UNIVERSITY OF CALIFORNIA
RIVERSIDE

Acceleration of Compute-Intensive Applications on Field Programmable Gate Arrays

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Jose Milet Rodríguez Borbón

March 2020

Dissertation Committee:

Dr. Walid Najjar, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Amit Roy-Chowdhury
Dr. Sheldon X.-D. Tan

The Dissertation of Jose Milet Rodríguez Borbón is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to thank my advisor, **Dr. Walid Najjar** whose patience was immovable at every turn. In the difficult moments, your insight and experience provided the much needed guidance to my research. I would like to thank the members of my dissertation committee, Dr. *Nael Abu-Ghazaleh*, Dr. *Amit Roy-Chowdhury*, and Dr. *Sheldon X.-D. Tan*, for reviewing my dissertation and for listening to my research findings.

I would like to thank all the fellow students in my lab. *Xiaoyin Ma* for your early support during my first days in the lab. The end of your work in the lab was the beginning of mine. *Skyler Windh* for your friendship and for showing me the fundamentals of the digital world. It was always fun to chat and to drink tons of coffee with you. *Prerna Budhkar* for been a perceptive and reliable friend. At the times when the darkness was getting too close, you provided flashes of light to illuminate my road. *Bashar Romanous* for your patience and attentiveness. Although we had different points of view, at the end of the day, a bar of chocolate and a few minutes of discussion were enough to find common ground. *Amin K. Chahouki* for your friendship. While working with you, I learned to distinguish between sharp and rocky drawings. *Junjie Huang* for your devotion. One email was enough to get you on the road to the school.

I would like to express gratitude to my family. To my wife, *Ting*, for all your hard work, support, and patience during my studies. You always encouraged me to pursue my dreams; to my little daughter, *Emily*, for providing me tons of happiness and hope.

I would like to express gratitude to my father *Emilio*, and my mother *Alva Virginia*. From Dad and Mom, I learned to get up early, to work hard, to face hard times, and to

enjoy time with my relatives and friends. To my sisters and bothers-in-law *Olga*, *Sofia*, *Maidy*, *Gustavo*, *Rubio*, *Oscar*, and all my nephews.

I would like to express gratitude to my uncles, aunties, cousins, and all my extended family. To my uncles and aunts *Carlos*, *Jesus*, *Mario*, *Rey*, *Julia*, and *Cristina* for all their help. To my cousin *Rey Ariel* for showing me the roads to the college. To my cousins *Jose Hernan* and *Luz Mariana* for their great character.

I would like to acknowledge all my friends and fellow students who provided guidance and assistance: *Victor*, *Jose Hugo*, *Boris*, *David*, *Albenis*, *Frankly*, *Aldemar*, *Harvey*, *Fernando*, *German*, *Jonathan*, *Elizabeth*, *Sankalp*, *Mark*, *Dawn*, *Jason*, *Nhat*, *Joobin*, *Stefan*, *Uy*, and many others.

In addition, chapter three of this dissertation contains two of my previous published works. The full citations are:

Jose M. Rodriguez-Borbon, Xiaoyin Ma, Amit K. Roy-Chowdhury, and Walid Najjar. Heterogeneous acceleration of HAR applications. IEEE Transactions on Circuits and Systems for Video Technology, Vol. 30, No. 3, March 2020.

Xiaoyin Ma, Jose M. Rodriguez-Borbon, Amit K. Roy-Chowdhury, and Walid Najjar. Optimizing hardware design for human action recognition. IEEE 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, 2016, pp. 1-11.

To my family for all their hard work and support.

ABSTRACT OF THE DISSERTATION

Acceleration of Compute-Intensive Applications on
Field Programmable Gate Arrays

by

Jose Milet Rodríguez Borbón

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2020
Dr. Walid Najjar, Chairperson

In recent years, the field of high performance computing has been facing a new challenge: achieving *high throughput* at the *lowest energy cost*. Recent interest in field programmable gate arrays (FPGA) has been spurred by their significant growth in *density* and *speed*. While they were, until recently, considered an alternative to application-specific integrated circuits (ASIC) for low volume designs, they have become an alternative compute platform that achieves much higher floating point operations (FLOPS) per unit of energy.

To partially offset the *massive* cost of the energy consumption in CPUs and GPUs, this dissertation explores the design and implementation of high-throughput energy-efficient compute-intensive applications on FPGAs. I show how these demanding applications can be built. To this end, I have chosen three applications from diverse domains: (a) *Human Action Recognition* from the field of computer vision and image processing, (b) *Quantum Dynamics Simulations* from the field of computational physics, and (c) the *QR decomposition of Tall-and-Skinny Matrices* from the field of high performance linear algebra. Regarding (a), I show that FPGAs combined with GPUs outperforms homogeneous platforms by a factor of

1.3 while consuming 50% less energy. In regards to (b), for systems having over a thousand atoms, I show that FPGAs using *wide* pipelines oriented towards the processing of sparse matrices surpasses competing platforms by a factor of 1.5 while consuming 4.0× less energy. In terms of (c), for tall-and-skinny matrices having over 50K rows, I show that FPGAs using *wide* and *deep* pipelines can exceed the performance of competing platforms by a factor of 1.5 while executing as much as twice more FLOPS per unit of energy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Compute-intensive applications execute a large number of operations per clock cycle, transfer extensive amounts of data between the on-chip and off-chip memory, and consequently require important on-chip memory resources. In this dissertation, I explore the acceleration of compute-intensive applications on field programmable gate arrays (FPGA). The acceleration of such applications is important because of the advent of three trends: the *massive increase* of operational costs in data centers due to energy consumption, the *exponential increase* of resources available on FPGAs, and the *high energy efficiency* demonstrated by FPGAs while executing arithmetic operations.

The first critical trend taking place is the *massive increase* of the operational cost of data centers due to energy consumption [14, 49, 138] as shown in Figure 1.1. Over the years, chip makers, along with the high performance community, have sought to improve performance without accounting for energy consumption. As shown in this figure, between the year 2000 and 2005, the energy consumption in data centers in the U.S. doubled [14].

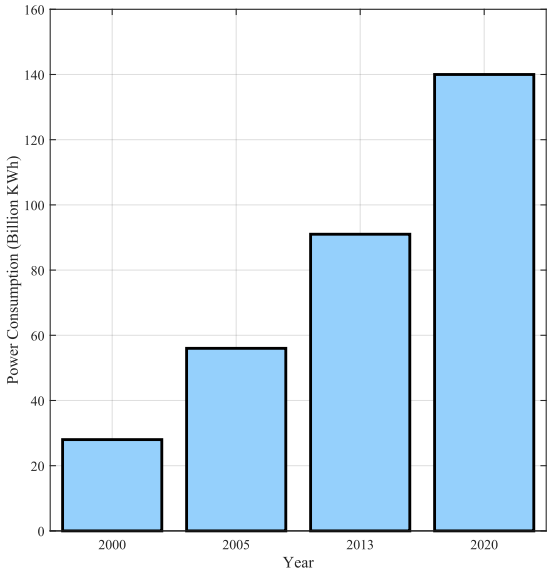


Figure 1.1: U.S. data center power consumption [29].

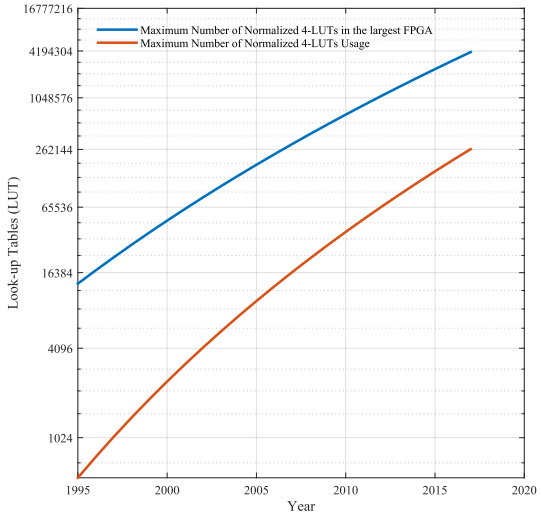
For the year of 2008, it has been reported that the annual cost due to energy consumption exceeded the acquisition cost for small and medium-size data centers [7]. For the year 2013, it was estimated that data centers and servers consumed over 70 billion kWh (kilowatt hours) i.e. the equivalent to 1.8 percent of the total electricity consumption in the U.S., costing about $ 7.0 billion in electricity per year [49]. These energy expenditures are the equivalent to the energy consumption of over 6.0 million U.S. households [49]. In addition to the energy required for the operation of the servers, the infrastructure needed to operate these data centers (cooling systems, air-conditioning, and power delivery devices among others) requires additional energy encompassing up to 50% of the energy required by the data centers [14].

Due to such large energy expenditures, DARPA has declared energy-efficient computing as the next frontier for the high performance computing community. For the new

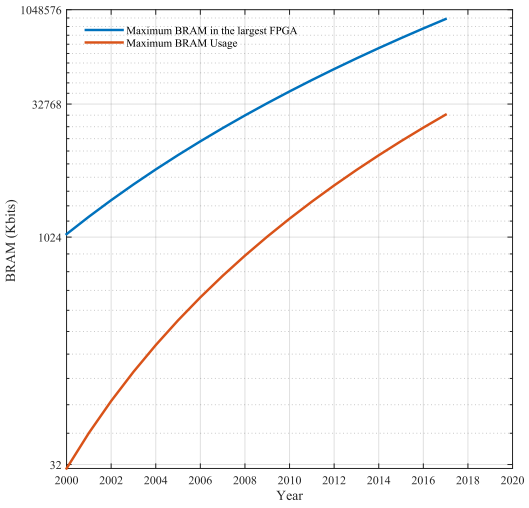
decade, the most energy-efficient computer in the U.S. should deliver at much as one exaflop (10^{18} floating point operations per second) using only 20 million watts (MW) [138]. For today's standards, achieving such impressive performance per unit of energy would require over 50× gains in throughput with minimal increases in energy expenditures [9].

The second important trend taking place is the *exponential* increase in FPGAs density and throughput since their inception. To illustrate FPGA improvements in *resources* and *performance*, I report the increases in the amount of Look-up Tables (LUT), block RAMs (BRAM), digital signal processors (DSP), and frequency over time. In addition, I show how these resources have been utilized. *Shannon* [127] and her team have measured the resources gains of FPGAs in the last 25 years. In their work [127], they sampled the amount of resources (LUTs, BRAMs, DSPs) of the largest FPGA per year per vendor. In addition, for the same period of time, they sampled the resource utilization (LUTs, BRAMs, DSPs) for a number of projects. Given the availability of resources per device per year and the utilization of resources per project per year, they interpolate the data and report the trends.

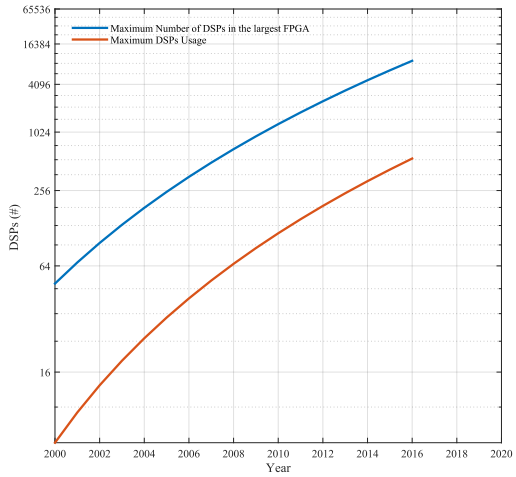
For the availability of LUTs and its utilization, the observed trends are shown in Figure 1.2a. To construct this figure, first, the maximum size FPGA per vendor per year is found. Next, using the FPGA specifications, a scatterplot showing the LUTs per FPGA per year is constructed. Then, a regression curve modeling the relation between the number of LUTs and time is built as shown in the curve at the top. Moreover, for the same period of time, a number of research projects are sampled along with the LUTs utilization. As before, a scatterplot showing the LUTs utilization per project per year is constructed.



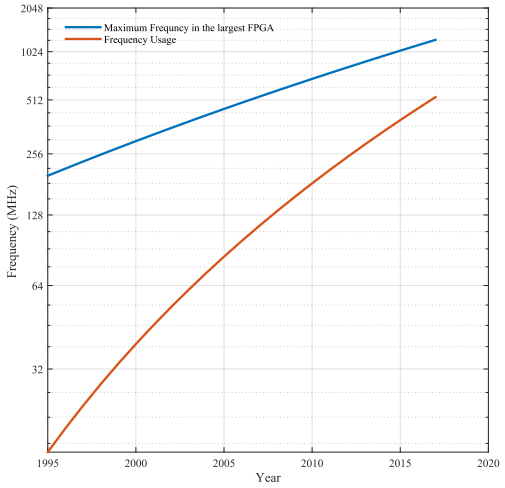
(a) Evolution of 4-LUTs in the largest FPGA and the usage per year [127].



(b) Evolution of the BRAMs in the largest FPGA and the usage per year [127].



(c) Evolution of DSPs in the largest FPGA and the usage per year [127].



(d) Evolution of the maximum frequency of the largest FPGA and the frequency achieved per year [127].

Then, a regression curve modeling the relation between the parameters is built as shown in the bottom line.

The same methodology is used to measure the availability of BRAMs and DSPs as shown in Figures 1.2b and 1.2c respectively. Moreover, to achieve high performance, FPGA applications have to target high operating frequencies as well. Using the methodology described above, *Shannon et al.* [127] reports on the maximum operating frequency of the largest device per vendor per year and the operating frequency of the sampled projects per year as shown in Figure 1.2d. By inspection of Figures 1.2a, 1.2b, 1.2c, and 1.2d, I conclude that over time, there has been an *exponential increase* of the resources and the operating frequency of the FPGAs.

To better assess the computational power of FPGAs with respect to CPUs and GPUs, in terms of FLOPS, the following table shows the nominal peak performance of typical platforms. The FLOPS for CPUs and GPUs is as per the specifications. To derive the number of FLOPS in the FPGA, we proceed as follows. A single precision fused multiply-and-add operation uses 5 DSPs while the same operation in double-precision takes 10 DSPs. The chip has 6840 DSP units. As a result, we get $2 \times 1368 \times 400 \times 10^6 = 1094$ single precision GFLOPS per second. Similar calculations apply to the peak performance in double precision.

Table 1.1: Comparison of the peak number of FLOPs per platform

Platform	Frequency	Single FLOPS/s	Double FLOPS/s
NVIDIA K40 GPU	745 MHz	4.29 TFLOPS	1.43 TFLOPS
Xilinx UltraScale(VU9P) FPGA	400 MHz	1.09 TFLOPS	547 GFLOPS
Intel E5-2697V2 CPU (12 Cores)	2.7 GHz	518.4 GFLOPS	259.2 GFLOPS

In addition to the *massive increase* of the operational cost due to energy consumption in data centers and the *exponential increase* of resources in the FPGA, another trend has made its mark on the field: FGPAs have shown to be more *energy efficient* than competing platforms at the processing compute-intensive workloads. Figure 1.3 compares the energy efficiency of CPUs and FPGAs. In the work of *Horowitz* [55], it is shown that

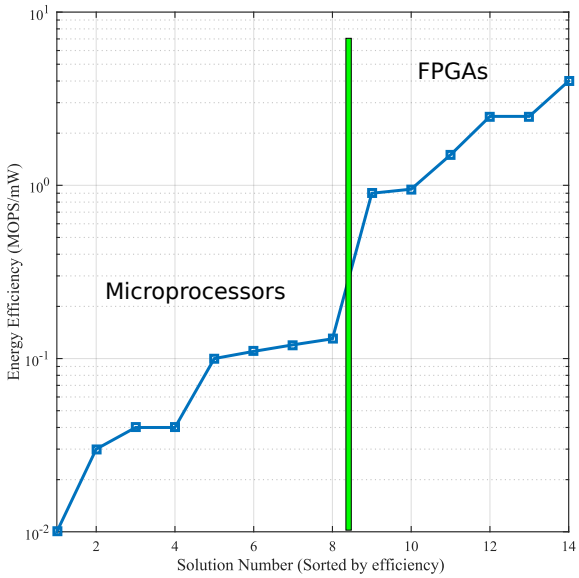


Figure 1.3: Energy efficiency as a function of solution number [55]. All the solutions to the left of the vertical are software-based while the the solutions to the right are hardware-based.

conventional Von Neumann architectures (CPUs and GPUs) are capable of achieving a high throughput at the expenses of consuming excessive amounts of energy. For instance, this work shows that the overall execution of a 32-bit *add* instruction on a 45 nm CPU takes about 70 pJ, with the actual *add* operation consuming only 0.1 pJ; 99.9% of the energy is wasted on tasks including fetching (and decoding) the instructions and controlling the datapath. In addition, this work also shows that one can achieve important energy savings

(on the order of a hundred-fold or more) through the use of hardware specialization (digital signal processors, FPGAs, or ASICs).

By observing these *trends*, this dissertation explores whether it is possible to build compute-intensive applications able to match, or even outpace, the performance of competing platforms while delivering better throughout per unit of energy. To do so, I have chosen three applications from diverse domains. They are (a) *Human Action Recognition* from the field of computer vision and image processing, (b) *Quantum Dynamics Simulations* from the field of computational physics, and (c) the *QR decomposition of Tall-and-Skinny Matrices* from the field of high performance linear algebra. When taken together, these three applications have the following characteristics:

- They require the execution of hundreds of arithmetic operations (i.e. DSP) per clock cycle. For example, in the case of (a), the clustering algorithm requires the execution of hundreds of multiply-and-add operations per clock cycle. In the case of (b), the blocked multiplication of complex matrices requires wide pipelines able to execute hundreds of floating point operations per clock cycle. In the case of (c), the fast decomposition of matrices requires deep and wide pipelines able to execute hundreds of floating point operations per clock cycle.

- To support a high number of arithmetic operations per clock cycle, these applications require extensive usage of block RAMs. In the case of (b), to support the multiplication of complex matrices on the FPGA, it is required to store the input values, partial results, and the final results in large BRAMs. For (c), the use of on-chip tiles is paramount and as a result, the requirements for BRAMs are extensive as well.

- The requirements of hundreds of operations per clock cycle along with extensive block RAMs, implies the need for additional resources to glue together DSPs and BRAMs. Those resources can include LUTs, registers, multiplexers, and decoders.

- Along with the extensive usage of resources, these applications require a high operating frequency in order to match, or surpass, the performance of competing platforms.

- For these applications, it is desirable to execute a high number of floating point operations per unit of energy. For example, in the case of (b), executing a simulation with a few thousand atoms can take days, and as a result, the use of energy-efficient platforms is crucial.

The rest of this dissertation is organized as follows. Chapter two details previous work. There, I describe the experimental platforms used in this work. In addition, I present previous approaches to human action recognition, quantum dynamics simulations, and the QR decompositions of tall-and-skinny matrices in FPGAs versus competing platforms such as CPUs and GPUs.

In chapter three, I present my compute-intensive human action recognition engine. By taking advantage of an heterogeneous approach, my design is able to match and outpace the performance of similar applications running on homogeneous platforms by a factor of 1.3 while using 50% less energy. The high performance achieved by my design is due to the high utilization of resources. My design exploits the extensive number of DPS in the target device, and as a result, it executes hundreds of fix-point operations per clock cycle while operating at over 150 MHz.

In chapter four, I show my design for the execution of quantum dynamics simulations on FPGAs. By offloading the most intensive calculations of these simulations onto an FPGA, I show that FPGAs can exceed the performance of commercial libraries running on GPUs and CPUs. For systems having thousands of atoms, my design is 1.5× faster and has lower expenditures of energy. To achieve such performance, my engine relies on the execution of over five hundred single-precision floating point operations per clock cycle while operating at 166 MHz.

In chapter five, I present my design for the QR decomposition of tall-and-skinny matrices on FPGAs. Compared with commercial libraries running on GPUs, my design is 3.0× faster for matrices having up to 256 columns. Compared with highly optimized libraries running on CPUs, my design is 1.5× faster for matrices having over $50K$ rows. Additionally, my design uses less energy. The high performance of my engine relies on the execution of over 256 double-precision floating point operations per clock cycle while operating at 266 MHz. In chapter six, I present my conclusions.

Chapter 2

Background

2.1 Reconfigurable Architectures

FPGAs devices appeared at the beginning of 1980. At first, FPGAs mostly consisted of programmable logic devices (PLD) and complex programmable logic devices (CPLD) that facilitated the communication between digital entities as well as the communication with the surrounding environment. As these programmable devices gained popularity, they made inroads in other markets, namely the implementation of network and memory interfaces. Due to the challenges involved in processing millions of networks packets quickly, FGPA designers added more resources to the device. As a result, FPGAs grew in performance and density. Around 2010, FPGAs made another wave of expansion in the field of general purpose computation. To achieve this, FGPA designers added additional LUTs, BRAMs, DSPS, and fast interfaces to move data to and from the FPGA. By the year 2015 or so, FPGAs achieved density and performance comparable to low- and

middle-end CPUs. As a result, academic and industrial computational solutions based on FPGAs increased.

Today, in addition to high throughput interfaces to access off-chip memories, FPGAs incorporate millions of LUTs, thousands of DSP cores, and thousands of BRAMs. As such, FPGAs are making their way in to the field of high performance computing and its applications. Figure 2.1 shows the fundamental components of a modern FPGA [66, 24]. As shown in this figure, the architecture consists of configurable logic blocks (CLB), pro-

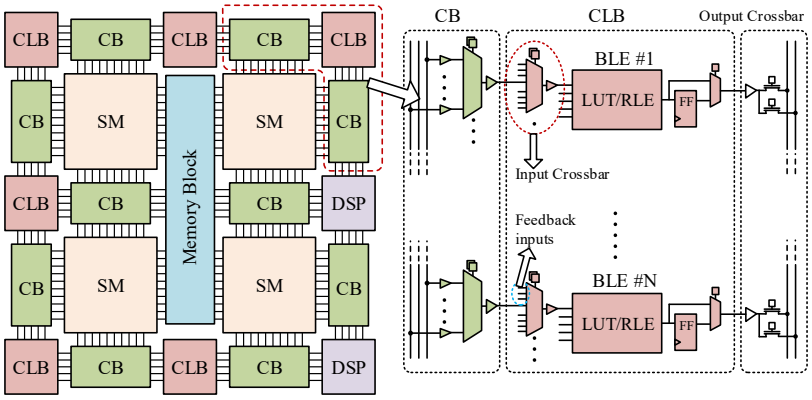


Figure 2.1: The FPGA reconfigurable architecture.

grammable switch matrices (SM), configurable block memories (BRAM), configurable I/O blocks, DSP blocks, among others.

CLBs are the fundamental building blocks of FPGAs. These blocks contain combinational and sequential logic oriented towards the implementation of small finite state machines (FSM). CLBs usually contain RAM units to implement combinational logic functions in the form of LUTs, flip-flops to implement registers, and multiplexers to facilitate the routing of the signals. In addition, CLBs contain clock and reset signals to drive and reset the execution of the internal logic.

Programmable SMs facilitate the communication between the CLBs, BRAMs, DSPs, and other components inside the FPGA. By using long lines, these switches make it possible for fast communication between blocks that might be far apart. In addition, these switches have short lines to facilitate the communication between nearby blocks. In order to enable or disable connections, these SMs use transistors.

Block RAMs are distributed units of memory able to hold up data. Typically, these BRAMs can store either 18 or 36 Kbits of data. These BRAMs can be shaped in multiple ways. BRAMs can be cascaded (or adjoined) so as to create taller (or wider) blocks. In addition, BRAMs can be addressed in various modes. For example, single port BRAMs can be addressed by a single digital process (for example, a FSM) while dual port BRAMs can be addressed by multiple digital processes (for example, multiple FSMs).

Configurable I/O blocks allow communication between the FPGA and the surrounding environment. They are designed to read and write signals to and from the surrounding devices. Usually, these I/O pins are connected to input and output buffers and they can be programmed for active high or active low signaling.

DSP blocks facilitate the implementation of operations in either integer, fix-point, or floating point arithmetic. These operations include addition, multiplication, and multiply-and-accumulate among others. Usual inputs and outputs to these block include signals having 18, 25, or 32 lines. Because DSPs are implemented in hard logic, they are able to operate at high frequencies.

In addition to these cores, FPGA vendors also include traditional CPU processors (soft-processors) in the FPGA fabric. These soft-processors use limited amount of resources

and operate with low power budgets. As a result, FPGAs equipped with soft-processors are mostly deployed in embedded systems. Other FPGA cores include analog cores and high density RAM cores (ultra RAMs).

2.2 The Micron Wolverine Co-Processor Series

Having described the fundamental components of FPGAs, in this section, I describe the development platform I have used through my work. The Wolverine platform is a heterogeneous platform [94] that offers the best features of two different worlds. While the CPU unit allows the execution of highly optimized software libraries, the FPGA unit allows the execution of highly optimized hardware pipelines. The Wolverine co-processor is a programmable hardware platform that can be reconfigured so as to meet the needs of different workloads. These reconfigurable solutions are usually called *personalities*. These personalities can be used to fully, or partially, accelerate workloads coming from multiple domains. The architecture of the Wolverine heterogeneous co-processor is shown in Figure 2.2.

As shown in this figure, the Wolverine platform is made of a CPU, a co-processor, and the global shared virtual memory. The communication between the host and the co-processor is via PCI express lines. The Wolverine co-processor has four main components: the host interface, the application engines (AE), the memory crossbar network, and the memory controllers (MC).

These AEs are reconfigurable and can be programmed as per the application requirements. As shown in this figure, each AE is able to both send and receive instructions to and from the host interface. Typical instructions include *starting*, *resuming*, and *finishing*

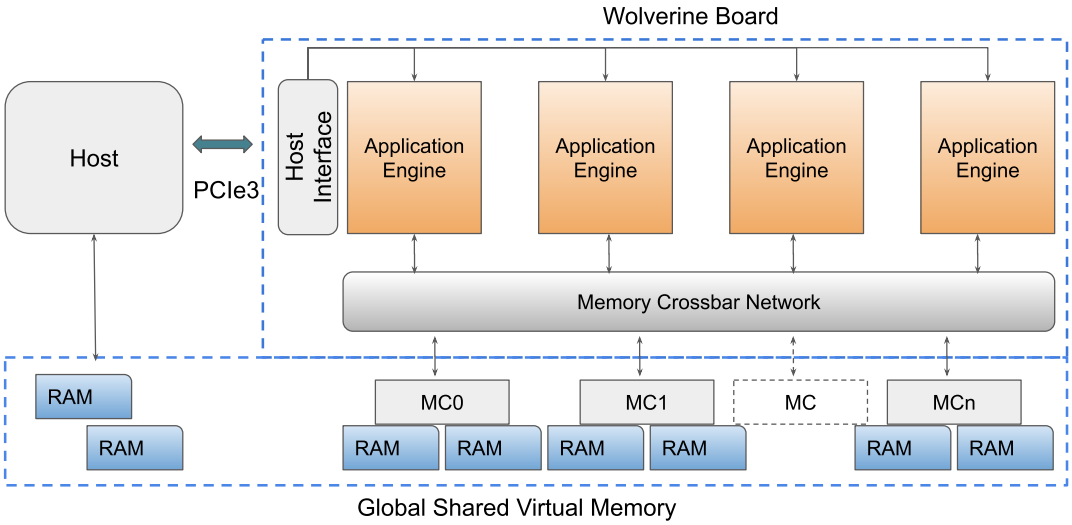


Figure 2.2: The Wolverine reconfigurable co-processor.

a task. Likewise, AEs can issue request to the memory crossbar network. Typical requests to the memory crossbar network include executing a read, executing a write, executing an atomic write, among others. For every memory request made, the memory subsystem issues a response. These responses can include the confirmation of a write operation and the response to a read request. In addition, some Wolverine boards allow direct communication between the AEs through an AE-to-AE interface. Otherwise, the communication between AEs is via the host or the FPGA off-chip memory.

The host interface allows the communication between the host and the co-processor. This interface is responsible for processing all the instructions coming (or going) from (or to) the host application. When the destination of a host instruction is an AE, the host interface redirects that instruction to the respective AE. Likewise, this interface also redirects the instructions originating in AEs going to the host.

As mentioned above, the memory crossbar network allows the communication between the AEs and the MCs. Each time that an AE issues a command to the FPGA off-chip memory, this crossbar is responsible for redirecting the request to the appropriate MC. Likewise, each time that the a MC sends a response to a AE, this crossbar redirects the response to the appropriate destination. In short, this crossbar presents a uniform interface of the off-chip memory subsystem allowing for an easy-to-use communication mechanism between the AEs and the off-chip memory.

The Wolverine memory subsystem typically has four memory controllers (MC). Each MC has one DIMM and each DIMM has two ranks. Moreover, each rank has eight banks. As a result, this memory subsystem has 64 memory banks.

2.3 Wolverine Co-Processor Series Comparison

Table 2.1 compares the Wolverine co-processors used in this work.

Table 2.1: Micron Wolverine comparison

Feature	Wolverine I	Wolverine II-A	Wolverine II-B
Year	2014	2018	2018
FPGA	Virtex-7 (VX2000)	UltraScale+ (VU7P)	UltraScale+ (VU9P)
- Registers	2443K	1576K	2364K
- Lookup Tables (LUT)	1221K	788K	1182K
- Block RAMs	1203	1440	2160
- Block Ultra RAMs	-	640	960
- DSPs	2160	4560	6840
Memory Channels	32	32	32
Off-chip Memory	32 GB	64 GB	64 GB
Bandwidth	42.5 GB/s	68 GB/s	68 GB/s
Frequency	166 MHz	266 MHz	266 MHz

One of the main differences between the Wolverine I and the Wolverine II series is the operating frequency. In addition, the Wolverine II has more in-chip memory and has a larger bandwidth. In this work, initial projects have targeted the Wolverine I co-processor while recent developments have targeted the Wolverine II board.

2.4 Related Work

Having described the co-processors used in this work, next, I present relevant work in the acceleration of *Human Action Recognition*, *Quantum Dynamics Simulations*, and the *QR Decomposition of Tall-and-Skinny Matrices*.

2.4.1 Human Action Recognition (HAR) Applications on FPGAs

Early HAR applications are based on HCF and consist of four steps: sampling the video signal, computing features per region of interest, merging these features to get a fixed-size video feature, and finally, training a classifier. Sampling is dense or sparse [51]. Techniques to compute the features of a region include the scale-invariant feature transform (SIFT) [85] and the histogram of oriented gradients (HOG) [90, 69]. The features of the regions are usually merged via a bag-or-words approach [74, 26], and SVMs are commonly used for classification [123]. Initial work in this field includes a behavioral recognition system via sparse spatio-temporal features [31]. Similarly, spatio-temporal features, along with local SVMs, have been proposed [123].

Recent approaches to HAR algorithms are based on learned features [76]. In this approach, a machine learning algorithm samples the video at predetermined positions,

learns the local features, aggregates these features, and finally, classifies them. Early work using learned features includes a biologically-inspired system for action recognition [61]. This system takes inspiration from the dual stream organization of the visual cortex: one stream processes the shapes while the second stream processes the motion. Also, a CNN containing a three-dimensional receptive field learns to classify human actions [54]. This network generates action descriptions and uses a feed-forward NN in the classification stage.

In order to improve the accuracy of traditional CNNs, one stream CNNs, researchers have studied two-streams CNNs [64, 132, 148, 36, 156, 33]. In a typical configuration, the first stream learns the spatial features while the second stream learns the temporal features. Variations on this model set one of the streams to learn the features of the optical flow, the motion flow, or the context of the scene, among others. Moreover, the outputs of the streams are usually fused via a fully connected feed-forward neural network. Further, to reduce the computational complexity of two-streams CNNs, factorized CNNs are proposed [140]. Factorized CNNs using spatial convolutional kernels along with temporal convolutional kernels are designed to reduce the complexity of the CNNs while maintaining the recognition accuracy.

Hybrid methods using HCF and learned features have been studied as well. In this approach, the fusion of HCFs boost the performance of the CNNs. Likewise, the fusion of learned features boost the performance of HCF-based classifiers. These designs include a method for recognizing human actions via the fusion of HCF features, based on dense trajectories, and deep-learned features [147]. Also a system for human detection and tracking that uses learned features and SVM classifiers [155]. Further, to save computations,

it has been evaluated whether features extracted from CNNs can be re-purposed for related tasks [32].

Hou et al. [57] proposes an FPGA real-time HAR system operating at 600 fps. It has a recognition rate of 93.2% when working with a human gesture database with four actions. The recognition rate drops to 80.8% when a few additional gestures are added. Although this system has a competitive throughput, its recognition rate is nontrivial to predict when working with challenging benchmarks having a larger number of classes. Conversely, my system achieves *competitive accuracy* with benchmarks having over 50 classes. Additionally, my design has a *larger throughput* ranging from 455 fps to $1,304$ fps.

2.4.2 Quantum Dynamics Simulations on FPGAs

Modern quantum chemistry techniques depend critically on massively-parallelized hardware to enable the calculation of both ground- and electronic-excited states. Platforms including CPUs, application-specific integrated circuits (ASICs), GPUs, and FPGAs have been used in the task of simulating the classical dynamics of atoms and molecules, and lately, these platforms have been used in the task of simulating the behavior of systems governed by the laws of quantum dynamics.

The Anton machine [129] is one of the pioneering ASIC platforms dedicated to the simulation of the classic dynamics of molecules. This massive parallel device is composed of hundreds of processing elements interacting via a fast tri-dimensional communication network. Each processing element is composed of two high-throughput interacting subsystems. The first subsystem calculates the forces between interacting particles and the second computes the Fast Fourier Transform (FFT) among other calculations. While ASICs chips

can be orders of magnitude faster than general purpose computational platforms, they are very expensive to design and hard to modify.

In addition, researchers have targeted the acceleration of classical and quantum dynamics simulations in GPUs. The work in [3] shows that molecular dynamics simulations can be fully implemented on GPU notwithstanding the lacking of support for double precision floating point arithmetic in early devices. Moreover, the work in [43, 120] develops a set of libraries targeting the acceleration of molecular dynamics entirely on GPUs. By taking advantage of the message-passing interface (MPI), as well as the support for double-precision floating point calculations, these libraries are able to run in either single or multiple GPU environments. Due to the availability of more single precision floating point units than double precision units in GPUs, the use of dynamic precision arithmetic [86] has been proposed. In this work, it is shown that the error of the calculations of the electron repulsion integrals can be minimized by calculating the large integrals in double precision and the other integrals in single precision.

Moreover, the design of hardware engines for the simulation of molecular dynamics in FPGAs has been addressed as well. For instance, a large-scale reconfigurable cluster for the simulation of molecular dynamics has been proposed [75]. This system features a high-bandwidth, low latency 3D torus network that makes possible the communication between the kernels. The work in [154] presents an end-to-end engine targeting the simulation of molecular dynamics. This engine features online particle-pair generation, short and long range force evaluation, bonded interactions, motion updates, and particle migration. In

this work, they propose a number of micro-architectures to compute bounded interactions, force summations with motion updates, and FFTs, among others.

2.4.3 QR Decomposition of Tall-and-Skinny Matrices on FPGAs

Previous researchers have addressed the development of efficient software and hardware solutions to decompose matrices via the QR method. In the area of CPUs, important techniques to increase the performance of the QR decomposition has been proposed [12, 30, 46]. For instance, methods to consolidate the application of Householder reflectors via matrix multiplications [12] make it possible to accelerate the computation in platforms that have large caches. Furthermore, communication-avoiding methods are gaining traction [30]. These techniques are able to execute the QR decomposition in multiple nodes while minimizing the exchange of data. As far as GPUs, a communication-avoiding QR factorization routine for TSMs has been presented [4]. In this work, the entire decomposition is executed on the GPU via compute-bound kernels. A high-performance method to execute the QR factorization on GPUs is described in [65]. This method takes advantage of the highly optimized matrix multiplication routines in GPUs and outperforms existing libraries such as the MKL and MAGMA routines for large matrices.

Researchers have proposed cores targeting the QR decomposition of matrices in FPGAs as well. In the case of QR decomposition via GSs, CHs, and GRs, these works include [125, 150, 40, 13]. The QR decomposition via the HR method has not received as much attention despite its compelling features including lower complexity, higher stability,

Table 2.2: Comparison of QR decomposition via Householder reflectors designs in FPGAs

Work	Ref. [141]	Ref. [114]	**This Work**
Year	2011	2012	2020
Synthesis Tool	ISE 10.0	ISE 10.0	Vivado 17.2
FPGA	Virtex-5	Virtex-6	Virtex-7 (Ultrascale+)
Frequency MHz	150	315	266
Peak GFLOPs	10.2	129	68
Max. FLOPs/Cycle	64	409	256
Efficiency for TSM (%)	7.0 - 11.0	36.0	28.7 - 54.2
Target Matrix Shape	Square	TSM	TSM
Matrix Shape (R,C)	(10K, 10K)	(10K, 51)	(10K, 64-512)
Block Parallelization	2	1	16
Dot-Products	Stream Reduction	Tree	Stream Reduction
Pipelined Reflectors	16	1	4

and its larger degree of parallelism [42]. Table 2.2 compares my work with existing HR approaches.

In previous work, *Tai et al.* [141] proposed a QR decomposition engine for the decomposition of large *square* matrices. In this design, the input matrix A is divided into *square* tiles, and then the HR decomposition is executed in multiple steps. In the first step, HR decomposition is applied to the top leftmost tile, and then, the computed reflectors are saved into the off-chip memory. Next, the engine reads the remaining tiles in the top row (one at a time), reads the reflectors, and applies them. In the second step, the HR decomposition takes the top leftmost tile as input (an upper-triangular tile) resulting from the previous step, and the tile right below and executes the HR decomposition. As before, the reflectors computed in this step are saved to the off-chip memory. Next, the engine reads the remaining tiles in the first and second row, reads the saved reflectors, and applies them. At the end, by combining steps one and two, the QR factorization of the input matrix is achieved.

My work is distinct from the previous study in multiple ways. First, the work by *Tai et al.* targets the decomposition of large *square* matrices, whereas my work targets the decomposition of *TSMs*. For *TSMs*, I make use of recent developments including the decomposition of matrices via binary trees and the fast decomposition of upper triangular matrices [30]. Second, my work targets the decomposition of a large number of tiles in parallel whereas the work by *Tai et al.* targets the decomposition of fewer tiles. Third, due to the shape of the input matrices, my work targets *Tall-and-Skinny* tiles instead of *Square* tiles, since the former favors a higher performance for the problem at hand. On the other hand, both works include common techniques such as the use of stream reduction circuits [40], the application of reflectors via deep pipelines, and the decomposition of on-chip tiles.

In previous work, *Rafique et al.* [114] proposed an FPGA engine targeting the decomposition of *very skinny* matrices, matrices having up to 51 columns. In their work, the input matrix is first divided in blocks having twice as many rows as columns. Next, these blocks are brought to the on-chip memory, and then they are decomposed via a HR decomposition engine. The output of the first step is a series of upper triangular blocks. In the next step, two triangular blocks are brought to the on-chip memory, and then they are decomposed. The results are written back to the off-chip memory. The process of reading, merging, and writing upper triangular blocks continues until the final decomposition is found.

My work has a number of deviations from this prior work. First, while the work by *Rafique et al.* targets the decomposition of up to two blocks in parallel, my work

targets the decomposition of multiple blocks simultaneously. Second, my work targets the decomposition where large blocks have to be *tiled* before they are processed, whereas the work by *Rafique et al.* targets the decomposition of matrices where the individual blocks fit in on-chip memories (blocks size 102×51), Third, as described in chapter five, the execution of the HR decomposition requires the execution of large dot-products. While the work by *Rafique et al.* executes these products via *resource-intensive* reduction circuits, my work uses *resource-aware* reduction circuits [40]. As stated in the Introduction, *tree-based reduction circuits* are very fast at the cost of using prohibitive amounts of hardware resources. Although the proposed reduction trees have an impressive peak performance for the problem at hand, achieving as much as half of this performance is not feasible. For example, when the input blocks are upper triangular, the reduction tree operates over zero elements most of the time, and as a consequence, these reduction trees only deliver about one quarter of their peak performance. In this scenario, my engine achieves over 50% of the peak performance.

Chapter 3

Acceleration of HAR Applications

Human action recognition (HAR) algorithms take one or more video sequences as input, usually a few hundred frames, and produce one or more output(s) categorizing the possible action(s) executed by the actor(s) within the video clip(s). Applications of HAR algorithms include health care, assisted living, surveillance, automated video indexing, security, autonomous navigation, robotics, mobile computing, etc. Even though significant progress has recently been made in the design and implementation of HAR applications, several challenges remain: higher throughput for handling large video sequences, lower complexity for real-time applications, highly parallel implementations for faster response times, and energy efficient designs for embedded and mobile applications [51, 146, 56].

HAR algorithms rely on the extraction of video features. These can be computed at regular positions (called dense sampling) or at points of interest (sparse sampling). Video features can be designed by experts in the field, called hand-crafted features (HCF), such as histogram of gradients (HOG), or they can be inferred using machine learning techniques

or learned features, such as convolutional neural networks (CNNs). Once the video features are extracted, they can be used to train a classifier, such as a support vector machine (SVM) or a softmax classifier.

HAR implementations based on CNNs have been shown to achieve a higher recognition accuracy than HCF HAR algorithms. However, this advantage comes at a price: lower throughput, higher computational load and costly energy consumption per frame. *Suleiman et al.* [139] shows that HCF HAR algorithms are $311X$ more energy efficient than their CNN counterparts. When the features are learned with larger CNNs, the throughput gap grows to the order of the thousands. Moreover, *Zou et al.* [161] shows that HAR algorithms based on learned features with only three convolutional layers have comparable accuracy and $100X$ higher energy usage than HCF HAR algorithms. As the accuracy of the CNN increases, the energy gap grows dramatically.

The proliferation of video cameras and other forms of image sensing technologies have pushed a large part of the video processing tasks to the edge devices and hence have increased the pressure on achieving high processing rates at low energy budgets. *My objective is the explore and evaluate the designs of HOG3D-based HAR that can achieve both high throughput and low energy consumption while maintaining acceptable levels of accuracy.*

In this chapter, I extend the work presented in [88], whose focus was the fixed-point performance evaluation of HOG3D HAR algorithm [69] on FPGAs, by evaluating the performance and energy consumption of HOG3D implementations on FPGAs, GPUs, and CPUs. I have profiled the performance of the different HOG3D stages, namely preprocessing, cell descriptor computation, block descriptor computation and video descriptor

computation. Based on this analysis, along with the supporting experimental data, I have identified the strengths and weaknesses of each accelerator for HOG3D. By combining the strengths of each platform, I propose a high performance heterogeneous implementation that takes advantage of the strengths of both FPGAs and GPUs, thereby achieving a higher throughput as well as a lower energy consumption per frame than either homogeneous implementation.

For the FPGA, I have implemented the HOG3D application on the Micron Wolverine 2000 with a Xilinx Virtex 7 FPGA and 32 GB of local memory [94]. For the GPU, I have implemented the design on the NVIDIA $K20$, $K40$ and $K80$ [102]. These implementations are compared to a multi-threaded software application running on the Intel Xeon-E5520 quad-core CPU. The contributions of my work are [118]:

- A high-throughput GPU implementation of the HOG3D algorithm that achieves $166.8X$ speedup over the CPU one as well as $3.1X$ speedup when compared with the FPGA design. A high-throughput FPGA implementation of the HOG3D algorithm that achieves $53.8X$ speedup over the CPU. Furthermore, while the energy efficiency of the software implementation is well below one frame/joule, the GPU design energy efficiency is 5.4 frames/joule.

- A detailed I/O and computational complexity analysis for each of the four modules I have identified in the HOG3D design. Based on this analysis, along with the experimental measurements of the throughput and energy consumption per platform, I have identified the strengths and weaknesses of both FPGAs and GPUs accelerators.

- I propose and evaluate a heterogeneous design that seamlessly combines both FPGA and GPU platforms in a single system: the video pre-processing is executed on the FPGA and the video descriptor extraction is executed in the GPU. This heterogeneous design demonstrates a $1.3X$ speedup over the GPU and is $1.5X$ more energy efficient than either homogeneous designs when applied on VGA data as opposed to QVGA data as in [88]

3.1 Problem Description

The four stages of the HOG3D algorithm are as follows[1]:

(a) Pre-processing: In this step, the algorithm computes the partial derivatives along the x, y, and t axes

$$\begin{aligned} dx &= p[x+1, y, t] - p[x, y, t] \\ dy &= p[x, y+1, t] - p[x, y, t] \\ dt &= p[x, y, t+1] - p[x, y, t] \end{aligned} \tag{3.1}$$

Next, the algorithm computes the integral of the derivatives

$$v_{\partial x}[x, y, t] = \sum_{y' \leq y} \sum_{x' \leq x} dx[x', y', t] \tag{3.2}$$

$v_{\partial y}[x, y, t]$ and $v_{\partial t}[x, y, t]$ are computed in a similar fashion. Finally, the routine computes the integrals videos

$$iv_{\partial x}[x, y, t] = \sum_{t' \leq t} v_{\partial x}[x, y, t'] \tag{3.3}$$

$iv_{\partial y}[x, y, t]$ and $iv_{\partial t}[x, y, t]$ are computed similarly.

[1]In this work, the terms *features* and *descriptors* are used interchangeably.

(b) Cell Descriptor Computation: HOG3D considers the set of integral videos as a spatiotemporal volume. Volumes are sampled using a 3D block. Blocks are further divided into $r \times r \times r$ cells. In addition, cells are divided into $s \times s \times s$ sub-blocks. For each sub-block, the algorithm computes the mean gradient vector $\bar{\boldsymbol{g}}_b = [\bar{g}_{b\partial x}, \bar{g}_{b\partial y}, \bar{g}_{b\partial t}]^T$. The component $\bar{g}_{b\partial x}$ is computed as

$$\bar{g}_{b\partial x} = J(t+l) - J(t) \tag{3.4}$$

where $J(t) = iv_{\partial x}[x, y, t] + iv_{\partial x}[x + w, y + h, t] - iv_{\partial x}[x, y + h, t] - iv_{\partial x}[x + w, y, t]$. Here, w, h and l are implementation parameters. Similar equations are used to compute $\bar{g}_{b\partial y}$ and $\bar{g}_{b\partial t}$. Subsequently, the algorithm quantizes each vector $\bar{\boldsymbol{g}}_b$ using a regular icosahedron. To quantize $\bar{\boldsymbol{g}}_b$, the routine centers the icosahedron at its origin in a three dimensional space. Let $\mathbf{P}_{k,3}$ be the matrix where each row contains the icosahedron coordinates of the central point of face i

$$\mathbf{P}_{k\times 3} = \begin{bmatrix} p_{10} & p_{11} & p_{12} \\ p_{20} & p_{21} & p_{22} \\ \dots & \dots & \dots\dots \\ p_{k0} & p_{k1} & p_{k2} \end{bmatrix}$$

HOG3D calculates the normalized quantization vector $\hat{\boldsymbol{g}}_b$ by computing

$$\hat{\boldsymbol{g}}_b = \frac{P \times \bar{\boldsymbol{g}}_b}{||\bar{\boldsymbol{g}}_b||_2} \tag{3.5}$$

Next, the algorithm thresholds the elements of vector $\hat{\boldsymbol{g}}_b$ using a given parameter α_1. If $\hat{\boldsymbol{g}}'_b$ is the resulting vector after the threshold operation (if $\hat{g}_b[j] < \alpha_1$ then $\hat{g'}_b[j] = 0$

else $\hat{g'}_b[j] = \alpha_1 - \hat{g}_b[j]$) then, the routine uses a scaling factor to obtain the sub-block descriptor

$$\boldsymbol{g}_b = \frac{||\bar{\boldsymbol{g}}_b||_2}{||\hat{\boldsymbol{g}}'_b||_2}\hat{\boldsymbol{q}}'_b \tag{3.6}$$

Then, HOG3D computes the vector $\boldsymbol{c'}$ by adding, element by element, the $s \times s \times s$ sub-block descriptors inside the cell

$$c'[j] = \sum_{i=0}^{s\times s\times s-1} g_{bi}[j] \quad j = 0, .., k-1 \tag{3.7}$$

In addition, the routine normalizes $\boldsymbol{c'}$. The resulting vector is the cell descriptor

$$\boldsymbol{c} = \frac{\boldsymbol{c}'}{||\boldsymbol{c}'||_2} \tag{3.8}$$

(c) Block Descriptor Computation: HOG3D calculates the block descriptor $\boldsymbol{h}$ by concatenating the cell descriptors inside the block

$$\boldsymbol{h} = \{\boldsymbol{c}_{r\times r\times r-1}, .., \boldsymbol{c}_1, \boldsymbol{c}_0\} \tag{3.9}$$

Here $\boldsymbol{h} \in R^d$ a d-dimensional space. The result of this step is a set of block descriptors $H = \{\boldsymbol{h}_i\}_{i=0..n-1}$.

(d) Video Descriptor Computation: Because the number of descriptors changes from video to video, a technique for aggregating varying size descriptors into a fixed-size descriptor has to be implemented [26]. In here, a vocabulary $\mathbf{D} = \{\boldsymbol{d}_j\}_{j=0..m-1}$ with $\boldsymbol{d}_j \in R^d$ is given. To compute fixed-size descriptors, HOG3D computes the distances between each block descriptor $\boldsymbol{h}_i$ and each visual word $\boldsymbol{d}_j$. Next, the algorithm increments by one the histogram slot of the visual word $\boldsymbol{d}_j$, i.e. $x[j]$, that is closest to $\boldsymbol{h}_i$. The resulting histogram

$\boldsymbol{x} \in R^m$ is used as the video descriptor. Finally, the routine uses the video descriptor $\boldsymbol{x}$ as input of a classifier. Notice that $J(t+l) - J(t)$ is the sum of the pixels between t and $t+l$, excluding t, in the area of rectangle (x, y, w, h). In my design, l is always two, and as a result, the design gets simplified. First, the integral video images are computed between adjacent integral images only

$$iv_{\partial x}[x, y, t] = v_{\partial x}[x, y, t'] + v_{\partial x}[x, y, t' - 1] \tag{3.10}$$

Second, because the computation of $J(t+l) - J(t)$ excludes t, $\bar{g}_{b\partial x}$ is computed as

$$\bar{g}_{b\partial x} = J(t+l) \tag{3.11}$$

Further, $\bar{g}_{b\partial y}$ and $\bar{g}_{b\partial t}$ are computed similarly.

As shown above, the implementation of the HOG3D algorithm requires the normalization of a number of low dimensional vectors, see (3.5), (3.6), and (3.8). As a result, the Euclidean norm has to be computed. To optimize hardware resources, the Euclidean norm can be approximated as proposed in [116]

$$\|\mathbf{u}\|_2 \approx (1-\lambda) Max(|u[i]|_{i=0,..,p-1}) + \lambda \sum_{i=0}^{p-1} |u[i]| \tag{3.12}$$

with $\lambda < 1$. As shown in this equation, this method is inexpensive to implement in hardware as it does not require the implementation of the resource-hungry square root operation.

3.2 Fixed-Point HOG3D HAR

In this section, I report on the evaluation of the fixed-point HOG3D recognition accuracy using four benchmarks:

- The KTH benchmark is a collection of 599 videos with six actions [123].
- The UCF11 benchmark is a collection of 1,600 videos with 11 action categories including basketball shooting, horseback riding, swinging among others [82, 83].
- The UCF50 benchmark is a collection of 6,680 videos with 50 actions [115]. This benchmark includes all the actions in UCF11, plus 39 additional actions. As with UCF11, this dataset is challenging due to its diverse conditions as well as the number of actions.
- The UCF101 benchmark [135], a collection of 101 human actions containing 13,320 video clips. This benchmark extends the UCF50 by adding additional actions. This benchmark is particularly challenging due to the diverse set of conditions including illumination, viewpoint, scale, camera motion, backgrounds, etc.

My fixed-point HOG3D implementation is based in the double-precision floating-point implementation described in [69]. Starting from this source code, I added dense sampling, fixed-point arithmetic, and half-precision floating arithmetic. To sample the input video, my routine uses a 3D block. The overlapping between adjacent blocks is 50%. While my routine keeps the temporal scale fix, the spatial scale is increased by a factor of $\sqrt{2}$ until the size of the block is larger than the size of the image. The algorithm divides each 3D block into 64 cells, four cells per dimension. Furthermore it divides each 3D cell into eight sub-blocks, two sub-blocks per dimension. As a result, the size of the block descriptor is 640 elements: $64 = 4 \times 4 \times 4$ cell descriptors and ten elements per cell descriptor when the algorithm uses half of the icosahedron orientations. For each of the HOG3D stages, I set the input bit-width as well as the output bit-width; if m is the number of integer bits and

n is the number of fractional bits, the total bit-width is $m + n$. To minimize overflows and underflows, the operands have been normalized whenever possible. The maximum bit-width is set to 27 bits and the minimum to eight bits. For further details refer to my previous work [88].

To evaluate the accuracy of my HAR recognition method, my algorithm uses reduced fixed-point arithmetic along with a modified version of the SVM library LIBSVM [19]. Here, I added a χ^2 kernel. In addition, my algorithm observes the experimental settings described in [69]. In particular, I use leave-one-group-out cross validation. Since the videos in every dataset are grouped, said N groups, I train a SVM with $N - 1$ groups and make predictions about the videos in the left-out-group. If the left-out-group has k videos and p predictions are correct, the recognition accuracy is p/k. I repeat this process for all the groups and report the average recognition accuracy for both floating point and fixed-point precision.

The results are shown in Figure 3.1. The $'half'$ and $'single'$ results are from my modified HOG3D implementation working in half and single precision floating point. The $'fxp27'$ down to $'fxp8'$ results correspond to the fixed-point HOG3D implementation when working with 27 down to eight bits. For the UCF101 benchmark, I only report the recognition accuracy for single precision floating point and for $'fxp27'$, $'fxp16'$, and $'fxp8'$ fixed-point precision.

The accuracy of the original double-precision floating point implementation [69], and my $'single'$ precision floating point implementation are comparable for all the benchmarks [88]. Moreover, the recognition accuracy for the KTH benchmark is high, it decreases

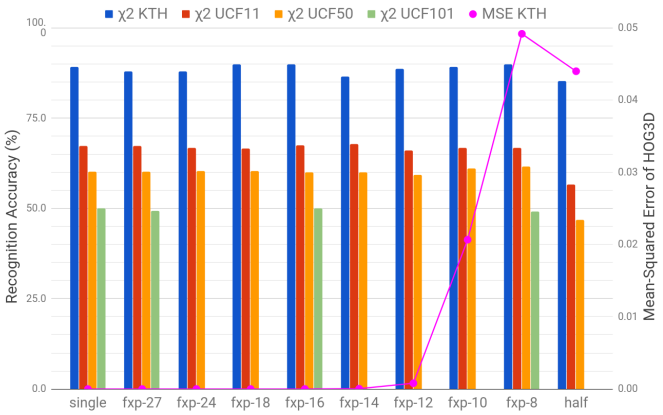


Figure 3.1: On the left, the reduced fixed-point recognition accuracy using χ^2 kernel versus bit-width. On the right, the mean-squared error (MSE) of the video descriptors for the KTH dataset.

for the UCF101 benchmark for all fixed-point precisions. This is consistent with the fact that the UCF101 is the hardest benchmark to recognize. The recognition accuracy behavior is significant for reduced fixed-point arithmetic. As show in the figure, as the bit-width decreases from 27 bits to eight bits, the recognition accuracy is comparable to that of the single precision floating point albeit small fluctuations. The half-precision implementation has the lowest overall recognition accuracy. This behavior is mostly due to the characteristics of the range and the precision of half-precision floating point numbers. In the case of reduced fixed-point arithmetic, the range and precision are dynamic; they change from stage to stage while the range and precision of the half-precision floats remain static [88].

Moreover, I compute the mean-squared error (MSE) by comparing the values of the fixed-point video descriptor with those of the double-precision video descriptors, the ground truth. In Figure 3.1, I only report the MSE of the KTH dataset because it has the largest value. As shown in the plot, the MSE is well below 1×10^{-2} for twelve bits and above. For ten bits and eight bits, the MSE increases, although it always remains

below 5×10^{-2}. In brief, these results show it is feasible to implement HOG3D in reduced fixed-point arithmetic without compromising its accuracy.

3.3 FPGA Implementation

In this section, I describe the implementation of HOG3D in FPGAs. In this design, all arithmetic operations use reduced fixed-point operands. Operations such as multiplications and divisions have been implemented in Xilinx fixed-point cores [37]. When the result of an arithmetic operation overflows the result is saturated on-the-fly. Due to the design of the DSP units in the Virtex-7 FPGAs and to minimize logic usage, the result of operations including multiplications and divisions are always truncated [52].

The input of the algorithm are streams of gray-scale videos consisting of 97 images. The output is the video descriptor vector $\boldsymbol{x}$ with 1000 elements. Unless otherwise described, four videos are moved from the CPU to the FPGA off-chip memory for processing. Then four engines process each video. The description of each engine is given in what follows.

3.3.1 Pre-processing Engine

This engine is responsible for computing the integral videos. Figure 3.2 shows the modules responsible for computing the integral videos along the x, y, and t axes. In this design, all communications between modules are implemented via FIFOs [41]. As shown in Figure 3.2, the computation of the integral videos is straightforward. The *read image* module reads three images at a time from off-chip memory, the images at indexes t, $t+1$, and $t+2$, in a row by row fashion. Next, the *gradients* module computes the derivatives

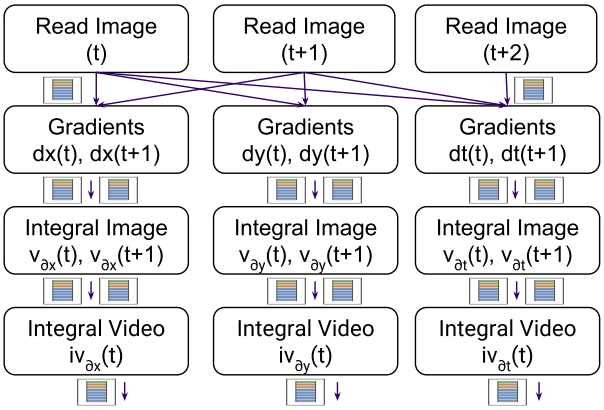


Figure 3.2: Pre-processing engine: Four modules are responsible for the computation of the integral videos along the x, y, and t axis.

of the input pixels along the x, y, and t axes. The *integral image* module computes the integrals of the gradients. To do so, for each input array, it calculates the integral of the current row in a register. Also, this module maintains an on-chip copy of the integral of the previous row. By adding these two integrals, this module obtains an integral image. The *integral video* module takes as input two integral images per axis, adds them together, and writes the integral video into a FIFO. Finally, the resulting integral videos are written to the off-chip memory.

Notice that in my design, the use of FIFOs facilitates the communication between modules as well as the modularization of the design. Each module reads from inputs FIFOS, execute the required computations, and write results to the output FIFOs. In summary, this design reads 97 gray-scale images per video and outputs $144 = 48 \times 3$ pairwise integral videos with two bytes per element. On-chip computations are performed in reduced fixed-point arithmetic with either 8 or 16 bits operands. To improve throughput, I replicate this engine eight times. As a result, this engine can process eight videos in parallel.

3.3.2 Cell Descriptor Engine

Figure 3.3 shows the modules responsible for the computation of the cell descriptors. The *read integral videos* module reads the integral videos from off-chip memory as

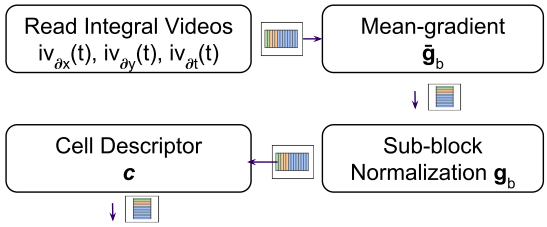


Figure 3.3: Components of the cell descriptor engine.

required in the computation of the mean gradient vector $\bar{\boldsymbol{g}}_b$. The *mean-gradient* module computes vector $\bar{\boldsymbol{g}}_b$ along with its norm. The *sub-block normalization* module computes the sub-block descriptor $\boldsymbol{g}_b$ by executing the matrix vector multiplication $\mathbf{P}\times\bar{\boldsymbol{g}}_b$. Matrix $\mathbf{P}_{10\times3}$ is stored on-chip.

The *cell descriptor* module computes the normalized vector $\boldsymbol{c}_j$. Since the computation of the sub-block descriptors proceeds in a cell by cell order, each time a sub-block descriptor is computed, the unnormalized cell descriptor is updated as described in (3.7). Next, this module computes the normalized cell descriptor and writes the results into a FIFO. Finally, the normalized cell descriptors are written to the off-chip memory.

To take advantage of the FPGA resources, I replicate this engine four times. Hence, my design processes four videos in parallel. For each incoming integral video and for each cell descriptor, this engine computes two sub-block descriptors in parallel. Furthermore, parallel calculations have been implemented when feasible. In the case of the operation $\mathbf{P}\times\bar{\boldsymbol{g}}_b$, thirty multiplications are executed in parallel. In the case of vector normalizations,

divisions and multiplications are executed in parallel as well. In brief, for each incoming video, this engine reads the pairwise integral videos, computes the sub-block descriptors, and outputs the normalized cell descriptors vectors. While the inputs to this module are 2-bytes arrays, the outputs are ten-element vectors. On-chip operations are executed in 16 bits. After vector normalizations, the width of the elements in the output vector reduces to eight bits.

3.3.3 Block Descriptor Engine

This engine reads the normalized cell descriptors, computes the block descriptors, transposes the block descriptors, and writes the results to the off-chip memory. This engine is composed of three modules, as shown in Figure 3.4.

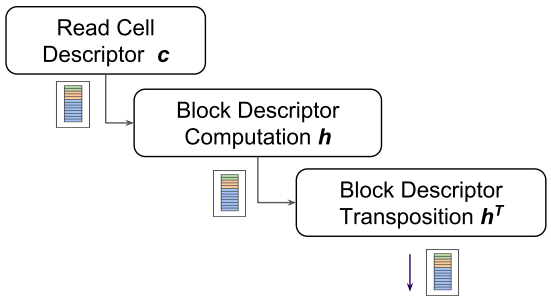


Figure 3.4: Components of the block descriptor engine.

The *read cell descriptor* module reads the normalized cell descriptors $\boldsymbol{c}_j$ from the off-ship memory. Next, the *block descriptor* module concatenates sixty-four cell descriptors and writes the resulting vector, $\boldsymbol{h}$ size 64×10, into a FIFO. Then, the *transposition* module transposes the descriptors. In this process, the output of the *block descriptor* module is written into eight FIFOs, with each FIFO containing one block descriptor. Finally, this

module pops the eight FIFOs and writes the results of the off-chip memory, one column per FIFO, i.e. the transpose operation.

To gain performance, I replicate this engine four times, and as a result, four videos are processed in parallel. For each incoming video and for each block descriptor, this engine reads and writes four cell descriptors in parallel. In short, the input of this module is an array of cell descriptors and the output is an array of block descriptors. All operands in this module are one-byte wide.

3.3.4 Video Descriptor Engine

The next step is to compute the video descriptors. This engine takes two inputs; the first input is the set of block descriptors $\mathbf{H}$, and the second input is a set of pre-computed cluster centers $\mathbf{D}$. The goal of this module is to find for each element in $\mathbf{H}$ the nearest neighbor in $\mathbf{D}$, and finally, to find the distribution of the block descriptors per each given center. In this design, the set $\mathbf{D}$ is mapped to the reference matrix $\mathbf{R}_{m\times d}$, and the set $\mathbf{H}$ is mapped to the query matrix $\mathbf{Q}_{d\times n}$. As a result, nearest neighbor problem can be formulated as a matrix multiplication problem i.e. $\mathbf{C}_{m\times n} = \mathbf{R} \times \mathbf{Q}$ with $c[i,j] = \sum_{k=0,..,d-1}(r[i,k] - q[k,j])^2$. Thus, matrix $\mathbf{C}$ contains all the distances between the given m centers and the n query points.

Matrix multiplications on FPGAs has been studied extensively [160, 35]. In this work, I have followed the directions of the design proposed in [72] with modifications. The computation of matrix $\mathbf{C}$ is blocked. For illustration purposes, let us assume that the size of every block $\mathbf{C}_{ij}$ is $p \times p$, moreover, that $m = k * p$ and $n = s * p$. Matrix $\mathbf{C}$ can be written

as

$$\mathbf{C}_{m\times n} = \begin{bmatrix} C_{10} & C_{11} & ... & C_{1s} \\ C_{20} & C_{21} & ... & C_{2s} \\ ... & ... & ... & ... \\ C_{k0} & ... & ... & C_{ks} \end{bmatrix}$$

The computation of sub-matrices $\mathbf{C}_{ij}$ is from top to bottom and from left to right. In this work, matrix $\mathbf{R}$ has $n \times 640$ elements. The reference centers have been normalized off-line. The components of this engine are shown in Figure 3.5.

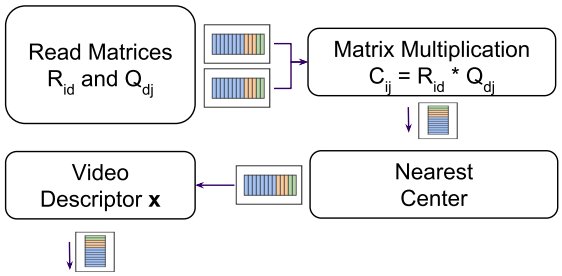


Figure 3.5: Components of the video descriptor engine.

The *read matrices* module is responsible for reading the columns of sub-matrix $\mathbf{R}_{id}$ one at the time into a FIFO, in addition, the rows of sub-matrix $\mathbf{Q}_{dj}$ one at the time into p FIFOs. The *matrix multiplication* module executes the multiplication $\mathbf{C}_{ij} = \mathbf{R}_{id}\mathbf{Q}_{dj}$. The layout of this component is shown in Figure 3.6. At the beginning, this module reads the first element in the FIFO containing $r[0,0]$ and it also reads the p FIFOs containing $q[0,0], q[0,1], .., q[0,p-1]$. Next, p subtraction-and-multiplications are executed in parallel. The results are stored in p BRAM accumulators with each accumulator having p addresses and 16 bits per address. This process continues until the last element, $r[p-1,0]$, of the first

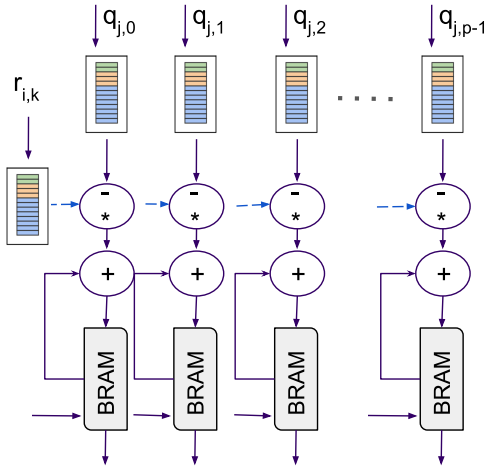


Figure 3.6: Matrix multiplication component. The top p FIFOs contain the elements rows of matrix $\mathbf{Q}$. The left-most FIFO contains the elements of the columns of matrix $\mathbf{R}$. In the center, the distances are computed and accumulated.

column in $\mathbf{R}_{id}$ is multiplied by the current row $q[0,0], q[0,1], .., q[0,p-1]$. Next, this module executes the outer-product between the elements in the second column of $\mathbf{R}_{id}$ and the second row of $\mathbf{Q}_{dj}$. This module continues to execute outer-products until the calculation of the sub-matrix $\mathbf{C}_{ij}$ is complete. Results are written into p FIFOs. Also, by incrementing the number of operation executed in parallel, the parameter p, I can take advantage of the DSPs present in the FPGA: the larger is p, the greater the performance.

The *nearest center* module finds the nearest center for every object in $\mathbf{Q}$. Each time a sub-matrix $\mathbf{C}_{ij}$ is computed, this module reads from p FIFOs. For every FIFO, i.e. for every object $\boldsymbol{q}_j \in \mathbf{Q}$, this module keeps track of the minimal distance and the associated center thus far. Since the sub-matrices $\mathbf{C}_{ij}$ are computed top-to-bottom and left-to-right, each time that a bottom sub-block is computed i.e. $\mathbf{C}_{k0}, \mathbf{C}_{k1}, .., \mathbf{C}_{ks}$, this module outputs the centers associated with $\boldsymbol{q}_j$. These centers are written to p FIFOs.

The *video descriptor* module reads the outputs of the previous module. Every time that this module reads a center, the BRAM memory address associated with that center is incremented by one. When all the nearest centers are found, this module outputs the video descriptor vector $\boldsymbol{x}$, to the off-chip memory. Notice that the computation of vector $\boldsymbol{x}$ can potentially harm the throughput as the *nearest center* module outputs as many as 320 centers per cycle. To speed up the computation of $\boldsymbol{x}$, I use a reduction tree with ten nodes at the top level. Each of those node computes local video descriptor by processing 32 inputs. In the next level of the tree, the local video descriptors are merged into pairs. This reduction continues until the final video descriptor is found. To improve throughput, I replicate this engine four times such that four videos are processed in parallel. For each engine, the parameter p has been steadily increased until the resources in the FPGA are nearly exhausted. In this design, I set p to 320 such that 1,280 multiplications are executed in parallel. While all input elements are one byte, the output elements are two bytes. On-chip computations are executed in two bytes.

3.4 GPU Implementation

In this section, I describe the implementation of HOG3D in GPUs. I use 32-bit integer arithmetic and single precision floating point arithmetic. My implementation processes eight videos in parallel by taking advantage of CUDA streams [102, 105]. In this scenario, each stream is responsible for processing one video. Moreover, kernel calls are issued in a breadth-first fashion across all the running streams. For the purpose of illustration, let us assume that eight CUDA streams $S1$,..,$S8$ are running in parallel and

each stream has two kernels $K1$ and $K2$. The GPU executes kernel K_1 on all eight streams: $S1(K_1), .., S8(K_1)$ followed by $S1(K_2), .., S8(K_2)$. In my design, the GPU executes eight streams. For each stream, one video having 97 gray-scale images is transferred from the host main memory to the GPU off-chip memory. Eventually, the HAR algorithm is executed using four engines as described below.

3.4.1 Pre-processing Engine

The *pre-processing* engine computes the pairwise integral videos along the x, y, and t axis. Figure 3.7 shows the kernels used in this engine. The *image gradients* kernel

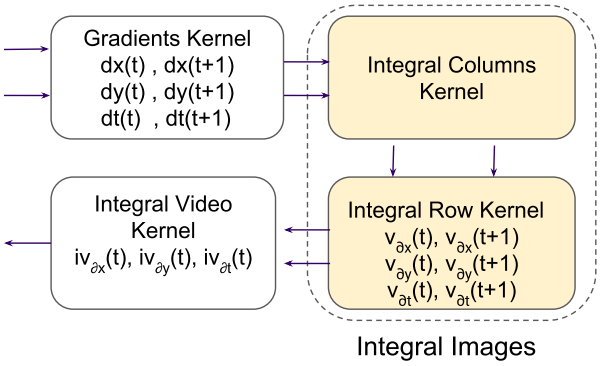


Figure 3.7: Components of the pre-processing engine.

computes the gradients along the x, y, and t axis. In this step, each image is divided in tiles of size 16×16. The partition of an image into tiles facilitates coalesced I/O operations. Each tile is then loaded into shared memory along with the halo elements. For every tile, the kernel computes the gradients along the x, y and t axis.

Subsequently, the *integral images* kernel, the right hand side of Figure 3.7, reads the gradients and computes the integrals of the images. Calculating the integral of the

images is challenging for GPUs due to the presence of thread divergences [124, 11]. My design is similar to the work presented in [11], with modifications.

In the first step, the *integral columns* kernel reads the gradients, using one CUDA thread per column, integrates the values of the columns, and writes the resulting integrals to the off-chip memory. While the first step can be executed efficiently by one CUDA thread, the computation of integrals along the rows requires synchronization between the threads in a CUDA block. In the second step, the *integral row* kernel reads the computed column integrals into shared memory. The algorithm computes the row integrals in two phases: the up-sweep phase and the down-sweep phase. In the up-sweep phase, the kernel computes the prefix-sum for all odd elements. In the down-sweep phase, the kernel computes the prefix-sum for all even elements. In my work, because the rows of the input images are no larger than 640, the prefix-sum per row can be implemented in shared memory; the routine sets the row size to $N = 1024$ and pads the data as necessary. After padding, the kernel uses $N/2$ threads, takes $2Log_2(N) - 1$ steps, and executes $2(N - 1)$ additions [11]. The resulting array integrals are then written to the off-chip memory.

Finally, the *integral video* kernel reads two integral images per axis, adds their values, and writes the results back to the off-chip memory. In this kernel, threads are mapped to the columns such that coalesced memory accesses is achieved.

3.4.2 Cell Descriptor Engine

In this engine, the algorithm computes the cell descriptors $\boldsymbol{c}_j$. Figure 3.8 shows the kernels involved in this computation.

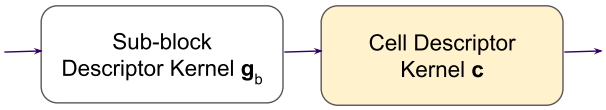


Figure 3.8: Cell descriptor engine. Two kernels are responsible for the computation of the cell descriptors.

The*sub-block descriptor* kernel computes the mean-gradient vector $\boldsymbol{g}_b$. In this design, a thread is responsible for computing the mean-gradient. To improve the performance, the matrix $\mathbf{P}_{10\times3}$ is stored in constant memory and the mean-gradient is stored in shared memory. The *cell descriptor* kernel computes the vector $\boldsymbol{c}$. Specifically, a thread reads the normalized sub-block descriptors inside the cell, adds their values, executes vector normalization, and finally, writes the resulting vector to the off-chip memory. In this design, a thread is responsible for computing the normalized cell descriptors using shared memory.

3.4.3 Block Descriptor Engine

The *block descriptor* engine computes the block descriptors using two kernels as shown in Figure 3.9. The *block descriptor* kernel reads the cell descriptors $\boldsymbol{c}$ and computes

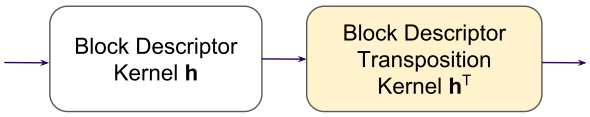


Figure 3.9: Block descriptor engine. Two kernels are responsible for computing the block descriptors.

the block descriptor $\boldsymbol{h}$. Results are written to the off-chip memory. Because this kernel is I/O bounded, its performance is improved by increasing the number of threads executing off-chip reads and writes. In this work, a thread is responsible for reading and for writing

each element in $\boldsymbol{h}$. The *block transposition* kernel reads, transposes, and writes an array size $640 \times 10,240$. Matrix transposition using tiles is a well-studied kernel [104, 119, 11].

3.4.4 Video Descriptor Engine

In this engine, the video descriptor is computed via a nearest neighbor clustering algorithm. Given two vectors $\boldsymbol{x} \in R^d$ and $\boldsymbol{y} \in R^d$, their Euclidean distance is given by

$$\rho(\boldsymbol{x}, \boldsymbol{y})^2 = (\boldsymbol{x} - \boldsymbol{y})^T(\boldsymbol{x} - \boldsymbol{y}) = ||\boldsymbol{x}||^2 + ||\boldsymbol{y}||^2 - 2\boldsymbol{x}^T\boldsymbol{y} \tag{3.13}$$

Furthermore, distances between vectors can be computed via matrices [38]. Let $\mathbf{R}$ and $\mathbf{Q}$ be two matrices size $d \times m$ and $d \times n$ respectively. $\mathbf{R}$ represents m reference centers and $\mathbf{Q}$ represents n block descriptors. Let $\boldsymbol{\rho}^2(R, Q)$ be a $m \times n$ matrix containing the distances between the reference centers and the block descriptors. Then $\boldsymbol{\rho}^2(R, Q)$ can be computed as

$$\boldsymbol{\rho}^2(R, Q) = \mathbf{N}_R + \mathbf{N}_Q - 2\mathbf{R}^T\mathbf{Q} \tag{3.14}$$

In this equation, the elements of the j^{th} row of $\mathbf{N}_R$ are all equal to $\sum_{i=0}^{i=d-1} (R[i, j])^2$. The elements of the j^{th} column of $\mathbf{N}_Q$ are all equal to $\sum_{i=0}^{i=d-1} (Q[i, j])^2$. To save memory, in this design, I represent $\mathbf{N}_R$ and $\mathbf{N}_Q$ as vectors. Further, because the reference centers are predefined, $\mathbf{N}_R$ and $\mathbf{R}^T$ are computed off-line. Figure 3.10 shows the kernels involved in the computation of the video descriptors.

The *norms* kernel computes vector $\boldsymbol{N}_Q$. In this design, a CUDA thread is responsible for computing $N_Q[j]$. This assignment makes it possible to optimize off-chip memory bandwidth. The *matrix multiplication* kernel executes $\mathbf{R}^T\mathbf{Q}$ by means of the CUBLAS

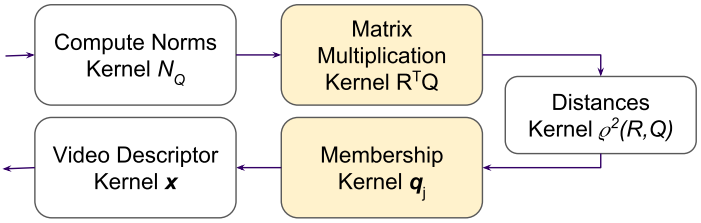


Figure 3.10: Video descriptor engine. Kernels used in the process of computing the video descriptors.

library [104]. The *distances* kernel calculates the matrix $\boldsymbol{\rho}^2(R,Q)$. In this work, a CUDA thread is responsible for computing the distances between the reference centers and the block descriptors i.e. $\rho^2(R,Q)[i,j]$. To do so, it reads elements $N_R[i]$ and $N_Q[j]$ along with element $\rho^2(R,Q)[i,j]$. Next, it computes the distance, as shown in (3.14), and writes the values to off-chip memory.

The *membership* kernel finds the membership of every object in $\mathbf{Q}$. To optimize the bandwidth, a CUDA thread is responsible for computing the membership of $\boldsymbol{q}_j$ by means of scanning column j in matrix $\boldsymbol{\rho}^2(R,Q)$. Results are written to the off-chip membership vector. Finally, the *video descriptor* kernel computes vector $\boldsymbol{x}$ in two steps. In the first step, the privatized step, threads read the elements of the membership vector. As the coalesced reads are executed, the video descriptor is computed, in shared memory, by means of atomic adds. In the second step, the global step, threads write atomically to the shared video descriptor vector in the off-chip memory [121].

3.5 Complexity Analysis

In this section, the complexity of the FPGA and the GPU design per engine is given. I assume the HOG3D algorithm takes as inputs 97 gray-scaled images having M

rows and N columns. Also, I assume that each sub-block descriptor contains K elements, each cell descriptor contains eight sub-block descriptors, and each block descriptor contains 64 cell descriptors. Moreover, the design processes C cell descriptors, n block descriptors, and m reference centers. Each block descriptor and each reference center vector has $64 \times K$ elements. Moreover, the matrix multiplication $\mathbf{R}^T \times \mathbf{Q}$ operation is blocked, and the size of the block is p. Without loss of generality, I assume that m/p and n/p are integers.

3.5.1 Pre-processing Engine

Table 3.1 shows the results of the complexity analysis.

Table 3.1: Pre-processing engine complexity analysis

I/O operations	FPGA	$MN(97 + 144)$
	GPU	$MN(97 + 144 + 144 \times 8)$
Arithmetic Operations	FPGA	$144(7MN - 4M - 2N)$
	GPU	$144(9MN - 6M - 2N)$

In FPGAs, the pre-processing engine requires the reading of 97 images and the writing of 144 integral video images, 48 for each dimension. In GPUs, the pre-processing engine requires $8MN$ additional I/O operations per integral video image: $2MN$ writes due to computation of the gradients, $2MN$ reads and $2MN$ writes due to the computation of the integral of the derivatives, and $2MN$ reads due to the computation of the integral video image.

In FPGAs the computation of the integral videos is executed in three steps. (a) The computation of the derivatives per input image takes $M(N-1)$ operations. (b) The integral of the derivatives, takes $2MN - M - N$ operations. Hence, the computation of

the integrals of the gradients for two images requires $2(3MN - 2M - N)$ operations. (c) The computation of the pairwise integral videos requires MN additions. In GPUs, steps (a) and (c) are as in the case of FPGAs. Step (b) takes $3MN - 2M - N$ operations: the integrals along the columns require $(M-1)N$ operations and integrals along the rows require $2M(N-1)$ operations. Hence, the computation of the integrals of the gradients for two images requires $2(4MN - 3M - N)$ operations.

Taking into consideration the complexity analysis and the proposed design for FPGAs and GPUs, I observe that this engine is I/O bounded. In the case of FPGAs, the bound applies despite the larger number of arithmetic operations. This engine computes three integral videos in parallel. In addition, the computation of the gradients, the integral of the gradients, and the pairwise integral videos is pipelined.

3.5.2 Cell Descriptor Engine

The complexity analysis is given in Table 3.2.

Table 3.2: Cell descriptor complexity analysis

I/O operations	FPGA	$C(8(12) + K)$
	GPU	$C(8(12) + 8K + 9K)$
Arithmetic Operations	FPGA	$C(8(11K) + 11K + c)$
	GPU	$C(8(11K) + 11K + c)$

In FPGAs, the computation of one sub-block descriptor takes twelve reads, see (3.11), and the computation of the cell descriptor takes K writes. In GPUs, the computation of each sub-block descriptor takes K additional writes and the computation of the cell descriptors takes $8K$ additional reads. In FPGAs, as well as GPUs, the computation of the

sub-block descriptor $\boldsymbol{q}_b$ takes $11K$ operations[2]. The computation of the cell descriptor $\boldsymbol{c_j}$ requires $11K$ operations[3].

The computation of the cell descriptors in FPGAs is I/O bounded despite the larger number of arithmetic operations. While the arithmetic operations are parallelized and pipelined, at least K operations are executed in parallel per pipeline, the reading of the integral videos is serial. Moreover, in GPUs, the performance of this engine is bounded by the number of arithmetic operations as a CUDA thread is responsible for the computation of each sub-block and cell descriptor.

3.5.3 Block Descriptor Engine

In FPGAs, the computation of each block descriptor involves the reading and writing of 64 cell descriptors i.e. $2n(64C) = 2n(d)$ I/O operations. In GPUs, additional $2nd$ I/O operations due to transpositions must be executed. For both FPGAs and GPUs, this engine is I/O bounded.

3.5.4 Video Descriptor Engine

Table 3.3 shows the results of the complexity analysis.

In the FPGA, the computation of each sub-matrix $\mathbf{C}_{ij}$ size $p \times p$ takes $2dp$ reads. Moreover, writing the video descriptor vector $\boldsymbol{x}$ takes m writes. Computing each element $c[i, j]$ takes d subtractions, d multiplications, and $d-1$ additions. As a result, the total number of operations per sub-matrix $\mathbf{C}_{ij}$ is $(3d-1)(p^2)$. Finding the membership of each block

[2] $5K$ operations due to $P \times \bar{\boldsymbol{g}}_b$, K divisions, K comparisons, $3K$ operations due to squared roots, plus K multiplications.

[3] $7K$ operations are due to additions and $4K$ operations are due to normalizations. The constant c accounts for few additional operations.

Table 3.3: Video descriptor complexity analysis

I/O operations	FPGA	$2(dmn/p) + m$
I/O operations	GPU	$2(dmn/p) + mn + n(d+2) + 2mn + mn + n$
Arithmetic Operations	FPGA	$(3d-1)(mn) + nm$
Arithmetic Operations	GPU	$(2d-1)mn + 3mn + n(2d-1) + (m-1)n + n$

descriptor takes $(m-1)$ comparisons. Moreover, the computation of the video descriptor vector takes n additions.

The computation of the video descriptor in GPUs is described in three steps. First, the computation of matrix $\mathbf{R}^T\mathbf{Q}$ takes $2pd*(mn/p^2)$ reads, mn writes, dmn multiplications, and $(d-1)mn$ additions. Second, the computation of $\boldsymbol{\rho}^2(R,Q)$ is executed in two parts. (a) Computing $\mathbf{N}_Q$ requires dn reads, n writes, dn multiplications and $(d-1)n$ additions. (b) Computing $\boldsymbol{\rho}^2(R,Q)$ requires n reads and the reading and writing of a matrix size mn. Moreover mn multiplications and $2mn$ additions are required. Third, the computation of the video descriptor $\boldsymbol{x}$ requires mn reads, n writes, $(m-1)*n$ comparisons, and n additions. Finally, this engine is computed bounded.

3.6 Experimental Results

In this section, I discuss the throughput and the energy efficiency of the HOG3D design in FPGAs and GPUs. At the end, I propose a heterogeneous HOG3D (HHAR) algorithm.

3.6.1 FPGA Synthesis

In this part, I describe the results of the synthesis, placing, and routing. The testbed is composed of two Intel Xeon CPUs E5-2640 and two Virtex-7 FPGAs [94]. Each FPGA has 32 memory channels. To achieve maximum bandwidth per channel, 1.25 GB/s, a 64-byte exclusive request has to be issued. Otherwise, a channel can handle request sizes of 1,2,4, or 8 bytes at the expenses of decreasing the effective bandwidth. While I implemented all the engines in Verilog HDL, the synthesis is executed in Vivado 16.4. First, simulations are executed to attest the accuracy of the results. Next, I addressed timing errors until the design meets the timing requirements (166 MHz). Table 3.4 shows the percentage of utilization of the resources in the FPGA per module.

Table 3.4: FPGA resource utilization percentages per engine. Image size 320×240

Available Resources	Pre-processing (%)	Cell Desc (%)	Block Desc (%)	Video Desc (%)
Registers (2443K)	15.49	17.33	14.26	21.22
LUTs (1221K)	20.91	24.79	17.96	25.06
LUTRam (344K)	19.87	16.80	16.41	25.20
Block Rams (1.2K)	40.21	42.52	39.36	87.41
DSPs (2.1K)	0.00	20.00	1.48	59.44
Memory Channels (32)	100.00	100.00	100.00	75.00

To analyze the resource utilization per engine, I split the engines into two groups taking as the dividing factor the utilization of memory channels. Group one, the pre-processing, the cell descriptor and block descriptor engine, uses 100% of the memory channels while the second group, the video descriptor engine, uses 75% only. By inspection of the resource utilization table, I notice that engines in the first group have high I/O utilization and lower on-chip resource utilization whereas engines in the second group have high

on-chip resource utilization and low I/O utilization. In other words, the number of I/O operations constrains the performance of the first group of engines, whereas the number of arithmetic operations constrains the performance of the second group of engines.

3.6.2 FPGA Throughput

In this section, I describe the performance of the FPGA design. The performance per engine is shown in Table 3.5. In this table, the time to move the data from the host to the FPGA and back is not reported.

Table 3.5: Virtex-7 FPGA throughput per engine. Image Size 320×240

Engine	Videos Processed	Throughput (fps)
Pre-processing	8	11,184
Cell Descriptor	4	3,110
Block Descriptor	4	11,186
Video Descriptor	4	3,036
Overall Throughput (fps)	4	1,088

Table 3.5 shows that the pre-processing and the block descriptor engines have the highest throughput while the other two engines have the lowest. The high performance of the pre-processing engine is the result of two factors. First, it contains eight kernels with each kernel processing three images in parallel. Second, it benefits of the high I/O performance offered by WX-2000 memory system due to data locality during reads and writes [94]. The use of pipelining increases the throughput further. The block descriptor engine has a high performance as well. Notice this engine is fully constrained by the off-chip bandwidth. In this regard, this engine partially benefits from contiguous memory reads

since cell descriptors are represented as 16 contiguous bytes. Writes are always issued in eight-byte chunks.

The cell descriptor engine has the next best performance. The performance of this engine is limited by the sparsity of the off-chip reads. Although the number of I/O operations the engine issues is low, memory requests are issued to non-contiguous memory regions. In addition, because the design uses 100% of the memory channels, further gains in performance by means of increasing the processing pipelines is not feasible. The video descriptor engine has the lowest performance. The performance of this engine is limited by the computational complexity of the matrix multiplication operation, see Table 3.1. Further gains in performance are not feasible as resources have been nearly exhausted, see Table 3.4.

Overall, when working with images size 320×240, the maximum throughput in steady state is 3,036 fps when four FPGAs are used. In steady state engine one processes four videos. Moreover, not considering reconfiguration time, the maximum throughput achieved by one FPGA is 1,088 fps. This calculation accounts for the time it takes for an image to move across each engine.

3.6.3 GPU Throughput

In this section, I analyze the performance of the GPU design. The first testbed consists of an Ubuntu workstation equipped with an Intel I7-860 processor, 8GB of RAM, and a $K20$ GPU. The second testbed consists of an Ubuntu workstation equipped with an Intel Xeon E5-520 processor, 24GB of RAM, and a $K40$ GPU. Finally, the third testbed consists of a CentOS workstation equipped with an Intel Xeon E5-2680 processor, 32GB of RAM and a $K80$ GPU. The code is compiled with the CUDA compiler release 7.5

and the Basic Linear Algebra Subroutines. In all the experiments, the error correction capabilities (ECC) are disabled. Table 3.6 shows the throughput per engine for each GPU. The discussion that follows applies to the $K20$ GPU. Similar analysis applies to the $K40$ and $K80$ GPUs as these devices share the same architecture.

Table 3.6: K20, K40 and K80 throughput per engine when processing eight videos in parallel. Image size 320×240

	K20 (fps)	K40 (fps)	K80 (fps)
Pre-processing	3,310	4,044	5,306
Cell Descriptor	13,241	17,143	23,594
Block Descriptor	118,154	128,000	243,810
Video Descriptor	9,458	11,294	16,203
Overall Throughput(fps)	2,033	2,487	3,370

From the table, I notice the $K20$ is very fast at computing the block descriptors, engine three, and very slow at pre-processing the videos, engine one. Two elements provide insight on the performance of engine three. First, the performance of this engine is purely I/O bounded. Second, the $K20$ off-chip memory bandwidth is high, i.e. 208 GB/s. The engine computing the cell descriptors, has the next best performance. Close inspections of the performance of this engine shows the kernel computing the sub-block(cell) descriptors taking 66(34) % of the running time. The performance of this engine is limited by the amount of work the engine must execute and by the uncoalesced nature of the reads during the computation of the sub-block descriptors. Thread divergences present during vector normalizations limit the performance as well.

The performance of engine four, namely the computation of the video descriptors, has the next best performance. Inspecting the performance of this engine reveals that about 60% of the time is spent executing the matrix multiplication. The remaining time is spent

in nearly equal parts in the kernels responsible for computing Q_N, $\rho(R,Q)^2$ and $\boldsymbol{x}$. In this engine, elements limiting gains in performance include the complexity of the matrix multiplication (see Table 3.3 in Section VII), the use of block barriers, and the use of atomic primitives during the computation of the video descriptor vector $\boldsymbol{x}$.

Notably, the pre-processing engine has the lowest performance. The pre-processing complexity analysis (see Table 3.1 in Section VII), explains in part this behavior. When compared with the FPGA complexity, the GPU executes $MN(144 \times 8)$ additional I/O operations. Moreover, while the number of the arithmetic operations per integral video in FPGAs is proportional to $7MN$, this complexity is proportional to $9MN$ in GPUs. On closer inspection, the GPU running times show that engine one spends 55.0%, 28.9%, 12.2%, and 3.8% computing the row integrals, the columns integrals, the image gradients, and the integral videos respectively. Issues affecting the performance of the row integrals kernel include the presence of *control flow divergences*, the presence of *synchronization primitives*, and the *effective amount of work* that a thread executes per step.

Bialas [10] shows that block thread divergences on Kepler GPUs cost as much as 116 clock cycles. *Letrendre* [79] shows that the extra cost of using block synchronization primitives in the presence of global memory reads ranges from few hundreds up to a thousand cycles. Likewise, the cost of global memory writes in the presence of synchronization primitives is comparable, although it tops at about 700 cycles. The extra cost of using block synchronization primitives in the presence of shared memory reads is near 350 clock cycles. Similarly, when synchronization primitives are used, the cost of shared memory writes is near 220 cycles. Furthermore, the row integral kernel executes $2(N-1)$ additions

in $2Log_2(N) - 1$ steps when $N/2$ threads are used. If $N = 512$, the number of additions per step is $61 \approx 1022/17$. In this case, the amount of work per thread per step is $0.24 \approx 61/256$. In other words, during row integration threads do not execute any useful work 76% of the time.

In addition, I notice that, although recent GPU architectures include novel software and hardware optimizations [23], in my work, those optimizations do not increase the throughout of the row integral kernel notwithstanding the expected gains in performance due to the new architecture. While the single instruction multiple thread (SIMT) execution model supports independent thread scheduling, this model does not increase the performance of the kernel under analysis because synchronization between the collaborating threads during the up-sweep and the down-sweep is still required i.e. in the best scheduling scenario, the integral row kernel still requires $2Log(N) - 1$ steps. Further research reveals that the low performance displayed by the row integral kernel is part of a broader set of performance challenges faced by GPUs when processing workloads with irregularities as shown in [16, 21, 48].

I notice that, although the $K20$ GPU can process fifteen videos in parallel, the gains in performance are diminishing as the number of videos increases. The peak performance is achieved when the number of videos processed is ten. Above ten videos, the performance remains constant. Below seven videos, the throughput drops by 30 fps and below. In brief, when working with images size 320×240, the throughput of the $K20$, $K40$, and $K80$ is 2,033 fps, 2,487 fps, and 3,370 fps respectively. The $K80$ speedup is $1.3X(1.6X)$ when

compared with the $K40(K20)$. The $K80$ implementation takes advantage of the dual GPU design.

3.6.4 Heterogeneous HAR

Based on the throughput results obtained for the $K20$ GPU and the Virtex-7 FPGA, in this section, I develop a heterogeneous HAR (HHAR) design. In this design, the pre-processing is executed in the FPGA. The data is then moved from the FPGA to the host and from the host to the GPU, and finally, the cell, block, and video descriptors are computed in the GPU. Figure 3.11 shows the steps required by my HHAR design. Table 3.7

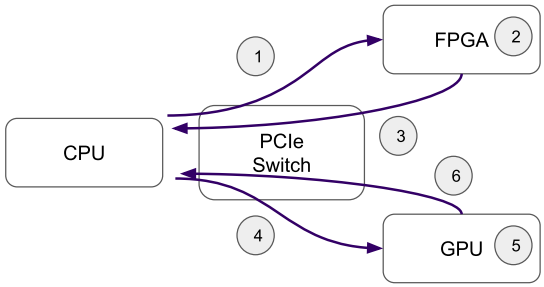


Figure 3.11: Heterogeneous HAR desing. (1) The transferring of data from the host to the FPGA (2) The execution in the FPGA (3) The transferring of data from the FPGA to the CPU (4) The transferring of data from the CPU to the FPGA (5) The execution in the GPU (6) The transferring of data from the GPU to the CPU.

shows the execution times of the steps involved in the algorithm for images size 640×480.

Table 3.7: Heterogeneous HAR design execution times per task. Image size 640×480

Step	Resource	Time (ms)
(1) Host to FPGA Data Transfer	PCIe G3 $\times 16$	65.24
(2) Pre-processing	Virtex 7	274.70
(3) FPGA to Host Data Transfer	PCIe G3 $\times 16$	379.16
(4) Host to GPU Data Transfer	PCIe G2 $\times 16$	225.84
(5) Cell, Block and Video Desc.	K20	591.50

Table 3.7 shows the times it takes to process eight videos in parallel. In this design, $776 = 97 \times 8$ gray-scale images are transferred from the host to the FPGA. Next, the pre-processing engine is executed in the FPGA. The resulting 8×144 2-byte integral videos are then transferred from the FPGA to the host and from the host to the GPU. Finally, the GPU executes the cell, block, and video descriptor engines. The time to transfer the video descriptor back to the CPU, step six, is below one millisecond, and as a result, it is not reported.

By inspection of Table 3.7, I notice that the execution of the block, cell, and video descriptor engines on the GPU takes the longest time followed by the time it takes to transfer data from the FPGA to the host. Moreover, it is possible to overlap the movement of data from the host to the FPGA (and vice versa) with the execution of the kernel in the FPGA; the call *wdm_dispatch* in the Convey Development Kit is non-blocking [93]. Likewise, it is possible to overlap the movement of data between the host and the GPU with the execution of a kernel in the GPU given that several practices are observed [105]. Considering these overlaps, the communication time between the host and the FPGA dominates the FPGA execution time. Similarly, the execution time in the GPU dominates the communication time between the host and the GPU.

Based on these observations, I propose a host controlled four stage pipeline, see Figure 3.12. In this plot, the notation *Bx,Rx* and *B*x, R*x* identifies the set of double buffers used in the FGPA and the GPU. In steady state, reached at step six, the maximum latency of the pipeline is 591.50 ms. This is the time it takes the GPU to process eight videos

Step 1	*Step 2*	*Step 3*	*Step 4*	*Step 5*	*Step 6*
H-FPGA(B1)	FPGA(B1,R1)	FPGA(B2,R2)	**H-GPU(B*1)**	**GPU(B*1,R*1)**	**GPU(B*2,R*2)**
	H-FPGA(B2)	H-FPGA(B1) FPGA-H(R1)	FPGA(B1,R1)	**H-GPU(B*2)**	**H-GPU(B*1)**
			H-FPGA(B2) FPGA-H(R2)	FPGA(B2,R2)	FPGA(B1,R1)
				H-FPGA(B1) FPGA-H(R1)	H-FPGA(B1) FPGA-H(R1)

Figure 3.12: Heterogeneous HAR pipeline. The pipeline has four steps: (a) the transferring of data from the host to the FPGA *(H-FPGA)* and from the FPGA to the host *(FPGA-H)*; (b) the execution in the FPGA *(FPGA)*; (c) the transferring of data between the host and the GPU *(H-GPU)*; and (d) the execution in the GPU *(GPU)*.

in parallel. Based on these considerations, the HAR design has a cumulative throughout of 1,311 fps; 163 fps per input video.

3.6.5 Energy Efficiency Comparison

Next, I compare the energy efficiency of each platform. For the GPUs, the power is measured using the NVIDIA Management Library [105]. Once the power plot is drawn, corrections have been made to have an accurate power estimation [17]. The FPGA power consumption is measured using the Convey Development Kit [93]. This API allows the user to query the power usage of the FPGA as the application is executed. In all platforms, the computed power accounts for the idle power and the dynamic power consumption. The energy usage and energy efficiency per platform and per engine are shown in Table 3.8. Although my heterogeneous HAR design works with any GPU, I report the results with the $K20$ GPU.

Table 3.8: Energy usage (Joules), energy efficiency (Frames/Joule), and throughput (FPS) per platform. Image size 640×480

	HHAR	K20	K40	K80
Pre-processing (J)	17.2	74.0	80.3	125.2
Cell Desc (J)	41.0	14.4	13.8	20.5
Block Desc (J)	4.8	1.7	1.4	2.0
Video Desc (J)	98.9	34.7	35.7	35.6
Total Energy (J)	161.9	124.8	131.2	183.3
Efficiency (F/J)	8.0	3.6	3.9	5.4
Throughput (FPS)	1,304	455	517	998

For the pre-processing stage in the HHAR design, I report the energy measured via the Convey Development Kit and for all other stages, I report the energy measured via the NVIDIA Development Kit. To obtain the throughput and energy efficiency of the pre-processing engine in the FPGA, I have synthesized its design for VGA images (640×480) and measured the throughput and power consumption. The resource usage is shown in Table 3.4. Eight engines processing gray-scale images have a cumulative throughput of 2,796 fps while requiring about 36.8 joules i.e. 13.1 mJ/F.

Moreover, the HHAR energy calculation shown in Table 3.8 does not takes into account the energy used by the host or the PCIe buses. My HHAR design requires additional energy to move $707.8MB$ from the FPGA memory to the host memory and from the host memory to the GPU memory. My research indicates this additional energy is minor compared to the energy used by a kernel running in either the FPGA or the GPU. It is estimated that DDR3 memories dissipate approximately 1.5 W/GBit on average and close to 2.5 W/GBit at peak usage [56, 89]. In the case of memory reads, the reading of 32 bits requires close to 620 pJ [56]. Using these figures, I estimate the energy required for reads and writes $707.8MB$ is below one joule. In addition, my experiments reveal that the transfer of 5.6 GBits from the host to the GPU requires about a dozen Watts, as reported

by the sensor in the GPU, although precise measures of the energy required bit the PCIe links is challenging. More importantly, adding few joules to the energy consumption of my HHAR design will not alter the overall results.

From this, I notice that the HHAR design has the highest throughout, in frames per second (fps), and it is the most energy-efficient design, in frames per joules (F/J), followed by the K80. My HHAR design has a cumulative throughput of 1,311 fps: 163 fps for each incoming video. In addition, it achieves $2.0X(2.2X)$ higher energy efficiency when compared with the K40(K20). The K20 and K40 GPUs have comparable comparable energy efficiency and the $K80$ is more energy efficient by a factor of 1.5 and 1.4 respectively. Notice that if my HHAR design uses the $K40$ or $K80$, instead of the $K20$ GPU, the design will further increase both the energy efficiency and the throughout.

3.6.6 Comparison With Related Works

Prior work on HOG has focused mostly on two dimensions (HOG2D) for object recognition. Instead, I use histogram of gradients in three dimensions (HOG3D), which is particularly important for HAR. Working with the temporal dimension adds to the *complexity* of the algorithm in all its stages.

Previous research of HOG2D for object recognition in GPUs includes the work presented in [53, 113, 81]. When processing images size 640×480, as in this work, these designs achieve throughputs ranging from 16 fps up to 38 fps. While the focus of the work in [53] is the identification of vehicles in real time, the work in [113, 81] focuses in the identification of pedestrians using batch approaches. In addition, these researchers focus

their attention on achieving high throughput and high energy efficiency using well-establish algorithms.

Research of HOG2D for object detection in FPGAs includes [63, 98, 95, 47, 101, 87]. When processing images size 640×480, these designs achieve throughputs ranging from 30 fps up to 526 fps although the work in [47] processes higher resolution images at the expense of lower throughput. As in the case of GPUs, the focus of this work is in achieving high throughput. In addition, lowering the computational complexity of the design without sacrificing the recognition accuracy is paramount.

The acceleration of HOG3D has not received the same attention as that of HOG2D. The work in [57] targets HAR applications in FPGAs although it operates at 600 fps while using images size 320×240. This design has a recognition rate of 93.2% working with a small set of actions. Its recognition drops to 80.8% when a few more actions are added. In comparison, my work achieves a throughput of 1,311 fps on 640×480 images when eight videos streams are processed in parallel. Also, while the work in [1] and [57] target datasets having few classes, my work targets datasets having over 50 classes.

Furthermore, although my work is orthogonal to those focused into improving the accuracy of HAR applications, I state that my HCF design, with multiple scale support, has recognition accuracy comparable to state-of-the-art CNNs. In the case of the HMBD-51[4] [71], the recognition accuracy of CNNs [132] is 59.4% when two-stream CNNs are used. When only the temporal or spatial stream is used, the recognition accuracy drops to 54.6% and 40.5% respectively. When hybrid approaches are used [147], the recognition accuracy

[4]This benchmark is comparable to the UCF50 benchmark. It has 51 action categories and 7,000 video clips

reaches 65.9%. My 16-bits reduced fix-point HHAR design achieves 60.1% recognition accuracy in the UCF50.

Finally, the higher accuracy demonstrated by CNNs on HAR applications [70, 64, 142] comes at the cost of higher power consumption and lower throughput. The results in [139] show that feature extraction using HOG is $311X$ and $13{,}486X$ more energy efficient and has $34.7X$ and $1{,}562X$ higher throughput than AlexNet [70] and VGG-16 [133] respectively. The work in [161] shows that a five-layer CNN has comparable accuracy to those of HOG designs while consuming $100X$ more energy. In addition, my experiments show that my hybrid design is $44.7X$ more energy efficient and achieves $13.4X$ higher throughput than AlexNet on the Titan X GPU [103].

3.7 Conclusions

In this work, I have investigated the throughput and energy efficiency of HOG3D-based HAR applications acceleration on FPGAs and GPUs for edge computing where high performance and energy economy are at a premium. I have identified four stages in this application and have explored the design constraints of each stage on the target platforms. I have developed a detailed I/O and computational complexity analysis of each of these stages and used this insight to guide my heterogenous implementation. My results show that a heterogeneous implementation where the first stage, the video pre-processing, is implemented on the FPGA and the other three stages are implemented on the GPU achieves the highest throughput and energy efficiency. Specifically, the heterogeneous HAR algorithm achieves $1.3X$ speedup when compared with the $K80$ GPU, $2.5X$ when compared with the

$K40$ GPU, and $2.8X$ when compared with the $K20$ GPU. Similarly, my heterogeneous HAR design is $1.5X$ and $2.0X$ more energy efficient when compared with the $K80$ and $K40$ GPUs. I have shown that HOG3D can be implemented via a reduced fixed-point processing pipeline without compromising the recognition accuracy. Additionally, my design has comparable accuracy to those of HAR design using five-layer CNNs while been more energy efficient.

Chapter 4

Acceleration of Quantum Simulations

4.1 Introduction

Modern quantum chemistry techniques depend critically on massively parallelized computational hardware to enable accurate calculations of the many-body electronic Schrödinger equation. Indeed, over the past two decades, the quantum chemistry community has witnessed tremendous technological advancements in computing that have enabled simulations of chemical/material systems of increasing complexity. These advancements have become even more prominent as we rapidly approach the dawn of exascale computing, with machines capable of performing a million trillion floating-point calculations per second [126, 91, 77]. However, to enable these massive calculations, recent exascale computing guidelines [78, 39, 117] have strongly cautioned that this increase in computing power should

only require a modest increase in power consumption (to offset both operation costs and deleterious climate change effects). Maintaining this delicate balance between computational performance vs. energy efficiency is extremely difficult since recent reports [92, 55] have shown that even small supercomputing centers regularly consume 500-1000 kW of power over the course of the year, resulting in over $1 million for power costs alone. These estimates do not even account for cooling costs, which have been reported to make up 25–50% of total power required by large data centers [122]. To partially mitigate these issues, this work is a first attempt to address these emerging parallelization and power-usage concerns via FPGAs.

While my use of FPGAs bears some resemblance to the techniques used by Shaw and co-workers to accelerate molecular dynamics calculations with the customized Anton machine [129, 130, 131], the approach utilized in my work has several distinct differences. In particular, Anton belongs to a class of computing architectures known as application-specific integrated circuits (ASICs), which are extremely expensive to design and hard to modify when new types of calculations are desired. Compared to ASICs, FPGAs can be re-configured for a variety of applications. In addition, FPGA solutions are significantly less expensive to manufacture and to power.

In this chapter, I present the first application of FPGAs for use in massively parallelized quantum dynamics of large chemical systems (up to 3,338 atoms). My motivation for implementing RT-TDDFTB with FPGAs is two-fold: (1) the RT-TDDFTB formalism is highly parallelizable, and (2) the techniques presented in this work can be used as a first step towards full DFT-based electron dynamics simulations on FPGAs. To assess the advantages

of the proposed design, I compare its performance with that of the RT-TDDFTB simulations using two highly optimized libraries: (1) the RT-TDDFTB simulations implemented in the CUBLAS Linear Algebra Library [104] (running on a NVIDIA K40 GPU), and (2) the RT-TDDFTB simulations implemented in the Intel Math Kernel Library (MKL) [145] (running on a Intel Xeon processor) in conjunction with OpenMP for multi-threading. The contributions of my work are three-fold:

- I have implemented a highly optimized engine that focuses on the execution of RT-TDDFTB simulations on FPGAs. The engine takes advantage of various hardware optimization techniques such as *tiling*, *deep pipelining*, and *memory bursting*. Multiple parallel instances of this engine are placed and routed on a Virtex FPGA running at 166 MHz. By exploiting the structure of the input matrices, the engine is able to execute over 256 complex-value floating point operations per clock cycle.

- For medium and large RT-TDDFTB simulations, the proposed engine outperforms the competing platforms. In particular, when the RT-TDDFTB simulation has over a thousand atoms, my engine achieves an 1.4× speedup compared with the competing libraries. Furthermore, because the performance of the proposed engine increases linearly with the number of atoms under simulation, the performance gaps are expected to increase for larger systems.

- In addition, my experimental results show that the proposed engine is energy-efficient. On average, CPUs and GPUs consume 3.77 and 4.05 times more energy respectively. These gains in energy efficiency are due to the presence of highly optimized *wide* and *deep* pipelines. While the *wide* pipelines allows for the parallel execution of tiled

matrix multiplication operations, the *deep* pipelines allows for the serial execution of dozens of complex-value floating point operations within these blocks.

4.2 Theory and Computational Methodology

Before proceeding with a detailed description of my FPGA parallelization enhancements, I first give a brief overview of the RT-TDDFTB formalism. Over the past few years, the RT-TDDFTB approach has garnered significant attention as an extremely efficient technique for probing the non-equilibrium electron dynamics of extremely large chemical systems. Specifically, the RT-TDDFTB approach have been used to understand photo-injection dynamics in dye-sensitized TiO_2 solar cells [110, 99, 100], many-body interactions in solvated nanodroplets [109], and excitation energy transfer dynamics in plasmonic arrays [59, 60]. These real-time quantum dynamics calculations are carried out by applying a time-dependent electric field to the initial ground state density matrix, resulting in an explicitly time-dependent Hamiltonian $\hat{\mathbf{H}}(t)$=$\hat{\mathbf{H}}^{\mathbf{0}} - \mathbf{E}_0(t) \cdot \hat{\boldsymbol{\mu}}(t)$, where $\mathbf{E_0}(t)$ is the applied electric field, and $\hat{\boldsymbol{\mu}}(t)$ is the dipole moment operator. Since the quantum system is directly propagated in the time-domain, $\mathbf{E}_0(t)$ can have any arbitrary time-dependent form. For example, if $\mathbf{E}_0(t)$ is a Dirac delta function, $\mathbf{E}_0(t) = \delta(t - t_0)$, this yields an optical absorption spectrum (obtained after a Fourier transform of the time-evolving dipole moment). However, if $\mathbf{E}_0(t)$ takes the form of a sinusoidal perturbation, it represents a continuous interaction of the system with monochromatic light in the time domain. When either of these time-dependent fields are applied, the density matrix $\hat{\boldsymbol{\rho}}$ evolves according to

the Liouville-von Neumann equation of motion which, in the nonorthogonal-DFTB basis, is given by

$$\frac{\partial \hat{\boldsymbol{\rho}}}{\partial t} = \frac{1}{i\hbar}(\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}] \cdot \hat{\boldsymbol{\rho}} - \hat{\boldsymbol{\rho}} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}] \cdot \mathbf{S}^{-1}), \tag{4.1}$$

where $\hat{\mathbf{H}}$ is the Hamiltonian matrix (which implicitly depends on the density matrix), $\mathbf{S^{-1}}$ is the inverse of the overlap matrix, and $\hbar$ is Planck's constant. When the applied incident fields are smaller than the internal fields in a molecule or material, the system is in the linear response regime [96]. Under these conditions, the time evolution of the dipole moment operator can be expressed as the convolution between the applied electric field perturbation, resulting in the following response function of the system

$$\langle \hat{\boldsymbol{\mu}}(t) \rangle = \int_0^\infty \boldsymbol{\alpha}(t-\tau)\mathbf{E}(\tau)d\tau, \tag{4.2}$$

where $\mathbf{E}(\tau)$ is the electric field that induces a perturbation in the Hamiltonian, and $\boldsymbol{\alpha}(t-\tau)$ is the polarizability tensor. Upon application of the convolution theorem, Equation 4.2 can be expressed in the frequency domain as $\langle \hat{\boldsymbol{\mu}}(\omega) \rangle = \boldsymbol{\alpha}(\omega)\mathbf{E}(\omega)$. The imaginary part of the average polarizability, $\bar{\alpha}$ is an experimentally measurable quantity related to the photoabsorption cross section by the expression $\sigma(\omega) = 4\pi\omega/c \cdot \mathrm{Im}(\bar{\alpha})$, where c is the speed of light, and $\mathrm{Im}(\bar{\alpha})$ is the imaginary part of the average polarizability.

In this work, I utilized the DFTB+ code [5] to construct the ground-state Hamiltonian, overlap matrix elements, and the initial single-electron density matrix within the self-consistent DFTB approach. With these ground-state quantities pre-computed, excited-state electron dynamics calculations were carried out with a customized RT-TDDFTB implementation on both GPU and FPGA hardware architectures. To enhance the efficiency

of the RT-TDDFTB calculations, the majority (roughly over 75%) of the computation of Equation 4.1 was offloaded to a co-processor either an FPGA or a GPU as described previously. To enable this efficiency, Equation 4.1 was computed in multiple steps as follows.

1) In the CPU, the self-consistent charge (SCC) and non-SCC Hamiltonian matrices are parsed in conjunction with the overlap matrix, orbital-wise electron fillings, and spatial coordinates of the system. The corresponding data structures for the density, overlap, and Hamiltonian matrices are subsequently generated.

2) Within the CPU, the matrix product $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)]$ is computed.

3) The matrices $\hat{\boldsymbol{\rho}}(t)$, $\hat{\boldsymbol{\rho}}(t)^T$, $\hat{\boldsymbol{\rho}}(t-\Delta t)$, and the matrix resulting from step 2 are transferred to the co-processor (i.e., a GPU or an FPGA).

4) $\hat{\boldsymbol{\rho}}_1(t+\Delta t) = \frac{1}{i\hbar}\left\{(\mathbf{S}^{-1}\hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)])\hat{\boldsymbol{\rho}}(t)\right\}(2\Delta t) + \hat{\boldsymbol{\rho}}(t-\Delta t)$ is computed in the co-processor.

5) $\hat{\boldsymbol{\rho}}_2(t+\Delta t) = \frac{1}{i\hbar}\left\{(\mathbf{S}^{-1}\hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)])\hat{\boldsymbol{\rho}}(t)^T\right\}(2\Delta t)$ is computed in the co-processor.

6) The resulting matrices from steps 4 and 5 are transferred to the CPU where the three-point formula $\hat{\boldsymbol{\rho}}(t) = \hat{\boldsymbol{\rho}}_1(t) - \hat{\boldsymbol{\rho}}_2(t)^T$ is computed (i.e., a simple subtraction of two pre-computed quantities with little computational overhead).

7) The density $\hat{\boldsymbol{\rho}}$ and Hamiltonian $\hat{\mathbf{H}}$ matrices are updated in the CPU.

8) The entire process starting with step 2 is repeated to propagate the electron dynamics for the desired time duration. The time-dependent charges, dipole moment, and density matrices are subsequently processed.

4.3 Chemical Systems and General FPGA Matrix Operations

Since the main focus of this work is to implement and understand FPGA performance gains for computing electron dynamics, I have chosen a representative set of large chemical structures to assess its efficiency and computational scaling. To this end, I have constructed a set of hydrogen-terminated carbon nanoribbons [152] ranging from 62 – 3,338 atoms, and Figure 4.1 depicts a subset of these structures as a function of size. It is worth mentioning that I specifically chose 3,338 atoms as the upper limit since this corresponds to the maximum matrix size that can be held in the memory of the GPU used in my performance benchmarks.

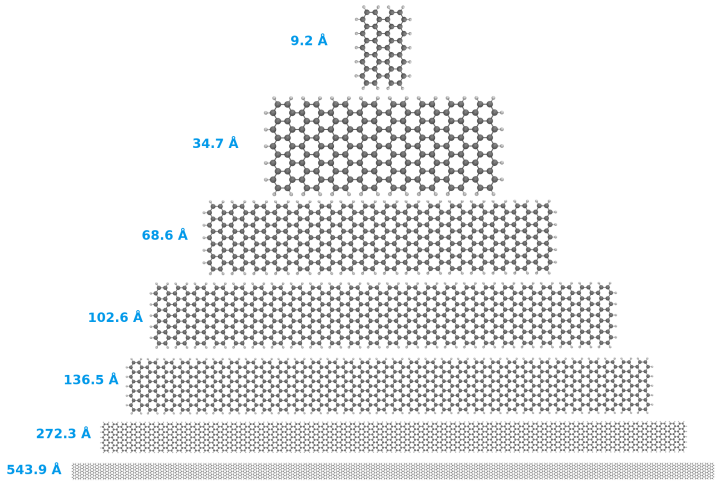


Figure 4.1: A representative subset of the carbon nanoribbons with various lengths examined in this work.

In computing the electron dynamics of these large nanoribbons, it is worth noting that both the Hamiltonian matrix $\hat{\mathbf{H}}$ and the inverse of the overlap matrix $\mathbf{S}^{-1}$ in Equation 4.1 are real-valued, whereas the density matrix $\hat{\boldsymbol{\rho}}$ is complex-valued. Moreover, while

the density matrix $\hat{\boldsymbol{\rho}}(t)$ is dense, the matrix product $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}]$ is sparse, which increases as a function of the nanoribbon size as shown in Figure 4.2.

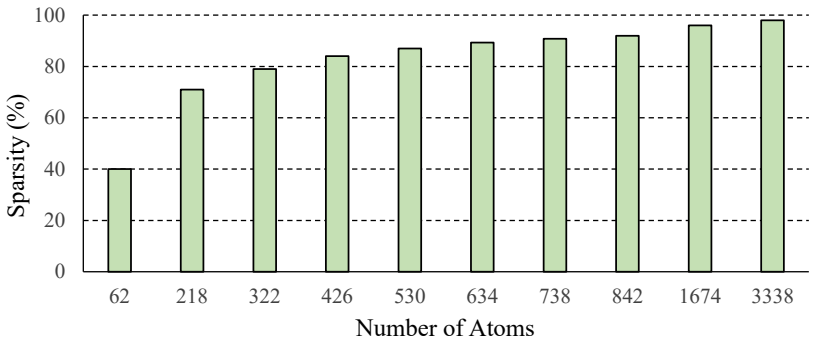


Figure 4.2: Sparsity of the matrix product $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}]$ as a function of nanoribbon size.

To efficiently parallelize the RT-TDDFTB calculations on FPGAs, I designed a software/hardware kernel that executes steps four and five (which are the most computationally demanding steps, as described above) and has the capacity for transferring matrices to and from the co-processor. Moreover, by supporting the matrix operations described in step four, the implementation for the matrix operations described in step five is already satisfied since the reading of $\hat{\boldsymbol{\rho}}(t - \Delta t)$ can be omitted in the latter step. To this end, I created a general-purpose hardware kernel to support the operation

$$\mathbf{C}^k = \alpha \mathbf{A}\mathbf{B} + \beta \mathbf{C}^{k-1}, \tag{4.3}$$

where the superscript $\mathbf{k}$ denotes the kth iteration, $\mathbf{A}$ is a real-valued matrix, and the matrices $\mathbf{B}$ and $\mathbf{C}$ along with the parameters α and β are complex-valued. As such, the RT-TDDFTB simulations in the co-processor can be enabled by setting $\mathbf{A} = \mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)]$, $\mathbf{B} = \hat{\boldsymbol{\rho}}(t)$ or $\mathbf{B} = \hat{\boldsymbol{\rho}}(t)^T$, and $\mathbf{C}^{k-1} = \hat{\boldsymbol{\rho}}(t - \Delta t)$. In addition, the parameter α was set to $\frac{1}{i\hbar}(2\Delta t)$ while the parameter β is real and set to one. As described in the next, my kernel

exploits the sparsity of $\mathbf{A}$ and allows us to decrease both the input/output (I/O) and the computational complexity of the matrix operations to be offloaded to the FPGA.

4.4 Baseline FPGA Design and Architecture

I first present a general (but detailed) hardware design for carrying out parallelized matrix multiplications. I designate this as the "baseline" FPGA hardware design, with Section 4.4.1 describing my baseline implementation for real-valued matrix multiplications and Section 4.4.2 giving my modifications for complex-valued matrix operations. Section 4.5 presents additional acceleration techniques tailored specifically to the efficient propagation of RT-TDDFTB electron dynamics on FPGAs.

4.4.1 Real-Valued Matrix Multiplications on FPGAs

The multiplication of real-valued matrices on FPGAs continues to be a topic of interest, [35, 72, 62, 160, 159, 80, 8] and to enable the computations required by the RT-TDDFTB simulations (Eq. 4.3) I have modified a previous design [72] that was used for real-valued, dense matrix multiplication. I commence with Figure 4.3, which depicts a general-purpose schematic for parallelization of real-valued matrix multiplication on FPGAs. To allow my baseline design to be completely general, the size of the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are $n \times m$, $m \times l$, and $n \times l$, respectively. Moreover, matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ have been partitioned into sub-blocks of size $p \times m$, $m \times p$, and $p \times p$, respectively (with n/p and l/p being integer numbers). Within this schematic, the computation of each block $\mathbf{C}_{ij}$ can be obtained via multiplications of the corresponding blocks in $\mathbf{A}$ and $\mathbf{B}$ (i.e., $\mathbf{C}_{11} = \mathbf{A}_{11}\mathbf{B}_{11}$).

More generally, each block $\mathbf{C}_{ij}$ can be calculated as outer products ($\cdot$) between the columns of block $\mathbf{A}_{i1}$ and the rows of block $\mathbf{B}_{1j}$ such that $\mathbf{C}_{ij} = \boldsymbol{a}_1 \cdot \boldsymbol{b}_1 + \boldsymbol{a}_2 \cdot \boldsymbol{b}_2 + ... + \boldsymbol{a}_m \cdot \boldsymbol{b}_m$, where the column vector $\boldsymbol{a_i}$ is the ith column of block $\mathbf{A}_{i1}$ and the row vector $\boldsymbol{b_i}$ is the ith row of block $\mathbf{B}_{1j}$. In the terminology of computational linear algebra algorithms, matrices having the form $\boldsymbol{a}_k \cdot \boldsymbol{b}_k$ are rank-one matrices, and the addition of rank one matrices is called a rank one update [151, 143]. By using these rank one updates, I can improve both I/O bandwidth and parallelism, since if one element of the column vector $\boldsymbol{a}_k$ as well as the row vector $\boldsymbol{b}_k$ are available, I can execute p multiply-and-accumulate operations simultaneously. Moreover, $\boldsymbol{b}_k$ can be reused p times to improve the performance of matrix multiplications on FPGAs [35, 72].

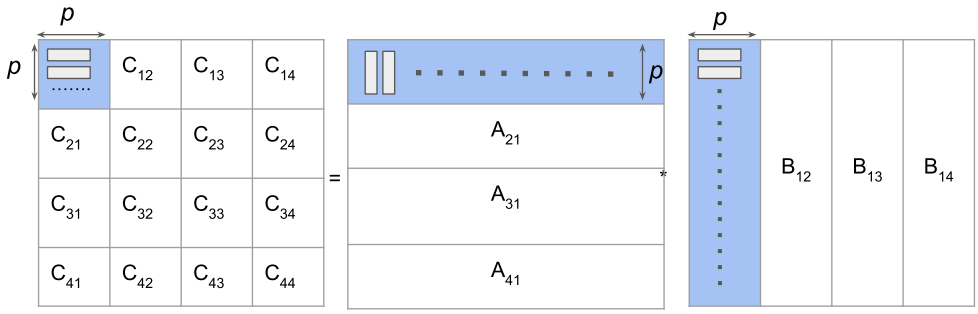


Figure 4.3: Schematic of parallelized matrix multiplication on FPGAs. The computation of the block $\mathbf{C}_{11}$ can be obtained via the outer-products between the columns of $\mathbf{A}_{11}$ and the rows of $\mathbf{B}_{11}$.

The general framework for carrying out these parallelized matrix multiplication operations on FPGAs is shown as a high-level flowchart in Figure 4.4. This hardware engine is comprised of six modules designated as the *Scheduler*, *Reader*, *Read A Controller*, *Read B Controller*, *Multiply-and-Accumulate*, and the *Write C Controller*. All communication between these modules is executed via first-in first-out blocks (FIFOs) [41]. While the

design of FPGA *Readers*, *Writes*, and *Controllers* is generally well established, multiple designs have been proposed for the implementation of the *Multiply-and-Accumulate* unit. In this work, I have modified a previous design [72] in which one FIFO (shown in the top left of Figure 4.5) stores the elements of the columns of matrix $\mathbf{A}_{i1}$ (i.e., the column vectors $\boldsymbol{a}_k$). Similarly, at the top of Figure 4.5, the module has p FIFOs to store the rows of the matrix $\mathbf{B}_{1j}$ (i.e., the row vectors $\boldsymbol{b}_k$). In the center of Figure 4.5, p *Multiply-and-Accumulate* units execute the multiply-and-accumulate operation. At the bottom of Figure 4.5, p FIFOs are used for storing the final values of $\mathbf{C}_{ij}$ with each FIFO having capacity for p elements. The right part of Figure 4.5 shows the components of a *Multiply-and-accumulate* unit. A block RAMs (BRAM) is used to store the partial values of $\mathbf{C}_{ij}$. Each block RAM have p addresses with either 32 or 64 bits per address to hold single or double precision numbers. A multiplexer is used to multiplex one of the inputs of the adder. During most of the computation, the input to the adder is a numerical value from the BRAM; however, the multiplexer outputs zero when the calculation of a new block $\mathbf{C}_{ij}$ starts.

The computation of $\mathbf{C}_{ij}$ commences as follows. First, the *Controller* signals the reading of the first row of block $\mathbf{B}_{1j}$ and the first column of block $\mathbf{A}_{i1}$, which are executed by the *Read B* and *Read A* Controller, respectively. These values are stored in the p FIFOs labeled $b_{k,0}, ..., b_{k,p-1}$ and the FIFO labeled $a_{i,k}$ respectively. The *Controller* then commands the *Multiply-and-Accumulate* module to carry out p multiplications in parallel; i.e., $a[0,0] * b[0,j]$ for $j = 0, ..., p-1$. The results of these multiplications are subsequently added to the zero values coming from the multiplexers and the results are stored in the BRAM at address zero. After the element $a[1,0]$ arrives to the top-left FIFO, the *Controller*

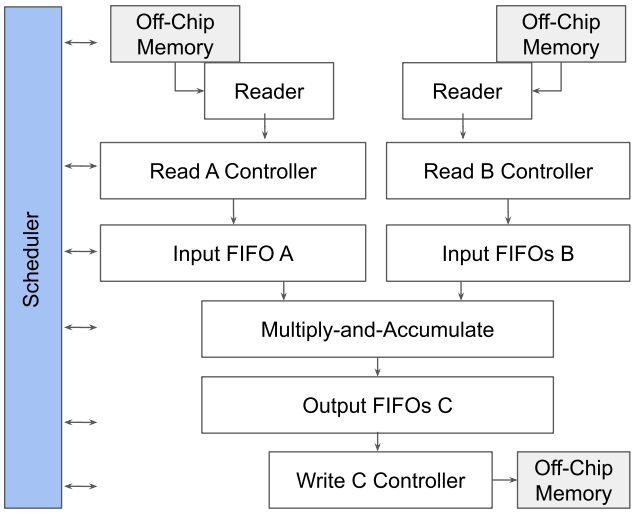


Figure 4.4: High-level view of the design for parallelizing the RT-TDDFTB simulations in hardware. In this figure, the *Scheduler* directs the execution of tasks to the other modules. The *Read (Write)* controller reads (writes) one input matrix from (to) the off-chip memory. Finally, the *Multiply-and-Accumulate* module executes the matrix multiplication operation.

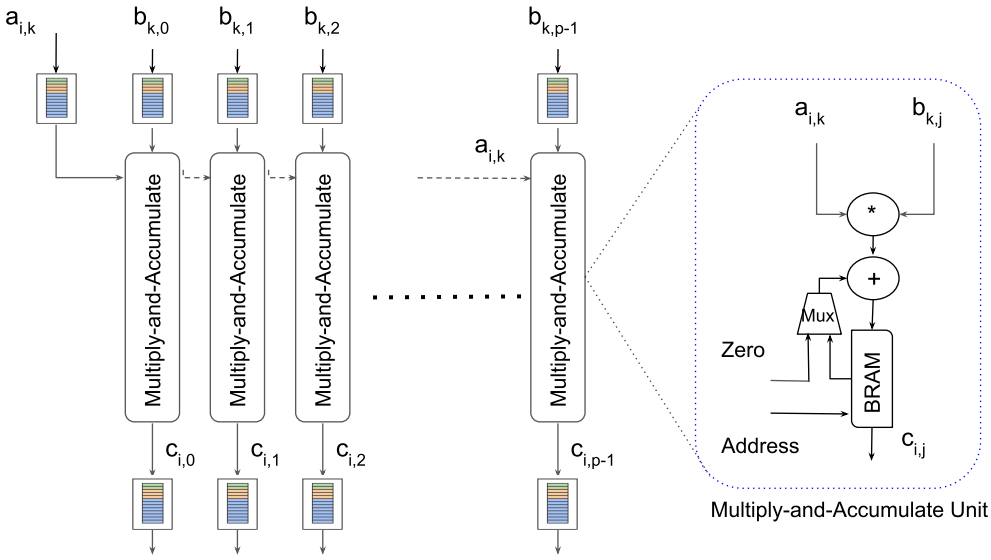


Figure 4.5: Hardware implementation of the *Multiply-and-Accumulate* module. This module executes the real-valued outer-products between the columns of matrix **A** and the rows of matrix **B**. The partial results are stored in block RAMs. The final results are stored in the FIFOs shown at the bottom. On the right part, the components of the *Multiply-and-Accumulate* unit are shown.

signals the execution of p new multiplications and p new additions. Finally, the results are stored in RAM at address one, and this process continues until the outer product between the first column of $\mathbf{A}_{i1}$ and the first row of $\mathbf{B}_{1j}$ is completed.

In addition to the tasks mentioned above, the *Controller* directs the execution of the outer product between the second column of $\mathbf{A}_{i1}$ and the second row of $\mathbf{B}_{j1}$. The results of these multiplications are then added to the previous values stored in the BRAM. This process continues until the outer product between the last column of $\mathbf{A}_{i1}$ and the last row of $\mathbf{B}_{j1}$ is executed. At this point, the operation $\mathbf{C}_{ij} = \mathbf{A}_{i1}\mathbf{B}_{1j}$ is completed, and the results are stored in the p BRAM. The content of the BRAM is written to the $c_{i,0}, ..., c_{i,p-1}$ FIFOs one row at the time. Finally, the *Scheduler* signals to the *Write C Controller* to write the content of these FIFOs to the off-chip memory, and this process continues until all the $\mathbf{C}_{ij}$ blocks are computed.

It is worth mentioning a few practical notes that can be used to enhance the efficiency of real-valued matrix multiplication on FPGAs. First, the block $\mathbf{C}_{ij}$ does not have to be square, and its size can be tailored to any specific FPGA hardware platform [35]. For example, if the block $\mathbf{C}_{ij}$ has dimensions of $p \times q$, the parameter p can be increased to yield higher efficiency on FPGA platforms that have more on-chip memory. Second, for FPGAs with abundant floating point units (FPU), the parameter q can be increased as well. Third, if the delay in the floating point addition is v cycles, it is desirable to have $p \geq v$ to maintain computational efficiency. This constraint arises since the accumulations of the previous outer product must be finished before the next outer product starts, or the pipeline

will be stalled. Finally, if one has access to large FPGAs, or multiple FPGAs, several $\mathbf{C}_{ij}$ blocks can be computed in parallel using the computational techniques discussed previously.

4.4.2 Complex-Valued Matrix Multiplications on FPGAs

In this section, I describe my customized baseline design for complex-valued matrix multiplications on FPGAs. While the FPGA engine described in the previous section supports the real-valued matrix multiplication in the expression $\mathbf{C} = \mathbf{AB}$, I implemented a new design for computing complex-valued matrix multiplications required for propagating RT-TDDFTB electron dynamics (cf. Eqs. 4.1 and 4.3). To support this new capability, I first compute an intermediate matrix $\mathbf{T}_{ij}$ given by

$$\mathbf{T}_{ij} = \mathbf{A}_{i1}\mathbf{B}_{1j}, \tag{4.4}$$

where the blocks of the matrix $\mathbf{A}$ and $\mathbf{B}$ are real- and complex-valued, respectively. Figure 4.6 depicts my customized complex-valued multiply-and-accumulate module that executes this parallelized operation. Compared to the real-valued multiply-and-accumulate module shown previously in Figure 4.5, p multiply-and-accumulate units have been added so as to compute the real ($t^r_{i,k}$) and imaginary ($t^i_{i,k}$) values of $\mathbf{T}_{ij}$ in parallel. In this figure, the elements of $\mathbf{T}_{ij}$ are serialized (on row at the time) into two FIFOs, labeled s^r and s^i that contain the real and imaginary elements of $\mathbf{T}_{ij}$. With these values in hand, I next compute the following matrix

$$\mathbf{C}^k_{ij} = \alpha\mathbf{T}_{ij} + \beta\mathbf{C}^{k-1}_{ij}, \tag{4.5}$$

which is carried out by the *Complex Accumulator* module shown in Figure 4.7. The design depicted in Figure 4.7 has been harnessed with a new *Read C Controller* module. This additional FPGA module reads the complex values of matrix $\mathbf{C}^{k-1}$ from the off-chip memory into the FIFOs labeled c^r and c^i (i.e., the real and imaginary parts of $\mathbf{C}^{k-1}$). The *Complex Accumulator* module executes eight multiplications and six additions, with the real and imaginary values of $\mathbf{C}^k_{ij}$ written into the FIFOs labeled t^r and t^i, respectively. At the end of the computation, the *Writer C Controller* writes the block $\mathbf{C}^k_{ij}$ into the off-chip memory.

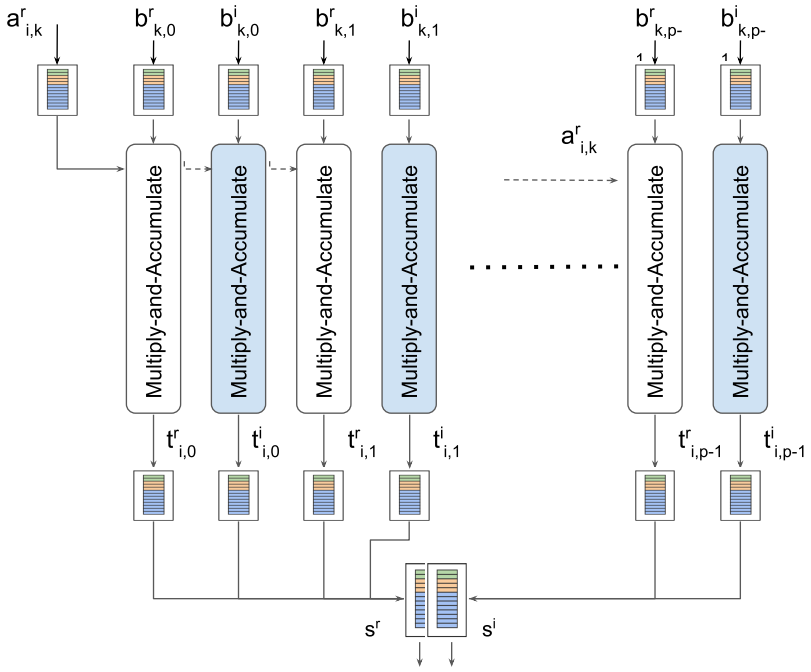


Figure 4.6: Hardware implementation of the *Complex Multiply-and-Accumulate* module. This module executes the complex outer-products between the real columns of matrix $\mathbf{A}$ and the complex rows of matrix $\mathbf{B}$. The values of the resulting matrix are serialized to the bottom FIFOs s^r and s^i.

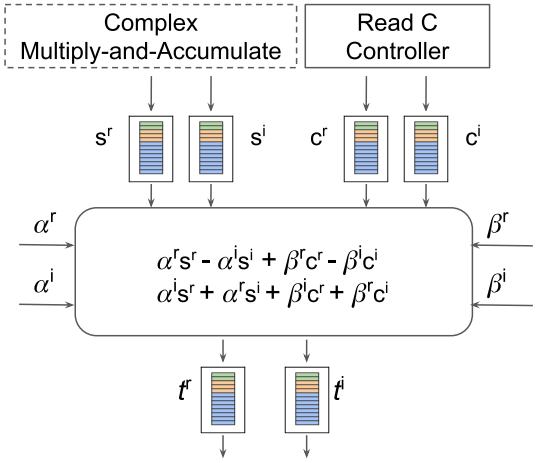


Figure 4.7: Hardware implementation of the *Complex Accumulator* module. This module executes the operation $\alpha\mathbf{T}_{ij} + \beta\mathbf{C}_{ij}^{k-1}$. The values of $\mathbf{T}_{ij}$ are in the top-left FIFO while the values of $\mathbf{C}_{ij}^{k-1}$ are in the top-right FIFO. The complex results are stored in the bottom FIFOs.

4.5 Optimized FPGA Design for Efficient Propagation of RT-TDDFTB Electron Dynamics

Having described my baseline FPGA design, I now present further optimizations that were added to speed up the RT-TDDFTB simulations. In these simulations, as mentioned above, the matrix $\mathbf{A} = \mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)]$ is sparse, whereas the matrices $\mathbf{B} = \hat{\boldsymbol{\rho}}(t)$ and $\mathbf{C}^{k-1} = \hat{\boldsymbol{\rho}}(t - \Delta t)$ are dense. As a result, my baseline design was modified to take advantage of the sparsity of $\mathbf{A}$ to satisfy the following three constraints:

1) Since the input matrix $\mathbf{C}^{k-1}$ is dense, all the elements of the matrix $\mathbf{T}_{ij}$ are required to execute the addition shown in Equation 4.5. Thus, a number of operations in the multiplication of $\mathbf{A}_{i1}\mathbf{B}_{1j}$ must be executed to generate all the elements of $\mathbf{T}_{ij}$.

2) To generate all the values of $\mathbf{T}_{ij}$, one must initialize and output the values of the BRAM into the corresponding FIFOs within my baseline *Complex Multiply-and-Accumulate* module depicted in Figure 4.6. The initialization of the BRAM can be achieved by executing the outer products between the first column of the block $\mathbf{A}_{i1}$ and the first row of block $\mathbf{B}_{1j}$. Similarly, the outputs can be generated by executing the outer products between the last column of block $\mathbf{A}_{i1}$ and the last row of $\mathbf{B}_{1j}$.

3) One can take advantage of the sparsity of $\mathbf{A}$ by utilizing a sparse matrix representation scheme. For instance, in computations where all the elements in the columns of $\mathbf{A}_{i1}$ are zero, it is not necessary to read the corresponding row in the matrix $\mathbf{B}_{1j}$ since the results of these multiplications are zero. The only exception to this situation is the second constraint mentioned previously.

To address the requirements mentioned above, I utilized a compressed sparse blocks (CSB) representation [15] with additional customized modifications. A schematic of this representation is shown in Figure 4.8. To enable these parallelized calculations, Ref. [15] utilizes an integer array that contains the number of nonzero elements per block, where each block is represented using the compressed sparse row (CSR) representation. In this work, I also utilized an integer array containing the number of elements per block; however, I represent each block using a coordinate list format (COO) representation. In the COO representation, the input matrix can be represented in row- or column-major order; I chose the latter convention since this representation meets the requirements of my design. For computational efficiency, my implementation browses the block $\mathbf{A}_{i1}$ one column at a time; thus, before matrix $\mathbf{A}$ is sent from the CPU to the FPGA, it is first divided into blocks,

and its CSB representation is generated. The matrices **B** and **C** are then sent to the FPGA as flat two dimensional arrays.

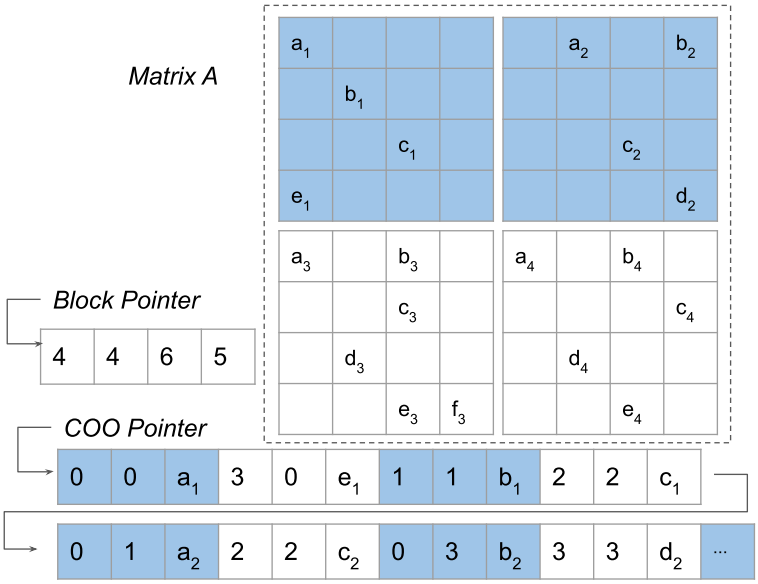


Figure 4.8: Schematic of the compressed sparse blocks (CSB) matrix representation used in this work. The input matrix is divided into four blocks of size 4×4. While the block pointer points to an array containing the number of nonzero elements per block, the coordinate list (COO) pointer points to an array containing the column index, row index, and the value of the nonzero elements in each block.

To accommodate the CSB representation of matrix **A**, additional modifications of my baseline implementation are required. These modifications only alter the *Read A Controller* and *Read B Controller* modules, with minor changes to the *Complex Multiply-and-Accumulate* module. Figure 4.9 depicts my enhanced FPGA design where the *Read A Controller* now includes two additional input signals: the *Block Pointer* and the *COO Pointer*. My enhanced design operates as follows:

1) To compute $\mathbf{T}_{ij}$, the *Read A Controller* signals the *Reader* to read the elements of block $\mathbf{A}_{i1}$. This operation makes use of the *Block* and *COO* signal.

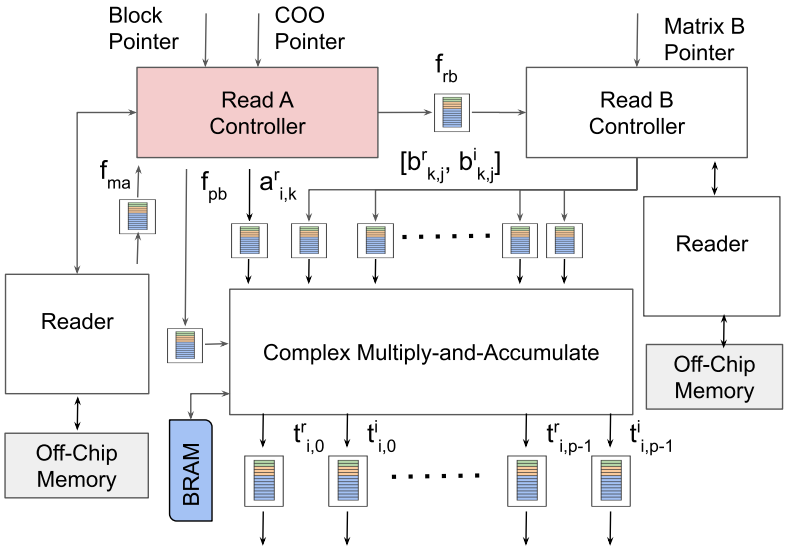


Figure 4.9: Hardware implementation of the optimized blocked complex-matrix multiplication module. My implementation harnesses the *Block* and *COO* pointer to exploit the sparsity of $\mathbf{A}_{i1}$, and, as a result, dramatically decreases the complexity of the computation $\mathbf{A}_{i1}\mathbf{B}_{1j}$.

2) The *Reader* places the elements of the COO array (the column index, row index, and real values of $\mathbf{A}_{i1}$) into the FIFO labeled f_{ma}.

3) The *Read A Controller* reads the elements in the f_{ma} FIFO and signals the *Read B Controller* to read the next row (the column index in the COO array indicates the next row to read in the block $\mathbf{B}_{1j}$). In addition, the *Read A Controller* places the elements of the columns of $\mathbf{A}_{i1}$ into the $a^r_{i,k}$ FIFO.

4) The *Read A Controller* notifies the *Complex Multiply-and-Accumulate* module (via the f_{pb} FIFO) when a new outer product between the columns of $\mathbf{A}_{i1}$ and the rows of $\mathbf{B}_{1j}$ has to be executed.

5) The *Complex Multiply-and-Accumulate* module reads the input FIFOs $b^r_{k,0}, b^i_{k,0}, ..., b^r_{k,p-1}, b^i_{k,p-1}$ as indicated in FIFO f_{pb}. In addition, this module reads the input FIFO $a^r_{i,k}$. These

reading operations correspond to the next row in $\mathbf{B}_{1j}$ and the next column-element in $\mathbf{A}_{i1}$, respectively.

6) Once these $2p+1$ FIFOs are populated, the *Complex Multiply-and-Accumulate* module executes p complex multiply-and-accumulate operations and stores the results into the BRAM.

7) As in my baseline design, this process continues until all the outer products between the columns of $\mathbf{A}_{i1}$ and the rows of $\mathbf{B}_{1j}$ are completed. At the end, $\mathbf{T}_{ij}$ is fully calculated.

8) Finally, the *Complex Accumulator*, shown in Figure 4.7, takes $\mathbf{T}_{ij}$ as input and computes $\mathbf{C}_{ij}^{k}$ as described in my baseline design.

The FPGA design, as described previously, functions properly and efficiently in steady state, assuming that the pipelines do not have to be stalled. However, due to the sparsity of the input block $\mathbf{A}_{i1}$, I must account for stalls, which occur when a row of $\mathbf{T}_{ij}$ is updated at cycle k, and later, when the same row has to be updated at cycle $k + s$. Because the *add* operation in the FPGA takes v cycles, these updates are allowed if $s \geq v$, otherwise I intentionally stall the pipeline for $v - s$ cycles. The BRAM block shown at the bottom left of Figure 4.9 is used to track when a row in $\mathbf{T}_{ij}$ is updated. When row w of $\mathbf{T}_{ij}$ gets updated, the *Complex Multiply-and-Accumulate* module writes the wth position of this BRAM with the value of a counter. If row w requires an update, the *Complex Multiply-and-Accumulate* module queries the BRAM at position w and determines whether the pipeline has to be stalled by comparing the current value of the counter with the value stored in the BRAM.

It is worth noting that when the elements in the f_{ma} FIFO are processed, the *Read A Controller* is able to signal to the *Read B Controller* which specific rows in block $\mathbf{B}_{1j}$ to read. Each time that a row of $\mathbf{B}_{1j}$ is skipped, significant savings in bandwidth, as well as in the number of floating point multiplications, are achieved. Thus, for every row skipped, $4(2p)$ bytes are saved in I/O bandwidth, and p complex-valued multiplications and additions, are also avoided. As a result, by implementing all the FPGA acceleration strategies discussed previously, both the I/O as well as the complexity of computing $\mathbf{T}_{ij}$ are significantly lowered.

4.6 Experimental Results and Discussion

4.6.1 Experimental Environment

Figure 4.10 depicts a schematic of the FPGA hardware used in this work, which is composed of an Intel Xeon CPU E5-2460 interfaced with a Virtex-7 FPGA [94]. This specific FPGA configuration has 32 memory channels, each of which has a theoretical bandwidth of 1.25 GB/s. As such, to achieve maximum I/O performance, I configured my FPGA to execute read/write requests of 64 bytes aligned to 64-byte addresses (the FPGA can carry out read/write requests of 8, 4, 2, or 1 byte, but these smaller sizes result in a lower I/O performance). My FPGA implementation was written using Verilog HDL, and my hardware design was simulated with ModelSim [44] to test its accuracy. My FPGA implementation was synthesized, placed, and routed with Vivado 17.3 [37]. The target frequency of the

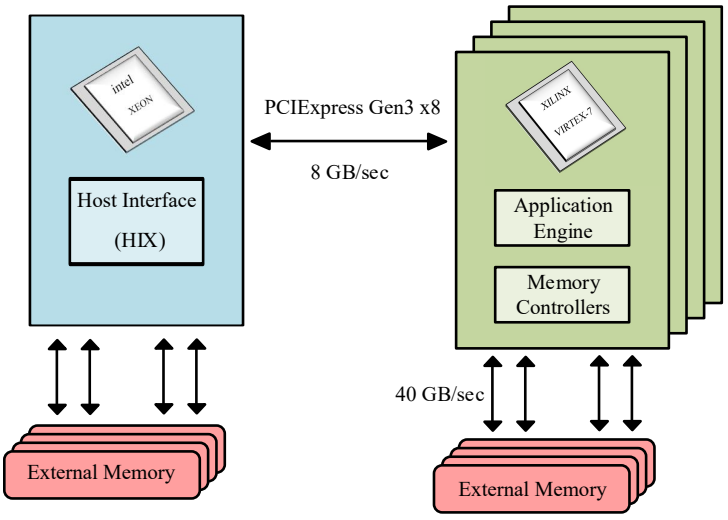


Figure 4.10: Schematic of the Micron Wolverine FPGA used in this work. This hardware architecture is comprised of a CPU with one FPGA attached via a PCIExpress Line. The FPGA is first configured with the specific simulation to be executed, and the CPU sends commands to the FPGA via the host interface. These commands include operations such as writing (reading) data to (from) the FPGA external memory, executing the computation, and querying the status of the computation.

FPGA was set to 167 MHz. All arithmetic operations were implemented on Xilinx cores [52] by taking advantage of either digital signal processors (DSP) or lookup tables (LUTs).

To assess the performance of my FPGA implementation against other computational hardware, I also examined computational timings and energy expenditures of both GPUs and CPUs. For the GPU benchmark tests, I utilized an NVIDIA K40 GPU equipped with an Intel Xeon E5-520 processor and 24 GB of RAM. To ensure a fair assessment of computational efficiency, my GPU-based RT-TDDFTB code was compiled with CUDA (release 9.0) in conjunction with the CUBLAS linear algebra library [104] to achieve optimal computational performance on the GPU. In all my GPU-based tests/comparisons, error correction capabilities (ECC) were disabled. For the CPU tests, I utilized an Intel Xeon E5-2643 processor operating at 3.40 GHz with 256 GB of RAM. Similar to my GPU bench-

mark tests, the CPU implementation utilized optimized routines within the Intel Math Kernel Library (MKL) in conjunction with OpenMP for multi-threading.

4.6.2 Single vs. Double Precision

I first present various metrics/benchmarks for calculating absorption spectra as a function of system size. Figure 4.11 shows the effects of carrying out the RT-TDDFTB simulations in single/double precision for several of the nanoribbons described above. The absorption spectrum for each nanoribbon was generated by propagating Eq. 4.1 in the presence of a Dirac delta electric field impulse applied along three mutually orthogonal directions to compute the polarizability tensor. The resulting time-varying dipole moment was then Fourier transformed to give the absorption spectrum. Regardless of the nanoribbon size, Figure 4.11 shows that the resulting spectra were extremely similar, independent of whether it was computed in single or double precision.

Table 4.1 gives a more quantitative comparison of numerical accuracy by calculating the mean squared error (MSE) of the computed spectra according to the following expression

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{4.6}$$

where y is the absorption spectrum of the nanoribbon computed in double precision and $\hat{y}$ is the corresponding spectrum calculated in single precision. I obtained a maximum MSE of 4.2 (corresponding to the largest nanoribbon), with many of the smaller nanoribbons exhibiting much lower errors. These benchmark results are important since RT-TDDFTB calculations performed in single precision significantly reduce the I/O bandwidth as well

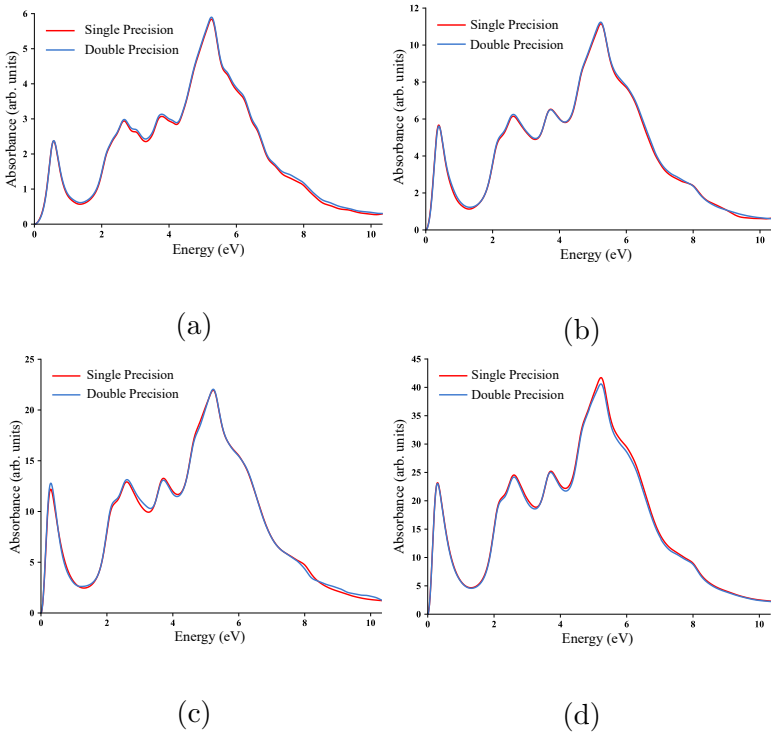


Figure 4.11: Absorption spectra of various carbon nanoribbons computed in single- and double-precision comprised of (*a*) 426, (*b*) 842, (*c*) 1,674, and (*d*) 3,338 atoms. In all cases, the absorption spectra computed in single precision is nearly indistinguishable from the double-precision spectra.

Table 4.1: System size (number of atoms and Hamiltonian matrix size) and mean squared errors (MSEs) for the various carbon nanoribbons computed with the FPGA-enabled RT-TDDFTB approach.

Number of Atoms	**Matrix Size**	**MSE**
62	194	~ 0
218	746	0.1
322	1,114	0.1
426	1,482	0.2
530	1,850	0.5
634	2,218	0.8
738	2,586	0.9
842	2,954	1.1
1,674	5,898	2.2
3,338	11,786	4.2

as the DSP resources [112] in FPGAs. Specifically, in FPGAs, the multiplication of two double or two single precision numbers requires 11 and 3 DSPs, respectively [52, 37]. As such, notwithstanding other hardware considerations, FPGAs can execute at least three times more multiplications per clock cycle when single precision arithmetic is used.

4.6.3 Computational Speedup of FPGAs vs. GPUs and CPUs

In this section, I compare the computational efficiency of the RT-TDDFTB FPGA implementation against execution times obtained with the GPU and CPU. Because FPGAs are configured at the hardware level, I can take advantage of all the resources available on the FPGA by including all of the I/O channels and most of the BRAM (nearly 75%). In particular, my RT-TDDFTB simulations were replicated such that four $\mathbf{C}_{ij}$ blocks of size 64×64 were computed in parallel, which allows $512 = 4(64 \times 2)$ single-precision floating

Table 4.2: Virtex-7 FPGA utilization for computing RT-TDDFTB electron dynamics

Resource	Available	Total Utilization (%)	Utilization per FPGA Engine (%)
Registers	2443K	45.11	11.27
LUTs	1221K	49.42	12.35
LUT RAM	344K	25.25	6.31
Block RAM	1.2K	74.27	18.56
FPUs	2.1K	55.56	13.89
Memory Channels	32	100	25

point operations per clock cycle. Table 4.2 provides a detailed accounting of the resources utilized by my FPGA implementation.

To ensure a fair comparison of computational efficiency, my GPU-based RT-TDDFTB code used optimized cuSPARSE libraries to compute the $\mathbf{S^{-1}} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}]$ matrix in Compressed Sparse Column (CSC) format. However, I did not observe any gains in efficiency (GPUs are less efficient for sparse matrix operations, as discussed further in the paragraphs below); as such, I report GPU performance and energy metrics for calculations that only utilized the CUBLAS dense routines [104]. For the CPU calculations and comparisons, my RT-TDDFTB simulations were executed on two and eight threads. Figure 4.12 compares the computational speedup obtained with the FPGA, GPU, and CPU for nanoribbon systems containing 62 – 3,338 atoms. As is customary in hardware performance profiling, the computational speedup of each hardware platform is normalized by dividing its execution time by the timings of the CPU running two threads.

For small RT-TDDFTB simulations on systems containing 62 – 530 atoms, the 8-thread CPU outperforms both the FPGA and GPU (with the GPU being slightly faster than the FPGA). The lower performance of the FPGA can be attributed to the size of the input matrices: since the FPGA relies on *wide* and *deep* pipelines to achieve high throughput,

these pipelines are not able to reach a steady-state when the input matrices are small. Moreover, the latency of the off-chip memory (which is on the order of hundreds of cycles for the hardware used in this work [94]), results in further inefficiencies. These latencies deepen the pipelines, and as a result, larger inputs are required before the pipelines achieve a steady-state. Thus, for RT-TDDFTB simulations on chemical/material systems with a small number of atoms, the pipelines are heavily underutilized. However, when the system size reaches 634 atoms, my FPGA implementation becomes more efficient than the GPU and is competitive with the 8-thread CPU. Finally, for large RT-TDDFTB simulations on systems containing over 842 atoms (where the sparsity is over 90%, as shown in Figure 4.2), my FPGA implementation outperforms both the GPU and CPU. Most importantly, as the number of atoms increases, the performance gap between the FPGA and the other competing hardware platforms increases as well (with the FPGA achieving a 5× speedup for the largest system).

It is also worth mentioning that the computational performance of the GPU and CPU starts to saturate/plateau for large systems, whereas the performance of the FPGA continues to increase. My FPGA implementation outperforms other platforms since it was specifically designed to take advantage of the sparsity of $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}]$, which effectively decreases the I/O and computational complexity of the problem (even more efficiently than GPUs). In particular, the performance of my FPGA implementation grows as a function of system size since the sparsity of $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}]$ increases with the number of atoms (cf. Figure 4.2).

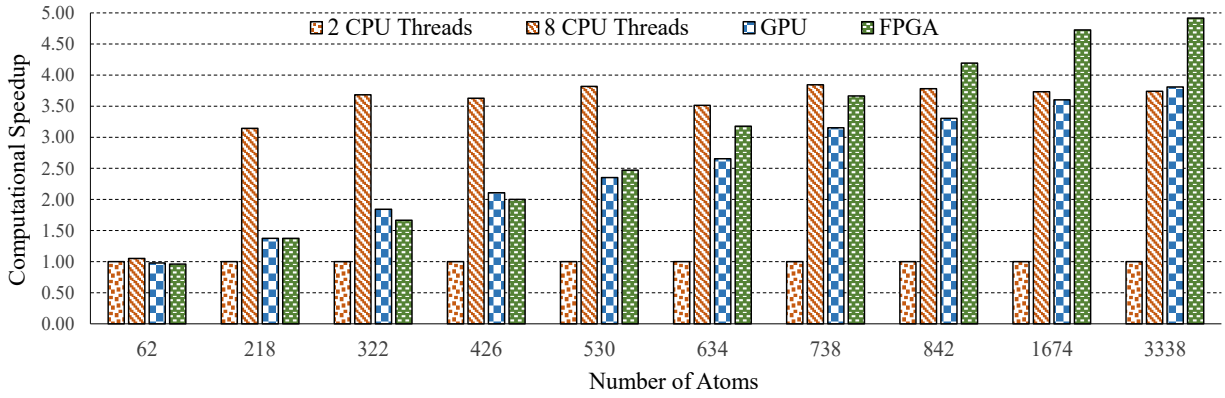


Figure 4.12: Comparison of computational speedup for the CPUs, GPUs (K40 architecture), and FPGAs (Virtex-7 architecture). For clarity, the speedup of each hardware platform is normalized by dividing its execution time by the timings of the CPU running two threads..

As demonstrated in my benchmark comparisons, it is important to emphasize that the multiplication of large, sparse matrices on GPUs is quite inefficient. Specifically, GPUs are much better suited for dense matrix operations and achieve about 60% of their theoretical peak performance, as measured in floating point operations per second, or FLOPS [27, 153]. However, GPU performance significantly degrades as the sparsity of the input matrices increases (even when a sparse library is used), resulting in about 10% FLOPS of the theoretical peak performance [27, 157, 48]. GPUs suffer this significant drop in efficiency since they belong to a hardware classification known as single instruction multiple data (SIMD) architectures [50] – a class of computational architectures that can only execute *the same instruction over multiple streams of data.* In short, GPUs were specifically designed to only (1) access contiguous chunks of data in off-chip memory via coalesced reads and writes, (2) provide a high off-chip memory bandwidth, (3) store efficiently small blocks of data in on-chip memory, and (4) execute a maximum number of floating point operations [67]. However, when the inputs of the matrices are sparse, GPUs encounter several

difficulties that incur immense computational overhead, including: (1) storing the input matrices in a sparse matrix representation format, (2) accessing the data in an indirect fashion, since the metadata describing the input matrix has to be accessed before the data itself is read, (3) not having enough inputs (due to the sparsity of the input data) to fully saturate the floating point units, and (4) having a non-trivial distribution of equal work among the stream processors [84]. While FPGAs encounter the first, second, and fourth difficulties mentioned previously, their pipelines can be *customized* to take advantage of the granularity of the input data. Moreover, FPGAs can be adapted to provide fine-grained access to off-chip memory, flexible on-chip memory storage, and wide/deep pipelines to fully tackle the problem at hand [134, 48]. As such, the use of FPGAs for these RT-TDDFTB electron dynamics applications shows significant performance gains (*even beyond modern GPUs*), particularly for large chemical/material systems.

4.6.4 Energy Consumption of CPUs, GPUs, and FPGAs

For the RT-TDDFTB electron dynamics performed on the GPU and FPGA, I measured the raw power by utilizing the NVIDIA management library and the Micron development kit [94, 2], respectively. For the GPU benchmarks, I utilized the correction scheme in Ref. [17] to give an accurate power estimation; in the CPU, the power was measured via the Likwid [144] suite. In my assessment of the FPGA and GPU platforms, the reported energy consumption does not include the energy consumed by the CPU since the majority of the computation (over 75%) was carried out on the co-processor (i.e., either

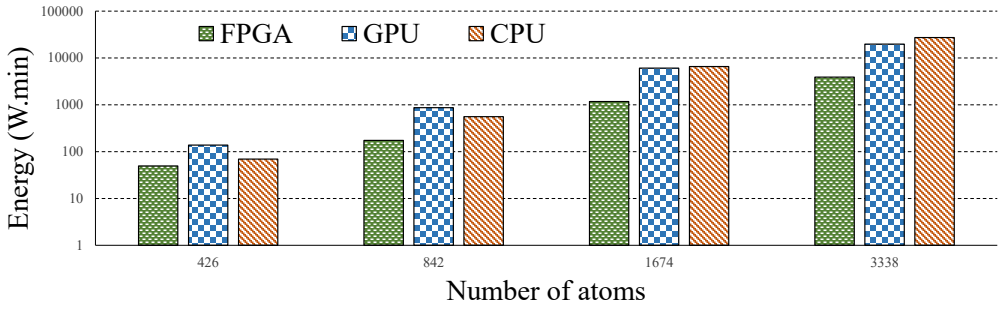


Figure 4.13: Comparison of energy consumption for FPGAs, GPUs, and CPUs.

the FPGA or GPU), and accounting for the power expenditures consumed by the CPU did not alter the observed trends.

Figure 4.13 compares the energy consumption in units of Watt-minutes for my experimental platforms. The total energy was calculated by integrating the power as a function of time that each hardware platform consumes per calculation. In the GPU, the power usage was limited to 150 W. As shown in Figure 4.13, the energy gap between the FPGA and GPU/CPU increases dramatically with the number of atoms in the system (note the logarithmic scale on the vertical axis). On average, the CPU and GPU consume 3.77 and 4.05 times more energy, respectively, than the FPGA, and this difference in energy efficiency is expected to further increase with system size. Similar to the computational speedups already described, these massive calculations can be executed in an energy-efficient manner since my FPGA implementation was specifically designed for the task at hand. In general, CPUs and GPUs are relatively expensive to power with an energy efficiency of 10 MOPS/nW, whereas FPGAs are about 5 times more cost effective with an efficiency of 50 MOPS/nW [55].

4.6.5 Performance on Recent FPGA Hardware Architectures

While my previous simulations were conducted on modern server-grade Virtex-7 FPGAS, I have migrated (synthesized, placed, and routed) my hardware design to a Virtex Ultrascale chip to forecast the gains in performance due to these newer hardware architectures. This migration is straightforward since the components used in my design are fully compatible with each other [37]. Specifically, (1) the Verilog modules used previously were directly implemented on this newer hardware, and (2) the Xilinx FPUs and block RAMs were migrated effortlessly since these units are forward compatible. With these relatively easy modifications, my RT-TDDFTB electron dynamics could be executed at 266 MHz on the *VU9P* Virtex UltraScale chip (when my design is routed in this device, 52.5%, 50.8%, and 53.3% of the LUTs, block RAM, and ultra-block RAM resources are used, respectively). In short, because my design is quite general and can be placed/routed in a newer FPGA operating at 266 MHz (compared to the 167 MHz of my current FPGA), my implementation has the capability to run even faster, with additional performance gains of 62%, even beyond the computational speedup observed in Figure 4.12.

4.7 Conclusion

In this chapter, I have presented the first application of field programmable gate arrays (FPGAs) for the fast and energy-efficient calculation of real-time electron dynamics in large chemical/material systems. Because FPGAs have not been used by the quantum dynamics community, I have provided a detailed description of my approach as a self-contained reference, followed with additional acceleration techniques tailored specifically

to the efficient propagation of RT-TDDFTB electron dynamics. To thoroughly test and understand the performance of the proposed FPGA engine, I have examined a variety of performance benchmarks that include single vs. double precision tests, computational speedup comparisons against GPUs/CPUs, detailed energy consumption measurements, and an assessment of performance gains on competing platforms.

My implementation allows the parallel execution of *wide* and *deep* pipelines tailored for the efficient execution of RT-TDDFTB electron dynamics. By offloading the most intensive and repetitive calculations into the FPGA, I show that the computational performance of my hardware implementation can surpass that of optimized commercial mathematical libraries running on high-performance GPUs and CPUs. In addition to this computational speedup, I show that FPGAs are energy-efficient and consume about four times less energy than the competing platforms. Moreover, FPGA performance has doubled in the last few years[128], and the implementation techniques and performance metrics demonstrated in this work indicate that FPGAs could also play a promising role in the acceleration (and energy-efficient calculation) of other types of quantum chemistry and materials science applications in the near future.

Chapter 5

Acceleration of the QR Decomposition of Tall-and-Skinny Matrices in FPGAs

5.1 Introduction

One of the fundamental problems in high performance computing is the fast decomposition of a matrix A into two or more factors. Notable algorithms include the Cholesky, LU, QR, SVD, spectral, singular value, and Schur decompositions [136]. In this chapter, I examine the design of energy-efficient high-throughput cores for the QR decomposition of tall-and-skinny matrices (TSMs); that is, matrices with a few hundred columns and tens of thousands of rows. The QR decomposition of TSMs has pervasive applications, with the most well-known being the solution of least squares problems [42]. In least squares,

the input matrices have a few hundred columns that correspond to the observed parameters, and thousands of rows, which represent the number of observations. In the field of video and image processing for stationary background subtraction [18], the input matrices have a few hundred columns that correspond to the given images and tens of thousands of rows representing the pixels. In the field of wireless communication [20], the input matrices have dozens of rows that correspond to the number of receiving antennas and a few columns that represent the number of transmitting antennas. Additional applications include communication-avoiding algorithms [30], the computation of eigenvalues [137], and the computation of Krylov subspaces [143].

While multiple studies [6, 97, 111, 108, 73] have addressed the design of efficient cores for the QR decomposition of matrices using methods such as Gram-Schmidt Orthogonalization (GS) (and its modifications), the Cholesky Decomposition (CH), and Givens Rotations (GR), the QR decomposition via Householder reflectors (HR) has, surprisingly, received less attention despite the fact the algorithm is numerically stable, parallelizable, and has a favorable computational complexity [58, 143]. Indeed, the HR decomposition method has been named as one of the ten most important algorithms of the last century [34].

In this chapter I propose an engine capable of decomposing TSMs in parallel with *resource-aware* reduction circuits [40, 158], thereby achieving a higher computational efficiency. Moreover, I take advantage of additional optimizations including tiling, double buffering, wide and deep pipelines, and memory burst accesses. I have implemented an HR decomposition core that targets the factorization of TSMs on a Micron SB-852 board [94]. The performance of the proposed core is compared against two configurations: (1) the QR

solver within the Intel MKL routines [145] running on an Intel quad-core processor, and (2) the QR solver in the CUDA basic linear algebra subroutines (CUBLAS) [104] running on a K40 GPU.

The contributions of my work are three-fold:

- I develop a *flexible* and *scalable* QR solver core that targets the decomposition of TSMs on FPGAs. By taking advantage of the numerical stability of the HR method, along with *resource-aware* reduction circuits, my design splits the input matrix into a series of blocks and executes the QR decomposition in parallel. Multiple parallel instances of this core are placed and routed on a Virtex UltraScale+ FPGA running at 266 MHz. This architecture can be easily scaled up or down for implementations on embedded or server-scale FPGAs. For the task at hand, my design achieves the highest efficiency (54%) compared to similar FPGA designs (36%) [114].

- The performance of the proposed engine matches and surpasses that of the Intel MKL QR solver [145], a highly optimized library, running on a quad-core CPU. For matrices having a few thousands rows, my engine matches the performance of the MKL QR solver. As the number of rows in the input matrix increases, my design outperforms the MKL QR solver by a factor of 1.5×. Compared to the performance of the CUBLAS QR solver [104] running on a GPU, my design achieves a speedup ranging from 1.5× to 3.0× for matrices having up to 256 columns. When the input matrix has 512 columns, my design closely follows the performance of the library running on the GPU; however, it executes more floating point operations (FLOPS) per Joule.

- My experimental results show that while CPUs and GPUs execute at most 0.45 and 0.60 GFLOPs/Joule, respectively, my design executes 1.03 GFLOPS/Joule. These gains in energy efficiency are obtained because the proposed engine uses *wide* and *deep* pipelines: when the input matrix has a few hundred columns, the proposed engine executes 2.3 × (12.4×) more FLOPS per clock cycle than the GPU (CPU).

5.2 QR Decomposition

QR decomposition factors a real valued matrix $\mathbf{A}_{n\times n}$ into two matrices, $\mathbf{Q}_{n\times n}$ and $\mathbf{R}_{n\times n}$, such that $\mathbf{A} = \mathbf{QR}$ with $\mathbf{Q}$ an orthogonal matrix ($\mathbf{QQ}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I}$) and $\mathbf{R}$ an upper triangular matrix ($\mathbf{R}_{i,j} = 0$ for $i > j$) [151, 143, 42]. When the input matrix $\mathbf{A}$ is nonsingular, the decomposition is unique, given that the diagonal elements of $\mathbf{R}$ are positive. More generally, when $\mathbf{A}$ is $m \times n$, with $m \geq n$, the decomposition is still possible with $\mathbf{Q}$ being an $m \times m$ matrix and $\mathbf{R}$ being an $m \times n$ matrix. Specifically,

$$\mathbf{A} = \mathbf{QR} = [\mathbf{Q}_1\ \mathbf{Q}_2] \begin{bmatrix} \mathbf{R}_1 \\ \\ \mathbf{0} \end{bmatrix} = \mathbf{Q}_1\mathbf{R}_1, \tag{5.1}$$

with $\mathbf{R}_1$ of size $n \times n$, $\mathbf{Q}_1$ of size $m \times n$, $\mathbf{Q}_2$ of size $m \times (m-n)$, and the matrix $\mathbf{0}$ having dimensions of $(m-n) \times n$.

5.2.1 QR Decomposition For TSMs

Although the QR decomposition can be applied to square matrices, my focus is on decomposing TSMs $\mathbf{A}_{m\times n}$ such that $m \gg n$ [30]. As shown in figure 5.1, the input matrix

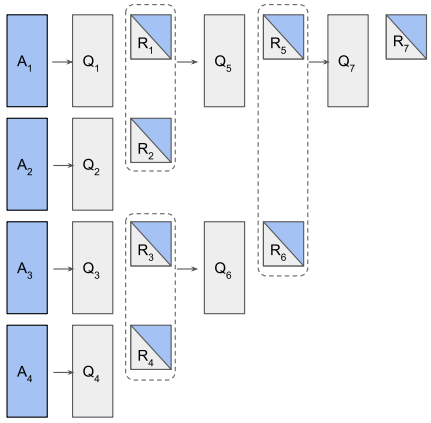


Figure 5.1: QR Decomposition for tall-and-skinny matrices (TSMs). This binary tree represents the QR decomposition of $\mathbf{A}$ such that $\mathbf{A}_i = \mathbf{Q}_i\mathbf{R}_i$ and $\binom{\mathbf{R}_j}{\mathbf{R}_{j+1}} = \mathbf{Q}_a\mathbf{R}_a$.

$\mathbf{A}$ of size $8n \times n$ is partitioned into four blocks $\mathbf{A}_1, ..., \mathbf{A}_4$, with $\mathbf{A}_i$ of size $2n \times n$. The QR decomposition is implemented in three steps.

1. Four processors decompose $\mathbf{A}_1, .., \mathbf{A}_4$ such that $\mathbf{A}_1 = \mathbf{Q}_1\mathbf{R}_1$, $\mathbf{A}_2 = \mathbf{Q}_2\mathbf{R}_2$, $\mathbf{A}_3 = \mathbf{Q}_3\mathbf{R}_3$, and $\mathbf{A}_4 = \mathbf{Q}_4\mathbf{R}_4$ are computed in parallel.
2. Two processors decompose $\mathbf{R}_1, ..., \mathbf{R}_4$ such that $\begin{pmatrix} \mathbf{R}_1 \\ \mathbf{R}_2 \end{pmatrix} = \mathbf{Q}_5\mathbf{R}_5$ and $\begin{pmatrix} \mathbf{R}_3 \\ \mathbf{R}_4 \end{pmatrix} = \mathbf{Q}_6\mathbf{R}_6$ are computed in parallel.
3. One processor decomposes $\mathbf{R}_5$ and $\mathbf{R}_6$ such that $\begin{pmatrix} \mathbf{R}_5 \\ \mathbf{R}_6 \end{pmatrix} = \mathbf{Q}_7\mathbf{R}_7$.

In this figure, each $\mathbf{Q}_i$ has a size of $2n \times n$ and $\mathbf{R}_i$ is $n \times n$. Notice that stage (b) takes the factors $\mathbf{R}_1, \mathbf{R}_2, \mathbf{R}_3$, and $\mathbf{R}_4$ from stage (a) as inputs. Likewise, stage (c) uses the factors $\mathbf{R}_5$ and $\mathbf{R}_6$ from stage (b). As a result, in this approach, one only requires the computation of matrices $\mathbf{R}_i$. Once $\mathbf{R}_7$ is found, finding matrix $\mathbf{Q}$ is immediate because $\mathbf{Q} = \mathbf{A}\mathbf{R}^{-1}$.

As described, the QR Decomposition of matrices can be achieved using four methods. The classical GS method is known to be numerically unstable due to rounding errors in finite precision arithmetic, although its instabilities can be removed via a modified approach [143]. Likewise, CH is known to be numerically unstable because the condition number of the matrix $\mathbf{A}\mathbf{A}^T$ is the square of the condition number of $\mathbf{A}$ [42]. On the other hand, GR and HR methods are known to be numerically stable given that certain practices are observed [42]. In this work, I investigate the performance of HR for decomposing TSMs in FPGAs. My decision is mainly based on the fact that this decomposition is numerically stable, and as a result, no additional hardware is dedicated to maintain its stability. Moreover, it has a lower computational complexity compared to the CH method [151, 42].

5.2.2 QR Decomposition Using Householder Reflections

Let $\boldsymbol{x} = (x_0, x_1, ..., x_{n-1})^T$ be a vector. The HR method [151, 143] transforms $\boldsymbol{x}$ into $\boldsymbol{y} = (y_0, 0, ..., 0)^T$ by constructing a matrix $\mathbf{Q}_{n\times n}$, usually called a Householder reflector, such that

$$\mathbf{Q}\begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ 0 \end{pmatrix}. \tag{5.2}$$

The HR decomposition transforms vector $\boldsymbol{x}$ into vector $\boldsymbol{y} = (-\sigma, 0, ..., 0)^T$ with $\sigma = \pm||\boldsymbol{x}||_2$. The matrix $\mathbf{Q}$ that achieves such a transformation is built as follows. Define vector $\boldsymbol{u}$ as

$$\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{y} = (x_0 + \sigma, x_1, ..., x_{n-1})^T \tag{5.3}$$

and the parameter γ as $\gamma = \frac{2}{||\boldsymbol{u}||_2^2}$. The matrix $\mathbf{Q}$ is given by

$$\mathbf{Q} = \mathbf{I} - \gamma \boldsymbol{u}\boldsymbol{u}^T, \tag{5.4}$$

where $\mathbf{I}_{n\times n}$ is the identity matrix. In addition to annihilating multiple elements in a vector, the HR decomposition can be used to calculate the $\mathbf{R}$ component in the QR decomposition $\mathbf{A} = \mathbf{QR}$. In this approach, the application of a set of Householder reflectors $\mathbf{Q}_1$,$\mathbf{Q}_2$,...,$\mathbf{Q}_n$ [151, 143] to matrix $\mathbf{A}$ leads to the computation of matrix $\mathbf{R}$. Algorithm I shows the canonical implementation of this decomposition.

Algorithm I - Canonical QR Decomposition of $\mathbf{A}$ size $m \times n$.

```
1   for k = 1 to n − 1 do
        // Step One - Generate Householder reflector
2     x_k = A(k : m, k)
3     d_1 = ddot(x_k, x_k)
4     d_2 = √d_1
5     u_k = x_k
6     u_k(1) = x_k(1) + sign(x_k(1))d_2
7     d_3 = d_1 u_k(1)
8     γ_k = −2/d_3
        // Step two - Update trailing columns of A
9     for j = k to n do
10      a_j = A(k : m, j)
11      d_4 = (γ_k) ddot(a_j, u_k)
12      t_j = axpy(a_j, d_4, u_k)
13      A(j : m, j) = t_j
14    end for
15  end for
```

In this routine, the operation ***ddot***$(\boldsymbol{x}, \boldsymbol{y})$ executes the dot-product between the arguments. Likewise, the operation ***axpy***$(\boldsymbol{x}, d, \boldsymbol{y})$ executes the vector subtraction $\boldsymbol{x} - (d)\boldsymbol{y}$. As shown in this routine, the canonical QR decomposition has two major steps: (1) the computation of the HR reflector, see Equation 5.4, and (2) the updating of the trailing

columns

$$\mathbf{Q}_j \boldsymbol{a}_i = \boldsymbol{a}_i - \gamma_j (\boldsymbol{u}_j^T \boldsymbol{a}_i) \boldsymbol{u}_j. \tag{5.5}$$

where $\boldsymbol{a}_i$ is a column of $\mathbf{A}$.

5.2.3 Householder Reflectors - Complexity Analysis

Now I analyze the computational complexity of applying the HR method to the decomposition shown in figure 5.1. Table 5.1 summarizes my analysis. In this analysis, the

Table 5.1: Computational complexity analysis

Task	Complexity
Householder Vector (A_i)	$3n^2 + n$
QA Mults + Adds (A_i)	$4n((5/6)n^2 + n + 1/6)$
Householder Vector (R_i)	$n^2 + 3n$
QR Mults + Adds (R_i)	$4n(n+1)(n/6 + 5/6)$

matrices A_i have a size of $2n \times n$ while the matrices R_i are $n \times n$. The computation of the first vector $\boldsymbol{u}_1$ requires at least $2n$ multiplications with $2n$ additions plus the computation of the square-root operation. Next, the vector $\boldsymbol{u}_2$ has to be computed for a $(2n-1) \times (n-1)$ sub-matrix. Thus, the computation of all the vectors $\boldsymbol{u}_i$ requires at least $\sum_{i=0}^{n-1}(2n - i) = (3/2)n^2 + n/2$ multiplications and an equal number of additions. The complexity of the application of matrices $Q_n, ..., Q_2, Q_1$ can be computed in a similar fashion. In the second case, parts (b) and (c) in figure 5.1, the computation of the vectors $\boldsymbol{u}_i$ is executed over columns of size 2 up to $n + 1$ so as to take advantage of the upper triangular matrices R, otherwise the calculations are as before.

5.2.4 QR Decomposition in CPUs and GPUs

The HR decomposition in CPUs and GPUs is typically implemented via blocks. In this approach, the input matrix $\mathbf{A}_{m\times n}$ is divided in tiles [65, 12], such that $\mathbf{A} = [\mathbf{A}_1\mathbf{A}_2...\mathbf{A}_k]$ with $\mathbf{A}_i$ of size $m \times r$ and $k = n/r$ an integer. The method is shown in Algorithm II.

Algorithm II - QR Decomposition in CPUs and GPUs

1		***for*** $j = 1$ ***to*** r ***do***
2	S1	$[\boldsymbol{u}, \gamma] = \mathbf{house}(\boldsymbol{A}_1(j:m, j))$
3	S2	$\boldsymbol{A}_1(j:m, j:r) = \boldsymbol{A}_1(j:m, j:r) - \gamma\boldsymbol{u}(\boldsymbol{u}^T\boldsymbol{A}_1(j:m, j:r))$
4	S2	$\boldsymbol{V}(:, j) = [\boldsymbol{zeros}(j-1, 1); \boldsymbol{u}]$; $\boldsymbol{B}(j) = \gamma$
5		***end for***
6	S3	$\boldsymbol{Y} = \boldsymbol{V}(:, 1)$; $\boldsymbol{W} = -\boldsymbol{B}(1) \cdot \boldsymbol{V}(:, 1)$
7		***for*** $j = 2$ ***to*** r ***do***
8	S3	$\boldsymbol{u} = \boldsymbol{V}(:, j)$
9	S3	$\boldsymbol{z} = -\boldsymbol{B}(j) \cdot \boldsymbol{u} - \boldsymbol{B}(j) \cdot \boldsymbol{W}(\boldsymbol{Y}^T\boldsymbol{u})$
10	S3	$\boldsymbol{W} = [\boldsymbol{W}\ \boldsymbol{z}]$; $\boldsymbol{Y} = [\boldsymbol{Y}\ \boldsymbol{u}]$
11		***end for***
12	S4	$\boldsymbol{A}(:, r+1:n) = \boldsymbol{A}(:, r+1:n) + \boldsymbol{Y}\boldsymbol{W}^T\boldsymbol{A}(:, r+1:n)$
13	S5	***execute step one***

This algorithm takes five steps. In step one (S1), the reflectors $\mathbf{Q}_1, \mathbf{Q}_2, ..., \mathbf{Q}_r$ for the tile $\mathbf{A}_1$ are computed. In step two (S2), the reflectors are applied to tile $\mathbf{A}_1$. In step three (S3), the reflectors $\mathbf{Q}_1, \mathbf{Q}_2, ..., \mathbf{Q}_r$ are transformed. In step four (S4), the reflectors are applied to the remaining tiles of $\mathbf{A}$. In the last step (S5), the previous steps repeat starting from $\mathbf{A}_2^{(1)}$, where $\mathbf{A}_2^{(1)} = \mathbf{Q}_1, \mathbf{Q}_2, ..., \mathbf{Q}_r\mathbf{A}_2$

As far as the computational complexity of this routine, I notice that S1 has a complexity proportional to $O(mr)$ since the execution time is dominated by the calculation of dot-products involving vectors of size m. The complexity of S2 is proportional to $O(r(mr))$ since each iteration operates over matrices of size $m \times r$. The complexity of S3 is proportional to $O(r(rm))$ due to the presence of the matrix-vector product Y^Tu in addition

to the product $W(Y^T u)$. Finally, the complexity of S4 is proportional to that of matrix multiplication.

5.3 Proposed Micro-architecture

In this section, I describe the design and implementation of the HR accelerator engine. The engine makes use of techniques to increase the performance in FPGAs, namely tiling, deep pipelines, double buffering, replication of pipelines, and memory bursting [25, 22].

5.3.1 Proposed Optimizations

My parallel-blocked approach optimizes the QR decomposition of TSMs via a set of optimization techniques including (a) parallel blocked decomposition, (b) tile QR decomposition, (c) efficient processing of the tiles via deep pipelines, (d) efficient processing of the tiles in the $\mathbf{R}_i$ blocks, (e) efficient computation of the dot-products, (f) efficient access to the off-chip memory, (g) and efficient use of FPGA resources. In the following, I describe each optimization.

(a) Parallel blocked QR decomposition. As shown in Figure 5.1, the QR decomposition of TSMs can be executed in parallel by multiple processing engines. In this regard, the decomposition of multiple blocks $\mathbf{A}_i$ in parallel is advantageous due to the large number of rows in the input matrix. Likewise, once the blocks $\mathbf{A}_i$ are decomposed, the decomposition of the blocks $\mathbf{R}_i$ in parallel is highly beneficial as the large number of rows in the input

matrix implies the presence of multiple levels in the decomposition tree. In my work, I decompose multiple blocks in parallel since the limiting factor is the availability of resources in the target device.

(b) Tiling the QR decomposition. At iteration j, as shown in Figure 5.2(a), the QR decomposition has to be applied to a $(2n - j) \times (n - j)$ block. Due to the iterative nature of the decomposition, (see Algorithm I), it is useful to store a large potion of this block in on-chip memory since storing the entire block is not feasible due to the limited memory resources on the FPGA. Thus, I tile the QR decomposition as shown in Figure 5.2 (b), (c), and (d).

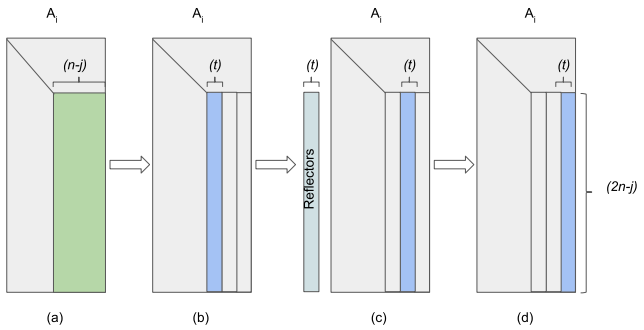


Figure 5.2: Tiling the QR decomposition. Instead of applying the QR Decomposition on blocks of size $(2n - j) \times (n - j)$ as shown is part (a), I partition the decomposition in tiles of size $(2n - j) \times t$. Next, I apply the QR decomposition to the left-most tile and save the reflectors as shown in part (b). Finally, I apply these reflectors to the remaining tiles as shown in parts (c) and (d).

In this figure, the maximum size of the tile is $2n \times t$ where $2n$ is the maximum number of rows in A_i and t is the number of columns in the tile. The QR decomposition using tiles involves two steps. In the first step, the QR decomposition is applied to the

most-left tile as shown in Figure 5.2 (b). This step involves the computation of t reflectors and the application of these reflectors to the t columns in the tile. While the first reflector $\mathbf{Q}_1$ is applied t times, the last reflector $\mathbf{Q}_t$ is applied once. In addition, these reflectors are saved into the on-chip memory. In the second step, the saved reflectors are applied to the remaining tiles as shown in Figure 5.2 (c) and (d). Here, each reflector is applied t times per tile. Notice that by adjusting t, I can tailor the decomposition in environments with copious, as well as scarce, on-chip memory resources.

(c) Efficient processing of the tiles via deep pipelines. While the canonical approach presented in Algorithm I assumes that one reflector $\boldsymbol{Q}$ is applied to each incoming vector $\boldsymbol{a}_j$ per iteration, in my work, I apply multiple reflectors [1] via deep pipelines. Figure 5.3 illustrates my approach when four reflectors are applied in a pipeline fashion.

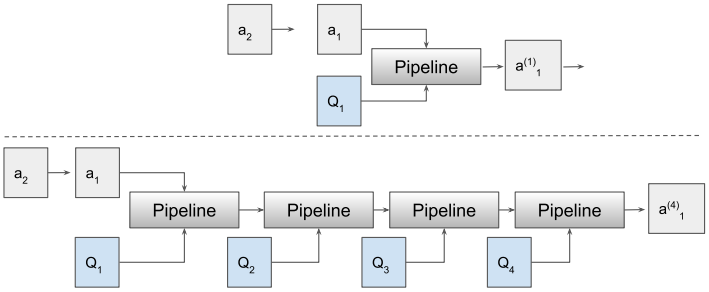


Figure 5.3: QR decomposition using Householder reflectors. At the top, the operation $\boldsymbol{a}_1^{(1)} = \mathbf{Q}_1\boldsymbol{a}_1$ is executed via a shallow pipeline. On the bottom, the operation $\boldsymbol{a}_1^{(4)} = \mathbf{Q}_4\mathbf{Q}_3\mathbf{Q}_2\mathbf{Q}_1\boldsymbol{a}_1$ is executed via a deep pipeline.

[1] While technically the word Householder reflector refers to the matrix $Q = I - \gamma \boldsymbol{u}\boldsymbol{u}^T$, in this work I use the word reflector to refer to the vector $\boldsymbol{u}$ also. The context of the discussion makes it clear if I am talking about Q or $\boldsymbol{u}$.

At the top of this figure, I apply reflector $\boldsymbol{Q}_1$ to all the columns of the current tile, one column at a time via a shallow pipeline. The result of the operation is $\mathbf{A}^{(1)} = \mathbf{Q}_1\mathbf{A}$. In the bottom part, I apply the reflectors $\mathbf{Q}_1$,..,$\mathbf{Q}_4$ to each column of the tile. This operation is illustrated in Equation 5.6.

$$\mathbf{A}^{(4)} = \mathbf{Q}_4\mathbf{Q}_3\mathbf{Q}_2\mathbf{Q}_1\mathbf{A} = (\mathbf{I} - \gamma_4\mathbf{u}_4\boldsymbol{u}_4^T)(\mathbf{I} - \gamma_3\mathbf{u}_3\boldsymbol{u}_3^T)(\mathbf{I} - \gamma_2\mathbf{u}_2\boldsymbol{u}_2^T)(\mathbf{I} - \gamma_1\mathbf{u}_1\boldsymbol{u}_1^T)\mathbf{A} \quad (5.6)$$

Notice that by applying multiple reflectors for each incoming vector, I can take advantage of the copious resources available in the FPGA namely BRAMs, DSPs, and LUTs. Furthermore, I also use this approach in the processing of the $\mathbf{R}_i$ blocks.

(d) Efficient processing of the tiles in the $\mathbf{R}_i$ blocks. As described in the canonical QR decomposition (see Algorithm *I*), the QR decomposition has two main steps: (1) the generation of the reflectors $\mathbf{Q}_1$, and (2) their application. When the inputs to the QR decomposition are the upper triangular matrices $\mathbf{R}_i$, further optimizations [30] for both steps are possible as shown in Figure 5.4.

At iteration j (see Figure 5.4(a)), the non-optimized HR decomposition works over tiles of size $((n-j)+n) \times t$. Because the elements below the diagonal in matrices $\mathbf{R}_1$ and $\mathbf{R}_2$ are zero, the computation of the reflector $\mathbf{Q}_j$ can be optimized as shown in the left-most tile in part Figure 5.4(b). The optimized HR decomposition works over tiles of size $(t + (j + t)) \times t$.

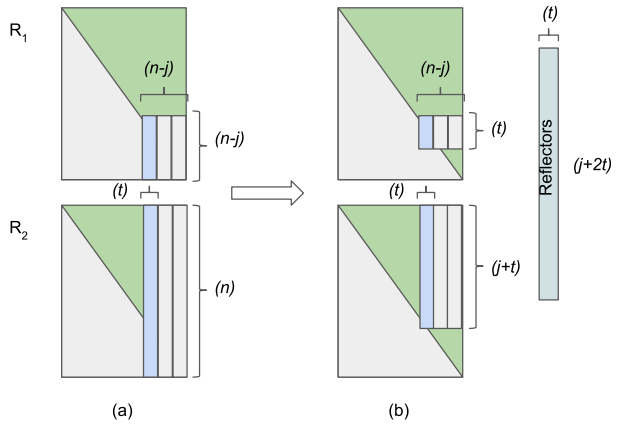


Figure 5.4: Iteration j of the QR decomposition for upper triangular matrices $\mathbf{R}_1$ and $\mathbf{R}_2$. On (a) the non-optimized QR decomposition, and on (b), the optimized QR decomposition.

(e) Efficient computation of dot-products. As shown in Algorithm I, the computation of the reflectors $\mathbf{Q}_i$, and their application, requires the efficient computation of dot-products. As the decomposition of the tiles in $\mathbf{A}_i$ advances, the size of these dot-products decreases. The size of the vectors involved in these dot-products goes from $2n$ to $t + n$. Likewise, as the decomposition of the tiles in $\mathbf{R}_i$ advances, the size of the vectors goes from $2t$ to $t + n$. As a result, it is important to implement a flexible circuit that easily adapts to these requirements. To meet these needs, I have implemented a *resource-aware* reduction circuit as described in [40]. This circuit uses two FIFOs, two multiplexers, one adder, one register, and one controller. In addition to being resource-aware, this circuit has a latency proportional to the size of the input.

(f) Efficient access to the off-chip memory. By inspection of Algorithm I, I observe that accessing the matrices $\mathbf{A}_i$, and $\mathbf{R}_i$, is in a column major fashion. Moreover, my benchmarks indicate that in the FPGA development environment [94], accessing off-chip

arrays via columns (when the arrays are stored in row major fashion) drastically reduces the performance of the I/O memory subsystem. Because such a low I/O performance (about 10% of the nominal peak performance) negatively impacts the performance of the decomposition, I transpose the input matrix in the host before sending it to the off-chip memory in the FPGA. In addition, the target coprocessor favors the access of 64-byte chunks of data aligned to the 64 memory channels addressed. As a result, I align the FPGA memory arrays to 64-bit addresses and access the off-chip memory using 64-byte chunks of data whenever possible.

(g) Efficient use of FPGA resources. In addition to having a resource-aware reduction circuit, I have taken other steps to minimize resource utilization. For example, to coordinate the execution of tasks between the modules, I make extensive use of small FIFOs, including one-bit FIFOs; i.e., signaling FIFOs. Finally, all floating-point operations are implemented via hard DSP cores to save hardware logic.

5.3.2 RTL Implementation

In this section I describe the RTL engines responsible for executing the QR decomposition of TSMs via HR. First, I introduce the processing element (PE) responsible for computing the HR and the PE responsible for applying these reflectors. Then, I introduce the architecture of my design.

Figure 5.5 shows the PE that computes Householder reflectors. This PE follows the steps described in Algorithm *I* regarding the generation of the Householder reflectors.

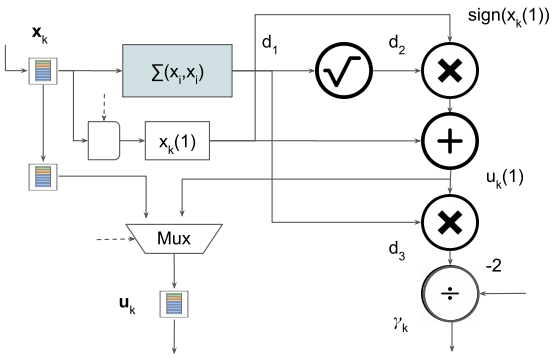


Figure 5.5: Processing element (PE) responsible for the computation of the Householder reflector $\mathbf{Q}_k$ i.e. the vector $\boldsymbol{u}_k$ along with the parameter γ_k.

The input to this PE is the vector $\boldsymbol{x}_k$, the top-left FIFO, and the output is the reflector $\boldsymbol{Q}_k$ (the vector $\boldsymbol{u}_k$ along with the parameter γ_k). In this figure, notice that the computation of $\sum_{i=0}^{i=n-1}(x_i)^2$ is via the *resource-aware reduction circuit* as described in section 5.3.1. Moreover, to facilitate the flow of data during the computations, this PE makes use of three FIFOs.

The PE responsible for applying the HR is shown in Figure 5.6. This PE follows

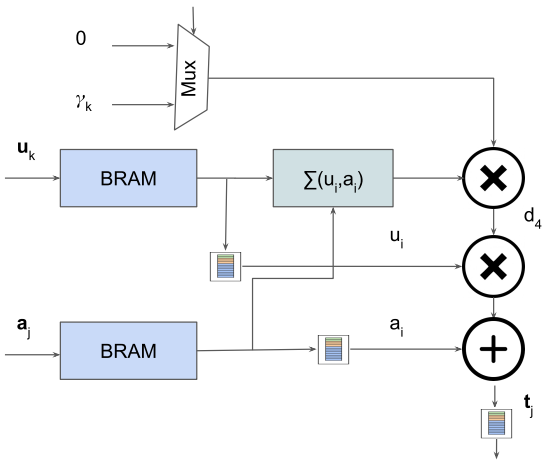


Figure 5.6: Processing element (PE) responsible for applying the Householder reflectors $\mathbf{Q}_k$ to an incoming vector $\boldsymbol{a}_j$ such that $\boldsymbol{t}_j = \mathbf{Q}_k \boldsymbol{a}_j$.

the steps described in Algorithm I regarding the updating of the trailing columns. The inputs to this PE are the reflector $\mathbf{Q}_k$ (the pair $\boldsymbol{u}_k$ and γ_k) and the target vector $\boldsymbol{a}_j$; the output is the transformed vector $\boldsymbol{t}_j$. As in the case of the previous PE, this PE makes use of a *resource-aware* reduction circuit as well as FIFOs. In this figure, notice that by setting the output of the multiplexer to zero, this PE can execute the identified operation i.e. $\boldsymbol{a}_j = \mathbf{Q}_k \boldsymbol{a}_j$.

The engine responsible for executing the QR decomposition is shown in Figure 5.7. This engine is made of four modules: *Scheduler*, *Reader*, *Cache*, and the *Writer* along with

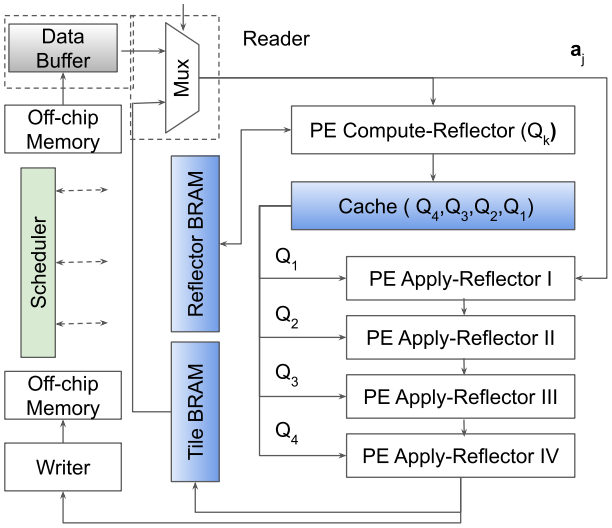


Figure 5.7: HR Decomposition engine, which computes the reflectors $\mathbf{Q}_k$ and applies these reflectors to the incoming vectors $\boldsymbol{a}_j$.

five PEs. To exchange messages between components, I use FIFOs [41], and the tile and reflector BRAMs store the data and the reflectors of the tile under decomposition. The *Scheduler* controls the execution of the aforementioned modules and PEs. At the beginning, the BRAM blocks and the caches are initialized with zero values.

As shown in Figure 5.2, the tile QR decomposition involves two steps. The computation of the reflectors for the current tile, and the application of these reflectors to the remaining tiles. To simplify the description, I assume that each tile has $2n \times 8$-sized elements such that each tile contains eight columns. In what follows, I explain the operation of this engine when it executes step one, and then, its operation when it executes step two. Regarding the first step, the computation of the reflector $\mathbf{Q}_1$ is as follows.

1. The *Scheduler* begins the execution by signaling to the *Reader* to read the first column of the current tile. The off-chip memory responses arrive in an orderly fashion to a FIFO within the *Reader* module.

2. The *Reader* makes two copies of the incoming vector: one copy goes to the PE *Compute-Reflector*, and the other goes to the PE *Apply-Reflector* I as shown in Figure 5.7.

3. The PE *Compute-Reflector* computes the reflector $\mathbf{Q}_1$ as shown in Figure 5.5. Moreover, this PE writes $\mathbf{Q}_1$ to the reflector BRAM. This BRAM unit has two dual-port blocks: one to store the vectors $\boldsymbol{u}_k$ and the other to store the parameters γ_k.

4. The *Cache* module is responsible for storing four reflectors. As explained above, I take advantage of these reflectors to build deep pipelines. When the reflector $\boldsymbol{u}_1$ arrives, this module writes it into a dual-port BRAM. Likewise, the parameter γ_1 is written into a register.

5. The PE *Apply-Reflector* I applies the reflector $\mathbf{Q}_1$, arriving from the *Cache*, to the incoming vector $\boldsymbol{a}_1$, arriving from the *Reader*, as depicted in Figure 5.6.

6. The PEs *Apply-Reflector* II, III, and IV execute the identity operation to the incoming vector. The output of the PE *Apply-Reflector* IV is vector $\mathbf{Q}_1\boldsymbol{a}_1$. Moreover, this module writes its output to the tile BRAM and to a FIFO within the *Writer*.

7. The *Writer* writes $\mathbf{Q}_1\boldsymbol{a}_1$ to the off-chip memory.

At this point, the reflector $\mathbf{Q}_1$ is on cache. Moreover, once the reflector $\mathbf{Q}_1$ is computed, the engine proceeds to compute the vector $\mathbf{Q}_1\boldsymbol{a}_2$.

1. This step is similar to step one described above, but this time, the engine reads the second column of the current tile.

2. The *Reader* copies the incoming vector to a FIFO within the PE *Apply-Reflector I*.

3. The PE *Apply-Reflector I* computes the vector $\mathbf{Q}_1\boldsymbol{a}_2$ by using, in addition, the reflector $\mathbf{Q}_1$ in the *Cache* module.

4. The PEs *Apply-reflector* II, III, and IV apply the identity operation over the incoming vector. Next, the output of the PEs *Apply-reflector* IV (i.e., the vector $\mathbf{Q}_1\boldsymbol{a}_2$), is written to the tile BRAM as well as the off-chip memory.

The computation of the reflector $\mathbf{Q}_2$ is as follows.

1. The *Reader* reads $\mathbf{Q}_1\boldsymbol{a}_2$ from the tile BRAM and copies this vector in a FIFO inside PE *Compute-Reflector* and to a FIFO inside the PE *Apply-Reflector* I.

2. The PE *Compute-Reflector* computes the reflector $\mathbf{Q}_2$. Furthermore, it stores this reflector into the respective BRAM and into the *Cache* module.

3. In parallel, the PE *Apply-Reflector* I outputs the vector $\mathbf{Q}_1\boldsymbol{a_2}$ by executing the identity operation.

4. The PE *Apply-Reflector* II applies reflector $\mathbf{Q}_2$ to the incoming vector $\mathbf{Q}_1\boldsymbol{a}_2$. Afterwards, the modules *Apply-Reflector* III and *Apply-Reflector* IV execute the identity operation over the incoming vector.

5. Finally, the resulting vector $\mathbf{Q}_2\mathbf{Q}_1\boldsymbol{a_2}$ is written into the tile BRAM and into the off-chip memory.

At this point, the reflectors $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are available in the *Cache* module as well as the reflector BRAM. The computation of the reflectors $\mathbf{Q}_3$ and $\mathbf{Q}_4$ is executed similarly. Once the reflectors $\mathbf{Q}_1$, $\mathbf{Q}_2$, $\mathbf{Q}_3$, and $\mathbf{Q}_4$ are computed, the engine applies these reflectors to the remaining four columns in the current tile. In this process, it reads one column at a time, and subsequently applies these four reflectors via a deep pipeline as shown in Figure 5.7.

Since the current tile has eight columns, one requires the computation of another set of reflectors. This process is as described above with the difference that the computation of reflectors starts at column five in the tile. Once these reflectors are computed, the decomposition of the current tile finishes. At this point, the reflector BRAM contains eight reflectors, and all results of the decomposition of the first tile are written to the off-chip memory. At this point, the first step finishes.

In the second step, the engine makes use of the eight reflectors stored in the BRAM and then applies them to the columns in the next tile, four reflectors at a time. In this

step, the PE *Compute-Reflector* does not computes reflectors, it only reads reflectors from the BRAM.

5.4 Experimental Results

My experimental work was carried out on the Wolverine II [94] co-processor series. All of my experimental work (placement, routing, and execution) utilized the SB-852VU7P version of the Wolverine II board. I have also placed and routed my design for execution on the SB-852VU9P version of that board. Table 5.2 compares these co-processors.

Table 5.2: Micron Wolverine II comparison

Feature	SB-852VU7P	SB-852VU9P
FPGA	VU7P	VU9P
- Registers	1576K	2364K
- Lookup Tables (LUT)	788K	1182K
- Block RAMs	1440	2160
- Block Ultra RAMs	640	960
- DSPs	4560	6840
Memory Channels	32	32
Off-chip Memory (DDR4)	64 GB	64 GB
Bandwidth	68 GB/s	68 GB/s
Frequency	266 MHz	266 MHz

As shown in the table above, these co-processors are very similar, and the main difference is the amount of FPGA resources per board. The first board uses a Xilinx VU7P FPGA, and the second board uses a Xilinx VU9P FPGA. As stated above, my experimental testbed consists of an Intel CPU E5-2460 with a SB-852VU7P board [94] . All the engines are implemented in Verilog, and were synthesized, placed, and routed Vivado 17.3 [37]. I address all timing errors until the design meets the timing requirements of 266 MHz, which is imposed by the Micron Wolverine II board design. My engines take double precision

floating point (DPFP) matrices as input, and all arithmetic operations are implemented on Xilinx DSP cores [52].

5.4.1 Area Utilization

The decomposition engines are replicated on the FPGA to process as many blocks in parallel as possible. Table 5.3 shows the resources required by these engines when they are placed and routed in each of the two co-processors. The SB-852VU7P and SB-852VU9P

Table 5.3: Area utilization per co-processor

Resource	SB-852VU7P (Total)	(%) Utilization (10 Engines)	SB-852VU9P (Total)	(%) Utilization (16 Engines)
Registers	1576K	61.11	2364K	54.4
Lookup Tables (LUT)	788K	71.09	1182K	63.0
LUT RAMs	394K	29.0	591K	24.7
Block RAMs	1.4K	65.4	2.2K	55.9
Ultra RAMs	640	18.75	960	20.0
DSPs	4.6K	21.8	6.8K	23.1
Memory Channels	32	62.5	32	100

co-processors can accommodate 10 and 16 engines respectively. In both cases, I place the on-chip tile blocks and reflector blocks in URAM memories (see Figure 5.7) due to their large capacity, and all other memory blocks are placed in conventional BRAMs. Each engine uses one channels for reads and another for writes. By doing so, I prevent stalls in the pipeline due to starvation of data (pending reads), or stalls due to the saturation of the output FIFOs (pending writes). The usage of LUT RAMs is mostly due to the presence of distributed FIFOs to coordinate the execution of operations.

5.4.2 Execution Times and Efficiency

I generate an $A_{m \times n}$ TSM with $m \gg n$ as shown in Figure 5.1. Each A_i of size $2n \times n$ (with $m/2n$ being an integer) is a non-singular uniformly distributed matrix.

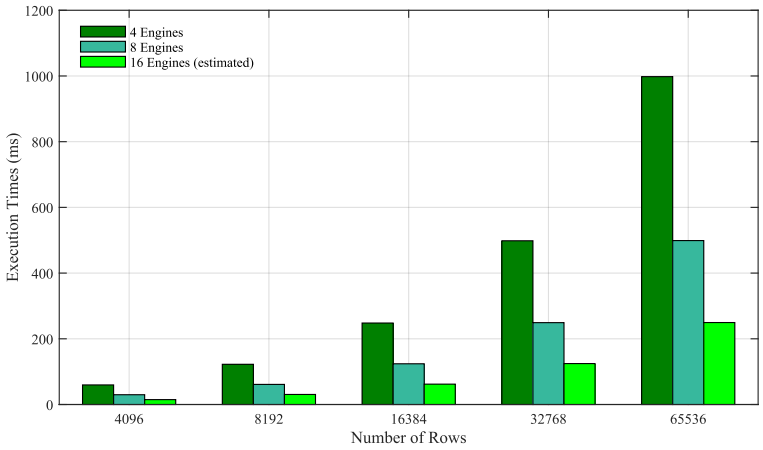


Figure 5.8: Execution times Vs. number of engines

Figure 5.8 shows the execution times for the QR decomposition of matrices having 256 columns and 4096 – 65536 rows, as computed on 4 to 16 engines. In this figure, the execution times for 4 and 8 engines are measured on the SB-852VU7P board. The execution times for 16 engines are an estimation based on the specifications of the SB-852VU9P board after placing and routing [94]. As shown in this figure, the execution times are inversely proportional to the number of engines. This result is expected because the engines are able to execute decomposed individual blocks independently and the amount of work per block is the same as shown in table 5.1. Because the performance of the proposed design is a *linear function* of the number of engines, for this point on, I only report the performance of the design for 16 engines.

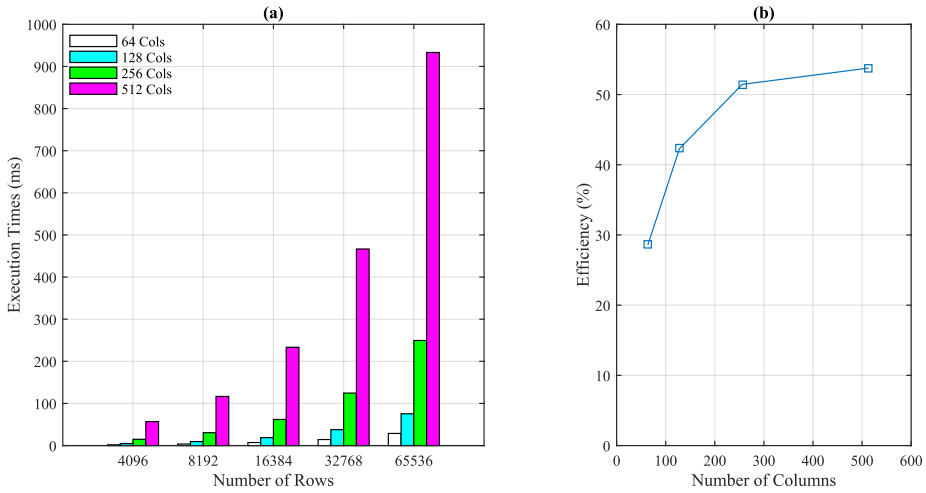


Figure 5.9: (a) Execution times and (b) efficiency of the QR decomposition via Householder reflectors for 16 engines.

Figure 5.9(a) shows the execution times when the number of engines is set to 16 while the number of columns increases from 64 to 512 and the number of rows from 4096 to 65536. In this figure, when the number of rows is fixed and the number of columns is increased, the execution time increases quadratically. Conversely, when the number of columns is fixed and the number of rows is increased, the execution time increases approximately linearly. In Figure 5.9(b), I show the efficiency of the engine (the ratio of the executed FLOPS and the nominal peak performance per clock cycle). Notice that in steady-state, each reflector executes four floating point operations simultaneously as shown in Figure 5.6. As a result, the 16 engines are able to execute a maximum of $256 = 16(4 \times 4)$ FLOPS per clock cycle. As shown in Figure 5.9(b), the efficiency of my design is a function of the size of the input matrices. My engine has a maximum efficiency of 54.2% when the matrices $\mathbf{A}_i$ have a size of 1024×512 and a minimum efficiency of 28.6% when the matrices have a size of 128×64. In these cases, the matrices $\mathbf{R}_i$ have sizes of 512×512 and 64×64, respectively.

5.4.3 Comparison with CPUs and GPUs

The CPU testbed consists of a workstation equipped with an Intel i7-3370 processor and 8 GB of RAM running the Intel MKL double precision QR solver [145]. The code is compiled with the gcc compiler version 7.4.0. In all CPU experiments, I use four threads, as the use of additional threads does not improve performance. The GPU testbed consists of a workstation with with an Intel E5-520 processor, 24 GB of RAM, and an NVIDIA $K40$ GPU [2]. The code is compiled with the CUDA compiler release 9.0 and the double-precision QR solver from the CUBLAS linear algebra library [104]. The frequency of the GPU is set to 562 MHz and the auto-boost feature as well as the error correction capabilities (ECC) are disabled. Table 5.4 compares the features of the accelerators.

Table 5.4: Comparison of the parameters of the three accelerators

Accelerator	Frequency	Peak GFLOPS/s	Cores
Intel i7-3370 CPU	3.4 GHz	108.8	4
NVIDIA K40 GPU	562 MHz	935.0	2,496
Micron SB-852VU9P	266 MHz	68.0	16

For the FGPA and the GPU, the data is first copied from the host to the accelerator local memory. Next, the QR solver is invoked, either in hardware or software. Finally, the resulting matrix is moved from the accelerator to the host for verification. The time to move the data to and from the accelerator local memory is not included in the execution time.

Figure 5.10 shows the execution times for the target platforms for matrices with 4096 - 65536 rows and 64 - 512 columns. From this figure, I notice that the FPGA and

[2] I use one of the $K40$ devices available within the $K80$ GPU.

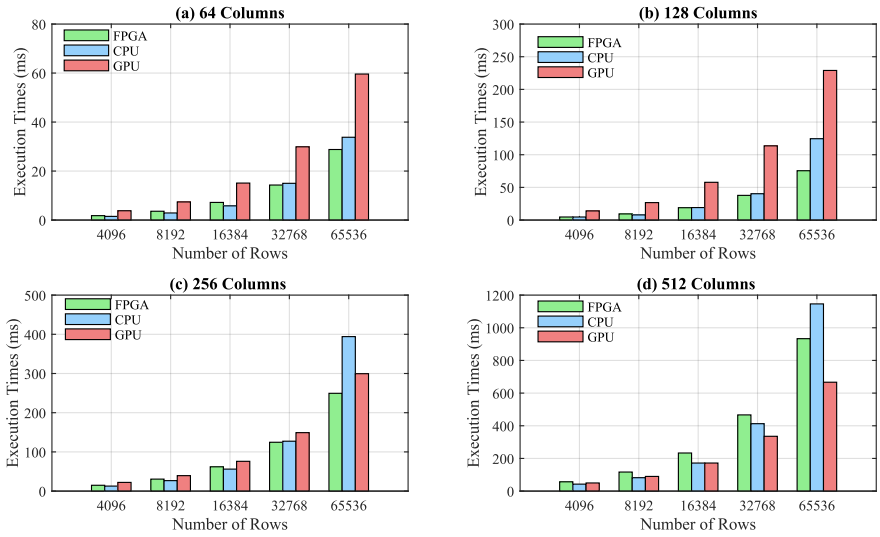


Figure 5.10: Execution times as a function of the number of rows for QR decomposition on FPGAs, GPUs, and CPUs.

the library running on the CPU are very fast when the input matrix has 256 columns and less, although the CPU library loses its edge for matrices having 256 columns and over 64K columns. For the cases of 64, 128, and 256 columns, the FPGA engine has a speedup of 2.0×, 3.0×, and 1.3× compared with the library running on the GPU. For the case of 512 columns, the routine running on the GPU edges the performance of the proposed FPGA engine. Moreover, the proposed engine and the CPU library have an equivalent performance for most of the cases, although for very tall matrices with $64K$ rows and more, the FPGA is faster by a factor of up to 1.5×.

The performance of the accelerators can be elucidated by analyzing the pipelines running in the FPGA as well as the QR solver running on the CPU and the GPU. In the FPGA, the QR decomposition of the tiles is divided into two steps as shown in Algorithm I. Because the computation of reflectors is serial; i.e., reflector $\mathbf{Q}_{i+1}$ has to be computed after reflector $\mathbf{Q}_i$ is available, the performance of the first step is limited by its sequential nature.

In addition, the computation of the reflectors does not favor high performance because the cost of this calculation is dominated by the dot-products as shown in Equation 5.3. Second, the application of the reflectors favors deep (the number of reflectors applied) and wide pipelines (the number of running engines); and as a result, higher performance is possible. In short, when the input matrix has a low number of columns, the loss in performance of the first step limits the overall performance of my design. Otherwise, as the number of columns increases, the performance of my design increases as well.

In the CPU and the GPU, the QR decomposition can also be divided roughly in two major steps, namely the computation of the reflectors (S1, S2, and S3) and their application (S4) as shown in Algorithm II. Because the computation of the reflectors (S1), the application of the reflectors to the current tile (S2), and their transformation (S3) are serial in nature, the first step has limited performance. Moreover, the second step is dominated by the matrix products of the form $(I + YW^T)A^{(i)}$ and, as a result, greater performance is achieved due to the highly optimized matrix multiplication routines available on the CPU and the GPU [28, 149].

In addition, for TSMs, I attribute the rather low performance of the QR solver on the CPUs and GPUs to the existence of trade-offs in the implementation of Algorithm II. Regarding this routine, setting the parameter r has broad consequences. If r is small, the algorithm does little progress per iteration because the resulting number of tiles is large. The large number of tiles implies that steps S1, S2, and S3 have to be executed multiple times. Moreover, step S4 suffers because the multiplication of matrices has to be executed over small matrices [68, 28], namely W and Y. If r is larger, it enhances the performance

of step S4 at the cost of increasing the execution times of the other steps. Moreover, my experiments indicate that typical values of r are 8, 16, and 32. In GPUs, larger values of r are not practical due to the limited capacity of shared memories [68].

5.4.4 Operations per Clock Cycle and Efficiency

I measure the number of FLOPs executed per clock cycle as well as the efficiency of the platforms as shown in Figure 5.11. In this figure, I only report the FLOPS per

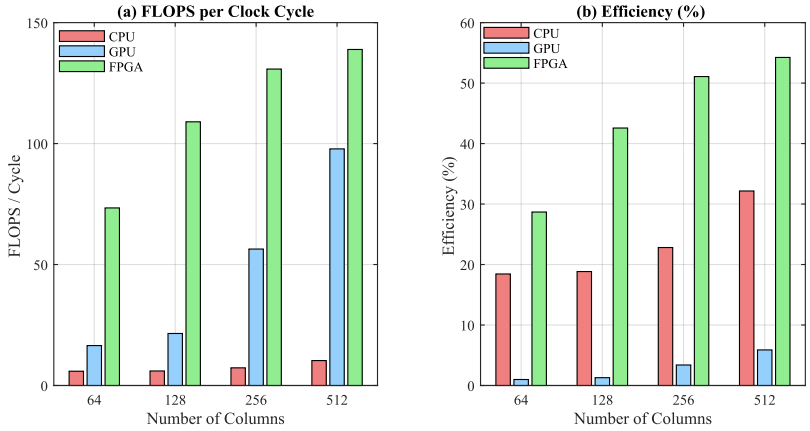


Figure 5.11: Double-precision floating point operations (FLOPS) per clock cycle for CPUs, GPUs, and FPGAs and their efficiency.

clock cycle and efficiency for 65536 rows; these metrics are nearly the same for 32768 rows and lower. I observe that in my design, and the GPU, there is a sustained increase in the number of FLOPS per clock cycle as the number of columns in the input matrix increases from 64 to 512. In short, for 512, 256, 128, and 64 columns, the proposed engine executes 1.4×, 2.3×, 5.0×, and 4.4× more FLOPS per cycle compared to the other platforms. In addition, in terms of efficiency (i.e., the achieved performance divided by the nominal peak performance), my engine comes out first as it is able to achieve 54.2% of the nominal peak

performance. The libraries running on the CPU (GPU) achieve 32.1% (5.8%) of the peak performance of the hosting platform.

I note that the proposed design is placed and routed at 266 MHz since the interface to the off-chip memory in the development board is hardened at this frequency. Because I use standard Xilinx cores, namely floating point cores, FIFOS, and BRAMs, I rationalize that my design can be placed and routed at higher frequencies with minimum effort. In particular, memory interfaces running at higher frequencies have been available on the market for a while [134, 45].

5.4.5 Energy Efficiency

Lastly, I compare the energy efficiency in (FLOPS/Joule) for each platform. For the CPU, GPU, and FPGA, I measure the raw power by taking advantage of the LIKWID monitoring tools [144], the NVIDIA management library [106], and the Convey development kit [94] respectively [3]. In the CPU and GPU, I measure the FLOPS per each task by taking advantage of hardware counters [107, 105]. In the FPGA, I analytically derive the operations executed by the engine. On the GPU, once the power data is obtained, corrections are made so as to have an accurate power estimation [17]. Figure 5.12 shows the energy efficiency per platform. I only report the energy efficiency when the number of rows is fixed at 65536. In all cases, the energy efficiency of the FPGA is higher compared to other platforms. Compared to the GPU, the engine running on the FPGA is 5.4×, 7.7×, 2.8×, and 2.3× more energy efficient when the matrices have 64, 128, 256, and 512 columns respectively. Compared

[3] In the case of the FPGA, I measure the power consumption of 10 engines (the SB-852VU7P), and then, I extrapolate the power consumption to 16 engines (the SB-852VU9P).

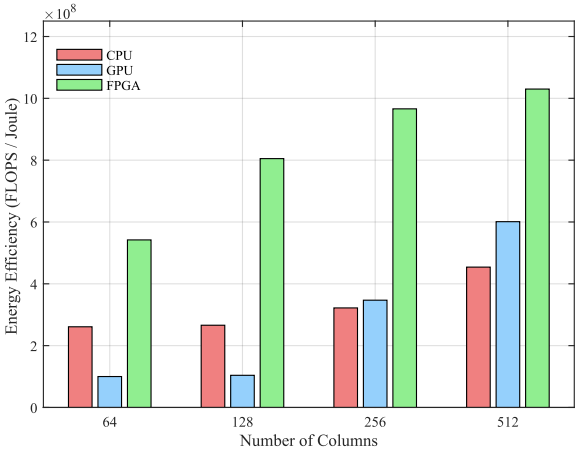


Figure 5.12: Energy efficiency for CPUs, GPUs, and FPGAs.

to the CPU, the proposed engine is 2.0×, 3.0×, 3.0×, and 2.3× more energy efficient for the same task. It should be noted that FPGA technological capabilities continue to increase [128] in both clock frequency and available on-chip resources (memory, DSP, etc.). As such, the expected performance of FPGA-based accelerators is expected to increase even further with the added benefit of energy efficiency [55].

5.4.6 Conclusions

In this work I propose a high-throughput FPGA engine capable of executing the QR decomposition of *tall-and-skinny* matrices (TSMs). My design is based on the highly stable, parallelizable, and low complexity Householder (HR) decomposition method. The HR engine takes advantage of a series of performance optimizations including tiling, wide and deep pipelines, resource-aware reductions circuits, as well as fast access to off-chip memory. Due to these optimizations, my engine achieves the *highest computational efficiency* compared to previous studies: while previous approaches achieve up to 36% efficiency, my design achieves an efficiency of 54%. Because my design uses *resource-aware circuits*, it can

be used to tackle the QR decomposition of the full spectrum of *tall-and-skinny* matrices, including those with hundreds of columns and matrices with tens of thousands of rows. Moreover, by tailoring the number of engines in execution, as well as the number of Householder reflectors applied, the proposed engine can be implemented in embedded as well as server-grade FGPAs.

My experimental evaluation shows that the proposed engine outperforms the MKL solver on an Intel Quad-Core processor by a factor of 1.5× when the input matrices have over $50K$ rows. For matrices having up to 256 columns, my engine outperforms the QR solver running on the K40 GPU by a factor of 3.0×. An evaluation of the energy efficiency of these three platforms shows that CPUs and GPUs execute up to 0.45 and 0.60 GFLOPS/Joule respectively, while my design executes up to 1.03 GFLOPS / Joule. This energy efficiency is due to the use of highly efficient *deep* and *wide* pipelines executing over a hundred of FLOPS per clock cycle.

Chapter 6

Conclusions

In the new century, the field of high performance computing has witnessed the birth of computational devices (CPUs and GPUs) executing billions of arithmetic operations per second. To achieve such impressive performance, the high performance community has depended on very high operating frequencies in conjunction with power-hungry solutions such as large cache units, colossal branch predictors, and complex datapaths. High frequencies and power-hungry units are the leading factors towards the *massive* increase of the operational cost in data-centers due to the *extensive* use of energy.

Simultaneously, the field has also observed the birth, expansion, and consolidation of computing applications based on field programmable gate arrays (FPGA). While early FPGAs did not have enough resources and speed to compete with traditional computing platforms (CPUs and GPUs), today these devices can execute *billions of operations per second* and operate at *high frequencies*. In addition, due to the custom designs of the reconfigurable pipelines, FPGAs have been shown to be *energy-efficient*.

This dissertation has shown how to build high-throughput, energy-efficient, compute-intensive applications on FPGAs to partially offset the *massive* operational cost due to energy consumption in traditional computing platforms. I have shown that *wide* and *deep* pipelines running in FGPAs can achieve comparable performance to those of traditional computing platforms, and that these designs are capable of executing more operations per unit of energy.

In chapter three, I proposed a heterogeneous approach based on an extensive algorithmic and experimental analysis of the human action recognition (HAR) application. My results showed that my heterogeneous implementation where the video pre-processing is implemented on the FPGA and the remaining stages are implemented on the GPU, achieves the highest throughput and energy efficiency. The heterogeneous design combines the strengths of both FPGA and GPU platforms, and achieves a 1.3× speedup compared with competing homogeneous platforms while being 1.5× more energy-efficient.

In chapter four, I presented the first application of FPGAs to the field of quantum dynamics simulations. By taking advantage of the structure of the input matrices and offloading the most intensive calculations onto an FPGA, I showed that the computational performance of my design for real-time electron dynamics calculations exceeds that of highly optimized commercial libraries running on recent CPUs and GPUs. For quantum simulations having over a thousands of atoms, my engine is 1.5× faster while consuming 4.0× less energy. As a result, the proposed engine demonstrates that high-throughput energy-efficient designs based on FPGAs can play an important role in the acceleration of applications in the upcoming field of quantum dynamics.

In chapter five, I presented a high-throughput engine that targets the decomposition of tall-and-skinny matrices (TSM) on FPGAs. While comparable QR solvers based on FPGAs achieve an efficiency of 36%, my design has an efficiency of 54%. In addition, my experimental work showed that the proposed design outperforms highly optimized QR solvers running on CPUs and GPUs. For TSM having over 50K rows, my design outperforms a highly optimized QR solver running in an quad-core processor by a factor of 1.5. In addition, for TSMs having 256 columns and less, my design outperforms a commercial QR solver library running in a high-performance GPU by a factor of 3.0. On top of being high performance, my design is energy-efficient; it executes twice as many floating point operations per unit of energy.

Bibliography

[1] Ahmed Al Maashri, Michael Debole, Matthew Cotter, Nandhini Chandramoorthy, Yang Xiao, Vijaykrishnan Narayanan, and Chaitali Chakrabarti. Accelerating neuromorphic vision algorithms for recognition. In *Design Automation Conference (DAC)*, pages 579–584. IEEE, 2012.

[2] Jason H. Anderson and Farid N. Najm. Power estimation techniques for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(10):1015–1027, October 2004.

[3] Joshua A. Anderson, Chris D. Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of computational physics*, 227(10):5342–5359, 2008.

[4] Michael Anderson, Grey Ballard, James Demmel, and Kurt Keutzer. Communication-avoiding QR decomposition for GPUs. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 48–58, Anchorage, AK, USA, 2011. IEEE.

[5] Balint Aradi, Ben Hourahine, and Th Frauenheim. DFTB+, A sparse matrix-based implementation of the DFTB method. *The Journal of Physical Chemistry A*, 111(26):5678–5684, July 2007.

[6] Semih Aslan, Sufeng Niu, and Jafar Saniie. FPGA implementation of fast QR decomposition based on Givens rotation. In *IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 470–473, Boise, ID, USA, 2012. IEEE.

[7] Christian L. Belady. In the data center, power and cooling costs more than the it equipment it supports. `https://www.electronics-cooling.com/`, February 2007.

[8] Faycal Bensaali, Abbes Amira, and Reza Sotudeh. Floating-point matrix product on FPGA. In *2007 IEEE/ACS International Conference on Computer Systems and Applications*, pages 466–473. IEEE, 2007.

[9] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Steven Scarpelli,

Al an Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[10] Piotr Bialas and Adam Strzelecki. Benchmarking the cost of thread divergence in CUDA. In *International Conference on Parallel Processing and Applied Mathematics*, pages 570–579. Springer, 2015.

[11] Berkin Bilgic, Berthold K.P. Horn, and Ichiro Masaki. Efficient integral image computation on the GPU. In *2010 IEEE Intelligent Vehicles Symposium*, pages 528–533. IEEE, 2010.

[12] Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1987.

[13] Akkarat Boonpoonga, Sompop Janyavilas, Phaophak Sirisuk, and Monai Krairiksh. FPGA implementation of QR decomposition using MGS algorithm. In *International Symposium on Applied Reconfigurable Computing*, pages 394–399, Bangkok, Thailand, 2010. Springer.

[14] David J. Brown and Charles Reams. Toward energy-efficient computing. *Communications of the ACM*, 53(3):50–58, 2010.

[15] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *ACM Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, 2009.

[16] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012.

[17] Martin Burtscher, Ivan Zecena, and Ziliang Zong. Measuring GPU power with the K20 built-in sensor. In *ACM Proceedings of Workshop on General Purpose Processing Using GPUs*, pages 28–36, Salt Lake City, UT, USA, 2014. ACM.

[18] Emmanuel J. Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *Journal of the ACM (JACM)*, 58(3):11, 2011.

[19] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[20] Abha Chauhan and Rajesh Mehra. Analysis of QR decomposition for MIMO systems. In *IEEE International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pages 69–73, Nagpur, India, 2014. IEEE.

[21] Shuai Che, Bradford M. Beckmann, Steven K Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195. IEEE, 2013.

[22] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with GPUs and FPGAs. In *IEEE Symposium on Application Specific Processors*, pages 101–107, Anaheim, CA, USA, 2008. IEEE.

[23] Jack Choquette, Olivier Giroux, and Denis Foley. Volta: Performance and programmability. *IEEE Micro*, 38(2):42–52, 2018.

[24] Pong P. Chu. *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*. John Wiley & Sons, 2011.

[25] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of FPGAs and GPUs. In *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, Boulder, CO, USA, 2018. IEEE.

[26] Gabriella Csurka, Christopher Dance, Lixin Fan, Jutta Willamowski, and Cédric Bray. Visual categorization with bags of keypoints. In *Workshop on Statistical Learning in Computer Vision (ECCV)*, volume 1, pages 1–2. Prague, 2004.

[27] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *ACM Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[28] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing SHOC benchmark suite. In *ACM Proceedings of the 3st Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, New York, NY, USA, 2010. ACM.

[29] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):732–794, 2015.

[30] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[31] Piotr Dollár, Vincent Rabaud, Garrison Cottrell, and Serge Belongie. Behavior recognition via sparse spatio-temporal features. In *Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance*, pages 65–72. IEEE, 2005.

[32] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. DeCAF: A deep convolutional activation feature for generic visual recognition. In *arXiv preprint*, pages 647–655, 2013.

[33] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2625–2634, 2014.

[34] Jack Dongarra and Francis Sullivan. Guest editors' introduction: The top 10 algorithms. *IEEE Computing in Science & Engineering*, 2(1):22, 2000.

[35] Yong Dou, Stamatis Vassiliadis, Georgi Krasimirov Kuzmanov, and Georgi Nedeltchev Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 86–95. ACM, 2005.

[36] Christoph Feichtenhofer, Axel Pinz, and Andrew Zisserman. Convolutional Two-Stream network fusion for video action recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1933–1941, 2016.

[37] Tom Feist. Vivado design suite, 2012. https://www.xilinx.com/support/documentation.

[38] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*, pages 3757–3760. IEEE, 2010.

[39] Konstantinos Georgopoulos, Iakovos Mavroidis, Luciano Lavagno, Ioannis Papaefstathiou, and Konstantin Bakanov. Energy-efficient heterogeneous computing at exaSCALE—ECOSCALE. In *Hardware Accelerators in Data Centers*, pages 199–213. Springer, 2019.

[40] Marco Gerards, Jan Kuper, André Kokkeler, and Bert Molenkamp. Streaming reduction circuit. In *12th IEEE Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 287–292, Patras, Greece, 2009. IEEE.

[41] Kahn Gilles. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[42] Gene H. Golub and C.F. Van Loan. *Matrix computations*. The Johns Hopkins University Press, Baltimore, MD, USA, 2013.

[43] Andreas W. Gotz, Mark J. Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross C. Walker. Routine microsecond molecular dynamics simulations with AMBER on gpus. 1. Generalized born. *Journal of chemical theory and computation*, 8(5):1542–1555, 2012.

[44] Mentor Graphics. Modelsim. Advanced simulation and debugging. `https://www.mentor.com/`, 2012.

[45] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proceedings of the 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 162–170, New York, NY, USA, 2004. ACM.

[46] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, and Jack Dongarra. Tile QR factorization with parallel panel processing for multicore architectures. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10, Atlanta, GA, USA, 2010. IEEE.

[47] Michael Hahnle, Frerk Saxen, Matthias Hisung, Ulrich Brunsmann, and Konrad Doll. FPGA-based real-time pedestrian detection on high-resolution images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 629–635. IEEE, 2013.

[48] Robert J. Halstead, Jason Villarreal, and Walid Najjar. Exploring irregular memory accesses on FPGAs. In *ACM Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*, pages 31–34. ACM, 2011.

[49] Elizabeth Kopits Heather Klemick and Ann Wolverton. Data center energy efficiency investments: Qualitative evidence from focus groups and interviews. `https://www.epa.gov/sites/production/files/2017-11/documents/2017-06_0.pdf`, November 2017.

[50] John L. Hennessy and David A. Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[51] Samitha Herath, Mehrtash Harandi, and Fatih Porikli. Going deeper into action recognition: A survey. *Image and Vision Computing, Elsevier*, 60:4 – 21, 2017.

[52] Tom Hill. Xilinx DSP design platforms: Simplifying the adoption of FPGAs for DSP, 2009. https://www.xilinx.com/support/documentation.

[53] Manato Hirabayashi, Shinpei Kato, Masato Edahiro, Kazuya Takeda, Taiki Kawano, and Seiichi Mita. GPU implementations of object detection using HOG features and deformable models. In *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 106–111. IEEE, 2013.

[54] Kim Ho-Joon, Joseph S. Lee, and Yang Hyun-Seung. Human action recognition using a modified convolutional neural network. In *International Symposium on Neural Networks*, pages 715–723. Springer, 2007.

[55] Mark Horowitz. Computing's energy problem (and what we can do about it). In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, San Francisco, CA, USA, 2014. IEEE.

[56] Mark Horowitz. Computing's energy problem (and what we can do about it). In *2014 IEEE International Conference on Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014.

[57] Zuoxun Hou, Hongbo Zhu, Nanning Zheng, and Tadashi Shibata. A single-chip 600-fps real-time action recognition system employing a hardware friendly algorithm. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 762–765. IEEE, 2014.

[58] Alston S. Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342, 1958.

[59] Niranjan V. Ilawe, M. Belén Oviedo, and Bryan M. Wong. Real-time quantum dynamics of long-range electronic excitation transfer in plasmonic nanoantennas. *Journal of chemical theory and computation*, 13(8):3442–3454, 2017.

[60] Niranjan V. Ilawe, M. Belén Oviedo, and Bryan M. Wong. Effect of quantum tunneling on the efficiency of excitation energy transfer in plasmonic nanoparticle chain waveguides. *Journal of Materials Chemistry C*, 6(22):5857–5864, 2018.

[61] Hueihan Jhuang, Thomas Serre, Lior Wolf, and Tomaso Poggio. A biologically inspired system for action recognition. In *11th International Conference on Computer Vision (ICPR)*, pages 1–8. IEEE, 2007.

[62] Ž. Jovanović and V. Milutinović. FPGA accelerator for floating-point matrix multiplication. *IET Computers & Digital Techniques*, 6(4):249–256, 2012.

[63] Ryoji Kadota, Hiroki Sugano, Masayuki Hiromoto, Hiroyuki Ochi, Ryusuke Miyamoto, and Yukihiro Nakamura. Hardware architecture for HOG feature extraction. In *Fifth IEEE International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 1330–1333. IEEE, 2009.

[64] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1725–1732, 2014.

[65] Andrew Kerr, Dan Campbell, and Mark Richards. QR decomposition on GPUs. In *ACM Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 71–78, New York, NY, USA, 2009. ACM.

[66] Steve Kilts. *Advanced FPGA design: Architecture, implementation, and optimization.* John Wiley & Sons, 2007.

[67] David B. Kirk and Wen-Mei W. Hwu. *Programming massively parallel processors: A hands-on approach.* Morgan Kaufmann, Cambridge, MA, USA, 2016.

[68] David B. Kirk and Wen-Mei W. Hwu. *Programming massively parallel processors: A hands-on approach.* Morgan kaufmann, 225 Wyman Street, Waltham, MA, 02451, USA, 2016.

[69] Alexander Klaser, Marcin Marszalek, and Cordelia Schmid. A spatio-temporal descriptor based on 3D-gradients. In *19th British Machine Vision Conference*, pages 275:1–10, Leeds, United Kingdom, September 2008.

[70] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, pages 1097–1105. Curran Associates Inc., 2012.

[71] Hilde Kuehne, Hueihan Jhuang, Rainer Stiefelhagen, and Thomas Serre. HMDB51: A large video database for human motion recognition. In *High Performance Computing in Science and Engineering*, pages 571–582. Springer, 2013.

[72] Vinay B.Y. Kumar, Siddharth Joshi, Sachin B. Patkar, and H. Narayanan. FPGA based high performance double-precision matrix multiplication. In *ACM Proceedings of the 2009 22nd International Conference on VLSI Design*, volume 38, pages 341–346. ACM, 2009.

[73] Martin Langhammer and Bogdan Pasca. High-performance QR decomposition for FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 183–188, Monterey, CA, USA, 2018. ACM.

[74] Ivan Laptev, Marcin Marszalek, Cordelia Schmid, and Benjamin Rozenfeld. Learning realistic human actions from movies. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8. IEEE, 2008.

[75] Abhijeet G. Lawande, Alan D. George, and Herman Lam. Novo-g#: a multidimensional torus-based reconfigurable cluster for molecular dynamics. *Concurrency and Computation: Practice and Experience*, 28(8):2374–2393, 2016.

[76] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[77] Christopher T. Lee and Rommie E. Amaro. Exascale computing: A new dawn for computational biology. *IEEE Computing in science & engineering*, 20(5):18–25, 2018.

[78] Laurent Lefèvre and Jean-Marc Pierson. Introduction to special issue on sustainable computing for ultrascale computing. *ScienceDirect*, 17:25–26, 2018.

[79] James T. Letendre. *Understanding and modeling the synchronization cost in the GPU architecture*. PhD thesis, Rochester Institute of Technology, 2013. https://scholarworks.rit.edu/.

[80] Walter B. Ligon, Scott McMillan, Greg Monn, Kevin Schoonover, Fred Stivers, and Keith D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, pages 206–215. IEEE, 1998.

[81] Kirt Lillywhite, Dah-Jye Lee, and Dong Zhang. Real-time human detection using histograms of oriented gradients on a GPU. In *Workshop on Applications of Computer Vision*, pages 1–6. IEEE, 2009.

[82] Jingen Liu, Jiebo Luo, and Mubarak Shah. Recognizing realistic actions from videos in the wild. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1996–2003. IEEE, 2009.

[83] Jingen Liu, Yang Yang, and Mubarak Shah. Learning semantic visual vocabularies using diffusion distance. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 461–468. IEEE, 2009.

[84] Weifeng Liu and Brian Vinter. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350. ACM, 2015.

[85] David G. Lowe. Object recognition from local scale-invariant features. In *The proceedings of the Seventh International Conference on Computer Vision*, volume 2, pages 1150–1157. IEEE, 1999.

[86] Nathan Luehr, Ivan S. Ufimtsev, and Todd J. Martinez. Dynamic precision for electron repulsion integral evaluation on graphical processing units (gpus). *Journal of Chemical Theory and Computation*, 7(4):949–954, 2011.

[87] Xiaoyin Ma, Walid Najjar, and Amit K. Roy-Chowdhury. Evaluation and acceleration of High-Throughput Fixed-Point object detection on FPGAs. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(6):1051–1062, 2015.

[88] Xiaoyin Ma, Jose M. Rodriguez-Borbon, Walid Najjar, and Amit K. Roy-Chowdhury. Optimizing hardware design for human action recognition. In *26th IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–11. IEEE, 2016.

[89] Krishna T. Malladi, Frank A. Nothaft, Karthika Periyathambi, Benjamin C. Lee, Christos Kozyrakis, and Mark Horowitz. Towards energy-proportional datacenter memory with mobile DRAM. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 37–48. IEEE, 2012.

[90] R.K. McConnell. Method of and apparatus for pattern recognition, 1986. US Patent 4,567,610.

[91] Paul Messina. The exascale computing project. *IEEE Computing in Science & Engineering*, 19(3):63–67, 2017.

[92] Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Top 500 list. `http://www.top500.org/`, 2012.

[93] Micron. Convey PDK reference manual, 2015. http://www.conveysupport.com/.

[94] Micron. Micron SB-852 Wolverine II, 2020. https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/hpc-single-board-accelerators/sb-852.

[95] Kosuke Mizuno, Yosuke Terachi, Kenta Takagi, Shintaro Izumi, Hiroshi Kawaguchi, and Masahiko Yoshimoto. An FPGA implementation of a HOG-based object detection processor. *IPSJ Transactions on System LSI Design Methodology*, 6:42–51, 2013.

[96] Shaul Mukamel. *Principles of nonlinear optical spectroscopy*, page 543. Oxford University Press, New York, U.S.A, 1995.

[97] Sergio D. Muñoz and Javier Hormigo. High-throughput FPGA implementation of QR decomposition. *IEEE Transactions on Circuits and Systems II*, 62(9):861–865, 2015.

[98] Kazuhiro Negi, Keisuke Dohi, Yuichiro Shibata, and Kiyoshi Oguri. Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm. In *International Conference on Field-Programmable Technology*, pages 1–8. IEEE, 2011.

[99] Christian F. A. Negre, Valeria C. Fuertes, M. Belén Oviedo, Fabiana Y. Oliva, and Cristián G. Sánchez. Quantum dynamics of light-induced charge injection in a model dye–nanoparticle complex. *The Journal of Physical Chemistry C*, 116(28):14748–14753, 2012.

[100] Christian F. A. Negre, Karin J. Young, M. Belén Oviedo, Laura J. Allen, Cristián G. Sánchez, Katarzyna N. Jarzembska, Jason B. Benedict, Robert H. Crabtree, Philip Coppens, Gary W. Brudvig, and Victor S. Batista. Photoelectrochemical hole injection revealed in polyoxotitanate nanocrystals functionalized with organic adsorbates. *Journal of the American Chemical Society*, 136(46):16420–16429, 2014.

[101] Vinh Ngo, Arnau Casadevall, Marc Codina, David Castells-Rufas, and Jordi Carrabina. A high-performance HOG extractor on FPGA. *arXiv preprint arXiv:1802.02187*, 2018.

[102] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[103] NVIDIA. GPU-Based deep learning inference: A performance and power analysis. White Paper, 2015. https://www.nvidia.com/.

[104] NVIDIA. CUBLAS NVIDIA's dense linear algebra on GPUs, 2018. https://developer.nvidia.com/cublas/.

[105] NVIDIA. CUDA toolkit documentation, 2018. https://docs.nvidia.com/cuda/.

[106] NVIDIA. NVML API reference, 2018. https://docs.nvidia.com/deploy/nvml-api/.

[107] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, and Markus Püschel. Applying the roofline model. In *2014 IEEE International Symposium*

on Performance Analysis of Systems and Software (ISPASS), pages 76–85, Monterey, CA, USA, 2014. IEEE.

[108] Safaa S. Omran and Ahmed K. Abdul-Abbas. Fast QR decomposition based on FPGA. In *IEEE International Conference on Advanced Science and Engineering (ICOASE)*, pages 189–193, Duhok, Iraq, 2018. IEEE.

[109] M. Belén Oviedo and Bryan M. Wong. Real-time quantum dynamics reveals complex, many-body interactions in solvated nanodroplets. *Journal of chemical theory and computation*, 12(4):1862–1871, 2016.

[110] M. Belén Oviedo, Ximena Zarate, Christian F.A. Negre, Eduardo Schott, Ramiro Arratia-Pérez, and Cristián G. Sánchez. Quantum dynamical simulations as a tool for predicting photoinjection mechanisms in dye-sensitized TiO2 solar cells. *The journal of physical chemistry letters*, 3(18):2548–2555, 2012.

[111] Michael Parker, Volker Mauer, and Dan Pritsker. QR decomposition using FPGAs. In *IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*, pages 416–421, Dayton, OH, USA, 2016. IEEE.

[112] Pavel Pokhilko, Evgeny Epifanovsky, and Anna I. Krylov. Double precision is not needed for many-body calculations: Emergent conventional wisdom. *Journal of chemical theory and computation*, 14(8):4088–4096, 2018.

[113] Victor Adrian Prisacariu and Ian Reid. fastHOG - A real-time GPU implementation of HOG. `http://www.robots.ox.ac.uk/~victor/pdfs/prisacariu_reid_tr2310_09.pdf`, 2009.

[114] Abid Rafique, Nachiket Kapre, and George A. Constantinides. Enhancing performance of tall-skinny QR factorization using FPGAs. In *22nd IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 443–450, Oslo, Norway, 2012. IEEE.

[115] Kishore K. Reddy and Mubarak Shah. Recognizing 50 human action categories of web videos. *Machine Vision and Applications, Springer*, 24(5):971–981, 2013.

[116] Frank Rhodes. On the metrics of Chaudhuri, Murthy and Chaudhuri. *Pattern Recognition*, 28(5):745–752, 1995.

[117] Alvise Rigo, Christian Pinto, Kevin Pouget, Daniel Raho, Denis Dutoit, Pierre-Yves Martinez, Chris Doran, Luca Benini, Iakovos Mavroidis, Manolis Marazakis, Valeria Bartsch, Guy Lonsdale, Antoniu Pop, John Goodacre, Annaik Colliot, Paul Carpenter, Petar Radojković, Dirk Pleiter, Dominique Drouin, and Benoît Dupont de Dinechin. Paving the way towards a highly energy-efficient and highly integrated compute node for the exascale revolution: The ExaNoDe approach. In *2017 IEEE Euromicro Conference on Digital System Design (DSD)*, pages 486–493. IEEE, 2017.

[118] Jose M. Rodriguez-Borbon, Xiaoyin Ma, Amit K. Roy-Chowdhury, and Walid Najjar. Heterogeneous acceleration of HAR applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(3):888–902, March 2020.

[119] Greg Ruetsch and Paulius Micikevicius. Optimizing matrix transpose in CUDA, 2009. https://www.cs.colostate.edu/cs675/MatrixTranspose.pdf.

[120] Romelia Salomon-Ferrer, Andreas W. Gotz, Duncan Poole, Scott Le Grand, and Ross C. Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit solvent particle mesh Ewald. *Journal of chemical theory and computation*, 9(9):3878–3888, 2013.

[121] Jason Sanders and Edward Kandrot. *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley Professional, 1st edition, 2010.

[122] Richard Sawyer. Calculating total power requirements for data centers. `http://accessdc.net/Download/`, 2004. White Paper, American Power Conversion.

[123] Christian Schuldt, Ivan Laptev, and Barbara Caputo. Recognizing human actions: A local SVM approach. In *Proceedings of the 17th IEEE International Conference on Pattern Recognition (ICPR)*, volume 3, pages 32–36. IEEE, 2004.

[124] Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Workshop on Edge Computing Using New Commodity Architectures (EDGE)*, pages 26–27, 2006.

[125] Anatoli Sergyienko and Oleg Maslennikov. Implementation of Givens QR-decomposition in FPGA. In *Parallel Processing and Applied Mathematics*, pages 458–465, Berlin, Germany, 2006. Springer.

[126] Robert Service. Computer science. What it'll take to go exascale. *Science, New York, NY*, 335(6067):394, 2012.

[127] Lesley Shannon, Veronica Cojocaru, Cong Nguyen Dao, and Philip H.W. Leong. Technology scaling in FPGAs: Trends in applications and architectures. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–8. IEEE, 2015.

[128] Lesley Shannon, Veronica Cojocaru, Cong Nguyen Dao, and Philip H.W. Leong. Technology scaling in FPGAs: Trends in applications and architectures. In *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–8, Vancouver, BC, Canada, 2015. IEEE.

[129] David E. Shaw, Martin M. Deneroff, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, Kevin J. Bowers, Jack C. Chao, Michael Eastwood, Joseph Gagliardo, J.P. Grossman, C. Richard Ho, Douglas Ierardi, István Kolossváry, John Klepeis, Timothy Layman, Christine McLeavey, Mark Moraes, Rolf Mueller, Edward Priest, Yibing Shan, Jochen Spengler, Michael

Theobald, Brian Towles, and Stanley Wang. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, 51(7):91–97, 2008.

[130] David E. Shaw, Ron O. Dror, John K. Salmon, J.P. Grossman, Kenneth M. Mackenzie, Joseph A. Bank, Cliff Young, Martin M. Deneroff, Brannon Batson, Kevin J. Bowers, Edmond Chow, Michael Eastwood, Douglas Ierardi, John L. Klepeis, Jeffrey Kuskin, Richard H. Larson, Kresten Lindorff-Larsen, Paul Maragakis, Mark A. Moraes, Stefano Piana, Yibing Shan, and Brian Towles. Millisecond-scale molecular dynamics simulations on Anton. In *ACM Proceedings of the conference on high performance computing networking, storage and analysis*, page 39. ACM, 2009.

[131] David E. Shaw, Paul Maragakis, Kresten Lindorff-Larsen, Stefano Piana, Ron O. Dror, Michael P. Eastwood, Joseph A. Bank, John M. Jumper, John K. Salmon, Yibing Shan, and Willy Wriggers. Atomic-level characterization of the structural dynamics of proteins. *Science*, 330(6002):341–346, 2010.

[132] Karen Simonyan and Andrew Zisserman. Two-Stream convolutional networks for action recognition in videos. In *Advances in Neural Information Processing Systems*, pages 568–576, 2014.

[133] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[134] Scott Sirowy and Alessandro Forin. Where's the beef? Why FPGAs are so fast. *Microsoft Research, Microsoft Corp., Redmond, WA*, 98052, 2008.

[135] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. UCF101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[136] G.W. Stewart. The decompositional approach to matrix computation. *IEEE Computing in Science & Engineering*, 2(1):50–59, 2000.

[137] Gilbert Strang. *Introduction to linear algebra*. Wellesley-Cambridge Press, Wellesley, MA, USA, 1993.

[138] Balaji Subramaniam, Winston Saunders, Tom Scogland, and Wu-chun Feng. Trends in energy-efficient computing: A perspective from the green500. In *2013 IEEE International Green Computing Conference Proceedings*, pages 1–8. IEEE, 2013.

[139] Amr Suleiman, Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Towards closing the energy gap between HOG and CNN features for embedded vision. *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[140] Lin Sun, Kui Jia, Dit-Yan Yeung, and Bertram E. Shi. Human action recognition using factorized spatio-temporal convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICPR)*, pages 4597–4605, 2015.

[141] Yi-Gang Tai, Kleanthis Psarris, and Chia-Tien Dan Lo. Synthesizing tiled matrix decomposition on FPGAs. In *21st IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 464–469, Chania, Greece, 2011. IEEE.

[142] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3D convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 4489–4497, 2015.

[143] Lloyd N. Trefethen and David Bau III. *Numerical linear algebra*. SIAM, 3600 Market Street, 6th Floor, Philadelphia, PA, 19104, USA, 1997.

[144] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *39th IEEE International Conference on Parallel Processing Workshops*, pages 207–216, San Diego, CA, USA, 2010. IEEE.

[145] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library, 2014. https://software.intel.com/en-us/mkl.

[146] Heng Wang, Muhammad Muneeb Ullah, Alexander Klaser, Ivan Laptev, and Cordelia Schmid. Evaluation of local spatio-temporal features for action recognition. In *British Machine Vision Conference*, pages 124.1–124.11, London, United Kingdom, 2009. BMVA Press.

[147] Limin Wang, Yu Qiao, and Xiaoou Tang. Action recognition with trajectory-pooled deep-convolutional descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4305–4314, 2015.

[148] Limin Wang, Yuanjun Xiong, Zhe Wang, and Yu Qiao. Towards good practices for very deep two-stream convnets. *arXiv preprint arXiv:1507.02159*, 2015.

[149] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In *IEEE Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Denver, CO, USA, 2013. IEEE.

[150] Xiaojun Wang and Miriam Leeser. A truly two-dimensional systolic array FPGA implementation of QR decomposition. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):3, 2009.

[151] David S. Watkins. *Fundamentals of matrix computations*. John Wiley & Sons, 222 Rosewood Drive, Danvers, MA, USA, 2004.

[152] Bryan M. Wong, Simon H. Ye, and Greg O'Bryan. Reversible, opto-mechanically induced spin-switching in a nanoribbon-spiropyran hybrid material. *Nanoscale*, 4:1321–1327, 2012.

[153] Carl Yang, Aydın Buluç, and John D. Owens. Design principles for sparse matrix multiplication on the GPU. In *European Conference on Parallel Processing*, pages 672–687. Springer, 2018.

[154] Chen Yang, Tong Geng, Tianqi Wang, Rushi Patel, Qingqing Xiong, Ahmed Sanaullah, Jiayi Sheng, Charles Lin, Vipin Sachdeva, Woody Sherman, and Martin Herbordt. Fully integrated On-FPGA molecular dynamics simulations. *arXiv preprint arXiv:1905.05359*, 2019.

[155] Ming Yang, Shuiwang Ji, Wei Xu, Jinjun Wang, Fengjun Lv, Kai Yu, Yihong Gong, Mert Dikmen, Dennis J. Lin, and Thomas S. Huang. Detecting human actions in surveillance videos. In *TREC Video Retrieval Evaluation Workshop*, 2009.

[156] Bowen Zhang, Limin Wang, Zhe Wang, Yu Qiao, and Hanli Wang. Real-time action recognition with enhanced motion vector CNNs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2718–2726, 2016.

[157] Yan Zhang, Yasser H. Shalabi, Rishabh Jain, Krishna K. Nagar, and Jason D. Bakos. FPGA vs. GPU for sparse matrix vector multiply. In *2009 IEEE International Conference on Field-Programmable Technology*, pages 255–262. IEEE, 2009.

[158] Ling Zhuo, Gerald R. Morris, and Viktor K. Prasanna. High-performance reduction circuits using deeply pipelined operators on FPGAs. *IEEE Transactions on Parallel and Distributed Systems*, 18(10):1377–1392, 2007.

[159] Ling Zhuo and Viktor K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. In *18th IEEE International Parallel and Distributed Processing Symposium.*, page 92. IEEE, 2004.

[160] Ling Zhuo and Viktor K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):433–448, 2007.

[161] Will Y. Zou, Xiaoyu Wang, Miao Sun, and Yuanqing Lin. Generic object detection with dense neural patterns and regionlets. *arXiv preprint arXiv:1404.4316*, 2014.