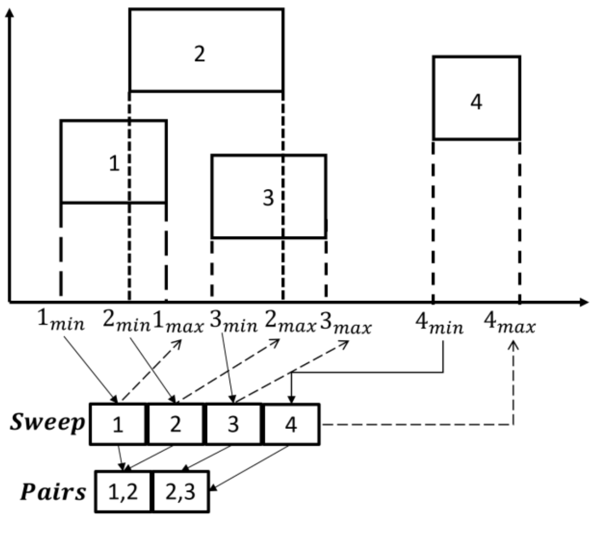


Figure 2. Sweep and prune (SAP)

Bounded volume hierarchy is a tree hierarchy that organizes a set of bounded volume objects into larger bounded volumes that can encompass one or more objects (Figure 3). The objects become leaf nodes on the tree structure while the larger bounded volumes become corresponding parent nodes. This method reduces the number of overlap tests as objects whose parent structures do not overlap do not need to be considered. (Ericson 2004). After the broad phase collision detection scheme has computed a list of potential collision pairs. A more complex and accurate algorithm is utilized in the second phase (narrow phase) to confirm or discard individual collision pairs from the list. Like broad phase there is more than one algorithm presented in literature. The two most notable algorithms are Gilbert-Johnson-Keerthi Algorithm (GJK) and Separating Axis Theorem (SAT).

GJK algorithm is based on the following principle. If object A and B comprised of two sets of position vectors A and B respectively. The Minkowski sum and difference are defined as (A+B) and as (A–B) respectively. The most important property of Minkowski difference in respect to collision detection is that when two objects collide, their corresponding Minkowski difference must contain the origin. Furthermore, the minimum distance between the origin and the Minkowski difference is equivalent to the minimum distance between the objects (A and B). However, direct computation of Minkowski difference is non-trivial and thus GJK is an algorithm that can only be implemented iteratively where solution converges to the minimum distance between the origin and Minkowski difference. For its precise implementation, real-time collision detection by Ericson (2004) can be consulted.

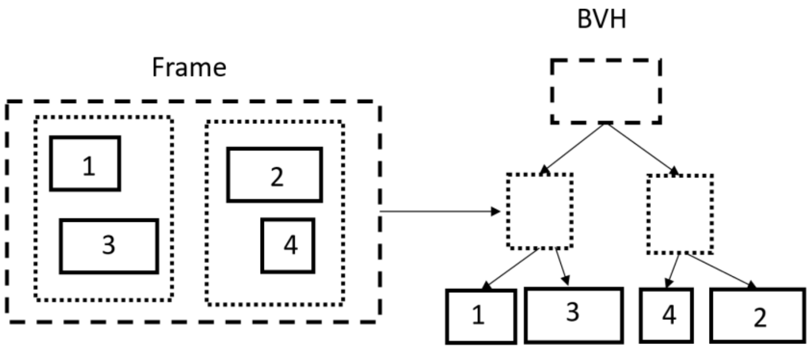
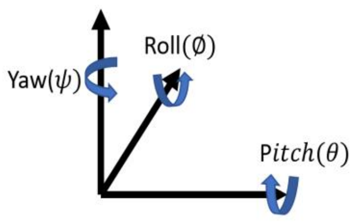


Figure 3. Bounded volume hierarchy (BVH)

Our focus is on the parallel implementation of spatial subdivision for the broad phase part of collision detection as well as the separating axis theorem for narrow phase collision detection. As such these algorithms and resulting implementation are given more attention to and are described in detail here in sections 2 and 3 respectively. In this paper, we refer to world as the abstract view that encompasses all the objects of an application, which are to be considered for collision detection. For the sake of completion, a brief overview of object rotation is provided as narrow phase algorithms do not require an axis aligned silhouette or object. Typically, object rotation is performed through Euler angles or quaternions. During creation of test environment (section 4) objects are distributed across the world and randomly rotated according to Euler angles to provide a more accurate representation of real scenarios. Euler angles are intuitive and can be computed through basic matrix multiplication, however it suffers from a scenario called “gimbal lock” (Chapala et al. 2016). There are three Euler angles (Roll, Pitch, Yaw) corresponding to the rotation angles across the three-coordinate axis (see Figure 4). Three rotation matrices ($R_x R_y R_z$) are formed from the Euler angles. A single rotation matrix is then computed as multiplication of all three matrices. We will arbitrarily define the rotation order as $R = R_x R_y R_z$. Then to rotate point $P = [X, Y, Z\}$ according to rotation matrix R is simply the calculation of $P' = R * P$.

2. SPATIAL SUBDIVISION - BROAD PHASE COLLISION DETECTION

Spatial subdivision techniques can be implemented in different forms (Teschner et al. 2003). Our implementation here can be classified in the form of a spatial hashing uniform grid. Optimizations of spatial hashing has been described by Teschner et al. and others (2005). The *world*, in a uniform grid based spatial subdivision, is divided into equal grid blocks. Where a grid block is at least as big as the largest axis aligned bounded box. It will reduce the time complexity as for each object, the overlap check needs only to be performed against objects whose centroid lie within the same grid block and which are directly adjacent to each other. Figure 5 explains how a *world* may be spatially subdivided in a 2D *world*. For 3D implementation, the same concept can be applied, however the number of adjacent cells for an object will increase. Uniform grid provides accelerated collision detection but suffers for objects varying in sizes. Due to the condition that the grid block is at least as big as the biggest object, and with one large object and multiple smaller objects, many overlap tests will occur as the probability of having smaller objects in adjacent cells increases due to large grid block. This issue can be resolved by having a hierarchical or octree grid (Wong et al. 2014).



$$R_x = \begin{matrix} 1 & 0 & 0 \\ 0 & \cos\emptyset & -sin\emptyset \\ 0 & sin\emptyset & cos\emptyset \end{matrix} \quad R_y = \begin{matrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{matrix} \quad R_z = \begin{matrix} cos\psi & -sin\psi & 0 \\ sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{matrix}$$

Figure 4. Euler rotation

The parallel implementation of collision detection found in Bullet (2019) is close to the GPU GEMS methodology (Nguyen 2007) that can be used for comparison. Figure 6 shows the main steps of the parallel spatial subdivision algorithm for GPU implementation. The algorithm is broken down into four steps that occur in a sequential order with each individual step occurring in parallel.

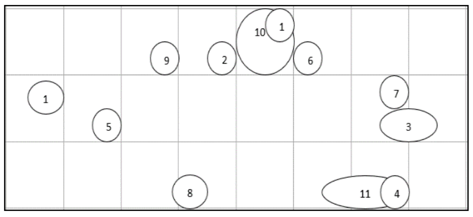


Figure 5. 2-D grid subdivision

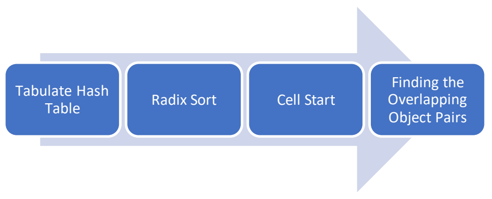


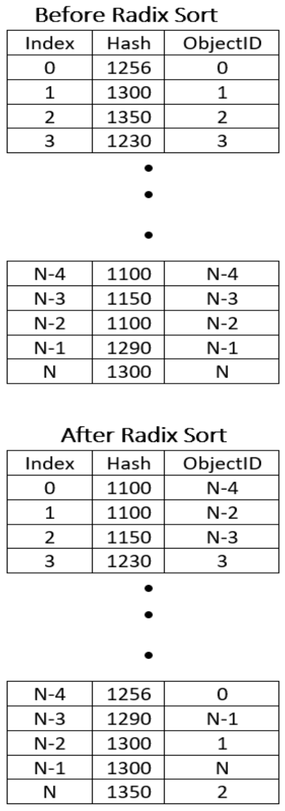
Figure 6. Flow of parallel spatial subdivision

2.1 Tabulation of Hash Table and Radix Sort

The first step in this process is to transfer the data for all axis aligned bounded boxes to the GPU. With the AABB data, the grid location for which each AABB centroid lies is calculated. Using the centroid location and the grid dimensions, equation (2) is used to tabulate the hash function. In the parallel implementation, each work item is responsible for determining the hash for an individual object. Each hash value represents a unique grid block in the world.

$$centriod_{pos\_z} * grid_{dimension\_y} * grid_{dimension\_x} + centriod_{pos\_y} * grid_{dimension\_x} + centriod_{pos\_x} \qquad (2)$$

GPU implementation of sorting algorithms are not specific to collision detection. There are several GPU-based sorting algorithms with some particularly on radix sort (Delorme et al. 2013, Nguyen 2007, Satish et al. 2009), which also provide performance profiling for the GPU implementation of parallel radix sort. The radix sort used in this application is to re-order the hash table such that the hash values for each object are in order. Figure 7 shows the effect of applying radix sort to hash table.



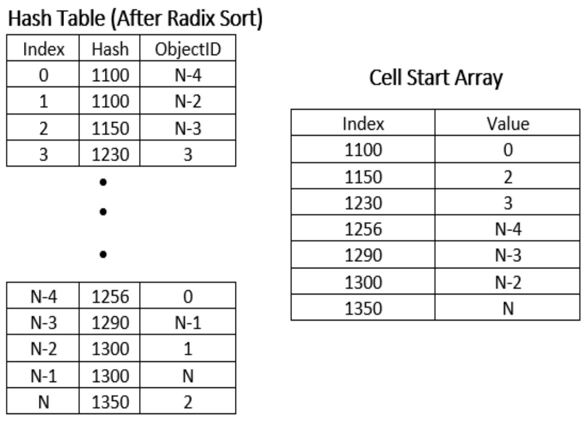
Before Radix Sort

Index	Hash	ObjectID
0	1256	0
1	1300	1
2	1350	2
3	1230	3
	•	
	•	
	•	
N-4	1100	N-4
N-3	1150	N-3
N-2	1100	N-2
N-1	1290	N-1
N	1300	N

After Radix Sort

Index	Hash	ObjectID
0	1100	N-4
1	1100	N-2
2	1150	N-3
3	1230	3
	•	
	•	
	•	
N-4	1256	0
N-3	1290	N-1
N-2	1300	1
N-1	1300	N
N	1350	2

Figure 7. Radix sort for hash table



Hash Table (After Radix Sort)

Index	Hash	ObjectID
0	1100	N-4
1	1100	N-2
2	1150	N-3
3	1230	3
	•	
	•	
	•	
N-4	1256	0
N-3	1290	N-1
N-2	1300	1
N-1	1300	N
N	1350	2

Cell Start Array

Index	Value
1100	0
1150	2
1230	3
1256	N-4
1290	N-3
1300	N-2
1350	N

Figure 8. Cell start array from the sorted hash table

2.2 Cell Start

In this stage of the spatial subdivision collision detection, a look up table is initialized to a default value of -1. Using the sorted hash table, it then maps each possible hash value to its starting index location. For instance, the hash value of 1300 occurs twice first at index (N-2) and then at (N-1) as shown in Figure 8. Therefore, in Cell Start Array (right) index 1300 corresponds to a value of (N-2). If there is no object that lies within that hash the default value of -1 will remain.

The parallel implementation of this process loads up memory locations such that each work item (GPU thread) can compare its current index hash value with the hash value of the next (neighboring) index. In this way, if the hash values are the same the work item will complete. However, when the hash values are different then the next neighboring index is the start position for the corresponding hash value and can be set in the cell start array.

2.3 Overlapping Object Pairs

In the final stage, overlapping object pairs are determined. GPU work items are created such that each item is assigned an object, whose hash value is calculated by equation (2). The hash values for the adjacent grid locations are also calculated. These values enable the work item to use the cell start array and sorted hash table to traverse through the objects within its neighborhood and adjacent to each work-item assigned object. The AABB overlap test of equation (1) is performed repeatedly with the surrounding objects. In case of collision, the corresponding collision pair is added to the collision list. Each collision pair results in the saving of both overlapping objects.

3. SEPARATING-AXIS THEOREM - NARROW PHASE COLLISION DETECTION

The collision pairs calculated from the overlapping object pairs kernel are sent to the narrow phase algorithm for collision confirmation i.e. either to be confirmed or removed. In this way, this process determines the final collision pair list. GJK has been a narrow phase algorithm mentioned earlier, but it is not easily parallelizable. Therefore, separating-axis theorem (SAT) is used for its implementation.

Visually, one can view SAT as the equivalent of finding a plane that can separate the two objects. This plane is referred to as the separating plane as depicted in Figure 9. Therefore, if a separating plane exists then the objects are not colliding and consequently when a separating plane cannot be found then the objects must be colliding. However, mathematically the separating plane is not of direct interest but rather the axis that is normal to the separating plane called the separating-axis. This is because an axis can be determined to be a separating-axis through a trivial mathematical formulation. To determine if an axis is a separating-axis, the two objects are first projected onto that target axis. The resulting projection of both objects on that axis will either overlap or not overlap. If the projections overlap that axis is not a separating axis. If the projection does not overlap, then the axis is a separating-axis and confirms the objects with no collision. Projections can be calculated by simply taking the dot product with respect to the target axis.

Theoretically, there can be an infinite number of axis to check to determine if a separating-axis exists as the axis need not to be the coordinate axis. However, two 3-D disjoint convex polytope can be separated by the planes parallel to the faces of the object or by a plane parallel to any edge. Therefore, limiting the amount of checks in practice. For example, for two object-oriented bounding box (OBB) there are 15 potential separating axes. Three unique axes forming the faces of each object and the cross product of these axis with each other (3 + 3 + 3*3). For two OBBs defining: W_N, H_N, D_N as half dimension values of the object N; N_x, N_y, N_z as the unit vector that form the edges of object N; C as the vector that is formed between the centroids of the two objects; and L as the axis. Equation (3) can be utilized to determine if L is a separating axis. If Boolean equation (3) results as TRUE, then L is a separating axis. On the other hand, if it results in a FALSE condition, then L is not a separating axis. (Gottschalk et al. 1996).

$$\left|\left|C \cdot L\right|\right| > |W_A A_x \cdot L| + \left|H_A A_y \cdot L\right| + |D_A A_z \cdot L| + |W_B B_x \cdot L| + \left|H_B B_y \cdot L\right| + |D_A B_z \cdot L| \quad (3)$$

This operation occurs in parallel by a kernel that creates a thread for each collision pair sent from the broad phase implementation. Each thread is tasked with determining if a separating axis exists between the two objects that make up the collision pair. If a separating axis is found it means the two objects are not colliding and the collision pair is removed from the list. On the other hand, if a separating axis is not found the collision is confirmed and will be present in the final collision pair list.

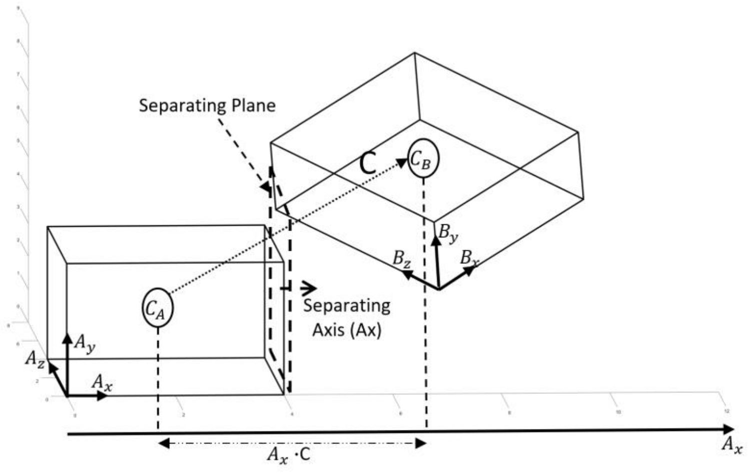


Figure 9. Separating axis theorem (SAT)

4. CPU-GPU IMPLEMENTATION

In this section, we outline and present the creation of a CPU-GPU implementation of the parallel subdivision and SAT collision detection algorithms. Figure 10 depicts the high-level GPU only implementation for the combined broad and narrow phase collision detection scheme outlined earlier. The time intervals encompassing T_{GPU1}, T_{GPU2}, T_{GPU3} and T_{BUS3} allow overlapping data transfers with execution. For simplicity, the role of CPU choreographing the data transfers and kernel execution is omitted. It can be observed that CPU resources are wasted as it does not perform any meaningful execution. Pabst et al presented a parallel subdivision-based collision detection for rigid and deformable surfaces and its CPU-GPU implementation (Pabst et al. 2010)]. They enabled CPU to do some overlapping tasks of collision detection with little inter CPU-GPU communication. We employ CPU to assist the GPU in taxing computation directly through workload sharing. Table 1 depicts the execution time of main kernels for various workloads. The profiling results point out location and timing of hotspots. It is evident that finding the overlapping pairs is a big contributor to lower execution time. The execution time for overlapping object pairs kernel for any workload setting consumes around 90% or more of the total execution time.

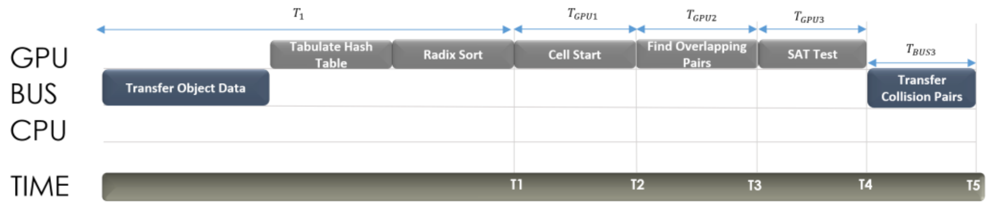


Figure 10. GPU-only implementation

4.1 Kernel Splitting

By splitting the overlapping pairs kernel, CPU and GPU are effectively assigned objects to determine the collision pairs. A CPU requires all the information about the state of the *world* to participate in a kernel execution. The *world* information is located at three structures: sorted hash table, cell start array and object data for all the objects. The third (data) structure is already placed in the main memory from where it originates. The other two structures are calculated by the GPU and therefore must be transferred before the CPU kernel execution. With all these data structures available, the CPU can be assigned a subset of objects to determine which other objects they collide with. To enable the CPU-GPU to work together in a heterogeneous environment, we argue to adapt the execution profile as shown in Figure 11. The main difference between the workload sharing and our past broad phase only implementation (Tayyub and Khan 2019) is that now the additional narrow phase part is executed immediately after finding the overlapping pairs. The narrow phase is encompassed in the kernel labeled *SAT test* whose functionality is described in section 3. We have determined that by performing the SAT kernel even on a CPU is relatively fast. As such rather than a costly transferring of the CPU output from the broad phase scheme to the GPU and allowing the GPU to perform SAT on all potential pairs. Both GPU and CPU are responsible for running SAT on their respective broad phase collision pair outputs. The additional time needed is accounted for by adapting the target execution times.

Ideally: $T_{CPU1} + T_{CPU2} = (T_{GPU1} + T_{GPU2} + T_{GPU3} + T_{BUS3}) - (T_{BUS1} + T_{BUS2})$ (4)

As the results demonstrate (in section 5) that allowing the computation power of the CPU to contribute for T_{CPU1} and T_{CPU2} reduces the overall application execution time. The total execution time for the GPU only implementation and heterogeneous implementation are defined in equations 5 and 6 respectively. T1 is omitted in both equations as it is simply an offset and same for both implementations.

$$TotalTime_{GPUonly} = T_{GPU1} + T_{GPU2} + T_{GPU3} + T_{BUS3} \quad (5)$$

$$TotalTime_{\frac{CPU}{GPU}} = \max(\max(T_{GPU1}, T_{BUS1}) + T_{BUS2} + T_{CPU1} + T_{CPU2}\,,\ T_{GPU1} + T_{GPU2} + T_{GPU3} + T_{BUS3}) \quad (6)$$

4.2 CPU-GPU Output Data Merging

CPU-GPU co-scheduling is met with a challenge of overcoming the inter CPU-GPU data transfer overhead. For kernels with discontinuous output data, the output buffers for CPU and GPU cannot be merged easily. Lee et. al (2015) use a custom kernel parser for keeping its ability to recreate, which memory addresses were modified and removes all the other operations. Using the parsed kernel, it can merge the GPU output data into main memory without using a copy buffer. Pandit and Govindarajan (2014) use the copy buffer technique. It keeps a copy of the original buffer to determine which data points the CPU has updated and merges those points into the GPU buffer.

For collision detection, the output data can be considered continuous, where each element holds a collision pair when indexing the final output collision pair buffer. Therefore, merging the collision pairs from CPU-GPU devices should not cause any overhead. To accomplish it, a collision pair wrapper class is created around the final two independent buffers that contain the collision pairs computed by CPU and GPU. The wrapper class is aware of the number of total collision pairs computed. When accessing final collision-pair data, the class is indexed to determine the buffer to return the pair from.

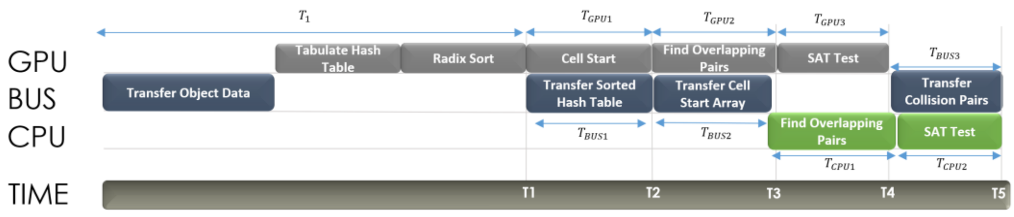


Figure 11. CPU-GPU kernel workload sharing

4.3 Workload Scheduling

To estimate an optimal workload split between the CPU and GPU, we present a successive approximation approach. We define, *r* as the ratio used to determine the amount of work given to the CPU. We define the upper threshold (r_{up}) that represents the fastest execution time (T_{up}) for which *r* is the largest (faster CPU). Similarly, the lower threshold (r_{dwn}) represents the fastest execution time (T_{dwn}) for which *r* is the lowest (faster GPU) and current ratio (r_{cur}) is the midpoint between upper and lower threshold ratios with a corresponding execution time defined as T_{cur}.

$$CPU_{work} = r * total_{work} \quad (7)$$

$$GPU_{work} = (1 - r) * total_{work} \quad (8)$$

Figure 12 provides a process of our successive approximation approach. It begins by initializing the parameters by running an iteration of the kernel at $r_{up} = 1$, $r_{dwn} = 0$ and $r_{cur} = 0.5$. For every iteration, total execution time is profiled. The maximum of the two execution times T_{up} and T_{dwn} are selected and compared with T_{cur}. If T_{up} is chosen and $T_{cur} < T_{up}$ then $r_{up} = r_{cur}$ and r_{dwn} is unchanged. Similarly, if T_{dwn} is chosen and $T_{cur} < T_{dwn}$ then $r_{dwn} = r_{cur}$ and r_{up} is unchanged. The new value of r_{cur} becomes the midpoint between r_{up} and r_{dwn}. An iteration is then executed with the new ratio. This process continues till either the new ratio results in worse performance than both r_{up} and r_{dwn} or the number of user defined iterations end. At the end of this process, our algorithm chooses the ratio that resulted in the best performance to execute remaining kernel instances.

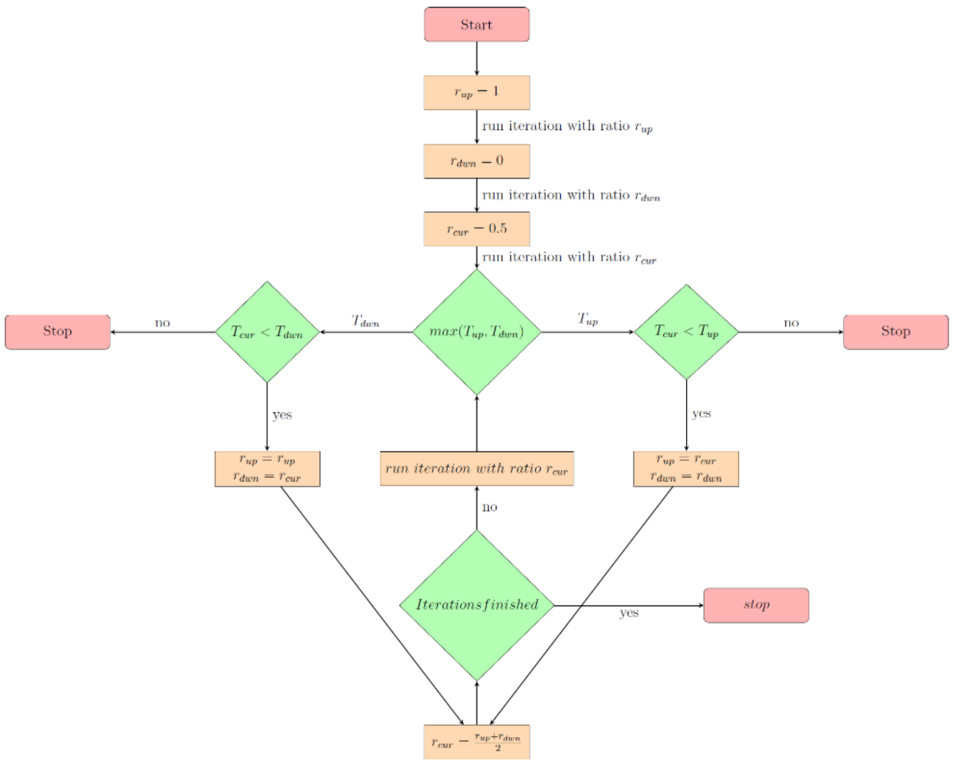

Start
$r_{up} = 1$
run iteration with ratio r_{up}
$r_{dwn} = 0$
run iteration with ratio r_{dwn}
$r_{cur} = 0.5$
run iteration with ratio r_{cur}
Stop
no
$T_{cur} < T_{dwn}$
T_{dwn}
$max(T_{up}, T_{dwn})$
T_{up}
$T_{cur} < T_{up}$
no
Stop
yes
yes
$r_{up} = r_{up}$ $r_{dwn} = r_{cur}$
run iteration with ratio r_{cur}
$r_{up} = r_{cur}$ $r_{dwn} = r_{dwn}$
no
Iterationsfinished
yes
stop
$r_{cur} = \frac{r_{up}+r_{dwn}}{2}$

Figure 12. Successive approximation approach

5. EXPERIMENTAL RESULTS

The workload scheduling (section 4) and collision detection algorithm (section 3) are partitioned by employing the CPU as a co-processor to the GPU. However, there is nothing about both the successive approximation approach and the collision detection scheme that is inherently CPU/GPU specific. It can be extended to multiple platforms and since the framework was written in OpenCL it does not require additional steps to run on other platforms. As such another common scenario is that a computer system is comprised of a CPU and a discrete GPU connected through a PCI-e bus where the CPU already contains an integrated GPU (IGPU). This is done as discrete GPUs are generally high orders of magnitude faster than the IGPU. Typically, once a user installs a discrete GPU, the IGPU is disabled and never used. However, this is a waste of silicon space as it contains a considerable amount of compute power and regardless of the usage, it exists. To test both the successive approximation approach and workload sharing collision detection scheme under various scenarios, two systems outlined in Table 1 are utilized. For system# 2 the IGPU scenario is omitted as the Intel Xenon E5-2620 does not contain an IGPU.

Table 1. System configurations

System #	CPU	IGPU	GPU (discrete)
1	Intel Core i5-9600K	Intel UHD Graphics 630	AMD RX 570
2	2 x Intel Xeon E5-2620	N/A	NVIDIA Tesla K20

For software test configuration a *world* dimension with a grid-size of 200 x 200 x 200 is created with each object dimension set to 1 x 1 x 1, which also defines the grid cell size. For heterogeneous implementation, an offline profile is first created for various workload tests. The allowed iterations for the successive approximation are capped to ten. Each benchmark is executed ten times and all results presented are the average of these ten runs.

Table 2 summarizes the profiling results when executing the collision detection on GPU alone. Values are presented as a percentage of total execution time. Finding overlapping pairs consumes ~90% or more of total execution time. Thus, it was the focus for workload splitting presented in section 4.

Table 2. Profiling results from GPU only execution (% of total execution time) / System# 1

Objects (million)	Hash Tabulation	Radix Sort	Cell Start	Find Overlapping Pairs	SAT Test	Transfer Collision Pairs
0.25	1.4	0.5	4.7	89.3	1.9	2.2
0.5	4.1	0.4	4.8	88.5	1.1	1.0
1	1.3	0.2	1.5	95.7	0.7	0.6
2	2.0	0.1	0.8	96.4	0.3	0.3

Figures 13 and 14 depict the total execution time for both systems under various configurations. For single device computation the results are consistent across all the workloads and according to the expectation. GPU is the fastest followed by IGPU (integrated GPU) in the case of system# 1. With CPU (for parallel implementations) is the slowest compute device in both cases. When workload sharing, all the configurations are faster than executing the application on GPU alone. However, performance improvement is varying depending on the workload size. Furthermore, best suitable configuration is also different depending on system and workload size. We speculate these changes relate to the drastic differences in clock speed and core count between compute devices. Thus, creating tradeoffs that make them unsuitable for different workload sizes. The performance results of workload sharing compared to GPU only execution is summarized and depicted in Figure 15.

6. CONCLUSIONS

We have achieved a considerable performance increase for our efficient CPU-GPU implementation of collision detection across multiple system and workload configurations. A peak performance increase of approximately 18% is achieved on system# 1 with a GPU/IGPU configuration for a 0.5 million object workload. However, under smaller workloads (0.25 million objects) system# 1 exhibits better performance with a GPU/CPU configuration (~15% compared with ~8%). This is most likely due to the Intel UHD not being fully saturated at small workloads allowing the much higher clock speed and lower latency of the Intel Core i5 to be better suited. For system# 1 configurations, performance degrades after their respective performance peaks. However, it still maintains a better performance as compared to GPU only execution. Moreover, the GPU/IGPU configuration degrades less rapidly than the GPU/CPU configuration. For system# 2 the average performance increase is less than that of both configurations of system# 1. However, it maintains a marginal increase over GPU only execution for most workload settings. Unlike system# 1, the configurations for system #2; the trend is that performance increases slightly with the workload increases. It can be due to the inherent tradeoff of having two very powerful multi-core CPUs (2xIntel Xenon) compared to one faster CPU (Intel Core i5).

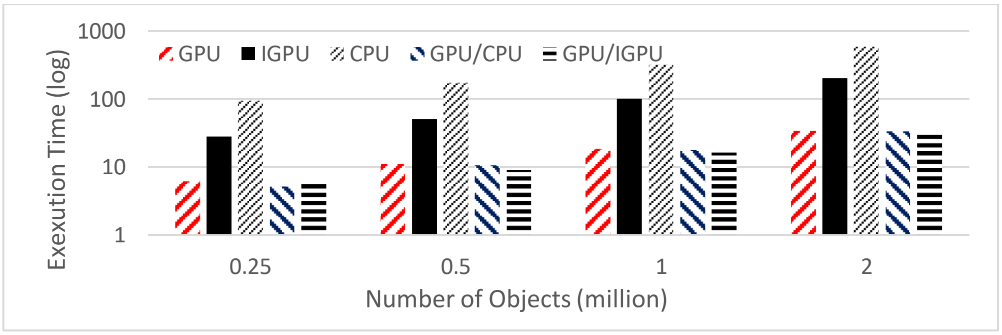


Figure 13. Total execution time for various configurations (System# 1)

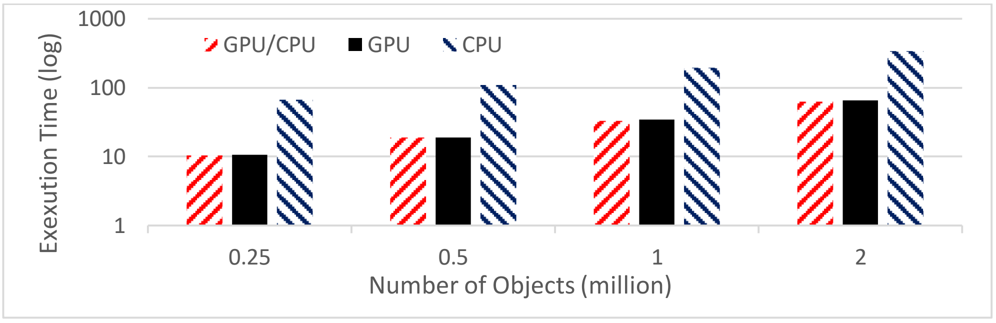


Figure 14. Total execution time for various configurations (System# 2)

In this paper, we took an interesting and real-world application of collision detection and showcase a highly parallel GPU/CPU broad and narrow phase collision detection CPU-GPU implementation. We have also presented a methodology to determine an optimal workload partition ratio that is applicable to multiple system configurations. The efficacy of the partitioning method was proven for the collision detection algorithm resulting in significant performance increases over GPU only execution.

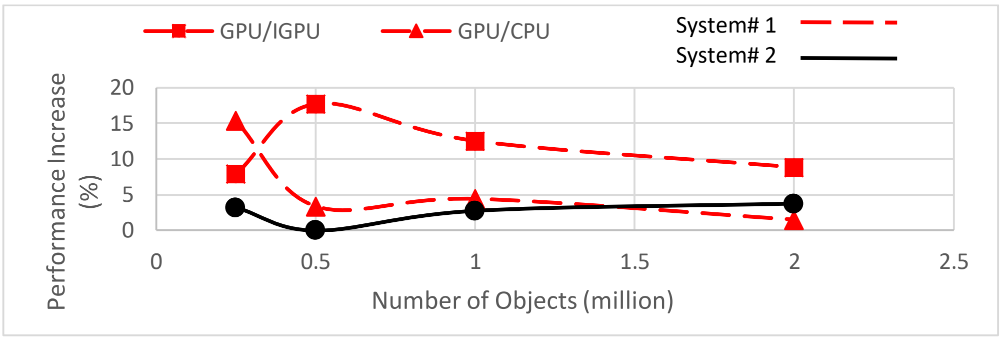


Figure 15. Performance increase with successive approximation approach

ACKNOWLEDGEMENT

This research is partly supported by NSERC Canada, an equipment Grant from CMC Microsystems Canada and FEAS Ryerson Univ. Dean’s Research Grant.

REFERENCES

Bullet Real-Time Physics Simulation, 2019. Home of Bullet and PyBullet: Physics Simulation for Games, Visual Effects, Robotics and Reinforcement Learning. Available: https://pybullet.org/wordpress/ (Accessed: 7 December 2019).

Chapala, S. R., et al, 2016. Determination of coordinate transformations in UAVS. *Proceedings Second International Conference on Cognitive Computing and Information Processing*, pp. 1–5.

Coming, D. S. and Staadt, O. G., 2006. Kinetic Sweep and Prune for Multi-Body Continuous Motion. *In Computers and Graphics.* Vol. 30, No. 3, pp. 439–449.

Delorme, M. C., et al, 2013. Parallel Radix Sort on the AMD Fusion Accelerated Processing Unit. *Proceedings 2nd Int. Conf. Parallel Processing*, Lyon, France, pp. 339–348.

Ericson, C., 2004. *Real-Time Collision Detection*. CRC Press, Inc., Boca Raton, FL, USA.

Gottschalk, S., et al, 1996. OBB-Tree: a hierarchical structure for rapid interference detection. *Proceedings ACM SIGGRAPH*, pp. 171–180.

Huang, R., 2012. Optimizing Collision Detection in 3D Games with Model Attribute and Bounding Boxes. *Proceedings IEEE Symp. Electrical and Electronics Engineering.* Kuala Lumpur, Malaysia, pp. 589–591.

Jiménez, P., et al, 2001. 3D Collision Detection: A Survey. *In Computers and Graphics*. Vol. 25 No. 2, pp. 269–285.

Lee J., et al, 2015. SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration. *In ACM Transactions on Computer Systems*, Vol. 33, No. 3, pp. 9:1–27.

Mirtich, B., 1997. Efficient Algorithms for Two-Phase Collision Detection. *In Practical Motion Planning in Robotics: Current Approaches and Future Directions* (Eds. Gupta K. and del Pobil A.P.) John Wiley and Sons Canada. pp. 203–223.

Moore, M. and Wilhelms, J., 1988. Collision Detection and Response for Computer Animation. *Proceedings ACM 15th Annual Conf. on Computer Graphics and Interactive Techniques*. Atlanta, Georgia USA, pp. 289–298.

Ng K. W., et al, 2012. Collision Detection Optimization on Mobile Device for Shoot'em up Game. *Proceedings Int. Conf. Computer and Information Science.* Kuala Lumpur, Malaysia, pp. 464–468.

Nguyen, H. (ed) 2007. *GPU GEMS 3,* Chapter 32, Broad-Phase Collision Detection with CUDA. Addison Wesley Professional: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch32.html (Accessed: 7 December 2019).

Pabst, S., et al, 2010. Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces. *In Computer Graphics Forum*, Vol. 29, No. 5, pp. 1605-1612.

Pandit, P. and Govindarajan, R., 2014. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. *Proceedings ACM Int. Symp. Code Generation and Optimization*, Orlando, FL, USA, pp. 273–283.

Satish, N., et al, 2009. Designing Efficient Sorting Algorithms for Manycore GPUs. *Proceedings IEEE Int. Symp. on Parallel and Distributed Processing*, Rome, Italy, pp. 1–10.

Stone, J. E., et al, 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *In Computing in Science and Engineering*, Vol. 12, No. 3, pp. 66–73.

Tayyub, M. and Khan, G. N., 2019. Heterogeneous CPU-GPU Implementation of Collision Detection. *Proceedings 16th International Conference on Applied Computing*, Cagliari, Italy, pp. 71–78.

Teschner, M., et al, 2003. Optimized Spatial Hashing for Collision Detection of Deformable Objects. *Proceedings Vision, Modeling and Visualization*, pp. 47–54.

Teschner, M., et al, 2005. Collision Detection for Deformable Objects. *In Computer Graphics Forum*. Vol. 24, No. 1, pp. 61–81.

Tracy, D. J., et al, 2009. Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal. *In IEEE Virtual Reality Conference*, Lafayette, LA, 2009, pp. 191–198.

Wong, T. H., et al, 2014. An Adaptive Octree Grid for GPU-Based Collision Detection of Deformable Objects. *In The Visual Computer*, Vol. 30, Issue 6-8, pp. 729–738.

Zahmann, G., 2002. Minimal Hierarchical Collision Detection. *Proceedings ACM Symp. Virtual Reality Software and Technology*, Hong Kong, pp. 121–128.