A hardware accelerator implementation for real-time collision detection

Fredy Augusto M. Alves[1], Lucas Bragança da Silva[1], Ricardo S. Ferreira[3], Peter Jamieson[2], José A. Miranda Nacif[1]
[1]Institute of Exact Sciences and Technology, Federal University of Viçosa, Campus UFV-Florestal, Brazil
[2]College of Engineering and Computing, Miami University, USA
[3]Departament of Informatics, UFV, Brazil
fredy.alves@ufv.br, lucas.braganca@ufv.br, ricardo@ufv.br, jamiespa@miamioh.edu, jnacif@ufv.br

***Index Terms*—Parallel Processor, Hardware Accelerator, Computer Architecture, Collision Detection**

I. Abstract

Collision detection algorithms are used to detect when virtual objects collide and the results of these collisions. This type of algorithm is used by many research areas such as simulation. automatic path finding, tolerance checking, among others. This type of algorithm is processed in real-time. Intel created the HARP Platform which uses a PCI with 8GB/s bandwidth, this decreased the memory latency allowing real-time processing of fine grain applications such as collision detection algorithms. In this paper we propose a heterogeneous system in order to use the FPGA on the HARP as an accelerator for collision detection algorithms. Our results show a speedup of 14.5% in execution time.

II. Introduction

With the recent acquisition of Altera by Intel, the tendency is that FPGAs will be included in CPUs as accelerators for a wide range of applications. Collision detection algorithms are used to detect when virtual objects collide and the results of these collisions. This type of algorithm is used by many research areas such as simulation. automatic path finding, tolerance checking, among others. Applications can be found in games, factory simulators, medical procedures training applications, virtual reality, etc [1].

Collision detection in games and simulators are usually implemented in Engines in the form of APIs which can be used out-of-the-box. Collision detection in applications such as games and simulators is done in real-time which means that collisions are processed as soon as they happen. With the increasing complexity of the applications that use this type of algorithm in order to achieve a better resemblance with reality, the number of collisions for each simulation step has been increasing so finding ways to improve its the efficiency is crucial.

In a single simulation step we have access to a set of collision data. One of the most important advantage of FPGAs over CPUs is its capacity for parallel processing which is ideal for collision detection algorithms. Before the acquisition of Altera by Intel, the use of FPGAs as accelerators for this type of algorithm was highly affected by memory latency but this changed with the new platform HARP (Heterogeneous Accelerator Research Platform) by Intel which consists of a Xeon Processor connected to an Altera Stratix V FPGA by a PCI of 8GB/s bandwidth which reduces drastically the memory latency. The ODE [2] (Open Dynamics Environment) is an open-source Engine used by a wide variety of games and simulators.

HARP uses the AAL (Accelerator Abstraction Layer) framework to develop applications for it, it uses the service abstraction for implementing applications for accelerators which allows the user to implement a method for accelerating an application which can be called as any other function in a C++ application such as ODE.

In this paper we developed a novel hardware accelerator for spheres collision detection as a proof of concept for optimizing this type of algorithm on FPGAs. It uses a many-processing units approach to process collisions in parallel with a producer-consumer approach in order to improve communication efficiency, it also uses pipeline methods to improve the throughput. The algorithm uses floating point arithmetic. We tested the design on the HARP platform and collected results in real-time for many ODE simulation benchmarks. We managed to get an improvement of up to 14.5%.

This paper is outlined as follows. In section III we briefly explain the background of the collision detection algorithm and its implementation on the HARP platform. In section IV, we explain how our results were fetched. In section V, we put our work into perspective. In section VI we present and discuss our results. Then, in section VII we conclude our work and explain the future works for it.

III. Background

A. Collision detection algorithm

The ODE offers a collision detection engine, it also provides an interface for reimplementing the methods on it. This engine receives information about the shape and position of each body in a space. At each simulation step, the collision engine is responsible for identifying which bodies are colliding and returning contact points which are structures responsible for holding information about the collision.

B. Contact points

Contact points are structs returned by the collision detection engine, they are composed by three attributes:

- Pos: The position of the contact point in a space.
- Normal: The resultant vector that is perpendicular to the collision contact point.
- Depth: The depth of penetration of the bodies in each other.
- G1 and G2: The bodies colliding for the contact point.

C. Sphere Collision detection algorithm

In our case of study, we reimplemented the method for collision detection between two spheres, the pseudocode for this algorithm is on Algorithm 1 with its inner methods. The inputs are the position of two spheres in a space (p1 and p2) and their respective radius (r1 and r2), all the data is in 32-bit ieee 754 floating point format. The output is a contact point for the collision.

The algorithm begins by calculating the depth for the collision and storing it on the variable *d*. Based on d, the algorithm has three possible execution flows. Flow 1 is when the collision doesn't happen, we call this a *fake collision*, Flow 2 is when the bodies barely touch each other, this is a *grazing collision*, Flow 3 is when the bodies collide with each other and the collision has a depth, a *real collision*. Flow 2 almost never happens so we focus our studies on Flow 1 and 3.

By studying the data dependencies on the algorithm, we were able to identify which calculations could be executed in parallel. We divided the calculations in stages which happens is sequence in a parallel processor such as an FPGA, the stages are listed on Algorithm 2. In this project we developed a Verilog RTL Design to execute those stages.

As we are dealing with bodies in a 3D Space, *p1*, *p2*,*cNormal*, *cDepth*,*cPos* are all vectors of size 3 to hold the x, y and z coordinates. This means that each operation on Algorithm 1 which uses any of these vectors is actually 3 operations.

D. The System architecture

The system architecture overview can be seen on Figure 1, on the FPGA side, in addition to the Accelerator Function Unit (AFU) for the collision detection algorithm on the FPGA, the system also uses another component provided by Intel, the System Protocol Layer 2 (SPL2) which is responsible for memory address translation since the FPGA uses virtual addressing, the shared memory can go up to 2GB.

The shared memory is divided in sections, the device status memory (DSM) is a 4kb pinned memory region which holds the status of the device. The control memory is a reserved memory space for handshake signals between the CPU and the FPGA, it is used for start and reset signals. The Src Buffer is used by the CPU to send data to the AFU and the Dst Buffer is used by the FPGA to send results to the SW application. The VAFU2_CNTXT holds the pointers to the buffers.

The CPU side is composed by the ODE simulations and the AAL application used to communicate with the AFU.

Algorithm 1 Spheres collision detection.

$d \leftarrow$ dCALCPOINTSDISTANCE3(p1,p2)
Flow 1: Fake Collision
if $d >$ (r1 + $r2$) **then return** 0
end if
Flow 2: Grazing Collision
if $d \leq 0$ **then**
  $cPos \leftarrow p1$
  $cNormal \leftarrow (1, 0, 0)$
  $cDepth \leftarrow r1 + r2$
  Flow 3: Real Collision
else
  $d1 \leftarrow$ dRECIP(d)
  $cNormal \leftarrow (p1 - p2) * d1$
  $k \leftarrow 0.5 * (r2 - r1 - d)$
  $cPos \leftarrow p1 + cNormal * k$
  $cDepth \leftarrow r1 + r2 - d$
end if
function dCALCPOINTSDISTANCE3($p1$,$p2$)
  $tmp \leftarrow$ dSUBTRACTVECTORS3(p1,p2)
  $res \leftarrow$ dCALCVECTORLENGTH3(tmp)
  return res
end function
function dSUBTRACTVECTORS3($p1$,$p2$)
  $res \leftarrow p1 - p2$
end function
function dCALCVECTORLENGTH3(tmp)
  $res \leftarrow sqrt(tmp[0]*tmp[0]+tmp[1]*tmp[1]+tmp[2]*tmp[2])$
end function

Algorithm 2 Paralell Spheres collision detection.

Stage 1 :
$d \leftarrow$ dCALCPOINTSDISTANCE3(p1,p2)
$rsum \leftarrow r1 + r2$
$psub \leftarrow p1 - p2$
$rsub \leftarrow r2 - r1$
Stage 2 :
$d1 \leftarrow$ dRECIP(d)
$d > rsum$
$r2 - r1 - d$
$r1 + r2 - d$
Stage 3 :
$cNormal \leftarrow (psub) * d1$
$k \leftarrow 0.5 * (rsub - d)$
Stage 4 :
$cnk \leftarrow cNormal * k$
Stage 5 :
$cPos \leftarrow p1 + cnk$

E. The Sphere Collision Detection AFU

The AFU is divided into two main components as seen on the datapath on Figure 2, the Processing Units Controller (PUC) and the Sphere Collision Processing Units (SCPUs),

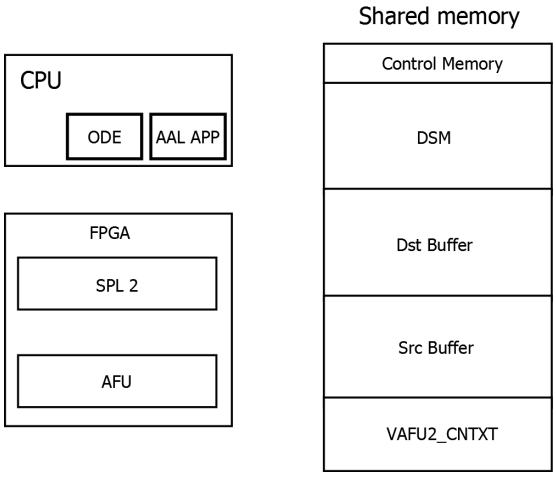


Figura 1. System architecture overview.

The handshake signals control protocol is implemented on the PUC, it also implements a collector responsible for fetching collision data from the Src Buffer and a Dispatcher responsible for writing the collisions processing results to the Dst Buffer. Besides the algorithm parallelism based on data dependence described on Section III-A, in our architecture we also use a many Processing Units approach, each SCPU process one collision in parallel to the other SCPUs. So, if we have N SCPUs, the collision execution time T for one collision processing becomes T/N. The PUC also controls the start and reset signals for the SCPUs, it receives from the CPU AAL application the number of collisions to be processed Ncol, it then execute each SCPU Ncol/N times.

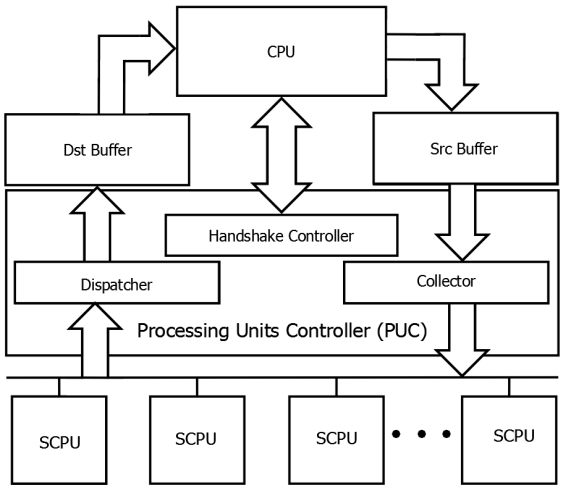


Figura 2. the AFU datapath.

IV. Methodology

A. Collision detection execution time on AFU

The AAL framework is service-oriented so it works with the concept of transactions. A transaction must be initiated, processed and stopped. The steps for executing a transaction from the AAL to the AFU is presented on Figure 3 and works as follows:

1) The AAL application gets pointers to the src buffer (pSource), destination buffer (pDest) and AFU context (pVAFU2_cntxt).
2) The application sends the collision Data to be processed to the Source Buffer and starts the SPL.
3) The application sends the start signal and how many collisions (nCol) should be processed to the AFU through the control memory.
4) The AFU writes its AFU ID to the DSM which is used to signal to the CPU that it is running.
5) After the AFU finishes processing the collisions, it signals to the CPU that it is done.
6) The CPU uses pDest to retrieve the collision processing results from the Destination Buffer.
7) The application ends the transaction by freeing the workspace.

The application re-execute the collision processing on the AFU X times and compute an average of all the execution times, this process is done in order to minimize the variations on the method for fetching execution time. As the path for executing a type of collision is always the same, it is expected that the execution time for a certain amount of collisions is always similar.

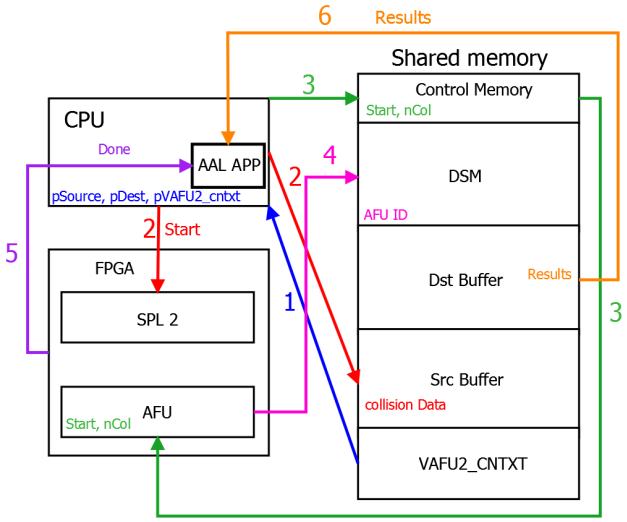
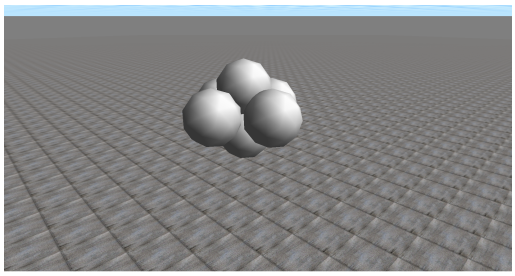


Figura 3. Transaction steps.

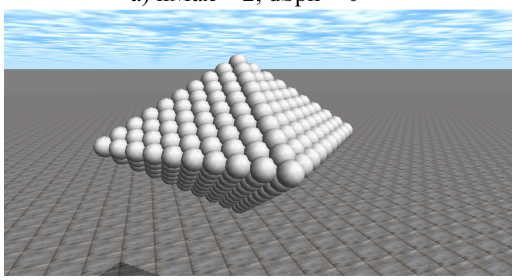
B. Collision detection execution time on ODE

For the ODE, a set of parameterizable benchmarks was developed. The benchmark creates a simulation environment with spheres organized in the form of a diamond, it receives the maximum height hMax for the diamond from its center as an input parameter in number of spheres. Figure 4 shows examples of three benchmarks and their respective hMax. It is also possible to increase the distance dSph between the spheres. After the spheres are created, a force is applied in

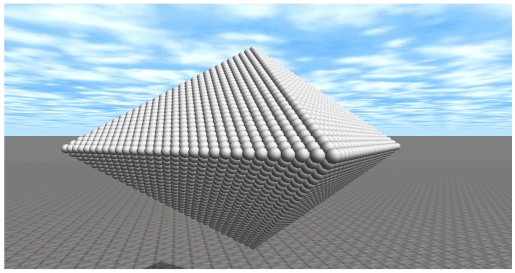
each one pointing to the center of the diamond so they can collide.



a) hMax = 2, dSph = 0



b) hMax = 10, dSph = 0



c) hMax = 30, dSph = 0

Figura 4. Benchmarks.

In order to get the execution time, the user must specify how many collisions and their respective type tCol (fake, real or grazing) should be taken into account. Figure 5 shows the execution flow for getting the ODE spheres collision execution times. First the benchmark parameters are set, the simulation starts, when an execution of the specified type happens, it is executed 10 times and the execution time average is added to the total execution time, if the number of collisions nCol is equal to the maximum number of collisions nColMax, the simulation stops and the benchmark returns the total execution time. The benchmarks were created in order to take into account any optimization made by the processor in a real time application.

V. Related Works

On [3] an algorithm to solve linear programs is used in order to improve the speed of collision detection algorithms, this paper uses preloaded data with memory initialization files in an FPGA kit to generate its results. On [4], a collision detection design for FPGAs using fixed-point arithmetics and bounded error is proposed, its focus is on saving space in order to improve area overhead, its results are given in terms

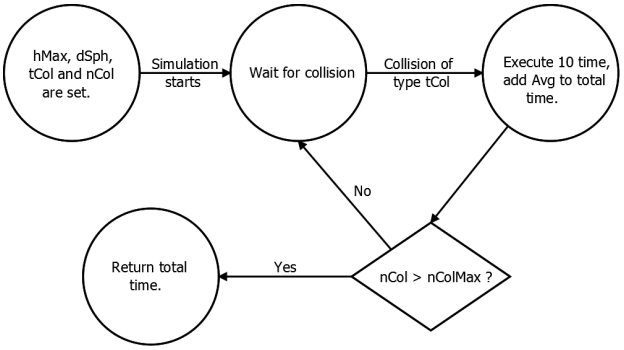


Figura 5. ODE execution flow.

of speedup but it also uses preloaded data. Works such as [5] [6] relies on GPU-CPU based systems to improve collision detection.

The main differentials for our work is that we propose a real-time collision detection design for real-time processing on a CPU-FPGA heterogeneous system without preloaded data, our results are executed in a real environment and are given in terms of execution time speedup.

VI. Results and Discussion

Our experiments were conducted by executing sets of real collisions on both ODE and the AFU. For the ODE, the tests were made using a benchmark of hMax equal to 3 and dSph equal to 0. We started with a set of 10 collisions and then kept increasing it by 10 until we reached 1000 collisions. We presented the results in terms of collisions processing time in a graph comparing ODE to the AFU times. For the AFU we used 10 SCPUs.

Figure 6 shows the graph for the sets of real collisions, the increasing in execution time is linear as expected since the path to execute a real collision is always the same. Above 250 collisions, the AFU is always better than the ODE, the reason why before this point the ODE is better is because the AFU has a configuration time before and after each transaction, this configuration time is on average 2000 milliseconds so the AFU must reach a minimum of collisions in order to compensate this. With the number of collisions that we collected, we can reach up to 14.5% speedup on the AFU for real collisions.

We then implemented another test where we start with a set of 1000 collisions and increase it by 1000 until we reach 90000 collisions, the results can be seen in Figure 7. For the HARP tests, the average execution time per collision was 86ns and for the ODE it was 102ns.

VII. Conclusion and Future Work

In this work we presented a hardware accelerator for a collision detection algorithm. We implemented a case of study with Sphere Collision Detection reaching a speedup of up to 14.5% with FPGA proven designs running in a real environment. The results show that our design is a great option for decreasing execution times for this type of algorithm in this new FPGA accelerators reality. As for future works, we

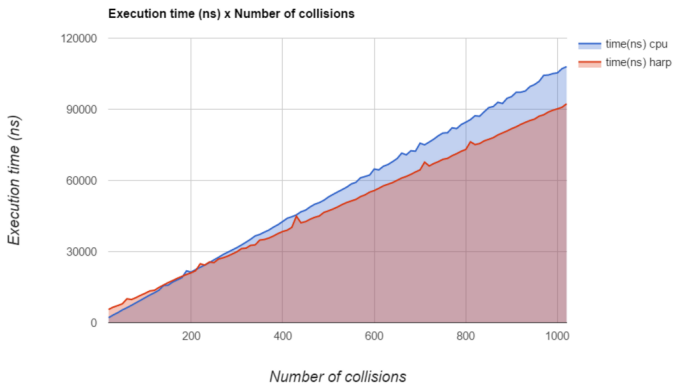


Figura 6. Graph for real collisions for 1000 set.

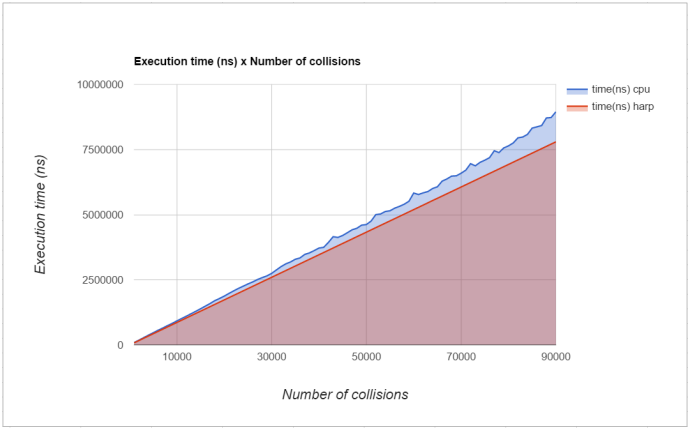


Figura 7. Graph for real collisions for 100000 set.

intend to expand the AFU library to another type of collisions detection algorithms in order to produce a new Engine.

ACKNOWLEDGMENTS

We would like to thank UFV for providing us the necessary materials and support, we also would like to acknowledge the students Connor Blandford and Oakley Katterheinrich for participating on the development of the SCPUs during a summer internship on the Miami University.

REFERÊNCIAS

[1] H. Wei and W. W. Gen, "A comprehensive fpga implementation of collision detection," in *IET International Communication Conference on Wireless Mobile and Computing (CCWMC 2011)*, Nov 2011, pp. 341–346.

[2] R. Smith. Open dynamics engine. [Online]. Available: http://www.ode.org/

[3] C. H. Wu, S. O. Memik, and S. Mehrotra, "Fpga implementation of the interior-point algorithm with applications to collision detection," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, April 2009, pp. 295–298.

[4] A. Raabe, S. Hochgurtel, J. Anlauf, and G. Zachmann, "Space-efficient fpga-accelerated collision detection for virtual prototyping," in *Proceedings of the Design Automation Test in Europe Conference*, vol. 2, March 2006, pp. 6 pp.–.

[5] A. Hermann, F. Drews, J. Bauer, S. Klemm, A. Roennau, and R. Dillmann, "Unified gpu voxel collision detection for mobile manipulation planning," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept 2014, pp. 4154–4160.

[6] A. Hermann, F. Mauch, K. Fischnaller, S. Klemm, A. Roennau, and R. Dillmann, "Anticipate your surroundings: Predictive collision detection between dynamic obstacles and planned robot trajectories on the gpu," in *2015 European Conference on Mobile Robots (ECMR)*, Sept 2015, pp. 1–8.