

FREDY AUGUSTO MACIEL ALVES

**DOMAIN-SPECIFIC AND GENERAL-PURPOSE
ACCELERATION IN RECONFIGURABLE AND
VECTOR PROCESSOR PLATFORMS**

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

Orientador: José Augusto Miranda Nacif

VIÇOSA - MINAS GERAIS

2019

**Ficha catalográfica elaborada pela Biblioteca Central da Universidade
Federal de Viçosa - Campus Viçosa**

T

A474d
2019
Alves, Fredy Augusto Maciel, 1993-
Domain-specific and general-purpose acceleration in
reconfigurable and vector processor platforms / Fredy Augusto
Maciel Alves. – Viçosa, MG, 2019.
39 f. : il. (algumas color.) ; 29 cm.

Texto em inglês.

Orientador: José Augusto Miranda Nacif.

Dissertação (mestrado) - Universidade Federal de Viçosa.

Inclui bibliografia.

1. Processamento paralelo (Computadores).
2. Processadores de matriz. 3. Arranjos de lógica programável
em campo. I. Universidade Federal de Viçosa. Departamento de
Informática. Programa de Pós-Graduação em Ciência da
Computação. II. Título.

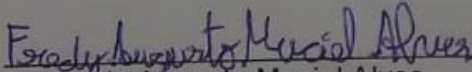
CDD 22. ed. 004.35

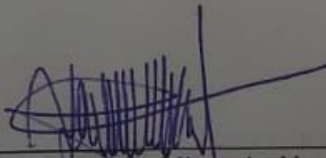
FREDY AUGUSTO MACIEL ALVES

**DOMAIN-SPECIFIC AND GENERAL-PURPOSE ACCELERATION
IN RECONFIGURABLE AND VECTOR PROCESSOR PLATFORMS**

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 07 de outubro de 2019.


Fredy Augusto Maciel Alves
(Autor)


José Augusto Miranda Nacif
(Orientador)

*Este texto é dedicado à minha família
que sempre esteve ao meu lado e aos meus amigos.*

AGRADECIMENTO

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

RESUMO

ALVES, Fredy Augusto Maciel, M.Sc., Universidade Federal de Viçosa, outubro de 2019, **Aceleração de uso geral e de domínios específicos em plataformas reconfiguráveis e processadores vetoriais**. Orientador: José Augusto Miranda Nacif.

Muitas plataformas heterogêneas CPU-FPGA emergiram nos últimos anos variando de grandes sistemas a nós compostos por um único chip. Neste trabalho nós apresentamos três artigos cuja implementação foi realizada neste tipo de plataforma heterogênea. O primeiro explica a implementação de um acelerador de detecção de colisões, o segundo compara esta implementação à um acelerador de redes booleanas de regulação de genes e à outras aplicações do estado da arte, lições são derivadas sobre o que deve ser levado em consideração antes de começar uma implementação para um sistema heterogêneo. O terceiro artigo compara duas ISAs Vetoriais diferentes, Vector Register (VR) e Vector Memory (VM). Este trabalho mostra as vantagens de se utilizar VM ao invés de VR. O foco do primeiro e segundo artigos é na aceleração de aplicações específicas para uma plataforma específica, o foco do terceiro artigo é comparar duas ISAs vetoriais em uma mesma plataforma para ambas.

Palavras-chave: FPGA. Processador Vetorial. Acelerador em hardware.

ABSTRACT

ALVES, Fredy Augusto Maciel, M.Sc., Universidade Federal de Viçosa, October, 2019, **Domain-Specific and General-Purpose Acceleration in Reconfigurable and Vector Processor Platforms**. Orientador: José Augusto Miranda Nacif.

Many heterogeneous CPU-FPGA platforms have emerged in the past few years ranging from large systems to single chip nodes. In this work we present three different papers implemented on this type of heterogeneous platform. The first one explains the implementation of a collision detector accelerator, the second one compares this implementation to a boolean gene regulatory network accelerator and other applications, it derives lessons learned about what to take into consideration before implementing for heterogeneous systems. The third paper compares two different vector processor ISAs, vector register (VR) and vector memory (VM). It shows the advantages of using VM over VR. The focus of the first and second work is on the acceleration of specific applications for a specific platform, the focus of the third one is to compare two different vector ISAs on the same platform for both.

Keywords: FPGA. Vector Processor. Hardware Accelerator.

SUMÁRIO

INTRODUCTION	8
PAPER 1: DESIGNING A COLLISION DETECTION ACCELERATOR ON A HETEROGENEOUS CPU-FPGA PLATFORM	11
PAPER 2: LESSONS LEARNED ON WHICH APPLICATIONS BENEFIT WHEN IMPLEMENTED ON CPU-FPGA HETEROGENEOUS SYSTEM	18
PAPER 3: MEMORY-BASED VECTOR INSTRUCTION SET ARCHITECTU- RES	26
CONCLUSION	38
REFERÊNCIAS	39

INTRODUCTION

A wide variety of heterogeneous platforms are emerging in the past few years, those can range from single chips with different types of cores such as CPUs, GPUs and FPGAs to large systems composed of many heterogeneous nodes. On large systems, processors are connected to communication and memory systems in many different ways with different topologies and interconnection fabrics (ALVES et al., 2018). Industry vendors such as Microsoft, Amazon, Intel and Xilinx have been working on heterogeneous CPU-FPGA platforms. Intel is working on integrating both CPU and FPGA on the same die, the FPGA on the heterogeneous CPU-FPGA Intel HARPv2 platform is capable of reaching a peak bandwidth of 25 GB/s while accessing the main memory. The FPGA is getting closer to an on-die core for the same reason that memories and data communication improves as the physical distance between them decreases. (ALVES et al., 2018)

Early vector processors such as the Cray-1 used vector register files due to their fast access speed. While a 16x4 flip flop, used to build the Cray-1 vector registers, had a 6ns cycle time, main memory chips with higher capacities (1Kb) had a much slower 50ns cycle time. (ALVES et al., 2019) Modern technologies such as the latest FPGAs place billions of transistors on chip, which makes it faster and less costly to implement large on-chip memories (ALVES et al., 2019). The Intel Arria 10 FPGA can have up to 28620 M20K and 4164 MLAB memory blocks which is equivalent to 48Mb total of internal memory running up to 800 MHz which consists of a 1.25ns cycle time. It is important to evaluate if vector processors should keep the named register files or if memory-based operands are now more suitable to this new technological reality. MXP is a memory-based vector architecture which uses the device internal memory in order to store the vector data called the vector scratchpad, it can vary from Kb to tens of Mb in size. It is possible to have the maximum frequency impacted by the amount of interconnection network to connect memory blocks but as we show on this work, 65535 Bytes of internal memory is already enough to achieve a considerable speedup when comparing a memory-based to a register-based vector architecture.

In this work we study the impact of these technology shifts in reconfigurable platforms implementations and vector processor architectures. The main contributions derived from this work are:

1. Heterogeneous CPU-FPGA implementation of a sphere collision detection algorithm showing that FPGAs can suffice as a useful co-processor even for the finest-grain collision detection calculations.
2. A set of lessons derived from two applications that need to be taken into account

when developing domain-specific accelerators for emerging heterogeneous systems.

3. Demonstration of advantages of memory-based over register-based vector architectures through diagrams, code fragments and benchmarks performance results.

Collision detection algorithms are used to detect collisions between bodies in a virtual space and calculate the results for these collisions, they are used in many domains such as simulation, tolerance checking, and video games. The later was the main reason for the creation of GPUs and the performance growth of CPUs and memory, and this indicates that algorithms used in these domains such as collision detection should be investigated as viable applications on co-processors such as FPGAs as these become more common on modern systems. Our first work ([ALVES et al., 2017](#)) is about the implementation of an application specific accelerator for collision detection on a CPU-FPGA platform, It implements the smallest piece of the collision detection pipeline, and shows how collision detection can benefit from FPGA co-processing even for the finest grain part of its pipeline.

With the recent emerging of heterogeneous systems composed of CPUs, GPUs and FPGAs, it is important to investigate which applications are best accelerated on this type of system and what needs to be taken into account when porting them. In ([ALVES et al., 2018](#)) we present the implementation of accelerators for two types of applications (Collision Detection ([ALVES et al., 2017](#)) and Boolean Gene Regulatory Networks) on the same CPU-FPGA platform as in ([ALVES et al., 2017](#)), and we compare each implementation to other works which also implemented these same applications on heterogeneous systems. We then provide four lessons learned about what should be considered before starting an implementation for this type of platform.

Modern platforms such as the latest FPGAs provide on-chip memories which rival the speed and capacity of register files. Due to this new reality, on ([ALVES et al., 2019](#)) we compare two different vector processor architectures, one using a vector register file (VR) with named registers and one using an addressable on-chip vector memory (VM) in order to store their vector data. We show that the VM approach shows advantages over VR such as better use of the vector storage, easier programming, improved flexibility with vector sizes and data sizes, improved data reuse (locality), better prefetching, less data movement and higher performance ([ALVES et al., 2019](#)). Both VM and VR are emulated on MXP which is originally a Vector Memory processor, it's architecture can be seen in more details on Figure 1. The DMA Engine and the Scalar core can both access main memory. The scratchpad is a vector memory built with fast on-chip memory ranging from KB to tens of MB. If data is outside the scratchpad, the DMA Engine exchanges data between main memory and the scratchpad. It works similar to a GPU shared memory.

This Master's thesis is structured according to the format of a paper's collection standardised by the technical and graduate council of Universidade Federal de Viçosa.

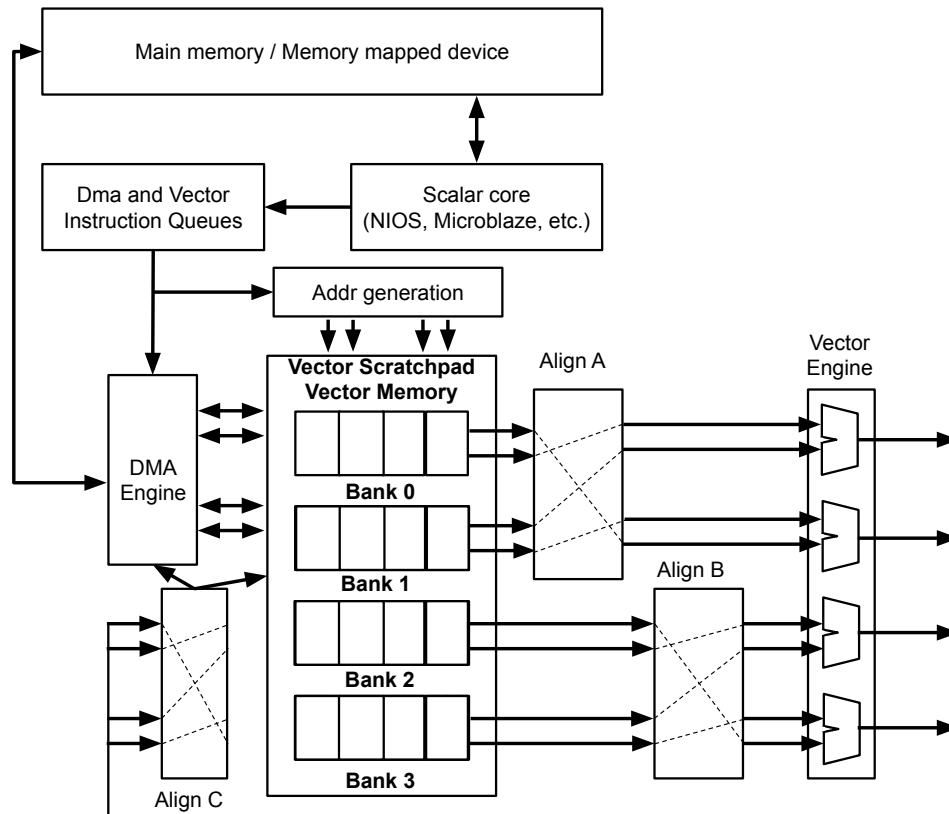


Figura 1: MXP Architecture

This format is composed by 5 chapters. Chapter 1 consists of a general introduction which correlates the themes addressed on the papers which compose this thesis. On Chapters 2, 3 and 4 we present the three papers produced during the Masters program and on Chapter 5 we present a general conclusion for the thesis.

**PAPER 1: DESIGNING A COLLISION DETECTION ACCELERATOR
ON A HETEROGENEOUS CPU-FPGA PLATFORM**

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

Fredy Augusto M. Alves¹, Peter Jamieson², Lucas B. da Silva¹, Ricardo S. Ferreira³, José Augusto M. Nacif¹

¹Science and Technology Institute, Universidade Federal de Viçosa, Florestal, Brazil

²Department of Electrical and Computer Engineering, Miami University, USA

³Departament 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 community¹ 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>

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

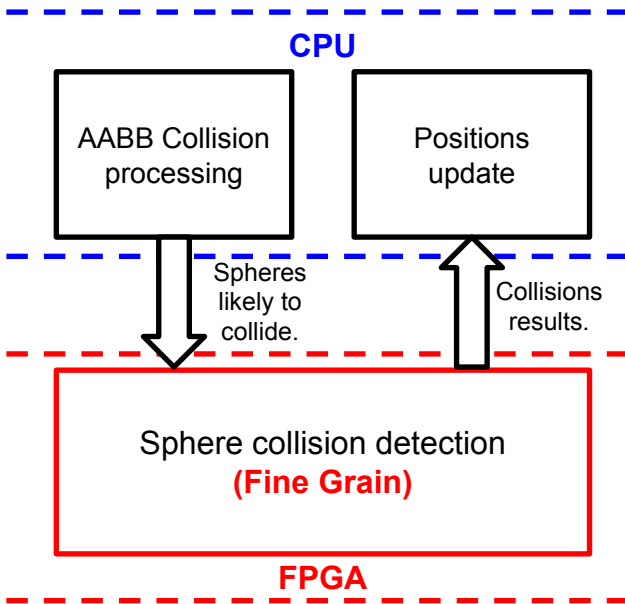


Fig. 1. Collision detection algorithm steps.

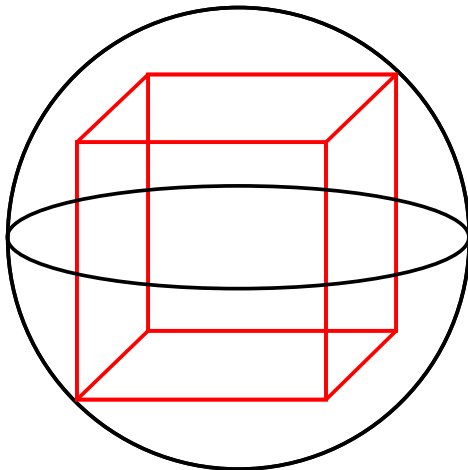


Fig. 2. Sphere AABB.

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.

$$d \leftarrow \text{DCALCPPOINTS_DISTANCE3}(p1, p2)$$

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

$$d1 \leftarrow \text{DRECIP}(d)$$

$$cNormal \leftarrow (p1 - p2) * d1$$

$$k \leftarrow 0.5 * (r2 - r1 - d)$$

$$cPos \leftarrow p1 + cNormal * k$$

$$cDepth \leftarrow r1 + r2 - d$$

end if

Algorithm 2 Parallel FPGA collision detection algorithm.

Stage 1 :

$$d \leftarrow \text{DCALCPPOINTS_DISTANCE3}(p1, p2)$$

$$rsum \leftarrow r1 + r2$$

$$psub \leftarrow p1 - p2$$

$$rsub \leftarrow r2 - r1$$

Stage 2 :

$$d1 \leftarrow \text{DRECIP}(d)$$

$$d > rsum$$

$$r2 - r1 - d$$

$$r1 + r2 - d$$

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.

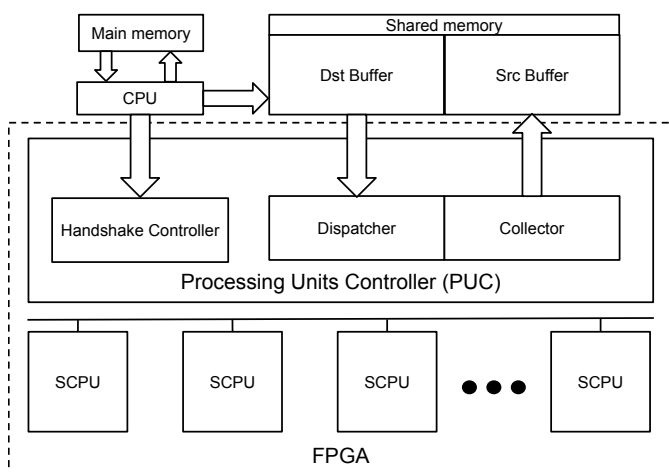


Fig. 3. Collision detection accelerator system.

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

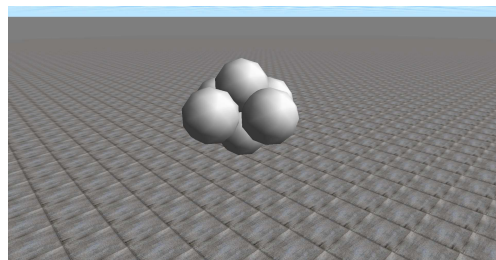
Clock	DSP Blocks	Logic Elements	Memory bits
200 MHz	256	234,720	52,428,000

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.

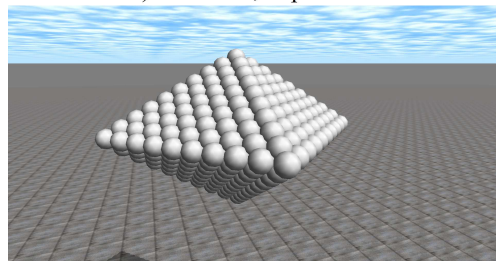
IV. BENCHMARKS AND MEASUREMENT METHODOLOGY

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.

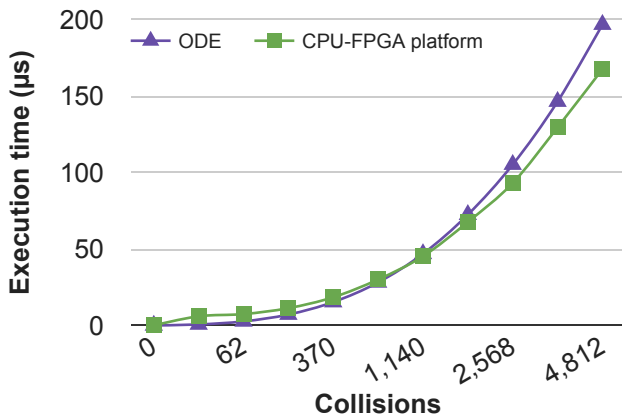


Fig. 5. Execution time for ODE and Intel CPU-FPGA platform.

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 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 coarser 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.

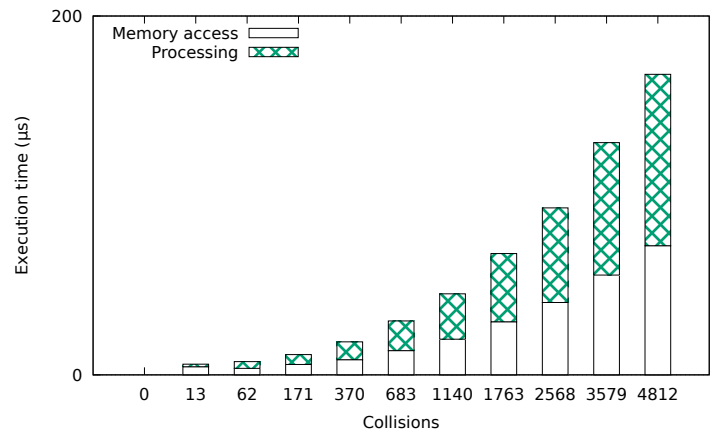


Fig. 6. Memory access and processing times for the Intel CPU-FPGA platform.

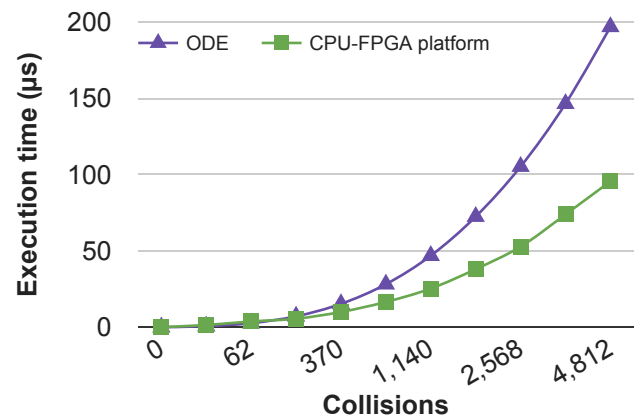


Fig. 7. Execution time for ODE and Intel CPU-FPGA platform without memory access time.

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 coarse 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 53rd 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>

**PAPER 2: LESSONS LEARNED ON WHICH APPLICATIONS
BENEFIT WHEN IMPLEMENTED ON CPU-FPGA
HETEROGENEOUS SYSTEM**

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.maciell@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, Accelerator, Xeon+FPGA, Verilog

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?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. *SAMOS XVIII, July 15–19, 2018, Pythagorion, Samos Island, Greece*
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

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

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 heterogeneous 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).

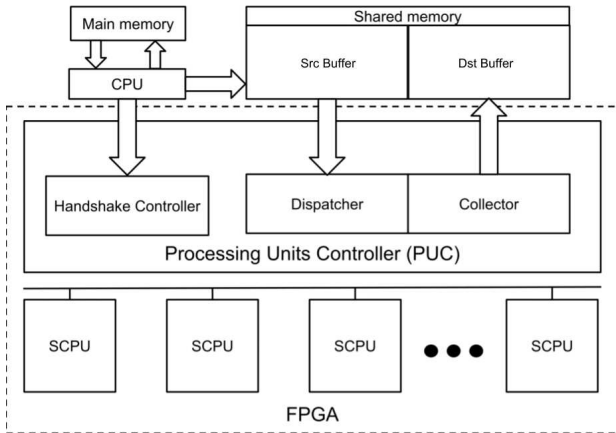


Figure 1: Collision detection accelerator system.

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 v_i 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 k_i nodes, where $k_i > 0$
- At each timestep t each node state is updated according to a Boolean function F_i .

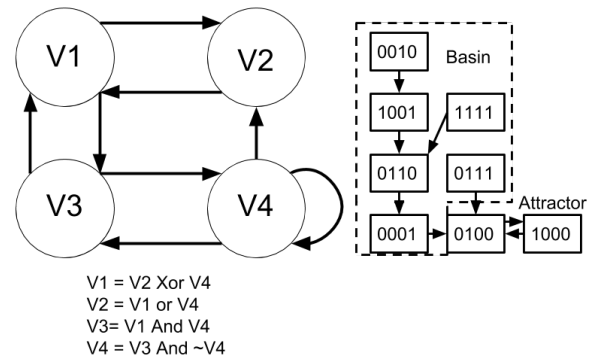


Figure 2: GRN example.

The new value of a node v_i is $v_i(t+1) = f_i(v_{i_1}(t), \dots, v_{i_k}(t))$ where k is the number of adjacent nodes of v_i . In Fig. 2 the functions are inside the nodes. A network state (S_i) is a vector of all the node states on it, and in the example seen on Fig. 2 the initial state $S_i = v_1v_2v_3v_4 = 0010$. 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 S_i will converge to an attractor after T_i timesteps where T_i is the size of the transient associated to S_i . A PE is composed by two copies of the GRN with all its nodes and connections. If we consider a state S_a as the first inside a cyclic attractor, the PE will return to S_a 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 $S_{n1} == S_{n2}$. 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 implementation 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/carlib/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>

PAPER 3: MEMORY-BASED VECTOR INSTRUCTION SET ARCHITECTURES

Memory-based Vector Instruction Set Architectures

Fredy Alves
Fed. Univ. of Viçosa, Brazil

José Nacif
Fed. Univ. of Viçosa, Brazil

Aaron Severance
VectorBlox

Guy Lemieux
The Univ. of British Columbia

Abstract—Modern computer systems demand both high performance and ease of programming. In this paper, we demonstrate that these can be achieved with a vector instruction set that operates on data stored in addressable on-chip memory rather than a register file. Known as a vector-memory (VM) architecture, this is in contrast to traditional vector-register (VR) architectures, an approach similar to RISC where data is first placed into named vector registers using vector-load and vector-store instructions. The main advantages of VM over VR are a reduction in unnecessary data movement, better utilization of the vector data store by allowing longer vectors or more vectors, elimination of vector data spill code when one vectorized function calls another, simplified compiler support, simplified prefetching to hide memory latency, and an extendable framework for plugging in custom accelerators. We demonstrate these advantages with diagrams, code fragments and performance results using exemplary VM and VR instruction sets implemented in a Xilinx Zynq-series FPGA.

I. INTRODUCTION

Early CISC instruction set architectures (ISAs) allowed the use of memory-based operands. However, this held back CPU performance as on-chip logic and registers exceeded the speed of external memory. In contrast, RISC ISAs use faster register-based operands for arithmetic instructions and access slower memory only through load/store instructions. As a result, RISC is sometimes called a load/store or register-based architecture. This decoupled the slow memory speed from pipeline processing and paved the way for caches, register renaming, and out-of-order execution to keep the arithmetic units as busy as possible.

Memory-based operands were also used in the earliest vector supercomputers [1], [2], but these were trumped by the Cray-1 [3], which had a RISC-like design that used register-based operands for data and decoupled load/store instructions. The Cray-1's eight named vector registers were built from discrete chips, each containing 16×4 flip-flops with a 6ns cycle time. In contrast, main memory chips had higher capacity (1Kb) and a much slower 50ns cycle time. Since the Cray-1, vector architectures have generally continued to be register-based. Like scalar processors, aggressive vector architectures need out-of-order execution and vector register renaming to keep the vector ALUs busy. However, due to large amounts of vector data, vector register renaming can be quite expensive.

The main purpose of this paper is to question whether vector architectures should continue to use a named register file as the primary storage mechanism for vector data? In our investigation, we have found many reasons why a memory-based approach might be considered superior. A VM approach provides advantages such as easier programming, improved

flexibility with vector sizes and data sizes, improved data reuse (locality), better prefetching, less data movement and higher performance. As will be shown, traditional register-based vectors also lack flexibility in how data is presented to the ALUs, so they suffer primarily from excessive data movement, which not only constrains performance but also leads to the use of costly hardware-centric design techniques like register renaming to help recover lost performance.

A. What's Different?

Obviously, scalar processors continue to use a register file for scalar data storage because of its ultra-small size and speed. However, a vector processor needs a much larger primary data store. For example, an aggressive scalar CPU design may have up to 256 physical 64-bit registers, resulting in a total of 2KB of data per CPU core. In contrast, the NEC SX Aurora has 128kB of vector data storage per CPU core [4]. Due to their large capacity, the vector data store should not be as heavily multi-ported as scalar register files. Instead, data ports can be made wider and multi-banked.¹ This is important because multi-porting leads to quadratic growth per bit of storage, whereas widening and multi-banking are closer to linear growth. Because of this difference, the vector data store should not be automatically built and used in the same way as the scalar register file.

Modern technology places billions of transistors on a chip. As a result, large on-chip memories are faster and far less costly to implement than they were in the Cray-1 era. They rival the speed and offer higher capacity than register files, and they fully integrated next to the arithmetic logic without any slow and power-hungry die-to-die communication. At some point, the speed differential that originally motivated the load/store architecture of the Cray-1 disappears. For example, the 16nm NEC SX-Aurora vector processor has eight cores running at 1.6GHz, with each core containing 32 vector lanes and 64 named vector registers holding 2048 bytes each (128kB per core) [4]. It can compute 768 double-precision MACs per cycle. In contrast, 14nm Intel Stratix 10 FPGAs have up to 11,721 embedded M20K memories (30MB total) running up to 1.0GHz [5]. While the FPGA has a lower clock speed, it offers more parallelism (5,760 single-precision MACs per cycle) and more on-chip memory. It is unclear how the NEC SX-Aurora implements its vector data store, but the large number of higher capacity, high speed of FPGA block RAMs suggest it is feasible to memory-based rather than register-file

¹It will be important to avoid bank conflicts to achieve performance.

Algorithm 1

VECTOR-REGISTER (VR) VERSUS VECTOR-MEMORY (VM) BASED ISA.

VR-based ISA:

```

vsetvl  x4
vlw     v1, 0(x1)
vlw     v2, 0(x2)
vadd    v3, v1, v2
vsw     v3, 0(x3)

```

VM-based ISA:

```

vdmardw x1, 0(x5), x4
vdmardw x2, 0(x6), x4
vsetvl  x4
vadd    x3, x1, x2
vdmawrw x3, 0(x7), x4

```

based. Despite this, the NEC SX-Aurora still uses a traditional named vector register file like the Cray-1. Because of these shifts in technology since the Cray-1, we believe it is important to re-evaluate the load/store architectural approach to vector processors in a modern context.

II. BACKGROUND

In this this paper, our goal is to compare a newer vector-memory (VM) architecture based upon addressable on-chip memories rather than a traditional vector-register (VR) approach. The key distinction behind these two approaches is illustrated in Algorithm 1. In this code, the `v1`, `v2` etc are vector register names that hold vector data, while `x1`, `x2`, etc are scalar register names that hold either scalar data or scalar pointers to vector data. Each of the VR and VM code fragments adds two vectors, reading data starting at addresses stored in scalar registers `x1` and `x2`, while writing data starting in `x3`. The *vector length*, or number of elements (words) to be added, is stored in `x4`. In a VR architecture, vector load-word instructions (`vlw`) must first transfer data from memory into the vector registers named `v1` and `v2`, the vector add writes the result to `v3`, and the vector store-word (`vsw`) transfers data back to memory. In a VM architecture, there is no need for load and store instructions, as the vector add is presented with scalar pointers directly as operands `x1`, `x2` and `x3`. However, there may be restrictions on which addresses may be used as scalar pointers. If the data originally resides at a restricted address, it must be moved to a valid scratchpad address using DMA instructions. It is possible to verify if data is already on the scratchpad by comparing its address to the scratchpad range of addresses. The `vadd` and `vdmawrw` are overlapped in time since the DMA Engine and the Vector Engine execute in parallel. When hardware detects a hazard, e.g. when a vector instruction attempts to read a value currently in the pipeline, pipeline bubbles are inserted until the values are written back to the scratchpad.

A. VM Variations

In the VM architecture model considered by this paper, we require all vector data to be stored in a specific address region, called the vector scratchpad, or scratchpad for short.

This region is finite in size because it must be located in fast on-chip memory, but it could be hundreds of KB to tens of MB in size. If data is located outside of this region, explicit vector DMA operations (like `vdmardw` for reading words, and `vdmawrw` for writing words) are required to exchange data between the scratchpad and main memory. In Algorithm 1, scratchpad addresses are held in `x1`, `x2` and `x3` while main memory addresses are `x5`, `x6` and `x7`.

An alternative VM architecture, not considered by this paper, uses an automatically managed cache instead of a scratchpad. In such a cached architecture, explicit vector DMA operations are not needed. Instead, vector data is always stored in main memory, but a cache retains a copy for future data re-use. For example, such an architecture is used by LAcore [6]. While it is a valid concept, we do not consider it in this paper because it requires highly effective caching policies; vector operations tend to be applied to large data structures, making it difficult to optimize for locality. In our experience, we have found the DMA-based scratchpad approach naturally encourages the programmer to think about data-reuse by making memory accesses explicit, thereby achieving higher performance with lower external memory bandwidth. In future work, it may be possible to automate the insertion of DMA operations at the compiler level, or to provide automated caching policies at the microarchitectural level.

B. Fixed-width SIMD

Register-based SIMD instructions have been added to practically all active CPU instruction sets. For example, the IA-32 instruction set has been augmented with several generations of register-based SIMD instructions including MMX, SSE, and AVX, with several sub-generations within each. Each (sub)generation adds more registers, or wider registers, to improve performance, making the previous generation obsolete. For this and other reasons, one article claims that SIMD instructions are “considered harmful”, suggesting that variable-length vector instructions are better [7]. All fixed-width SIMD instruction set designs are treated as a natural extension of the RISC load/store architecture, so they are all register based. Some offer access to wider data, but a key aspect to all of these approaches is the general expectation that these SIMD instructions execute in 1 clock cycle. This makes these extensions fundamentally different from true vector architectures, so they will not be considered any further.

C. Variable-length Vector

True vector architectures have variable-length vectors, where executing an instruction can take several, if not dozens or even thousands, of clock cycles. As a result, the vector execution stage needs to be decoupled from the instruction issuing pipeline. These architectures have a special *vector-length* register which holds the current number of data elements to operate on; this number is a constant value like 4 in the case of fixed-width SIMD instructions. Before any instruction executes, it must check the vector-length and run to completion. As a result, a vector engine can have any number

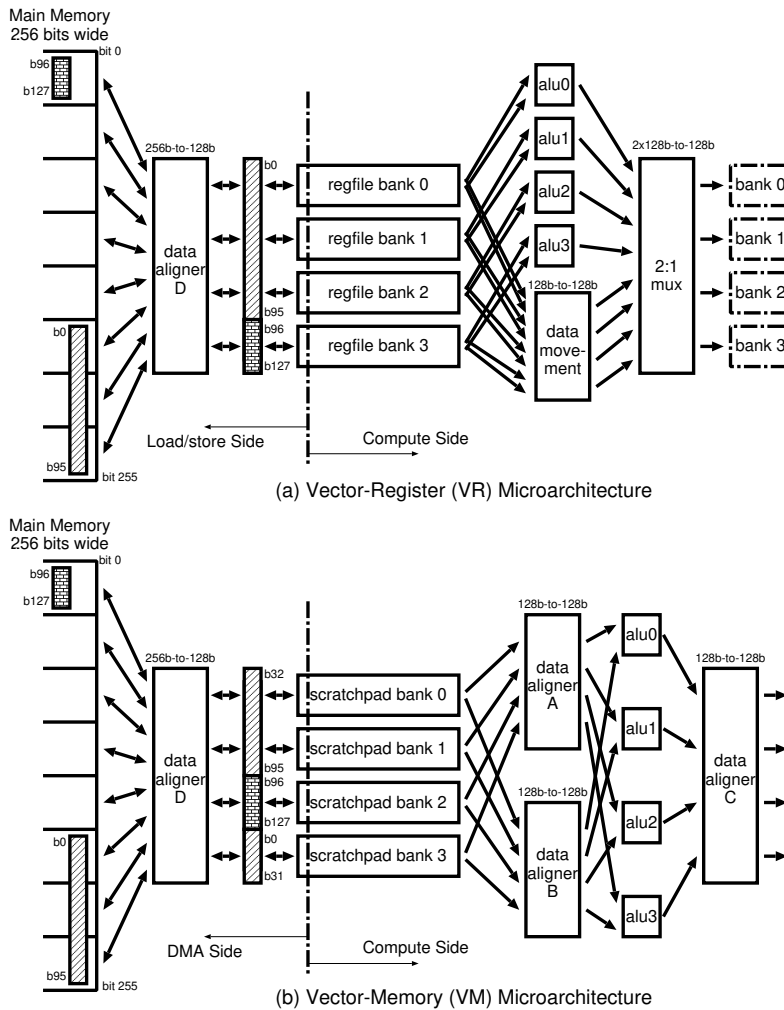


Fig. 1. Vector-register (VR) and Vector-memory (VM) microarchitectures.

of *vector lanes*, or parallel ALUs, which determines how many elements are executed in parallel every cycle. Portable software doesn't need to know the number of lanes, only the vector length. Since the only observable difference is execution time, this allows better portability of the vector instruction set to many differently sized vector engines, enabling forward portability or many available cost/performance points.

III. VM AND VR ARCHITECTURE DESCRIPTION

We know of only three modern memory-based vector architectures. One, VectorBlox MXP [8], is designed to accelerate image processing and computer vision tasks, and it is implemented on Intel and Xilinx FPGAs. Two, Cambricon [9], is designed to accelerate deep learning workloads, and a silicon prototype exists. The Cambricon architecture may have been used as a basis for Huawei's Kirin 970 processor [10]. Three, LAcore [6], is primarily designed to accelerate double-precision matrix arithmetic, but it only exists as within a software-based architectural simulator.

To facilitate a study between VM and VR, we have chosen to use the VectorBlox MXP² as a basis for VM, and the RISC-V Vector Extension³ as a basis for VR.

We say 'as a basis' because we have actually modified both MXP and RISC-V to make them as similar as possible, while retaining only the VM/VR aspect as a difference. For example, we have added an absolute-difference vector instruction to RISC-V, and min/max vector instructions to MXP. This removes performance differences that are simply due to inclusion of selected ALU operations, for example, and not due to having register-based or memory-based operands.

The key microarchitectural differences between VR and VM is shown in Figure 1. In these figures, we show how four 32-bit vector lanes might be organized with a 256-bit wide main

²LAcore and Cambricon are unsuitable; LAcore is cache-based and has no logic implementation, whereas few details are available about Cambricon.

³As it is under heavy revision, the RISC-V Vector Extension Specification draft 0.6 is used for this study. The current version is 0.7; we expect to use the latest version possible in the final version of the paper.

memory. These microarchitectures differ in how the vector store connects with memory, and also how the vector store connects with the ALUS. To simplify the diagrams, only the vector and memory datapaths are shown. Also, the banked outputs on the far-right in both figures represent the vector data writeback stage.

In the VR microarchitecture, vector registers are filled with instructions like `vlw` for loading a vector of words. The *vector-length* register determines how many words to read, and the `vlw` instruction provides the starting address which must be word aligned (bytes and halfwords will be discussed later). Since the loading address may not be aligned to the vector engine width or memory subsystem width, a data aligner D is needed to rearrange the words such that the starting address, which holds the first element, is always aligned with regfile bank 0. Vector data is stored in a banked register file, where elements 0, 1, 2, 3, and 4 are written to banks 0, 1, 2, 3 and 0, respectively. In other words, data elements are striped across the banks. The bank sizes for VM is defined by the amount of vector lanes, while on VR they are defined by the amount of vector registers, the scratchpad is divided equally between the banks. During computation, these elements are read out, starting at element 0, and provided to the parallel ALUs before being written back to the register file. Here, four elements per cycle would be read out for each ALU operand, until the entire vector length has been read out and the result written back. Even if source and destination vector registers are the same, the source is read one or more cycles before the destination is written, so no hazard exists. In a VR architecture, operations are done element-wise between vectors, and they retain the same ordinal position throughout. Since this does not suit all applications, the VR architecture has data movement instructions to rearrange elements within a vector. For example, `vslideup` will copy elements from position i to positions $i + N$, where N is the amount of the slide. To perform this, a special data movement block is needed in parallel with the ALUs. In the benchmarks written for this paper, we only used `vslideup` and `vslidedown`, but other operations like `gather` are possible.

Since the ALUs require two operands (eg, `add`) or three (eg, `multiply-add`), this register file can either be multi-ported or multi-banked. Although multi-ported is shown, it can also be multi-banked as in `Hwacha` [11]; in that case, at least 8 or 12 banks are needed to supply two or three operands, respectively.

Although only words are shown, these microarchitectures must be implemented down to the byte level. As a simplification, memory addresses can still be aligned to word boundaries, but anything more restrictive than that would be difficult to program, inefficient and not very portable. In a scalar CPU, when a byte is loaded into a register, it is zero-extended or sign-extended to the entire register width. This allows the ALU to do operations on the whole word, until the final result is written back with a `store-byte` instruction. In VR, we have support for vector `load-byte` and `load-half` (`vlb` and `vlh`), where the smaller data size is extended to fill the native word size; this is done by the data aligner D. For example,

if the native word size is 32-bits and we perform a `vlb`, the bytes would be extended to 32-bits and the operations would be in 32 bit words, not bytes. The ALUs for both VM and VR support 8 8-bit, 4 16-bit or 2 32-bit inputs. Since it is very wasteful to store and operate on bytes that are held in word-sized elements, VR also supports subword-SIMD. This means element sizes can be byte, half or word, and the vector-length register always specifies the number of elements. Operations specify the required element size, hence requiring at least 2 bits of encoding in the instruction. An important question is how to convert between element sizes without going through the memory subsystem with loads and stores. In VR and RISC-V, the default element size can be set, and most operations have a *widening* mode that generates data elements twice the default size. In addition, a set of *narrowing* instructions reduce data element sizes. The ALUs and the data movement block must all support bytes and half-words, as well as widening and narrowing operations. When widening or narrowing, there are obvious restrictions on the read/write bandwidth of the register file which must be maintained.

One of the drawbacks of the VR architecture is the complexity in ISA design that is created in order to handle all of the element sizes, widening, and narrowing. The RISC-V Vector Specification committee is currently wrestling with the best way to handle this issue – data element layout of the register file is presently exposed, and this makes non-portable code possible (perhaps even required, when attempting to solve certain problems).

In the VM microarchitecture, a scratchpad is filled or emptied with DMA instructions that read or write main memory. Main memory and scratchpad are *both byte addressable* and are demarcated by distinct address ranges. VM is scalable, the number of lanes is only limited by the platform resources. To further decouple memory from the compute side, the DMA length is specified in bytes and is independent of the current vector length. Vector elements are striped across the scratchpad banks just like the VR architecture. However, there are no alignment restrictions, so data at any main memory address can be copied to/from any scratchpad address. The maximum vector length is defined by the amount of free space on the scratchpad. Similarly, the source operands for any vector operation can start at any scratchpad address; before these can be presented to the ALUs, they must be re-aligned using data aligners A and B such that element i from both operands are presented to the same ALU. Likewise, the destination operand can start at any address, so data aligner C ensures it is written to the correct bank. Unlike the VR architecture, the memory side does not do zero/sign-extending of bytes or half-words into larger element sizes. Instead, this is done *on the fly* for each operand by data aligners A and B, and aligner C can write narrower results by truncating.

Custom compute accelerators can be attached to MXP by augmenting the original arithmetic ALUs with a parallel structure [12]. Up to 16 accelerators can be attached, each with its own custom engine width (up to the MXP width), and each with its own static pipeline latency. Memory-based

data in the scratchpad is streamed to each accelerator. A 2-input and 1-output structure is the simplest, but strided addressing modes enable other combinations. The scratchpad address generators in MXP support single-stream (1D vectors) as well as strided modes. The strided modes can iterate over a 2D or 3D dense matrix/volume or sub-matrix/sub-volume, or they can also perform sliding window readout (stride \neq data width) or repeated readout (stride=0) of data. While register-based vector architectures can stream a vector of data to an accelerator, they do not provide the same level of flexibility in data read-out.

IV. ADVANTAGES OF MEMORY-BASED OPERANDS

In this section, examples are given to illustrate the advantages of memory-based operands for vector computing.

A. Scratchpad Advantages

The VM architecture scratchpad with varying element sizes is much more efficient and flexible. First, it stores subword data at its natural size, and only extends to a larger size during computation. Second, it has no predefined boundaries, alignment restrictions, or vector length restrictions. Hence, any number of vectors can be created, and a vector can start at any address and be of any length up to the scratchpad capacity. Third, it naturally allows operations that support all combinations of mixed, narrowing, or widening element sizes. Fourth, it simplifies the data aligner D, since it does not need to do zero/sign extending. Fifth, a large swath of opcodes are eliminated, eg `vlb`, `vlh`, `vlw`, `vlbu`, `vlhu`, `vlwu` are all replaced with a simple vector DMA read (`vdward`). Also, strided versions of all of these loads can be replaced by a strided version of `vdward`. Sixth, element readout of vector data is no longer restricted to always start at element position 0; a sliding window can be achieved simply by incrementing the scratchpad pointer (i.e., `vslideup` or `vslidedn` instructions are redundant).

To gain some of this flexibility, the RISC-V Vector Specification goes to great lengths to allow its 32 vector registers to be regrouped into 16 or 8 or 4 larger vector registers.

Other advantages related to the scratchpad, such as simplified prefetching, will be described below.

B. Compiler, Portability, and Flexibility

The VM scratchpad holds an unlimited number of vectors, each with arbitrary and unlimited length, which are only limited by the total capacity of the scratchpad. Since vectors are merely a scalar quantity (the starting address), they can be manipulated in C like pointers to a buffer – they can be incremented, added to, or swapped with other pointers, without resulting in any actual vector data movement. This simplifies many operations that require sliding windows or sub-vectors to be extracted from larger vectors, and even allows vectors to overlap. This yields higher storage efficiency as well.

With VM, there are no vector register numbers to encode into an instruction. Instead, register numbers of the scalar host CPU are specified. Unlike vector registers, scalar registers

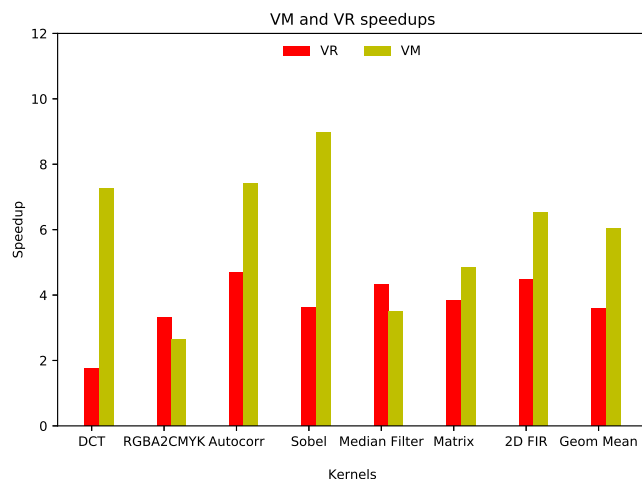


Fig. 2. Benchmark speedups for VR and VM.

DCT	RGBA2CMYK	Autocorr	Sobel	Median Filter	Matrix	2D FIR
1920x1088	800x800	Lags: 128 Data size: 1024	800x800	800x800	1600x1600	400x400 Taps: 4x4

TABLE I
DATA SIZES FOR KERNELS

can be very quickly loaded, and do not require extensive time or bandwidth to spill to main memory. Likewise, the compiler is simplified as no vector register allocation pass is needed. No compiler specialization makes the VM vector engine easier to port to multiple hosts. With a RISC-V host, for example, frequently used instructions can be 32b in size, while more advanced instructions use 64b. When used with an ARM core on the Zynq device, where the host ISA cannot be extended/modified, VM instruction issue uses memory-mapped stores to an instruction FIFO. One VM instruction copies three scalar register values and an opcode into the FIFO. Instruction issue overhead remains low because the ARM core runs over 6 times faster and VM instructions are usually multi-cycle.

C. Performance Comparison

To show the performance differences of VM versus VR, we have hand-coded several benchmarks for both architectures and ARM, and will run them on an FPGA-based system. The inner loops for these benchmarks are included as an appendix.

1) *Experimental Setup*: For experimentation, we use a Xilinx ZedBoard with 28nm Xilinx Zynq 7020 device and a hard Arm Cortex-A9. The VectorBlox MXP is implemented with 16 vector lanes which means 16 ALUs with their respective banks. This board has a single bank 32-bit DDR3-1600 DRAM with a peak bandwidth of 6400MB/s. All scalar C code is executed at 667MHz on the ARM A9; the MXP is fully decoupled and configured to run at 100MHz in logic within the FPGA. Vector instructions, which are almost always multi-cycle, are issued by the ARM writing opcodes and operands to

an FPGA-based vector instruction FIFO; a series of about 10 or so instructions, including four 32-bit store-word operations are needed to issue each vector instruction. The A9 will stall if the FIFO is full, but it will eventually proceed as the FIFO is drained. Before checking any vector results, the A9 must poll a busy bit on the MXP which indicates when the FIFO has been completely executed. The VectorBlox MXP accesses memory through an 800MB/s, 64-bit AXI link to the ARM processor complex. This is slower than the peak bandwidth of the memory, but it is sufficient to show acceleration because all of the benchmarks have high compute intensity. It can perform DMA accesses concurrently with vector compute operations, making it possible to hide memory latency through prefetching.

For benchmarking, we manually optimized all benchmarks separately for MXP and RISC-V Vector Extensions. Benchmarks were written using vector intrinsics for both platforms. This gives the performance of assembly language, but enables the full use of C for function calls, loops, conditionals, and so forth. Next, we narrowed the performance differences that were due to missing instructions by modifying both versions of the benchmarks to ensure only memory-based versus register-based differences remain, producing VM and VR versions.

To measure the runtime of VM, we remapped VM intrinsics into MXP intrinsics. This remapping is done by a simple header file with `#defines`. In doing this, we did not have to actually change the logic of MXP to add the new instructions needed by VM. Instead, we substituted VM instructions with something that would be timing-equivalent in MXP (but yields the wrong functional result). Since the run-time of all benchmarks is data-independent, this does not affect performance.

To measure the runtime of VR, we remapped VR intrinsics into RISC-V Vector intrinsics. However, we do not have a full logic implementation of the RISC-V Vector (RVV) specification. Instead, because of its generality, MXP can actually mimic the RISC-V Vector instruction set using a simple software translation layer as another simple header file with `#defines`. We designed two such translation layers – one to implement correct functionality of the RVV instructions (but at a penalty in run-time), and one to implement accurate timing of RVV instructions (to measure benchmark performance). Again, since run-time is data-independent, this does not affect our measurements. Note that the functionally correct translation layer was crucial for debugging and verifying that our benchmark code is correct, but it serves no purpose for timing measurements.

Note the ARM processor already has a 128-bit wide SIMD accelerator, named NEON, that runs at the full CPU clock speed of 667MHz. However, we do not use NEON in any of these experiments.⁴

⁴Note to reviewers: despite having peak operations-per-cycle and memory bandwidth advantage, NEON is slower than both VM and VR architectures discussed here. We don't have the space to add this comparison, and we don't find it conducive to the central thesis; it may be the subject of a future paper. This footnote will be deleted in the final version of the paper.

2) *Performance Results*: To compare performance, several microkernels shown in Table IV are run on the A9, setting the baseline. The speedups of VM and VR architectures are shown in Figure 2 and the data sizes used are shown in Table IV-B. VR is fastest for RGBA2CMYK and Median Filter, whereas VM is fastest for the other 5 benchmarks. For DCT, VM is almost 4× faster than VR. According to the geometric mean of all benchmarks, VM is about 1.5× faster than VR.

VM is faster than VR, despite similar peak execution capacity, because VR needs to do excessive data movement in autocorr, 2Dfir, dct, and sobel. In VR, autocorr and 2Dfir use `vslide` instructions to implement sliding windows, which takes time to actually move the data; in VM this is simply done by incrementing a scalar pointer. With sobel, VR uses 3 `vslide` instructions and must rotate 6 data buffers using several vector assignment operations to perform a data copy, whereas VR uses scalar assignments to slide or move the buffer pointers. In these cases, the memory-based architecture of VR contributes to less data movement and faster execution times for these three benchmarks.

D. Prefetching

Data prefetching is essential to hide memory latency. Algorithm 2 shows one way to implement software-based prefetching in VR: the loop body is unrolled twice, and the algorithm alternates between using two sets of registers. Algorithm 3 shows the same thing in VM, where scalar assignment operations are used to swap data buffers and no unrolling is needed. Unrolling results in code duplication and fringe code to handle leftover iterations, making VR code harder to maintain. Although out-of-order execution and register renaming can sometimes provide the same benefits as prefetching in VR, this presents many new microarchitectural challenges for implementation and likely requires additional register file storage that cannot be used by the programmer (as the number of physical vector registers would exceed the number of architectural registers).

When prefetching requires rotating more than 2 buffers, the VR code becomes even more complex and slower as it requires more data movement. With VM, one more scalar assignment is needed for each additional buffer.

A further advantage for VM is that the prefetch buffer sizes can be allocated exactly according to need, as long as they fit in the scratchpad. With VR, an advanced mode being discussed is the ability to merge 2, 4 or 8 registers into a single longer register, but there are restrictions on its use.

E. Function Composition

Function composition is one function calling another. Most applications and libraries are compiled separately and used as binaries. If a vector-accelerated function in an application calls a vector-accelerated function in a library, a collision results when the same named vector registers are used. To solve this, regular scalar functions spill registers on the stack. In VR, the compiler must spill entire vector data registers on the stack,

Algorithm 2
EXPLICIT PREFETCHING EXAMPLE FOR VR.

```

vr_vlwu( v1, pArray );
for (int i=W; i < H*W; i += W*2) {
  vr_vlwu( v2, pArray+i ); // even
  vr_vadd_vv( v3, v3, v1 );
  vr_vlwu( v1, pArray+i+W ); // odd
  vr_vadd_vv( v3, v3, v2 );
}
vr_vsw( v3, pRowOut );

```

Algorithm 3
EXPLICIT PREFETCHING EXAMPLE FOR VM.

```

vm_dma_read( v_a1, pArray, W*4 );
for (int i=W; i < H*W; i += W) {
  vm_dma_read( v_a2, pArray+i, W*4 );
  vm_vadd_vv32( v_accum, v_accum, v_a2 );
  t=v_a1; v_a1=v_a2; v_a2=t; // ptr swap
}
vm_dma_write( pRowOut, v_accum, W*4 );

```

causing significant data movement; library acceleration will be eroded by spill overhead.

In VM architectures, the scratchpad can be used like a stack to avoid data movement. Here, scratchpad vectors are allocated in a stack-like fashion, with the most recent vector placed at the top of the stack. At any point, typically the beginning of a function, a user starts a new stack frame using the function `vm_push()`. This saves the current high water mark of the scratchpad, creating a new local scope for further allocations. At the end of the function/scope, all local allocations are removed using `vm_pop()`. When done, all stack frames and allocations from the scratchpad can be removed by calling `vm_free()`.

One concern is how to handle a scratchpad out-of-space event. Typically, if we are calling other vector-accelerated functions, we restrict the code to use only half of the scratchpad. This can be applied recursively, but at some point spilling

Algorithm 4
FUNCTION COMPOSITION EXAMPLE FOR VM.

```

void B( v8_t *v_rr, v8_t *v_b1, v8_t *v_b2 )
{
  v8_t *v_tmp1=vm_vmalloc(N); // local alloc
  v8_t *v_tmp2=vm_vmalloc(N); // local alloc
  vm_vmul_vs8( v_tmp1, v_b1, 255 );
  vm_vmul_vs8( v_tmp2, v_b2, 166 );
  vm_vadd_vv8( v_rr, v_tmp1, v_tmp2 );
}
void A( i8_t *pX, i8_t *pY, i8_t *pZ, int N )
{
  v8_t *v_a1=vm_vmalloc(N);
  v8_t *v_a2=vm_vmalloc(N);
  v8_t* v_result=vbx_vmalloc(N);
  vm_dma_read( v_a1, pX, N );
  vm_dma_read( v_a2, pY, N );
  B( v_result, v_a1, v_a2 );
  vm_dma_write( pZ, v_result, N );
}

```

may be required. This occurs far less frequently than register-based spilling (we haven't encountered this situation).

This method allows functions to be composed as shown in Algorithm 4. This cannot be done with VR architectures without excessive data movement.

F. Context Switching

When multiple threads try to use the vector unit, contention may arise. Within a VR system, the entire vector register file must be saved/restored upon a context switch. Within a VM system, it is possible to partition the scratchpad into smaller regions, as long as applications are well-behaved and the number of regions is small/known. If desired, this can be enforced with the virtual memory system.

G. Drawbacks

With a memory-based architecture, hazard detection for in-order execution requires more than comparing vector register numbers, but instruction dispatch usually has multiple cycles before issue. Similarly, out-of-order execution is more complicated, but static scheduling usually works well enough.

V. CONCLUSION

Modern on-chip memories are almost as fast and offer more capacity than register files, diminishing the historic justification against memory-based architectures for vector architectures. In this paper, we show that a vector-memory instruction set architecture is faster due to flexibility that reduces data movement: subvector extraction, sliding windows, double-buffering and rotating prefetch buffers require only pointer manipulation. Improved storage efficiency results from custom vector lengths and overlapping vectors. Code duplication from loop unrolling is reduced. Spilling vector data to main memory is reduced under function composition and context switching. Compiler register allocation is not needed. All of this makes VM architectures very compelling.

REFERENCES

- [1] H. G. Cragon and W. J. Watson, "The TI Advanced Scientific Computer," *Computer*, vol. 22, no. 1, pp. 55–64, Jan. 1989.
- [2] C. J. Purcell, "The Control Data STAR-100: Performance measurements," in *National Computer Conference and Exposition*, 1974, pp. 385–387.
- [3] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.
- [4] K. Komatsu *et al.*, "Performance evaluation of a vector supercomputer SX-Aurora TSUBASA," in *Int'l Conf. for High Performance Computing, Networking, Storage, and Analysis*, 2018, pp. 54:1–54:12.
- [5] Intel, "Intel Stratix 10 embedded memory user guide," <https://www.intel.com/content/www/us/en/programmable/documentation/vgo1439451000304.html>, 12 2018, (Accessed on 01/14/2019).
- [6] S. Steffl and S. Reda, "LACore: A supercomputing-like linear algebra accelerator for SoC-based designs," in *IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 137–144.
- [7] D. Patterson and A. Waterman, "SIMD instructions considered harmful," ACM SIGARCH Computer Architecture Today, 09 2017.
- [8] A. Severance and G. Lemieux, "Embedded supercomputing in FPGAs with the vectorblox MXP matrix processor," in *CODES+ISSS*, Sep. 2013.
- [9] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 393–405.

TABLE II
MATRIX MULTIPLY AND MEDIAN FILTER CODES.

	VR	VM
Matrix	<pre> vr_vsetvl(N1); for (int i = 0; i < M1*N1; i+=N1) { vr_vlwu(v1, addr_m1); vr_vlwu(v2, addr_mout); vr_vlwu(v3, addr_m2); vr_vlwu(v4, addr_m2+N1); for (int j = 0; j < N1; j+=4) { vr_vlwu(v5, addr_m2+2*N1); vr_vlwu(v6, addr_m2+3*N1); vr_vmv_sv32(v7, v1, j); vr_vmadd_vvs32(v2, v3, v7); vr_vmv_sv32(v7, v1, j+1); vr_vmadd_vvs32(v2, v4, v7); addr_m2 = addr_m2+4*N1; vr_vlwu(v3, addr_m2); vr_vlwu(v4, addr_m2+N1); vr_vmv_sv32(v7, v1, j+2); vr_vmadd_vvs32(v2, v5, v7); vr_vmv_sv32(v7, v1, j+3); vr_vmadd_vvs32(v2, v6, v7); } vr_vsw_vv32(v2, addr_mout); // Addr updates addr_m2 = m_in2; addr_m1 = addr_m1+N1; addr_mout = addr_mout+N1; } </pre>	<pre> for (int i = 0; i < M1*N1; i+=N1) { vm_dma_read(out_v, addr_mout, N1*sizeof(uint32_t)); vm_dma_read(m2_v0, addr_m2, N1*sizeof(uint32_t)); for (int j = 0; j < N1; j++) { addr_m2 = addr_m2 +N1; vm_dma_read(m2_v1, addr_m2, N1*sizeof(uint32_t)); vm_vmul_vs32(mul_v, m2_v0, addr_m1[j]); vm_vadd_vv32(out_v, out_v, mul_v); m2_tmp = m2_v0; m2_v0 = m2_v1; m2_v1 = m2_tmp; } vm_dma_write(addr_mout, out_v, N1*sizeof(uint32_t)); // Addr updates addr_m2 = m_in2; addr_m1 = addr_m1 +N1; addr_mout = addr_mout+N1; } </pre>
Median	<pre> static inline void vminmaxdwn_vr(rv_reg vmin, rv_reg vmax){ vr_vmerge_vv32(v15, v15, vmin, V0_T); vr_vmin_vv32(vmin, vmin, vmax); vr_vmax_vv32(vmax, v15, vmax); } //Median code vr_vlwu(v0, img_ptr); vr_vlwu(v1, img_ptr+1); vr_vlwu(v2, img_ptr+2); ... vr_vxor_vv32(v0, v0, v0); for (int j = 0; j < image_height - 2; j++) { vr_vmerge_vv32(v9, v9, v3, V0_T); vr_vmerge_vv32(v10, v10, v4, V0_T); vr_vmerge_vv32(v11, v11, v5, V0_T); vr_vmerge_vv32(v12, v12, v6, V0_T); vr_vmerge_vv32(v13, v13, v7, V0_T); vr_vmerge_vv32(v14, v14, v8, V0_T); sort_windows_vr (image_width, image_height); img_ptr2 = img_ptr+(j+3)*image_width; vr_vsw(v4, output+(j*image_width)); vr_vsetvl(image_width); vr_vlwu(v6, img_ptr2); vr_vsetvl(image_width-2); vr_vslidedn_vv32(v7, v6, 1); vr_vslidedn_vv32(v8, v6, 2); vr_vmerge_vv32(v0, v0, v9, V0_T); vr_vmerge_vv32(v1, v1, v10, V0_T); vr_vmerge_vv32(v2, v2, v11, V0_T); vr_vmerge_vv32(v3, v3, v12, V0_T); vr_vmerge_vv32(v4, v4, v13, V0_T); vr_vmerge_vv32(v5, v5, v14, V0_T); } </pre>	<pre> static inline void vminmaxdwn_vm (uint8_t* qmin, uint8_t* qmax, uint8_t* q_sub, uint8_t* q_tmp){ vm_vmov_vv8(q_tmp, qmin, 0); vm_vsub_vv8(q_sub, qmax, qmin); vm_vcmv_ltz_vv8(qmin, qmax, q_sub); vm_vcmv_ltz_vv8(qmax, q_tmp, q_sub); } //Median code uint32_t* img_ptr = input; vm_dma_read(q0, img_ptr, (image_width-2)*sizeof(uint32_t)); vm_dma_read(q1, img_ptr+1, (image_width-2)*sizeof(uint32_t)); vm_dma_read(q2, img_ptr+2, (image_width-2)*sizeof(uint32_t)); vbx_set_vl(image_width-2); for (int j = 0; j < image_height - 2; j++) { vm_vmov_vv32(q3_tmp, q3, 0); vm_vmov_vv32(q4_tmp, q4, 0); vm_vmov_vv32(q5_tmp, q5, 0); vm_vmov_vv32(q6_tmp, q6, 0); vm_vmov_vv32(q7_tmp, q7, 0); vm_vmov_vv32(q8_tmp, q8, 0); sort_windows_vm (q0, q1, q2, q3, q4, q5, q6, q7, q8, q_sub, q_tmp, image_width, image_height); img_ptr2 = img_ptr+(j+3)*image_width; vm_dma_write(output+(j*image_width), q4, (image_width-2)*sizeof(uint32_t)); vm_dma_read(q6, img_ptr2, (image_width)*sizeof(uint32_t)); vm_vmov_vv32(q7, q6+1, 0); vm_vmov_vv32(q8, q6+2, 0); q0 = q3_tmp; q1 = q4_tmp; q2 = q5_tmp; ... } </pre>

TABLE III
DCT AND RGB2CMYK CODES.

	VR	VM
DCT	<pre> vr_vsetvl(BLOCK_SIZE); vr_vxor_vv16(v0,v0,v0); for (y = 0; y < num_tile_y; y++) { for (x = 0; x < num_tile_x; x++) { imgcol = (x+start_x)*BLOCK_SIZE; for (i = 0; i < BLOCK_SIZE; i++) { imgrow = i + (y+start_y)*BLOCK_SIZE; index_img = imgrow * IMAGE_WIDTH + imgcol; vr_vlh(v5, image+index_img); vr_accmul_vv16(v6,v1,v5); vr_vsetvl(1); vr_vslideup_vv16(v7,v6,i); vr_vsetvl(BLOCK_SIZE); vr_accmul_vv16(v6,v2,v5); vr_vsetvl(1); vr_vslideup_vv16(v8,v6,i); vr_vsetvl(BLOCK_SIZE); vr_accmul_vv16(v6,v3,v5); vr_vsetvl(1); vr_vslideup_vv16(v9,v6,i); vr_vsetvl(BLOCK_SIZE); vr_accmul_vv16(v6,v4,v5); vr_vsetvl(1); vr_vslideup_vv16(v10,v6,i); vr_vsetvl(BLOCK_SIZE); vr_vsetvl(BLOCK_SIZE); vr_vsr_l_vl16(v11,v11,SHIFT_AMOUNT); vr_vsr_l_vl16(v12,v12,SHIFT_AMOUNT); vr_vsr_l_vl16(v13,v13,SHIFT_AMOUNT); vr_vsr_l_vl16(v14,v14,SHIFT_AMOUNT); vr_vsetvl(BLOCK_SIZE); for (i = 0; i < BLOCK_SIZE; i++) { blkcol = (x+start_x)*BLOCK_SIZE; blkrow = i + (y+start_y)*BLOCK_SIZE; if (i==0) vr_vsh(v11, block_s+blkrow * IMAGE_WIDTH + blkcol); if (i==1) vr_vsh(v12, block_s+blkrow * IMAGE_WIDTH + blkcol); if (i==2) vr_vsh(v13, block_s+blkrow * IMAGE_WIDTH + blkcol); if (i==3) vr_vsh(v14, block_s+blkrow * IMAGE_WIDTH + blkcol); } } } } vm_set_vl(DCT_SIZE); vm_vshr_vl16(res_mul_tran, res_mul_tran,SHIFT_AMOUNT); ... vm_set_vl(BLOCK_SIZE); for (i = 0; i < BLOCK_SIZE; i++) { blkcol = (x+start_x)*BLOCK_SIZE; blkrow = i + (y+start_y)*BLOCK_SIZE; vm_dma_write(block_s+blkrow * IMAGE_WIDTH + blkcol, res_mul+i*BLOCK_SIZE, BLOCK_SIZE*sizeof(int16_t)); } </pre>	<pre> vm_set_vl(BLOCK_SIZE); for (y = 0; y < num_tile_y; y++) { for (x = 0; x < num_tile_x; x++) { imgcol = (x+start_x)*BLOCK_SIZE; for (i = 0; i < BLOCK_SIZE; i++) { imgrow = i + (y+start_y)*BLOCK_SIZE; index_img = imgrow * IMAGE_WIDTH + imgcol; vm_dma_read(img_values, image+index_img, BLOCK_SIZE*sizeof(int16_t)); for (j = 0; j < BLOCK_SIZE; j++) { index_coeff = j*BLOCK_SIZE; vm_accmul_vv16(res_mul_tran+BLOCK_SIZE*j+i, coeff_values+index_coeff, img_values); } } } } vm_set_vl(DCT_SIZE); vm_vshr_vl16(res_mul_tran, res_mul_tran,SHIFT_AMOUNT); ... vm_set_vl(BLOCK_SIZE); for (i = 0; i < BLOCK_SIZE; i++) { blkcol = (x+start_x)*BLOCK_SIZE; blkrow = i + (y+start_y)*BLOCK_SIZE; vm_dma_write(block_s+blkrow * IMAGE_WIDTH + blkcol, res_mul+i*BLOCK_SIZE, BLOCK_SIZE*sizeof(int16_t)); } </pre>
RGB2CMYK	<pre> vr_vmv_vx32(v6,F_SUB,0); vr_vmerge_vs32(v6,v6,v6,V0_T); ... rv_vlwu(v1, input_addr); for (int i = 0; i < image_height; i+=2) { input_addr = input_addr + image_pitch; vr_vlwu(v9, input_addr); vr_vsub_vv32(v2,v6,v1); vr_vand_vv32(v3,v2,v7); vr_vsr_l_vl32(v4,v2,8); vr_vand_vv32(v4,v4,v7); vr_vsr_l_vl32(v5,v4,16); vr_vand_vv32(v5,v5,v7); vminmaxdwn_vr(v3,v4); vminmaxdwn_vr(v4,v5); vminmaxdwn_vr(v3,v4); ... vr_vlwu(v1, input_addr); vr_vsub_vv32(v2,v6,v9); vr_vand_vv32(v3,v2,v7); vr_vsr_l_vl32(v4,v2,8); vr_vand_vv32(v4,v4,v7); vr_vsr_l_vl32(v5,v2,16); vr_vand_vv32(v5,v5,v7); vminmaxdwn_vr(v3,v4); vminmaxdwn_vr(v4,v5); vminmaxdwn_vr(v3,v4); vr_vmul_vv32(v5,v3,v8); vr_vsub_vv32(v2,v2,v5); vr_vsr_l_vl32(v3,v3,24); vr_vor_vv32(v2,v2,v3); vr_vsw_vv32(v2, out_addr); out_addr = out_addr + image_pitch; } </pre>	<pre> vm_dma_read(v_row_in1, m_in+image_pitch*0, image_pitch*sizeof(vbx_uword_t)); for (int i = 0; i < image_height; ++i) { vm_dma_read(v_row_in2, m_in+image_pitch*(i+1), image_pitch*sizeof(vbx_uword_t)); vm_vsub_vs32(v_cmyk, 0xFFFFF, v_row_in1); vm_vand_vs8_32(v_c, 0xFF, v_cmyk); vm_vand_vs8_32(v_m, 0xFF, (vbx_uword_t)(((vbx_ubyte_t *)v_cmyk)+1)); vm_vand_vs8_32(v_y, 0xFF, (vbx_uword_t)(((vbx_ubyte_t *)v_cmyk)+2)); vminmaxdwn_vm(v_c,v_m,v_sub,v_tmp); vminmaxdwn_vm(v_m,v_y,v_sub,v_tmp); vminmaxdwn_vm(v_c,v_m,v_sub,v_tmp); vm_vmul_sv32_8(v_k_tmp, 0x00010101, v_c); vm_vsub_vv32(v_cmyk, v_cmyk, v_k_tmp); vm_vor_vv32_8((vbx_uword_t*) (((vbx_ubyte_t *)v_cmyk)+3), (vbx_uword_t)(((vbx_ubyte_t *)v_cmyk)+3) v_c); vm_dma_write(m_out+i*image_pitch, v_cmyk, image_pitch*sizeof(vbx_uword_t)); v_row_in1 = v_row_in2; v_row_in2 = v_row_in1; } </pre>

TABLE IV
AUTOCORR AND SOBEL CODES.

	VR	VM
Autocorr	<pre> vr_vsetvl(DataSize); vr_vlwu(v1, InputData); vr_vlwu(v2, InputData); vr_vxor_vv32(v0, v0, v0); // set v0 = 0 vr_vmv_vx32(v0, v0, lag); for(lag = 0; lag < NumberOfLags; lag++){ vr_vsetvl(DataSize-lag); vr_vslidedn_vs32(v5, v2, lag); vr_vmul_vv32(v3, v1, v5); vr_accsrl_vi32(v4, v3, Scale); vr_vmv_vx32(&sum, v4, 0); vr_vmv_vx32(v7, sum, lag); } vr_vsetvl(DataSize); vr_vsrl_vi32(v6, v7, 16); vr_vsw_vv32(v6, AutoCorrData); </pre>	<pre> vm_dma_read(input, InputData, DataSize*sizeof(vbx_word_t)); for(lag = 0; lag < NumberOfLags; lag++){ vm_set_vl(DataSize-lag); vm_vmul_sv32(temp, input, input+lag); vm_accshr_vv16(output+lag, temp, Scale); } vm_set_vl(DataSize); vm_vshr_vv16(output, output, 16); vm_dma_write(AutoCorrData, output, NumberOfLags*sizeof(vbx_word_t)); </pre>
Sobel	<pre> vr_vsetvl(image_width); vr_vlhu(v3, v_row_in); vr_vsetvl(image_width-1); vr_vslidedn_vv16(v6, v3, 1); vr_vadd_vv16(v7, v3, v6); vr_vsetvl(image_width-2); vr_vslidedn_vv16(v8, v7, 1); vr_vadd_vv16(v5, v7, v8); ... vr_vsetvl(image_width); vr_vlhu(v2, v_row_in); vr_vmv_vx16(v13, r_FF, 0); vr_vmv_wx16(v14, r_mul, 0); for (int i = 0; i < image_height-(FILTER_HEIGHT-1); i++) { v_row_in = v_row_in+image_width; // v_luma_bot vr_vsetvl(image_width); vr_vlhu(v4, v_row_in); // v_sobel_row_bot vr_vsetvl(image_width-1); vr_vslidedn_vv16(v6, v4, 1); vr_vadd_vv16(v7, v4, v6); vr_vsetvl(image_width-2); vr_vslidedn_vv16(v8, v7, 1); vr_vadd_vv16(v10, v7, v8); // gradient_x vr_vsetvl(image_width); vr_vsl_vv16(v6, v2, 1); vr_vadd_vv16(v7, v3, v4); vr_vadd_vv16(v7, v7, v6); vr_vsetvl(image_width-2); vr_vslidedn_vv16(v6, v7, 2); vr_vabsdiff_vv16(v8, v7, v6); // gradient_y vr_vabsdiff_vv16(v7, v10, v5); //SUM_RENORM vr_vsetvl(image_width-2); vr_vadd_vv16(v6, v7, v8); vr_vsrl_vv16(v12, v6, RENORM); vr_vsetvl(image_width); vr_vor_vv16(v11, v3, v3); vr_vor_vv16(v3, v2, v2); vr_vor_vv16(v2, v4, v4); </pre>	<pre> luma_input = luma_input + image_width; vm_dma_write(v_luma_bot, luma_input, image_width*sizeof(vbx_uhalf_t)); // Calculate edges for (y = 0; y < image_height-(FILTER_HEIGHT-1); y++) { v_tmp = v_sobel_row_bot; vm_set_vl(image_width); vm_sll_vv16(v_gradient_x, v_luma_mid, 1); vm_vadd_vv16(v_tmp, v_luma_top, v_luma_bot); vm_vadd_vv16(v_tmp, v_tmp, v_gradient_x); luma_input = luma_input + image_width; vbx_dma_to_vector(v_luma_top, luma_input, image_width*sizeof(vbx_uhalf_t)); vm_vabsdiff_vv16(v_gradient_x, v_tmp, v_tmp+2); vm_vabsdiff_vv16(v_gradient_y, v_sobel_row_top, v_sobel_row_bot); // Re-use v_sobel_row_top as v_tmp v_tmp = v_sobel_row_top; // sum of absolute gradients vm_set_vl(image_width-2); vm_vadd_vv16(v_tmp, v_gradient_x, v_gradient_y); vm_srl_vv16(v_tmp, v_tmp, renorm); // Rotate luma buffers tmp_ptr = (void *)v_luma_top; v_luma_top = v_luma_mid; v_luma_mid = v_luma_bot; v_luma_bot = (vbx_uhalf_t *)tmp_ptr; </pre>

-
- [10] Synced, “Makers put their specialized ai chips on the table,” <https://syncedreview.com/2017/11/21/makers-put-their-specialized-ai-chips-on-the-table/>, (Accessed on 04/15/2019).
 - [11] A. Ou, Q. Nguyen, Y. Lee, and K. Asanovic, “A case for mvps : Mixed-precision vector processors,” in *International Workshop on Parallelism in Mobile Platforms (PRISM)*, June 2014.
 - [12] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, “Soft vector processors with streaming pipelines,” in *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2014.

CONCLUSION

Despite FPGAs offering a high level of programmability, potential for parallelism and flexibility, there are many factors to be taken into account before implementation on this type of platform.

In this work we showed that although application specific accelerators tend to show a higher speedup than solutions implemented for general purpose processors, these may not have favorable characteristics for their implementation on FPGAs. One of the most valuable lessons learned through this work is that it is important to understand the ratio of communication over computation of the parallelizable parts of an application before implementing an accelerator for them.

We showed that vector processors have favorable characteristics when implemented on FPGAs. We show that the utilization of a Vector Memory (VM) ISA instead of a Vector Register File (VR) ISA creates many advantages due to its flexibility. Modern FPGAs offers a reasonable amount of on-chip memory which can be used to implement a VM in order to mask the design communication time since it rivals a register file both in speed and capacity.

REFERÊNCIAS

ALVES, F. A. M. et al. Designing a collision detection accelerator on a heterogeneous cpu-fpga platform. In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. [S.l.: s.n.], 2017. p. 1–4. Citado na página 9.

ALVES, F. A. M. et al. Lessons learned on which applications benefit when implemented on cpu-fpga heterogeneous system. In: *2018 International on Embedded Computer Systems: Architectures, Modeling and Simulation*. [S.l.: s.n.], 2018. p. 1–6. Citado 2 vezes nas páginas 8 e 9.

ALVES, F. A. M. et al. Memory-based vector instruction set architectures. In: *2019 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. [S.l.: s.n.], 2019. p. 1–10. Citado 2 vezes nas páginas 8 e 9.