

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica



Development of a multi-core and multi-accelerator platform for approximate computing

para optar por el título de
Ingenierio en Electrónica

con el grado académico de
Licenciatura

Pablo Felipe Osorio Marín

30 de noviembre de 2017

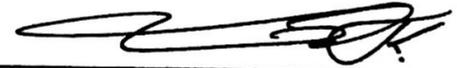
Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Proyecto de Graduación
Tribunal Evaluador

Proyecto de Graduación defendido ante el presente Tribunal Evaluador como requisito para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

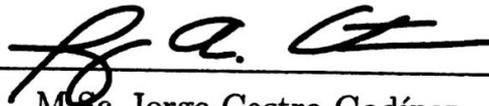
Miembros del Tribunal



M.Sc. Luis Paulino Méndez Badilla
Profesor Lector



M.Sc. Miguel Ángel Hernández Rivera
Profesor Lector



M.Sc. Jorge Castro-Godínez
Profesor Asesor

Los miembros de este Tribunal dan fe de que el presente trabajo de graduación ha sido aprobado y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica.

Cartago, 30 de noviembre de 2017.

Declaro que el presente documento de tesis ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos y resultados experimentales propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de tesis realizado y por el contenido del presente documento.



Pablo Felipe Osorio Marín

Cartago, 30 de noviembre de 2017

Céd.: 8 0111 0971

This work is licensed under a [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/) “Attribution-NonCommercial-ShareAlike 4.0 International” license.



to my dear family and friends

Acknowledgments

The result of this work and all the tough days of work, could not be possible without the constant support of many people. Many thanks to my adviser, Jorge Castro Godínez, through all the confusion and speculations every time I found guidance, advice and work to keep on going. Without him this opportunity could not have materialized for which am forever grateful. To my parents, Luis and Gloria, my sister and friends to whom I find myself deeply grateful for everything they have done to help me achieve this goal. Who have endured all this process with me and even when it seems impossible help me be above it. I will always be growing thanks to your support.

Imagination is more important than knowledge -Albert Einstein

Pablo Felipe Osorio Marín

Cartago. November 30th, 2017.

Abstract

Changing environment in the current technologies have introduced a gap between the ever growing needs of users and the state of present designs. As high data and hard computation applications moved forward in the near future, the current trend reaches for a greater performance. Approximate computing enters this scheme to boost a system overall attributes, while working with intrinsic and error tolerable characteristics both in software and hardware. This work proposes a multicore and multi-accelerator platform design that uses both exact and approximate versions, also providing interaction with a software counterpart to ensure usage of both layouts. A set of five different approximate accelerator versions and one exact, are present for three different image processing filters, Laplace, Sobel and Gauss, along with their respective characterization in terms of Power, Area and Delay time. This will show better results for design versions 2 and 3. Later it will be seen three different interfaces designs for accelerators along with a softcore processor, Altera's NIOS II. Results gathered demonstrate a definitive improvement while using approximate accelerators in comparison with software and exact accelerator implementations. Memory accessing and filter operations times, for two different matrices sizes, present a gain of 500, 2000 and 1500 cycles measure for Laplace, Gauss and Sobel filters respectively, while contrasting software times, and a range of 28-84, 20-40 and 68-100 ticks decrease against the use of an exact accelerator.

Keywords: Multicore, Accelerator, Memory mapped, Interfaces, Approximate, platform, Sobel, Laplace, Gauss, Parallelism, DMA, Processor, Power, Area, Delay, Time, Performance.

Resumen

El constante cambio en el ámbito de las nuevas tecnologías ha traído consigo una brecha entre las necesidades de los usuarios y el estado actual de los diseños. Computación aproximada entra en este esquema para mejorar las características de los diseños tanto en software como en hardware, al utilizar las propiedades intrínsecas y tolerables a errores de cada componente. Este trabajo propone el diseño de una plataforma multi-núcleo y multi-acelerador que usa versiones tanto exactas como aproximadas, además de recursos por software para asegurar ambas estructuras. Se muestra un grupo de cinco versiones diferentes de aceleradores aproximados, así como una versión exacta para los filtros de procesamiento de imágenes, Laplace, Sobel y Gauss, cada uno junto con su respectiva caracterización en términos de Potencia, Área y Tiempo de retraso. En estas se observarán mejores resultados para las versiones 2 y 3. Continuamente se muestran tres diseños diferentes de interfaces para la incorporación de aceleradores en conjunto con un procesador softcore, Altera NIOS II. Los resultados recopilados demuestran una mejora considerable al utilizar las versiones aproximadas en comparación con las versiones de software y acelerador exacto. Los tiempos de operación de filtrado y accesos a memoria, para dos tamaños de matrices diferentes, presentan una ganancia de 500, 2000 y 1500 ciclos en mediciones para los filtros Laplace, Gauss y Sobel respectivamente contrastando contra los tiempos por software, y un decremento en el rango de 28-84, 20-40 y 68-100 ciclos contra el uso de acelerador exacto.

Palabras clave: Multi-núcleo, Acelerador, Interfaces, Aproximado, Plataforma, Sobel, Laplace, Gauss, Paralelismo, DMA, Procesador, Potencia, Área, Retraso, Tiempo, Rendimiento.

Contents

List of Figures	iii
List of Tables	v
Abbreviations	vii
1 Introduction	1
1.1 Approximate Computing	4
1.2 Contribution	6
2 Background and Related Work	9
2.1 Background	9
2.2 Related Work	12
3 Characterization of approximate accelerators	17
3.1 Image processing filters	18
3.1.1 Laplace	18
3.1.2 Gauss	19
3.1.3 Sobel	20
3.2 Experimental Setup	22
3.3 Characterization Results	24
4 Interfacing accelerators and a softcore processors	29
4.1 Proposed Designs	31
4.1.1 Avalon Slave Interface	32
4.1.2 Custom Avalon Master-Slave interface	35
4.1.3 DMA Interface	38
5 A Multicore and Multi-Accelerator Platform	45
5.1 Multi-Core Design	45
5.2 Proposed Design	47
6 Conclusions	53
6.1 Future work	54
Bibliography	55

List of Figures

1.1	Power Consumption vs CPU Utilization (taken from [17]).	2
1.2	Power Dissipation for several microprocessors (taken from [7]).	3
1.3	Moore's Law (taken from [35]).	3
1.4	Utilization wall (taken from [27]).	4
1.5	Memory Wall (taken from [24]).	5
1.6	Approximate Architecture with Quality Control (taken from [8]).	6
2.1	Task parallelism (taken from [26]).	10
2.2	Application Mapping (taken from [4]).	12
2.3	Multicore design (taken from [21]).	13
2.4	SNNAP System Diagram (taken from [33]).	15
2.5	Methodology Flow (taken from [22]).	16
3.1	Camerman test image	17
3.2	Laplace filter data flow graph.	18
3.3	Gauss 3x3 filter data flow graph.	20
3.4	Sobel filter data flow graph.	21
3.5	Methodology diagram for accelerators characterization.	22
3.6	Laplace 8 Filter Versions Outputs	25
3.7	Gauss Filter Versions Outputs	26
3.8	Sobel Filter Versions Outputs	28
4.1	Avalon Memory Mapped Interface Transfers (taken from [1]).	30
4.2	NIOS Program Code Memory Map. Retrieved from [2]	32
4.3	Avalon Slave Interface for Accelerators.	33
4.4	Flow Diagram Avalon Slave Interface.	34
4.5	Read Timing Diagram for the Slave Interface	35
4.6	Write Timing Diagram for the Slave Interface	35
4.7	Avalon Master-Slave Interface for Accelerators	36
4.8	Finite State Diagrams	37
4.9	Write Timing Diagram for the Master Interface	38
4.10	Read Timing Diagram for the Master Interface	38
4.11	DMA Interface for Accelerators	39
4.12	Flow Diagram DMA Interface	40

5.1	Multicore Memory Partition. (taken from [2])	46
5.2	Proposed Multicore Design	48
5.3	Flow Diagrama Master Application	49
5.4	Flow Diagrama Slave Application	51

List of Tables

3.1	Quality characteristics of Laplace approximate accelerators	24
3.2	Laplace Accelerators Characterization	25
3.3	Quality characteristics of Gaussian approximate accelerators	26
3.4	Gaussian Accelerators Characterization	27
3.5	Quality characteristics of Sobel approximate accelerators	27
3.6	Sobel Accelerators Characterization	28
4.1	Performance Time Filters Software Versions	42
4.2	Performance Time Laplace 8 Accelerators	42
4.3	Performance Time Gauss 3x3 Accelerators	43
4.4	Performance Time Sobel Accelerators	43

Abbreviations

BSP	Binary Space Partitioning
CPU	Central Processing Unit
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
EBL	Embedded Logic Blocks
ER	Error Rate
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPP	General Purpose Processor
GPU	Graphical Processing Unit
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
IDE	Mean Error Distance
IP	Intellectual Property
MED	Integrated Development Environment
PSNR	Peak Signal to Noise Ratio
SD	Secure Digital
SoC	System on Chip
SRAM	Static Random Access Memory
SSIM	Structural Similarity Method

Chapter 1

Introduction

With the coming forth of newer applications, such as neural network, computer vision, machine learning, and data mining, it has become clear that characteristics of current technology no longer meet the performance requirements. There is an increasing demand of resources and efficiency for more complex applications and designs. It has been estimated that, on average, 4000 GB of data per day would be necessary for autonomous driving vehicles [23]. With the ongoing integration of Artificial Intelligence (AI) to end-user devices, and the continuous research on deep and machine learning, the need for better, faster, and efficient computing systems is on the rise.

Most of the problems with actual resources are not recent nor specific to a single cause. But even so, there are several factors that had cause this gap and will potentially generate a bigger treat. One of the problems resides in how to manage such complicated computations without disregarding execution time, power, or energy. For instance, by 2008 a single server based on Intel x86 processor, and used for data center operation, was estimated to consume 250 W on average, which turned to 23 billion W in a year for a single server farm operation [17]. In terms of the power bill, costs to maintain these facilities, and the fact that for the years to come *data is the new oil* [9], the growth is expected to be exponentially bigger with each year. As depicted in Figure 1.1, the power consumption of any server depends on its maximum utilization. The question now resides in how can it be possible to reduce the power consumption without reducing the processed data or incrementing the number of computing nodes.

Even as the power consumption problem arises, it is not the only characteristic to take in account when looking into current computing systems. Just as this is part of one of many constrains, other factors can come in the way for a complete integration or a desired development. As central processing units (CPUs) are the center component of every computation, the depth now resides in its optimization and structure.

Moore's Law has rules the design of CPUs for more than 50 years. This empirical law states that every two years the numbers of transistors in an integrated circuit will double. Many complex algorithms, dense data applications, and graphics processing, begun to

rely deeply in the advancement of new and more powerful generation of processors. As presented in 1.3, the feature size has decreased over the years, reaching the sub-micron era (less than 100nm) around 2005. Recently, Intel has achieved 10nm technology [19], which has pointed out that Moore’s law might not continue to fulfill the expectations for further application. As Intel continues to be one of the biggest chip manufacturer around the world, it only creates a bigger gap between what it is expected from manufacturers and the actual needs of the present applications. The International Technology Roadmap for Semiconductors (ITRS) expectations report that it will only be viable to keep shrinking transistors until 2021-2025 [5], which also leads to the need of rethinking the present solutions to compensate future lacks and flaws that could be met. As the number of transistors per area for a specific processor grows, it allows to develop faster and more complex system, it represents a cost in terms of power and energy consumption, as it can be seen in Figure 1.2. So, newer design trends are needed to overcome these design challenges.

But the problem with Moore’s law doesn’t end there as the need for a continuous integration of transistors persists. According to Dennard classic scaling theory, each transistor count scales by a factor of S^2 , where S is the scaling factor between two technology processes, while the switching frequency is scaled by a factor of S . The capacitance and the threshold voltage are scaled by a factor of $1/S$ and $1/S^2$, respectively. As feature size has decreased, this conception does not hold accordingly anymore. Reaching the barrier of 90nm, the characteristics foreseen by the Post-Dennard regime, or leakage-limited, changed drastically, as depicted in Figure 1.4. Under this scheme, the threshold voltage can not be further reduced without expecting an increase in leakage, rising the power density per chip, and thus limiting the utilization of the available silicon. This problem is now referred as the *utilization wall*. Due to this, portions of chip’s silicon stay under-clocked at operation for full frequency or just are not used full time, making that sections of the chip remain *dark* (this concept is referred as *dark silicon* in the literature). This leads to an underutilization of resources, but it also opens the possibility to utilize portions of

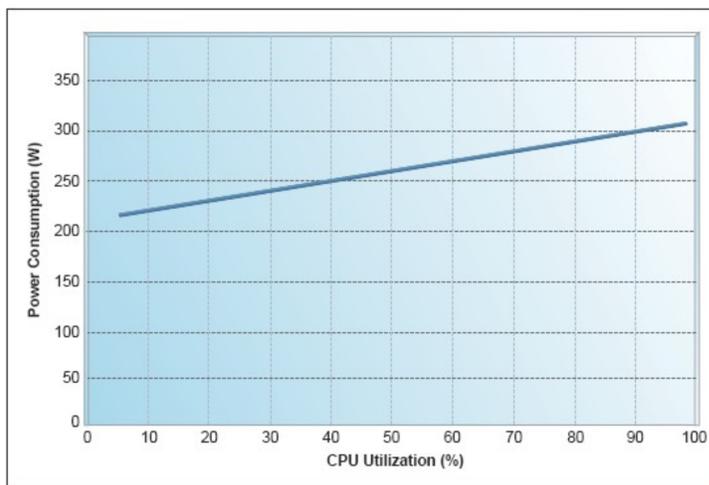


Figure 1.1: Power Consumption vs CPU Utilization (taken from [17]).

the chip to enhance energy efficiency that could then free up power budget, establishing a cycle that would allow more computations but with a small power cost.

There is still another more concerning challenge that single- and multi-core processors had not been able to counter, which is the difference of speed between microprocessors and memory. This problem is the so called *memory wall*. As the improvement rate in a single processor grow exponentially with each new design, the memory technology also experiences an exponential grow but at a much slower pace. As stated [37], on average, each 5th instruction on a program requires a memory access, which then leads to be almost 40% of the whole operation. This problem it is not an easy one since, on theory, DRAM (Dynamic Random Access Memory) speeds increases 7% per year, while processor trend is to augment by 80%. The growing gap between CPU and DRAM over the time is presented in Figure 1.5. Due to the cache characteristics, the associated cache misses cause that an overall continuous increase would end up degrading the general performance over time. Even for newer and more complex systems this challenge still remains as one

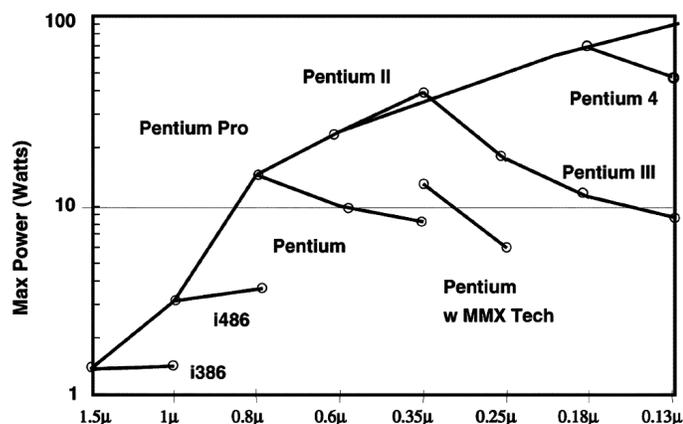


Figure 1.2: Power Dissipation for several microprocessors (taken from [7]).

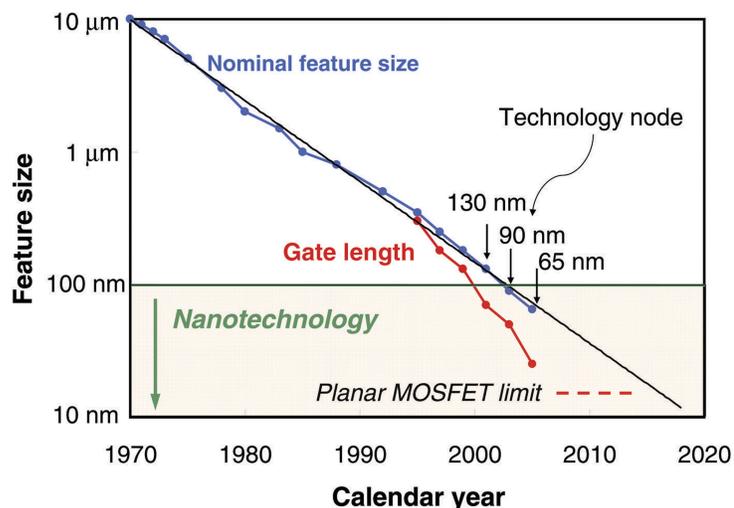


Figure 1.3: Moore's Law (taken from [35]).

of the most concerning factor when looking for different design opportunities to overcome this gap, and then reduce the averages cost of cycles in full functional operation [37].

As these problems seem to envelop the computational world, there have been several answers to each of them from different perspectives and studies. The firsts grasps at fighting these, came from the exploration of multi-core processors. This technology allows the users to still rely on the capabilities of a given chip or system by doing more task in parallel and thus enhancing the overall performance. Multicore has presented more energy efficiency in comparison to older single-core technologies allowing its integration in a diverse set of applications. As this technology is now in servers, laptops, mobile devices, and so on, it was thought to be the solution to the problems regarding Moore's law. But it also presents a worst problem for dark silicon, since the multi-core processing does not break the utilization wall, as the area is not scaled, the amount of chips that can be filled with more cores running at full frequency tends to decrease. The reality is that these cores still remain dark in most of their use [32], and in a multi-core architecture there's none real complete use of their whole resources. Even as these platforms are not the perfect solution, they present themselves as the most viable option for the near future [12] [11].

1.1 Approximate Computing

The diverse layout of problems surrounding the processors industry today has begun a quest for other design paradigms in order to tackled every part of this spectrum. Newer research trends help, but not overcome, every aspect of it. Current technologies are making efforts to instead of disregarding misses or bad logic as useless, to look them as a potential solution for generating a far better performance in total. In words of Tom

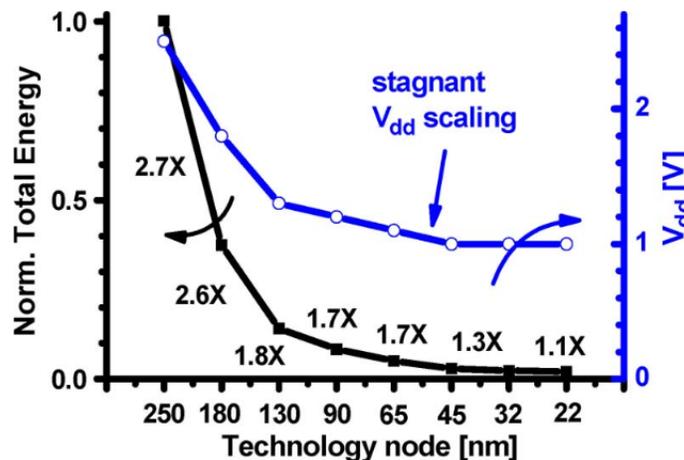


Figure 1.4: Utilization wall (taken from [27]).

Simonite, *a bad chip at math* can help develop the technology for the future to come [30]. All of these because of **Approximate Computing**.

Approximate Computing is a design paradigm that aims to exploit the error-tolerance in a wide set of applications by performing inexact computations that allow reductions in the required computing resources, and it can be noticeable in lower power, area, and execution time [36]. As several applications turns to the realm of estimation and probability, such as speech recognition, neural networks, mining, image processing, data analytics, and so on, approximate computing has a wide range of opportunities for its use and implementations taking a leverage in the intrinsic characteristics of these processes [20]. There had been several proposals to apply this concept to software, circuits and architectures.

The abilities in software to explore productive algorithms, while using the probabilistic opportunities has brought a clearer way to synthesize whole systems and elaborate new models on the run that facilitates their approximation. But this is not only one tool, a far better one resides in the coming of approximated compilers that enables to transform complete programs in order to enhance its performance and energy consumption, while allowing tolerable errors. The software techniques vary from a range of possibilities, beginning with error injection to specific parts in code moving forward to annotating approximable programs portions. Approximation has several usages and usable techniques in this area, such as loop perforation, where skipping several iterations can reduce overhead. This area has vast applications as signal and image processing, machine learning, data research, scientific computing and much more, all of this represent the future frontiers to be break by approximate computing, where not only the main possibilities reside but the applications characteristics tend to prefer an approximate usage [20].

The advancements made by approximate computing do not end with software. Considering hardware, this design paradigm has proven to be reliable for recent architectures

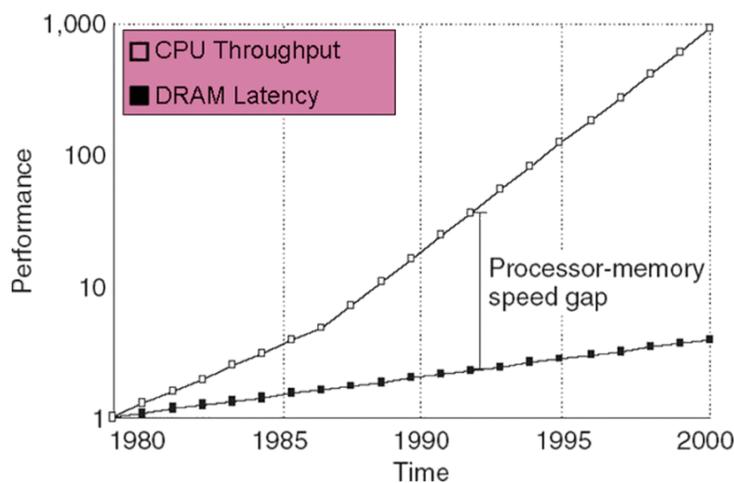


Figure 1.5: Memory Wall (taken from [24]).

covering two main areas regarding tolerable parts of software running on the processor and the translation from code to accelerators. This recent development also favors the incorporation of simpler units or processes, such as arithmetic circuits and synthesis techniques during hardware design, exploiting as well the intrinsic features of already codes that works with estimates and probabilities, as is the case of neural accelerators. These adders, multipliers, and automated approximate implementations, tend to present upgrades in power-efficiency characteristics by exploiting the control accuracy integrated in them [25]. Also in presence of intrinsic errors several hardware units can be useful for exploiting its approximate capabilities, such is the case with memory access skipping, voltage scaling, and refresh rate reduction, just to mention a few. In Figure 1.6 can be observed the conception of an approximate accelerator altogether with an error predictor unit and a host processor; this architecture presents the basis for an approximate approach capable of replacing regions of code for a hardware implementation, speeding up an application and predicting the errors that need to be corrected, when outside the tolerable range. This is just an example of the work done in this area and how can be integrated to real life end user process. By doing these the search for a far better and more effective technology, could have met its most palpable solution.

1.2 Contribution

This work aims to propose a multi-core and multi-accelerator platform for approximate computing. First, the design of approximate accelerators for 3 image processing kernels (Laplace, Gaussian, and Sobel), altogether with their characteristics in terms of error, delay, power, and area, are presented. For each kernel, five different designs are proposed using low-power and high-performance approximate adders, producing different accelerator versions for the same application. Centering around the characterization idea, on how this components will reflect on each accelerator, tests will focus on estimation tools from Altera to ease and enhance the results. Ranging from the synthesis tool, to PowerPlay analysis estimator and TimeQuest Analyzer. All of this combined give an accurate behavior and initial performance estimation.

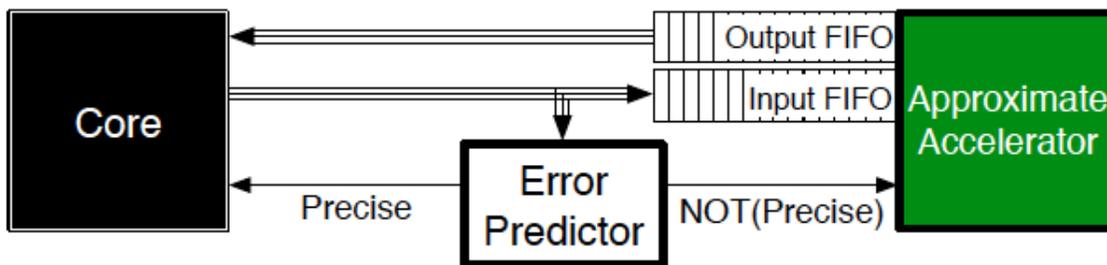


Figure 1.6: Approximate Architecture with Quality Control (taken from [8]).

Three different ways to integrate the accelerators with the softcore NIOS processor and different hardware components, from custom develop to Altera intellectual property, is presented. Each one is implemented following the bus and memory mapped protocols for slave/master interactions. The proposed designs differentiate in the protocols they need to sustain with the full system. The first shows the incorporation of a custom logic memory mapped slave who responds directly to the processor. The second has both memory mapped master and slave, to respond any read/write operations in memory sent from the processor and maintain the processing logic needed for validation. The last one has two direct memory access units (DMA) and a first in first out memory (FIFO), one for writing and the other reading, who also communicate with the processor to control the data flow.

Finally this work details a proposed multi-core platform that integrates approximate accelerators. Showing a structure capable of interact both with hardware and software, manipulating shared resources such as the accelerators, both exact and approximate, and perform adequately to the needs of master/slave interactions.

Chapter 2

Background and Related Work

2.1 Background

This section presents a compilation of basic notions related to image filtering, the uses of accelerators, and processor-accelerators relationship for building systems on Field Programmable Gate Array (FPGA). This reflects and establishes a practical basis in order to show a clearer picture on how these can be applied and used with the approximate computing design paradigm.

The hardware acceleration techniques are present in a diverse set of implementations, for instance, custom design circuits through hardware description language (HDL), and a variety of Graphical Processing Units (GPU) from vendors like Nvidia or ATI. The elaboration of such components have set a paradigm basis for parallelism exploitation, allowing to free resources from other parts of computing systems or relieving the amount of functions set for the main processor to do. In Figure 2.1 is possible to observe the task distribution for a single group of accelerators; this represents the basis for any design in the subject of parallelism. The integrations into commercial systems has shown to reach peak performance far better, allowing for faster development and managing a continuous research around this technology [26]. It is worth to mention that C code has maintain well establish patterns for progress around processors, but the integration of parallelism and the requirement of fine data in large amounts, tend to failed when setting an effective application. That is why, with the understanding of the desired architecture and with the flow chart around the C code software, is possible to develop Verilog- or VHDL-based accelerators capable of doing the same computations faster and with far more data.

Image processing has been a viable and reliable choice to develop hardware acceleration in recent years and it has been at the forefront of current studies in different areas in order to achieve real time performance. One example is presented in [38], where an image detection application to detect cells in a micro-well is moved to a FPGA-based accelerator implementation. By doing this, it is possible to implement faster growing systems in different researches. The current systems incorporating accelerators facilitate

a way for faster tracking, less resources and detailed output, especially when dealing with large amounts of input data.

As real time applications continue to develop faster than ever, the processing envelop within the diverse group of sensors and data acquiring systems grow even larger. Problems now reside in the level of autonomy that newer designs require, in order to sustain a confident task building and routines management. The mixture of general purpose processors and hardware accelerators integrated in a FPGA, become a resilient option for embedded systems focusing in the organizations of resources and functions. In order to achieve a greater speedup for the case of image processing, due to the larger amounts of data needed, a hardware implementation should be used [31]. Since software can not sustain the intensive parallelism and information required for fulfilling this larger modes of acquisition. By doing this kind of implementation more sensitive and high resolution data can be obtained, facilitating far better solutions and improve the end user necessities.

Works regarding parallelization and customization has been presented several times. Most recent ones face challenges in the area of optimization while incorporating techniques such as scheduling, synthesis and virtualization. The structures presenting multicore solutions by itself does not solve the area and power problems common to the technical world in the past years. Networks of systems utilizing direct memory access (DMA) and series of scratchpad memory (SPM) with engines of accelerator has come to show improvement in comparison to commercial available processors. One of these designs, with a group of 24 accelerators, shows seven times more speed up and 20 percent increase in energy savings

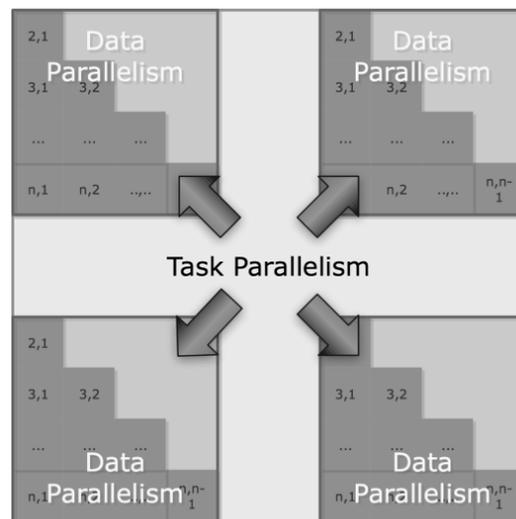


Figure 2.1: Task parallelism (taken from [26]).

in comparison to the 12 core 1.9GHz Intel Xeon processor [15]. These results present the effective performance of a group of monolithic accelerators, which tend to offer greater advantages in performance, programming and computational costs; improving lastly the different workloads whole systems can achieve in different platforms.

Not only recent systems should involve multi-accelerators techniques and scheduling, but also look at the need for incorporating these with multi-core and efficient mapping methodologies. Optimizing power consumption and resource usage and power has reached levels for which a single core at higher frequencies can not meet the specific requirements. The exploiting of parallelism and efficient low frequency multi-processor have shown a well distribution increasing performance and advancements. In order to achieve an adequate multi-core performance, first any given application needs to be separated into tasks to be executed in conjunction for each different core. This way it assures that the necessary memory hierarchy, synchronization and communication within the system are given to all assignments. Enhancement in mapping and management distribution, has presented the advantages of designing larger scale architectures with any number of cores needed. If the mapping requirements aren't met, the fully capabilities of any multi core design wouldn't be achieved [4]. In Figure 2.2 it is shown the mapping of a whole program using a graph to form tasks in different cores. By managing these processes one can incorporate any given set of instructions into an embedded platform with far greater overall characteristics, fulfilling this way the aim for user optimizations in the current system demands.

The specifics for an adaptive hardware acceleration algorithms resides in the characteristics of each algorithm in study. A propose methodology involve de PARO design flow [14], in which the compiler receives programs descriptions such as mapping, partitioning and localization. In this way it is possible to get the representation of the parallel architecture associated to it, which ensures the maximum data reuse for each combination of accelerator and processor array, while maintaining the resource constrains and optimizing the placement of logic components and the sequentially scheduling iterations. After this results are given, the PARO flow continues with the translation into hardware description language, enabling a road to the most efficient possible architecture and participating in hand tune if necessary. Such appliances are present and constantly selective in digital filtering, image analysis and neural network. For such designs it been proved, that incorporating this designs in a FPGA presents a more reliable way to use the available resources and present the optimal design.

As more architecture incorporate the potential of machines composing multi-core and multi-accelerator platforms, the risk reside in maintaining an appropriate communication system between the application running on the core and specific region of offload on accelerators [6]. In order to overcome this, its necessary to rely on the ability to dynamically schedule task over a full set of processing units. Several proposes overcome this barriers by introducing virtual shared memory, high level distribution or register mapping communication. StarPu, the first type of these, demonstrates a high-level programming that with help from automate prediction data transfers in the prefetching state, is able

to minimize the cost of each communication in the multi accelerator configuration and influence the decisions in the scheduler. By using this one is able to increase the speed and performance in multi-core and multi-accelerator machines.

2.2 Related Work

The increasing amount of computations needed for a full program running on a specific SoC, has brought different new types of available tools for increasing performance. There has been work done in this area, for instance, regarding an implementation of Wiener filter [10]. In this work the characteristic of a full integration with hardware acceleration or custom instructions is presented. For both cases the main focus is the capabilities of converting a C code from a image filter into hardware, following the corresponding data path. The proposed architecture considers that a receiver/sender structure for the image pixels to treat, a DMA controller and the image filter. The results gathered from it show that the design with the hardware accelerators is two orders of magnitude faster and required the least amount of logical gates. So, any design that requires a high level of processing data should consider as a first choice the development around accelerators. Taking in account that customs instructions gives more control around the circuit generated and variability, but at a much higher operational cost in terms of area and execution time.

With the large quantity of resources available, the use of softcores and FPGA presents great opportunity to develop and test different systems. By using processors such as the NIOS core, from Altera, it provides a base of studies to construct a multi-core platform and its viability for implementations it full embedded systems. A desired platform containing an adjustable size of cores is able to demonstrate that implementing image processing applications, the performance and speed tend to overcome the capabilities of a GPU. For the cases of FIR filter and matrix multiplication by 29.5% and 23.6%, respectively

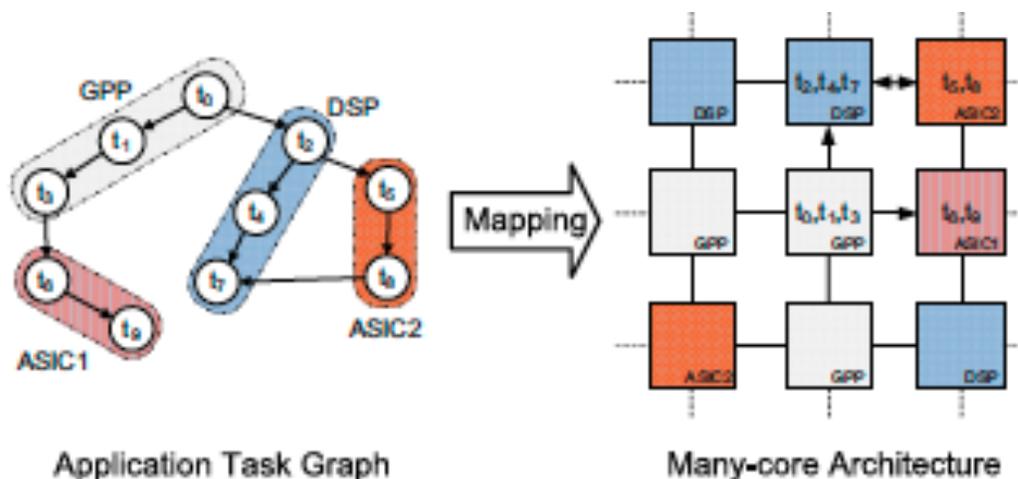


Figure 2.2: Application Mapping (taken from [4]).

[21]. In Figure 2.3 is presented the design of the multicore platform. As it can be seen, the reception from a software stage and the instruction router, first the desired application is receive and decomposed to a set of instructions to be executed. Then they will be distributed in each core so the output would then be saved in common memory. This interaction between software/hardware, as well with the incorporation of a NIOS processor, present a viable solution for programmable architectures that depend on big quantity of data. This work set a base for the optimizations needed in order for custom accelerators be implemented, and how by integrating these technologies such as approximate computing can continue increase the efficiency of such designs even more.

Research on approximate computing goes from simple blocks such as adders, multipliers and so on, in order to extend to accelerators and more complex systems, such as neural networks. These adders tend to integrate an Error Detection and Correction (EDC) unit, that enables flexibility due to requirement of the different applications. As this components becomes a decisive part in full systems for their adaptive capabilities, it is also clear the cost in area and energy. Considering this is also possible to incorporate a Consolidated Error Correction (CEC), which at a low cost enables to correct the accumulated error for any accelerator output by treating it with a specific model for approximation error, facilitating in this way a far better response in terms of area, speed and accuracy in results in comparison to modern accelerators [29]. To incorporate these type of error detection is necessary to take in account the unique requirements of each desired benchmark, so any architecture could apply it to improve its characteristics.

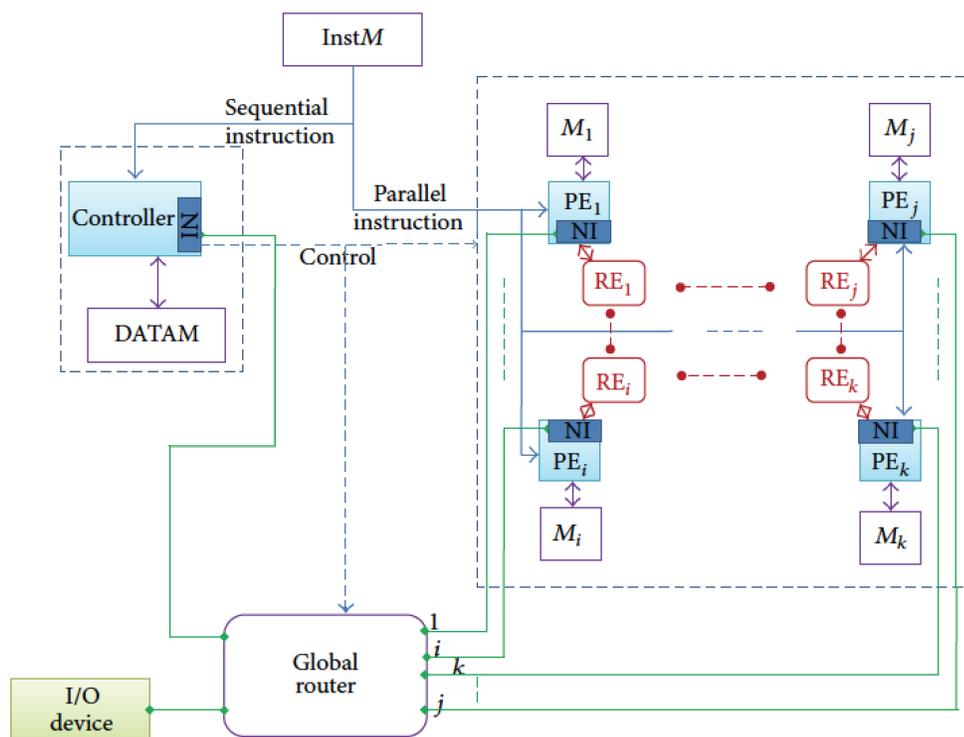


Figure 2.3: Multicore design (taken from [21]).

Also, exploring the possibility to declared programs, functions, and even hardware that can tolerate certain errors without compromising the overall results, while improving lastly the performance of each design, have been researched. More complex architectures are rising, involving dual voltage static random access memory (SRAM), interleave high and low voltage operations and approximate caches with registers. With the incorporation of approximate operations, the hardware achieves freedom to use less resources, sacrificing the accuracy but gaining the advantage of less energy used. In this aspect, the utilization of microarchitectures, with the capacity of managing its resources from a high voltage in exact mode to low voltage in approximate, become a big tool representing even a reduction ,for several benchmarks, of at least 43% in energy saves up [13]. The data in this structure would first undergo a mapping into approximate programming for then be applied into hardware, and the different possibilities it can sustain, from operation, to registers and loads, stores and caching in memory. This newer technology presents the usage of DMA to communicate with the SRAM, with reduced refresh rate and the implementations of registers that does not guaranteed and exact value, each of this features in approximate mode. Facilitating a whole assemble that would become reliable for data processing and acquisition in both modes.

The works in hardware acceleration approximation have come to two main threads, one where it is possible to exploit traditional architectures and another one with discrete accelerators [28]. In both it is possible to achieve more current research and future works. The traditional aspects limits the efficiency gains, such example its presented above. The newer approach seeks a co-design between accelerator, software interface and compiler in order to maintain quality and reach meet the performance necessities. If the old architectures are combined with new accelerators its possible to achieve better levels of behavior.

Not all the performance for a given application resides in its structure in hardware and traditional operation. There have been several solutions in high level synthesis where its possible to use a compiler framework in order to compose approximate programs from exact ones in order to use these in neural networks, such as systolic neural network accelerator in programmable logic (SNNAP). While offloading regions in the program its possible to identify the best parts and strategies in order to incorporate sections of code with approximation for a better performance. Using an specific set of accuracy requirements its possible to observe an increment in the operation characteristics of SNAAP, while running different set of programs [33]. With the increasing support of imprecision in modern systems, its possible to use such compilers for already set program domains and show approximation in both hardware and software. Including programs in embedded sensing and machine learning can drastically impact performance without impacting the final output, facilitating a whole new form of programming and integration in complete systems.

Due to its intrinsic characteristics for error tolerability, and probability background, neural networks are case of implementation subject for tests in this area. The most versatile and complete research up to now, resides in SNNAP. This type of neural acceleration was

designed to be implemented on SoC's and based on a FPGA for approximate programs. The mixture between its workflow and the neural network topology and weights maintains an effective senses that enables its use in commercial devices without the need of hard reconfigurations. The approach of understanding and passing C code to gate level enables a wide range of benchmarks. Utilizing this scheme in a Zynq board, it was possible to demonstrate up to 3.8 times the speed up and 2.8 times the energy savings [34]. In Figure 2.4 the configuration of this accelerator it is shown, the dual core enables a reception an storage of the program to be accelerated, the neural processing unit its the accelerator part from which sections of the approximated code are run and the result its then send to the ARM on chip memory via the accelerator coherency port. The design incorporating such technologies demonstrates how FPGA present an advantage tool for accelerating code regions on fixed hardware and it is able to exploit neural network for hardware approximation.

As more approximate accelerators are develop, the classical way of invocation is linked to specific code sections. But by working this way, there has been an associated error to any results given. The introduction of a predictor would help to prevent large quantity of data degradation by evaluating the different cases, in these manner is possible to run both modes as needed just by taking in account the signal from the predictor. Such systems are also presented in neural network to ensure realistic reference points. If the predictor is present, such systems could achieve 2.6 times more speedup and 2.8 times energy reduction by the specification of a 5% error [8]. In neural predictors would gain at least 17% in energy. For optimizing approximate systems the requirements of architectural mechanism that assist in output control are a necessity in order to achieve a viable design.

In more adaptive systems the incorporation of approximate blocks require to adaptively change in order to reduce the area and power overhead and maintain the error analysis of each design. The needs for logic that enables the change between modes of accurate

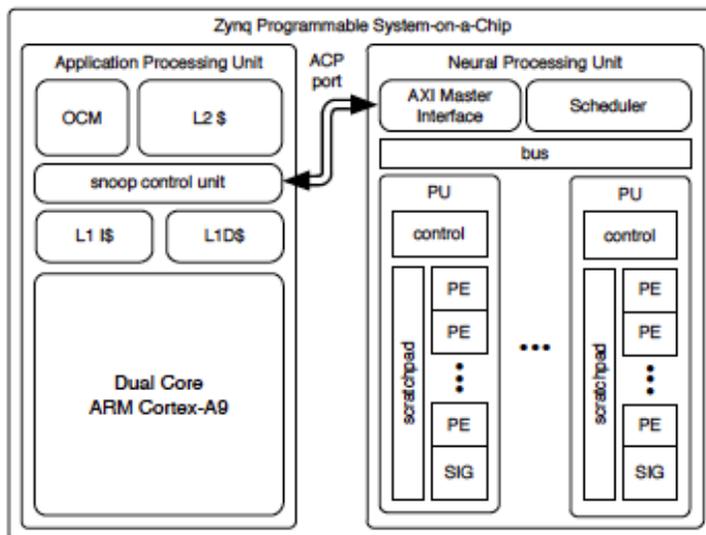


Figure 2.4: SNNAP System Diagram (taken from [33]).

and approximation, becomes clearer as the diverse use of multi accelerators in a single architecture begins to fail in flexibility and adaptivity [22]. In Figure 2.5, it is presented the custom logic for a full flow between high level synthesis generating accelerators from logic blocks, and its implementation in multi-accelerator architectures. For such, it is needed a way to analyze the continuing masking of error propagation and the change between modes, so that the result from the approximate computing are efficient and are able to propose viable solutions. The data flow that this methodology follows, is first the generation of the approximate accelerator, for this to be a viable option it is required a library of approximate logic blocks who by a given set of error analysis, in previous work, and a overall characterization, can construct the component that adapts to this criteria. After the generation, an analysis of the characterization and resilience is done, previous to the incorporation to the final computing architecture. This final architecture presents a n number of accelerators and its given core, both undergoing a serious of features that would finally enable the given application to run separately in each, by doing these a more efficient and efficient unit is present. The last block is the Approximation Management Unit, which its in charged of selecting the appropriate mode for each accelerator. By controlling this signals it is possible to keep the performance and quality necessary, but with the advantage of reducing the overall energy consumption. Such a tool for control needs to keep track for resilience in the data path and properties of different applications at run times in this type of architectures.

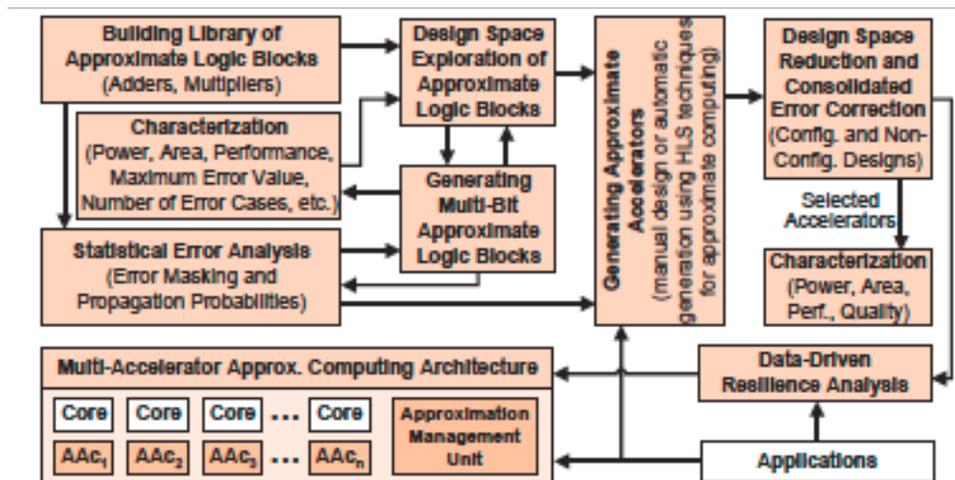


Figure 2.5: Methodology Flow (taken from [22]).

Chapter 3

Characterization of approximate accelerators

In this chapter, approximate accelerators for three image processing filters: Laplace, Sobel and Gaussian, are described. For each filter, five approximate versions are presented, varying their error metrics due to the approximate arithmetic components used. This chapter is divided in three main sections: the accelerators designs, experimental setup, and the results of the characterization in each case. In the first section a depiction of each filter is provided ranging from the operations needed, circuitry implementation and kernel specification. Later, the experimental setup is described, which contains all the information regarding the tools used for the characterization as well as the procedure followed. A flow diagram is presented, so the experimental structure is easily delineated and follow all along the text. Finally, the results from the experimentation and implementation of each filter, using test image Figure 3.1, are shown in the last section.



Figure 3.1: Cameraman test image

3.1 Image processing filters

This section presents three different filters used along this work. A full depiction of the operations can be seen in the following diagrams and explanations. For each filter, there is a brief kernel description, in order to understand the mathematical process behind it, and also a group of pictures are placed to demonstrate the different effects of filtering on a specific image.

3.1.1 Laplace

The Laplace filters are derivative filters used to find edges in images. They are represented by equation (3.1).

$$L(x, y) = \frac{\partial^2 f(x, y)}{\partial x^2} + \frac{\partial^2 f(x, y)}{\partial y^2} \quad (3.1)$$

There are several ways to get the discrete representation on the effect of a Laplacian model, the most common being the convolution around a central negative peak. For the scenarios studied in this work, the kernel of the Laplace filter is described as:

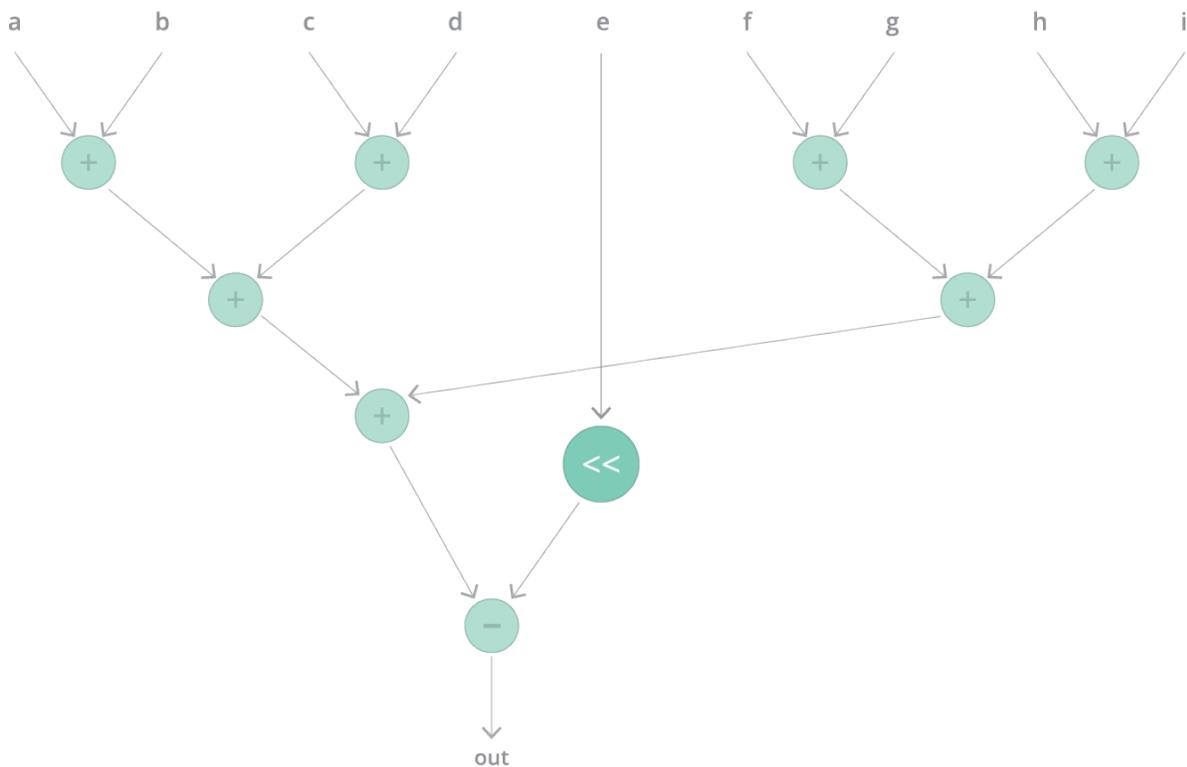


Figure 3.2: Laplace filter data flow graph.

$$\Delta_{xy}^2 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

As can be seen in Figure 3.2, a data flow graph representing this filter is composed of combinational logic, taking in account 7 adding operations, 1 subtraction and 1 shift left operation. In the first part of the operation, nine inputs of eight bits are taken, the e entry is associated specifically with the pixel in study and the other inputs are the surrounding matrix in order to execute the convolution. The diagram proposed is the outcome after the convolution is executed between the kernel, presented above, and the matrix associated with the desired pixel. Just as the e input undergoes a left shift of two, in order to get a multiplication by four, the other eight inputs (a , b , c , d , f , g , h , and i), are added between each other. Then, the result of the multiplication is subtracted from the result of all additions, to get the final output of the laplacian filter. Finally the output is constrained, so that the obtained value stays positive and does not surpass an integer value of 255. The operations described are the basics, both exact and approximate accelerators would have to complete. For each of the approximate cases, the proposed variations would concern the adders in every component, varying these with its approximate counterparts in order to understand how they would reflect in the behavior and results provided by each accelerator.

3.1.2 Gauss

For the Gauss smoothing filter, or Gaussian filter, the two dimension convolution are used to blur images and remove details and noise. This filter provides a method for reducing intensity, but it diverges from other types since the kernel used for it is bell shaped. The equation (3.2) presents the form a Gaussian distribution has.

$$G(x, y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (3.2)$$

It is worth mentioning that the implantation of this operator type is necessary to obtain a kernel that allows to perform convolution operations, just as for Laplace filter. The kernel in study is for Gauss filtering, using a 3x3 dimension filter, is giving by:

$$G = \frac{1}{16} \cdot \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

In Figure 3.3, the data flow graph for the 3x3 Gaussian filter is depicted. In this design, the component is constructed with combinational logic, receiving at the start nine inputs, representing the matrix in study. The logic describes the result after performing the convolution, beginning by a one bit left shift, in order to apply a multiplication by 2, on

the inputs b , d , f , and h ; for the e input is performed a two bit left shift, so its performed a multiplication by four. After this is complete, the logic proceeds to add the entire a with the result of the shift in input b , the same with c and the shift in d , g and the shift in h , and finally the adding of both shifts in e and f . The results are then added between each other, and with the input i . This final operation undergoes a right shift of four, so that the division by 16 is completed, obtaining the output of the accelerator. This output will then be constrained in order to maintain the values in a range between 0 and 255, by doing this is possible to limit the result and check its reliability. The same logic presented in 3.3 would be used for the different Gauss filter approximate models done, since the main change is the replacement of its adders by an inexact counterparts.

3.1.3 Sobel

The Sobel filter of two dimensions performs a spatial gradient measurement on images, and is able to remark areas of high spatial repetition. The kernel is able to respond directly to vertical and horizontal operation respectively to the pixel under verification. The way it works is by taken two different kernels for each dimension, so it give different computations in every orientation. After the result is obtained, both magnitude and direction can be calculated. The procedure is basically a first order derivate and calculates the difference of pixel magnitude in a edge region. The kernels that describe the operations are as follows:

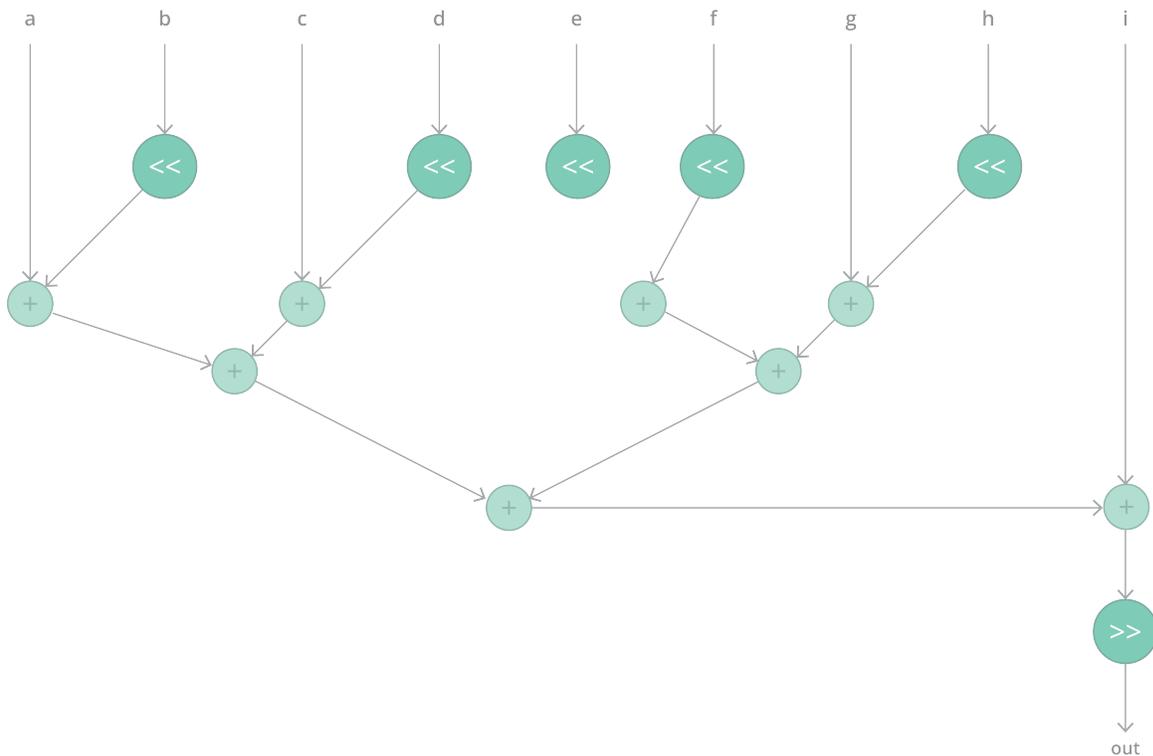


Figure 3.3: Gauss 3x3 filter data flow graph.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Taking in account the above 3x3 kernels is possible to realize the convolution between these and matrices in study, with this kernel the appropriate characterization is realized. Looking at Figure 3.4, it can be noticed the operations all data undergoes for both the vertical and horizontal position. This diagram describes as well the circuitry involving the Sobel accelerator, which is composed of eight inputs ($a, b, c, d, f, g, h,$ and i). In this case the center pixel is not taken in account, as one can extrapolate from the kernels described. The first step in the accelerator, as marked out by the road in green, is the process of the horizontal kernel. The inputs d and f are multiplied by two, while the $a-g$ and $c-i$ pixels are added respectively. The results from the $a-g$ operation is added with the d multiplication, and then it is subtracted with the result of the $g-i$ and f multiplication addition. The following step, marked by road blue, is the vertical kernel. The inputs b and h are multiplied by two, while $a-c$ and $g-i$ are added. The result of $a-c$ operation will be added with the output of the b multiplication, and then subtracted with the $g-i$ operation and h multiplication adding. The output of both subtractions will then be added giving the final output of the filter. This is denoted as *out* in Figure 3.4. Same as

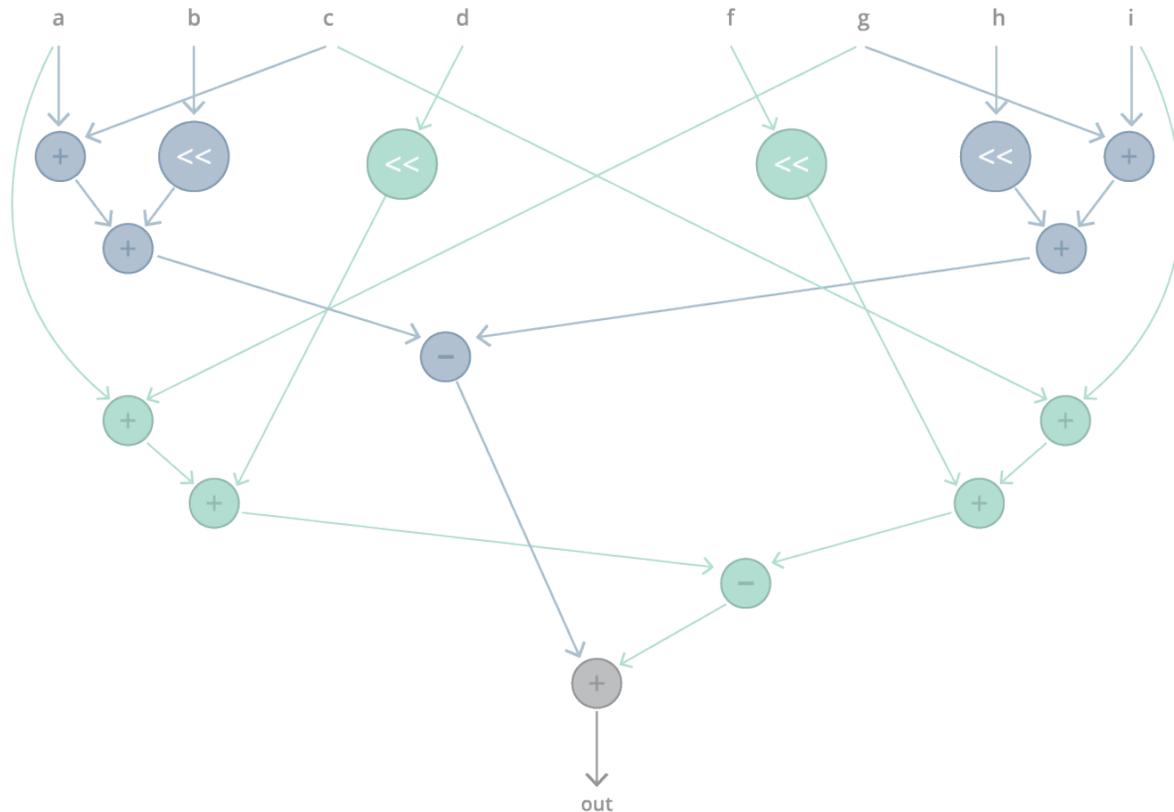


Figure 3.4: Sobel filter data flow graph.

the other cases, the output value will then be constrained between 0 and 255.

3.2 Experimental Setup

Once every filter is implemented using HDL, for both exact and approximate cases, the characterization is performed. Intel FPGA (Altera) provides a programmable logic device design software called Quartus II. This suite enables analysis and synthesis of HDL designs, perform timing analysis, and power consumption estimation. The tools necessary are Power Play, TimQuest and the synthesis report. For the results section, a DE2-115 Altera board is considered as target platform. Inside the board, a Cyclone IV EP4CE115 chip is available, containing 114480 logic elements (LEs), 3 888 Embedded memory (Kbits) and three 50 MHz oscillator clock inputs.

The characterization procedure utilizing these tools is depicted in Figure 3.5. The first instances needed are the design files of each accelerator. The HDL file, describing the flow and interactions between the inputs and outputs, is the main part of this procedure.

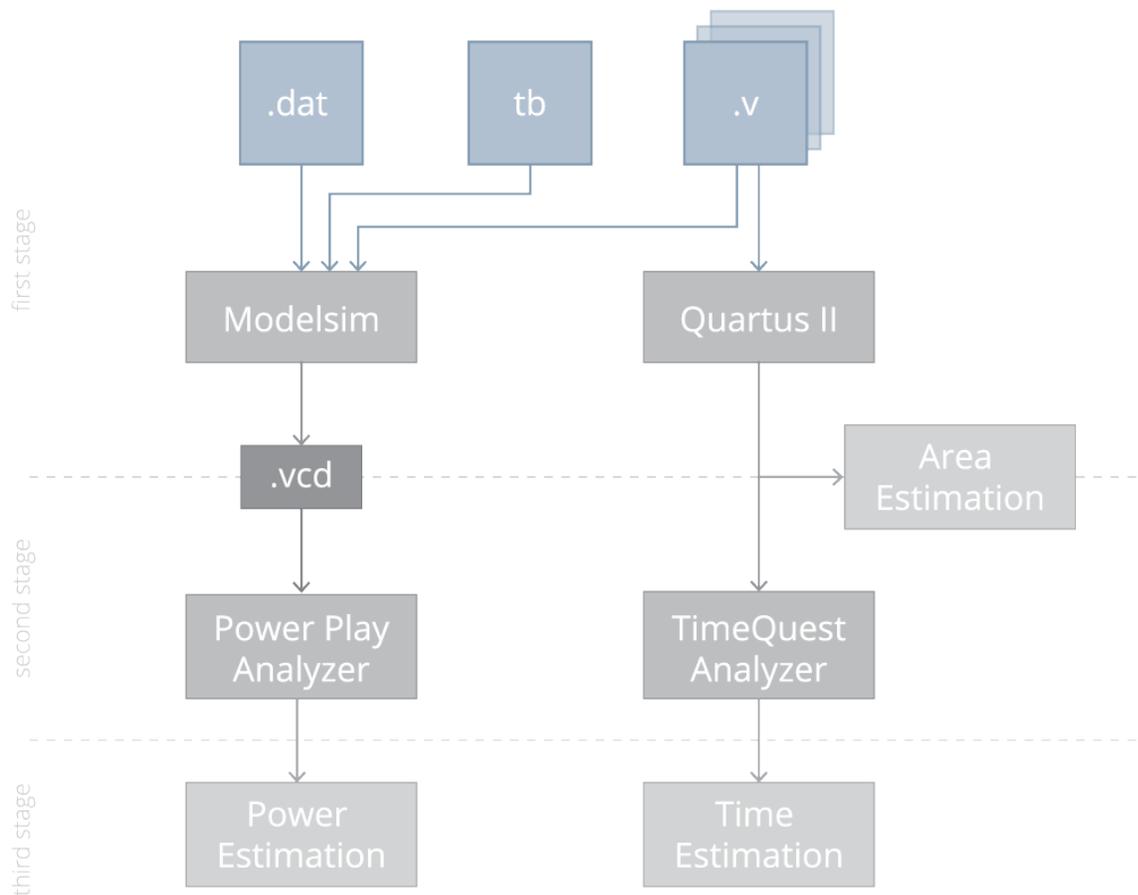


Figure 3.5: Methodology diagram for accelerators characterization.

The design files comprise a three set basis, between the VHDL (.v) file, the test bench (.tb) and the image data (.dat). Each one of these would compromise the dynamic and structural basis for the accelerators. The hardware description file describes the logic correspondent between its inputs and outputs, making it the main component of the study flow. Next is the test bench file, which it assures the behavioral model while performing a simulation study in Modelsim or any other verification tool. Finally the last component, of the design files, would be the image data file (.dat). A matrix that composes the entirety of the image in study can be seen in Figure 3.1. Since each value in the matrix accounts for a different input value for the accelerator, it would be read continuously in the test bench while performing the verification.

In order to rely on the design files, it is first needed to set the schematic file (.bdf). Fixing a main block of study for continuous test, also facilitating the exchange of top files for different study cases in the project. Since the work envelops close to 20 different versions of accelerators, such tool favors every repetition without increasing the amount of work. Likewise allows to maintain every design under study, in a single main file and save the reports without exception.

After the necessary aspects are met, the next step begins by compiling each top module. The main reasons to undergo this stage are verification, Fit and Place route for the DE2-115 board, and evaluation of Timing Analysis. So far in the second stage, as shown in Figure 3.5, the main use of the compiler is verification and fitting. Using the specified board, one can study the area report by analyzing the consumed percentage of logical elements and embedded memory. This record would help summarize the average size each accelerator could take and the required resources (associated with the area used), and enable comparison between approximate versions. Also sharing second stage, as seen in Figure 3.5, is the Modelsim simulation. Each accelerator has to undergo a stimulus study and functional verification, to guarantee correct behavior, taking the test bench and data files as inputs. The stimulus study output, value change dump file (.vcd), is required for Power Play analysis. In it is contained all the functional data from the accelerator simulation, inputs/outputs from the accelerator and clock cycles taken.

For stage three, there are two main operations. The first one is the Power Play analysis, which receive the .vcd file in order to accurately estimate the power consumption the design would have. By doing this, it is possible to have an evaluation of what is to be expected in full operation for a specific accelerator, understanding the optimal performance requirements for a complete system. The second analysis is the timing estimation, using the TimeQuest Analyzer. This tool is capable of perform timing analysis by validating the design logic, using industry standard constrains (sdc) and report methodology. Now, for a full characterization it presents a data-sheet report, where is possible to observe all the possible delay paths and determine the critical circuit path. Through this data is possible to set the implications and quality metrics needed to implement such circuitry in a complete system.

After the above analysis is done, the metrics for both exact and approximate design are

determined. For each version of accelerator it will be presented a two low, two highs and a mixture of both, as can be seen in Table 3.1. In every version the error estimate predictor is varied. Such information would be vital in order to grasp performance metrics needed to influence development on any design. Also in Tables 3.1, 3.3 and 3.5, are presented the characteristics of a quality study done by the execution of each accelerator

3.3 Characterization Results

Laplace

In the table 3.1, it can be seen the five different versions and there given quality output. This metrics can be observe by the parameters mean error distance (MED), error rate (ER), peak signal to noise ratio (PSNR) and the structural similarity index method (SSIM). The first two metrics are commonly used in the context of approximate computing [18] [16], while the remaining in image processing to compare quality among images. In Figure 3.6 can be seen the software outputs of each accelerator version for Laplace, listed from A to F one can appreciate the differences between each other. The effect of this filter can be seen, noting a darker image with edges accented, in both background and foreground. Image F is the exact version, while the other are approximate, comparing these with the data in Table 3.3 one can see the effect the quality characteristics has on an output image.

Table 3.1: Quality characteristics of Laplace approximate accelerators

	MED	ER	PSNR (dB)	SSIM
Laplace 1	32.47	0.237	21.41	0.79
Laplace 2	64.90	0.231	15.38	0.61
Laplace 3	6.84	0.472	32.89	0.84
Laplace 4	6.05	0.521	33.52	0.89
Laplace 5	39.18	0.584	14.18	0.48

The results obtained from the characterization of all Laplace accelerators is present in table 3.2. In this table, the three main characteristics are shown: area, delay (between inputs and outputs), and power consumption estimation. From this data one can see in terms of area measurement the exact version, noted as Laplace, have a similar area in comparison to version 1 and 5. The accelerator with less consumed blocks is version 4, while version 2 is the one taking more. In terms of time delay there are three aspects varying from minimum, average and maximum time. Each of these represent the full functioning of the accelerator. In the minimum case four of the approximate ones have higher times in comparison with the exact version. The highest and smallest versions been 5 and 4, with 10.47 and 7.88 nanoseconds (ns) respectively. The average presents more variability between each version, laying version 3 with 17.73 ns as the fastest. For

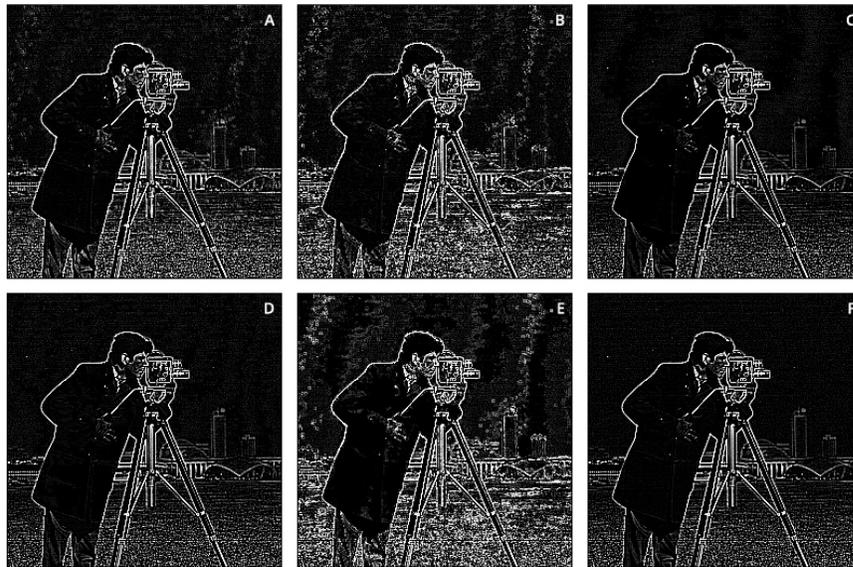


Figure 3.6: Laplace 8 Filter Versions Outputs

the maximum cases version 1, 2 and 5 are similar between one another, but faster than the exact. The least and highest times are 24.5385 and 31.331 ns, for versions 3 and 4 respectively. Finally for the power estimation only version 2 decrease, while version 1 sustains. For the other approximate versions the Power increase by a factor of ten in version 5, being the highest, and a factor of five in version 4. Both of these representing the cases with the most variability. This can lead to the expectation that version 2 can be better replacement both for power, but considering delay version 3 its an optimal option. Considering area is more difficult to choose a better option, since one must consider every characteristic and the final gain in a full system to opt for one choice.

Table 3.2: Laplace Accelerators Characterization

Accelerator	Area(ELB)	Delay (ns)			Power Estimation (mW)
		Minimum	Average	Maximum	
Laplace	136	7.99	21.94	29.36	156.74
Laplace 1	136	8.37	20.41	28.06	156.78
Laplace 2	151	9.49	21.51	28.07	154.37
Laplace 3	122	8.37	17.72	24.54	162.38
Laplace 4	121	7.88	19.75	31.33	161.02
Laplace 5	136	10.47	20.59	28.99	165.23

Gauss

In the Table 3.3, it is shown the quality metrics study for the Gauss filter, as is the case with the Laplace filter the MED, ER, PSNR and SSIM set the base for this study.

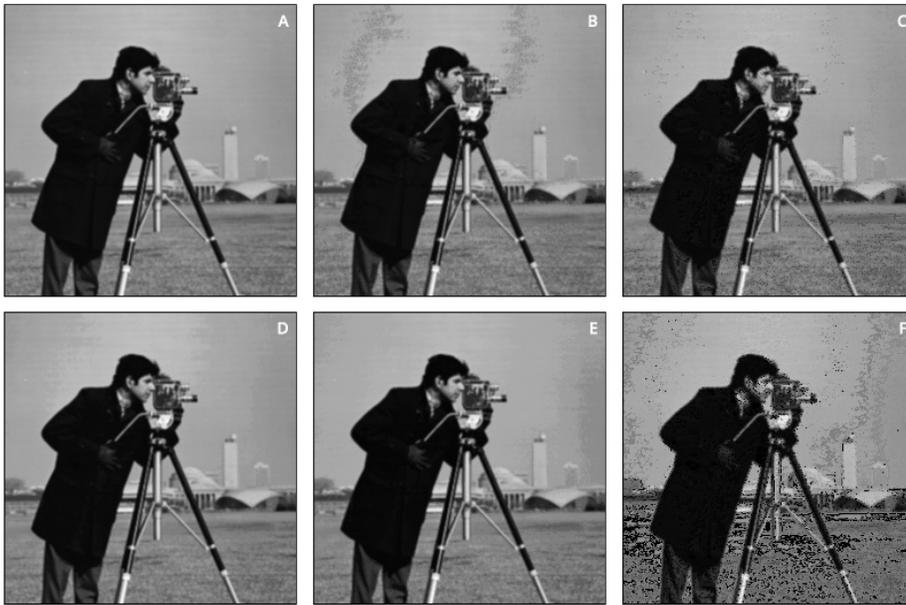


Figure 3.7: Gauss Filter Versions Outputs

In the table can be appreciated the five different version for the inexact accelerators. Along with this data, in Figure 3.7 can be seen the software outputs of each accelerator version for Gauss filtering, listed from A to F one can appreciate the differences between each other. The effect of this filter are seen, noting the removal noises and small quality stains around the main image parts. Image A is the exact version, while the other are approximate, comparing these with the data in Table 3.3 one can see the effect the quality characteristics has on an output image.

Table 3.3: Quality characteristics of Gaussian approximate accelerators

	MED	ER	PSNR (dB)	SSIM
Gaussian 1	8.53	0.210	33.92	0.93
Gaussian 2	7.82	0.272	32.15	0.89
Gaussian 3	2.31	0.799	40.49	0.97
Gaussian 4	3.26	0.934	37.18	0.96
Gaussian 5	5.79	0.725	28.54	0.83

The characterization for this accelerator can be seen in Table 3.4. From which Gauss represents the exact version, all the observations from this data would be done in comparison regarding it. In terms of area, only versions 1, 2, and 6 increase, while versions 3 and 4 decrease between 30-40 units. For the delay in the minimum case the most notable variant is given by version 3 with a change of almost 1 ns. In the average subject are present more changes between the obtained data, almost all tend to increase with the exception of version 4 which decrease. The most notable case been version 5 with an increment near to 5 ns. The last column present the data gathered from the Power Estimations Analysis,

in which can notably see increment in versions 1 and 2, however in comparison among one another the difference is almost ineffective. One can extract also a decrement for versions 3 and 5. Taking in account power results version 5 give the higher benefits. But in terms of delay, version 4 surpass all other characteristics, same as with the consumed logic blocks.

Table 3.4: Gaussian Accelerators Characterization

Accelerator	Area(ELB)	Delay (ns)			Power Estimation (mW)
		Minimum	Average	Maximum	
Gaussian	120	7.92	16.42	32.08	157.06
Gaussian 1	127	7.71	17.29	23.87	161.11
Gaussian 2	134	7.91	16.27	22.15	161.15
Gaussian 3	98	8.59	19.71	25.99	153.62
Gaussian 4	82	7.89	15.07	21.42	158.15
Gaussian 5	126	7.49	21.61	31.62	153.03

Sobel

Table 3.5, shows the quality metrics study for the Sobel filter, as is the case with the Gauss and Laplace filters the MED, ER, PSNR and SSIM are used. Along with this data, in Figure 3.8 can be seen the software outputs of each accelerator version, listed from A to F one can appreciate the differences between each other. Noticing the effect of Sobel filter where image boundaries are highlighted. Image A is the exact version, while the other are approximate, comparing these with the data in Table 3.3 one can see the effect the quality characteristics has on an output image.

Table 3.5: Quality characteristics of Sobel approximate accelerators

	MED	ER	PSNR (dB)	SSIM
Sobel 1	61.86	0.052	24.36	0.90
Sobel 2	11.77	0.737	21.59	0.77
Sobel 3	41.13	0.336	17.42	0.68
Sobel 4	6.00	0.718	31.80	0.91
Sobel 5	6.44	0.811	30.88	0.88

Further more during the study of Sobel filter accelerator and each of the different versions, is possible to procure several characteristics from them, each of these are show in Table 3.6. From this table, one can see that in terms of area versions 1 and 3 present an increment, while the others decrease, versions 5 been the lowest. For minimum delay about every versions maintains the same time, with the exception of version 1 which shows a small decrease. In average delay versions 1 and 4 stays near to the exact one, while version 5 increases and version 2 gives a slight trim. Maximum delay gives the more variability,

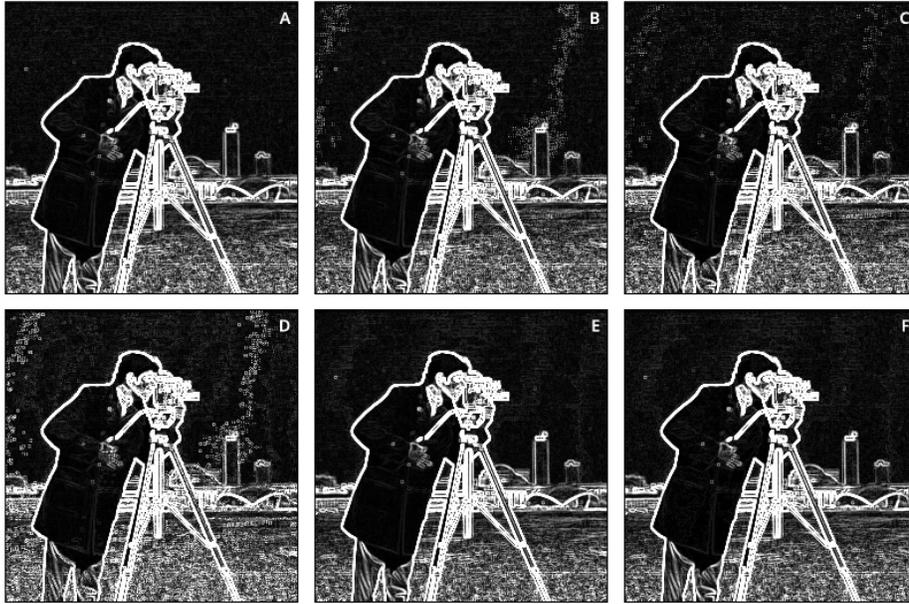


Figure 3.8: Sobel Filter Versions Outputs

showing higher and smaller times for version 1 and 5 respectively. Version 2 and 3 offer a narrow change between one another. Slight difference is demonstrated in versions 2, 3 and 4, in comparison with the exact one. From the power estimation, version 5 indicates the least consume. Versions 1 and 3 stays near the same value as the exact version. As for version 2 and 4 show a slim variant. Overall version 2 of the proposed accelerator return the highest benefits in terms of power and time, also is the case with version 5. The difference been in the average delay time, since in the last version a considerable increment can be seen and since it is a prevail occurrence affecting most of the process, the highest benefits comes from the second version.

Table 3.6: Sobel Accelerators Characterization

Accelerator	Area(ELB)	Delay (ns)			Power Estimation (mW)
		Minimum	Average	Maximum	
Sobel	197	10.30	22.54	26.12	162.62
Sobel 1	207	9.97	22.80	29.25	161.60
Sobel 2	192	10.78	20.41	25.90	158.18
Sobel 3	219	10.12	21.86	25.64	161.27
Sobel 4	191	10.52	22.59	26.04	159.72
Sobel 5	183	10.0295	24.19	20.49	154.74

All the data gathered from the above study would help understand the characteristics behind all the approximate accelerators, but the behavior in a full system would be study in the next chapter and how can it surpass its exact counterpart and software usage.

Chapter 4

Interfacing accelerators and a softcore processors

With the the design of approximate accelerators, the next step towards a complete multicore system leads to the construction of a model that implements a single core and an accelerator. By doing this, it becomes clearer the requirements and appropriate settings in order to proceed into a larger, multi-core design. The architectures presented are based on the NIOS II softcore processor, which consists of a high performance 32 bits design elaborated from a Harvard architecture, and it has been optimized for applications on FPGAs, enabling configuration of several capabilities, at real time, accordingly to performance needs.

The NIOS II softcore, as a standalone processor, has several advantages in its use in designing complex systems that incorporate a mixture of custom logic and Altera own IPs (Intellectual Property). It allows the use of two different versions of processors, NIOS II/f is adapted for faster performance and has the most configurable options, such as a unique memory management and protection units, removal of master ports, external interrupt controller, advanced arithmetic logic units and more. On the other hand, the NIOS II/e is an economy core, it is designed in order to have the smallest core possible size, resulting in a limited option and functions; but such is a reliable option in slave architectures when not many resources are needed for CPU operation.

The interface and implementation design was done in the Altera Quartus II software, which contains the Qsys Pro tool. This integration system allows to save time and effort by applying automatic interconnect logic to use along with IPs functions and subsystems. Qsys allows to include custom HDL designs, but in order to do so, it is necessary that these are enclosed by one or several interfaces depending on the architecture specific requirements. For any custom component to be used in Qsys, one has to declare the new component creation enabling the import of the design files. After these are imported, they need to be synthesized in order to Qsys check the logic, and map the inputs/outputs signals to a corresponding interface. In the case being it will be analyzed three different interfaces include methods for the involvement of the accelerators: the first using one slave,

the next includes two masters and one slave, and a last one utilizing DMA transactions and slave logic.

The library present at Qsys, shows a gallery of components already develop that sustains the interactions protocols between subsystems. Varying from streaming interfaces to memory operations, most of the components use the Avalon Memory Mapped maser/slave interface, in order to make specific transfers for both data and commands. In Figure 4.1, it can be seen the signals needed for such implementation, as well as the timing diagram revolving around the specifics of both read and write operations. In the read case, the address command is set so the interface can access an specific action or memory space, the read bit becomes high until waitrequest is deasserted and one cycle after the readdata value is transfer. The write action works very similar, the main difference relies on the fact that the writedata is kept for the duration of the write and waitrequest signal as the waitrequest is deasserted the writedata keeps its value until the write signal turns to low. By introducing this interfaces to the system, is possible to interact between the NIOS II processor, since this is configured as a master, and a custom slave that involves any of the previous accelerator designs.

Once the complete design is finished and generated in Qsys tool, the next step goes into importing the “.qip” file into the Quartus software. This is necessary in order to add all the design files into the project, both customs and from the Altera suite. By naming the Qsys file the same as the top module is possible to generate the component that encapsulates the whole system into the schematic file “.bdf”. Finally is possible to start compiling the project to get the “.sopcinfo” and “.sof” files. The first one relates to the hardware information extracted from the Qsys design, this file is loaded as an input in the beginning of the programming in the NIOS II Eclipse integrated development environment (IDE), this way the tool knows the specific component libraries it needs to add in the project, and also possible processor targets for the software. The second file correspond

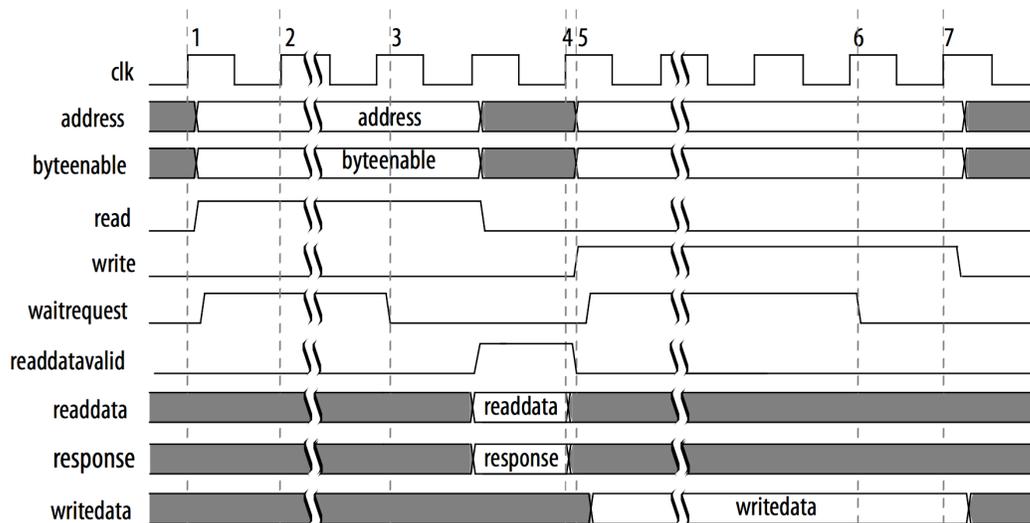


Figure 4.1: Avalon Memory Mapped Interface Transfers (taken from [1]).

to the downloadable direct volatile configuration, this would set the structure needed in the target FPGA as it is programmed directly into the SRAM cells.

For programming the NIOS II processor Eclipse IDE brings a specific C-code variant, that permits an easier construct and incorporates every element of the hardware design. Facilitating access to any given component by set of commands or Avalon specific address. The NIOS II processor in conjunction with the IDE, also has the advantage of a specific Hardware Abstraction Layer (HAL) which brings a series of specific functions and libraries. This grants the user a more versatile option for configuring specific targets in hardware and managing the desired flow easily. The Binary Space Partitioning (BSP) file, available with each application develop in the IDE, incorporates every library and header files for the execution of the program. The BSP “system.h” header file displays every component characteristics and its address, making possible to reference any part of the design in the application just by naming the specific component or port in it, doing a more precise map in complete designs. For its use in software design, each library is treated as any other in C programming, declaring it, and the needed headers, at top of the file before the coding starts.

The approach in this work is implemented by using the On Chip RAM Memory revolving around the 300 Kb of memory available, for both the program in the processor and the image matrix under test. The typical program partition, in NIOS II, is reflected in Figure 4.2. It contains five parts stack, heap, rwdata, rodata and text. The stack is used for temporary data storage and function call parameters, the heap is the dynamically allocate memory, rwdata are the read-write variables and pointers are store, rodata is read only data used for the execution of the code, and finally the text is the complete executable code the processor would perform at running time. Every processor starts its bootload operation after the software is download to the memory and finds it in the specified address vector in memory. For the download operation the software is send to the board via de USB-BLASTER cable, the code would then be stored beginning at the NIOS II reset and offset vector addresses set in Qsys, each of these separated between one another by 32 bytes space. After the code its allocated in memory the target processor leaves the IDLE-Pause state and enters the Running state, as long as the reset bit is not active. As point of reference and as stated by Altera’s User Manuals, the reset input on every component would be associated globally, the same with the debug reset in each NIOS target. This way gathers an optimal interaction between each component preserving its logic and behavior when the downloading stage is in course.

4.1 Proposed Designs

This work explores three different designs, in order to successfully achieve the goal of maintaining a direct use between both accelerator and the processor. Each design has its own advantages and disadvantages which will be explored within every case. By doing this, one is capable of discerning the most optimal interface, moving forward to the

incorporation in a full multicore platform. The different scenarios exposed show a slave interface, a design integrating a slave and two masters, and lastly the integration of two DMA's components.

4.1.1 Avalon Slave Interface

The first design explored is a complete slave interface. From Figure 4.3, it can see the main components of this. Taking the On Chip memory, JTAG-UART, NIOS II/f processor and the custom built Accelerator Interface. The NIOS II processor is set this way in order to obtain the fastest possible interaction between it and the custom interface, since the design is completely dependent and does not have any logic to interact directly with memory. On the memory side all 300 Kb were allocated, the application will take around 150 Kb since the matrices to be studied are completely set in the same software and required a lot of data space. The rest of the memory is leave free in order for the processor to interact directly with it, performing both reading and writing instructions at execution time. The JTAG-UART component is selected in order to favor communication between the host computer and the board where the software is running, by doing this is possible to not only download the software but also interact directly with the system console, reviewing steps and setting break points to verified the behavior of the software.

The flow diagram in Figure 4.4 shows the NIOS II application, the process it follow is:

1. **Packing:** At the beginning of the operation the system starts by making a packing process in the pixel values. The data packing consists of a four stage process, it starts by reading the values from the static matrix, defined in the software, each member has a specific weight of 8 bits, grouping at least four of them in order to comply with the 32-bit bus present with the Altera devices. After the values are

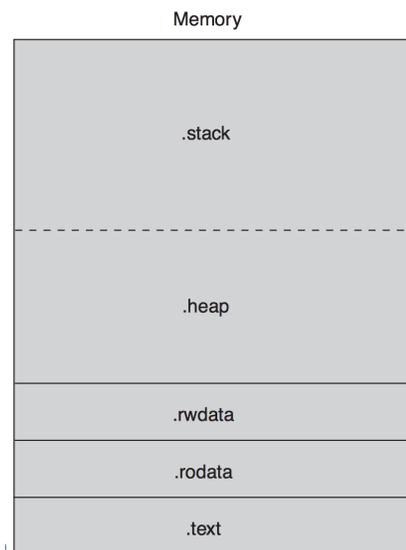


Figure 4.2: NIOS Program Code Memory Map. Retrieved from [2]

read, the packing starts by joining each new member in a 32-bit variable ,full_pack, applying in this way a bitwise operation moving to the left side by eight with the entering of every new value. This is done twice in order to incorporate the required eight pixels values. Once the full_pack1 and full_pack2 are finished, the pixel would then be stored in the same way to another 32-bit variable. In the scenario where Sobel filters were applied, the packing procedure changed in order to comply with the needed inputs, since the pixel value is not needed for the complete the operation.

2. **For Loop Matrix:** The next step in the software, is a cycle representing all the values that want to be study are pack or are not yet done. The cycle is composed of two nested for loops in order to read the matrix in terms of rows and columns. So it is possible to get every position that composes the total image.
3. **Slave interface operations:** The interactions with the slave interface commences, the first operation is a write utilizing the IOWR command from NIOS HAL¹, this instruction allows to perform a write action to the specific address that contains a component. This operation would send the full_pack variable and also and an offset. This would translate to the address input from the timing diagrams in Figures 4.5 and 4.6. The next step is performing another write with the second data, so the accelerator now contains all the information it needs and the start command is given by placing an interrupt into the processor. This signal read continuously by and IORD command, it is the counterpart to the last command allowing to read and store in an integer the result, until the process in the accelerator is finished. Once it stops the interruption is clear and the output value is read from the interface storing in both memory and in a 32 bit variable in software.

¹For more information consult the Nios II Gen2 Software Developer's Handbook [3]

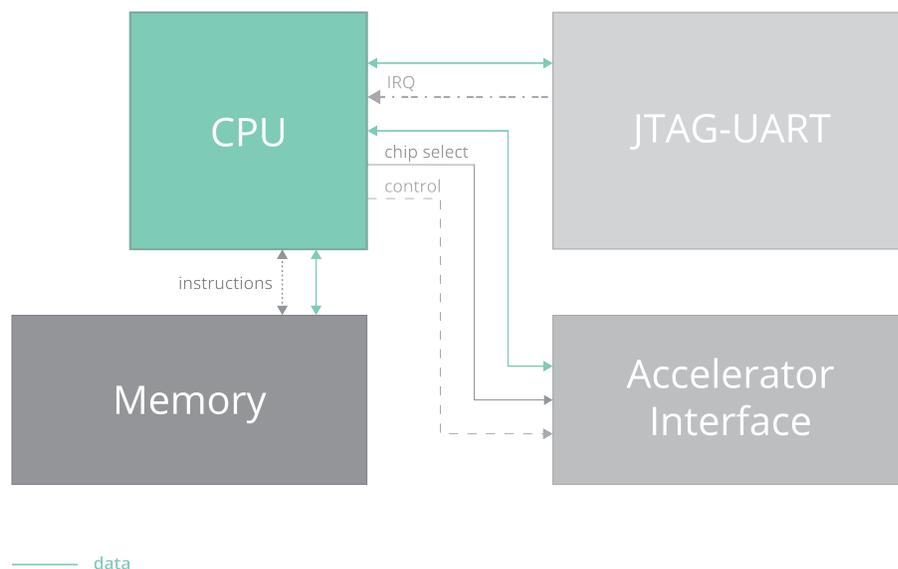


Figure 4.3: Avalon Slave Interface for Accelerators.

4. **Procedure cycle and application-end:** The NIOS continues the execution of this application until every pixel is read and applied to the accelerator, or until a specified amount of matrices is studied.

The interface design in Verilog HDL, its composed from a decoding logic set around the address signal. The IOWR and IORD command from NIOS, allows to send an offset to the base address, this translated to the input represented as “control” in Figure 4.3. The name control is given, because is representing the specific set of instructions to perform the decoding logic as needed. As the NIOS core select to perform a write actions, the chip_select and write inputs are set and the control is set to “0”. The first set of data is allowed to pass to the accelerator, corresponding to input a , b , c and d from Figure 3.2. The next set of values containing the f , g , h and i will be given when the control input is specified to do so, by a value of “1”, this occurs one more time in order to receive and save the main pixel. After gathering all necessary data the control would then receive the signal to start the accelerator and the interrupt output is set as long as this process continues. As stated above in operation “3” the NIOS will continue to read until the

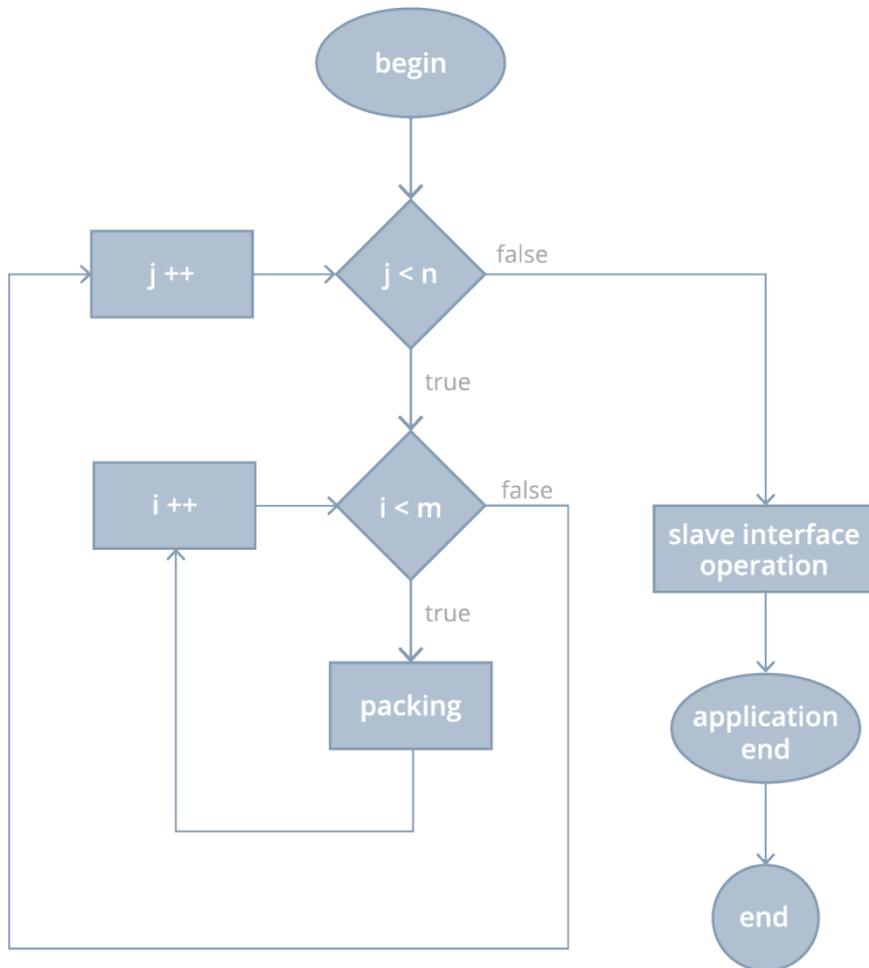


Figure 4.4: Flow Diagram Avalon Slave Interface.

operation is done, once completed the interface deasserts the interrupt after receiving the control signal. Finally the data present in the output of the interface can be read at any point by using the control value “4”, if the operation process its not finished the output will be set to low.

In Figures 4.5 and 4.6, timing diagrams are shown both from the execution process of a read and write operations. Similar to the interface diagram in 4.1, the inputs and outputs maintain its use with the exception of the wait_request signal, which for this design is not needed. From the timing diagrams also it can be extracted that for each read operation one clock cycle is required and for a write is at least three, in order to ensure the complete and valid data is transferred.

The main disadvantages with this interface are the amount of dependence from the NIOS processor, since it requires a continuous interactions of command signals from the core, leaving a great gap for multicore employment. But the main concern resides on the fact that is not capable of accessing memory by itself, it strictly relies upon the read and write operations perform in NIOS, leaving it to not be the most reliable of the proposed designs.

4.1.2 Custom Avalon Master-Slave interface

The second design involves the creation of an interface that involves both master and read procedures. By using custom master logic to read the different values from the On Chip Memory memory and writing the output of any accelerator back to it, creates an almost complete independent process, which only takes from the processor some control signals

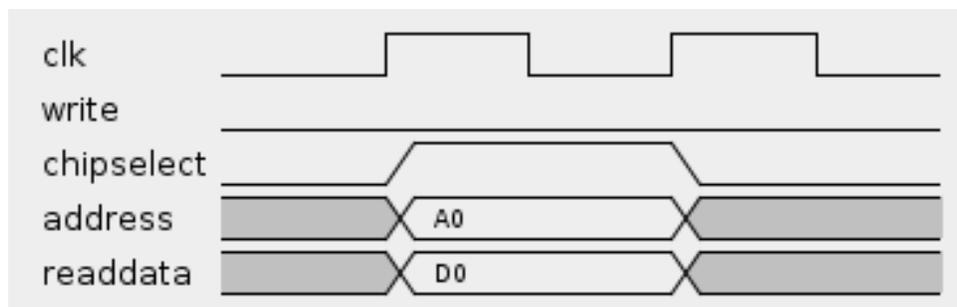


Figure 4.5: Read Timing Diagram for the Slave Interface

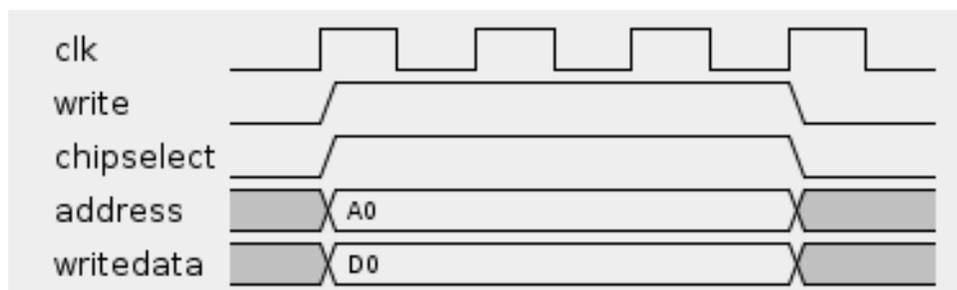


Figure 4.6: Write Timing Diagram for the Slave Interface

stating when to start, finish or interrupt the processor if needed. The proposed unit is composed from two finite state machines (FSM) both involving the appropriate master protocol needed. One in charged of the read process and the second one of the writing, both would be controlled from an slave interface that keeps communication from the NIOS and also contains the accelerator. The complete data flow can be seen in Figure 4.7, the Master Read takes the data from the On Chip RAM memory, passes it first to a 32 bit register, after the second read is perform the new data is stored again in a second register of 32 bits, repeating this one last time with an 8 bit register. After the three registers are set the accelerator interface then gathers this data and passes it to the accelerator, so it can then execute its operations and the output is saved in a 16 bit register. Finally this data is taken by the Master Write performing the write operation back to Memory.

For this interface, the flow diagram 4.4 show the application design to run in the processor. The procedures that undergo are:

1. **Packing:** At the beginning of the operation the system starts by making a packing process with the pixel values from the matrix, same as was the case in the slave interface.
2. **For Loop Matrix:** The next step in the software is a cycle representing if all the values that want to be study are pack or are not yet done. The cycle is composed of two nested for loops in order to read the matrix in terms of rows and columns. So it is possible to read every position that composes the total image, and send it to Memory already pack so it can be then easier to pass the data to the accelerators.
3. **Slave interface operations:** The interactions with the slave interface commences by a writing. Utilizing the IOWR command in which the control signal is given, and the specific address where data is store, this will be the *read address*. After performing this a waiting cycle begins until the read operation is finished. When the

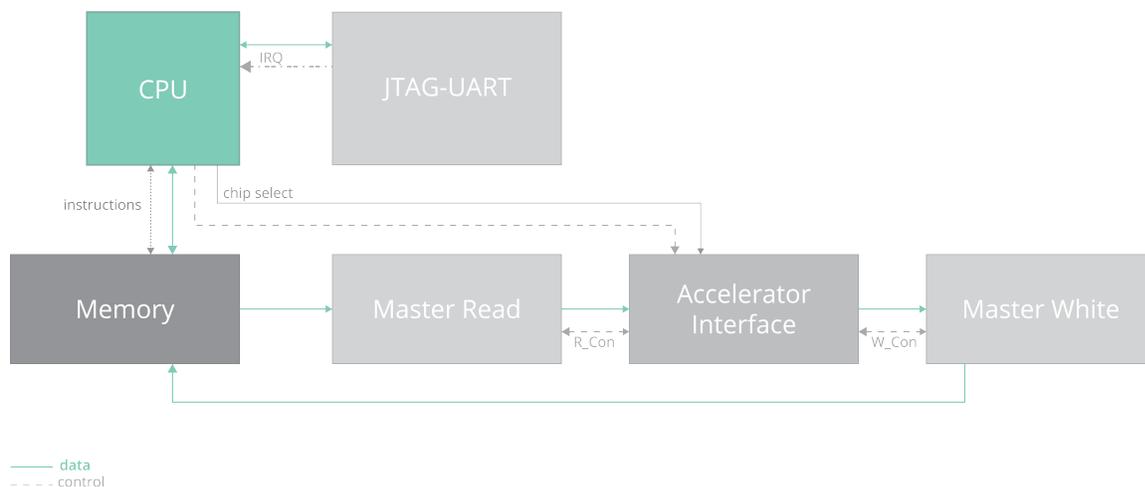


Figure 4.7: Avalon Master-Slave Interface for Accelerators

cycle is done, another write operation is given to the interface, in which the offset will set the writing operation to memory, and also receive the specific memory address where it can start storing the data, this will be the write address.

4. **Procedure cycle and application-end:** The NIOS continues the execution of this application until every pixel is read or until a specified amount of matrices is studied.

As for the master write and read, the state diagrams are presented in Figure 4.8. Graphic A shows the read state machine, while Graphic B the write counterpart. The logic for both masters is very similar. Beginning with the Idle state, both operations start and the address is set from which the data would be read or stored. For the read operation a counter is also set to “0”, due to the need for each accelerator to gathered at least two data blocks, performing this way a continuous state favoring a continuous reading. This counter also controls the enable logic for the registers before passing to the accelerator. During the running state both masters check a logic low value for the wait_request signal to perform any write or read procedure. By doing this is possible to validate the data and check if any operation is correct, for both cases this signal will be taker from the On Chip Memory. The difference between these two during the running state, is that the master read will check for the input read signal to be asserted and the wait_request is not, then the process can continues to retrieve the data and also increment the count.

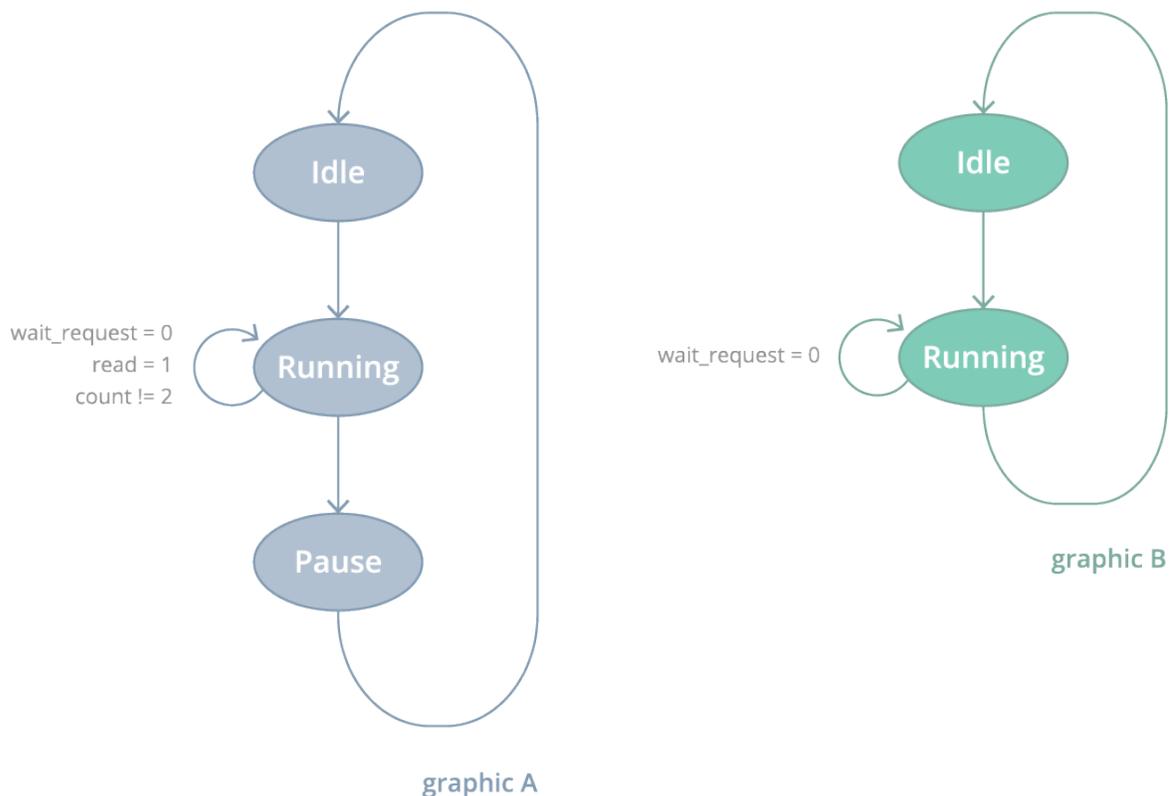


Figure 4.8: Finite State Diagrams

This state will continue in the read master until the count is equal to the number of data blocks needed by the accelerator, in each pass the read address will also increment by four to read the next 32 bit memory position on the On Chip RAM. For master write, the running state is in charge of setting the write signal for the memory unit, and assign the output of the accelerator, stored in the input register, to the writedata signal from the timing diagram 4.9, this will finally be the data stored in memory. By performing this last step the master write returns to Idle, differing from master read. For this last one a Pause state is given where the receive data is asserted and a wait cycle is perform, in order to comply with the Avalon Memory Mapped Interfaces logic and make sure all the data block are read accurately.

In Figures 4.9 and 4.10 the timing diagrams for both write and read master processes are shown. In both cases the main differences with the slave interface resides with utilization of the wait_request signal. The main information that extracted from both Figures is the requirement of two clock cycles for a write process and three, in case of performing any type of read.

The main concerns surrounding this design are the approach to the reading and process of data, as well the optimization any custom design has in comparison to the use of IP's elaborated by Altera. While this design provides sufficient computation, it still can be improved, incorporating a more continuous reading without the need of stoping every few positions. A use of a DMA, FIFO and register could improve the logic drastically, and also using optimized structure such as Altera's blocks.

4.1.3 DMA Interface

The third and final design explored is the incorporation of three IP's from Altera and a slave interface for the accelerator. Two of these are direct memory access units (DMA)

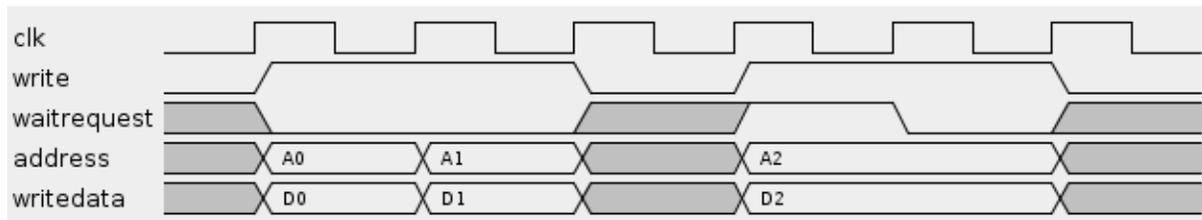


Figure 4.9: Write Timing Diagram for the Master Interface

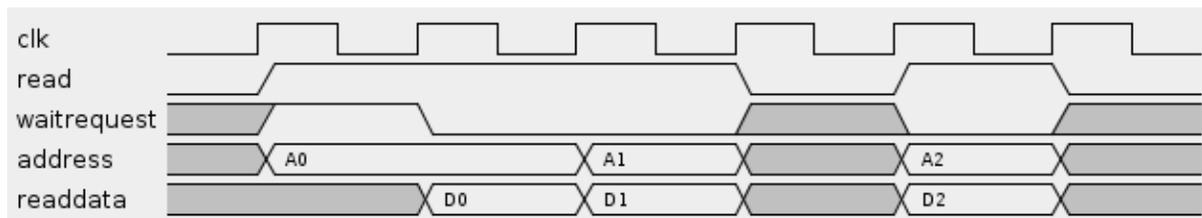


Figure 4.10: Read Timing Diagram for the Master Interface

and a first in first out memory (FIFO). In the layout shown in Figure 4.11, the CPU controls the sending and receiving process by utilizing the control commands for both DMA's. The processing begins by the CPU issue the control for DMA Read in order to perform continuous data transmit from memory to the accelerator, the accelerator interface then gathers at least three data blocks and saves each one in a register. The interface would then passes the information to the accelerator, as soon as the count logic reaches the specified amount of data blocks needed. The interface would then send the output of the accelerator to the FIFO where it will be stored until the DMA Write receives the control signal from the CPU, to start writing to memory from FIFO. Once this process is complete both the initial and output matrices would be present in memory, separated accordingly to specific memory addresses, both also would be accessible by the CPU. The JTAG-UART block in the structure allows to maintain communication with the platform and also acquire data from the input/result matrices to assure behavior. The accelerator interface for this case is a Memory Mapped Slave since it only performs actions for reception and sending of data, the DMA IP already has two masters, so it can execute separately a read and write operation. The already incorporated master logic in this unit facilitates the data flow, allowing read/write actions directly in memory and a completely independent unit from the CPU.

For this structure, the flow diagram in Figure 4.12 shows the application design to run in the processor. The procedures that undergo are:

1. **Packing:** At the beginning of the operation the system starts by making a packing process with the pixel values from the matrix, same as was the case in the slave interface.
2. **For Loop Matrix:** The next step in the software is a cycle representing if all the values that want to be study are pack or not yet done. The cycle is composed of two nested for loops in order to read the matrix in terms of rows and columns. So

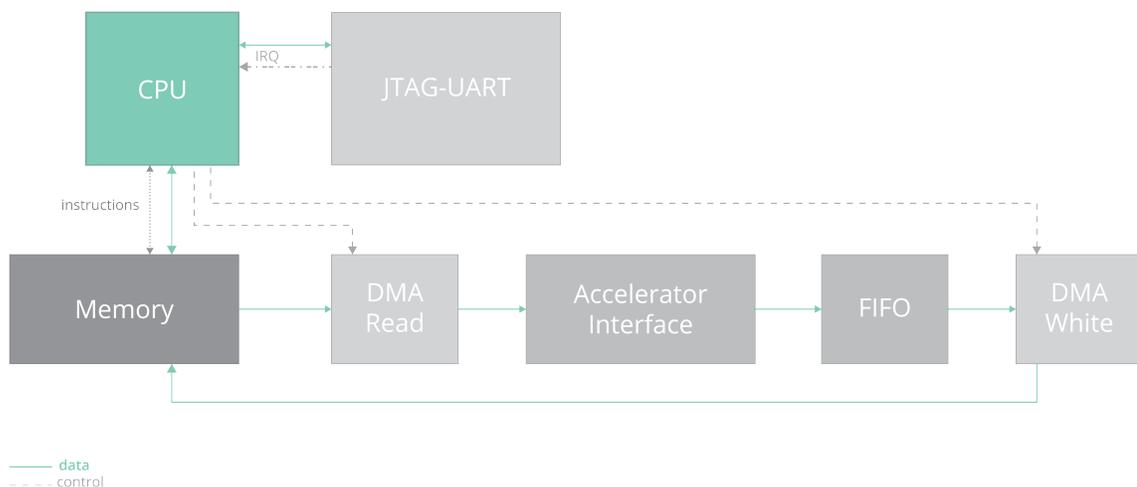


Figure 4.11: DMA Interface for Accelerators

it is possible to read every position that composes the total image, and send it to Memory already pack.

3. **Transmit operation:** In order to pass the information to the accelerator a custom function is design utilizing the NIOS HAL library, specifically the DMA configuration. The function will receive the start and destination address, along with the count, mode and the DMA name. The address gives reference to the specific point in memory where the first pack pixels are stored and the base address where the accelerator is located, accordingly to the Avalon bus. For this last instance is utilized the reference definition in the "system.h" library, for example in a Gauss test this definitions is `DMA_0_WRITE_MASTER_GAUSS3_FIFO_0_BASE`. The count takes the number of bytes to be read from memory in each operation, this data will not stop until all bytes are transmitted. Each memory block pass is composed of four

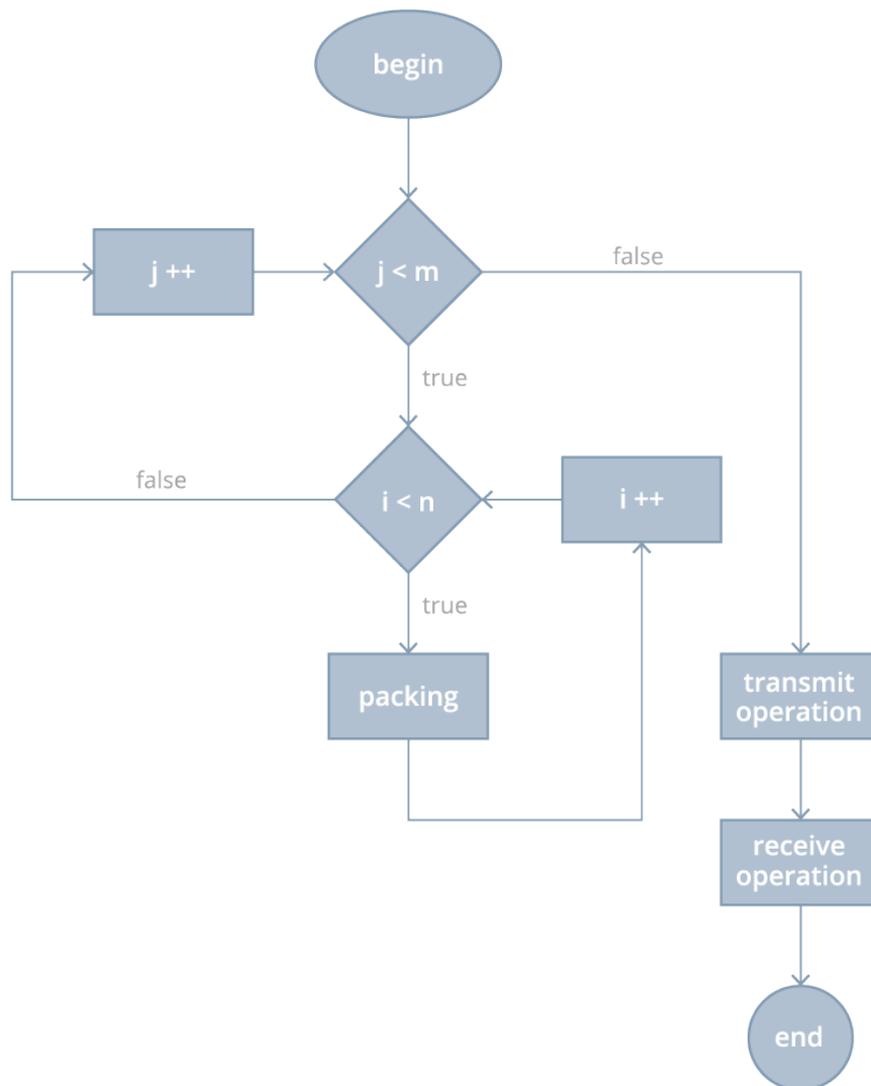


Figure 4.12: Flow Diagram DMA Interface

bytes, since every group is made of 32 bits, the amount of spaces are given by the For Loop Matrix process, and by multiplying this result times four one can get the value of count. The mode makes reference to a configuration specific operation of each DMA, since it can be set to perform transmits ranging from 8 to 64 bits. In every proposed case of study the DMA is configure to 32 bit mode by using the "sys/alt_dma.h" library specifics command ALT_DMA_SET_MODE_32, this command sets the length register to 32 bit. By doing this, the transfer will finish when this amount of bits are passed and then the initial address will increment by four. Initially the DMA is selected, then the function establish the DMA control registers for continuous transmit from memory to a specific hardware component, and to pass blocks of four bytes. Lastly the DMA begins sending data to the accelerator from the specific memory address until the count number is reached.

4. **Receiving operation:** The receiving end of the process begins by getting the same inputs as the transmit operation. Both are similar, just present a couple of variants between one another. At the beginning of this procedure is required the start and destination address, count, mode and DMA name. The count is also given in bytes, but the number is half of the transmit block since for a transmit operation at least two memory positions are needed for retrieving the pixels required. The mode stays the same for both cases, this is due to the fact that in both actions the block are 32 bits. The main difference reside in the DMA name, since in the architecture seen in 4.11, each DMA is specific to a function. The transmit operation makes reference to DMA Read while the receiving is performed by DMA Write. In the receiving end of the memory block the writing starts in the destination address, adding four to each space, accordingly to the specified mode. The start address makes reference to the DMA_1_READ_MASTER_FIFO_0_BASE, which is a static variable also set in the library "system.h", the DMA Write control registers are set to allow streaming from FIFO to memory in order to send data constantly until the count amount is reached finishing this step.
5. **Procedure cycle and application-end:** The NIOS continues the execution of this application until every pixel is read or until a specified amount of matrices is studied.

Finally, this design presents the optimal interaction and characteristics required to work properly in conjunction with the accelerators and the processors. Due to this, the following tests of speed were taken by using this structure. The speed is measure in ticks by adding a Performance Counter, which as the name states is a series of counters specifically used for tracking clock cycles and timing multiple sections of software. This are also found as IP's in the Qsys library and are connected directly to a source clock and the NIOS core, making the rest of the design completely independent. The performance counters are started and ends with the PERF_BEGIN and PERF_END functions respectively. The other two instructions used for setting the counters are PERF_START_MEASURING and

PER_STOP_MEASURING, the first one, resets the counter to initial state values and the second one, completely blocks the counter and deselect its logic.

The results obtained in the measurements using this interface with six different versions of accelerators for the three different filter types and their corresponding software versions, are shown in the following tables. Each table is set with a minimum and maximum cases corresponding two different matrices lengths, resulting in a varying amount for memory addresses needed in DMA read/write operations. The minimum will be of 253 and the maximum of 1012. In Table 4.1, is only shown one single case for maximum values, this is due to the requirement of a larger amount of data in order to compute and compare the results with the hardware implementation.

Table 4.1: Performance Time Filters Software Versions

Filter	Time (ticks)
Laplace8	4615
Gauss	6187
Sobel	5619

Table 4.2 shows the results for Laplace 8 filter, in clock cycles, in this can be seen the difference between the exact versions and the approximate ones. Overall can be seen that for the maximum matrix gain, it range from 37-87 cycles setting version 3 as the slowest and version 2 the fastest. In the minimum case a gain range can be seen from 28 to 84 ticks. This allows to infer version 2 will presents the better performance in a specific end user design. For both matrices cases studied, it can be seen a better performance noting that all approximate versions tend to decrement time in comparison with the exact version. The software version in contrast to any accelerator will present a slower performance by an average near to 500 ticks.

Table 4.2: Performance Time Laplace 8 Accelerators

Accelerator	Time (ticks)	
	Minimum	Maximum
Laplace	3445	4196
Laplace 1	3354	4135
Laplace 2	3361	4109
Laplace 3	3417	4159
Laplace 4	3358	4116
Laplace 5	3353	4112

Moving forward with Gauss filter, the results are presented in Table 4.3. For Gaussian acceleration an overall improvement in the maximum matrix can be seen by an average of 40 cycles, excluding versions 5 which presents a gain of 20 cycles, while comparing exact and inexact versions. It is worth noting that for versions 1 through 4 the difference bear

to be similar, presenting a difference of just a few ticks among one another. Although this does not contradict performance, since every approximate version improve among the exact one, let a gap for advancement, or new version developing for furthering benefit and appliance. Even considering all of this, the acceleration shows a far better performance in contrast to software, since the gain is an average more than 2000 ticks, this is due to the amount of computations needed for a complete Gauss filter application.

Table 4.3: Performance Time Gauss 3x3 Accelerators

Accelerator	Time (ticks)	
	Minimum	Maximum
Gauss	3399	4206
Gauss 1	3394	4168
Gauss 2	3395	4162
Gauss 3	3396	4160
Gauss 4	3406	4164
Gauss 5	3397	4186

Finally, Sobel filtering acceleration is shown in Table 4.4. It can be gathered from it that an approximate acceleration will considerably advance this tool, since an average gain of 100 ticks and 68 ticks, maximum and minimum case respectively, is present in almost all versions with the exception of number 5, in which the gain is of 52 cycles. Version 3 poses the better results from an advancement of 131 ticks, setting this as a better design option. As with software the benefit from utilizing acceleration and approximations, is in average 1500 showing a complete growth in this usage.

Table 4.4: Performance Time Sobel Accelerators

Accelerator	Time (ticks)	
	Minimum	Maximum
Sobel	3417	4238
Sobel 1	3349	4134
Sobel 2	3349	4118
Sobel 3	3349	4109
Sobel 4	3357	4114
Sobel 5	3420	4186

Chapter 5

A Multicore and Multi-Accelerator Platform

5.1 Multi-Core Design

Once the interface design between a computing core and an accelerator is completed and results are gathered, one can proceed with the final integration, involving the fulfillment of the multi-core implementation. This multi-core architecture aims to facilitate a way, in which the accelerator functions independent of each core, and also present a form of interaction that allows a master processor to send messages to each slave, which can use any of the available accelerators. To achieve this goal, first it is analyzed the memory partitioning requirements for every processor, to run on a single block, and the specific components selection that, at the end, would help the accessing of accelerator interface and processor addressing.

For multi-core memory partition is necessary to look back at the program's memory map from Figure 4.2. The space occupied by each program will be the length needed for every core access. In order store all programs, it is required to stack one another, as shown in Figure 5.1. Every partition will begin with the *.text part* and ends with the *.stack*. NIOS II needs to have a special configuration on its reset vector memory and exception vector memory for multi-core development. Between this two vectors it should be presented a memory space of 0x20, in hexadecimal values, and also every core should be set specifically following the memory map proposed. In this design case every program will have a weight of 50 KB in average, making it an address space of 0xC800. Every processor have accessed to this memory both for data and instructions, since the main operating memory and logic flow is stored within this block. Finally, for every boot load operation at the beginning, every core will try accessing this memory simultaneously given permission only by the arbitration logic done by Qsys and the CPU ID configuration. This last one, is present again in the configuration tab in Qsys, allowing to select the ID and priority of each CPU, enabling custom hierarchy. For this case the master is set as the top hierarchy and the

slaves one after the other since its behavior does not affect the end point at the beginning of the running time.

After the partition is set, the message transfer between master slave is develop. Several options are available to be integrated into the platform ranging from I2C protocol to Altera Specific IP's. After considering the actual needs of each CPU, the desired transferred weight should be one data block in which it can be easily obtained by the accelerator. The use of a complete protocol seems to not meet the practical advantages, considering that it lacks the quantity dynamic to be properly introduce as a whole communication protocol. In other instance, the Altera IP presents a Mailbox Core, specifically suited for sending messages between processors, favoring logic and introducing a operation dynamic in which two messages, one know as "command" and the second one as "message", is easily communicated. In this way, the requirement of decoding and integration resides in the software of each core. As useful as this last tool implies to be, it does not favor a more independent interaction between hardware/software, delaying the whole core process by sustaining a wait phase to get messages, in which not only the processor is blocked but the information is not writable or readable, until the register arbitration logic allows it to be. To overcome both options, it is implemented a separated memory block addressable by the master and which the slave core can read from. In this memory, the master will write to a specific address, seated aside for each core, and it will be read at the operating beginning of all slaves. Gathering this way the required identifier for the accelerator to

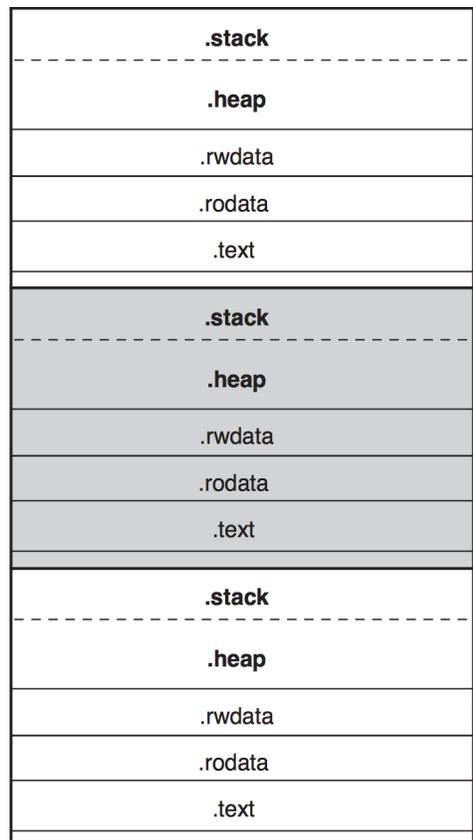


Figure 5.1: Multicore Memory Partition. (taken from [2])

utilize.

Lastly, examining the behavior of a complete structure that allows the access of a single peripheral from multiple masters, it can be noticed a logic checking shortfall over the block use. This could lead to data corruption and multiple selections without answer, being the reason to incorporate a Mutex Core into the system. The mutex core facilitates in each master or processor a policy to utilize the specific hardware target, by incorporating a flag that is set only when the mutex is in used and deselect it when the specific target is finished. Such technique favors the interaction of shared resources in the desired architecture. Since all slave processors will have access to each accelerator this method of flag checking, becomes the most viable option to reduce data corruption and false assertion. Once the mutex core is checked, the running phase for every core can be utilized for both hardware or software operation, stopping undesired delayed and improving results. The point of flag checking is accessing the hardware block but if this is in use, the processor logic will then favor to run the filter code directly from software in the core. Sustaining its operating stages all along the run cycle.

5.2 Proposed Design

The design elaborated is shown in Figure 5.2, in this can be seen the basic structure composed of one CPU Master, and four CPU's slave. Each slaves is label by a number from one to four, this is done for guidance when debugging and downloading the specific software. The scheme continues with the presence of three memory blocks. The signal memory is composed of four memory addresses where the CPU Master will write the correspondent accelerator to be used, ranging from zero to four. Through this the CPU slave will know which mutex to check and by default what accelerator interface to initiate. This signal memory present a bidirectional communication for CPU Master, but just as an input in the case of the slave cores. The program memory will contain every application from the Master and the four Slaves, stacking each other as describe in Figure 5.1. This applications will be the main software for each processor. In all cases both data and instructions paths are allocated to this memory. Continuing with the Data Memory, it will contain the matrices values and specific pixels to be used by the accelerators, also every accelerator interface will perform read and write operation to it. The JTAG-UART component is used as a verification and communication unit, due to the needs for a sustain transmission with the host computer. Usable as well, for downloading and debugging the program in every processor. Moving forward, the diagram also presents the mutex cores which are used for flag checking on each slave, this way assuring the accelerator interface, denoted as Acc.Int. in the diagram, will be selected by the processor requesting it. Every accelerator name goes from one to five, its selection resides completely on the mutex. For example if the CPU 1 is to access the Accelerator Interface 2 it should check first the Mutex 1, and in the message sent from the CPU Master, an integer with the value of "1" would represent the mutex and accelerator to be selected in this case. Finally the

Accelerator Interface will represent the DMA Interface described in the last chapter, each one of this will have access to memory and be controlled by the core posting its request. Any kind of filter can be used in this interface varying just in the requirements of the platform at any given point.

The flow diagram for the CPU Master can be seen in Figure 5.3, it contains the following operations:

1. **Packing:** At the beginning of the operation the system starts by making a packing process with the pixel values from the matrix, same as was the case in the slave interface.
2. **For Loop Matrix:** The next step in the software is a cycle representing if all the values that want to be study are pack or are not yet done. The cycle is composed of two nested for loops in order to read the matrix in terms of rows and columns. So it is possible to read every position that composes the total image, and send it to Memory already pack so it can be then easier to pass the data to the accelerators.
3. **Signal Writing:** Once the packing process is done, the application then set the accelerator for each slave by sending a specific code to the signal memory. This code will be an int value ranging from zero to four. This selection will be given by the

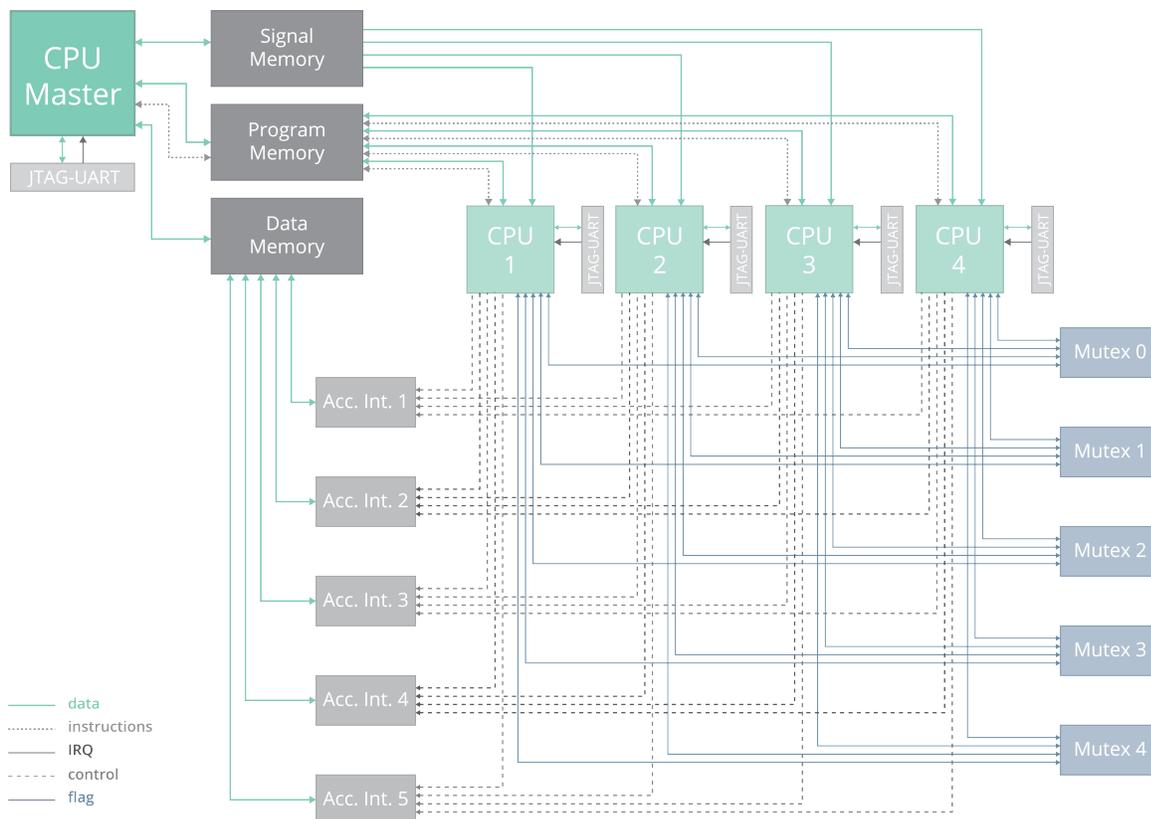


Figure 5.2: Proposed Multicore Design

user, utilizing for this the JTAG-UART NIOS II Terminal window. The terminal interface asks the user to type the accelerator for every CPU, starting from the slave one until four. The application in each core will write to a specific address, where the slave will then read the input. The memory addresses for all slaves are:

- CPU 1: 0x00060000
- CPU 2: 0x00060010
- CPU 3: 0x00060020
- CPU 4: 0x00060030

4. **Procedure cycle and application-end:** The NIOS II Master continues the execution of this application until the terminal session is ended by the user.

Next, the flow diagram for the slave CPU is shown in Figure 5.4. Every slave core follows the same procedure. The main differences resides in the address to be access, describe in the signal writing procedure above, but the rest continues the same way. Because of this, the application will operate as follows:

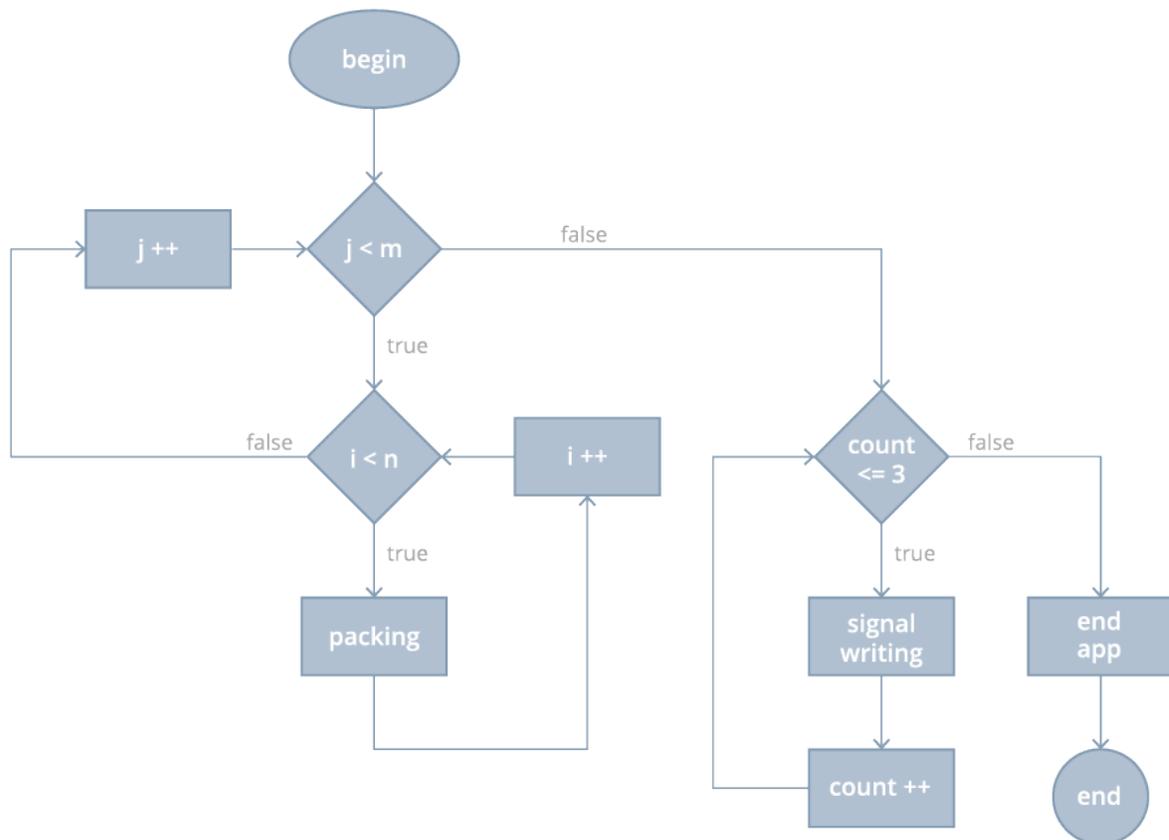


Figure 5.3: Flow Diagrama Master Application

1. **Signal Gathering:** The first step in any slave application is the selection of the appropriate mutex in order use the right accelerator. This is execute by performing an IORD operation into the specific address in Signal Memory, an then applying the result into a integer variable. Once this variable is set, the rest is done in a case statement that enables the core to know which mutex to access and, by default, the accelerator it will use.
2. **Mutex Flag Check:** After the signal is gathered, and the processor now knows which mutex to check, it proceed to verify the selection of this component. If the flag is deasserted then it will select the mutex by locking it, and setting the flag as high for any other core. As this is done, the selection of the accelerator can proceed. But if the core encounters the accelerator in use, it will then has the option to run the software version by software filtering. This way maintaining the resource distribution and running cycles optimization for every core.
3. **Accelerator selection:** Succeeding the mutex flag check, the accelerator selection is performed. In this stage the application will undergo a Case function operation. The message sent by the master is required as an input, then the appropriate accelerator will be chosen and the operation of the DMA Interface can begin.
4. **DMA Accelerator Interface:** Once the mutex is selected, checked and the accelerator is selected, the core will proceed to initiate the DMA interface. This will be divided between two functions one for reading and another for writing in memory, just as described in the last chapter for DMA Interface. This cycle is maintained until every matrix is read and the results are pass back on memory.
5. **Mutex deasserted:** Lastly after the Accelerator Interface is finished, the slave core will then free the mutex flag by an unlock process. Making the accelerator available again for any other core in the system, sustaining the shared resources cycle and running operation.
6. **Procedure cycle and application-end:** The NIOS Master continues the execution of this application until the terminal session is ended by the user.

After all the applications starts its cycle, the core is capable of working completely independent from one another and from the master. Since every accelerator interface is connected to all slaves, each processor will have the possibility of using this component just by checking the mutex core. Once the specific filter its taken, any other core that request control over it will be denied any permission. Leaving, in this case, the software version option as the only viable one. This multi core approach facilitate the user a platform capable of realizing studies both in software and hardware architectures. Measure performance time and preserve the processing logic for both approximate and exact set ups.

It is important to note, that for the behavioral testing of the platform a small number of matrices was used, utilizing a portion of the Figure 3.1. This is due to the presence

of just 150 KB available in the On Chip Memory. In future work this memory could be replaced with a SDRAM off chip making available up to 128 MB, for implementing such component is necessary to design and SDRAM controller or custom interface for directing access from both read and write operation. Another most valuable solution should be incorporating a SD Card interface, from which the CPU Master could perform read and write operations directly on it. Making available only information that cores will require, and overwriting it with the accelerators outputs for each run cycle, reducing access to memory blocks and its usage altogether.

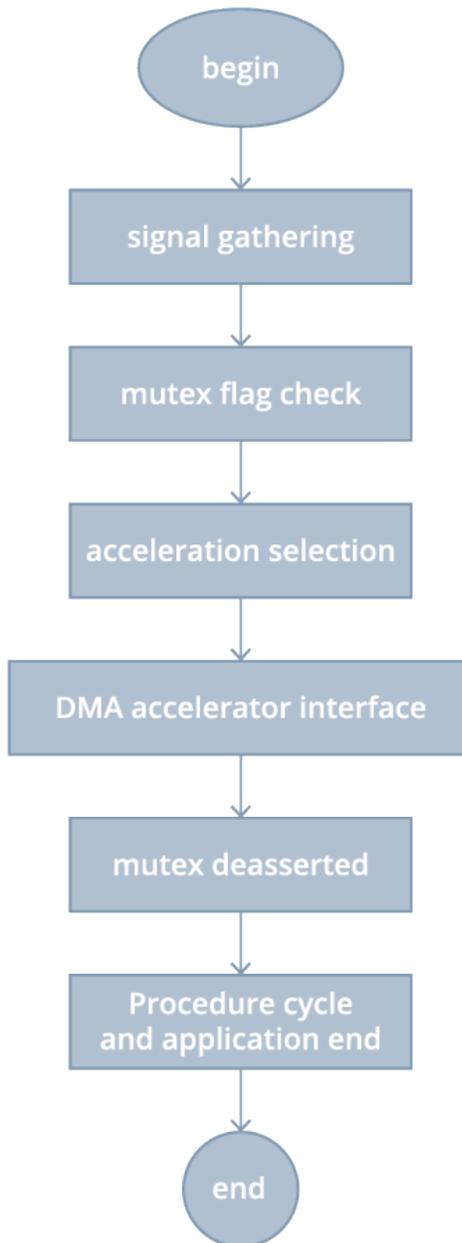


Figure 5.4: Flow Diagrama Slave Application

Chapter 6

Conclusions

In order to tackle the incoming gap designs in newer technologies, it is necessary to study the incorporation of different tolerable characteristics, and how to use them to develop more efficient architectures, improving overall performance. This work presents how different components can be integrated into a specific end user design platforms and the final benefits this provides.

From the exploration done during the development of approximate components and their interfaces incorporation, one can definitely see an improvement in performance and a reduce amount of tasks performed by a processor, specifically the NIOS II softcore. During approximation of the three image processing filters, Laplace, Gauss and Sobel, various benefits can be seen in contrast from both software and exact HDL design. Looking at Laplace layout, version 2 is a better replacement for power reduction, but version 3 has improved delay times. Gauss filtering on other hand, offer an enhanced power estimation utilizing version 5, while version 4 benefits in delay times. Lastly, Sobel acceleration version 2, return higher advancement in terms of power and average times. It is worth noting that area characterization will not represent by itself a superior choice in acceleration constructs, the delay time and power estimation are necessary, as well, to understand full performance integration.

Proposed designs for interfacing contemplate a range of opportunities, using both Altera's master and slave memory mapped interfaces as a start point show advantages for simplicity and communication. But in order to applied these, an extensive custom logic exploration beforehand is needed to accordingly meet the desired constrictions. Moving to the DMA architecture provides a far better optimized structured than custom master logic, assuring behavior and sustaining appropriate interaction all along the running stage, favoring cycles counts and processing for lager data bulks. Using this last interface, a number of test were done showing that design versions 2 and 3 on every accelerator, provides a considerable decrement for each computation in clock cycles. As for comparison between the three approximate hardware filters versions, Laplace 8 and Sobel shows a far better implementation than for Gauss, leaving a gap for restructuring and remodeling for this kernel.

Moving from single core processes, to multi-core and multi-accelerator platforms leads the way to reduce in large amounts computation costs, while maintaining shared resources logic and access. The shared resource access problem is resolved by the mutex core usage, since it enables interactions and blocks any unauthorized entry to pass. Leaving software logic intact and keeping the processor free to operate once the permission is denied. By adopting a platform design capable of using both software and hardware from any processor, that also includes exact and approximate accelerators, demonstrates enhancement in resource usage and data manipulation.

6.1 Future work

This work opens up a door for further, incorporating a larger amount of approximate accelerators varying from model to interfacing. The next step to consider for a deeper research using multi-core platform requires the adding of a SDRAM controller and a SD card reader complement. This is necessary in order to increment available memory within the system, so larger matrices and complete images can be tested. Memory rising gives a lot more variability for higher program loads in each processor, suggesting the advantageous use of custom and laborious libraries. Favoring as well, with the incorporation of the SD card interface, a viable option for output data studying and comparison analysis between platform approximations. Also in acceleration design, the incorporation of a predictor based logic block could help reveal efficiency and error anticipation for approximate components outputs.

Further more than increasing physical capabilities, harder tests are needed with far efficient utilities. Estimation for Power and Delay are optimized in Quartus II, but this does not mean it is perfect. In order to counter it another board or real time measuring system is required to obtained more accurate results.

Bibliography

- [1] Altera. *Avalon Interface Specifications*. Altera, 101 Innovation Drive, San Jose, CA, 13 edition, August 2010.
- [2] Altera. *Creating Multiprocessor Nios II Systems*. Intel, 101 Innovation Drive, San Jose, CA, 11 edition, June 2011.
- [3] Altera. *Nios II Gen2 Software Developer's Handbook*. Intel, 17 edition, May 2017.
- [4] Akash Kumar Jörg Henkel Amit Kumar Singh, Muhammad Shafique. Mapping on multi/many-core systems: Survey of current and emerging trends. *Design Automation Conference (DAC), 50th ACM/EDAC/IEEE*, 2013.
- [5] Sebastian Anthony. Transistors will stop shrinking in 2021, but moore's law will live on [online]. July 2016 [visitado el November 02, 2017]. URL <https://arstechnica.co.uk/gadgets/2016/07/itrs-roadmap-2021-moores-law/>.
- [6] Samuel Thibault Raymond Namyst Cédric Augonnet, Jerome Clet-Ortega. Data-aware task scheduling on multi-accelerator based platforms. In *Parallel and Distributed Systems (ICPADS), IEEE 16th International Conference on*, IEEE, Shanghai, China, 2010.
- [7] Wu chun Feng. The importance of being low power in high performance computing. *CTWatch Quarterly*, 1(3), August 2005.
- [8] J Park B Thwaites H Esmailzadeh D Mahajan, A Yazdanbakhsh. Prediction-based quality control for approximate accelerators. In *Second Workshop on Approximate Computing Across the System Stack (WACAS)*, 2015.
- [9] The Economist. The world's most valuable resource is no longer oil, but data [online]. May 2017 [visitado el November 02, 2017]. URL <https://www.economist.com/news/leaders/21721656-data-economy-demands-new-approach-antitrust-rules-worlds-most-valuable->
- [10] Robert Grou Mame Maria Mbaye Yvon Savaria Eric Granger, Serge Catudal. On current strategies for hardware acceleration of digital image restoration filters. 2004.
- [11] David Geer. Industry trends: Chip makers turn to multicore processors. *IEEE Computer Society Press*, 38(5):11–13, 2005.

- [12] Pam Frost Gorder. Multicore processors for science and engineering. *Computing in Science and Engineering*, 9(2):3–7, 2007.
- [13] Luis Ceze Doug Burger Hadi Esmaeilzadeh, Adrian Sampson. Architecture support for disciplined approximate programming. In *ASPLOS XVII Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312, New York, USA, March 2012. ACM, ACM.
- [14] Jürgen Teich Benno Heigl Heinz Hornegger Hritam Dutta, Frank Hannig. A design methodology for hardware acceleration of adaptive filter algorithms in image processing. *Application-specific Systems, Architectures and Processors (ASAP'06) IEEE*, 2006.
- [15] Michael Gill Beayna Grigorian Glenn Reinman Jason Cong, Mohammad Ali Ghodrati. Architecture support for accelerator-rich cmps. *DAC '12 Proceedings of the 49th Annual Design Automation Conference*, pages 843–849, 2012.
- [16] Honglan Jiang, Jie Han, and Fabrizio Lombardi. A comparative review and evaluation of approximate adders. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI, GLSVLSI '15*, pages 343–348, New York, NY, USA, 2015. ACM. URL <http://doi.acm.org/10.1145/2742060.2743760>.
- [17] Brad Ellison Lauri Minas. The problem of power consumption in servers. 2009.
- [18] Jinghang Liang, Jie Han, and F. Lombardi. New metrics for the reliability of approximate and probabilistic adders. *IEEE Transactions on Computers*, 62(9):1760–1771, 2013.
- [19] Kaizad Mistry. 10 nm technology leadership [online]. 2017 [visitado el November 02, 2017]. URL <https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/03/Kaizad-Mistry-2017-Manufacturing.pdf>.
- [20] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4), 2016.
- [21] Mohamed Abid Mouna Baklouti. Multi-softcore architecture on fpga. *International Journal of Reconfigurable Computing*, 2014.
- [22] Semeen Rehman Walaa El-Harouni Jörg Henkel Muhammad Shafique, Rehan Hafiz. Cross-layer approximate computing: From logic to architectures. In *Design Automation Conference (DAC), 53rd ACM/EDAC/IEEE*, Austin, TX, USA, 2016. IEEE, IEEE.
- [23] Patrick Nelson. Just one autonomous car will use 4,000 gb of data/day [online]. December 2016 [visitado el November 02, 2017]. URL <https://www.networkworld.com/article/3147892/internet/one-autonomous-car-will-use-4000-gb-of-dataday.html>.

- [24] Okan Erdogan Paul M. Belemjian Jin-Woo Kim Michael Chu Russell P. Kraft John F. McDonald Kerry Bernstein Philip Jacob, Aamir Zia. Mitigating memory wall effects in high-clock-rate and multicore cmos 3-d processor memory stacks. In *Proceedings of the IEEE*, volume 97. IEEE, 2009.
- [25] Nam Sung Kim Qiang Xu, Todd Mytkowicz. Approximate computing: A survey. *IEEE Design Test*, 33(1):8–22, 2015.
- [26] Robert J. Hinde Gregory D. Peterson Rick Weber, Akila Gothandaraman. Comparing hardware accelerators in scientific applications: A case study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, 2011.
- [27] David Blaauw Dennis Sylvester Trevor Mudge Ronald G. Dreslinski, Michael Wieckowski. Near-threshold computing: Reclaiming moore’s law through energy efficient integrated circuits. In *IEEE*, volume 98, February 2010.
- [28] Adrian Sampson, James Bornholt, and Luis Ceze. Hardware–software co-design: Not just a cliché. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32, pages 262–273, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [29] Rehan Hafiz Muhammad Shafique Jörg Henkel Sana Mazahir, Osman Hasan. An area-efficient consolidated configurable error correction for approximate hardware accelerators. *Design Automation Conference (DAC), 2016 53rd ACM/EDAC/IEEE*, 2016.
- [30] Tom Simonite. Why a chip that’s bad at math can help computers tackle harder problems. *MIT Technology Review*, 2016.
- [31] Wolfram Hardt Stephan Blokzyl, Matthias Vodel. A hardware-accelerated real-time image processing concept for high-resolution eo sensors. *Deutscher Luft- und Raumfahrtkongress*, 2012.
- [32] Michael Taylor. A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19, 2013.
- [33] Andre Baixo Mark Wyse Ben Ransford Jacob Nelson Luis Ceze Mark Oskin Thierry Moreau, Adrian Sampson. Compilation and hardware support for approximate acceleration. In *TECHCON*, 2015.
- [34] Jacob Nelson Adrian Sampson Hadi Esmaeilzadeh Luis Ceze Mark Oskin Thierry Moreau, Mark Wyse. Snnap: Approximate computing on programmable socs via neural acceleration. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Burlingame, CA, USA, February 2015. IEEE.

-
- [35] Scott E. Thompson and Srivatsan Parthasarathy. Moore's law: the future of si microelectronics. *materialstoday*, 9(6):20–25, June 2006.
- [36] Kaushik Roy Anand Raghunathan Vinay K. Chippa, Srimat T. Chakradhar. Analysis and characterization of inherent application resilience for approximate computing. *Proceedings of the 50th Annual Design Automation Conference on - DAC*, 2013.
- [37] Sally A. Mckee Wm. A. Wulf. Hitting the memory wall: Implications of the obvious. Technical report, University of Virginia Charlottesville, 1994.
- [38] Faycal Bensaali Arti Mishra Xiaojun Zhai, Fadi Jaber. Hardware acceleration of an image processing system for dielectrophoretic loading of single neurons inside micro-wells of microelectrode arrays. *17th UKSim-AMSS International Conference on Modelling and Simulation (UKSim)*, 2015.