Tecnológico de Costa Rica

Computer Engineering Department

Licentiate Degree Program in Computer Engineering

TEC Tecnológico de Costa Rica

Approximate kernel generation for convolutional neural networks using OpenCL

Final report submitted in partial fulfillment of the requirements for the degree of Licentiate in Computer Engineering

Esteban Calvo Vargas

Cartago, November 19, 2019

a mis queridos padres

Agradecimientos

A todos aquellos compañeros y profesores que me han acompañado durante mi tiempo en la universidad, desde mi inicio en la carrera de Mecatrónica hasta la finalización de mis estudios en Computadores.

Al profesor Jorge Castro Godínez por la oportunidad que me dio de trabajar en el presente trabajo y por compartir sus conocimientos conmigo.

A Natalia, por estar siempre presente en todo momento de mi carrera y ser el más grande apoyo que he tenido.

A mis papás y hermanos, sin el amor, paciencia y sacrificios realizados yo no hubiera podido llegar tan lejos.

Esteban Calvo Vargas

Cartago, November 19, 2019

Resumen

El uso de FPGAs para la aceleración de aplicaciones de aprendizaje de máquina es un área en desarollo debido al alto nivel de especialización de este tipo de plataformas. Otra técnica de aceleración de aplicaciones es la computación aproximada. El presente trabajo muestra técnicas de computación aproximada siendo utilizadas para dirigir el diseño de una implementación en FPGA de redes neuronales convolucionales por medio de OpenCL, una herramienta para la definición de procesos en plataformas heterogéneas. Se demuestra cómo la combinación de ambas técnicas puede llevar a ganancias significativas en tiempo de ejecución y uso de recursos de la plataforma con pequeñas pérdidas en exactitud.

Palabras clave: computación aproximada, *hardware* aproximado, FPGA, aprendizaje profundo, OpenCL, *High-level Synthesis*

Abstract

The use of FPGAs for accelerating Machine Learning applications is a research area due to the high specialization level of this type of platforms. Another technique for application acceleration is approximate computing. The current work shows approximate computing techniques being used to guide the design of an approximate CNN implementation on an FPGA through the use of OpenCL, a framework for process definition on heterogenous platforms. It is shown how the combination of both techniques can lead to significant gains on execution time and resource usage with little losses on accuracy.

Keywords: approximate computing, approximate hardware, FPGA, deep learning, OpenCL, High-level Synthesis

Contents

List of Figures

Table index

List of abbreviations

Abbreviations

CES	Chair for Embedded Systems
CNN	Convolutional Neural Network
DNN	Deep neural Network
FPGA	Field-Programmable Gate Array
KIT	Karlsruher Institut für Technologie
ML	Machine Learning

Chapter 1

Introduction

Computers were invented to accelerate manual processes. In the beginning, computers offered speeds far superior than what a human could manage on specific activities, but their use was limited to relatively low data processing.

In the current era of increasingly advanced semiconductor technologies, a need for using computers with applications for high-level data processing has risen along with higher amounts of data to process. This has generated a race for maintaining high performance while keeping equal or even reduced energy, time and storage consumption. The main solution, for some years, has been to increase the computing power of the computers via mechanisms such as, for example, increasing the number of transistors per area. This solution, however, has brought with it a lot of considerations and problems, specially regarding energy consumption.

A new approach has surfaced as one of the solutions to the energy consumption problem is, approximate computing. This computing paradigm was born on the assumption that there are cases on which an exact result, with high precision, is not needed. A lot of data comes from inexact sources (sensors, readings) or do not require a precise processing algorithm (ML, user recommendation programs, statistics). This type of applications is known as error-tolerant. Approximate computing looks to use these types of data to create algorithms, languages, compilers, circuits and computer architectures that have the common objective of lowering the energy consumption and increasing the performance at the cost of having an approximate result.

Machine Learning is a current area of research that can take advantage of the benefits of approximate computing. Its main property is decision making based on processing big amounts of data. This data can be written, visual (images, video) or audio information, as well as taking the feedback into account to improve the learning process. Approximate computing can take advantage of this fact and use it to reduce the computational effort required and, in this manner, improve significantly the investigation area of ML.

This work looks to explore methods and techniques for approximate hardware definition on FPGA, such that it could be used on ML applications, specifically by generating approximate hardware kernels for CNNs through the use of OpenCL. The current project will deliver useful tools for any other developer that requires to accelerate their ML algorithms through the use of FPGAs.

1.1 Project background

In this section the background of the project is presented, including: the organization in which the project was developed, the knowledge area of the project in relationship to computer engineering and similar works that serve as a starting point for the project.

1.1.1 Organization

The project is developed in the KIT, a university focused on the development of technology and science. KIT was created on 2009 after the convergence of the University of Karlsruhe, funded on 1825, and the Investigation Center of Karlsruhe. It is located in Karlsruhe, in the Baden-Württemberg state, to the southwest of Germany. Figure 1.1 shows a map with the location of Baden-Württemberg inside the european country and the location of Karlsruhe inside the aforementioned state.

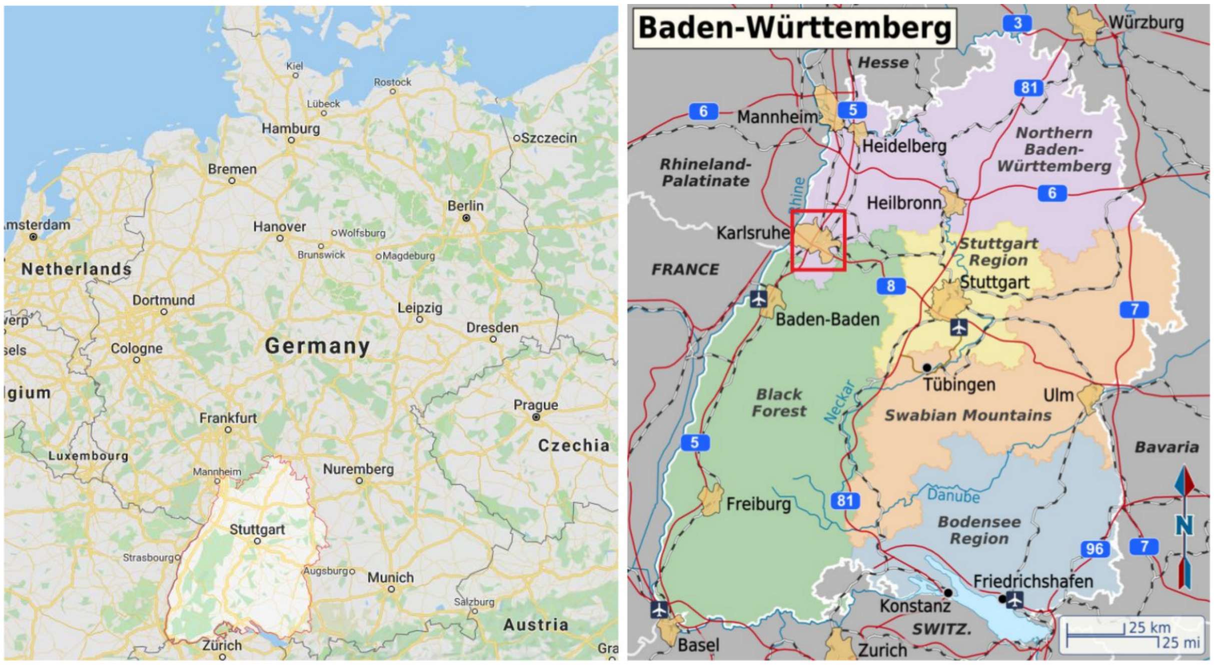

Figure 1.1: Map of Germany with Baden-Wüttemberg marked (left) [13] and location of Karlsruhe within the state (right) [23]

Nowadays, KIT is one of the most prestigious technical universities in Germany, specializing on engineering and science. Figure 1.2 shows the current logo of the university. KIT is conformed by a scientific organization separated by divisions:

Figure 1.2: Logo of the Karlsruher Institut für Technologie [21]

- Division I: Biology, Chemistry and Process Engineering.
- Division II: Informatics, Economy and Society.
- Division III: Mechanical and Electrical Engineering.
- Division IV: Natural and Built Environment.
- Division V: Physics and Mathematics.

These divisions are constituted by departments destined to investigational work, innovation and teaching. Each department has institutes responsible for university education. Aside from the institutes, there are KIT centers, on which topics related to investigation and innovation go beyond the divisions, supporting an interdisciplinary cooperation [22].

The Department of Informatics is one of the first to be established on Germany. This department is formed by different institutes focused on teaching and investigation of topics associated with informatics. As part of the Department of Information the CES comes as a center of investigation on aspects related with the design of embedded systems, from the reliability of electrical circuits to the management of electrical power on systems with multiple and many cores.

1.1.2 Knowledge area

The project is developed within the technical area of approximate computing, which is part of the areas of interest of computer engineering. Approximate computing looks to relax the numerical equivalence between the specification and implementation of such applications promising significant energy-efficiency improvements and it has gained significant traction over the past few years [51]. Within approximate computing, the project focuses on the utilization of Intel's® FPGA SDK for OpenCL™to develop approximate hardware kernels and their application on CNNs. The goal is to highlight the effect of these kernels on the improvement in performance and reduction on resource usage in comparison to exact kernels.

This kernels must be designed using high-level synthesis tools. OpenCL is a tool that allows to define procedures on heterogeneous platforms through the use of a high-level programming language. The knowledge on hardware changes via the use of a software description is necessary to develop the project.

1.1.3 Similar works

In the area of neural network implementation on FPGA using OpenCL, notable previous works are:

- Suda et al. introduce a complete OpenCL-based accelerator design for CNNs based on matrix multiplication. They also explain algorithms to apply convolution by rearranging the components of the input image. This work is an exact CNN that could be used as a base for the current project. [42]
- Wang et al. propose a CNN implementation on FPGA specifically taking advantage of the ability to allow communication between layers through data channels [48].
- Zhang and Li present a model to analyze the resource usage on an FPGA, as well as implementing their own accelerator based on OpenCL [54].

Regarding the use of approximate computing for CNNs, we can find the following related works:

- Moons et al. show that using approximations on popular CNNs can increase the energy efficiency of the accelerators and show some of the techniques that can be applied to other CNNs [31].
- Kamel et al. compile a lot of techniques that can be used on CNNs and some other information regarding the application of CNNs on FPGAs [3].

Also, there is the work done by D. Spies for his masters degree on the KIT on which some CNN frameworks were implemented for low-power FPGAs, reducing the need for high-end FPGAs and allowing for further research on the area.

1.2 Problem statement

This section looks to expose the main problem the project is trying to solve as well as why approximate computing and FPGA implementation of CNNs can be a solution to the problem.

1.2.1 Problem context

In the last years, discussions on the physical (and economical) limits of the very-large-scale integration of transistors have surfaced [26][27], where statements like Moore's Law exert pressure the big chip manufaturers. Gordon Moore himself has assured that this trend cannot be maintained for a long time [10]. This leads to the search of new paradigms or techniques that allow to sustain the high requirements on performance and energy consumption of applications in the technological world, where a need to process even higher amounts of data is increasing. Some solutions to this increasing demand have appeared, such as multicore computers, multithreading architectures, large scale computers, GPU processing and more.

Despite these solutions, there exist other problems that cannot be solved just by improving on the architecture of the processor. Some of these problems are:

- The memory wall: Wulf and McKee [50] describe an imminent problem on which the superior speed increase of the processors is a lot higher that the improvement on memory technologies. This requires solutions that try to reduce the amount of memory accesses.
- The utilization wall: Taylor et al. [47] noticed a phenomenon that appears due to the increase on the amount of transistors per unit of area in a chip. The problem is related to an exponential reduction of the usable percentage of the chip depending on the scale of integration of the transistors.
- Problems with thermal dissipation: with the increase of frequency of the microprocessors, the level of heat dissipation has increased, forcing a reduction on the operational voltage. Some solutions have been proposed to this problem, like the utilization of multicore processors with cores that deactivate themselves to reduce the workload [16].

Approximate computing tries to solve this type of problems with the help of the recent increase of error-resilient applications (e.g. [43]). This paradigm tries to eliminate, or reduce, the need for precision during processing with the goal of obtaining gains in energy efficiency and processing speed.

Furthermore, one area of interest within the error-resilient application world is Machine Learning. The goal of this area is to allow a computational system to execute tasks without ML for a prior specific programming and, in some cases, using previous results as feedback to improve the execution. Algorithms used in ML look to build mathematical models based on ”training” data to perform tasks without explicit programming [7]. Due to their nature, ML applications do not present an exact response immediately, or even never, instead they require multiple feedback cycles to achieve the expected response. This means that approximate computing is a potential paradigm to work with this type of applications [5].

1.2.2 Justification of the problem

Approximate computing is an area that is still in the upswing. There are diverse investigations and designs that look to take advantage of the existence of error-tolerant applications. However, it is necessary to continue advancing the paradigm and develop new techniques in order to observe its contributions in daily life. The importance of this paradigm resides on the fact that it does not depend on the current state of the technology to bring forth energetic and performance improvements.

The use of FPGA in the area of approximate computing is still under explored. At the same time, machine learning applications are of great interest to a lot of fields. An improvement on current ML implementations through the combination of both areas would generate new opportunities and solutions with low energetic consumption and high adaptability levels. Following this thought, the project is important due to the following:

- It allows to advance the investigation on the area of approximate hardware definition using popular tools such as OpenCL, which could be used as the basis for future investigations with FPGA based applications.
- The use of a popular and easy-to-use tool could speed up the generation of results in a less explored area, such as FPGA implemented neural networks.
- The project generates new tools easily adaptable to ML applications based on neural networks. These tools can be individual kernels or sets of customizable kernels.
- New investigations on error tolerant applications and approximate computing can easily make use of the findings of this project.

1.2.3 Problem definition

Machine learning applications are based on different methods used to obtain results. One of the most used techniques are called neural networks, with the objective of imitating the work done by the human brain to obtain similar results. Furthermore, there is a branch of neural networks called deep learning. It consists in the increase of the amount of processing layer in order to obtain a bigger amount of details from an input of data [37]. Layers are represented by nodes (neurons) that process data from the input. This processing requires a high level of computation, but its results are mostly approximations. CNNs are part of the deep learning neural network implementations, with most of its usage focused on analyzing input images.

Now, FPGAs are devices that have been recently the object of investigations with the objective of accelerating the current processing being done. The interest in using FPGAs for high amounts of computation comes from the limitation of general use CPUs, which do not offer enough processing power (multiple operations with high level of complexity at the same time); and GPUs that, even with the capability of processing high amounts of data at the same time, are not specialized enough or even customizable to perform specific tasks. The use of a device that can be programmed at the hardware level for specific tasks (in this case, neural networks) offers possibilities on processing speed and energy efficiency that surpass GPUs [15].

The current project emerges with the objective of contributing to the ongoing investigation efforts in the area of FPGAs for machine learning applications through approximate hardware definition. Each neuron in a neural network is represented as a “hardware kernel" that gets defined the calculations and processes that it needs to do on the input data to analyze it. Through the use of software tools like OpenCL, it is possible to define kernels that perform machine learning tasks. Combining all these areas and tools, an improvement on performance and energy consumption must be possible, allowing machine learning applications to keep up with the global demand for higher amounts of data processing.

1.3 Objectives

The project tries to find a solution to the ever increasing energy consumption problem of current world technologies. This is accomplished by using low energy platforms such as FPGAs applied to CNNs with the use of high-level definition tools like OpenCL.

1.3.1 Main objective

Design an approximate hardware implementation of CNNs through the use of Intel's® FPGA SDK for OpenCL™.

1.3.2 Specific objectives

- Describe the viable changes on the OpenCL tool for approximate hardware generation on FPGA.
- Create approximate and reusable hardware kernels for CNNs using OpenCL.
- Determine the error-tolerance level of CNNs when using approximate kernels instead of exact kernels.
- Ascertain the reduction of computational resources when using approximate kernels compared to a base FPGA implementation of the CNN.

1.4 Scope, deliverables and limitations

The scope and deliverables of the project are defined in this section with the limitations the student faced during its development.

1.4.1 Scope

The project's end goal is designing, developing and implementing approximate hardware kernels on FPGA. These kernels represent each of the layers of a CNN and will allow for an image processing algorithm to be performed on them. The end result must be more energy efficient and provide better performance over existing implementations of CNNs.

The project is subdivided in the following stages:

Theoretical investigation

A first step is to investigate on the following topics:

- Convolutional neural networks.
- OpenCL and hardware definition.
- Approximate computing on neural networks.
- CNN implementation on FPGAs.

This will provide the necessary information to use OpenCL to implement CNNs on FPGA. Futhermore, this information can also be used to generate mathematical models to measure the error and precision of the generated CNNs, be them exact of approximate, as well as the comparison between both implementations.

Exact CNN implementation on FPGA

An exact CNN implementation must be developed in order to have a baseline to develop the approximate kernels. This exact CNN implementation is also used to compare results in order to evidence the performance and energy efficiency improvements. The implementation can be developed on different hardware types such as CPU-based, GPU-based or a combination of both; the scope of the project is limited to the CPU implementation. Finally, the approximate CNN implementation must be a modification of the exact model.

Approximate CNN implementation on FPGA

This implementation must be completely based of the exact CNN implementation and will reflect the OpenCL changes that allow for a different hardware definition, one that is approximate and adds errors in the processing of the input data. Also, the approximate implementation must have a set maximum error, which should be measurable and used in the mathematical models.

Comparison against base implementation

Using the results of both the exact and approximate implementation, an analysis of the effects of approximate modifications against the exact implementation must be performed. This analysis should reflect which approximate changes are better in terms of performance gain, reduction of resource usage and accuracy loss.

1.4.2 Deliverables

Most of the deliverables listed here are located within the rest of this project report. Source code can be found at the following link. Minutes of meetings and the current report are kept here.

1. Theoretical investigation

    1.1. Compilation information of the use of neural networks on FPGA: theoretical background chapter of this report will contain relevant information to be used in the rest of the project and will guide the decision making when designing neural networks on FPGA.

    1.2. Report with viable approximate modifications: the solution chapter will have the modification possibilities with OpenCL and that affect the hardware definition on FPGA.

2. Design stage

    2.1. Documentation on how to approximate neural networks: contains the necessary information on the principles of approximate computing that are applicable on neural networks, specifically for CNNs. It should contain mathematical models that can be compared against the results of the project.

    2.2. Design of the error calculation model: contains the mathematical formulations that allow to create precise calculations of the obtained results so they can be compared with the gain in performance.

3. Code development

    3.1. Source code: source code of the OpenCL application and any other code necessary to compile, execute and test the kernels.

    3.2. Exact kernels: source code of the exact kernels used as a base line for the approximate kernels.

    3.3. Approximate kernels: source code of the approximate hardware kernels that will be used on the testing phase.

4. Testing stage

    4.1. Results of the exact tests: measurements of performance, precision and resource usage while testing the exact kernels.

    4.2. Results of the approximate tests: measurements of performance, precision and resource usage while testing the approximate kernels.

    4.3. Results documentation: comparison of the results between exact and approximate tests.

5. Project administration

5.1. Design documentation: contains the design of every tool developed during the project.

5.2. Meeting minutes: contains all information obtained in the progress and validation meetings.

5.3. Final report: final report of the project with all relevant information that was generated during its course.

1.4.3 Limitations

LIM-01: the student must mobilize to Germany in order to reduce conflicts with the project supervisor. Because of the lack of a visa, the time in the european country is limited to 3 months.

LIM-02: the student's budget is only 1000 euros during the stay in Germany. This money will only cover feeding, staying and commuting costs, limiting the possibility of buying new tools to develop the project. The student must use any tools that the supervisor or the university can give him.

LIM-03: the student must comply with the regulations and limitations of the KIT regarding international students.

LIM-04: no face-to-face meetings are possible with the advisor teacher due to the geographical location of the student during the project. Due to timezone differences, the available times for meetings is limited to 8 hours or less per day and the meetings depend on having good internet service and working audio equipment.

LIM-05: as the student does not have access to a computer with a middle to high end GPU, the base software implementation of the CNN is limited to CPU only.

LIM-06: some of the modifications to be done on the CNN are changes to the original definition of the network. These changes may require a new training phase for the network which represents weeks on training. Any approximation proposed must work with the existing training weights for the selected CNN.

LIM-07: the exact CNN implementation is CPU only. No other processing units such as GPUs are used, limiting the scope of the results and comparisons done against the approximate implementation.

LIM-08: the FPGA implementation is done on a DE1-SoC board, containing a Cyclone V as the FPGA chip. This is not a powerful FPGA to test big changes on, limiting the size of the approximate modifications that can be used.

Chapter 2

Theoretical framework

2.1 Convolutional neural networks

A standard neural network consists of many simple, connected processors called neurons, each producing a sequence of real-valued activations. Input neurons get activated through sensors perceiving the environment, other neurons get activated through weighted connections from previously active neurons.

Learning is about finding specific weights that make the network exhibit desired behavior, such as driving a car. Depending on the problem and how the neurons are connected, such behavior may require long causal chains of computational stages, where each stage transforms the aggregate activation of the network. Deep Learning is an area of ML that tries to replicate this behavior by increasing the number of stages [38]. Figure 2.1 shows the difference between regular NNs and deep NNs.

CNNs are part of deep learning, a type of neural network based on the visual system

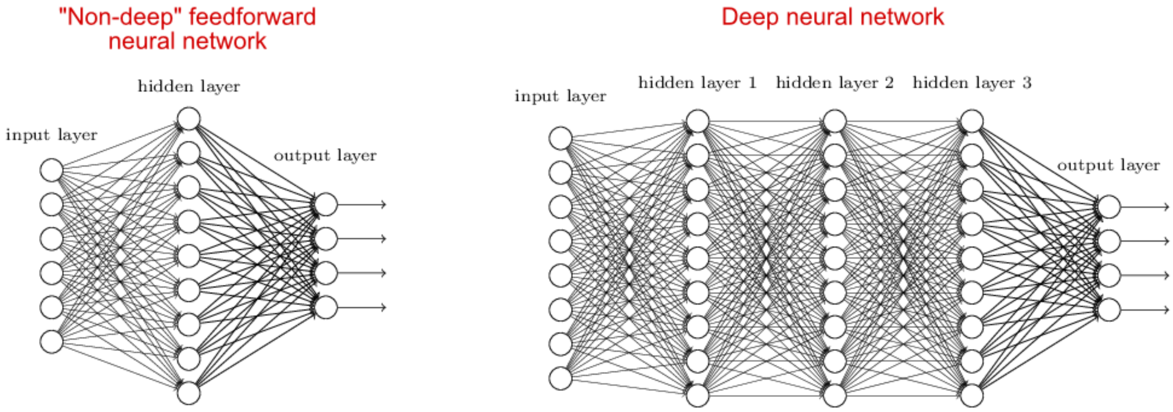

Figure 2.1: Illustration of the difference between a regular neural network and a deep neural network [33]

of mammals [11][17]. A CNN is usually comprised of three types of layers: convolution, pooling and fully-connected [19]. There could also be normalization and activation phases.

2.1.1 Convolution layer

The most important layer in CNNs, as it represents the main operation done throughout the layers of the network and makes it possible to highlight features of the input image. It is based on the convolution operation, defined as such:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.1}$$

However, in CNNs the input image and output result are typically multidimensional arrays with a defined size, which means that convolution must be implemented as a discrete function. Assuming a two-dimensional input I and an applied two-dimensional filter (kernel) the resulting operation is defined as following:

$$S(i,j) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i-m,j-n)K(m,n) \tag{2.2}$$

If the kernels are smaller than the input image, each of them will only filter a specific window of the image at a time while sliding this window to cover the full image, generating an activation map. Also, because the images are color images, the input will be a three-dimensional matrix which can be transformed (flattened and rearranged) and processed as a matrix multiplication following the process found on [42].

This sliding window will allow to cover the whole input image by moving a certain number of pixels after every operation, this number is called stride. Also, because the filter operates on every pixel of the window and to ensure a proper output size, a border of ceros could be added to the input image with a specific width, called zero-padding [19]. The last parameter to take into account is the number of kernels to use per layer. Another relevant term is a feature map, which is any output of the layers conforming the CNN [12].

Suda et al. define the equation that describes the output of a single neuron convolution operation applied to a specific (x,y) position on the input matrix as:

$$out(f_o,x,y) = \sum_{f_i=0}^{N_{if}} \sum_{k_x=0}^{K} \sum_{k_y}^{K} wt(f_o,f_i,k_x,k_y) in(f_i,x+k_x,y+k_y) \tag{2.3}$$

Where $out(f_o,x,y)$ and $in(fi,x,y)$ represent the neurons at location (x,y) in the feature maps f_o and f_i, respectively and $wt(f_o,f_i,k_x,k_y)$ are the weights at position (k_x,k_y) that gets convolved with input feature map f_i to get the output feature map f_o. This is done for all N_{if} input features [42].

2.1.2 Pooling Layer

A pooling function replaces the output of the NN at a certain location with a summary of the nearby outputs. For example, the max pooling [55] operation reports the maximum output within a rectangular neighborhood. The objective is to discard unnecessary details in the input matrix and preserving the important information.

Just like convolution, a sliding window is used and the pooling operation is performed for each movement of the window. This process reduces the output size (downsampling) [19].

2.1.3 Fully-connected Layer

A fully-connected layer is similar to convolution with the difference being that a convolution layer is connected only to a local region of the input matrix, while a fully-connected one operates over every activation in the previous layer. Their objective is to take every feature found by the other layers and classify them depending on the objects the CNN is trying to find, that is why they need all information gathered by the previous layers. For this reason, fully-connected layers are generally found at the end of the neural network architecture.

2.1.4 Normalization

A normalization layer is common in CNN implementations to speed up convergence, accelerating the process of getting the correct answer when training and evaluating. A common form of normalization is local response normalization (LRN), defined using the equation 2.4, found on Krizhevsky's paper [24].

$$out(f_o,x,y) = \frac{in(f_o,x,y)}{\left(k + \alpha \sum_{f_i=max(0,i-n/2)}^{min(N-1,i+n/2)} in^2(f_i,x,y)\right)^{\beta}} \tag{2.4}$$

α, β, k and n are constants determined by a validation set, while N is the amount of neurons per layer.

2.1.5 Examples of CNN architectures

One of the first successful CNN was LeNet, which was used for number detection and classification on zip codes, digits and more. It consisted of three convolution layers, two subsampling layers similar to pooling and one single fully-connected layer at the end [28].

VGG is another successful CNN which features up to 16 convolution layers, as well as using max-pooling and fully-connected layers [40]. It shows that increasing the depth of conventional CNNs is enough to achieve the necessary performance needed for large-scale image classification.

One of the most import CNNs developed in recent years is the one developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. Known as AlexNet, it innovated by stacking convolutional layers and increasing the depth of the CNN [24]. This project uses a partially modified version of AlexNet.

2.2 Approximate computing

Approximate computing is a paradigm that tries to take advantage of the fact that many applications can still work properly even with errors being introduced, be them manually or by accident. Chippa et al. [8] show that current world applications are error-resilient, meaning that they can withstand the introduction of errors while having meaningful outputs.

Approximate computing can be applied to many fields of computer programming and organization stack. There can be approximate programming languages that introduce imprecise algorithms, to approximate compilers that generate imprecise but fast machine code. There are also approximate computer architectures, going down into approximate memory and hardware that allow for less energy consumption and better performance [51].

2.2.1 Approximate hardware definition

Using approximate hardware is a good approach when having access to transistor-level definition. There have been efforts that try to design approximate logical units [20][53] which could improve the energy levels of existing arithmetic units. But this approach is not viable when working with FPGAs as we can only change the functionality of the hardware instead of its electric properties or composition.

For high-level synthesis, introducing errors must be done partially, as not every part of the hardware can be imprecise. Some of the most important approaches have been:

- Finding the minimum area under a given error rate constraint. Shin and Gupta show that an error of at least 1% can lead to reductions of up to 9% in the area utilized by the design [39].
- Using an iterative process to improve the synthesis by taking into account the error magnitude [30].
- SALSA. A methodology to allow for better control of the quality constraints and transforming the approximate hardware synthesis problem into a regular synthesis

problem [46]. There is also an extension to it called ASLAN that works for sequential circuits [34].

- Axilog is an extension to Verilog that allows the designer to choose which parts of the design can be approximate or exact [52].
- Working on fixed point arithmetic can be analyzed and bitwidth optimizations applied for precise control of the accuracy of the output [29].

2.2.2 Approximate neural networks

Agrawal et al. demonstrate that DNNs are resilient to numerical errors from approximate computing. By removing random computations on the convolution layers and using single-precision floating-point number representation, they achieved an error of about 18% [5].

Moreau et al. created a neural accelerator, called SNNAP, that approximates the filter functions and communications between each component within a FPGA-based application [32]. This can be used to generate approximate kernels adapting the SNNAP replacements to OpenCL and the neural network definition.

Another technique is the use of fixed-point arithmetic with less bitwidth for the data being computed inside the neural network. Wang et al. showed that a neural network can be trained using 8-bit floating point numbers without reduced accuracy on the final result [49]. Using fixed-point arithmetic on different layers could be used to change the precision of each of the layers, adjusting them and experimenting to obtain the best results energywise while trying to maintain an accepted output.

Most of the current CNNs have a lot of layers for fine tuning the output classification. Simonyan presents various CNN models with different number of layers, finally setting for 16 convolution layers and 3 fully-connected layers [40]. Looking at the results, the other configurations with less layers could be used in order to improve performance, reduce implementation area and reduce energy consumption, while the error is only reduced a little after many layers are included.

2.2.3 Approximate FPGA implementations

Lopes et al. explore the opportunities to explore on playing with the number of iterations on FPGA computations [35]. By using iterative solutions to linear systems, the final precision of each calculation and the accuracy can be adjusted by executing less or more iterations. This can be used in conjunction with the operations defined by the neural network kernels to finely control the precision of each layer or the whole network.

As already mentioned, Moreau's neural accelerator can be used to approximate functions implemented on FPGAs. This approach allows to create a specific implementation that does not interfere with other techniques. There is also the framework introduced by

Sampson [36], which enables programmers to manually adjust the approximation while providing automation for the process. Sampson showed good results using the framework on a SoC with an FPGA as an approximate accelerator.

There can also be optimization on the arithmetic operations defined on each of the functions to be run on the FPGA. Approximate multipliers can be used to accelerate the performance of matrix multiplications, convolution and other operations that use multiplications [45]. This can be improved upon by ways of memoization, a technique that stores a cache for expensive calculations and returns this cache whenever those calculations are repeated. Using memoization has already been proved to work on FPGA implementations [41] and, while it comes with a small cost on area, it can be used to speed up calculations by large amounts.

2.3 Intel® FPGA SDK for OpenCL™

The use of heterogeneous devices, from general-purpose GPUs to FPGAS, to execute ML algorithms and neural networks is of big interest in the current world. There are current efforts to bring all types of devices together by taking advantage of each device strengths [1].

OpenCL is a framework developed by Apple Inc. that allows for a seamless programming experience for heterogeneous devices. OpenCL is based on the C++ 14, including most constructs from C++ in order to work as a clone of it. This allows high-level programmers to develop applications that can be run on different platforms without having to create new code for any new platform.

Intel® FPGA SDK for OpenCL™is a development environment that allows the creation of synthesizable kernels, blocks of code that can be executed within the FPGA. Because of the high-level focus of OpenCL, it can be used to design in short time big application such as CNNs. As for its effectiveness, it has been demonstrated that using OpenCL instead of low-level languages such as Verilog does not yield bad performance results, at the cost of a little more memory usage [2].

OpenCL requires two different pieces of code: a host code and a device code. The host code is in charge of determining the device on which the main computation is going to run, prepare the data to be processed and, in the case of FPGAs, run the programming process in order to start the execution. The device code is the code that is programmed into the FPGA and represents the hardware execution process.

Suda et al. [42] implement an FPGA accelerator for CNNs. They take advantage of the parallelism offered by OpenCL in order to set vectorization and loop unrolling factors as to have better control of the actual implementation on the FPGA. Also, Suda's implementation applies convolution using matrix multiplication, achieving good performance on these layers. Suda uses AlexNet and VGG to benchmark its results, finding that it can be significantly better than the CPU time.

Wang et al. [48] created a reconfigurable accelerator for CNNs based on OpenCL. This implementation features OpenCL's extended channel support on FPGAs enabling communication between layers without using onboard memory, allowing for reduced memory usage and faster communication between layers. This work demonstrates high performance and shows the low energy consumption of FPGAs. It also uses AlexNet and VGG to get its results.

Chapter 3

Methodology

The current project is a research on the effect of approximate computing techniques to the improvement of performance and energy consumption on FPGAs. First, the research will be classified based on different perspectives. The classification will be followed by the methods used on creating the CNNs and validating the changes in execution time, energy consumption and accuracy degradation.

3.1 Type of research

The following classification is based on the different perspectives created by [25].

3.1.1 Application perspective

This work is categorized as an applied research. The objective is to apply concepts from approximate computing theory into another field of work or active research such as the study of CNNs on FPGAs. These concepts will be mixed with other topics from computer architecture and hardware generation.

3.1.2 Objectives perspective

From an objectives standpoint, this is an exploratory research due to trying to explore the possibilities of performance and energy gain through the use of approximate hardware generation. The area of approximate computing on FPGAs, specifically for CNN implementations, is currently underexplored. This research will help grow the field and could enable future research or applications on using FPGAs to accelerate the field of Machine Learning.

3.1.3 Mode of enquiry

Finally, this is an unstructured research. The objective is to find how different techniques affect the final measurements and tests in relation to exact kernels. The specific techniques to be applied are not predetermined. As such, there is no expected accuracy value to be achieved. This project tries to push performance to the maximum and energy consumption to the minimum while maintaining an acceptable accuracy, but there is no set number for any of these values.

3.2 Kernel creation

The research is based on modifications done to an exact CNN implementation on an FPGA. The next sections specify the methods used to generate the kernels and apply the approximate modifications.

3.2.1 Exact kernel generation

The exact kernel generation will be based on the CaffeNet implementation. This is an AlexNet implementation with the pooling and normalization layers reversed [9].

The kernels are implemented using OpenCL on a DE1-SoC board, a hardware board with an on-chip processor and a Cyclone V FPGA. This board was provided by the university and its use as a CNN accelerator has been tested previously by other authors [42][48]. Two codes must be generated, the host code to control the execution that will be executed on the embedded chip of the board and the device code that will perform the CNN calculations.

The kernels already contain some approximations due to limitations on the FPGA being used, mainly on using fixed-point values, as it does not contain enough resources to implement a non-fixed-point implementation of the CaffeNet neural network.

3.2.2 Approximate kernel generation

The approximate kernels will be generated based on modifications done on the exact kernels. Different techniques will be used to apply these modifications. Any modification must be tested and different combinations of these modifications must be applied to show the change on accuracy, performance and resource usage.

The modifications must be applied on varying levels of abstraction, measuring the output on different layers up to the output of the full CNN.

3.3 Validation

In this section, the validation to be used on the project is explained.

3.3.1 Accuracy

An algorithm based on the formulas found on [14] is created using the Python. This algorithm calculates accuracy of the approximate modifications against a Caffe [18] based AlexNet implementation, also written in Python.

The training weights to be used for evaluation are the ones found on [9]. The original network was trained using the Image-net Large Scale Visual Recognition Challenge 2012 image set [6]. The measurements will be done for top-1 and top-5 accuracy.

3.3.2 Resource usage

For resource usage, the reports generated by Intel's® FPGA SDK for OpenCL™will be used in order to get the estimated resources used in terms of registers, memory, processing units and logic modules. These reports will be compared against the first implementation on FPGA. While this does not represent a direct measurement of energy, it can give a small look into the energy consumption reduction of applying approximate kernels.

3.3.3 Performance

Performance will be measured by a simple difference between the time at the end of the image classification and at the start. This will be compared to the performance of the Caffe implementation and the exact implementation with no modifications.

Chapter 4

Solution description

In this chapter the processes, techniques and design decisions made to get the final results of the project are shown, as well as an analysis of these results and how they reflect the principles of approximate computing.

4.1 Solution

The CNN design and implementation are described in the following sections. Python is used for the base exact implementation done on a CPU while any approximation is done using OpenCL on the DE1-SoC board.

4.1.1 Base CaffeNet implementation

Caffe is a framework created to easily implement, train and execute NNs using Python or C++. The developers made example models based on popular CNNs, one of which is CaffeNet, a modification of the original AlexNet configuration with the pooling and normalization layers switched.

AlexNet was the winner of the ILSVRC in 2012. Trained using GPUs on 3 million images, it is one of the best examples of how increasing the depth of a NN helps increase its performance and accuracy.

Caffe is used to implement a script that uses the training data from [9] and executes the network over 4000 images downloaded from the ImageNet web page. This generates an output that is used as the baseline for the FPGA implementation and any approximate modification done on the network.

This is a full CPU implementation of CaffeNet using Python. As it is one of the most popular frameworks for CNN training and implementation, the performance gains and accuracy loses can be measured against it. As mentioned before, this project does not reflect the gains over a GPU implementation due to equipment limitations.

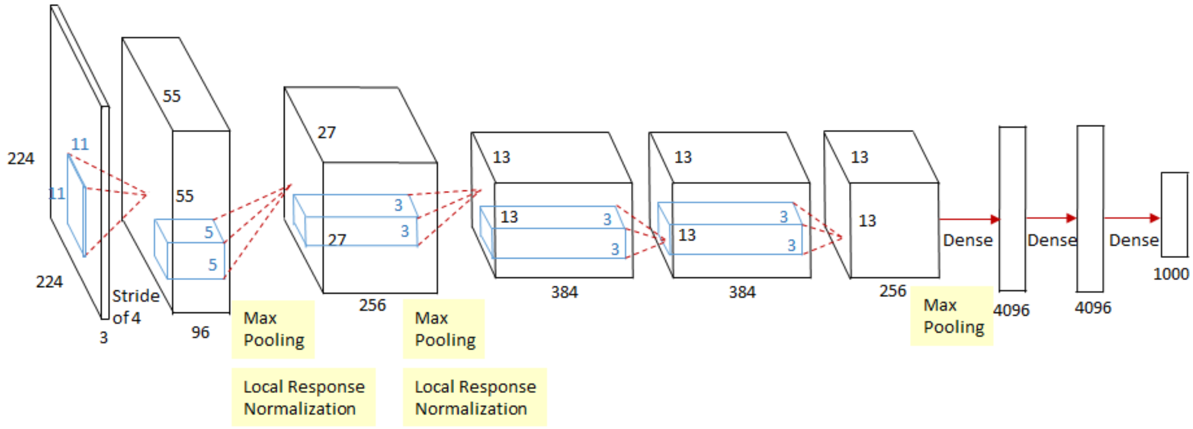

Figure 4.1: Layer configuration of CaffeNet [44]

Figure 4.1 shows the configuration of the CNN. The network has an input matrix of 224x224x3 and contains the following structure of layers and parameters [44]:

1. Convolutional Layer: 96 kernels of size 11x11x3 (stride: 4, pad: 0) with 55x55x96 feature

   3x3 Overlapping Max Pooling (stride: 2) 27x27x96 feature maps

   Local Response Normalization with 27x27x96 feature maps

2. Convolutional Layer: 256 kernels of size 5x5x48 (stride: 1, pad: 2) with 27x27x256 feature maps

   3x3 Overlapping Max Pooling (stride: 2) with 13x13x256 feature maps

   Local Response Normalization with 13x13x256 feature maps

3. Convolutional Layer: 384 kernels of size 3x3x256 (stride: 1, pad: 1) with 13x13x384 feature maps

4. Convolutional Layer: 384 kernels of size 3x3x192 (stride: 1, pad: 1) with 13x13x384 feature maps

5. Convolutional Layer: 256 kernels of size 3x3x192 (stride: 1, pad: 1) with 13x13x256 feature maps

   3x3 Overlapping Max Pooling (stride: 2) with 6x6x256 feature maps

6. Fully Connected (Dense) Layer of 4096 neurons

7. Fully Connected (Dense) Layer of 4096 neurons

8. Fully Connected (Dense) Layer of 1000 neurons (for each of the 1000 classes)

4.1.2 Base FPGA implementation

This section shows the implementation used on the first iteration of the CaffeNet neural network. Each of the layers were implemented as OpenCL kernels that the compiler transforms into a binary file that can be used to reprogram the FPGA directly from the Yocto Linux distribution ran on the ARM Cortex-A9 processor.

Figure 4.2 shows the general configuration of the OpenCL implementation. The host code exchanges information with the kernels programmed into the FPGA. Communication between kernels is done through OpenCL channels, a FIFO buffer that reduces latency by removing the need to read and write to memory the calculations done on every layer.

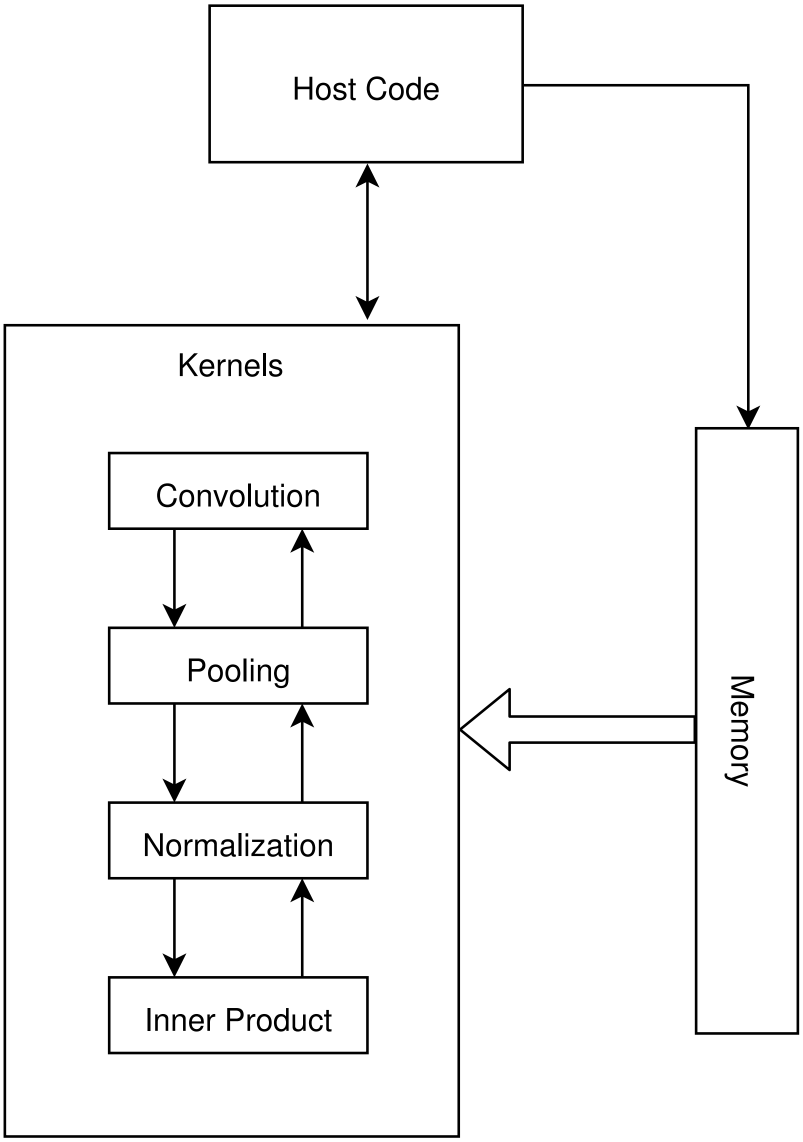

Figure 4.2: Flow diagram of the OpenCL implementation

Host code

The host code is developed using C++, following common Intel's® FPGA SDK for OpenCL™examples. The code for this project is in charge of inserting the training weights file into the global memory for each of the layers of the FPGA, as well as resizing the images to fit the 224x224x3 requirement. The code is capable of processing only images with .jpeg or .jpg extensions and accepts directories in order to process a batch of images.

After finishing evaluating each of the images, the code retrieves the values of the output matrix in the FPGA and prints the final classification on the screen, including up to the first 5 possibilities in order to properly evaluate top-1 and top-5 accuracy.

Compilation of OpenCL binaries

Using a JSON description of the network, the OpenCL offline compiler is run for each of the different kernels that compose the network. These kernels are configured to be compiled and form the complete neural network inside the FPGA after programming. Parameters such as number of neurons, number of layers, input and output sizes and fixed-point precision are preconfigured before running the compilation process.

Convolution layer

As specified on [42], the convolution layer performs a series of 3-dimensional multiply and accumulate (MAC) operations. These operations can be simplified by using flattening and rearranging of the input features. This way, the MAC operations end up being simple multiplications that are accumulated and sent to the output buffer.

The general algorithm is as follows:

1. Compute the address locations of the input and kernel matrixes of the specific work-item identifier.
2. Load the kernel and input 2-D matrixes into local memory. The input is loaded as input[x][y] and the kernel/weights is loaded as kernel[y][x].
3. Apply the MAC operation as follows:

   output += kernel[y][k]*input[x][k]
4. Wait until all work-items finish the work.
5. Save the output to the output buffer using the SDK channels.

These steps must be repeated for every neuron. This process can be optimized using OpenCL pragma directives to allow for Single Instruction Multiple Data (SIMD) processing. The parameters that define the convolution layer can be seen in Table 4.1. Static parameters are defined for every layer, while dynamic ones are defined per layer using a JSON description of the network.

The stride and pad are used to determine the index of the input matrix. Also, each kernel uses a local memory buffer to store the results of each operation. The buffer size is modified by the SIMD factor, as doing multiple operations at once requires multiple sections of memory.

Table 4.1: Parameters used on the convolution layer.

Value	Type	Usage
CHANNEL_N_VECTORIZE	Static	MAC SIMD factor
CHANNEL_N_WIDTH	Static	Neuron SIMD factor
CONV_CH_SIZE_BUF	Static	Local memory buffer
CONV_CH_SIZE_BUF	Static	Local memory buffer
CONV_CH_STRIDE	Dynamic	Stride to use per layer
CONV_CH_PAD	Dynamic	Padding to add to input feature

Pooling

The pooling layers are implemented using overlapping max pooling, following AlexNet's configuration. This means that the kernel downsample the input by selecting the maximum value of each input window matrix and the stride used results in overlapping results.

The algorithm is a simple iteration over each of the values of the window and selecting the maximum. The parameters used in this type of layer are shown in Table 4.2 The SIMD factor determines the size of the local memory to be used.

Table 4.2: Parameters used on the pooling layer.

Value	Type	Usage
CHANNEL_N_WIDTH	Static	Neuron SIMD factor
CONV_SIZE_BUF	Static	Local memory buffer
POOL_CH_EXTEND_MAX	Static	Local memory buffer
POOL_EXTEND	Dynamic	Internal SIMD factor
POOL_STRIDE	Dynamic	Stride to use per layer

Normalization

Local response normalization (LRN) is applied using the formula described in section 2.1.4. The algorithm followed to compute normalization for each kernel is the following:

1. For each *input_feature i*:

2. For each neuron j in feature i:

3. Compute sum_of_squares[j] += input_feature[i+n/2][j]

4. Compute output_feature[i][j] = input_feature[i][j]*pwlf(k+α*sum_of_squares[j])

5. Update sum_of_squares[j] −= input_feature[i−n/2][j]

Where pwlf is an approximate function with precomputed values. This function is hardcoded to avoid doing the calculations in the FPGA and returns the value of the exponential part of the equation in section 2.1.4. Exponential functions are hard to implement hardware-wise, a version with approximate values using a dictionary implemented directly as a ROM section is faster and provides the same functionality.

Table 4.3 shows the parameters used to configure the execution of the normalization layer. Just like the other layers, a SIMD factor is used and can be modified to increase the resource usage of the FPGA being used.

Table 4.3: Parameters used on the normalization layer.

Value	Type	Usage
CHANNEL_N_VECTORIZE	Static	Neuron SIMD factor
NORM_N_SLICES_MAX	Static	Local memory buffer
NORM_SIZE_IN_MAX_0	Static	Local memory buffer
NORM_N_SLICES	Dynamic	Memory access index
NORM_N_SLICES_MIN	Static	Internal SIMD factor

Normalization also allows for parallel SIMD operations. Each neuron can do its work separate from the others and the normalization operation can also be applied in parallel for each input feature slice. The slices determine how much work can be done per neuron on each part of the input matrix.

Fully Connected Layer

The fully connected layer does the same work as the convolution layer, with the difference that it is connected to every output from the layer before it. The parameters used are the same.

Activation Function

AlexNet uses Rectified Linear Unit (ReLU) as the activation function. It is defined as f(x) = max(0,x) and it is applied after every convolution layer. It allows to filter out negative values and provides non-linearity between layers. It also has low computational complexity, as it is easy to implement.

4.1.3 Approximate changes

Any approximate changes done in order to reduce the accuracy but improve the performance and resource usage is described in the following sections.

Arbitrary precision

The fixed-point precision is defined for each layer on the JSON description of the network. Each number contains an integer and fractional part but are stored in the same variable. To determine the bit length and position of the variable, a simple notation is used: *[l,p]*, where l is the length and p is the position of the decimal point from right to left.

The main objective of using lower precision values for different layers is to reduce the resources needed in order to execute the network in the FPGA.

As an example, the decimal value 3.25 can be represented using the notation *[5,3]*. This yields the following binary number:

$$11_2.010_2$$

Fixed-point precision can change from the input of one layer to its output. This is achieved by aligning the precision before applying the specific operation. The limitation is that the input precision of a layer must match the output precision of the layer before it.

While it is possible to manually implement fixed-point precision integers, Intel's® FPGA SDK for OpenCL™ allows for implementation of compilable arbitrary precision integers. Listing 4.1 shows a simple multiplication of two 7-bit integers, the result being saved into a 14-bit integer. The compiler automatically assigns the desired number of bits for each variable.

```
// Intel FPGA arbitrary precision integers
uint7_t a2, b2;
uint14_t c2;
c2 = a2 * (uint14_t)b2;
```

Listing 4.1: Arbitrary precision integers

Approximate operations

A CNN spends most of its computational resources on operations within the layers. These operations can be changed to an approximate version of themselves. Some of the approximations used are:

- Approximate MAC: using Adams's [4] implementation of an energy-efficient and approximate MAC unit brings gains in resource usage. The execution time changes are unknown and will be experimented on, as Adams did not measure this specific parameter.

- Faster pooling operation: pooling operation iterates over the input matrix looking for the maximum value on each window. A faster operation can ignore specific values or skip loop iterations.

OpenCL compiler optimizations

OpenCL offers optimizations that allow increases in performance at the cost of accuracy. These optimizations are set on compile time and are specific to OpenCL, not Intel's implementation for the FPGA. The main approximations used are:

- -fp-relaxed: relaxes the order of arithmetic operations. This is only applied to operations on which the order does not affect the end result.
- -fpc: Removes intermediary roundings and conversions when possible. Changes the rounding mode to round towards zero for multiplies and adds.
- -cl-mad-enable: changes MAC operations to a mad. This is an approximate version of MAC with reduced accuracy.

There is no control on where these optimizations are applied, as it is set for the whole compilation process.

Memoization

Memoization is applied mainly on convolution layers with small stride values. This is because the objective is to approximate the overlapping calculations for each neuron. This is achieved by skipping every other loop call in the main convolution functionality and using the latest result as the output for every skipped iteration.

CaffeNet definition changes

Some approximations that can be applied to the CNN are changing the configuration of the network in order to reduce the amount of computation done or approximate its results.

- Reduction of filter size: AlexNet uses 11x11 filters on the first layer. This size can be modified to a lower number in order to reduce the number of operations and the resource usage of the network. Another possibility is to change to specific constant values some of the iterations in the calculations.

- Increasing stride value: similarly to memoization, a bigger stride results in big gains on computation speed. This change should not affect the resource usage.
- Removal of layers: AlexNet achieves its low error-rate by increasing the number of layers in its definition. But a higher error-rate with gains in performance and resource usage is possible by reducing the depth of the network. This change can only be applied to the later layers, as a propagation from the initial layers would require retraining the network, which takes longer than the allocated time for the project.

4.1.4 Validation and measurements

The methods used to measure and validate the accuracy gains and losses, as well as the improvement or degradation on performance and resource usage are described in the following sections.

Accuracy

The original validation set used by AlexNet consists of 120000 images from 1000 classes. Due to limitation in time, the set used in the project consists of only 4000 images. Accuracy is measured against top-1 and top-5 classification. Each image has a specific class assigned to it and the validation process returns a matrix with possibilities for each class.

A Python 3.5 script is used in order to evaluate the validation results of each compiled CNN against the expected results. The process is a counting of *hits* (good results) for the first class (top-1) and counting of *hits* for the first 5 classes (top-5).

The actual error-rate is calculated as follows:

$$Top1_{error} = \frac{\#images - \#top1_{hits}}{\#images}$$

$$Top5_{error} = \frac{\#images - \#top5_{hits}}{\#images}$$

Performance

After the input image is resized and transformed into the input matrix, the start time is measured and when the process is finished, the end time is measured again. The difference between both times is used as the execution time of the network.

Resource usage

Intel's® FPGA SDK for OpenCL™ provides HTML reports of the estimated resource usage during and after compilation time. The report contains information on the usage of the following resources:

- Adaptive look-up tables (ALUTs): represent the percentage of available logic resources used in compiled designed. Each adaptive logic module (ALM) can be used to implement two ALUTs.
- Registers (FFs): represent the rest of the logic implementation within the FPGA.
- Digital signal processing units (DSPs): specialized DSP units used by the FPGA on complex operations.
- Random access memory (RAMs): local memory available within the FPGA. No external RAM is being used for this project, so this percentage represents the actual memory usage of the whole network.

All of these resources should go down when implementing approximate versions of the CNN.

4.2 Results and analysis

The following sections show all relevant results from applying approximate modifications to the base CNN implementation. These results are compared to the CPU implementation in regard to accuracy loss and performance. Resource usage is evaluated against the first version of the working FPGA implementation.

Table 4.4 shows the different measurements for every major modification. The original AlexNet implementation data is used to compare against the results obtained in the project. Discrepancies between the original AlexNet implementation and the Python implementation are due to using different validation data sets, as the current project uses a newer data set obtained from downloading test images from image-net.org.

Table 4.4: Execution time, error rate and resource usage for different types of kernel and approximate modifications.

Implementation	Top-1 Error (%)	Top-5 Error (%)	Execution time (ms)	ALUTs (%)	FFs (%)	RAMs (%)
AlexNet	40.7	18.2	-	-	-	-
Python CaffeNet	25.53	9.37	451	-	-	-
Base FPGA (8-bit precision)	62.79	35.39	205	92	53	11
Fixed-point 7-bit (all layers)	94.94	79.88	203	81	45	9
Approximate pooling	70.34	46.58	198	92	53	11
All OpenCL optimizations	62.79	35.39	-	107	57	11
Memoization (all pooling layers)	71.45	47.00	196	92	53	12

The results from Table 4.4 are a summary of the main modifications done on the CNN implementation. They show implementations done on the extreme cases for every type of modification (applying changes to all/none of the layers). As such, the accuracy varies a lot from the base implementation. As for performance, the initial performance of 205 ms on the 8-bit implementation is used as the base for any comparison to other modifications.

Fixed-point

Figure 4.3 and Figure 4.4 show the accuracy degradation and performance gain after modifying different layers and with two fixed-point configurations, 8-bit and 7-bit. 8-bit is used on all layers and represents the original version of the FPGA implementation. This is due to limitations on the DE1-SoC FPGA.

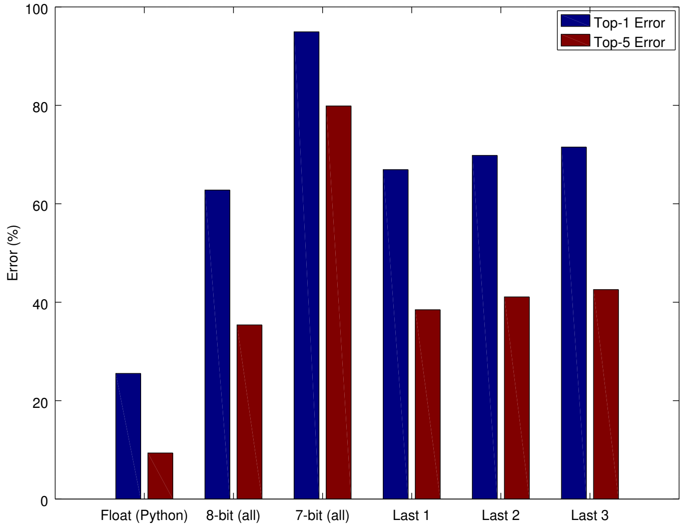

Figure 4.3: Error rate results on top-1 and top-5 accuracy for various fixed-point configurations

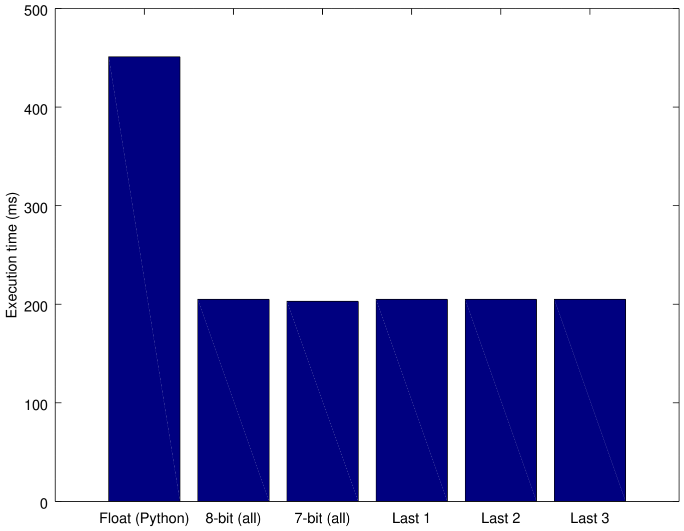

Figure 4.4: Execution time results for various fixed-point configurations

Making changes to all layers of the CNN greatly degrades the accuracy of the network. This is because of propagation of the error from the first layers to the later ones. Because this error is also being approximated, the final result shows a big reduction on accuracy. For 7-bit precision on all layers, the loss in accuracy is unacceptable as the Top-1 error rate falls way above 50%. But, the loss in accuracy for applying this precision on the later layers reflects better results that get close to the 8-bit implementation.

Regarding performance for fixed-point modifications, the changes do not reflect big reductions in execution time. The main reason is that the number of operations stay the same. The reduction in resource usage, shown in Table 4.5, can be utilized to increase parallelism of the neuron operations and, in doing so, reduce execution time.

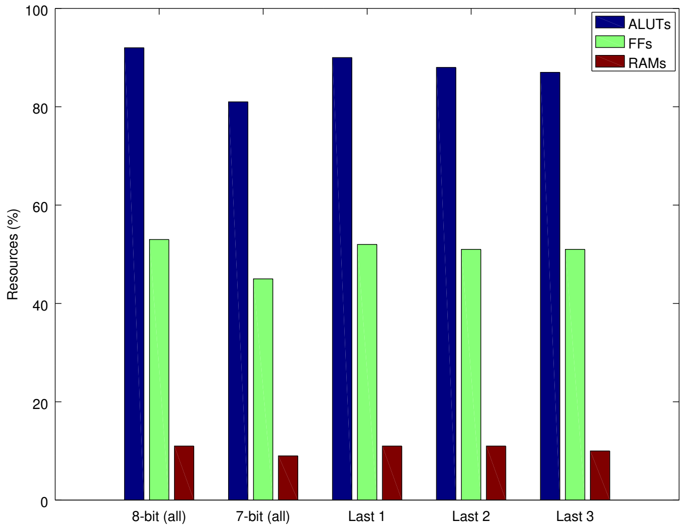

Figure 4.5: Resource usage results for various fixed-point configurations

Approximate operations

Approximate MAC operations on convolutional/fully-connected layers was tested in combination with approximate pooling operation. Figure 4.6 shows the accuracy loss after applying one of each operation and their combination. The error rate is greatly increased for the MAC operation. This is because MAC is the most common operation in the CNN, while pooling is only done on three of the layers. These operations are applied to the all layers of the same type.

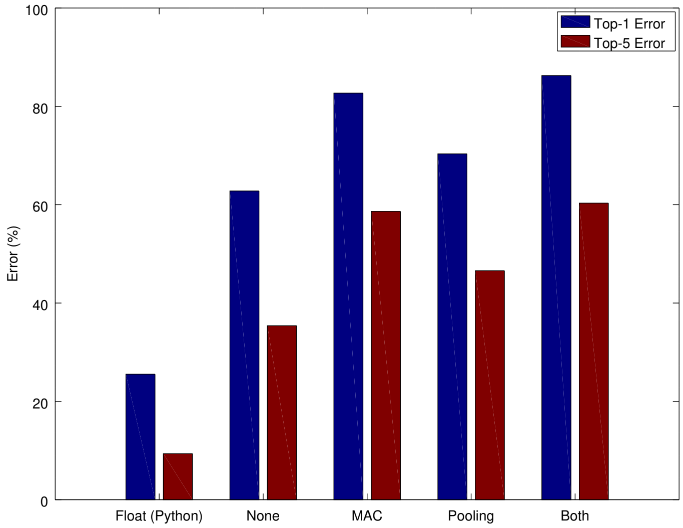

Figure 4.6: Error rate results on top-1 and top-5 accuracy for various approximate operation implementations

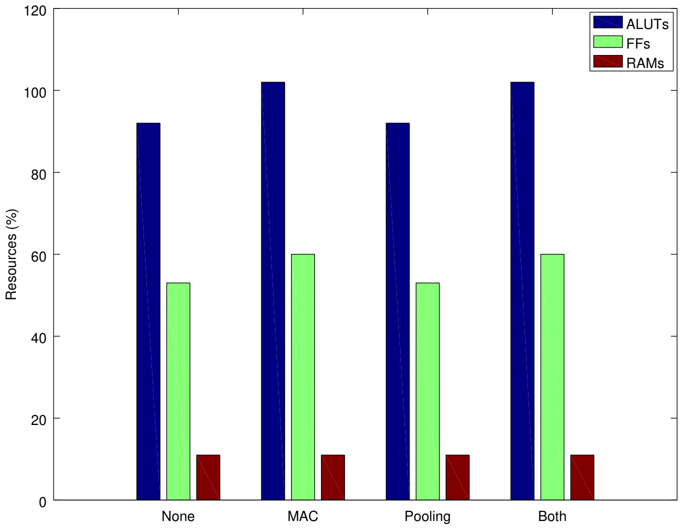

Figure 4.7: Resource usage for various approximate operation implementations

On figure 4.7, the resource usage for each combination is seen. The resource usage for the MAC operation exceeds the available resources on the DE1-SoC board because of the way they had to be implemented on OpenCL, using bit-level shifting to apply approximation. As OpenCL does not allow for bit manipulation within a variable, the resulting implementation requires more resources than the regular multiplication. Pooling does not show any changes in resource usage, as the memory used for iteration skipping on the inner loops is minimal.

Performance on approximate operations was only measured for the approximate pooling operation. Because the resource usage is above the limits of the DE1-SoC board, the performance could not be measured. Accuracy was measured using an emulation of the implementation, which allows resource usages above 100%. The execution time for the approximate pooling operation is 198ms, almost 4% faster than the base implementation. It could be possible to reduce the parallelism of the pooling operation in order to reduce resource usage.

OpenCL optimizations

OpenCL optimizations did not yield any relevant results in terms of performance or resource utilization. Using the *-fpc* flag resulted in higher than 100% resource usage but no changes in accuracy. The most likely reason for no significant differences is because most of the implementation already contains most of these optimizations within the actual code. The OpenCL approximation flags work better on native float variables and operations with these values.

Memoization

Memoization on window shifting was applied only to layers with big filter matrix size and low stride values (less or equal than 2). As such, it was only applied to the three pooling layers. It is applied to reduce the number of operations in each layer in half, so it is only storing values once every two iterations of the main inner loop of the layer. Table 4.5 shows the results of applying memoization to the first, second and third pooling layers and combination of these. Resource usage increased on memory usage, but the performance is improved because of it, while accuracy is reduced slightly for every change.

Table 4.5: Execution time, error rate and resource usage for memoization on different layer combinations.

Layers applied	Top-1 Error (%)	Top-5 Error (%)	Execution time (ms)	ALUTs (%)	FFs (%)	RAMs (%)
None (Python)	25.53	9.37	451	-	-	-
None (FPGA)	62.79	35.39	205	92	53	11
1st	67.56	42.51	200	92	53	12
2nd	64.91	38.22	202	92	53	11
3rd	63.83	36.88	204	92	53	11
2nd + 3rd	71.45	47.00	201	92	53	12
1st + 2nd + 3rd	71.45	47.00	196	92	53	12

Due to implementing this technique on the pooling layers that provide downsampling, the accuracy is not affected greatly. Pooling is a good choice for approximation as it provides little changes to accuracy, but the gains in area or performance are not that significant because of the little amount of operations in comparison to convolution.

CaffeNet definition changes

Most of the changes applied in this section should require a retraining of the network. This is because of the nature of CNNs, as any change in the layer configuration or parameters requires new weights data for use in future executions. As such, the results are highly subjective to changes if retraining is done after applying them. Most of the relevant results are related to resource usage and performance. Accuracy was not measured for these changes due to limitations on network retraining time.

Filter size was reduced for the convolutional layers. The results can be seen on Table 4.6, with different filter size on different layers. The performance and memory gains can be significant for smaller filter matrixes. Reducing filter size is a change that greatly changes the configuration of the network but can be applied with retraining.

Table 4.6: Execution time and resource usage for different configurations of filter size.

Layer	Filter size	Execution time (ms)	ALUTs (%)	FFs (%)	RAMs (%)
Base FPGA	-	205	92	53	11
Convolution 1	10x10	198	89	50	10
	9x9	194	87	49	9
Convolution 2	4x4	200	89	50	10
	3x3	198	88	49	10
Convolution 3	2x2	201	90	51	10

Changes in stride value were done on the convolution and pooling layers. Table 4.7 shows the changes in performance and resource gain for this change on different combinations of layers, increasing the value of the stride 2 per layer. Resource usage stays the same due to not changing the resources used per operation, but the performance gain is significant.

Table 4.7: Execution time and resource usage for different stride values.

Layer	Execution time (ms)	ALUTs (%)	FFs (%)	RAMs (%)
Base FPGA	205	92	53	11
Convolution + Pooling 1	199	92	53	11
Convolution + Pooling 2	198	92	53	11
Convolution + Pooling 3	199	92	53	11
Convolution + Pooling 1,2	194	92	53	11
Convolution + Pooling 1,2,3	189	92	53	11

Finally, removal of layers was only done as experimentation. This is the least recommended change as it means a completely different configuration as Alex/CaffeNet. The only layers that can be easily removed were the fifth convolution layer and second fully-connected layer, as the outputs from the layers before and inputs from the layers after match without any further changes. Table 4.8 shows the results in terms of execution time and resource usage. There is a big gain on both, but retraining is completely required for accuracy measurements.

Table 4.8: Execution time, error rate and resource usage on removing layers from the configuration.

Layers removed	Execution time (ms)	ALUTs (%)	FFs (%)	RAMs (%)
Base FPGA	205	92	53	11
Convolution 5	190	84	48	10
Fully-connected 2	191	86	50	10
Convolution + fully-connected	176	78	45	9

Chapter 5

Conclusions and recommendations

In the next section, the main discoveries of the project are presented. Also, some recommendations are given on what future projects can focus on and how the current work can be expanded.

5.1 Conclusions

OpenCL can be used to describe approximate operations on neural networks through the use of compiler flags and approximate algorithm implementation. On CNN implementations, OpenCL allows for high-level implementation of hardware definition with performance gains and resource usage reductions.

Approximate computing techniques can be used on CNNs to show their error-tolerance property. Iteration skipping and memoization on pooling layers had the best results in terms of performance with a minimal accuracy loss.

One of the biggest advantages of using FPGA is being able to properly control the precision of variables being used. Because of this, changing the precision on arithmetic operations and input values represents significant reductions on resource usage on FPGA implementations of CNNs.

The biggest losses in accuracy are observed when making changes to the initial layers of the CNN. Approximate computing works better on the final layers of the CNN where the propagation of data between layers has the minimum effect on accuracy.

OpenCL optimizations did not yield relevant results, as most of the implementations are irrelevant to CNNs or were manually implemented through low-level definitions.

The best accuracy-performance ratio found was an increase of 5.66% on the top-1 error with a performance gain of 4.39%. This performance can be boosted by increasing parallelism and maximizing the resource usage of the FPGA. A combination of techniques could be used to achieve even lower execution times with different levels of accuracy.

5.2 Recommendations

The utilization of approximate computing techniques on FPGA-implemented CNNs can be further explored through the combination of the different types of techniques shown in this work. The current project's scope can be expanded to show the maximum performance and resource usage gain from approximate techniques.

A way to improve the results of the project would be to increase the number of modifications on each of the layers and making a bigger number of combinations between these modifications. A future project could use the current one as a baseline to improve the current results.

OpenCL represents a good entry into the hardware definition area. The project, however, could be expanded upon through the use of low-level hardware definition languages such as Verilog to produce finely-tuned approximate solutions.

Changing the base CNN configuration is not recommended as it requires retraining of the network. As CNNs have a large number of layers, the training process takes a long time (days or weeks). Depending on the time constraints of the project, this may not be feasible.

Using up-to-date CNN configurations with better accuracy and performance than AlexNet/-CaffeNet could significantly improve the results of the project. For future projects, it is recommended to look at newer implementations of CNNs on FPGAs, as it is a growing knowledge area.

In some cases, there is a trade-off between accuracy and memory usage that can be explored in order to reduce general resource usage and energy consumption. The current project uses less than 20% of the available memory on the FPGA, which could be increased to have big gains on performance through the use of techniques such as memoization.

The current project targets a relatively low-performance FPGA. The use of a more powerful platform could lead to better performance in comparison to a CPU/GPU CNN implementation.

Bibliography

[1] Martın Abadi et al. “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”. In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. “Gzip on a chip: High performance lossless data compression on fpgas using opencl”. In: *Proceedings of the international workshop on openCL 2013 & 2014*. ACM. 2014, p. 4.

[3] Kamel Abdelouahab et al. “Accelerating cnn inference on fpgas: A survey”. In: *arXiv preprint arXiv:1806.01683* (2018).

[4] Elizabeth Adams, Suganthi Venkatachalam, and Seok-Bum Ko. “Energy-Efficient Approximate MAC Unit”. In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2019, pp. 1–4.

[5] Ankur Agrawal et al. “Approximate computing: Challenges and opportunities”. In: *2016 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE. 2016, pp. 1–8.

[6] Alex Berg, Jia Deng, and L Fei-Fei. *Large scale visual recognition challenge 2010*. 2010.

[7] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[8] Vinay K Chippa et al. “Analysis and characterization of inherent application resilience for approximate computing”. In: *Proceedings of the 50th Annual Design Automation Conference*. ACM. 2013, p. 113.

[9] Jeff Donahue. *BVLC Reference CaffeNet*. 2012.

[10] Manek Dubash. “Moore’s Law is dead, says Gordon Moore”. In: *Techworld. com* 13 (2005).

[11] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological cybernetics* 36.4 (1980), pp. 193–202.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[13] Google. *Baden-Württemberg*. URL: https://www.google.com/maps/place/Baden-W%C3%BCrttemberg/%7B@%7D48.6618471,9.0036649,8z/data=!3m1!4b1!4m5!3m4!1s0x47911b62826406df:0xc68c48bf0d244860!8m2!3d48.6616037!4d9.3501336?hl=en.

[14] Google. *Classification: Precision and Recall*. URL: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall.

[15] Kaiyuan Guo et al. "A survey of fpga-based neural network accelerator". In: *arXiv preprint arXiv:1712.08934* (2017).

[16] Jim Held, Jerry Bautista, and Sean Koehl. "White paper from a few cores to many: A tera-scale computing research review". In: (2006).

[17] David H Hubel and Torsten N Wiesel. "Receptive fields and functional architecture of monkey striate cortex". In: *The Journal of physiology* 195.1 (1968), pp. 215–243.

[18] Yangqing Jia et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[19] Andrej Karpathy et al. "Cs231n convolutional neural networks for visual recognition". In: *Neural networks* 1 (2016).

[20] Yongtae Kim, Yong Zhang, and Peng Li. "An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems". In: *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press. 2013, pp. 130–137.

[21] KIT. *KIT logo*. URL: http://www.kit.edu/img/intern/10jahre_de.svg.

[22] KIT. *Organization and Governance*. URL: https://www.kit.edu/kit/english/organization.php.

[23] kjunix. *Map of Baden-Württemberg*. URL: https://en.wikivoyage.org/wiki/Baden-W%C3%BCrttemberg#/media/File:Baden-Wuerttemberg_travel_map_EN.png.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[25] Ranjit Kumar. *Research methodology: A step-by-step guide for beginners*. Sage Publications Limited, 2019.

[26] Suhas Kumar. "Fundamental limits to Moore's Law". In: *arXiv preprint arXiv:1511.05956* (2015).

[27] Suhas Kumar. "Fundamental limits to Moore's law". In: (Nov. 2015). DOI: arXiv:1511.05956.

[28] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[29] Chaofan Li et al. “Joint precision optimization and high level synthesis for approximate computing”. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2015, pp. 1–6.

[30] Jin Miao, Andreas Gerstlauer, and Michael Orshansky. “Approximate logic synthesis under general error magnitude and frequency constraints”. In: *Proceedings of the international conference on computer-aided design*. IEEE Press. 2013, pp. 779–786.

[31] Bert Moons et al. “Energy-efficient convnets through approximate computing”. In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. 2016, pp. 1–8.

[32] Thierry Moreau et al. “SNNAP: Approximate computing on programmable socs via neural acceleration”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 603–614.

[33] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA: 2015.

[34] Ashish Ranjan et al. “ASLAN: Synthesis of approximate sequential circuits”. In: *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association. 2014, p. 364.

[35] Antonio Roldao-Lopes et al. “More flops or more precision? accuracy parameterizable linear equation solvers for model predictive control”. In: *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE. 2009, pp. 209–216.

[36] Adrian Sampson et al. “Accept: A programmer-guided compiler framework for practical approximate computing”. In: *University of Washington Technical Report UW-CSE-15-01* 1.2 (2015).

[37] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117.

[38] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117.

[39] Doochul Shin and Sandeep K Gupta. “Approximate logic synthesis for error tolerant applications”. In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE. 2010, pp. 957–960.

[40] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[41] Sharad Sinha and Wei Zhang. “Low-power FPGA design using memoization-based approximate computing”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.8 (2016), pp. 2665–2678.

[42] Naveen Suda et al. “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2016, pp. 16–25.

[43] Darshan D Thaker et al. “Characterization of error-tolerant applications when protecting control data”. In: *2006 IEEE International Symposium on Workload Characterization*. IEEE. 2006, pp. 142–149.

[44] Sik-Ho Tsang. *Review: AlexNet, CaffeNet — Winner of ILSVRC 2012 (Image Classification)*. Aug. 2018. URL: https://medium.com/coinmonks/paper-review-of-alexnet-caffenet-winner-in-ilsvrc-2012-image-classification-b93598314160.

[45] Salim Ullah, Sanjeev Sripadraj Murthy, and Akash Kumar. “SMApproxlib: library of FPGA-based approximate multipliers”. In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE. 2018, pp. 1–6.

[46] Swagath Venkataramani et al. “SALSA: systematic logic synthesis of approximate circuits”. In: *DAC Design Automation Conference 2012*. IEEE. 2012, pp. 796–801.

[47] Ganesh Venkatesh et al. “Conservation cores: reducing the energy of mature computations”. In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 1. ACM. 2010, pp. 205–218.

[48] Dong Wang, Ke Xu, and Diankun Jiang. “PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks”. In: *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE. 2017, pp. 279–282.

[49] Naigang Wang et al. “Training deep neural networks with 8-bit floating point numbers”. In: *Advances in neural information processing systems*. 2018, pp. 7675–7684.

[50] Wm A Wulf and Sally A McKee. “Hitting the memory wall: implications of the obvious”. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[51] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. “Approximate computing: A survey”. In: *IEEE Design & Test* 33.1 (2015), pp. 8–22.

[52] Amir Yazdanbakhsh et al. “Axilog: Language support for approximate hardware design”. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium. 2015, pp. 812–817.

[53] Rong Ye et al. “On reconfiguration-oriented approximate adder design and its application”. In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2013, pp. 48–54.

[54] Jialiang Zhang and Jing Li. “Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2017, pp. 25–34.

[55] Yi-Tong Zhou and Rama Chellappa. “Computation of optical flow using a neural network”. In: *IEEE International Conference on Neural Networks*. Vol. 1998. 1988, pp. 71–78.

Appendix A

Implicaciones sociales y éticas del proyecto

El presente proyecto busca aportar al área de investigación en redes neuronales convolucionales. Este tipo de redes neuronales es utilizado principalmente para el reconocimiento y clasificación de imágenes. Esto implica que se puede determinar el contenido de una imagen de la misma forma que el cerebro humano procesa e identifica a la misma imagen.

Con el incremento en la generación de datos por medio de múltiples dispositivos y la creciente tendencia de el "internet de las cosas", existe una creciente responsabilidad de parte de la sociedad de utilizar las redes neuronales convolucionales de forma beneficiosa para esta. Sociedades actuales hacen uso de este tipo de aplicaciones como forma de control de masas o para aumentar ganancias monetarias de poderosas empresas.

Sin embargo, también existen usos positivos para el reconocimiento de imágenes. En aplicaciones médicas, aeroespaciales, tecnológicas, científicas y más, es posible ver los beneficios que las redes neuronales tienen para la sociedad en general. Se espera que este proyecto represente un avance que pueda ayudar a mejorar la vida de las personas.