Lessons Learned on which Applications Benefit when Implemented on CPU-FPGA Heterogeneous System

Fredy Alves
Science and Technology Institute, Universidade Federal de Viçosa
Florestal, Minas Gerais, Brazil
fredy.maciel@ufv.br

Peter Jamieson
Department of Electrical and Computer Engineering, Miami University
Oxford, OHIO, USA
jamiespa@miamioh.edu

Lucas Bragança
Science and Technology Institute, Universidade Federal de Viçosa
Florestal, Minas Gerais, Brazil
lucas.braganca@ufv.br

Ricardo Ferreira
Departament of Informatics, Universidade Federal de Viçosa
Viçosa, Minas Gerais, Brazil
ricardo@ufv.br

José Augusto M. Nacif
Science and Technology Institute, Universidade Federal de Viçosa
Florestal, Minas Gerais, Brazil
jnacif@ufv.br

ABSTRACT

In this work, we provide "lessons learned" from implementing two applications, collision detection and Boolean Gene Regulatory Networks (GRNs) simulation, on a CPU-FPGA heterogeneous platform. Both of these applications have, previously, been implemented and accelerated on FPGA-only devices, but when implemented on a more complete host and co-processor system the additional system factors, such as input and output data communication, impact the results. Using our two applications, we illustrate a set of lessons that need to be considered when porting applications to these emerging heterogeneous systems.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; **Reconfigurable logic applications**;

KEYWORDS

Hardware,Collision Detection,Accelarator,Xeon+FPGA,Verilog

SAMOS XVIII, July 15–19, 2018, Pythagorion, Samos Island, Greece

ACM ISBN 978-1-4503-6494-2/18/07…$15.00
https://doi.org/10.1145/3229631.3229648

ACM Reference Format:
Fredy Alves, Peter Jamieson, Lucas Bragança, Ricardo Ferreira, and José Augusto M. Nacif. 2018. Lessons Learned on which Applications Benefit when Implemented on CPU-FPGA Heterogeneous System. In *SAMOS XVIII: 2018 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, July 15–19, 2018, Pythagorion, Samos Island, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3229631.3229648

1 INTRODUCTION

More and more heterogeneous computing systems are emerging including single chips that include a heterogeneous set of cores (including CPUs, GPUs, and FPGAs) to larger systems composed of multiple heterogeneous nodes. In these larger systems, the computation chips are connected to numerous types of communication technologies and topologies, and memories are integrated in these systems in a variety of ways. A number of questions emerge with these systems, including the ones we focus on - what algorithms are accelerated on these new systems, and how should they be designed for the new architecture?

In particular, this work focuses on looking at implementing two algorithms on Intel's Heterogeneous CPU-FPGA Platform - collision detection and Boolean Gene Regulatory Networks (GRN) simulation. These two applications have been, previously, implemented on FPGAs with demonstrated speedups, and therefore, many would assume that a heterogeneous system should have similar benefits when implementing these algorithms. This is not always the case since system-level limitations such as CPU to FPGA communication introduce additional factors in the overall success of implementing these applications efficiently. The question is, which factors should be considered when implementing applications on these emerging architectures?

We look at how the two implementations of these algorithms on the CPU-FPGA platform have varying success. Boolean GRN simulation can be implemented with similar speedups to FPGA-only implementations (which is GOOD), and collision detection does not get significant speedup results compared to FPGA-only implementations (which is BAD). We use these two cases to provide a "lessons learned" about these types of CPU-FPGA systems, which is very useful as we expect more and more systems to include both of these cores. There is an expectation that soon we will see both FPGAs and CPUs on the same die.

Our main lesson is a reminder that the co-computation, performed on the FPGA, needs to be significant in computation cost as compared to the FPGA communication of both input and output data. What's interesting is this communication process (data to co-processor) is constantly changing, meaning that depending on how the system connects the memories, CPUs, and co-processors has significant impact on the application. Understanding this architecture and its capabilities is a necessity in porting applications to these devices.

2 BACKGROUND - CPU-FPGA PLATFORM

The recent acquisition of Altera by Intel is an indicator that reconfigurable architectures such as FPGAs are becoming interesting computation chips in mainstream computing. As a result of this new reality, Intel has developed heterogeneous CPU-FPGA platforms for researching purposes. Some recent papers that used this platform are [1, 3].

This specific heterogenous CPU-FPGA platform consists of an Intel Xeon E5-2680v2@2.80GHz processor and an Altera Stratix V@200MHz FPGA. These are connected through a QPI Bus capable of a bandwidth of 8GT/s. In order to make the most out of the bandwidth, two sockets are used on the motherboard, where one is used by the Xeon and the other by the Stratix V. All the communication between the CPU and the FPGA is done through a 64KB cache. A new version for this platform is already available for researches and it can reach up to 28GB/s.

This drastic improvement in communication bandwidth allows the use of FPGAs for accelerating algorithms with a potential for parallelism, even when these are considered fine grain applications. CPU-GPU heterogeneous platforms are the main option for parallelizing and accelerating applications, but it only allows parallelism through SIMD where all of its cores perform the same operation at the same time. FPGAs are fully configurable logic devices, which allows them to better address problems such as execution and data divergence, and FPGAs are a low power device compared to GPUs for many applications.

3 APPLICATIONS

In this section, we briefly introduce both algorithms used in our case study - collision detection and Boolean GRN simulation.

3.1 Collision Detection

3.1.1 ***The algorithm:*** Whenever a simulation is composed of many bodies interacting with each other through forces, one important algorithm is collision detection. These algorithms detect when virtual objects collide, and they calculate the result of these collisions.

The collision detection algorithm is composed of 3 main steps. The first is the **Bounding volume collision detection (BVCD)**: during this phase, bounding volumes which were attributed to the bodies on the simulation are tested against each other. In the case of the algorithm used in Alves *et al.* [1], the bounding volume is a box, called the AABB (Axis Aligned Bounding Box). The second step executes **Specialized collision detection methods (SCDM)**. This step uses the two bodies of two specific types and computes if they collide or not by returning the normal vector and position for the collision. In our implementation, we [1] accelerate this step on an FPGA. The last step is the **Virtual world state update (VWSU)**. VWSU uses the collision results and physics calculations to update the position for each virtual object.

3.1.2 ***The implementation:*** Our [1] implementation of SCDM is for collision detection between spheres. The input is the 3D coordinates for two spheres and their respective radii, and the output is a structure named **contact point**, which is composed of the normal vector for the collision and its position on the space. Another output is one bit which specifies the collision type. The algorithm has been translated to an FPGA accelerator where the high level software implementation is from an open source engine called Open Dynamics Environment [12]. In that software framework, it is possible to re-implement all of its SCDM directly on user code without the need to recompile the whole engine.

The algorithm starts by computing the collision depth d, and based on the result, it classifies the collision as one of three types. A **Fake Collision** happens when d is greater than the sum of the spheres radii and means they are not touching each other. A **Grazing Collision** exists when $d \leq$ and the spheres are barely touching. The focus of this work is on **Real Collisions**, which occur when $d > 0$ and the spheres are hitting each other with a depth d.

We divide the sequential algorithm into 5 stages, executing all the operations in each stage in parallel, although we execute each stage sequentially. The parallelized version of the algorithm is written in Verilog, implemented on the FPGA, and is called the Accelerator Function Unit (AFU).

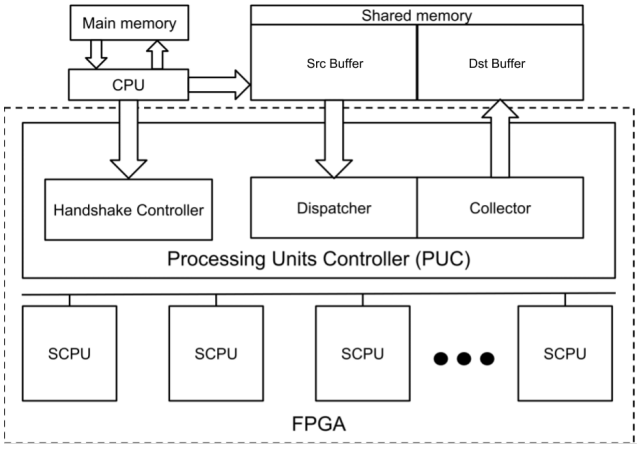

Figure 1: Collision detection accelerator system.

The AFU works as a co-processor on the FPGA that receives the spheres which are likely to collide from the BVCD step and sends the results back to the CPU for the VWSU step. Figure 1 shows the complete system. The ODE application is executed on the CPU while the AFU executes on the FPGA, and data is transferred between these two through a shared memory accessed via a QPI bus.

As seen on Figure 1 the AFU is divided into two parts. The Spheres collision processing unit (SCPU) implements the parallelized sphere collision detection algorithm, and it is replicated many times in order to be able to process more than one collision in parallel. The Processing Units Controller (PUC) is responsible to dispatch the data fetched from the source buffer on the shared memory to the SCPUs, and then collects the results in order to send them to the destiny buffer where the CPU can use the data for the VWSU step.

We have implemented this system on the Intel CPU-FPGA research platform [7] as described in the background. The CPU is a Xeon Processor E5-2680v2 which is connected to an Altera Stratix V model 5SGXEA7N1F45C1 through a QPI bus. For more details on both the algorithm and its implementation as an AFU, refer to [1].

3.2 Gene Regulatory Networks

3.2.1 ***The algorithm:*** The DNA in a cell composes its genome where a short region of DNA is called, a gene. Genes go through the a process called "gene expression" where they act as a template for producing protein molecules which the cell uses to adapt to its environment. There is a type of protein which acts as a molecular regulator known as Transcription Factors (TFs), and these TFs modulate the frequency with which genes are expressed. TFs can also regulate the expression of genes that code for other TFs. These group of connections between TFs and genes are known as Gene Regulatory Networks (GRNs).

A GRN can be modeled as graph where:

- Each node vi represents a gene.
- The state for each gene is a logic value that indicates if the gene is active or not.
- Each node is connected to ki nodes, where $ki > 0$
- At each timestep t each node state is updated according to a Boolean function Fi.

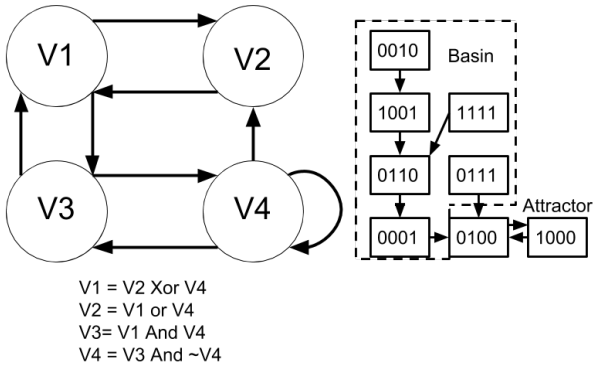

Figure 2: GRN example.

The new value of a node vi is $vi(t+1) = fi(vi_1(t), ...vi_k(t))$ where k is the number of adjacent nodes of vi. In Fig. 2 the functions are inside the nodes. A network state (Si) is a vector of all the node states on it, and in the example seen on Fig. 2 the initial state $Si = v1v2v3v4 = 0010$. After a certain number of timesteps the network converges to a set of stable steady states. An example can be seen on the network on Fig. 2 where the network performs the transactions 0010 $\rightarrow$1001 $\rightarrow$0110 $\rightarrow$0001 $\rightarrow$0100 $\rightarrow$1000 and then it keeps updating from 0100 to 1000 and back to 0100 in a loop. This set of stable steady states is called an attractor.

Since the amount of possible states for a network is exponential, the solution space is too large to optimize. If the state update is synchronous for all nodes, then it is possible to partition the space into fully connected disjointed graphs, and these are composed each by an attractor and the group of states that converge into it, called its basin. Silva *et al.* [3] implement a framework that computes the basin histograms for the dynamics of the network, the length of all attractors, and the average number of steps to reach an attractor from a random initial state.

3.2.2 ***The implementation:*** Our design consists of an interface unit, a thread control unit, and a set of processing elements (PEs). Each PE is composed by a control unit and

two functional units. The functional unit is the Boolean function implementation for each node in the network, and it is implemented as Boolean expressions with a one-bit register to compute and store the node state. Since the state computations for each node are independent, they can all be computed at the same time in one clock cycle.

A network state Si will converge to an attractor after Ti timesteps where Ti is the size of the transient associated to Si. A PE is composed by two copies of the GRN with all its nodes and connections. If we consider a state Sa as the first inside a cyclic attractor, the PE will return to Sa after L steps where L is the size of the attractor. The strategy used in this work is to use two copies of a network $N1$ and $N2$, that is, one PE. Every time $N1$ performs one simulation step, $N2$ performs 2, an attractor is identified when $Sn1 == Sn2$. This approach is $O(1)$ in memory usage, and this is important because the solution space is exponential meaning it is too large to fit in memory. The PE is composed by two FUs ($FUp1$ and $FUp2$), $FUp1$ performs one clock cycle while $FUp2$ performs two. The PE computes the attractor length by keeping $P2$ stopped, while $P1$ performs L clock cycles until $P1$ is equal to $P2$ again.

Each PE receives a state for its FUs as an input which they use to compute the attractor and the transient length in a few cycles. The design proposed on this work can manage light and irregular thread loads by using asynchronous FIFOs and PEs. The thread control unit feeds the FIFO buffers with a sequence of states to be computed. The PE gets a state from the input FIFO and computes the transient and the attractor length and writes them to an output FIFO.

4 APPLICATION IMPLEMENTATION COMPARISONS

In this section, we present implemenation details of the Collision Detection and Gene Regulatory Network applications.

4.1 Collision Detection Implementation Comparison

Table 1 shows collision detection implementations across a set of platforms. We report the platform, CPU, and FPGA/GPU in columns 2, 3, and 4, and include communication method, parts of the collision detection pipeline, data format, data width, platform, and finally speedup in the remaining columns. Our implementation is at the top and is in bold.

The key ideas we want to show is first that FPGA implementations of collision detection tend to show their results on data sets that are either pre-loaded or fit entirely on the device. This is not the reality if collision detection FPGA implementations are to exist in real applications. Second, one criticism of our work is that why would we put collision detection onto an FPGA. Even though the algorithm can be implemented on many platforms, in the key two instances collision detection is used (simulation and video games) these other computation devices are usually in use and the algorithm is interleaved with other work. For example in a video game, the GPU is used for its main purpose, graphics, and the CPU is used in all sorts of other aspects of maintaining the virtual world. This means that the FPGA could be used for offloading collision detection.

There are many ways to implement collision detection on FPGAs, and our model described in 3.1.1 uses a strategy based on the structure of the bodies using common physics and geometry computations. Wu *et al.* [15] treat the same problem with a mathematical approach where the interior-point algorithm is used, and this is a typical algorithm used for optimization and it needs a mathematical model, which can be adapted for multiple problems. But even when the strategy changes, the idea of the three collision detection steps is kept. Their works use of pre-loaded data, which allows them to reach orders of magnitude of speedup, and they do not compare their results to other systems.

The data format and width representing the objects is also an important part of design performance. Specifically, this impacts memory usage efficiency, communication throughput, and the cost for arithmetic operations. Raabe *et al.* [11] use fixed point in order to implement the BVCD step, and the reason why most implementations use this data format for implementing hardware accelerators is so an arithmetic operation can fit inside a DSP block on an FPGA, which is usually 18 bit wide for each input. In our work, the data is single precision floating point, and the DSPs are 18 bit, and therefore, more than one DSP is used to create a 32 bit DSP, which causes an overhead due to the need for programmable routing, which impacts the maximum clocking frequency of our design. Raabe *et al.* [11] presents a speedup of 4x, but it does not specify how data is transferred to the FPGA, and if this transfer time was taken into account.

When more parts of the collision pipeline are implemented on the FPGA, the speedup is better since the ratio of data to processing changes in favor of computation. Zhang *et al.* [5] is a great example of this since both the BVCD and the SCDM are calculated on the FPGA, and a speedup of 8x is achieved. This work does not specify the communication method between host and FPGA, or if the time for data transfer is taken into account. Our work only implemented the SCDM and the ratio of data to processing is higher.

Both works ([4, 8]) that use GPUs are applied to robotics. It does not always make sense to use a GPU for collision detection since the GPU could be busy processing the graphics. However, in robotics the GPU can be used to optimize robot movement through a physical space modeled as a collision detection problem. Their implementations do not compare to CPUs, and A. Hermann *et al.* [4] provide no details about

Table 1: Collision Detection Implementations on Various Platforms.

Research Group	Platform	CPU	FPGA/GPU	Comm method	Steps	Data format	Data size	Speedup CPU
Hybrid Collision [1]	**CPU-FPGA**	**Xeon Processors E5-2680 v2 2.8 GHz**	**Altera Stratix V 5SGXEA7N1F45C1**	**Shared Memory**	**SCDM**	**Floating Point**	**32-bits**	**0.14x**
Linear Solution [15]	Altera DE2 board	-	Cyclone II EP2 2C35F672C6	Pre-loaded RAMs	SCDM	Floating Point	-	-
Fixed point [11]	Alpha Data ADM-XRC-II board	Pentium III 1 Ghz	Xilinx Virtex II (XC 2V6000, speed grade -4)	-	BVCD	Fixed point	11 - 44-bits	4x
Robot FPGA [5]	-	Intel Core TM i7 CPU 860 2.8 GHz	Xilinx Virtex-6 XC6VHX565T	-	BVCD SCDM	Floating Point	32-bits	8x
Mobile GPU [8]	CPU-GPU	Core i7, 3.4 GHz	NVIDIA Titan GTX 6 GB GDDR5 RAM	Shared Memory	All	-	64-bits	-
Robot GPU [4]	-	-	NVIDIA GeForce GTX TITAN	Shared Memory	All	-	-	-

Table 2: GRN Implementations on Various Platforms.

Research Group	Platform	CPU	FPGA/ GPU	Comm method	Model	Speedup CPU	Net Type	Attractor
Reconfig 2017 FPGA [3]	**CPU-FPGA**	**Xeon E5-2680v2 2.8 GHz**	**Stratix V**	**Shared memory**	**Sync**	**349.4**	**Real**	**yes**
FPL 2017 FPGA [10]	CPU-FPGA	IBM POWER8	Xilinx Kintex UltraScale KU060	Shared memory	Sync/ Async	11.7	Real	no
FPL 2010 FPGA [6]	CPU-FPGA	Intel Core 2 duo (2.8GHz)	Virtex-6	Serial	Sync	1300	Synth	yes
FPL 2005 FPGA [13]	FPGA	Pentium 4 2.4GHz	Xilinx Virtex II	Pre-loaded	Sync	76	Synth	no
MWSCAS 2004 FPGA [17]	ARM-FPGA	Pentium I.3GHz	Virtex 2000	Shared memory	Sync	1285	Synth	yes
PLOSONE 2014 GPU [14]	CPU-GPU	Intel Core2 Quad Q9400 2.66	NVIDIA GeForce GTX 680	Shared memory	Sync	453	Synth	yes

the CPUs in their system. Also, max performance is not necessarily the main concern in robotics. For example, a robot responsible for moving boxes in a factory does not need to process at a rate of 200 Km/h if the maximum mechanical speed of the system is 30 Km/h. We provide the GPU results for completeness.

4.2 Gene Regulatory Network Implementation Comparison

Table 2 shows the Boolean GRN across a set of platforms structured similar to Table 1 above.

GRNs are synchronous when all genes are updated at the same time, and asynchronous when only a subset of these nodes are updated at the same time. Just a few of the works for GRN acceleration on FPGAs implement the asynchronous model. One example in this direction is [10] where a

framework for generating both synchronous and asynchronous designs from Boolean network models was created, but as we can see on Table 2 it has the smallest speedup. Their designs are compared to an implementation on a high performance server, and the more generic the design is the less speedup you get since it is supposed to cover several possible setups, and this makes it hard to optimize the design for all instances. Additionally, synchronous designs broadcast their clock and reset signals to all genes, while asynchronous designs have to create networks in order to be able to control each gene separately. These large networks tend to create many switches, which are one of the main causes for creating a slow down (because of slacks and pipeline bubbles) in their design.

As expected and seen on Table 2, all the other works other than [10] present significant speedups. Since these are all synchronous designs, this is expected. Among these works [6, 13, 17], Tagkopoulos *et al.* [13] has the lowest speedup because their system uses the Jtag communication to pre-load the data to the FPGA. The designs, which do not compute the attractor for the GRN, have the lowest speedups, and this happens because the reuse of a GRN initial state is lower for these designs. In order to find an attractor, a design can execute many iterations for a single initial state, which makes the ratio data to processing lower than a design which, for example, only checks genes of interest.

The speedups for Ferreira *et al.* [6] and Zerarka *et al.* [17] are almost the same, but Zerarka *et al.* [17] embed an ARM processor to communicate with the FPGA, which is a higher level of integration than a general purpose CPU, and this decreases the time to transfer data. The communication on [6] is through a serial port, which is a slow communication channel and is not accounted for when computing the speedup. Our implementation, Silva *et al.* [3], takes into account all the data transfer time for the speedup calculation, and it uses real literature based networks as benchmarks. We can see that, even when the transfer time is fully considered, the speedup is still close to a CPU-GPU system as in [14]. Additionally, our implementation is more efficient in power consumption, because FPGAs, typically, have a power consumption as great as 10 times lower than GPUs.

5 LESSONS LEARNED ABOUT HETEROGENEOUS ARCHITECTURES

In this section, we use the comparison breakdown to provide our "lessons learned" for the CPU-FPGA platform based on our GOOD - Boolean GRN algorithm and BAD - collision detection results. Note that our designs are the bold entries in tables 1 and 2.

5.1 Lesson 1 - It's all about ratios

In the CPU-FPGA coupling systems the communication of data is fundamental. The reason that the FPGA is becoming closer and closer to an on die core is the same reason that memories and communication of data is improved as the distance between them physically decreases. The data tends to be the limiting factor in high performance computation.

In our case the ratio of time spent communicating data as compared to the computation on that data is a major factor in our results. Our collision detection only performs a portion of the collision detection pipeline, which results in a small speedup. In Boolean GRN, however, the computation is significant compared to the data communication.

The key is understanding your application in terms of a ratio of this data communication to computation cost as it applies to the CPU-FPGA platform.

5.2 Lesson 2 - Simplification matters

If you can simplify the FPGA computation in terms of data representation as compared to software implementations, then there is potential for benefit. Mainly, the idea is to use approximations of the data that still work with the problem.

For example, in collision detection the collision detection pipeline is used as a filtering system to determine which objects need to be compared with one another. Early in this pipeline objects tend not to be colliding, and therefore, rough calculations can be used to determine if this is the case. The question is can the implementation take advantage of this.

The Boolean GRN application does take advantage of this by using bitwise calculations (very efficient on an FPGA as that's the fundamental building block of the device) to determine gene expression.

5.3 Lesson 3 - Don't Forget Amdahl and Remember Design Time

Amdahl's law [2] is very useful for a quick way to estimate speedup before spending significant time to speedup the application. On the CPU-FPGA system you can use both Amdahl's law and an estimate of the ratio of communication to computation to have a quick estimate of speedup.

This estimate, however, is a best case estimation where you are assuming that you can parallelize a certain aspect of your application very efficiently. Xilinx's technical report on creating FPGA designs [16] has a diagram on page 7 that reminds us that there is a significant design time cost to optimizing acceleration on any platform (especially FPGAs). This should be highlighted in research papers like ours to make non-experts aware of the time spent to get any speedup on reconfigurable fabrics, which is hard to design for.

5.4 Lesson 4 - How well does application map to the acceleration architecture?

One aspect to consider is how well does the application map to the computing architecture. In our examples, Boolean GRN maps very well to an FPGA which smallest components are designed as Lookup Tables (LUTs) that implement Boolean functions. Collision detection, on the other hand, does not map as easily to the architecture since the main focus is comparing Cartesian points and a number of DSPs are needed for various calculations in this process. This doesn't mean that if the algorithmic version of a problem doesn't easily map that it shouldn't be converted, but it may take significant time to figure out how to do the mapping efficiently.

In this case, parallel design patterns [9] may allow an understanding of how to map to particular architectures. Unfortunately, nobody has done this work yet for reconfigurable fabrics.

6 CONCLUSION AND FUTURE WORK

In this work, we presented two applications mapped to a CPU-FPGA platform where one application (Collision Detection) got very little speedup, and the other application (Boolean GRN) got significant speedup. These two applications allowed us to look at some "Lessons Learned" with respect to porting applications to this type of system. The key lesson is to understand the communication to computation ratio with respect to the parallelizable parts of the algorithm. In these host to co-processor systems, just like the data to CPU problem, can be the key limiting factor on how much benefit we will gain by porting the application.

In the future, we are interested in looking at applications on CPU-FPGA systems, since we expect the two to be more tightly coupled until the day when the FPGA is just one of the cores on a die. Understanding how the communication technology impacts these systems will be key to determining which applications to map to these. Additionally, we are interested in improving our applications by finding ways to improve the computation. For example, in collision detection this is possible by implementing more of the pipeline on the FPGA.

7 ACKNOWLEDGEMENTS

The authors thank CAPES, FAPEMIG and CNPQ for the financial support. We also thank Intel Altera and Synopsys for the software licenses and the hardware used during this work.

REFERENCES

[1] F. A. M. Alves, P. Jamieson, L. B. da Silva, R. S. Ferreira, and J. A. M. Nacif. 2017. Designing a collision detection accelerator on a heterogeneous CPU-FPGA platform. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6. DOI: http://dx.doi.org/10.1109/RECONFIG.2017.8279786

[2] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 483–485.

[3] L. B. da Silva, D. Almeida, J. A. M. Nacif, I. Sánchez-Osorio, C. A. Hernández-Martínez, and R. Ferreira. 2017. Exploring the dynamics of large-scale gene regulatory networks using hardware acceleration on a heterogeneous CPU-FPGA platform. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–7. DOI: http://dx.doi.org/10.1109/RECONFIG.2017.8279791

[4] A. Hermann et al. 2015. Anticipate your surroundings: Predictive collision detection between dynamic obstacles and planned robot trajectories on the GPU. In *2015 European Conference on Mobile Robots (ECMR)*. 1–8.

[5] Zhang et al. 2016. FPGA-Based High-Performance Collision Detection: An Enabling Technique for Image-Guided Robotic Surgery. *Frontiers in Robotics and AI* 3 (2016), 51. http://journal.frontiersin.org/article/10.3389/frobt.2016.00051

[6] R. Ferreira and J. C. G. Vendramini. 2010. FPGA-accelerated Attractor Computation of Scale Free Gene Regulatory Networks. In *2010 International Conference on Field Programmable Logic and Applications*. 550–555. DOI: http://dx.doi.org/10.1109/FPL.2010.108

[7] P K Gupta. 2015. Intel Xeon+FPGA Platform for the Data Center. (2015). https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[8] A. Hermann, F. Drews, J. Bauer, S. Klemm, A. Roennau, and R. Dillmann. 2014. Unified GPU voxel collision detection for mobile manipulation planning. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 4154–4160. DOI: http://dx.doi.org/10.1109/IROS.2014.6943148

[9] Timothy G Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for parallel programming*. Pearson Education.

[10] M. Purandare, R. Polig, and C. Hagleitner. 2017. Accelerated analysis of Boolean gene regulatory networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–6. DOI: http://dx.doi.org/10.23919/FPL.2017.8056778

[11] A. Raabe, S. Hochgurtel, J. Anlauf, and G. Zachmann. 2006. Space-efficient FPGA-accelerated collision detection for virtual prototyping. In *Proceedings of the Design Automation Test in Europe Conference*, Vol. 2. 6 pp.–. DOI: http://dx.doi.org/10.1109/DATE.2006.243875

[12] Russell Smith. Open Dynamics Engine. (????). http://www.ode.org/

[13] Ilias Tagkopoulos, Charles Zukowski, German Cavelier, and Dimitris Anastassiou. 2003. A Custom FPGA for the Simulation of Gene Regulatory Networks. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI (GLSVLSI '03)*. ACM, New York, NY, USA, 132–135. DOI: http://dx.doi.org/10.1145/764808.764843

[14] Hung-Cuong Trinh, Duc-Hau Le, and Yung-Keun Kwon. 2014. PANET: A GPU-Based Tool for Fast Parallel Analysis of Robustness Dynamics and Feed-Forward/Feedback Loop Structures in Large-Scale Biological Networks. *PLOS ONE* 9, 7 (07 2014), 1–9. DOI: http://dx.doi.org/10.1371/journal.pone.0103010

[15] C. H. Wu, S. O. Memik, and S. Mehrotra. 2009. FPGA Implementation of the Interior-Point Algorithm with Applications to Collision Detection. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. 295–298. DOI: http://dx.doi.org/10.1109/FCCM.2009.38

[16] Xilinx. 2013. *Introduction to FPGA Design with Vivado High-Level Synthesis*. Technical Report. Xilinx Inc.

[17] M. T. Zerarka, J. P. David, and E. M. Aboulhamid. 2004. High speed emulation of gene regulatory networks using FPGAs. In *Circuits and Systems, 2004. MWSCAS '04. The 2004 47th Midwest Symposium on*, Vol. 1. I–545–8 vol.1. DOI: http://dx.doi.org/10.1109/MWSCAS.2004.1354048