Designing a Collision Detection Accelerator on a Heterogeneous CPU-FPGA Platform

Fredy Augusto M. Alves[1], Peter Jamieson[2], Lucas B. da Silva[1], Ricardo S. Ferreira[3], José Augusto M. Nacif[1]
[1]Science and Technology Institute, Universidade Federal de Viçosa, Florestal, Brazil
[2]Department of Electrical and Computer Engineering, Miami University, USA
[3]Departament of Informatics, Universidade Federal de Viçosa, Viçosa, Brazil
{fredy.alves, lucas.braganca, ricardo, jnacif}@ufv.br, jamiespa@miamioh.edu

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

Abstract*—Collision detection algorithms are used to detect when virtual objects collide with one another and calculate the results of these collisions. These types of algorithms are, typically, critical real-time calculations needed for applications such as simulation, tolerance checking, and video games. In this work, we present an implementation of the smallest piece of a collision detection pipeline implemented on Intel's Heterogeneous CPU-FPGA Platform. This platform includes both an FPGA and CPU that allows real-time processing of fine grained applications such as collision detection. We present a heterogeneous implementation that uses the FPGA to accelerate a particular collision detection stage as an accelerated part of a complete collision detection pipeline on a real system to demonstrate how collision detection can benefit from co-processing even in its worst-case implementation. We believe as Intel continues to integrate FPGAs with processors on a single die that algorithms like these need to be both optimized and open sourced to the general computing community[1] so that they can be included and studied as part of a full simulation system where the GPU is dedicated to graphics and the CPU cores to world management. In our case, our results show a speedup of 14.81% with the FPGA as compared to a CPU only implementation. The importance of this result is that it demonstrates that even the worst implementation in terms of data communication to computation ratio for collision detection on a real heterogeneous system can be used as an accelerator, and more importantly, this is the starting point for researchers to investigate where these algorithms should be located in the system whether on the traditional CPU cores, the GPU, or a co-processing FPGA.

I. Introduction

Hardware accelerators or co-processors are being used as alternatives for high-performance computing [1], [2]. Recently, industry vendors such as Microsoft, Intel, Xilinx, and IBM have released heterogeneous CPU-FPGA platforms that show promising results in terms of performance and energy efficiency. Intel is working on a commercial CPU-FPGA platform that integrates both CPU and FPGA on the same die increasing the bandwidth available for these devices. Because of these trends, various algorithms that, typically, would run on a CPU should be examined in terms of being accelerated and included as pieces of larger computation systems. At the least, system designers need to understand the trade-offs of putting these algorithms on the ever increasing heterogeneous computing choices that include CPUs, Graphics Processing Units (GPUs), and FPGAs.

One domain, video games, has pushed capabilities of PCs including the invention of the GPU and the performance of CPUs and memory. This suggests that algorithms used in games should be explored as viable implementations on a co-processing FPGA as these new computation units become a part of modern systems. For this reason, our work looks at accelerating collision detection algorithms that detect when virtual objects collide and then calculate the result of these collisions implemented on a real system. These algorithms are not only employed in games, which are virtual simulations of game worlds, but are used in other areas such as general physics simulation, manufacturing simulators, medical procedures training applications, virtual reality, etc [3]. For instance, General Electric continues to develop technologies to implement virtual twins where a real product has a virtual twin that can be used to observe, test, diagnose, and help in design. These "Mirror Worlds" require sophisticated computation including collision detection [4].

In this work, we present a hardware accelerator for spheres collision detection on an FPGA as part of the Intel CPU-FPGA platform. We use the Open Dynamics Environment (ODE) [5] as the open-source engine that includes a collision detection pipeline (CDP), and this tool is used by a wide variety of games and simulators. Our FPGA implementation uses a many-processing units approach to process collisions in parallel in order to improve the efficiency of these calculations and implements the finest grained piece of the CDP. We tested our design on the Intel CPU-FPGA platform and collected results showing a speedup of 14.81%. Though this number is not large or comparable to other implementations, the importance of this result is, one, it represents the finest grain of a CDP meaning we can expect improvements as we coarsen this, and two, this is the first implementation of collision detection on a heterogeneous system where shared memory and synchronization are considered.

The contribution of this work is twofold: First, we implement a heterogeneous implementation to perform fine-grained sphere collision detection in parallel using the Intel CPU-FPGA platform. Second, we analyze the memory/processing bottlenecks of our implementation in terms of performance.

[1]https://github.com/fredyamalves/Collision-detection-for-a-CPU-FPGA-heterogeneous-System

978-1-5386-3797-5/17/$31.00

Moreover, we have made our accelerator source code available for the community to reproduce the results and use our design as a starting point to implement similar applications and decide where these calculations should be done. Our results suggest that the FPGA will suffice as a useful co-processor for even the finest-grained collision detection calculations, and these results will only improve as both the CPU-FPGA system is more tightly integrated in terms of memory access and communication and more course-grained calculations in the collision detection are moved onto the FPGA.

II. Related Work

Many researchers have implemented collision detection algorithms. For a good introductory review of collision detection and the collision detection pipeline (CDP), we suggest chapter 2 of Weller's book [6]. The CDP consists of two major steps called a "broad phase" and a "narrow phase" where the narrow phase can be further split into finer detailed calculation steps. The purpose of the CDP is to filter out non-collisions between objects with coarse calculations that save computation resources, and finer more costly calculations are done after filtering out non collisions leaving only objects that have a high potential to collide. In this work, we look at implementing the finest-grained portion of the CDP as implemented in the ODE software.

Some example work of improvements to collision detection on various computational devices include the following. Wu *et al.* [7] created an algorithm on an FPGA to solve linear programs and used it to improve the speed of collision detection algorithms, this work pre-loads data with memory initialization files to generate results meaning all of the data needed for the calculations is already stored in memory before experimental timing is started. In Raabe *et al.* [8], a collision detection design for FPGAs uses fixed-point arithmetic and bounded error, and the focus is on saving space in order to improve area overhead. Their results have a speedup, and again the data is pre-loaded. Works such as [9], [10] use a GPU-CPU based system to improve collision detection algorithms. More, recently, a paper by Zhang *et. al.* showed an 8x improvement in speed to collision detection on an FPGA where the entire system is implemented on the FPGA [11].

We note that our work differs, significantly, from this literature in the following key ways: 1) We do not make use of pre-loaded data and take into account the shared memory access time in our speedup results, which is left out in all existing FPGA implementations and sometimes in GPU implementations; 2) We implement the most fine-grained piece of the CDP on a CPU-FPGA platform as a demonstration that even the finest calculations in the CDP will provide some benefit when co-processed by the FPGA. In addition to these key differences, our system is compared to a single core CPU implementation and not to a GPU implementation. The reason for this is we are looking at the benefit of implementing collision detection on a co-processor where in applications such as video games the GPU is not available for significant computations as it is already dedicated to its' primary purpose. Therefore, our goal is not just acceleration of the application, but evidence that the FPGA co-processor is a viable target for these algorithms and there is room for significant improvement of collision detection in these real-systems. This type of work is needed on a larger scale so that system designers can determine where various computation activities should be executed according to trade-offs in speed and power.

III. Collision Detection Hardware Accelerator

In this Section we provide details of our collision detection hardware accelerator.

A. Collision Detection Algorithms

The ODE software includes a collision detection engine that allows for re-implementation of its calculating methods. This engine uses information about the shape and position of each object in the world. The collision detection algorithm is divided in 3 steps as seen in Figure 1 including where the calculations are performed in our system:

- AABB collision detection: The ODE on the CPU executes the Axis-Aligned Bounding Box (AABB) collision detection algorithm, an AABB is basically a box which describes one or a set of geometric figures, in the case of a sphere, the AABB is a cube inside it as seen in figure 2, this step only identifies which objects are potentially colliding and then it sends the objects to their specialized collision detection algorithms, in our case it sends the spheres which are likely to collide to the Spheres collision detection accelerator on the FPGA.
- Sphere collision detection: The spheres are processed on the FPGA and the collision results are sent back to the CPU, it is important to point out that this is a fine grain step, the processing time is low when compared to the amount of data needed to compute it, the whole collision detection algorithm is considered coarse grain as it generates a big amount of intermediate data when compared to the inputs, the number of collision tests could go up to 2^N where N is the amount of spheres on the virtual world.
- Virtual world state update: This step is processed on the CPU and it is responsible for fetching all the collision results and, through physics computations based on the collision vector resultants, it updates the virtual objects positions for the next simulation step.

Algorithm 1 starts by calculating the collision depth and storing it in *d*, based on the result three different possible cases occur: 1) *Fake collision*: Collision does not happen; 2) *Grazing collision*: bodies barely contact each other; 3) *Real collision*: Bodies collide with each other and the collision has a depth. Note that grazing collisions almost never happen and we focus our attention on the real collisions.

In our design, we have re-implemented the method for collision detection between spheres - (*Real Collision*). The inputs are the position of two spheres in a space and their respective radius. The output is a contact point. The spheres

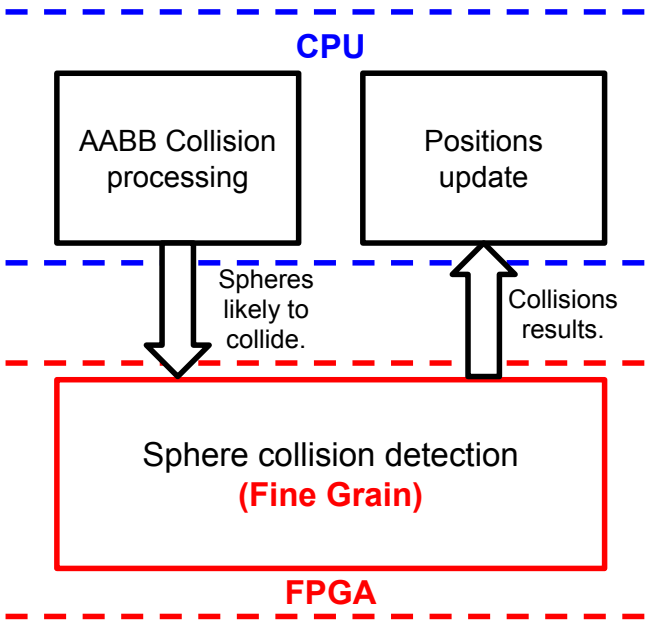


Fig. 1. Collision detection algorithm steps.

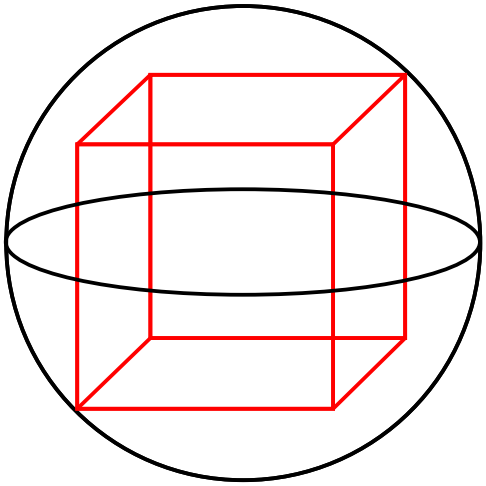


Fig. 2. Sphere AABB.

representation and the resulting contact points use, respectively, 4 and 7 32-bit IEEE 754 floating points numbers. We identify parallelizable calculations, dividing these calculations into 5 stages as seen in Algorithm 2, where each stage is executed sequentially but for multiple possible collisions in parallel. We call the parallelized version of this the Accelerator Function Unit (AFU), which we implement on the FPGA. For the real collision execution flow, we execute 26 floating point operations per collision.

B. Accelerator Datapath

The AFU is implemented as the co-processor on the FPGA to calculate collision detections as one piece of the entire system. Figure 3 shows the larger system in which the ODE software is executed on the CPU, and data is transferred

Algorithm 1 Spheres collision detection algorithm.

$d \leftarrow$ dCalcPointsDistance3(p1,p2)
Case 1: Fake Collision
if $d >$ (r1 + $r2$) **then return** 0
end if
Case 2: Grazing Collision
if $d \leq 0$ **then**
  $cPos \leftarrow p1$
  $cNormal \leftarrow (1, 0, 0)$
  $cDepth \leftarrow r1 + r2$
  Case 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

Algorithm 2 Parallel FPGA collision detection algorithm.

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 :
$cPos \leftarrow p1 + cnk$

in real-time to the AFU on the FPGA for the collision calculations via shared-memory. Note that the CPU can access the systems main memory, which the FPGA cannot, and the shared memory is accessible via a QPI bus.

The AFU is divided into two main components as seen in Figure 3. First, a Processing Units Controller (PUC) is responsible for handling communication of data between the CPU and the FPGA via the shared memory as well as synchronizing when the local FPGA processing units can perform their fine-grained calculations. Second, the Sphere Collision Processing Units (SCPUs) can access the loaded data to perform their respective collision detection calculations.

A handshaking control protocol is implemented on the CPU and PUC as follows:

1) The simulation step starts with the CPU sending all the virtual world information to a Source (Src) buffer,

where a collector fetches all the spheres and collision information that composes the virtual world from this buffer and stores it in local FPGA memory, which we call the Virtual World info RAM block (VWIRB). This is done at each simulation step. The collision information stored on the VWIRB is used to tell each of the SCPUs which pair of spheres participates in a collision. The collision information is stored in 512 bit lines with 32 sphere addresses each, each pair of addresses tells a SCPU which spheres on the local RAM block are used on the collision it is supposed to process.

2) The sphere's information on the VWIRB is replicated to local RAM blocks which are connected as inputs to each of the SCPUs, this replication is done once every simulation step in parallel.
3) Each SCPUs processes its respective collision and indicates when it has completed the collision detection.
4) The PUC collects all the collision information to the SCPUs and when it is done processing all of them, it sets a "done" signal, that the CPU reads, and then, fetches the results from the destination (Dst) buffer.
5) The CPU can then update the virtual world state, take new inputs, and start the next simulation step.

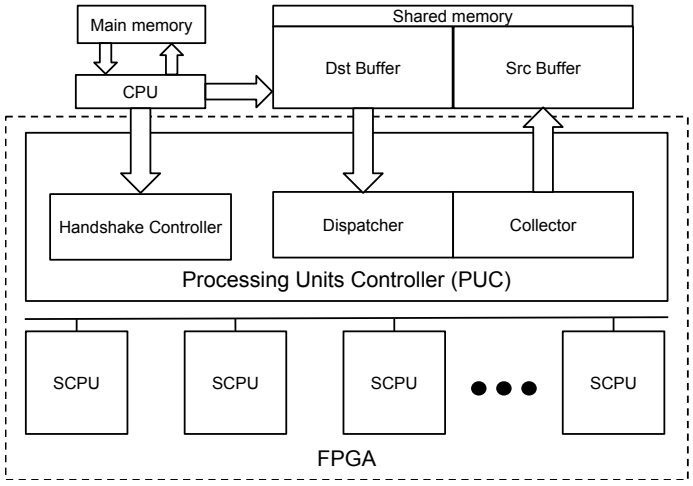


Fig. 3. Collision detection accelerator system.

C. Intel Heterogeneous CPU-FPGA Platform

We have implemented the collision detection accelerator using the first version of the Intel CPU-FPGA Platform [12]. The computer used for communication with the FPGA consists of Xeon Processors E5-2680 v2, its specifications can be seen in Table I. The CPU is connected to an Altera Stratix V model 5SGXEA7N1F45C1 FPGA by a 6.4 GT/s Intel QPI bus, the FPGA specifications are included in Table II. We have used the Accelerator Abstraction Layer (AAL) framework to develop our accelerator. AAL allows C/C++ implementations to manage transactions between the CPU and the FPGA hardware accelerator.

TABLE I
XEON E5-2680 V2 SPECIFICATIONS

Clock	Cache	RAM	SSD
2.8 GHz	25 mb	96 gb	120 gb

TABLE II
ALTERA STRATIX V MODEL 5SGXEA7N1F45C1 SPECIFICATIONS

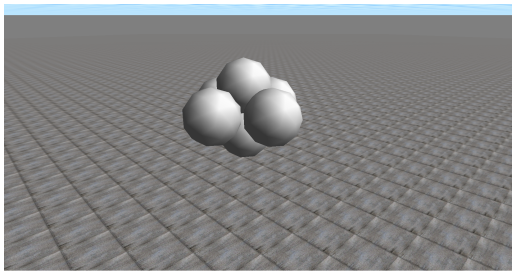
Clock	DSP Blocks	Logic Elements	Memory bits
200 MHz	256	234,720	52,428,000

The system CPU side is composed of the ODE software and the AAL application used to communicate with the AFU. The FPGA side includes the AFU to perform the collision detection calculations which is implemented via the System Protocol Layer 2 (SPL2) provided by Intel, which is responsible for memory address translation since the FPGA uses virtual addressing.

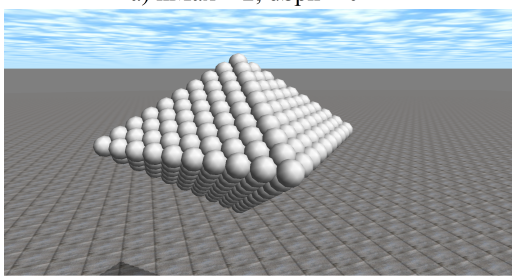
IV. BENCHMARKS AND MEASUREMENT METHODOLOGY

The AAL framework is service-oriented working with the concept of transactions. A transaction must be initiated, processed and finished. Our speedup results compare the cost of this transaction with the FPGA as compared to the same transaction executed only on a single CPU with the specifications described in the previous section.

The application processes each set of collision on the AFU 10 times, computing the average execution time. As the path for executing a specific type of collision is always the same, we expect that the execution time for a certain amount of collisions is always similar and the growth in execution time is linear and proportional to the number of collisions.



a) hMax = 2, dSph = 0



b) hMax = 10, dSph = 0

Fig. 4. Benchmarks examples.

In order to generate results, we use a set of parameterizable benchmarks with spheres organized in the form of a diamond. We define our benchmarks based on the distance between spheres ($dSph$) and the maximum height ($hMax$) for the diamond in terms of number of spheres. Figure 4 presents two of these examples. After we create the simulation environment a force is applied to each sphere in the direction to the center of the diamond so that they will collide with each other.

V. Results and Discussion

Our experiments were conducted by executing sets of real collisions for both ODE and the AFU. We generated 10 benchmark sets varying $hMax$ from 1 to 10 and keeping $dSph$ equal to 0. For the AFU, we used 16 SCPUs. Table III shows the FPGA usage to implement both the PUC and 16 SCPUs in terms of logic elements, DSPs, and memory bits. The design uses almost all the logic fabric and DSPs available on the FPGA.

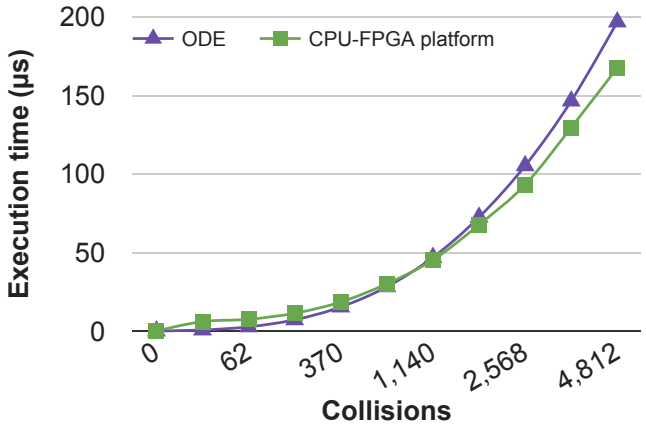


Fig. 5. Execution time for ODE and Intel CPU-FPGA platform.

TABLE III
Design FPGA resources consumption

DSP Blocks	Logic Elements	Memory bits
224 (88%)	200,844 (86%)	13,600,474 (26%)

Figure 5 presents the execution time for the number of collisions generated by each benchmark set. The Intel CPU-FPGA platform is faster than the single CPU when the collision numbers are greater than 1,000. We reached the highest speedup of 14.81% when the number of collisions is 4,812 ($hMax$=10). The average execution time for one collision on the Intel CPU-FPGA platform platform is 36 ns while for the ODE running on the CPU is 41 ns. It is important to note that the Intel CPU-FPGA platform execution time includes a 5 μs overhead due to configuration time. This speedup may not seem significant, but our heterogeneous implementation is only performing the finest-grained calculations of the CDP in this case, and therefore, the cost of transferring all of this data versus the amount of computation needed is very expensive. However, this does prove that in a real CPU-FPGA system that fine-grained CDP implementation can provide a co-processing speedup, and courser grained implementations of the CDP should shift the data to computation ratio even further towards benefiting these systems further even in situations with less collisions. Similarly, as the communication latency and bandwidth between FPGA, CPU, and shared-memory improves as their dies become more integrated, we should see additional speedups. Finally, if the FPGA had direct access to the main memory, we would expect better results.

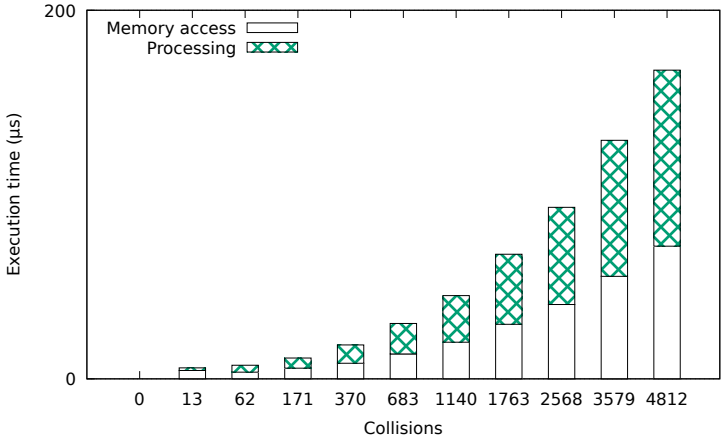


Fig. 6. Memory access and processing times for the Intel CPU-FPGA platform.

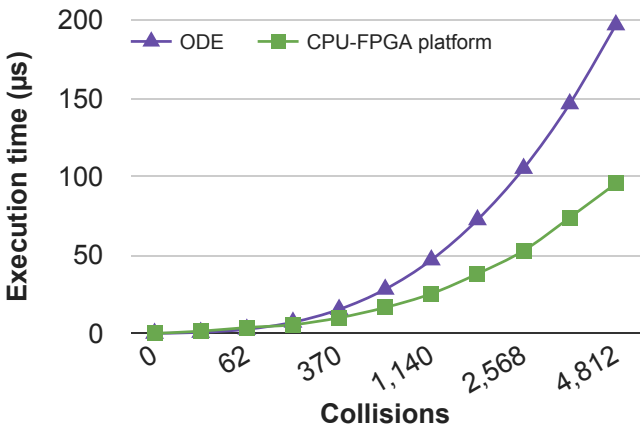


Fig. 7. Execution time for ODE and Intel CPU-FPGA platform without memory access time.

In order to perform a more in-depth study of the Intel CPU-FPGA platform execution time components, we have instrumented the design with counters. Figure 6 presents the memory access and processing times for each benchmark set. The memory access time is responsible for approximately 48%

of the total execution time while the collision processing is responsible for 52%. For this Intel CPU-FPGA architecture, the communication overhead is still a significant computation cost. It is common in FPGA accelerator papers to consider that data is pre-loaded in memory and not take in account the memory access time. In this case, our speedup would be 43%, for 4,812 collisions. Figure 7 shows the execution time comparison when the memory access time is considered instantaneous.

The value of these results and work is a deeper understanding of these heterogeneous systems (which include FPGAs) and the importance of the computation-to-communication ratio. For the finest grained portion of the CDP, the communication costs are just, barely, smaller than the parallel computation benefit, hence the small speedup results. However, as the granularity of computation is increased for what the FPGA co-processor does in terms of the CDP, the computation increases for less or equivalent communication. The question remains what is the area/resource cost for this additional computation?

One could imagine a scenario where the FPGA maintains data on the state of the world and only communications are made between CPU and FPGA on a "miss" scenario or (a relevant change in the virtual-world) where the communication is only needed in a cache-like model. In these heterogeneous systems that include FPGAs, we, the FPGA designers, will need to leverage lessons learned from the parallel research community on data distribution and optimization techniques that have been explored for the last 60 years to maximize the computation-to-communication ratio while optimizing computation in terms of resource usage on the FPGA.

VI. Conclusion and Future Work

In this work, we presented a hardware accelerator implementation of the most fine-grained part of a CDP. We implemented a case of study with sphere collision detection reaching a speedup of 14.81% with FPGA proven design running on a real system that includes shared-memory transfers. The results show that our design is faster when compared to a high-end CPU and the FPGA is a realistic accelerator. Our design is available released as open source. An important general idea drawn from this work is that even small computational kernels can get speed improvements as the FPGA and CPU are more closely integrated, but the quality these results are highly dependent on the communication-to-computation ratio.

Our results do not approach some of the previous researchers' results, but we highlight the importance of this work as it demonstrates a CDP implementation at the finest-grain can be speed up, and this implementation is on a real heterogeneous system where memory access and synchronization must be considered. Even under these restrictions, our implementation still improves results, and this suggests that further pursuits in implementing the courser stages of the CDP will provide even more benefit. Additionally, this works helps provide system designers an understanding of what trade-offs they can expect when implementing algorithms on these types of heterogeneous systems.

As for future work, we intend to expand our AFU implementation to other types of collisions detection algorithms and more course grain portions of the CDP in order to produce a more efficient accelerated engine. We will also execute the design in a newer Intel CPU-FPGA platform with increased integration of FPGA and CPU. We expect with the increased improvements of these platforms that our implementation will only become better. Finally, we plan on looking at the energy consumption of our design as this is another important metric to consider in the system. At present, our speedup suggests that the FPGA will consume more energy than the processor, but as speedup increases we would expect energy to be equal or even less when collisions are executed on the co-processing FPGA.

Acknowledgments

We would like to thank UFV and the research agencies FAPEMIG, CAPES, CNPq for their financial support. We would also like to acknowledge Connor Blandford and Oakley Katterheinrich for participating in early development at Miami University.

References

[1] A. M. Caulfield et al., "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[2] Y. k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[3] 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.

[4] N. Carriero and D. Gelernter, "A computational model of everything," *Commun. ACM*, vol. 44, no. 11, pp. 77–81, Nov. 2001. [Online]. Available: http://doi.acm.org/10.1145/384150.384165

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

[6] R. Weller, *New geometric data structures for collision detection and haptics*. Springer Science & Business Media, 2013.

[7] 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.

[8] 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.–.

[9] 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.

[10] 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.

[11] Z. Zhang, Y. Xin, B. Liu, W. X. Y. Li, K.-H. Lee, C.-F. Ng, D. Stoyanov, R. C. C. Cheung, and K.-W. Kwok, "Fpga-based high-performance collision detection: An enabling technique for image-guided robotic surgery," *Frontiers in Robotics and AI*, vol. 3, p. 51, 2016. [Online]. Available: http://journal.frontiersin.org/article/10.3389/frobt.2016.00051

[12] P. K. Gupta. (2015) Intel xeon+fpga platform for the data center. [Online]. Available: https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf