Hardware Accelerators for Machine Learning Applications. Case Study : Autonomous Vehicles

Mohamed Ayoub Neggaz

PhD Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Université Polytechnique Hauts-de-France
Discipline : **Computer Engineering**

Presented and defended by : Mohamed Ayoub Neggaz.
On Mai 28th, 2020, à Valenciennes.

École doctorale : Sciences Pour l'Ingénieur (ED SPI 072).
Laboratoire : LAMIH UMR CNRS 8201.

HARDWARE ACCELERATORS FOR MACHINE LEARNING APPLICATIONS. CASE STUDY: AUTONOMOUS VEHICLES

Jury president	:	Frederic Pétrot.	Professor, Université Grenoble Alpes.
Reporters	:	Andreas Herkersdorf.	Professor, Technische Universität München.
		Frederic Pétrot.	Professor, Université Grenoble Alpes.
Examiner	:	Catherine Dezan.	Professor, Université de Brest.
Thesis directors	:	Smail Niar.	Professor, UPHF.
		Fadi Kurdahi.	Professor, University of California, Irvine.
Thesis co-supervisor	:	Ihsen Alouani.	Professor, UPHF.

May 2020

Thése de doctorat

Pour l'obtention du grade de Docteur de

l'Université Polytechnique Hauts-de-France
Discipline : **Informatique**

Présentée et soutenue par : Mohamed Ayoub NEGGAZ.
Le 28 Mai 2020, à Valenciennes.

École doctorale : Sciences Pour l'Ingénieur (ED SPI 072).
Laboratoire : LAMIH UMR CNRS 8201.

ACCÉLERATEURS MATÉRIELS POUR L'INTÉLLIGENCE ARTIFICIELLE. ÉTUDE DE CAS: VOITURES AUTONOMES

Président du jury	:	Frederic Pétrot.	Professeur, Université Grenoble Alpes.
Rapporteurs	:	Andreas Herkersdorf.	Professeur, Technische Universität München.
		Frederic Pétrot.	Professeur, Université Grenoble Alpes.
Examinateurs	:	Catherine Dezan.	Professeure, Université de Brest.
Directeurs de thèse	:	Smail Niar.	Professeur, UPHF.
		Fadi Kurdahi.	Professeur, University of California, Irvine.
Co-encadrant de thèse	:	Ihsen Alouani.	Maître de Conférences, UPHF.

Mai 2020

Résumé

Depuis les débuts du défi DARPA, la conception de voitures autonomes suscite un intérêt croissant. Cet intérêt grandit encore plus avec les récents succès des algorithmes d'apprentissage automatique dans les tâches de perception. Bien que la précision de ces algorithmes soit irremplaçable, il est très difficile d'exploiter leur potentiel. Les contraintes en temps réel ainsi que les problèmes de fiabilité alourdissent le fardeau de la conception de plateformes matériels efficaces. Nous discutons les différentes implémentations et techniques d'optimisation de ces plateformes dans ce travail. Nous abordons le problème de ces accélérateurs sous deux perspectives : performances et fiabilité. Nous proposons deux techniques d'accélération qui optimisent l'utilisation du temps et des ressources. Sur le volet fiabilité, nous étudions la résilience des algorithmes de Machine Learning face aux fautes matérielles. Nous proposons un outil qui indique si ces algorithmes sont suffisamment fiables pour être employés dans des systèmes critiques avec de fortes critères sécuritaire ou non. Un accélérateur sur processeur associatif résistif est présenté. Cet accélérateur atteint des performances élevées en raison de sa conception en mémoire qui remédie au goulot d'étranglement de la mémoire présent dans la plupart des algorithmes d'apprentissage automatique. Quant à l'approche de multiplication constante, nous avons ouvert la porte à une nouvelle catégorie d'optimisations en concevant des accélérateurs spécifiques aux instances. Les résultats obtenus surpassent les techniques les plus récentes en termes de temps d'exécution et d'utilisation des ressources. Combinés à l'étude de fiabilité que nous avons menée, les systèmes ou la sécurité est de priorité peuvent profiter de ces accélérateurs sans compromettre cette dernière.

Mots Clés Voitures Autonomes, Intélligence Artificielle, FPGA, GPU, ASIC, CNN, Systèmes Embarqués, Fiabilité.

Abstract

Since the early days of the DARPA challenge, the design of self-driving cars is catching increasing interest. This interest is growing even more with the recent successes of Machine Learning algorithms in perception tasks. While the accuracy of these algorithms is irreplaceable, it is very challenging to harness their potential. Real-time constraints as well as reliability issues heighten the burden of designing efficient platforms.

We discuss the different implementations and optimization techniques in this work. We tackle the problem of these accelerators from two perspectives: performance and reliability. We propose two acceleration techniques that optimize time and resource usage. On reliability, we study the resilience of Machine Learning algorithms. We propose a tool that gives insights whether these algorithms are reliable enough for safety critical systems or not.

The Resistive Associative Processor accelerator achieves high performance due to its in-memory design which remedies the memory bottleneck present in most Machine Learning algorithms. As for the constant multiplication approach, we opened the door for a new category of optimizations by designing instance specific accelerators. The obtained results outperforms the most recent techniques in terms of execution time and resource usage. Combined with the reliability study we conducted, safety-critical systems can profit from these accelerators without compromising its security.

Keywords Autonomous Vehicles, Artificial Intelligence, FPGA, GPU, ASIC, CNN, Embedded Systems, Reliability.

Acknowledgments

To my wife, Amina, my son, Isaac, and my sister, Maria.

Contents

List of Figures

List of Tables

Chapter 1

Motivations and Thesis Objectives

Contents

1.1 Deep Learning for Autonomous Vehicles

The recent years have known a constant increase in road users. This rapid growth has a direct impact on road accidents. The security and economic challenges that stem from this increase are of an immense magnitude. Human error, whether caused by fatigue, alcohol use or unsafe driving, is the major cause of traffic accidents. In a 2016 National Highway Traffic Safety Administration (NTHSA) report [205] in the USA, more than 94% of accidents are caused by human error.

According to the World Health Organization[1], road traffic injuries are the **leading cause of death** for children and young adults aged 5-29 years. The number of death is estimated to 1.35 million per year. To put it into perspective, this number represents more deaths than those caused by alcohol use, drug use, fire, terrorism and natural disasters combined according to a statistic published by Amnesty International[2]. While human life concern is visibly the main impact, road traffic accidents cost most countries 3% of their gross domestic product. The annual social benefits of Autonomous Driving Systems are projected to reach $800 billon by 2050 [255].

On a parallel comparison, fatalities caused by road accidents are the main challenge in transportation. Moreover, when comparing the death tolls in the field of transportation, we clearly see that most fatal accidents occur in road traffic. Figure 1.1 illustrates these tendencies as reported by the National Transportation Safety Board (NTSB) between the years 1990 and 2017[3].

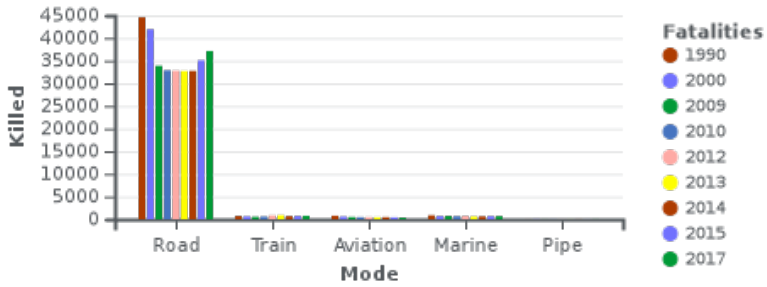


Figure 1.1: Comparison of fatalities caused by accidents in different transportation modes, data reported by the US' NTSB.

The European Commission released a new policy framework for road safety 2021-2030 [44] in May 2018 in order to deal with fatalities caused by road injuries. This new policy has a set of actions including the following:

- To make the vehicles we drive even safer.
- To make the infrastructure we drive on safer.
- To fund and finance investment in transport safety.

These policies have for goal to reduce fatalities by relying on safe autonomous systems. However, these systems have proven to be of high complexity. The unpredictability of road users' behaviour and the constant change of environment due to weather (sunny, rainy or snowy), time (day or night) and location (highway, city road or off-road) are major contributors to this complexity.

The recent advances in Artificial Intelligence (AI) ease the task of autonomous driving by providing highly capable algorithms that accurately assesses a situation. These algorithms rely on complex networks that are able to cope with the environment change and other users' behaviour. Successful deployment of these networks

[1] https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries
[2] https://ourworldindata.org/causes-of-death
[3] https://www.ntsb.gov/investigations/data/Pages/Data_Stats.aspx

have shown impressive reduction in fatalities per 100 million kilometers. The Tesla Autopilot for example have driven more than 3 billion kilometers with less than 10 reported fatalities. This accounts to 0.20 fatalities per 100 million kilometers. This number is estimated to be at 0.73 under traditional human control by the US Department of Transportation's 2016 fatal traffic crash data[4].

The consistency of these numbers is only guaranteed with systems capable of running the most recent AI algorithms. However, the computing requirements are out of reach for today's platforms. In order to process a single 400×400 image and locate driveable area as well as detect obstacles with only 53% accuracy, more then 140 GFlops[5] are needed. Running such algorithm constantly on all cameras deployed in the car and expecting an output at real time is beyond possibility. In addition, other sensors are needed when the environment changes. Cameras can not provide accurate perceptions in dark and obscure environments. The hardware platform needs to process all these inputs to carefully put the car in the right track. With more specialized hardware in the market, finding a single platform that can satisfies these performance constraints is challenging, especially with a limited budget in order to maintain a reasonable system overhead on the final price. Therefore, designing faster and efficient hardware is a key challenge for feasibility of ADS.

1.2 Aim of Thesis

Deep Learning has been established as a leading actor towards the developments of autonomous systems by revitalizing AI. For modern systems, it is inevitable to use these algorithms –or networks[6]– to cope with their increased complexity. While increasing the accuracy is a challenge, being able to deploy these networks should be a priority.

There are two phases when designing a deep neural network, namely, training and inference. The training phase is an offline process that generates a working instance of the network. The inference phase is when the network is ready to process inputs. During the deployment phase, we only need the training phase[7]. In this thesis, we focus on the deployment phase. At this stage, the network takes inputs from the environment using the various sensors deployed on the car and tries to predicts the best decision.

1.2.1 Problem statement

Neural networks are compute hungry algorithms. The inference run on a single camera image requires more than 20 billion Multiply and Accumulate (MAC) operation. At peak performance, a 5 GHz Central Processing Unit (CPU) takes 4 seconds to process the image. By that time, the decision is obsolete and the situation is irrecoverable. In this thesis, we tackle this performance issue by proposing hardware architectures capable of handling this computation load.

This need in compute power is being fed by the increase in sensory demand. With higher levels of automation, the vehicle needs a complete environment perception. As seen in Figure 1.2, at least 28 sensory units are required for a mundane autonomous ride. These sensors continuously stream data to the compute units that need to be alerted at any given time to detect and react in case of danger.

Future autonomous vehicles will need an equivalent of a supercomputer rolling down the highway, generating and transmitting a overwhelming volume of data, up

[4] https://www.nhtsa.gov/press-releases/usdot-releases-2016-fatal-traffic-crash-data

[5] These numbers are obtained using a trained instance of Mask RCNN with 32-bit floating point weights and activations.

[6] Deep learning algorithms have similar structures as networks of artificial neurons. They are referred to as neural networks instead of the general appellation of algorithms, which is vaster.

[7] Some networks are continuously trained, however we do not consider these algorithms in this study.

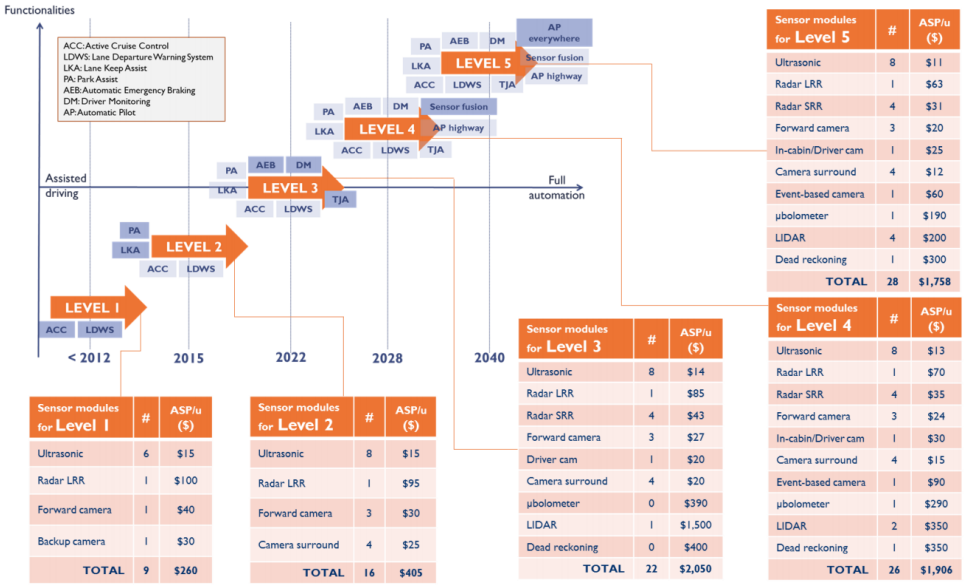


Figure 1.2: Sensor modules for each automation level from [251].

to 4 terabytes [8] per day per car. Deploying a massive hardware platform to handle these tasks would have dramatic impact on the cost, weight and energy consumption of the vehicle. For autonomous vehicles to be considered a safety replacement, efficient hardware need to be designed, aimed specifically at solving these challenges.

An other issue that is under-explored by researchers in machine learning is reliability. In this thesis we will also tackle the problem of whether these algorithms should be trusted for safety-critical systems such as autonomous driving. ADS should adhere to strict safety standards such as ISO 26262:2018 [100] and ISO/PAS 21448:2019 [101] specially addressing the safety of Advanced Driver Assistance Systems (ADAS) functions in road vehicles. The reliance on machine learning puts ADSs in a gray area when it comes to these standards due to their lack of traceability and unpredictability [120].

1.2.2 Constraints and Challenges

As far as inference is concerned, many platforms exist to satisfy these constraints. However, for a commercial system, efficiency is always the next challenge after feasibility. Designing efficient hardware for self-driven cars is crucial to their functioning. Many major companies addressed this problem for their autonomous vehicle projects. For instance, Tesla's Hardware 2 uses the NVIDIA DRIVE PX 2 AI computing platform. The platform costs around 15,000 € per unit. This price is compared to that of the most sold car in 2019, the Toyota Corolla. On the other hand, Google have designed a novel architecture, the Tensor Processing Unit (TPU), that is dedicated to machine learning algorithms. The TPU is deployed on the cloud and could be rented for research. In the academic community, Field Programmable Gate Array (FPGA) accelerators are widespread. However, these accelerators trade the accuracy of the system for performance. This trade-off is not possible for a safety-critical system such as autonomous vehicles.

As for reliability, this problem seems to be ignored. Researchers focus on improving the accuracy of the network. This accuracy is then traded for performance with no concern on reliability. This is partly since machine learning algorithms are not yet considered for safety-critical systems and are always supervised by a human expert.

[8] https://datacenterfrontier.com/rolling-zettabytes-quantifying-the-data-impact-of-connected-cars/

This trend is changing. The accelerator for machine learning needs to be resilient to faults if deployed in aggressive environments.

1.2.3 Thesis Objectives

In this thesis, we tackle the problem of performance and reliability for machine learning accelerators deployed in autonomous vehicles. In the performance study, the usual tradeoff between accuracy and performance is not tolerated. As for the reliability study, we aim to locate the vulnerable components of the software, i.e. the network, and the hardware.

1.3 Main Contributions

The thesis focus on designing hardware for autonomous vehicles. The problem was tailored down to designing machine learning accelerators. In this thesis, we propose two architectures of hardware accelerators for machine learning.

The first architecture can be seen in Chapter 5. In this chapter, we used a memristor-based implementation of an associative processor, the Resistive Associative Processor (ReAP), to solve the performance problem of machine learning for autonomous systems like self-driving cars. The motivations behind this choice are the von Neumann bottleneck and it's implications on the performance of the system. In our ReAP, all computations are done inside the memory which solves this problem. The accelerator compares very well to a Xilinx reference design with less than 2 ms for a 1024×1024 matrix multiplication, two orders of magnitude less than the Xilinx design. In terms of general compute efficiency, measured as GOPs per Watt per transistor, comparisons shows a staggering difference with a major gap between our design and the Xilinx design. This was mainly due to the efficient usage of resources for storage and compute at the same time. The results of this work has been published in [163],

Our second contribution is on FPGA and is detailed in Chapter 6. We exploit the fact that weights are constant after training. Therefore, we designed an accelerators dedicated to a trained instance of a network. In any given design, the multipliers require a weight and an input. Since weights are constant, we design dedicated multipliers that only takes one operand, the input. This results in a inflexible design since these multipliers can no longer be reused. However, this flexibility is traded for performance and resource utilization which is a novel contribution compared to other state of the art accelerators where the main tradeoff happens between accuracy and performance. Moreover, since weights are embedded in multipliers, no storage is needed. Simplifying storage have huge implication on the architecture since no time is needed to load weights from the input and the memory would be free to store more inputs. Results of this work has been partially published in [153].

As stated earlier, the second axis of this thesis is reliability. While machine learning has been extensively studied in the past few years from an acceleration and quality point of view, research about how reliable these network are against faults and attacks are meagre. The results of this work shows that reliability is an important issue for safety of machine learning-based systems. The accuracy with which the system has been deployed can be substantially compromised. This was a two step contribution given in details in Chapter 7. First, we propose a study to test if there is a problem when the hardware accelerator is exposed to errors. In this first part, we discovered that the combinatorial part of the circuit is resilient however, memory errors can prove to be dramatic. This first contribution was published in [165]. In the second study, we focus on the nature of errors. We compare the different representations and how they compare to each other. The results of this work is published in [161]. In parallel, a tool was made publicly available[9] to test the resilience

[9] https://github.com/cypox/CNN-Fault-Injector

of a CNN and to isolate the vulnerable layers that needs.

From a system point of view, we designed a streaming platform that uses CPUs and FPGAs to conduct data from a sensory unit, for instance, a camera, to the accelerator. The design undergoes a study on the input to choose the optimal size that allows components to communicate efficiently with minimal loss in accuracy and performance. The results of this work were published in [162]. An other study on sensor fusion was conducted to show a way of merging the camera and the LiDAR inputs, two totally different representations cooperating to perform road extraction and obstacle detection. This work was partially published in [168]. We also propose a high level framework in [81] that incorporate the cloud and the fog in the detection process. Communication with these components is key to achieving full safe driving.

1.4 Outline

This thesis is structured as follows. First we give our reasons on why we considered machine learning algorithms as a solution for the driving problem. This is presented int Chapter 2 along with the required knowledge on field to understand later chapters. We conclude the chapter with a review on recent algorithms, their architectures and performances. An in-depth study on ADS is given in Chapter 3. We present the challenges of autonomy in self-driving cars from a hardware point of view. In Chapter 4 we give our state-of-the-art study on hardware accelerators for machine learning that may apply to the autonomous driving scenario. Two novel accelerator designs were presented in Chapters 5 and Chapter 6. Our reliability study is shown in Chapter 7. We conclude the thesis with a summary and some research perspectives in Chapter 8. The dependencies between chapters are shown in the flow graph in Figures 1.3.

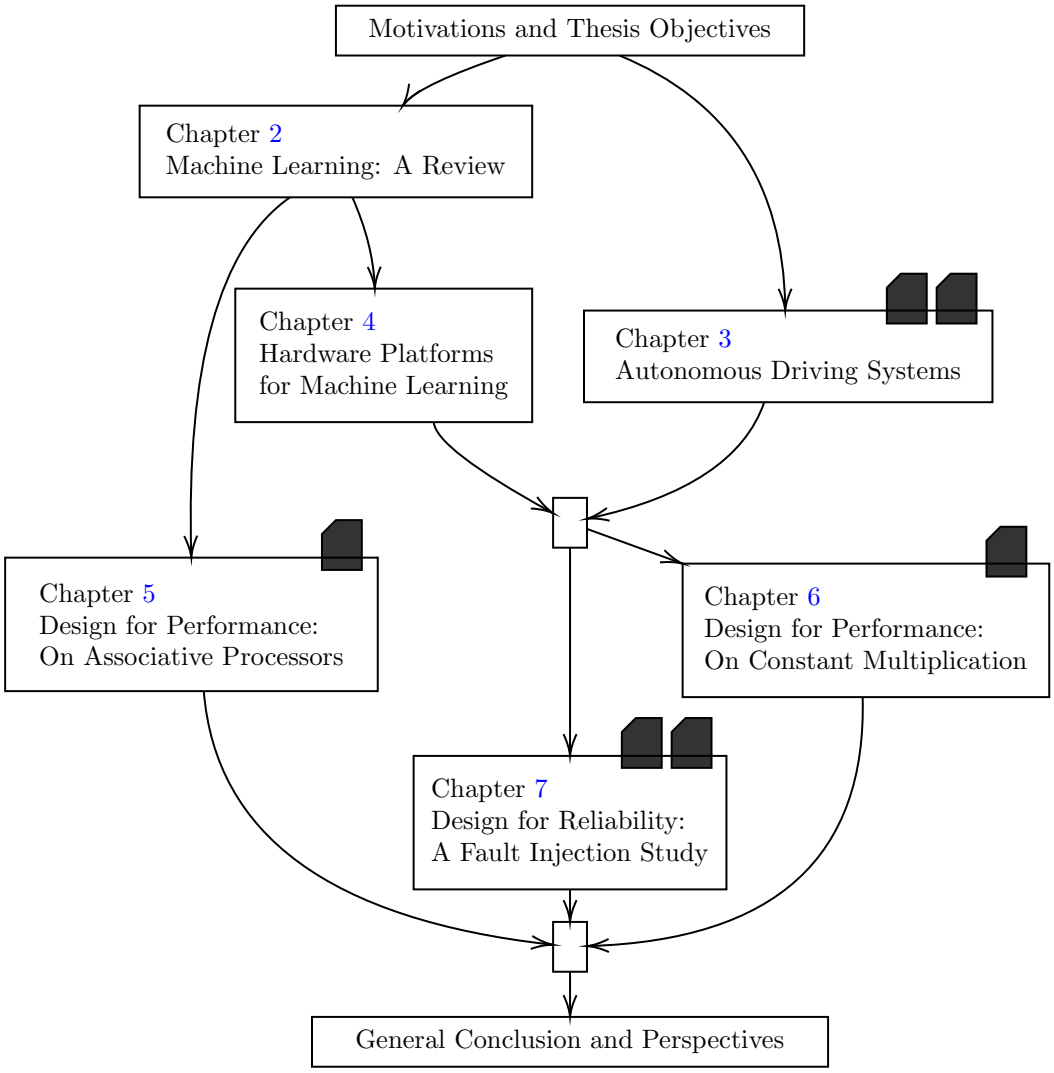


Figure 1.3: The reading dependencies between chapters. Chapters with the black document icon in the top right corner contain details of at least one peer-reviewed publication.

Chapter 2

Machine Learning: A Review

Contents

While Moore's law is in decline for single-core processors, parallel computing is correcting the slope for the past decade in order to maintain the performance increase each year as can be seen in Figure 2.1. With this law being true for half a century, processing power became abundant. Today's computers are able to perform incredible feats.

Computer vision is one of the fields that harnessed this power. Ideas that requires large data operations became possible. Limited by the available compute power back then, these algorithms remain dormant for a long time. The, arguably biggest, milestone achieved recently was witnessed in ILSVRC 2015 when AlexNet, a CNN, raised the bar by beating other algorithms and even human accuracy at guessing the class of an image from 1000 possible classes.

In this chapter we walk through these algorithms. We first explain the motivations that lies behind the choice of machine learning over other approaches. We also give an insight on how most famous machine learning algorithms operates. Later on, we present a profiling study to analyze the characteristics and requirements of these algorithms.

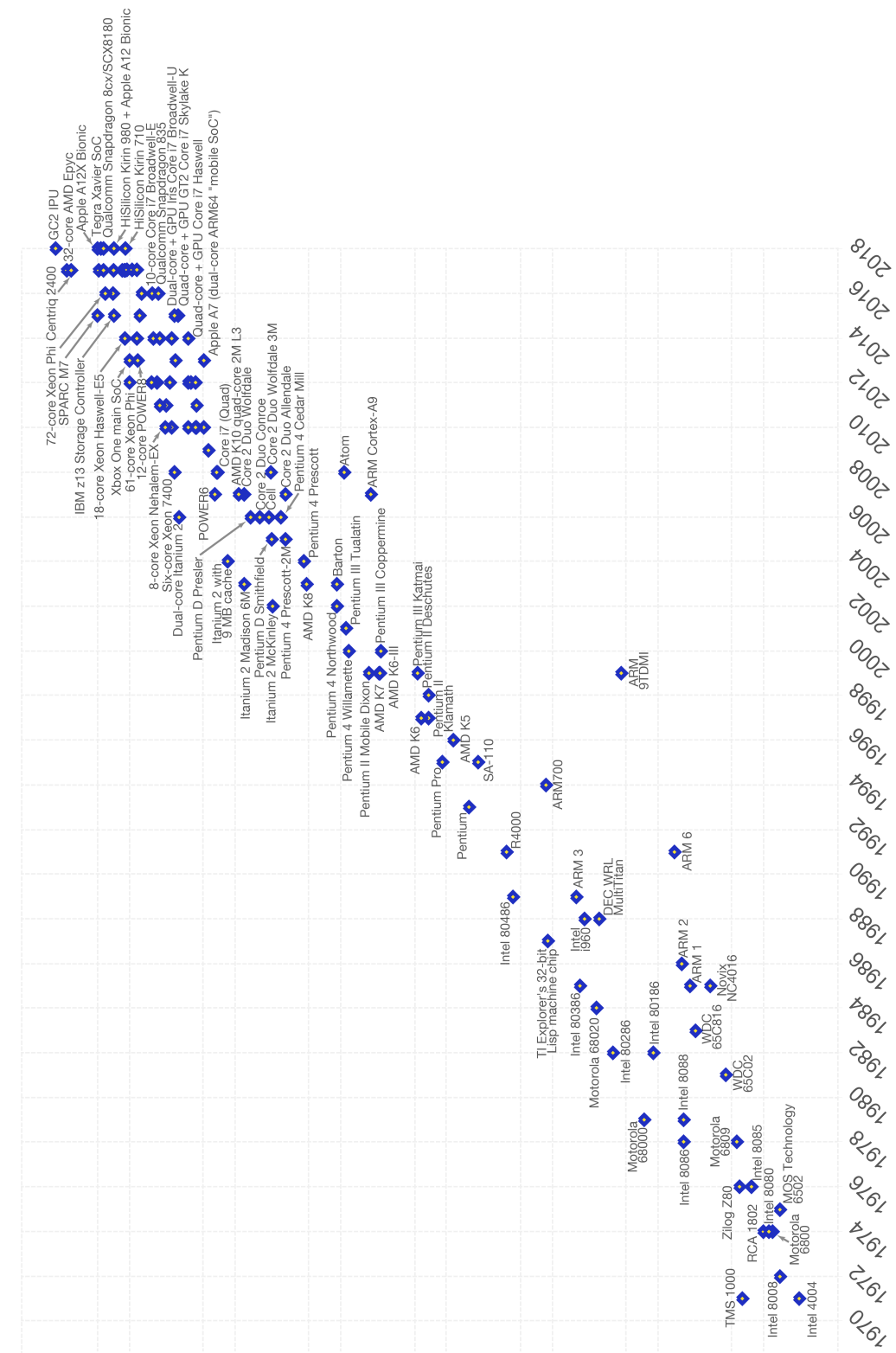


Figure 2.1: Moore's Law describing how the performance of compute platforms evolve with time.

2.1 Artificial Intelligence and Machine Learning

Machine learning is a sub-category of AI. AI encompass any algorithm that enables computers to mimic human behavior as can be seen in Figure 2.2. Examples of these algorithms are rule-based or expert systems usually used for medical diagnosis where all the outputs of a problem (in this cases diseases) are listed with all the possible inputs (symptoms in the case of medical diagnosis). While this basic is useful in some fields, it does not scale with the problem size where the human expert needs to pass a large amount of information to the computer. The ability to learn without being explicitly programmed was the follow-up of AI algorithms and is what we refer to as machine learning.

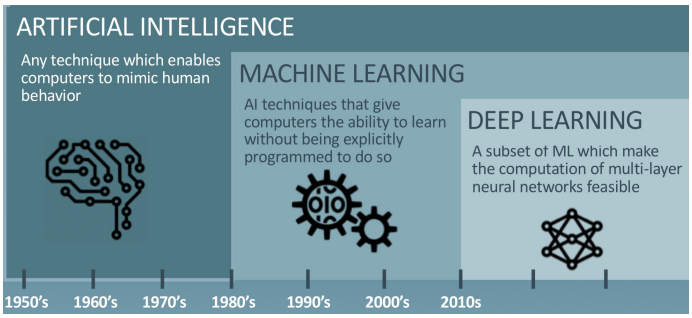


Figure 2.2: Timeline of the advances in artificial intelligence and the rise of machine learning.

In ML, the explicit programming was replaced with a learning phase. In this phase the machine is *taught* how to solve a problem. Once done, the second phase follow and is called the inference phase. In this phase, the algorithm is deployed and is ready to perform the task it was trained for.

Depending on the training phase, many types of ML algorithms arises. In [16], six types were discussed:

- Supervised learning: this is the most common form of ML algorithms. In this class, a labeled dataset is used to train the algorithm. During this training, the algorithm will extract the similar characteristics of elements with the same label to classify a given input into the most adequate class. A common misconception is that supervised learning is classification. This is false since regression -where the algorithms learns the evolution of an output based on input variations- and detection -where the algorithm learns how to distinguish and separate objects- are also algorithms with a supervised training.

- Unsupervised learning: the difference between the supervised and the unsupervised learning is the dataset. In this class, no labels are given. During the learning phase, the algorithm learns the existing similarities between samples in the dataset as stated in [69].

- Semi-supervised learning: this type of ML algorithms is somewhere between the first two. The dataset contains some labeled samples but may contains items with no labels. The training algorithm should be able to process both samples in order to learn features from the dataset.

- Reinforcement learning: in this class of algorithms, no dataset is present. Learning consist on finding the best policy that an agent should follow in a simulated environment in order to achieve a goal. When interacting with said environment, the agent receives reward for positive decisions and penalties for negative ones. This reward mechanism helps the agent find the best policy.

- Transduction: this is not a very common class in ML. Similar to supervised learning, the algorithm tries to map an input to an output based on a given dataset. The main difference being, transduction algorithms does not explicitly construct a mapping function but predicts outputs based on the dataset.
- Learning to learn: this is a new class of ML algorithms. It falls under the category of **self-aware** systems. In this class, the algorithms learns its own inductive bias based on previous experience.

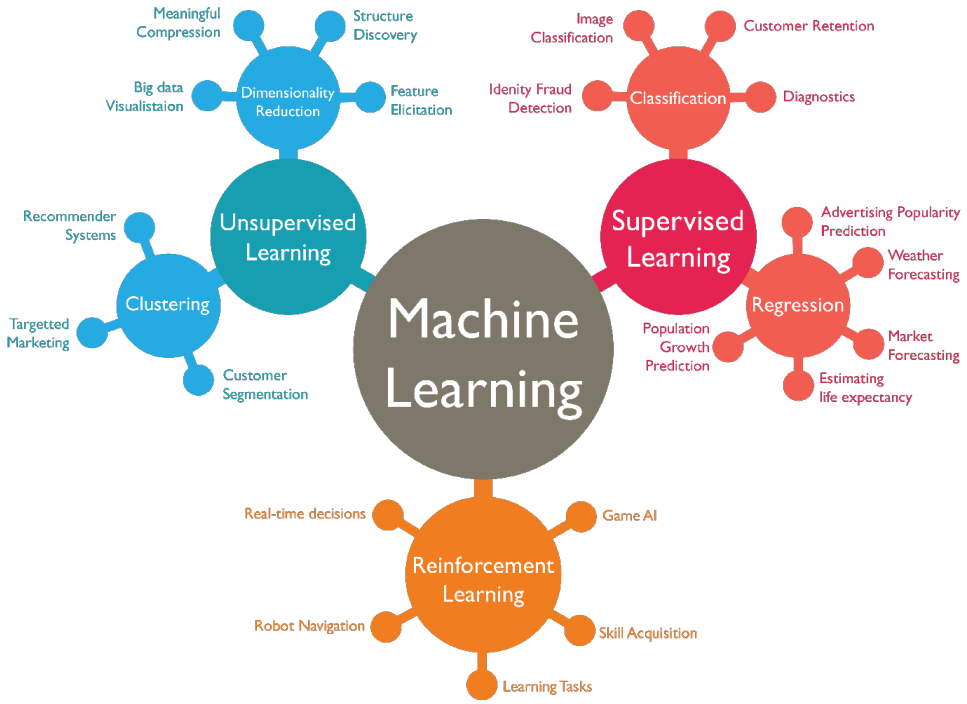


Figure 2.3: Classification of ML algorithms and sub-categories of each class with some example algorithms are given at the nodes of each sub-category.

In Figure 2.3 we show the most common types of ML algorithms, namely, supervised, unsupervised and reinforcement learning. In these algorithms, the quality of the algorithm highly depends on the quality of samples in the dataset. The exception being for reinforcement learning where no dataset is present. In this case, the quality depends on how well the environment is simulated and the reward-penalty system is designed.

2.2 Motivations Behind Machine Learning

As stated in [167], the two biggest drivers of the recent progress of machine learning are **computational scale** and **data availability**. The first cause was discussed in the introduction of this chapter with the introduction of parallel processing in the recent compute platforms. As for the second, data abundance is clearly visible with the widespread of social media applications.

It was estimated by Middlesex [150] that at least 1200 petabytes are stored in the databases of the giant four companies: Google, Amazon, Microsoft and Facebook. In the case of Facebook, more than 300 million photos are uploaded each day [32] and every minute, 510000 comments are posted and 293000 statuses are updated [169]. This large amount of data challenges the most sophisticated traditional algorithms that rely mostly on handcrafted policies and fine-tuned solutions to specific problems.

This large amount of information helps constructing bigger and better databases which, consequently, results in better algorithms. Combined with powerful platforms to accelerate processing, ML has been established as a *de-facto* approach in AI in the last decade and continues to thrive.

2.3 How it works?

Artificial Neural Networks are an example algorithm of ML that mimics the human brain by replicating, to certain extent, its architecture. They are by far the most used ML algorithms. For this, we dedicate Section 2.3.2 to present this category of algorithms. Section 2.3.3 details a variant of ANNs that deals mainly with images: CNNs.

2.3.1 The back propagation algorithm

The ability to grasp large amounts of data was mainly achieved by using the back propagation algorithm [87], [178]. This algorithm is used during the learning, or training, phase of almost every neural network. It consists on forwarding the inputs through the network. The results would most probably be erroneous. The error is quantified and its values is propagated back in order to update the network parameters.

In order to understand this learning process, we consider a function $y = f(x) = ax + b$ that takes an input x and gives an output y. In machine learning, the problem is to find the parameters a and b in order for this function to output a **desired** value y for a given input x. The relationship between x and y is explained as a list $\mathcal{L}$ of n pairs $(x_i, y'_i)_{i=1\rightarrow n}$ which corresponds to the desired output y'_i of the function f when the input is x_i.

The process of training, or learning, uses the list $\mathcal{L}$ to find the best parameter pair (a, b) in order to increase the output quality of the function f. The quality is measured using an error function $\mathcal{E}_i(a, b)$. The error function takes the current value of the parameter pair (a, b) and measures the distance between the actual output y_i and the desired output y'_i of f for a given input x_i from the list L.

$$\mathcal{E}_i(a, b) = |y_i - y'_i| \tag{2.1}$$

In Equation 2.1, we estimate the error, by choice[1], as the absolute value of the difference between the outputs y and the desired outputs y' for each value of $x \in \mathcal{L}$. Since $y_i = f(x_i) = ax_i + b$, Equation 2.1 could be written as follow:

$$\mathcal{E}_i(a, b) = |ax_i + b - y'_i| \tag{2.2}$$

For a given pair (a, b), the total error $\mathcal{E}(a, b)$ would be calculated as the sum over all pairs $(x_i, y'_i) \in \mathcal{L}$ as follows.

$$\mathcal{E}(a, b) = \sum_{i=1}^{i=n} |ax_i + b - y'_i| \tag{2.3}$$

This error can be visualized in a three-dimensional grid by varying a and b and calculating the error $\mathcal{E}$ every time. In the following, we consider the problem as stated before with $\mathcal{L}$ being the list of pairs from Table 2.1.

Figure 2.4 shows the distribution of the error function from Equation 2.3. These values represent the error on the output of the function $f(x) = ax + b$ for a given pair (a, b) and the outputs from the list $\mathcal{L}$. We can visually find the minimum of this error

[1] Many error functions exists such as the squared error. Error functions usually gives positive values by introducing operators such as the absolute value. The reason behind this it to avoid errors canceling out.

Input (x)	1	-2	4	3	2	-1	-1
Desired output(y)	2	5	17	10	5	2	2

Table 2.1: Example of a labeled dataset.

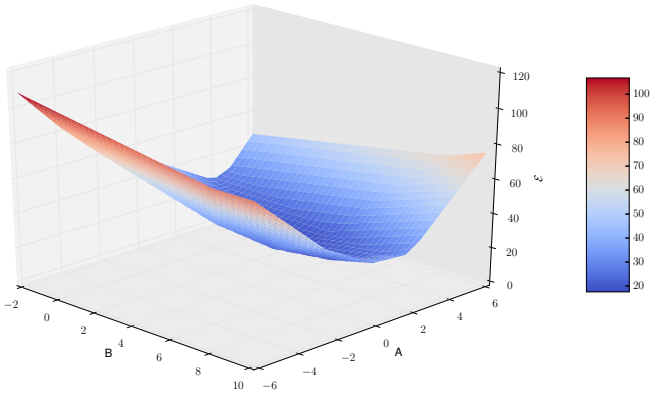


Figure 2.4: Distribution of the error function $\mathcal{E}$ from Equation 2.3.

function in the intervals $(a, b) \in ([-6, 6], [-2, 10])$ which is the goal of this phase. This minimum can be local, however, it was noticed that the presence of a local minima in the error function does not seem to be a major problem in practice [129].

However, this visualization might not be practical. It is highly dependant on the choice of the intervals and, for complex function, it would be hard to recognize a pattern that helps localize the minima. Therefore, in order to increase the quality of the output, we need to analytically find a value for (a, b) that minimizes this error function. The problem statement is shown in Equation 2.4.

$$\min_{(a,b)\in\mathbb{R}^2, i=1\to n} \mathcal{E}_i(a, b) = |ax_i + b - y_i'| \tag{2.4}$$

This problem can be solved numerically. We first initialize a and b randomly to (a^0, b^0) and then, we iteratively update their values while reducing the magnitude of the error function. The update of weights can not be arbitrary. Numerical methods for finding the minimum could be used such as Newton's method. In ML, the most common optimizer is the gradient descent due to its simplicity and efficiency. Similar to Newton's method, it relies on the slope, or gradient, of the function at a given point. If at a point a_0, the slope of $\mathcal{E}$ is negative, then the function is rising. Therefore, we should reduce the value of a_0 in the next iteration in order to get a lower value of $\mathcal{E}$. Since the error function takes two arguments, the slope is computed as the partial derivative with respect to each parameter. The update procedures are shown in Equation 2.5.

$$\begin{cases} a_{k+1} = a_k - \frac{\partial \mathcal{E}(a_k, b_k)}{\partial a} \\ b_{k+1} = b_k - \frac{\partial \mathcal{E}(a_k, b_k)}{\partial b} \end{cases} \tag{2.5}$$

In analogy with ML, the list $\mathcal{L}$ from our simplified example represent the training dataset. The training is supervised since the dataset contain the inputs with the correspondent outputs or labels in the case of an image classification task. The parameters a and b represent weights. A larger algorithm would require more than two parameters to explain complex relations, hence, the need of a weight vector and

a more complex function f. In such a case, visualization can no longer be considered since it results in an $N+1$-dimensional space with N^2 the number of parameters of the network.

The initial gradient descent algorithm for learning has been adapted and improved in many recent works. For example, in Equation 2.5, a learning rate α is introduced to slow down the descent towards the minimum. This addition has shown to be primordial in order for the algorithm to converge towards the minimum. One popular variation is the Stochastic Gradient Descent algorithm.

2.3.2 ANN Architecture

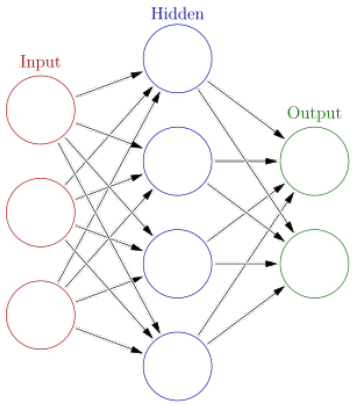


Figure 2.5: Architecture of an Artificial Neural Network.

ANNs are a brain-inspired class of ML algorithms. They map a set of inputs to a set of outputs by forwarding the former through a network of compute elements called neurons. This architecture can be seen in Figure 2.5, in it, we see three layers: an input layers, a hidden layer and an output layer. In Figure 2.6, we see an analogy between artificial neurons and a real neuron in the human brain.

The number of neurons in the input and the output layer depend on the problem while the number of neurons in the hidden layer is a design parameters. Also, multiple hidden layers could be present with the same interconnection scheme. The general rule is that more neurons and layers are require to yield better representations since it allows more complex representations. However this is not a sufficient to assure better accuracy and other design parameters needs to be tuned.

Neurons function in a simple way. Let us considering a vector X of n elements $x_{i=1\to n}$. A weight $w_{i=1\to n}$ is assigned to each connection between the inputs and the neuron. The first step of processing consist on multiplying each input x_i by the correspondent weight w_i. A bias b is then added to the weighted sum. For simplicity, the term b is usually padded to the weight vector as w_0 and is multiplied by a value x_0 added to the input vector. The final output is obtained by applying an activation function $\mathcal{A}$ on the resulting sum from the previous step. This can be summarized in Equation 2.6.

$$O = \mathcal{A}(\sum_{i=0}^{n} x_i * w_i) \tag{2.6}$$

The activation function $\mathcal{A}$ is important for many reasons. It's role is to keep the output of each neuron in check. A neuron compute the weighted sum of all of its

[2]In the given example, the number of parameters is two-a and b. This results in a 3-dimensional space shown in Figure 2.4. The two parameters constitutes the (x, y) axis and the z axis represent the value of the error function.

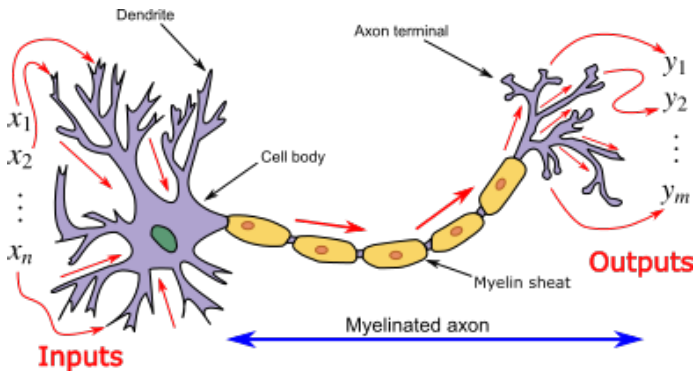


Figure 2.6: Analogy between a neuron in the human brain and artificial neurons.

inputs. Stacking multiple layers may results in large output sums that overshadows one another. Sigmoid function can be used since its output is guaranteed to be in $[0, 1]$.

Moreover, the activation function needs to be derivable. We have seen in Equation 2.5 that updating weights requires partial derivatives of the error function. In an ANN, the error function uses Equation 2.6 to compute outputs. Therefore, a valid derivative is required to compute the gradients. Finding the derivative, either numerically or analytically, might be tedious if the activation function is complex. This pushed many researchers to choose simpler activations for better performance. A common activation function used in recent architectures is the Rectified Linear Unit (ReLU) whose output is given by Equation 2.7.

$$ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

While this activation function does not solve the problem of large outputs, its easy computation and derivation made it the designer's choice. Normalization on the inputs and the outputs of each layer was introduced to cope with this problem whilst being able to use simple activation functions.

All layers in an ANN are composed of neurons with this particular architecture. Designers have to choose the ideal number of layers as well as the number of neurons in each layer. The connection between neurons is also a design parameter. Based on the connections between neurons, many types of neural networks can be distinguished. A fully-connected neural network is when every neuron in layer i is connected to every neuron in the subsequent layer $i+1$ and no neuron from layer $i+1$ is connected backwards to neurons from layer i. Layers with such constraints are called *fully connected layers*. Backwards connections are enabled in certain types such as Recurrent Neural Networks.

ANNs has proven to be extremely efficient in many classification and regression tasks. However, their structure does not allow them to process structured data. The indices of the input vector are superficial and the order is not exploited.

If we apply Equation 2.6 to the input layer of the network, the output is unchanged if two neurons x_i and x_j are interchanged with their respective weights w_i and w_j. This means that the training process will still achieve the same accuracy if the order of inputs is changed.

The order of inputs is important in certain types of inputs such as images where the meaning (of the image) is expressed by the value of each pixel **and** it's neighbourhood. This motivates the development of new types of architectures that exploits this aspect.

2.3.3 CNN Architecture

Images has different structures. The important information is not extracted from a single pixel but from its neighbourhood. While using a large ANN may solve this problem, the usage of convolutions can achieve similar results with smaller networks. The added convolutions act as feature extractors and the fact that they are trained automatically makes them automatic feature extractors. These features can then be classified or processed using an ANN in order to produce the desired outputs.

A CNN is a neural network architecture that has these automatic feature extractors embedded as layers in the architecture. The introduction of these layers results in large networks. In the jargon of computer vision, these networks are referred to as Deep Neural Networks. Most state-of-the-art CNNs has this quality which qualifies them as DNNs.

Exactly as machine learning is a branch of artificial intelligence, deep learning, which is the study of DNNs, is a sub-class of machine learning. This is illustrated in Figure 2.2. While many researchers struggle to define a clear barrier between the two, the most visible distinction is that deep learning automatically extracts features whilst in other machine learning approaches, features are human handcrafted. An other way of distinguishing the two classes is the number of layers where a network with less than 10 layers is considered shallow.

2.3.4 The Convolutional Layer

The convolutional layer is the trademark of CNNs. It takes a feature map $\mathcal{F}$ as input. A feature map is a 3-dimensional array[3] of dimensions $W \times H \times C$. In the case of an input layer, the value C represents the number of channels in the input image (3 in the case of RGB and 1 in the case of grayscale images). W and H represent the dimensions of the image, namely, width and height.

Neurons in this layers are filters. In ANNs, each neuron takes every input and multiply it by a weight. A convolutional layer does the same thing with two modifications. The first one being, the inputs of the neuron are not individual values of the feature map but windows of $K \times K$[4] neighbouring elements in order to incorporate the locality information. The second modification is weight sharing. The convolutional layer neuron would multiply every input window with the same filter greatly reducing the number of parameters. For the sake of comparison, an ANN neuron assign a distinct weight to every input.

The role of convolutional layers is to extract high level features from inputs, usually images. This operation is similar to previous computer vision filtering algorithms such as the Sobel operator [115] for edge detection. By convolving an input image with these filters, multiple representations of the image are generated. Each one of these new representations contain a separate feature of the image. By re-applying filters in a succeeding layer to these representations, higher features could be extracted. Repeating this process results in a set of complex features of an image that could later be classified using an ANN.

While these convolutional layers reduce the number of parameters, the number of outputs is tremendous compared to fully connected layers used in ANNs. For this reason, an aggregation layer usually follow convolutional layers to reduce the size of the output feature map. This is achieved by grouping, or pooling, neighbouring values into a single one. There are many types of pooling. The most common types are maximum pooling and minimum pooling. The maximum, respectively minimum,

[3] These arrays are also referred to as *tensors* in some libraries such as Tensorflow. Also, a fourth dimension could be added to the feature map in order to process multiple inputs at the same time. This practice is called batching and it is widely used in modern hardware accelerators as we will see later in Section 4.

[4] The input window can be non square with a size $K \times L$ with $L \neq K$. However, almost every major CNN architecture has square filters that takes square inputs. For this reason, we simplify the notation by only considering square input windows.

values are kept for each window. The size of the pooling window is a design parameter. The choice of maximum, respectively minimum, is also a design parameters and was adopted due to its simplicity in computation while maintaining information about the values in the window.

2.4 Datasets

As stated earlier and in [167], the availability of datasets allowed gradient-based learning and automatic feature extractors to flourish. The quality of said datasets ensures efficient training which leads to high accuracies.

Figure 2.7: Subset of MNIST images.

One of the first datasets to be used for traning machine learning algorithms is MNIST. A subset of this dataset is shown in Figure 2.7. It has 60.000 images of 28×28 pixel gray scale images. The images are for handwritten digits. The dataset is split into a training set with 50.000 images and a test set with the remaining 10.000 images. The classes of this dataset are the digits from 0 to 9. Algorithms trained on this dataset try to correctly classify an input image into one of the digits classes.

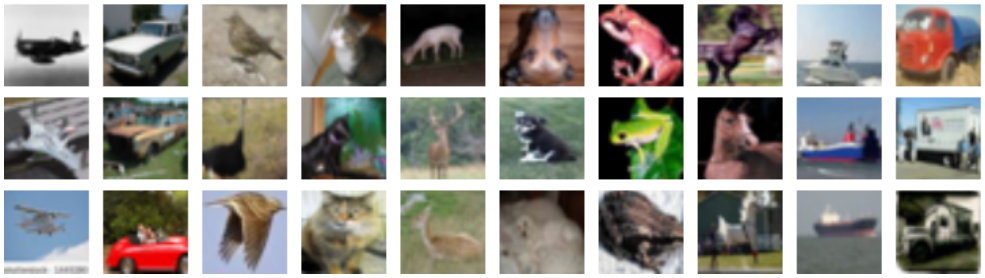


Figure 2.8: Subset of CIFAR-10 images.

A dataset with more general classes is CIFAR. It has two versions, CIFAR-10 which is shown in Figure 2.8 and has 10 classes and CIFAR-100 with 100 classes. The classes of this dataset are objects found in the world such as cars, people and animals. The 10-class variant has 80 million labeled images of size 32×32. It is designed for fast training with such small sized images.

The largest dataset to date is ImageNet [52] shown in Figure 2.9. It has more than 14 million images classified into 20.000 categories. A subset of ImageNet is used in ILSVRC, the most notorious challenge of image classification. In the challenge, only 1000 classes are considered.

2.5 Common CNN Architectures

2.5.1 LeNet-5

The first successful architecture of a CNN was introduced in LeNet-5 [129], or just simply LeNet. LeNet was designed for isolated character recognition. As shown in Figure 2.10, LeNet consists of two parts (from left to right): a feature extractor, and a classifier appended to it. Feature extraction is performed by two unpadded

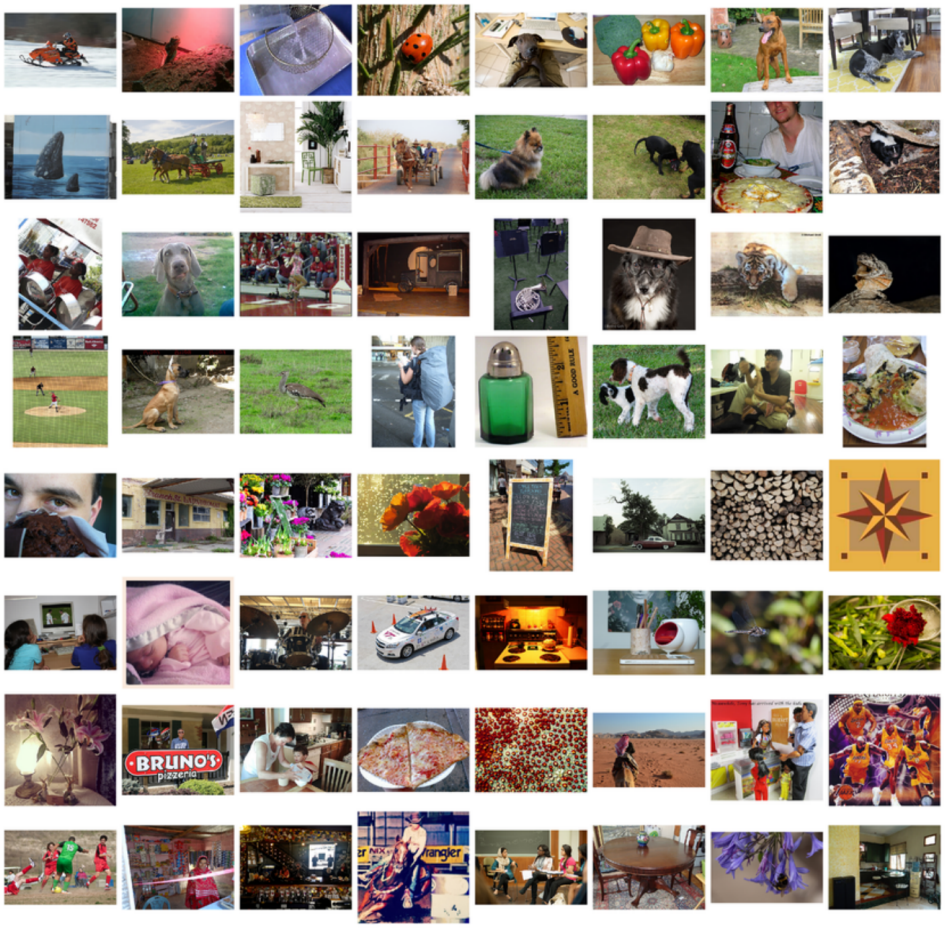


Figure 2.9: Subset of IMAGENET images.

convolutional layers with small filters of size 5×5, with 20 and 50 filters respectively, and followed by ReLU activations. In order to mitigate the increase of dimensionality, each convolutional layer is followed by a max pooling operation, downsampling the resulting feature vectors. The classifier is composed of 3 fully connected layers, terminated by a Softmax activation.

LeNet-5 achieved a 99.2% accuracy on the MNIST dataset. While it has been bested by many modern architectures recently[5], this result, and others that follow, cemented the position of computers as a human alternative even in high-risk fields such as banking and the medical sector.

2.5.2 AlexNet

The 2012 Imagenet Large Scale Visual Recognition Challenge (ILSVRC) was the first major success of machine learning. AlexNet [124], a CNN architecture, won the image classification challenge whilst beating human accuracy. This milestone was reached by using 5 convolutional layers separated by maximum pooling and normalization layers. Classification is done using 3 fully connected layers with Softmax activation at the end.

[5]The current state-of-the-art best is [226] with 0.21% error rate. This is a CNN with 2 convolutional layers, a fully connected layer, a ReLU layer and a special DropConnect layer.

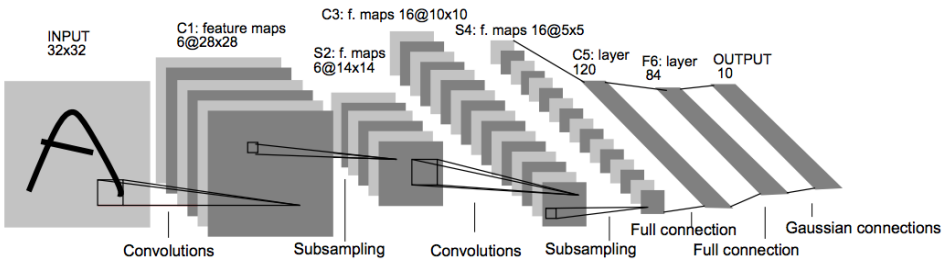


Figure 2.10: Architecture of LeNet5.

2.5.3 VGG16

VGG-16 [204] was an improvement on AlexNet. With more layers and similar architecture, the main difference is the kernel sizes. Convolutional layers use 3×3 kernels at all levels for easier hardware re-use[6].

2.5.4 GoogleNet

Winner of the 2014 ILSVRC, GoogleNet [212] is a 22 layers network. It introduces inception layers which is the concatenated output of three parallel convolutional layers with different filter sizes. The inception layer (or module) uses small size filters for fine resolution and large filters for larger receptive field.

2.5.5 SqueezeNet

SqueezeNet [98] is a lightweight CNN used for mobile devices. It was designed with 1×1 kernels which reduce the size of the network parameters vector. With a million weights, this network fits easily in fast caches and on-chip memories (such as BRAMs for FPGAs). Besides this change, the architecture of the network was inspired by AlexNet.

2.6 CNN Profiling

While CNNs reduce the number of required weights by sharing filters[7], the number of operations is greatly increased since multiplications are replaced by convolutions. This increased number of operations can be seen in Table 2.2.

The number of operations is in terms of tens of billions (Total workload row from Table 2.2). These operations needs to be performed for each new input. This raises the need for efficient hardware that is capable of dealing with such workloads.

Many operations are performed during a single inference run of a CNN. In Figure 2.11, we show the number of these operations for six different networks. It is obvious that the majority of workload is dedicated to MAC operations. In other words, one can focus on optimizing MAC hungry layers thereby ensuring the most impact on the overall performance of the CNN inference.

Figure 2.12 shows the distribution of MAC operations over network layers. The pie charts shows that the convolution layers are the most MAC hungry layers. With a good MAC implementation, a high parallel implementation of these two layers can be sufficient to achieve high overall performance. This is visible in the state of the

[6]FPGA implementations can design a 3×3 kernel processor which will then be used to execute the whole network.

[7]The same filter is applied to the whole input of the layer whereas in ANNs, each input x_i has it's own dedicated weight w_i (see Equation 2.6).

Table 2.2: Accuracy and workload of famous classification CNNs with the same input size.

Model	AlexNet	GoogleNet	VGG16	VGG19	ResNet50	ResNet101	ResNet152
Top1 err	42.9%	31.3%	28.1%	27.3%	24.7%	23.6%	23.0%
Top5 err	19.80%	10.07%	9.90%	9.00%	7.8%	7.1%	6.7%
conv layers	5	57	13	16	53	104	155
conv worload (MACs)	666 M	1.58 G	15.3 G	19.5 G	3.86 G	7.57 G	11.3 G
conv params	2.33 M	5.97 M	14.7 M	20 M	23.5 M	42.4 M	58 M
Activation	ReLU						
pool layers	3	14	5	5	2	2	2
FC layers	3	1	3	3	1	1	1
FC workload (MACs)	58.6 M	1.02 M	124 M	124 M	2.05 M	2.05 M	2.05 M
FC params	58.6 M	1.02 M	124 M	124 M	2.05 M	2.05 M	2.05 M
Total workload (MACs)	724 M	1.58 G	15.5 G	19.6 G	3.86 G	7.57 G	11.3 G
Total params	61 M	6.99 M	138 M	144 M	25.5 M	44.4 M	60 M

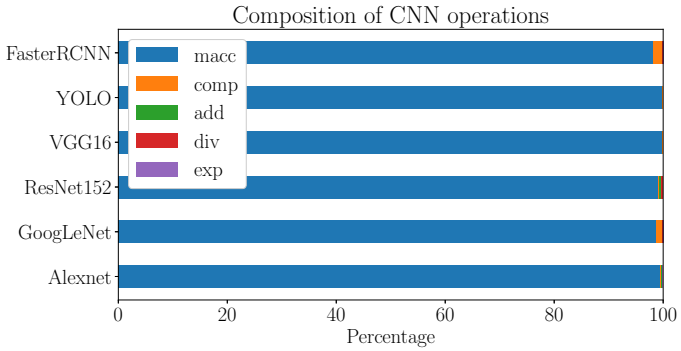


Figure 2.11: Percentage of Multiply and Accumulate (macc), comparison (comp), addition (add), division (div) and exponent (exp) operations for different networks

art study in Section 5.2 where most presented works ignores other layers and only focus on these two.

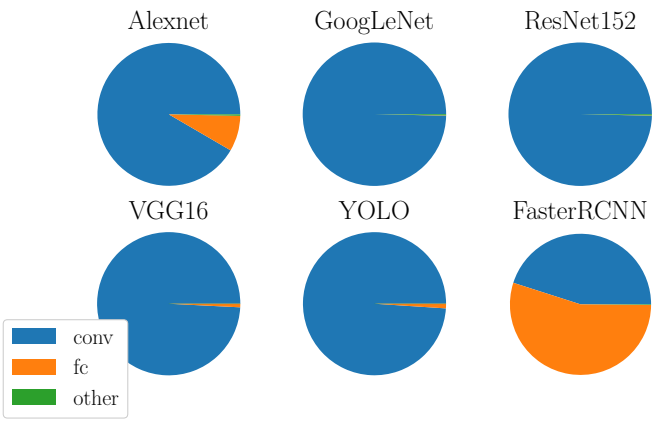


Figure 2.12: Distribution of MACC operations over different network layers.

Chapter 3

Autonomous Driving Systems

Contents

The race towards autonomous vehicles was initiated by the US' Defense Advanced Research Projects Agency (DARPA) with Navlab and ALV projects in 1984. However, with 749,000,000€ funding, the Eureka Prometheus Project is considered the largest in the field of driverless cars. In 2004, DARPA organized the first major fully automated driving challenge. Attendees were asked to finish an off-road parkour with **no** human intervention. No attendee managed to finish the challenge. However, in a second challenge in 2005, five teams reached the finish line [27]. With the advent of machine learning, this race gained in intensity. The high perception capabilities offered by state of the art algorithms solved many problems that were, at the very least, challenging.

In a released taxonomy for terms related to driving automation systems [195], the Society of Automotive Engineers defined six levels of automation. These levels goes from complete human control to fully autonomous driving systems. The fulfilled tasks in each system decide which level the system is classified. As shown in Figure 3.1, a system qualifies as autonomous by monitoring the driving environment and taking control over steering and acceleration ; if one of these two tasks is made, fully or partially, by a human driver, the system is considered manual with machine assistance.

SAE level	Name	Narrative Definition	Execution of Steering and Acceleration/ Deceleration	*Monitoring* of Driving Environment	Fallback Performance of *Dynamic Driving Task*	System Capability *(Driving Modes)*
***Human driver* monitors the driving environment**						
0	No Automation	the full-time performance by the *human driver* of all aspects of the *dynamic driving task*, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	n/a
1	Driver Assistance	the *driving mode*-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	Human driver and system	Human driver	Human driver	Some driving modes
2	Partial Automation	the *driving mode*-specific execution by one or more driver assistance systems of both steering and acceleration/ deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	**System**	Human driver	Human driver	Some driving modes
***Automated driving system* ("system") monitors the driving environment**						
3	Conditional Automation	the *driving mode*-specific performance by an *automated driving system* of all aspects of the dynamic driving task with the expectation that the *human driver* will respond appropriately to a *request to intervene*	System	**System**	Human driver	Some driving modes
4	High Automation	the *driving mode*-specific performance by an automated driving system of all aspects of the *dynamic driving task*, even if a *human driver* does not respond appropriately to a *request to intervene*	System	System	**System**	Some driving modes
5	Full Automation	the full-time performance by an *automated driving system* of all aspects of the *dynamic driving task* under all roadway and environmental conditions that can be managed by a *human driver*	System	System	System	**All driving modes**

Figure 3.1: The six levels of driving automation as defined by SAE International in J3016.

In this chapter we will give certain examples of automation levels. First we focus on the first three levels, considered manual with automatic driver assistance. Next, we pass to the other three levels which are still in research phase. Later on, we give some datasets and simulators that helps designing autonomous systems. Some example frameworks are given in the last section.

3.1 Driver-Assistance Systems

The goal of automation is to free the driver of certain tasks. Each automation system perform a set of driving tasks. These systems are also referred to as ADAS. ADAS are designed to limit human error [24]. It was estimated in [206] and [254] that more

than 90% of road accidents are attributed to human error. The source of these errors are human fatigue or inattention. The human accuracy in taking decision might not be perfect but is far from being the reason behind accidents. An ADAS performs tasks that are 1) repetitive, such as lane keeping and cruise control or 2) pre-defined, such as parking and overtaking. An ADS on the other hand is a system that replaces the human driver. Not only it will executes the decision like ADAS, but it will also take that said decision.

3.2 Autonomous Driving System

The architecture of an ADS was given [140]. This architecture is shown in Figure 3.2. The tasks of a driving system could be extracted from this figure.

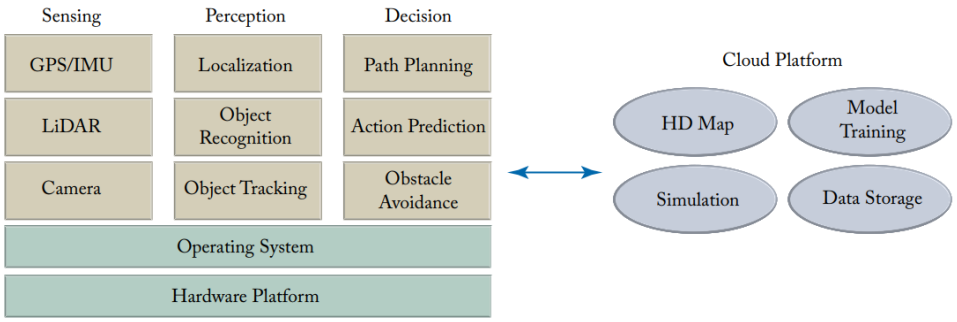


Figure 3.2: Autonomous driving system architecture overview from [140].

The three major components of an ADS are: the sensing module, the perception module and the decision module. These three software modules are run by an operating system on a hardware platform. The system is connected to a cloud platform that handles off-line tasks such as model training and data storage and some on-line tasks such as logging. The user interface is needed for production systems, however, it is not a required module for a functioning.

3.2.1 Sensory Module

This module provide the information required to understand the environment. RADARs were the first sensors used to perform ADAS tasks. Due to their accuracy in obstacle detection, they were used for braking and reversing. Modern cars are equipped with $60 - 100$ sensors. In the Automotive Sensors and Electronics Expo, it was estimated that this number will reach 200 sensors per car on average. In modern cars, these sensors and their locations are shown in Figure 3.3.

In Table 3.1, the main three sensors (RADAR, Lidar and Camera) were compared to each other and to the human driver. As can be seen, no single sensor is capable of providing enough information to accurately perform all the tasks at once. Sensor fusion is a must. From an other perspective, communication is a valuable asset that could be considered part of the sensory module. Being able to communicate with the infrastructure and other road users is crucial and gives the edge to connected systems over the human driver.

This observation leads to increasing number of deployed sensors which raises another problem, the need of compute power. In an estimation of ARM [13], autonomous vehicles will required $100\times$ more compute performance by 2024 compared to 2016 models.

In many cases, sensors are equipped with small compute units. These units are used for near-sensor operations such as decoding and data pre-processing [162]. These

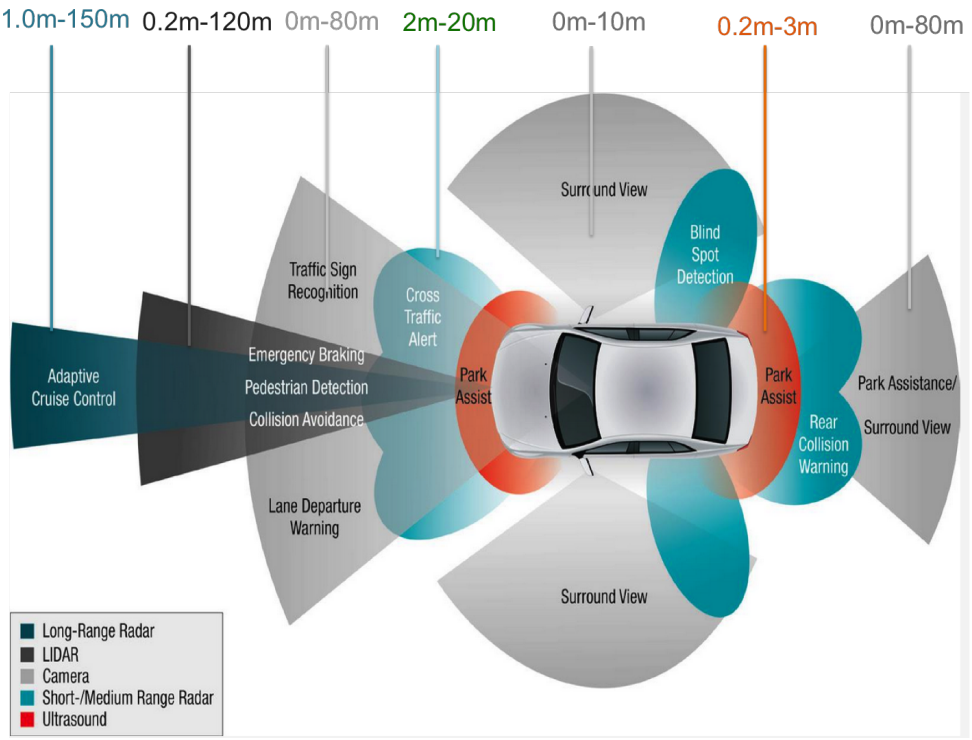


Figure 3.3: Automotive sensors and their deployment on autonomous vehicles.

Performance aspect	Human	AV			CV	CAV
		Radar	*Lidar*	*Camera*	*DSRC*	*CV+AV*
Object detection	Good	Good	Good	Fair	n/a	Good
Object classification	Good	Poor	Fair	Good	n/a	Good
Distance estimation	Fair	Good	Good	Fair	Good	Good
Edge detection	Good	Poor	Good	Good	n/a	Good
Lane tracking	Good	Poor	Poor	Good	n/a	Good
Visibility range	Good	Good	Fair	Fair	Good	Good
Poor weather performance	Fair	Good	Fair	Poor	Good	Good
Dark or low illumination performance	Poor	Good	Good	Fair	n/a	Good
Ability to communicate with other traffic and infrastructure	Poor	n/a	n/a	n/a	Good	Good

Table 3.1: Summary of the key operating characteristics of each sensor as they apply to autonomous vehicles from [200]. AV stands for Autonomous Vehicles, CV for Connected Vehicles, CAV for Connected Autonomous Vehicles and DSRC for Dedicated Short-Range Communications.

operations are important in order to reduce the impact of environment noise. Kalman Filter is widely used in this step for enhancements and error correction. Other operations such as sensor calibration and synchronization are also performed near the sensory unit as a preprocessing step.

3.2.2 Perception Module

The role of this module is to understand the environment using the information previously gathered by sensors. The vehicle needs to 1) localize itself in the environment

and 2) detect other actors and interferers.

Localization is performed by using inputs from Global Position System (GPS) units deployed on the vehicle. However, GPS readings might not be accurate, therefore, Inertial Measurement Unit (IMU) are deployed in order to enhance the estimated position of the vehicle. The estimated position is passed to the decision module in order to plan the trajectory. In [10], the localization and the navigation are performed using a special type of neural networks, Variational Neural Networks (VNNs). As shown in Figure 3.4, the VNN takes the camera input as well as the map position and try to navigate the driveable area towards the target path. Modules that process localization and navigation are called SLAM (Simultaneous Localization And Mapping). SLAMs are present in most robotic systems.

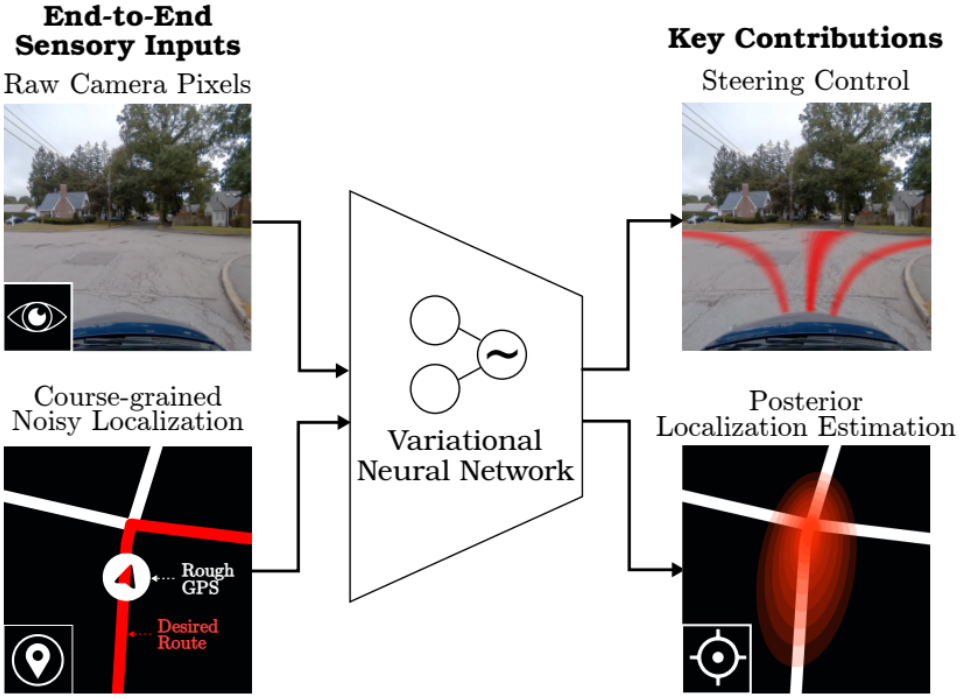


Figure 3.4: Localization, navigation and control using a neural network from [10].

The position is also used in the perception module to filter important actors in the environment. These actors are detected by using the inputs from cameras and Light Detection and Ranging. Important actors are other road users such as pedestrians and other cars. Road signals such as street lights and signs are also detected in this phase.

The perception module is also responsible for detecting the navigable road surface and the lanes. This area is important in order to maintain the vehicle centered on the road.

3.2.3 Decision Module

This is an aggregation module that uses the raw data from sensors and the outputs from perception units in order to drive the vehicle. The main tasks of this module are path planning, prediction and control.

The planning phase is divided into two components. First a high level planner will find the shortest way to the final destination. This service can be performed using internet resources such as Google Maps. A local path planner is needed however to plan local maneuvers such as lane changing, turn execution and overtaking.

The local plan is then executed by the control unit that will directly act on the steering wheel and the throttle to reach the desired goal.

3.2.4 Hardware Platform

The software components of the systems run on an operating system[1]. These components needs to take decision in an acceptable time in order for it to safely drive the vehicle and not collide with other road users. The role of the hardware platform is to host the operating system and all the components of the system.

With such a constraint, general purpose computers lack significantly in processing power. The design of fast dedicated hardware is a requirement. This is partly the aim of this thesis; choosing the best embedded system, capable of ensuring these constraints with the minimal possible overhead on the cost of the vehicle.

Besides performance, reliability is an other issue in autonomous driving. Hardware faults occurs all the time. Letting these faults propagate through the system would lead to disastrous consequences. The hardware component should, as well as the software component, be fault resilient or has recovery mechanism after faults occurs.

3.2.5 Example Systems

Existing systems perform these tasks with slight modification on the architecture from Figure 3.2. We distinguish three types of implementations:

- Pipelined systems: A pipelined system is a direct implementation of the architecture in Figure 3.2. Each module is implemented as such. Data flow through the system while being enriched and transformed by the different modules in the pipeline.
- End-to-End learning: In [167], the rise of end-to-end learning was evoked. As defined in [167], an end-to-end learning algorithm would take as input the raw original data and try to directly generate the final output. In the case of ADS, the desired steering and acceleration outputs are learned directly from the input images and LIDAR point clouds. End-to-end systems are therefore easy to design. The drawback of these methods is debuggability. If an error -or bad decision- occurs, it is hard for human experts to trace back its origin and even harder to modify the network to fix it. With the high risk value in ADS, end-to-end learning is used with high discretion.
- Reinforcement Learning: Similar to end-to-end learning, reinforcement learning tries to find the optimal policy that the car should follow in order to reach its goal. Besides inputs, the perception and decision modules are merged together and the final decision is directly executed on the vehicle.

In Section 3.6 we give concrete examples on the first two types with more details on the actual implementation.

3.3 Datasets and Challenges

The quality of machine learning algorithms depends on 1) the efficiency of the training algorithm when extracting features and 2) the richness of the training dataset. However, the training algorithm can never extract more features than there are in the dataset. For this reason, many tailored datasets exists. For ADSs, a selection of these datasets is shown in Table 3.2.

[1]It is possible to implement all the functions as bare-metal applications. However, a full system with interface, communications, scheduling and other tasks would be so complex. The use of operating systems on top of hardware is almost mandatory.

The available datasets for autonomous driving focus on driving scenarios. First, the sensors are considered. Rather than relying on images, a vehicle can use LIDAR and RADAR inputs to detect obstacles or understand the environment. Datasets such as KITTI [67] offer calibrated LIDAR point-clouds with synchronized images. The synchronization information can be very critical to enrich the information passed to the network which would reflect positively on the output quality. Other factors such as time and weather can be critical for a level 5 system where the car is expected to drive anywhere with full autonomy.

Synthetic data is also helpful to design better architectures. The test-bed of an autonomous vehicle is costly, especially for small research groups. Therefore, a simulator is used to create a cheap indoor test-bed. Simulation skips the need of manual annotation since data can be directly extracted. While synthetic data can not be directly ported to real life scenarios, the algorithm can be retrained, or even fine-tuned with real data in order to test it on real life scenarios.

Table 3.2: Driving datasets from [254].

Dataset	Image	LIDAR	2D annotation	3D annotation	ego signals	Naturalistic	POV	Multi trip	all weathers	day & night
Cityscapes [45]	✓		✓				Vehicle			
Berkley DeepDrive [252]	✓		✓				Vehicle		✓	✓
Mapillary [166]	✓		✓				Vehicle		✓	✓
Oxford RobotCar [148]	✓	✓					Vehicle	✓	✓	✓
KITTI [67]	✓	✓	✓	✓			Vehicle			
H3D [177]	✓	✓		✓			Vehicle			
ApolloScape [95]	✓	✓	✓	✓			Vehicle			
nuScenes [28]	✓	✓	✓	✓			Vehicle		✓	✓
Udacity [219]	✓	✓	✓	✓			Vehicle			
DDD17 [21]	✓		✓		✓		Vehicle		✓	✓
Comma2k19 [199]	✓				✓		Vehicle			✓
LiVi-Set [39]	✓	✓			✓		Vehicle			
NU-drive [213]	✓				✓	Semi	Vehicle	✓		
SHRP2 [218]	✓				✓	✓	Vehicle			
100-Car [119]	✓				✓	✓	Vehicle		✓	✓
euroFOT [20]	✓				✓	✓	Vehicle			
TorontoCity [231]	✓	✓	✓	✓			Vehicle, Aerial, Panorama			
KAIST multi-spectral [41]	✓	✓	✓				Vehicle			✓

3.4 Autonomous Vehicle Constraints

Due to ethical dilemmas such as the Trolley problem[2], ADSs needs to surpass human driving capabilities with a safe margin for it to be considered an everyday replacement. This safe margin dictates the constraints of designing an ADS. The main constraints are enumerated as follows:

- Safety: The major concern in autonomous vehicles is safety. The software and hardware components of the ADS should be able to detect dangers and prevent them. For the hardware, this can be seen as the reliability of the platform. The software component's safety measure lies in it's accuracy.

- Accuracy: The main constraint of an autonomous driving system is its accuracy in replacing the human driving. This is measured by the detection quality and the planning execution. The involved algorithms have a direct impact on these qualities.

- Performance: In general, the more sophisticated the algorithm, the more computations it needs. Therefore, the accuracy constraint could be transformed into a performance constraint where the hardware platform should be capable of running the different algorithms and provide outputs in time.

- Real-time: Due to it's nature, real-time responses are needed. The system should also be able to reply in given intervals for a safe driving. The difference between real-time and performance is especially important in machine learning. Most hardware accelerators, Graphics Processing Units in particular, reach very high throughputs when processing multiple inputs at a time. In an ADS, the number of inputs per instance is limited by the sensors, for instance cameras, yet, each individual input needs to be processes in a fixed time for the decision to be taken.

- Energy: It is a generic constraint in embedded systems. However, it is not that important for autonomous vehicles. The maximum power consumption of an NVIDIA Titan X GPU is 334 W which results in a 0.334 kWh consumption. From the Electric Vehicle Database [51], the average energy consumption of fully electric cars is 18.6 kWh/100km. With a speed of 100 km/h, the gpu would be responsible for 0.02 % ($\frac{0.334kWh}{18.6kWh \times 1h \times \frac{100km/h}{100km}}$) of the total energy of the car which is negligible.

- Time-to-Market: Besides technical constraints, time-to-market is a major player for any embedded system. The race towards fully autonomous vehicles is rapidly intensifying. Most companies are joining in and standards are yet to be put.

- Cost: While luxury cars can afford to deploy high-end hardware with limited to no limitations on cost, everyday cars are more compelling. It was estimated by the Global Automobile Database [70] that Toyota Corolla is the most sold vehicle in 2019. With a cost of 18,000 € per unit, a 7000 € NVIDIA Quadro RTX 8000 or a 9000 € Xilinx Alveo U200 would be highly overpriced.

Balancing these constraints is key to choosing the right hardware platform. In the rest of the thesis, we work on the reliability and performance constraints of the machine learning component of the system. As seen earlier, ML is a must of any ADS.

[2] https://en.wikipedia.org/wiki/Trolley_problem

3.5 Simulators

Training more sophisticated algorithms relies partly on having rich datasets. A rich dataset should contain instances of any given situation. In the real world, it is very complicated to collect a crash dataset from the point of view of the driver[3]. This is where simulators come in handy.

Moreover, real-time systems such as ADS are best modelized as agents in environment with a reward-penalty approach. With the rise of reinforcement learning, datasets became less of a hassle and better environment simulators are of a priority. Table 3.3 from [217] resumes the most famous simulators.

Table 3.3: Comparison of autonomous vehicle simulators. The environment abbreviations for urban and off-road columns are: T represent town, C is for city, R is for road track and H for highway, F for forest, D for desert, M for mountains, G for grassy field, U for underground mine and H for harbor.

		Documentation					Environment				Sensors					Output Training Labels		
							For Driving		Actors		Cameras			Others				
	Licence	Installation	Environments	Sensor config	Output labels	Engine	Urban	Off-road	Humans	Cars	RGB	Depth	Thermal / IR	Lidar	Radar	Semantic Segmentation	2D bbox	3D bbox
CARLA	MIT	+	+	+	+	UE	T	-	+	+	+	+	-	+	-	+	+	+
AirSim	MIT	+	+	+	+	UE	T C	F M G	-	+	+	+	+	+	-	+	-*	-*
Deepdrive	MIT	+	-	+	-	UE	R	-	-	+	+	+	-	-	-	-*	-*	+
LGSVL	Propr	+	+	+	+	Unity	C	-	+	+	+	+	-	+	+	+	+	+
Sim4CV	Propr	+	-	-	-	UE	T	D G	+	+	+	+	-	-	-	+	+	+
SynCity	Propr	NA[45]	NA	NA	NA	Unity	T C H	D F G U M	+	+	+	+	+	+	+	+	+	+
Unikie	Propr	NA	NA	NA	NA	NA	C	U H	+	+	+	-	-	+	-	NA	NA	NA[46]
rFpro	Propr	NA	NA	NA	NA	NA	T C R	-	+	+	+	+	NA	+	+	+	NA	NA
Cognata	Propr	NA	NA	NA	NA	Custo	C	-	+	+	+	NA	NA	+	+	+	NA	NA
SCANeR Studio	Propr	NA	NA	NA	NA	NA	C R	-	NA	+	+	NA	NA	+	+	+	NA	NA
Highwai	Propr	NA	NA	NA	NA	Unity	C	-	+	+	+	+	+	+	NA	NA[47]	+	+
NVIDIA Drive	Propr	NA	NA	NA	NA	UE	C H	-	NA	NA	+	NA	NA	+	+	+	+	NA

As discussed earlier, some simulators are used to generate synthetic data that would help training driving models. Realistic simulators can even use generated data to train real-world vehicles, however, no similar cases were reported.

On a parallel note, the rise of the digital twin model to replicate and study the behaviour of a system motivates the development of accurate simulators. The collected data from the real world help diagnose and significantly improve the driving model.

3.6 Frameworks for Autonomous Driving

Many frameworks exists for ADS. An ADS framework is a hardware/software solution to the self-driving problem. The first detailed example was given by the winner car of the 2005 DARPA challenge, Stanley [27]. Recently, almost each major car producer has it's own version of an autonomous vehicle. In this section we give two example systems: Stanely and NVIDIA's DAVE-2.

3.6.1 Stanley

Stanley was designed for desert driving with no manual intervention (Level 5 SAE ADS). It relied on ML in its perception module. The architecture of Stanley can be seen in Figure 3.5.

Stanley is composed of 6 modules. First the sensor interface feeds the system with raw inputs from sensory units. These inputs are then processed by the perception

[3]This is due to ethical and legal obligations for real life crashes and cost if crashes were to be scened

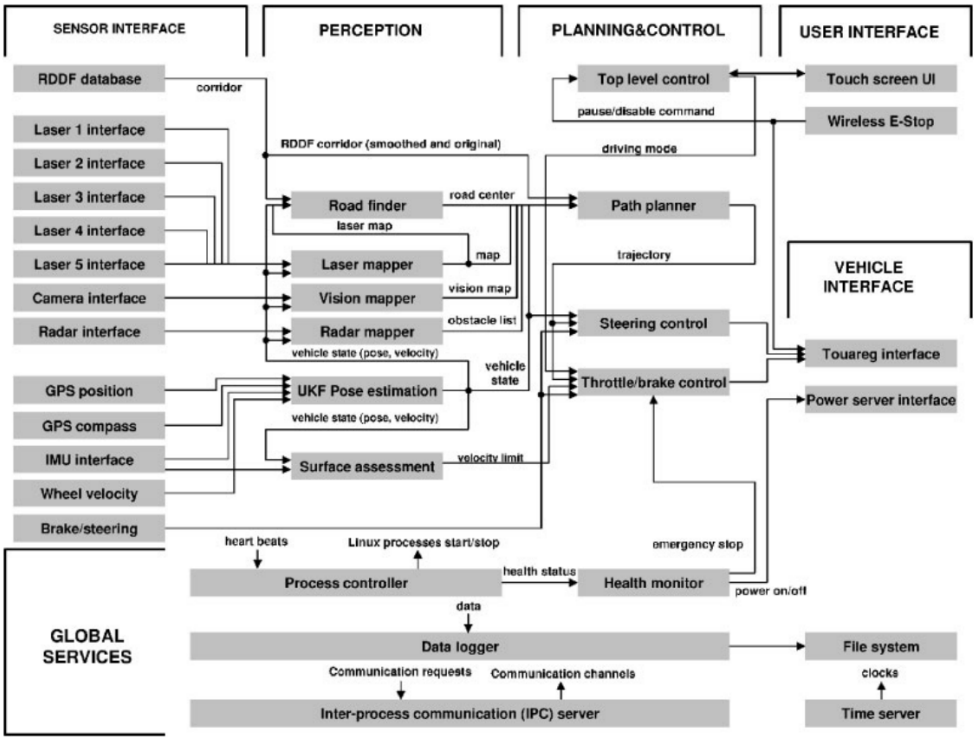


Figure 3.5: Flowchart of Stanley software system from [27].

module that would extract the navigable area, detect obstacles and assess the surface to estimate a velocity limit. The perception module is the heaviest in terms of compute needs. The outputs of the perception modules are passed to the vehicle via the planning and control module. These outputs are pre-processed and passed to the vehicle interface module to be executed as actions such as throttling down or steering the vehicle. The user interface module is used to show the state of the system and the actual outputs. Global services such as health monitoring and time syncing are packed in a separate module.

The huge popularity of autonomous vehicles led also to the development of software stacks similar to Stanley's. Apollo[4] and Autoware[5] are two examples. They are both open-source platforms that allows for easier development and integration of autonomous driving solutions. These platforms offers a pre-built Robotics Operation System (ROS) that implements the main components of an ADS. A designer will then have the choice on changing certain modules, such as replacing the obstacle detector. The missing component of these platforms is the hardware side where no material requirements are provided.

3.6.2 DAVE-2

In contrast to the previous *pipelined* solution. NVIDIA adopted an end-to-end solution in their DAVE-2 system [22]. They used a 9-layer CNN consisting of 5 convolutional layers, 3 fully connected layers and a normalization layer. The architecture is shown in Figure 3.6. The network takes a single image from a centered camera with a YUV encoding. The networks outputs the final steering command which is passed directly to the drive-by-wire interface.

The CNN is trained using real life recorded actions. A human driver drives the

[4] https://github.com/ApolloAuto/apollo
[5] https://github.com/autowarefoundation/autoware

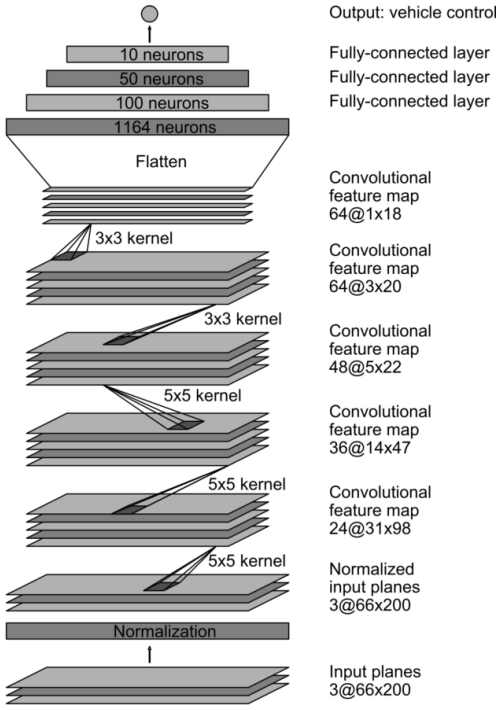


Figure 3.6: Architecture of the end-to-end system from NVIDIA's DAVE-2.

car for a certain period of time. During that time, the steering decisions are recorded along with the three camera inputs, a left camera, a right camera and a center camera. The camera images are passed to the network as inputs. The ground truth used to computed the error during training is the actual driver decision. This training architecture is shown in Figure 3.7.

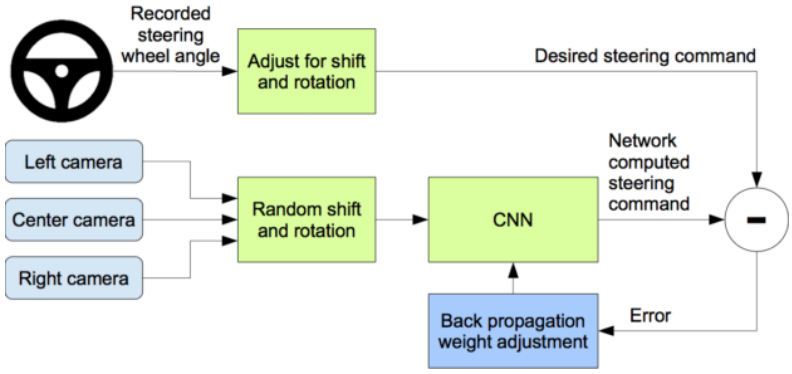


Figure 3.7: Training of the neural network used in NVIDIA's self-driving car.

With the constraints of autonomous driving, the hardware is as important as the software to guarantee safe driving. NVIDIA propose the Drive platform [172] to solve this problem for their system. They rely on GPUs in their system-on-a-chip that can ensures real-time execution of the software modules of DAVE-2.

3.7 ADS Design Example

Designing a self-driving system is a challenging task. In this section, we show our study on designing such systems. We only focus on the obstacle detection task. For

this task we present a system that would follow the data from the sensor to the final control outputs while picking an adequate hardware capable of handling the software load.

Each hardware platform presents strengths and weaknesses. Figure 3.8 from [262] gives a rough insight on how these platforms compare in terms of performance and energy consumption. More details on each platform are given in Section 4.

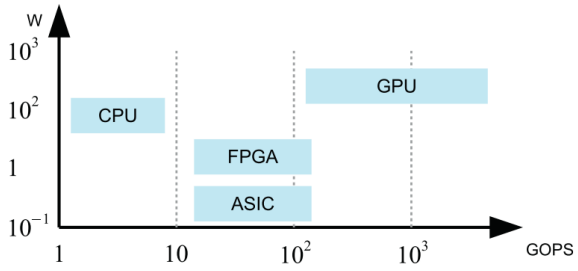


Figure 3.8: Comparison between different platforms.

For a system such as autonomous driving, a heterogeneous platform can be used to perform the various task while respecting the different constraints. In this chapter we show a simplified heterogeneous platform that we proposed as well as the full system optimization study that could be reproduced for other case studies.

3.7.1 Platform Design Example

In this section, we present a software/hardware co-design methodology to implement a full obstacle detection system based on deep learning. The results of this section were published in [162]. In [140], an ADS architecture is presented. First inputs are collected by sensors and sent to perception modules. These modules process the inputs and generate information to help decision modules control the vehicle. This pipeline is executed on a given platform. The sensory units vary in type. In autonomous driving systems LiDARs, RADARs, cameras and combinations of the above are used. In this work, we focus on images captured from cameras since they are the most dense and the require more complex computation to output a useful information. Results of this section were published in [162].

In a streaming environment, data comes from sensors and needs to be processed in real time. Due to the very complex nature of CNNs, such performance requires very delicate implementations. In the recent years, many researchers focused on this issue and proposed platforms in order to achieve real-time inference.

3.7.1.1 CPU

Since the CPU have a fixed architecture and a clear execution path, the only challenge is in the implementation. Many libraries were proposed to accelerate the inference by reducing the convolution problem to a matrix multiplication and using a fast algorithm for that such as Caffe [105]. However, the limited parallelism of a CPU can not be ignored and leaves the CPU-based implementations behind when compared to other platforms.

3.7.1.2 GPU

Similarly to CPUs, GPU have a fixed architecture. A GPU implementation is the software program that should be executed. Many CPU libraries have a GPU extension such as BLAS (cuBLAS). Dedicated platforms for DNN exists, most notably, Caffe and Google's Tensorflow [1].

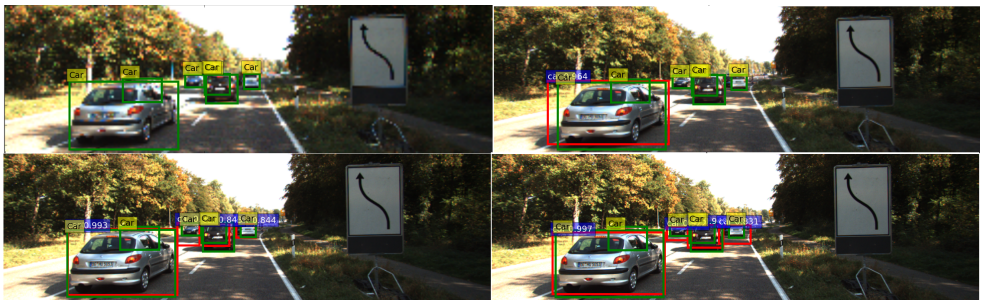


Figure 3.9: Evolution of detection when increasing the input resolution. From top-left to bottom-right: detection with 20%, 40%, 80% and 100% original size.

3.7.1.3 FPGA

In [143], authors combined the line-buffer storage policy with the winograd algorithm to obtain a staggering 2.94 TOPs. This performance was even compared to the top GPUs. The use of winograd algorithm gives a minimal number of multiplications is used during the convolution, however it is not portable to other networks where strides different than 1 are present.

As surveyed in [76], the top performance reached by an FPGA is 40.77 TOPs in [156]. This performance was reached by using 1-bit data and activation. In an FPGA, the multiplication operations are reduced to a simple gate logic operations (XNOR).

Other designs in [85] and [260], [258] also bypassed the 1 TOPs mark by employing a mixture of data quantization and data transfer techniques.

When compared to GPUs, the performance CNN implementations on FPGAs shines when data quantization is present. This is due to the fact that, in a GPU with 32-bit cores, a 1-bit operation and a 32-bit operation takes the same amount of time, resources and energy.

However, the main advantage of FPGAs over GPUs is energy efficiency (GOPs/Watt) where the same operation can be performed with minimal energy requirements.

The use of High-Level Synthesis (HLS) tools opened the doors for a variety of implementations. [15] bypassed the 1 TOPs mark on Arria 10 with 1.38 TOPs by using OpenCL to develop their accelerator. Xilinx also propose an implementation in [103] using their own HLS tool. This implementation supports quantization by using the Ristretto platform [80].

These implementations, and others that would be seen later on in Section 4, only focus on the accelerator without porting the design to a runtime environment. Having a high processing speed is a must for real-time systems, however, communication costs and memory bottlenecks are proved to be an even bigger problem than the computational load itself. Hence, we tackle this implementation challenge by integrating an already optimized CNN accelerator into a real-life scenario where images are streamed from a sensor.

3.7.2 Proposed Approach and Results

Our platform design operates in two phases. First, we applied some optimizations on the network by locating the optimal input size for our case study, obstacle detection for autonomous vehicles. Second, we considered the whole system where images are streamed from a camera through our pipeline.

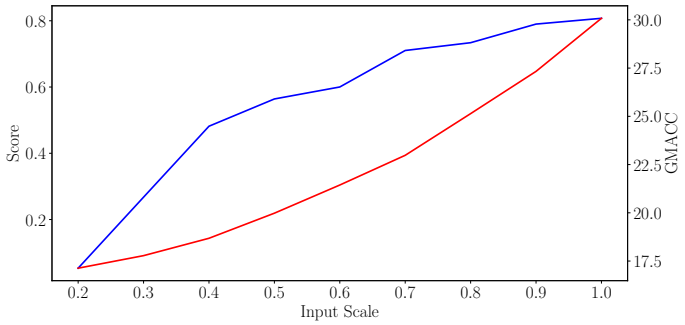


Figure 3.10: Impact of input size on network size (billion multiply-accumulates) and accuracy (compared to ground-truth.

3.7.2.1 Input size profiling

CNNs are known to be resilient to errors [165]. On the other hand, the same architecture can be retrained to process different sizes of images. Since filters are passed through the input size, it can generate an output for any input given. In this study, we vary the size of the input and measure the detection accuracy.

We used Faster R-CNN [191]. It is a state-of-the-art detection CNN in terms of accuracy. Many other detection algorithms are variants of Faster R-CNN and this study can be ported to most of them. The novelty in detection algorithms is that, not only the CNN generates a class for a given object, but it will also localize it and generate a bounding box. These networks are crucial for our task of obstacle detection for autonomous vehicles.

We tested Faster R-CNN on the KITTI [67] dataset. The novelty is in the input size. For the same image, we generate multiple low-resolution copies of the input ranging from 20% the original size to the actual image size (100%) with a 10% increment. We then process the new input using an already trained instance of Faster R-CNN. At the end, we record the accuracy of each copy. We use the F-score to compute the overlap with the ground truth detections which are provided by the KITTI dataset. We repeat this experimentation on multiple images of the KITTI dataset and we measure the mean accuracy of each input scale.

As for the computation load, we only considered multiply-accumulate (MACC) operations. This is the most compute-intensive task in a CNN [165]. In Faster R-CNN, it constitutes more than 99% of the overal operations. The number of MACC operations of a CNN can be accurately computed since the architecture is constant. A FC layer with 1000 neurons and 1000 input will always require a vector-matrix product of 1000×1000 by 1000 MACC operation.

In Figure 3.10 we show the evolution of the accuracy with respect to the computational load (Giga MACC operations). The number of operations grows exponentially when we increase the input image size. The accuracy however, does not follow the same growth. The maximum accuracy of detection is approached at 80% of the original size. Figure 3.9 presents the result of 4 different image scales and compares the CNN output (red boxes) to the ground truth (green boxes) for each run. The most critical vehicles for a safe driving are detected at 80% of the input scale which is in correlation with the graph in Figure 3.10.

This can be exploited to reduce the computational load while preserving an acceptable level of accuracy. By using the graph in Figure 3.10, we can even dynamically choose the perfect combination depending on the desired level of accuracy. In the next study, we use this information in order to accelerate a detection streaming platform.

3.7.2.2 Streaming platform

In an AV system, and many other similar situations, images come from a sensor as a stream. The processing unit needs to output a value for every given input in an acceptable time in order to take correct decisions. We simulate this behaviour in the design in Figure 3.11.

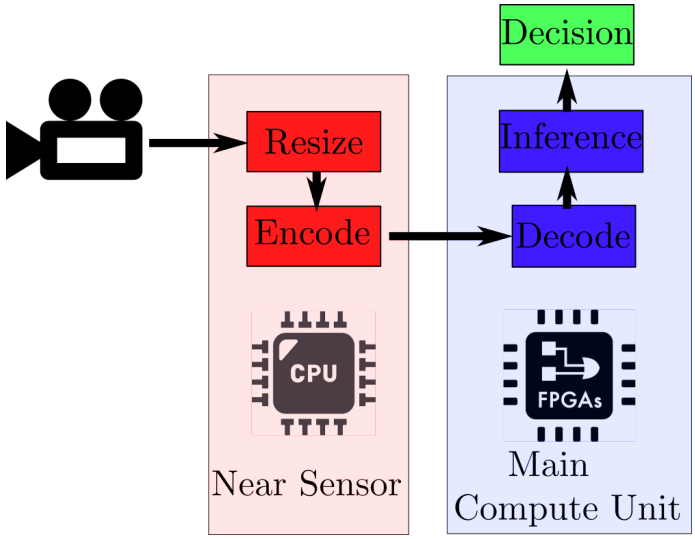


Figure 3.11: Architecture of the proposed encode/decode pipeline.

We propose a near-sensor computing model. These models allows high energy gains due to the reduced transfer costs. The drawback of near-sensor platforms is the limitations. The compute power and the energy requirements are very tight. For critical CNN applications with high accuracy requirements, a main compute unit needs to be in-place to handle the required compute power.

The two environments, the near-sensor unit and the main compute unit, needs to communicate. Even though they both reside in the vehicle, the communication cost is very expensive if we decide to send the raw input (image matrix). We propose the usage of encoding techniques to reduce the communication load.

The final design, as presented in Figure 3.11, takes an input from a sensor (camera). A near-sensor compute unit resizes the input image to the 300×300. This resolution is fixed using the previous study on the input size and guarantees a good accuracy with a relatively low compute requirements. This image is then encoded to reduce its size. The encoded information is sent to a main compute unit. Our main compute unit is an ZCU102 FPGA. The FPGA contains the latest Xilinx CNN accelerator [103]. Before running the inference, the received data is decoded and forwarded to the accelerator. Once finished, the result is sent to decision units.

We applied this architecture to the Xilinx design in [103]. We used the Ultrascale MPSoC ZCU102 platform. For input we used a laptop with a camera, the encoding is done using the OpenCV library. We first resize the input size to 300×300 and encoded the result into JPEG format. The FPGA is responsible on decoding the image and executing the inference. With this setup, we reached 14 FPS on Faster R-CNN. The Xilinx design already performs quantization on the network weights.

3.8 Summary

In this chapter, we introduced the architecture of an ADS. The software and hardware components of the architecture were presented in details. In the software component we enumerate the main tasks the system need to realise. The reliance on machine

learning algorithms was accentuated. Existing simulators and datasets tailored for the self-driving problem were presented. As for the hardware component, we give the constraints that justifies the pick of a platform over an other.

Some design examples were given, namely, Stanley from the DARPA challenge and DAVE from the NVIDIA team. These examples show how a full system is designed. We finally presented a novel design method to accelerate streaming and processing of real-life data. Our method is based on input size reduction and input compression in order to reduce network communications. We used JPEG for encoding and, although some losses are present, CNNs can tolerate these deformations and still outputs the correct results after inference. As for the hardware part, Near Sensor Processing is a well established paradigm in embedded design and needs to be employed whenever input data needs pre-processing.

Communication costs is a limiting factor when it comes to CNN performance due to the massive load of data involved. Extending this work by using In-Memory Computing is promising. The sensory unit only needs to store the data in a shared memory where computation also will take place.

Chapter 4

Hardware Platforms for Machine Learning

Contents

In this chapter, we give an overview of general and platform-independent CNN acceleration techniques in Section 4.1. Later on, we deeply discuss the different platforms with a brief architectural overview and previous works that focused on ML. We show how each platform make use of the general optimizations presented in Section 4.1 as well as its platform-specific optimizations.

For each platform, different design mindsets are present. If a given CPU only has a 32-bit Arithmetic Logic Unit (ALU), reducing the size of the operands would not be as efficient as if a custom ALU is designed in an FPGA. Based on this, we follow a different methodology in this section. The accelerators are classified first by the platform for which they are proposed and then, depending on the platform, the different optimizations are given.

4.1 CNN Optimization Techniques

A taxonomy of hardware accelerators for machine learning was proposed in [262]. We show this taxonomy in Figure 4.1. First, three levels of optimizations were distinguished: structure, algorithm and implementation level. At the structure level, only the data is considered. In a CNN, the data is the inputs, the activations and the weights. Using simpler representations such as the fixed point representation from [47] has a dramatic impact on performance with a manageable loss in accuracy. More aggressively, weights can be skipped which is the idea behind pruning techniques. The second level is algorithmic. For inference, authors considered other ways of computing the final result in order to simplify the execution graph. Fast Fourier Transform (FFT) transformation, Winograd algorithm and the matrix-multiplication transformation are very famous cases for convolutional layer algorithms. We see these transformation in more details later in Section 4.1.2. Lastly, at the architecture level, managing the hardware resources differently is a viable mean of trading performance, energy and resources in order to satisfy the environment constraints.

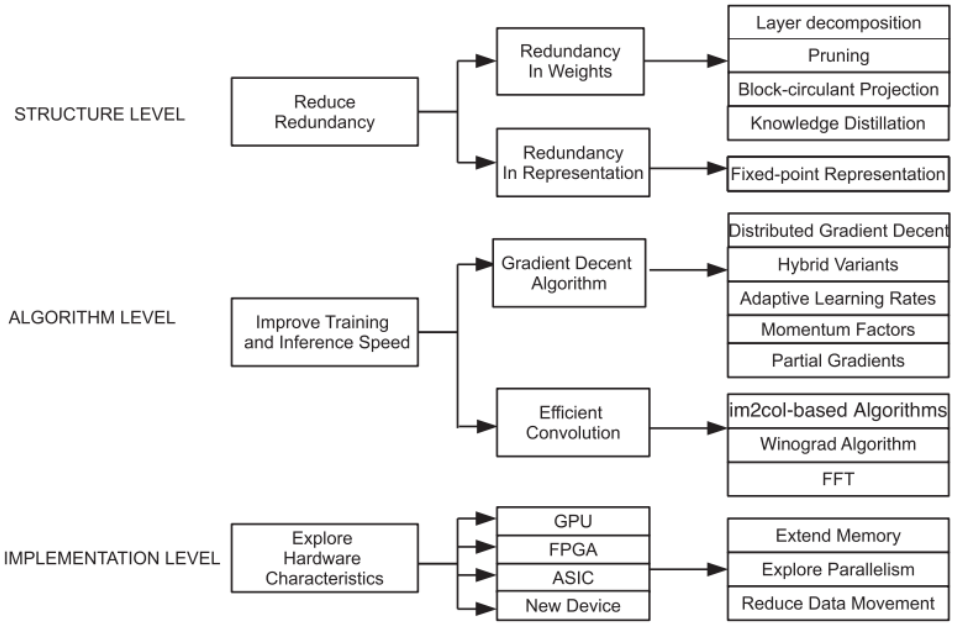


Figure 4.1: Taxonomy of CNN acceleration methods from [262].

In this section, we use this categorization. For this, we will focus in the next subsections on these three levels of classes. A general insight on how each class tend to optimize the inference time is given, later in this chapter, we show how each

platform (CPU, FPGA, GPU and Application Specific Integrated Circuit (ASIC)) exploit the optimization depending on it's architecture. All the previous works fall in one of these categories and is going to be presented later in this chapter. We followed a platform-aware classification to present the existing works on CNN acceleration. In the next two chapters (Chapter 5 and Chapter 6), we show how to exploit some of these techniques to design two hardware accelerators for machine learning inference.

4.1.1 Structure Level

These optimizations are performed at early stages and tend to reduce the complexity of the model before implementing it on hardware. Two main ideas are deeply explored in this field: 1) reducing complexity of data and operations known as approximate computing and 2) reducing the number of operations known as pruning. While these optimizations are performed on the model itself, the fact that they were motivated by hardware requirements justifies their status as hardware optimizations.

In reducing the complexity of data and operations, researchers reduced the number of bits required to represent the data (inputs, activations and weights). This is a common practice in approximate computing. A near-enough but fast result replaces the slow exact value which should not be a problem since CNNs are known to be resilient to small perturbations in inputs and outputs ; however this is not always true as we will see in Chapter 7. Besides the IEEE-754 floating point representation, two formats arises in reducing the number of bits: the static fixed point representation and the dynamic fixed-point representation such as the one in [47].

Reducing the number of bits, also known as **quantization**, can get as aggressive as it can gets. Binary and Ternary neural networks have shown the possibility of training a network with binary data. This not only reduces memory requirements with less bits but also the complexity of the operations. A multiplication in such networks is a mere AND operations which will further accelerates the network.

As seen in Figure 2.11, multiplication is the most common operation in a CNN. Reducing the complexity of this operation, even with a sacrifice on precision, can be extremely beneficial to the total performance. As far as we know, using less-complex operators is an under-explored field and was partly tackled in our second work in Section 6.

Reducing the complexity is not the only model optimization. Weights and feature maps contain redundant information. This is exploited to reduce the number of parameters and operations. Pruning uses optimization and exploration techniques to locate and eliminate, or at least minimize this redundancy.

Although this may leads to increase in fault rates as we will see in our reliability study in Chapter 7, many general-purpose CNNs use this optimization to lighten their models for fast response time and less energy consumption and resource usage.

4.1.2 Algorithmic Level

Most hardware optimizations target the software implementation before going into hardware. Machine Learning algorithms are no different. Simplifying the datapath in a CNN may lead to huge savings in execution time. The major challenge in implementing CNN algorithms, or ML algorithms in general, is optimizing the convolutional layer (*conv*) layer and the fully-connected (*fc*) layer. The *fc* layer can be boiled down to a vector-matrix multiplication where the input feature map is the vector and the weights and biases per neuron is the matrix. Vector-Matrix multiplication has predictable and an easily schedulable order of execution. This leads to efficient hardware implementation. As far as convolutions are concerned, transformations are not as straight forward. Executing the operations of this layer as-is is challenging. Three transformations are possible: convolutions as matrix multiplication, convolutions using winograd algorithms and convolutions in frequency domain.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \circledast \left(\begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix} \begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \end{bmatrix} \right) \Longleftrightarrow \begin{bmatrix} a_{11} & a_{12} & a_{21} & a_{22} \\ a_{12} & a_{13} & a_{22} & a_{23} \\ a_{21} & a_{22} & a_{31} & a_{32} \\ a_{22} & a_{23} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} w_{11}^1 & w_{11}^2 \\ w_{12}^1 & w_{12}^2 \\ w_{21}^1 & w_{21}^2 \\ w_{22}^1 & w_{22}^2 \end{bmatrix}$$

Figure 4.2: Transform of the convolution operations (left) to a GeMM (right).

4.1.2.1 Convolutions as GeMM

The idea is to re-organize multiply-add operations so that a single convolution between a filter and a window from the input feature map becomes a multiplication between a row vector and a column vector. The other convolutions are then stacked in order to obtain two matrices. The final product of these two matrices is none other than the original convolution. The motivation behind this transformation is the fact that General Matrix Multiplication (GeMM) can be rapidly optimized. The downside of this method is the replication of data; with small strides, input windows overlap with each other. When stacking them to form a matrix, the overlapping region is replicated in the two rows as can be seen in Figure 4.2. This transformation is adopted in Caffe [105], a very famous framework for machine learning. While architectures with abundant memory can live with this challenge, FPGAs, being mostly resource bound, are not very suitable for this transformation.

Our first work in Section 5 uses this observation. We accelerate matrix multiplication on a Resistive Associative Processor. We use a Content Addressable Memory (CAM) based on resistive devices. This memory is area-efficient, hence, can host the replicated data.

4.1.2.2 Winograd Convolutions

Since convolution and filtering are similar algorithms, many works tends to explore the Winograd algorithm, also referred to as the minimal filtering algorithm, to improve the performance of convolutional layers. The algorithm trades multiplications for additions to reach a minimum number of multiplications required to perform a given convolution between an input matrix and a filter. This number is reached by transforming the input and the filter to simplify the problem and reduce it to a general matrix-matrix multiplication (GeMM in Algorithm 1) and an element-wise matrix-matrix multiplication (EwMM in Algorithm 1). The final design contains less multiplications but more additions. However, since multiplications are more complex than additions, it outperforms the conventional algorithm by a factor of $3\times$ [128].

Input transformations consists of generating two intermediate matrices, U and V, as follows:

$$U = GWG^T \quad V = B^T IB \tag{4.1}$$

Where W is the weight matrix and G and B are constant matrices fixed in the algorithm. The output is then computed as follows:

$$O = A^T[U \odot V]A \tag{4.2}$$

Where A is an other constant matrix. The operator $\odot$ denotes element-wise multiplication.

Algorithm 1 shows the required transformations in the conventional algorithm to use the Winograd algorithm.

Algorithm 1 Convolutional layer using Winograd

procedure WINO(**Mat**: I, **Mat**: W, **Mat**: O)
$U = GeMM(G, W, G^T)$
$V = GeMM(B^T, I, B)$
$T = EwMM(U, V)$
$O = GeMM(A^T, T, A)$
end procedure

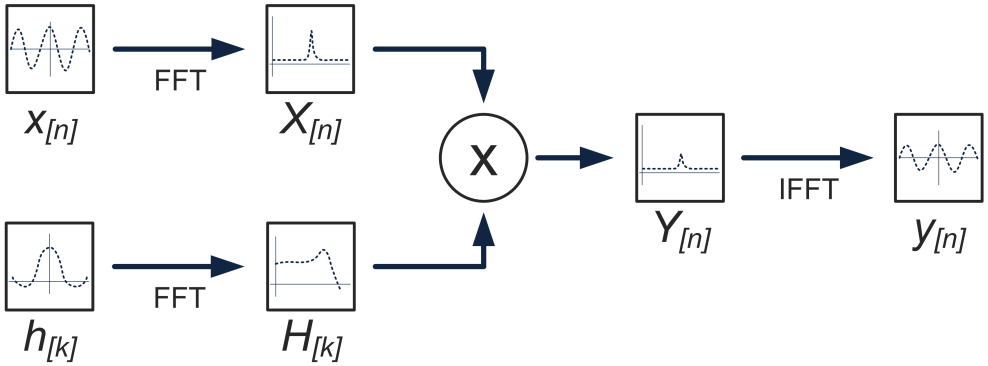


Figure 4.3: The FFT transformation of the convolution operation between an inputs X and a filter window h. The output y is obtained by applying an inverse FFT to the dot product of their FFT transformations.

The performance of the Winograd on FPGAs can be greatly improved. Input and weight transformations requires GeMM with the constant matrices B and G and their transposes. However, B and G only contains the values: 0, 1, -1, $\frac{1}{2}$, $-\frac{1}{2}$. This can be exploited to transform these multiplications to be done very efficiently using a simple MUX operation. This reduces the number of required multiplication even more to only the EwMM in Formula 4.2.

In [54], convolutions are handled using this algorithm. In this algorithm. Overlapped windows are exploited to reduce the number of computations.

4.1.2.3 Convolutions in Frequency Domain

The simplicity of computing convolutions in the frequency domain attracts many researchers to exploit it for their designs. The complicated operation of dot-multiplying a shifting window with a filter is a mere element-wise matrix multiplication in the frequency domain.

Let us consider the convolution in Figure 4.3 of an input X: a vector, resp. matrix, of n element to be convolved with h: a vector, resp. matrix, of k element representing the filter. The application of this transformation goes as follows. The first step is to compute the FFT of each vector. The outputs of the FFT operation are two vectors X and H. These vectors are the frequency domain representation of the input vectors x and h. The convolution theorem states that the convolution in the time domain equals point-wise multiplication in the frequency domain. Therefore, multiplying the values of X and H would result in Y the FFT of the expected result y. After this multiplication, an Inverse Fast Fourier Transform (IFFT) is applied to obtain y the final value of the operation.

The efficiency of this approaches shines for two reasons: 1) the fast FFT/IFFT calculation and 2) the unconstrained multiplications of the two matrices after transformation. As for the first, algorithms such as the butterfly diagram can be rapidly implemented and are easily paralellized. The output of such algorithms can be exploited in the second step as soon as it is computed. With this scheme, there would

be no delays and the resulting throughput of the overall circuit should be very high.

4.1.3 Implementation Level

Due to the differences in architectures, each hardware may, or may not, use a type of optimization. Using cache optimization techniques in a CPU is obsolete in an FPGA where the concept of cache is not present. Therefore, for implementation techniques, we present the platform independant optimizations, where the focus is on 1) choosing the best order in which the operations needs to be executed and 2) selecting the best structures that hosts the data for efficient manipulation.

A common practice in machine learning acceleration in FPGAs is automated hardware generation. It is a recurrent theme in many accelerators, therefore, we dedicate Section 4.1.3.2 to present the various possibilities of hardware generation.

4.1.3.1 Datapath optimization

Machine Learning algorithms are both compute and memory bound. Each run, billions of operations are performed and, over all the layers of one network, gigabytes of intermediate information need to be stored 2.2. A fully unrolled network is very tricky to deploy and almost no chip is capable of hosting it. In the second work we present in this thesis (Chapter 6), we study a novel method to tackle fully unrolled networks. The other alternative to solve this problem is reusing memory and Processing Elements which sacrifices performance for feasibility.

The optimal number and organization of processing elements that maximizes performance given a chip's configuration (resources, architecture, memory bandwidth, etc) is an optimization problem. For this, design-space exploration is largely used to find the most suitable accelerator for a given chip. High level tools such as HLS eases the burden of design space by proposing automated optimizations such as loop unrolling, pipelining and tilling. Add to that interchanging the loops during the software phase of designing, the vastness of the exploration space becomes within reach. In [257] authors tackled this design-space problem to figure out the most adequate parameters of the different loops that are required for a CNN execution. This allowed them to achieve state-of-the-art performance by the time their work was published.

Besides exploring PE duplication, Single Instruction-Multiple Data (SIMD) processors are also used. Since convolutions and fully connected layers could be replaced by matrix-matrix and matrix-vector multiplication respectively, it comes with no doubt that a vector processor is a promising candidate. Due to its efficiency, the architecture of SIMD processors has been adopted in many FPGA designs. A single PE is duplicated multiple times and data is forwarded as arrays (or matrices) back and forth between the PEs and the storage (BRAMs or off-chip memory).

While optimizing loops and exploring the design space proved to increase the throughput and the overall performance of the system, the best performance has been claimed by Google's TPU [117]. This model uses systolic arrays [126].

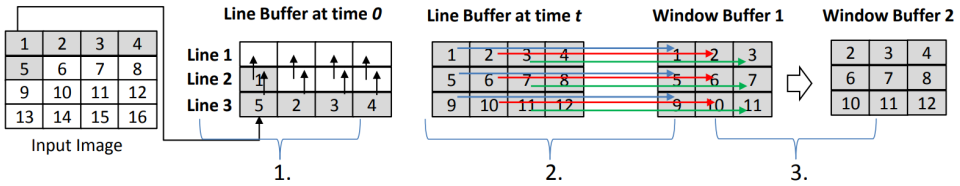


Figure 4.4: Line buffer and window buffer example for convolution operation from [149].

PE organization is not the only challenge. The way the data is stored, read and written is a task to deal with. One of the most efficient memory management patterns

is line buffers. This structure allows reading feature maps in the same order they are processed. The functioning of line buffers is shown in Figure 4.4. This structure was used in [143]. The idea is that a value will stay in memory as long as it is used; when this value is evicted, it will no longer be read again. This obviously reduces the required number of transfers between on- and off-chip memory which will reduce execution time.

4.1.3.2 Hardware Generation for FPGAs

Machine Learning algorithms in general and CNNs in particular are complex algorithms. Writing the hardware description of a full accelerator is challenging to say the least. The use of automated tools to ease this task is a common practice. While it is not considered an optimization on its own, this phase could be a deciding factor in obtaining good performing hardware.

Domain Specific Language (DSL) are common for various domains. CAPH[1] is a famous DSL for implementing dataflow applications on FPGAs. For ML, [268], [53] use these languages to propose hardware accelerators. Besides these two works and few others, the use of DSL is not frequent in CNN design.

The most common practices in CNN design are RTL and High-Level Synthesis (HLS) based generation. HLS solution are widely used due to the short time-to-market they offer. They come in two main categories, OpenCL based and Vivado HLS based[2]. While the former is platform independent, HLS still offers very handy set of tools that allows designers to implement their design in a very short time with minimum complications. Many researchers enjoy this simplicity to propose efficient implementations in terms of energy and performance such as [257], [224] and [220]. In the later, a framework (FINN) was built on top of Vivado HLS to facilitates generating hardware for any given architecture. Xilinx offers a set of tools on top of it's HLS such as the ViTIS framework that uses xDNN, a high level toolkit that ease the design of ML accelerators on its FPGAs. As for Altera's FPGAs, OpenCL is the main pick as a high level tool for designers. It was used in [210], [15] and [260].

The simplicity and high automation of HLS based approaches comes with a penalty on quality. For this reason, many hardware designers use hardware description languages to generate their code for better performance. An example of RTL based generated hardware is [158]. In this work, authors proposes a framework that generate a Verilog description of a given CNN architecture.

Later in this thesis, we will present two of our contributions to accelerating CNN inference. First, in Chapter 5, we explored an algorithmic optimization. As seen in Section 4.1.2, convolutions can be executed as matrix multiplication. We proposed a MM accelerator on an associative processor. The execution pattern of such algorithms on associative processor is faster. We exploit this fact to propose the architecture and the modified algorithm. In Chapter 6, we propose a second technique to accelerate CNNs. In this second proposition, we evaluated the approach on FPGA and on ASIC. In this rest of this chapter, we present the different works on accelerating machine learning inference on hardware.

4.2 CPU

CPUs are ubiquitous. Combined with the popularity of CNNs, many systems rely on their presence to satisfy their compute needs. Furthermore, complex systems requires an Operating System (OS) which, in general, are executed on a CPU. Hence, their deployment begets no additional resources since they are already present. This fact motivates most major companies, such as Intel [227] and Amazon [142], to exploit CPUs for CNN inference.

[1] https://github.com/jserot/caph

[2] Intel offers an HLS tool for their Altera FPGAs but it is not as common ax Xilinx' Vivado.

4.2.1 Architecture

Modern processors are composed of a number of cores[3], memory banks (in the form of cache levels), a memory controller and some input/output ports. Figure 4.5 shows the architecture of a first generation (Nehalem) Intel Core i7 [236].

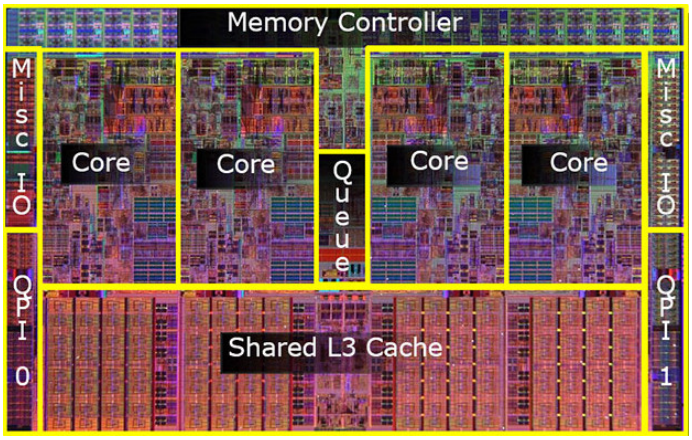


Figure 4.5: Architecture of Intel's processor Core-i7.

In general, the micro-architecture of each core contain some ALUs, a cache memory and a set of registers to control the execution and/or for storage. The micro-architecture of an Intel Core is shown in Figure 4.6.

In the literature the term Instruction Set Architecture (ISA) is used to refer to the set of atomic instructions a processor is capable of executing. The actual implementation of said architecture is called the micro-architecture. Two processors could have the same ISA but different micro-architectures such as AMD Opteron and the Intel Core i7. While the ISA decides what a processor can and can not do, the micro-architecture dictates how a processor will execute a program.

Due to high memory costs, multiple levels of caches are present inside a CPU. Ranging from the large and slow to the small and fast, the speed of a processor on a given task is highly decided by the efficient usage of these cache memories. When a core lacks a required data from the nearest cache, it has to look for this data in high level caches which will cause additional waiting times, hence, bad performance. This phenomena is refereed to as a *cache miss*.

In addition to the cache memory, the level of pipelining is an other deciding factor when performance is concerned. In order to be executed, an instruction is fetched from the instruction memory. It would then be decoded and passed to the correct execution module. At the end, the intermediate data is written back to free the registers of the execution cores. These four stages is a very simple example of a processor pipeline. This process could be accelerated by re-using subsequent modules for the next instruction after each step.

With all these optimizations, a single processor is still unable to efficiently execute compute intensive applications such as CNNs. In [92], five classes of CPUs were introduced; personal mobile devices, desktop, server, clusters and embedded CPUs. These classes could be seen in Table 4.1 ranked by their prices and performances.

Recent generations of processor support vector operations in the form of an extended instruction-set. This was motivated by the widespread of linear algebra applications. The most notable ones are the Streaming SIMD Extensions (SSE) and the Advanced Vector Extensions (AVX).

[3]When the number of cores is greater than 1, the processor is called a multi-core processor. If the emphasis is on parallelism and not on single-core performance, the processor is called a many-core processors; which is a special case of the former.

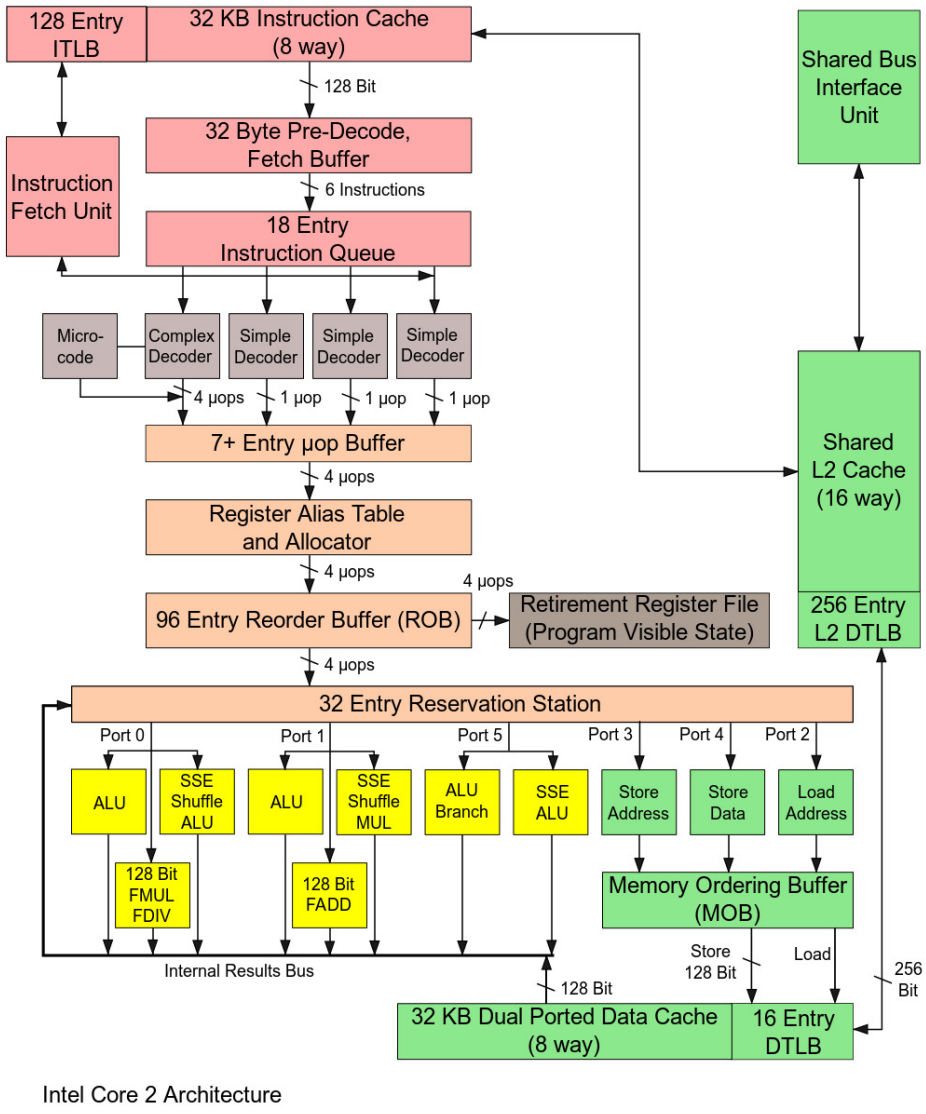


Figure 4.6: Intel Core micro-architecture.

Feature	Personal mobile device (PMD)	Desktop	Server	Clusters/warehouse-scale computer	Internet of things/ embedded
Price of system	$100–$1000	$300–$2500	$5000–$10,000,000	$100,000–$200,000,000	$10–$100,000
Price of microprocessor	$10–$100	$50–$500	$200–$2000	$50–$250	$0.01–$100
Critical system design issues	Cost, energy, media performance, responsiveness	Price-performance, energy, graphics performance	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality	Price, energy, application-specific performance

Table 4.1: CPU Classes from [92].

In our case, these classes could be reduced to only two. General purpose CPUs such as desktop, mobile and embedded CPUs and High Performance Computers present in servers.

4.2.2 CNN Acceleration on CPU

CPUs are designed for general-purpose computing. Acceleratorsfor ML applications on this platform comes in the form of libraries. Many of which are used in HPCs. These accelerations heavily rely on graph theory in order to optimize CNNs. Others possible optimizations focus on processor architecture such as reducing cache misses in [273].

4.2.2.1 Libraries

ML algorithms are dominated by vector and matrix operations. Libraries that focus on these operations are widely used to implement ANNs and CNNs algorithms. They share the same principle which is generating the execution graph and optimizing it for a given problem for better performance, memory usage, cache allocation efficiency and other important metrics.

Eigen [102] is a C++ linear algebra library that does exactly that. It's a template library that allows users to define their own types and use the different algorithms such as GeMM and Vector-Matrix Multiplication (VeMM). OpenBlAS [243] is a very similar library also for C++. It focus on Basic Linear Algebra Subroutines (BLAS). Its presence is wider than Eigen due to its better documentation. It provides basic linear algebra algorithms optimized for most common CPU architectures such as Digital Equipment Corporation (DEC) Alpha, Advanced RISC Machine (ARM) 32 and 64bit, Intel's Itanium architecture (IA64), Microprocessor without Interlocked Pipelined Stages (MIPS) 64-bit, Power Architecture, Oracle's Scalable Processor Architecture (SPARC) and Intel's x86 and x86-64 architectures.

The Automatically Tuned Linear Algebra Software (ATLAS) [235] is an other library that focus on BLAS subroutines. It focus on automating the process of writing efficient software for machines ranging from Personal Computer to embedded processors.

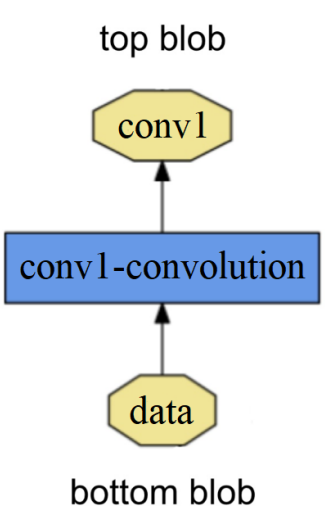
Intel developed its own library for its processors. Intel Math Kernel Library (IntelMKL) [227], in addition to linear algebra, focus on other math routines such as FFTs, vector statistics, data fitting and other miscellaneous solvers. It is highly optimized for most of Intel processors' families such as: the Xeon©, the Core™, the Atom©and the Xeon Phi™processor family.

While these libraries focus on BLAS algorithms which, then, can be used for ML algorithms, the Library targeting Intel Architecture for specialized dense and sparse matrix operations, and deep learning primitives (LIBXDMM) [90] focus on small matrix multiplications. This is very handy for CNNs since convolutions are majorly multiplications with small filters. It targets all recent x86 vector instruction set extensions up to Intel AVX-512.

The first major library dedicated to ML applications is Berkeley's Caffe [105]. It is written in C++ and makes use of other libraries such as ATLAS and OpenBLAS. It offers a full framework that, given a description of a network, gives the main routines to execute it. The description is written using Google's Protocol Buffers (ProtoBuf) by defining its own specific protocol named *prototxt*. Hence, the user only needs to provide the description of the network as in Figure 4.7 and, using the high level functions it provides, run most ML operations on the defined network, namely, training and inference. It has a support for GPUs.

Caffe provides most, if not all, ML operations. However, it is slightly complicated for experimenting. Google's Tensorflow [1] offers exactly that. Written in Python, it simplifies ML development by offering a very-high level Application Programming Interface (API). As for performance, the library generates a network graph that, once compiled, could be optimized and executed on the host hardware. It is based on tensors and is very optimized for TPUs which we will see later in Section 4.5. Other Python libraries exist ; the most notable ones are Theano [5], Keras [42][4] and

[4]Keras was integrated into the most recent version of Tensorflow (2.0).




```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1.0
    decay_mult: 1.0
  }
  param {
    lr_mult: 2.0
    decay_mult: 0.0
  }
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0.0
    }
  }
}
```

Figure 4.7: Example definition of network described in Caffe's *prototxt* format on the right. The equivalent architecture of the network is on the left. It has a single convolution layer where data is stored in *blobs*. A *blob* is Caffe's notation for a vector of elements.

Lasagne [55]. Their main goal is to ease the use of ML applications and facilitates experimentation to the detriment of performance.

MXNet [33] is a viable competitor to other major frameworks. It is an other deep learning library which automates ML design such as differentiation for gradient derivation and architecture declaration. In their experimentations, a similar performance to Caffe was reported while Tensorflow was lacking with 2× falloff.

4.2.2.2 Algorithmic and Graph-based Optimizations

The Winograd algorithm [239] is a very famous transformation for filtering operations. It is explained in details furthermore in Section 4.1.2.2. Basically, it trades multiplications for additions. It was used in [26] and [106]. In the two papers, techniques to extend the algorithm for larger filters with multiple dimensions were discussed. However, due to the nature of CPU architectures, multiplications and additions, the power of this algorithms is hard to harness and is best exploited in other platforms, especially FPGAs and ASICs.

A plethora of other algorithmic optimizations were presented in [68]. These algorithms focus on accelerating the conventional convolution algorithm, which they call *"direct convolutions"*, on x86 processor architecture. In their work they also discuss a strategy to efficiently deploy on multiple nodes. For ResNet-50 [88], a close to peak theoretical performance was reached.

An upper bound for execution time on multiple nodes can be computed using Brent's theorem.

Theorem 1 *Brent's theorem [78] states that if an algorithm with N operations can be executed with maximum parallelism, P_m, in T units of time then a processor with P cores ($P < P_m$) would be able to execute the same algorithm in T_P such that:*

$$T_P \leq T + \frac{N - T}{P} \tag{4.3}$$

In Figure 4.8 we show an example of an algorithm with $N = 11$ operations. The

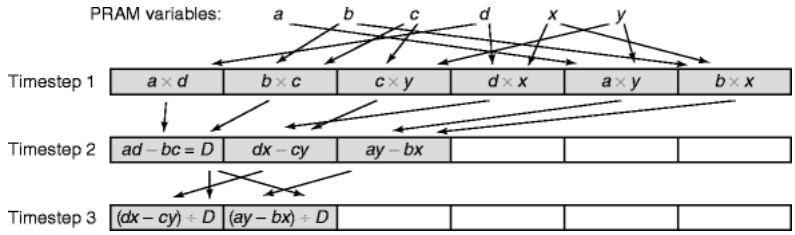


Figure 4.8: Parallel solution for a system of two equations with two unknowns. The considered processor has 6 cores. Each column represent one core and each row shows the instruction to be executed in the given core at a given time.

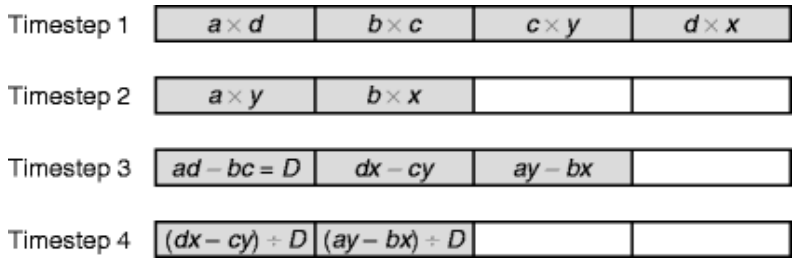


Figure 4.9: Solution to the system from 4.8 with 4 cores for u and v. The final values should be: $u = \frac{dx-cy}{ad-bc}$ and $v = \frac{ay-bx}{ad-bc}$.

algorithms solves a set of two equations:

$$\begin{cases} au + bv = x \\ cu + dv = y \end{cases}$$

The maximum number of cores required to fully unroll the algorithm is 6. With this many cores, the algorithm takes $T = 3$ timestamps to finish. The execution of the same algorithm on a 4-core processor is shown in Figure 4.9. As expected, it was possible to execute the algorithm in $T_P = 4$ timestamps which satisfies Equation 4.3.

While this theorem gives an upper bound on time, it does not shows how to obtain it. In [273] authors proposed a method to attain this performance. Compared to Caffe [105] and [5], they reached a speedup of over 90×. This implementation even compares to the GPU implementation of Caffe.

Low data reuse was identified in [261] as a root cause for performance issues in CPUs when executing RNNs. A framework that maximize data reuse was proposed. Authors claims an order of magnitude better performance compared to Tensorflow and can even compete with GPUs.

In [215], authors stated that, after generating the execution graph, the focus on maximizing the level of parallelism is more important than that of accelerating nodes in the graph. This was done in major frameworks where we see efficient implementations of individual operations such as most BLAS libraries without any concern of how the full application is going to be executed. This problem is more sever for many-core architecture. This was addressed in their paper. As a solution, they propose **Graphi**, an execution engine that minimizes interferences between software and hardware resources. Their results shows an acceleration range of 2.1× to 9.5× on a 68-core Intel Xeon Phi processor compared to Tensorflow.

Graph Lowering Compiler Techniques (GLOW) [192] is an other tool that uses the execution graph to optimize the ML code. It operates in two levels: a low level and a high level. The high level generates an optimized Intermediate Representation (IR) while the low level exploits the target architecture to perform memory-related optimizations. With this architecture, the generated code could be executed efficiently on any given architecture by parameterizing the low level optimizer. On a

single threaded Intel i7-7600U, it reaches 8.3 Frames Per Second (FPS) on ResNet-50 [88] compared to 3.0 for Tensorflow and 6.3 for TVM [34]. TVM is a similar graph optimizing framework that targets both CPUs and GPUs. It is based on Halide [187] which is a language that optimizes image processing tasks.

While some works proposes compilers to generate optimal code [37], the generation of an IR was the way to go. It was used in nGraph [50], an Intel deep learning compiler. With a fine-tuned low level enhancer, it achieves a 1.5× speedup when compared to their MKL library [227].

Built on top of TVM [34], Amazon's neoCPU [142] is an end-to-end stack which focus on optimizing both the execution graph and the operations without relying on other libraries. According to the authors, this gives more flexibility and allows better yield in terms of performance. Their results shows better execution time over Tensorflow and MXNet [33] on an 18-core Intel Skylake, a 24-core AMD EYPC and an ARM A72. The technique is also scalable and reaches a 3× speedup when the number of cores augments.

Edge devices have their share of ML ([240], [108]). Considered to be a must for many Internet of Things (IoT) systems, these devices have neither the space for large accelerators nor the performance of such high-end platforms. In [108] authors used the TVM [34] stack with specific edge optimizations such as quantization to obtain a 3× latency speedup over MXNet [33] for ResNet-18 and a 10× for MobileNet [94]. The runtime library they generated has a size of less than 1MB which is adequate for edge devices.

4.2.3 Summary

CPU implementations can be split into libraries and execution-graph optimization. Most major techniques falls in these two categories. However, when real-time responses are needed, CPUs are either sluggish or impractical if an HPC should be deployed. Their usage as a control unit is still inevitable. Their high simplicity, yet large coverability, makes of them the *de-facto* master in the majority of complex systems. Almost every platform has a CPU in it's core to manage the different accelerators and, in some cases, help with the execution.

4.3 GPU

GPUs are known to be the power-horse of machine learning. Many researchers considers this platform the sole responsible for the recent development of ML. In this section, we present the architecture of GPUs as well as the most notable CNN accelerations on them.

4.3.1 Architecture

In simple terms, a GPU is a many-core processor with different optimization objectives. The core ideas behind these optimizations were boiled down to three key concepts in [62]. Starting from a simple many-core processor, a GPU could be designed following these three rules.

- Simplify cores: A CPU core has many modules that helps a single instruction stream run faster. These modules could be an out-of-order control logic, a branch predictor, a memory pre-fetcher or cache memories. GPU cores are stripped down from all these components. By simplifying a single cores, more cores could be used to achieve higher parallelism. Basically, in this step we trade complex cores for more cores[5].

[5]This trend was also adapted in CPU design. The latest AMD EPYC 7742 has 64 cores which is unusual in common processors.

Figure 4.10: NVIDIA TU102 Diagram.

- Single-Instruction Multiple-Data (SIMD) processing: GPU cores are specialized for parallel computations. More ALUs are added to each core while simplifying their usage by introducing new instructions. The new instructions replace the old operations by vector operations[6]. The examples given in [62] is the NVIDIA GeForce GTX 285 with an 8 SIMD functional units per core and the AMD Radeon HD 4890 cores with 16 SIMD.

- Avoid stalls: In the first step, cores were simplified. Therefore, stalls caused by data dependencies, are no longer prevented since the responsible logic is simplified. Storing multiple contexts in each core can solve this problem. When facing a stall, the core will just load an other context without stalling the execution. This will increase the throughput of the GPU. The NVIDIA GeForce GTX 285 has a 64 KB of memory dedicated to context storage which can store up to 1024 contexts.

Figure 4.10 shows the architecture of an NVIDIA Turing, the TU102. It has 4086 cores spread over 72 Streaming Multiprocessor (SM). The architecture of an SM is shown in Figure 4.11. It is partitioned into 4 logical blocks sharing the same L1 cache. Each block has 16× 32-bit floating point ALU, 16× integer ALU and 32× 16-bit floating point ALU.

GPUs were originally designed for 2D rendering and 3D applications like video games. This was mainly due to the similarity between the architecture of GPUs and the data structure of these applications (independent matrices). CNNs have similar structures. A huge shift happened in the past few years; general purpose GPUs were

[6] A function that multiplies two 64-bit inputs using a basic instruction `mul` can be executed as 8 parallel 8-bit instances if the instruction set of the core has a vector multiplication instruction `mul8`. A vector operation takes vectors as inputs and perform the operation simultaneously on all cores.

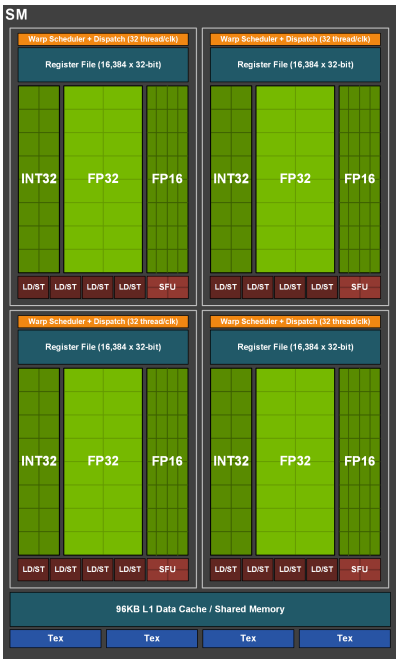


Figure 4.11: Architecture of the Streaming Multiprocessor (SM) in the NVIDIA TU102.

no longer limited to gaming and rendering but pioneered as ML training and inference platforms.

GPUs are not independent. A CPU is used to control the execution and forward the inputs[7].

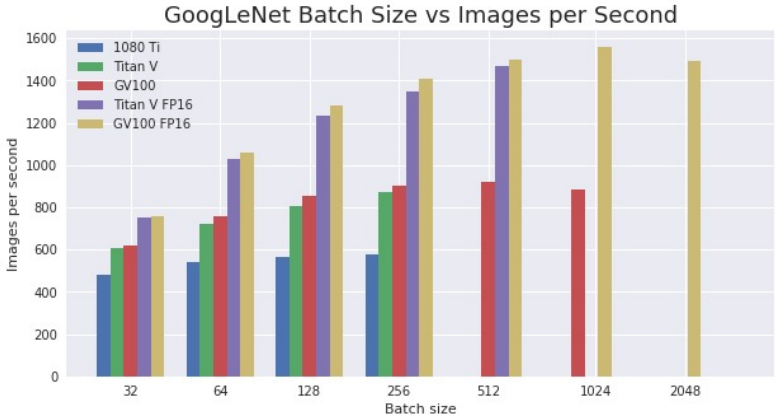
4.3.2 CNN Acceleration on GPU

Accelerating CNNs on a GPU is all about efficiently using the available cores. Restructuring the flow of execution may fix multiple memory problems caused by data dependency. However, the true power of GPUs lies in batching. In many cases, a CNN is asked to infer on multiple inputs. If multiple inputs are present at a given time, packing and processing them simultaneously will yield better performance since it eliminates the need of reloading weights for each input. In Figure 4.12, the impact of the batch size on the performance, measured as the number of processed images per second, is shown. Five GPUs from NVIDIA were considered: the 1080Ti, the Titan V, the GV100, the Titan V with 16-bit floating point and the GV100 with 16-bit floating point. Two networks are used for comparison, GoogleNet and ResNet. The trend is similar in the two networks as the batch size increases from 32 to 2048.

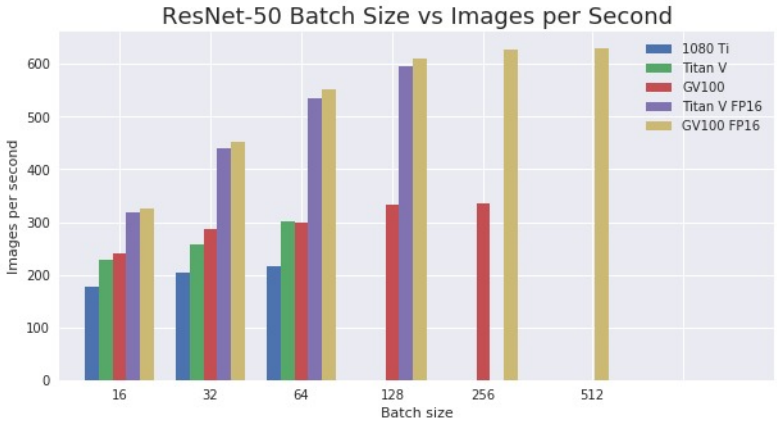
Increasing the batch size can be disadvantageous if used aggressively. As seen in Figure 4.12, the memory requirement increases when the number of simultaneous inputs increases. At a given point, the GPU can no longer satisfy this requirement and is unable to process the load. Even if the GPU is able to hold the massive batch, the number of images processed per second may decreases as seen in the difference between the number of processed images per second for a batch size of 1024 and 2048 for the two networks.

In many situations, few images are present at any given moment. An obstacle detector for a real-time driving system can not accelerate inference using batching since it needs images to be processes as soon as they are captured. However, this feat

[7]The inputs are not actually sent from the CPU but from the main memory. The CPU will just initiate the operation by sending the address from where the data should be fetched.



(a) GoogleNet



(b) ResNet

Figure 4.12: The impact of increasing the batch size on the performance measured as the number of processed images per second for CNN inference. The missing bars at high batch sizes represent a GPU failure due to its memory size.

is still important while in training. Massive amounts of data is present and needs to be forward/back propagated through the network. Using the same weights, these data is sent simultaneously to the chip achieving higher performance.

4.3.2.1 GPU Libraries

Similar to CPUs, many libraries were proposed for CNN programming on GPUs. The most notable one being Nvidia's CuDNN [12]. Built on top of CUDA, CuDNN provides most of the primitives needed by any ML algorithm. GRNN [93] is an other library that only focuses on RNNs. Authors first showed that poor data reuse, low on-chip resource utilization and the synchronization overhead are the major causes of weak RNN performance on GPUs. Their library addresses these problems by minimizing global memory accesses and reorganizing the data in order to balance on-chip resource usage.

In [232], DLVM, a deep learning compiler was proposed. This compiler is based on the famous LLVM [173]. It generates a linear algebra IR and target GPUs. DLVM supports major ML architectures and algorithms and is highly expressive.

Authors in [222] noticed that many high level libraries (TensorFlow, Torch/PyTorch, Caffe, MXNet, etc) does not support custom operators. If needed, integrating these operators in existent libraries and tool-chains requires high engineering costs and

usually, a massive hit in performance. Authors proposed Tensor Comprehensions, a mathematical language for expressing these operators as well as other deep learning primitives. A compiler is then used to convert this description to a CUDA kernel that will lately be executed by the GPU.

4.3.2.2 Desktop and Server-Class GPU

Server-class GPUs focus mainly on training. In [43] authors gives numbers of their Commodity Off-The-Shelf HPC. The COTS HPC was able to "*train 1 billion parameter networks on just 3 machines in a couple of days*". They show implementation details of deep learning algorithms in two steps. First they focus on optimizing the CUDA kernels to be executed in each GPU. Then, they give details about communication optimization in order to reduce latency.

Whether it is a server-class GPUs or a desktop General Prupose GPU (GPGPU), computation power is not an issue. The latest families can easily fit thousands of compute cores capable of running at very high frequencies. Nevertheless, feeding these cores with data is a major issue. In [99], a Content Addressable Memory (CAM) is used to store frequent patterns of computations. These patterns are then used to enhance data reuse. This CAM was integrated with an AMD southern Island GPU architecture to result in 68% energy savings and 40% speedup with a negligible loss in accuracy (2%).

A hardware/software co-designed system was proposed in [242]. The paper presents an en-to-end recognition system which authors called Deep Image. Deep Image uses highly parallel algorithms. The algorithm uses butterfly synchronization with a lazy update technique. The final design is highly scalable which is very important in GPU design.

Beside CNNs, RNNs acceleration on analytics servers was discussed in [170]. In this work, GPUs were compared to CPUs, FPGAs and ASICs. With large batch sizes, the performance of a server-class GPU shines. However, the power and computation efficiency of FPGAs is hard to beat.

4.3.2.3 Mobile and Edge GPU

The difference between mobile-class GPUs and server-class GPUs were discussed in [97]. These differences boils down to three main problems. Achieving shorter latency, relieving the burden of network connecting to the cloud and protecting user privacy are all gains that could be harvested by opting for an inference on edge devices [229]. Contrary to the previous class of GPUs, these devices have less resources to work with and more restrained energy constraints [151].

Natural Language Processing (NLP) is mostly used in Intelligent Personal Assistant (IPA) such as Apple's Siri and Microsoft's Cortana. Computation for these IPAs are usually performed in the cloud [114]. With previously mentioned reasons, [265] and [114] focused on accelerating Long Short-Term Memory (LSTM) networks on mobile devices. These networks are very successful in NLP which is widely used on such devices. They exhibit major inefficiency when it comes to data movement. Authors in [265] proposed inter- and intra-cell level optimizations to 1) parallelize LSTM sequential portions and 2) explore data locality between cells while trading accuracy by skipping computations and memory reads.

To lighten the burden of communication, Neurosurgeon was proposed in [114]. It is a lightweight scheduler that partition computations between cloud and edge devices. It achieves a 3.1× latency speedup on average with a staggering 40.7× energy consumption reduction on the edge device.

Besides scheduling operations and memory loads, an other way of dealing with performance issues is by carefully adjusting the batch size. In [207], authors managed to achieve the best user satisfaction by exploring this idea. A metric of this metric (user satisfaction) is also proposed in the same paper.

In [97] authors tackled the problem by addressing major performance problems in GPUs. Branch divergence [214] is one of the most common setbacks of performance. Authors proposed DeepSense, a GPU accelerator for CNNs by eliminating branch divergence and vectorizing memory.

From an other perspective, the use of IR -a common practice in CPU acceleration- is explored in [229]. This transition widen the range of target devices since the representation is generated in a platform-agnostic way. It encodes most computer vision primitives which will then be optimized in most integrated GPUs. When compared to vendor-provided libraries, the proposed end-to-end solution achieves 1.62× performance increase. It can also be scaled to server-class GPUs and is adopted in production Amazon Web Services (AWS).

4.3.3 Summary

Whether it is training or inference, GPUs have been established as the *de-facto* platform for executing CNN algorithms. Their highly parallel architecture, combined with the easy schedulability of CNNs result in best recorded performances.

To guarantee this performance, three ideas were presented: simplifying cores, using SIMD processing and avoiding stalls. Hardware designers and vendors incorporate these ideas in the newer generations GPUs.

The batching capability of CNNs is an other feat that allows GPUs to excel. Their SIMD design is very adequate to process multiple inputs at the same. The high presence of batching during training is a deciding factor and the reason why almost every training framework is based on GPUs.

The density of cores inside a GPU causes many heating problems. Cooling the device is a major concern for any kind of use. However, the major downside of GPUs is energy consumption. These power hungry devices could not meet the requirements for most embedded setups and makes them more adequate as cloud resources where energy is not as constrained.

4.4 FPGA

FPGAs have been around since the late 80s. They were aimed to fix the flexibility problem of ASICs. At the cost of few extra logic, the chip can be reprogrammed to perform different types of computations. Initially designed for prototyping, this reconfigurability feat was exploited in data-centers such as the AWS F1 instances which uses Xilinx Ultrascale+ VU9P cards. The flexibility can also be exploited for dynamic reconfiguration. In this case, the chip is dynamically reprogrammed in order to change the behaviour, fully or partly, of the deployed design to cope with a changing environment.

4.4.1 Architecture

Three main components are present in every FPGA: Configurable Logic Blocks, programmable interconnects and input/output ports.

- CLB: this is the fundamental piece of an FPGA. It is a logical block that can be programmed to implement any logical function. A CLB has input pins, output pins and, in most architectures, a small memory that can be used to output the desired output for a given input. An FPGA chip has thousands of CLBs[8]. A synthesis tool will transform the user program, written in C or in VHDL, into logical operations that can be performed using CLBs.

[8]The Xilinx KU15P for instance has 523000 CLB units used as Look-Up Tables.

- Programmable interconnects: the CLBs pins (input and output) are wired together with other CLBs through programmable connections. These connections allows higher level functions to be implemented. If a logical operation can not be performed in a single CLB, two or more CLBs can be connected together in order to implement it. These connections can be implemented as a simple transistor that would link two components if turned on and disconnect them otherwise. The user program will contain information on how each CLB is connected to the others. The FPGA re-programmability comes from these smart connections.
- Input/output ports: besides the logic done inside the CLBs, an FPGA is connected to the outside world through IO ports. Depending on the program, these IO ports are then connected to the logic in order to perform the desired task.

Although these are the main components, recent production FPGAs have many other modules. The goal is to achieve faster run-times. For example, MAC is a very common operation amongst FPGA applications, therefore, Digital Signal Processing (DSP) units were introduced to efficiently perform it. A DSP is a hardwired MAC unit that eliminates the trouble of designing a LUT-based multiplication, which is inefficient. Another example of additional modules is the Block Random Access Memory (BRAM). Memory is needed in any design and implementing large amounts of such circuits using CLBs could be inefficient in terms of time and resources.

In [77], it was stated that maximizing the usage of DSPs is a must to achieve higher throughput in FPGA. This is obvious since ASIC-like hard-coded circuits are usually faster and more efficient.

4.4.2 CNN acceleration on FPGA

CNN acceleration on FPGAs is a flourishing field. The designs on CPUs and GPUs are limited by the architecture and only focus on the execution graph. On the other hand, the architecture is fixed to a certain extent. These architectures are already designed for a given target metric; GPUs are designed for high parallel *simple* operations and CPUs thrive in energy consumption and can be easily deployed for any complex system. Whether it is performance, energy consumption, area or resource usage, FPGAs can be efficiently configured to handle any given constraints. Moreover, their flexible architecture allows many *exotic* enhancements such as the use of custom operators. A custom operator requires a custom circuit such as the one we used in our second contribution explained in more details in Chapter 6. These circuits are inefficient to execute on an ISA[9].

4.4.2.1 Surveys

Many surveys focused on CNN acceleration on FPGA. In [77], two categories of accelerators were proposed, hardware oriented model compression and efficient architectures for hardware design.

The first class of optimization includes quantization and weight reduction. Authors try to reduce the size and the complexity of the model in order to simplify the resulting logic. Quantization endures a loss in accuracy. In the survey, a comparison between the loss of different quantization approaches was presented. This comparison can be seen in Figure 4.13.

The second axis of optimization presented in [77] was at the architecture level where the user chooses the best implementation either by performing a transformation, such as the winograd algorithm, or by exploring the design space for the optimal

[9] An example of that is a bit-level operation that takes two inputs a and b and outputs a value c equal to b if at least two bits of a are equal to 1. In the other case, c takes the values zero. In an ISA, this is translated into multiple consecutive operations while a straightforward FPGA design can replicate this behaviour efficiently.

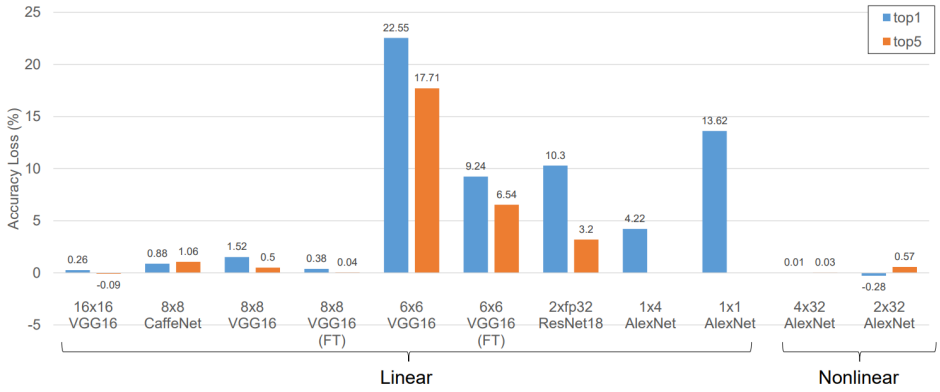


Figure 4.13: Comparison between different quantization methods from [75], [83], [131], [185], [270] and [271]. The quantization configuration is expressed as (weight bit-width) × (activation bit-width). The "(FT)" denotes that the network is fine-tuned after a linear quantization.

architecture. Efficient implementations were classified into optimizations that targets the computation unit design, the algorithm design and the system design. At computation unit level, the processing elements were optimized either by using low bit-width operators ([154], [73], [75], [82], [109], [96], [133], [156], [159], [160], [171], [182], [183], [185], [220], [244], [245], [256] and [267]) or by increasing the working frequency to reach the peak supported ([230] and [241]). At the algorithm level, the main loops of the convolutional or fully-connected layer are considered. researchers tend to fine-tune the level of parallelism of each loop in order to get the perfect design for a given chip ([257], [146], [157], [143], [96], [269], [264], [203] and [136]). At the same level, data transfers might be the key for better performance besides parallelism. This principle motivated the design of accelerators such as the usage of systolic arrays ([234], [146] and [15]) and line buffers ([185]). Finally, at the system level, high level models such as the roofline model [237] are used to compute the peak theoretical performance and evaluate the quality of designs.

A more detailed classification approach was adopted in [228]. First, quantization was presented as the category of hardware that uses simpler data representation in order to simplify logic. Approaches that reduce the number of weights by sharing or factorization are grouped into a second category. Their third category groups input dependant hardware. Finally, hardware that uses approximate computing to evaluate the activation function were discussed in a separate class.

The first two categories, quantization and weight reduction are similar to the first class from the previous categorization in [77]. The addition is the input-dependent computation class of implementations. In this category, two sub-classes arises based on the target of the optimization, the software, i.e. the algorithm or the hardware, i.e. the implementation. Algorithmic development of input-dependant accelerators focus on stochastic studies on the input in order to exploit redundancy ([19], [116], [193], [194], [8], [18], [138], [135] and [65]). From the hardware perspective, implementations are scarce, albeit, we can mention cascade networks ([122], [8] and [65]) which rely on a two-level implementation of a slow high-precision subnetwork and a fast low-precision main network. Depending on the confidence on a given input, the system decides to switch, or not, to the high-precision network if the confidence is low.

Besides model and algorithmic optimizations mentioned in the two previous categorizations, two other FPGA acceleration categories were extracted in [3], datapath optimization and hardware generation. The first consists on designing better data structures that could handle the data flow in a CNN. As for hardware generation,

many tools can be used to ease the deployment of CNNs on FPGAs. Hardware that used these high level generators were discussed in this category of the survey.

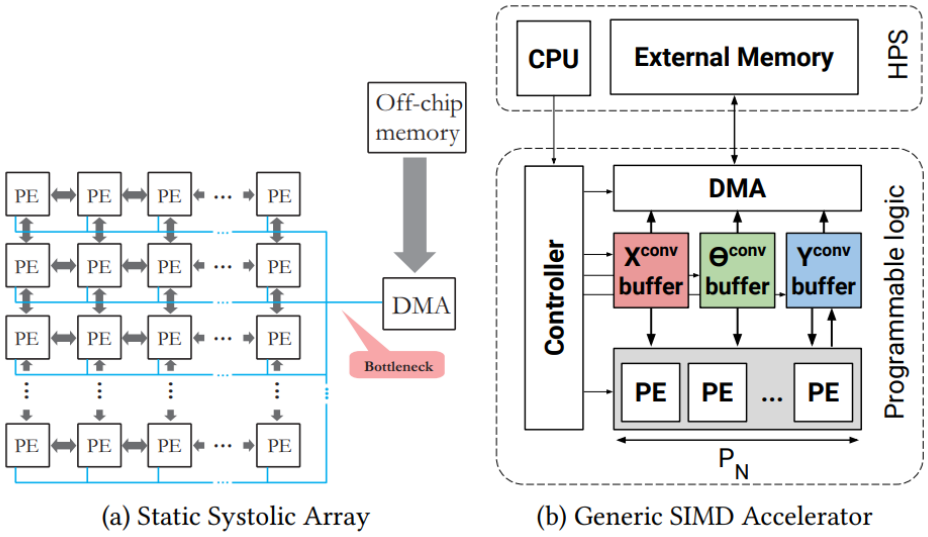


Figure 4.14: Generic data paths of FPGA-based CNN accelerators from [3].

In datapath optimization, we mention two sub-classes shown in Figure 4.14. Systolic arrays are grids of PEs connected to each other for faster communications. Their architecture is seen in more details in Section 4.5. They are more efficient as when implemented in ASIC but FPGA implementations exist ([197], [60], [31], [61] and [71]).

The second sub-class is SIMD accelerators. These accelerators employs a design space exploration to pick the best loop optimizations such as unrolling and tiling factors. These optimizations are the most common implementation methodologies and were explored in [257], [147], [258], [157], [210], [185], [96], [146], [188], [9] and [145] to design FPGA accelerators.

4.4.2.2 Architectures

Unlike CPUs and GPUs, designing on FPGA requires managing memory and compute units. While maximizing DSP usage is a requirement for performance [77], careful memory management is mandatory for feasibility. With a very limited on-chip memory, reading data from outside the FPGA each operation results in huge latencies that have dramatic impact on performance and energy consumption[10]. For this reason, the roofline model presented in [237] was used in many works such as [257] and [266]. In [257], authors explore all feasible solutions on a given platform and compare the efficiency to the maximum attainable performance. The design was generated using High Level Synthesis (HLS) and the optimal solution was picked by rearranging the different loops of the convolution algorithm (Algorithm 4). The global architecture for their accelerator as well as most standard FPGA accelerators is shown in Figure 4.15.

Memory was extensively studied for FPGAs. In [233], a layer conscious memory management framework was proposed for FPGA. The framework focus on maximizing data reuse and reducing prefetch operations. A similar study was performed in [179]. Authors designing a flexible memory hierarchy. Combined with a scheduler that uses loop tiling and reordering.

[10]It was shown in [113] that the memory is responsible for more than 80% energy consumption. This result is shown in Figure 5.1.

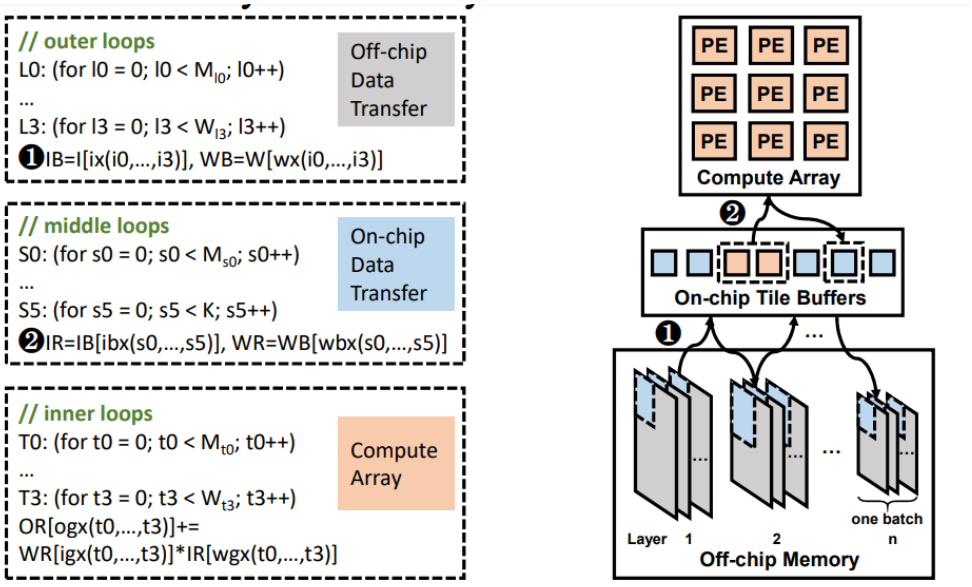


Figure 4.15: Standard dataflow of CNN/DNN accelerators.

In [143], line buffers were used to efficiently move data. For a CNN, this technique is probably the most efficient since each value will only be read once[11]. Two line buffers were considered, one for inputs and one for outputs. Data is read from the main memory and fed to the input line buffer. Input tiles are extracted from this line buffer and passed to the processing engines. These engines perform convolutions between the input tile and filters, which are stored in separate buffers residing in the on-chip memory (BRAMs). The output of the processing engines is stored to a output line buffer that is connected to the off-chip memory for final write-back. This architecture is shown in Figure 4.16. The input is an $W \times H$ matrix of M channels. This input is represented as an $M \times W$ by H matrix. The result of the convolution is an $N \times R \times C$ matrix where C is the number of filter, hence the number of output channels and N and R are the width and height of the output matrix depending on the stride and padding of the convolution. n and m are design parameters to be fine-tuned and represents the number of lines in the input and output line buffer respectively.

Processing elements used in [143] rely on the Winograd transformation [239]. This transformation was also used in [15] to reduce the number of multiplication for convolutions.

Systolic arrays is an other solution to the compute-heavy convolutional layers. They were used in [234], [197], [60], [31], [61] and [71]. They are designed for fast matrix multiplication. Therefore, inputs and weights are transformed to match this pattern. However, the performance gain can justify the cost of this transformation. The design from [234] reached 1.2 Tops for an 8-bit fixed point inference which is even comparable to GPU[12] with their immense power consumption.

4.4.2.3 Automatic Hardware Generators

CNNs have a straightforward architecture. Computations are generic and can be mapped to automatically generated hardware. Whether it is a matrix-multiplication transformation, an FFT transformation or the standard convolution algorithm, the

[11] If a value is brought to the line buffer from the main memory, it would be used as many times as the algorithm requires it. Once it is evicted, it would never be read again.

[12] The Titan X device delivers about 6 Tops speed during inference.

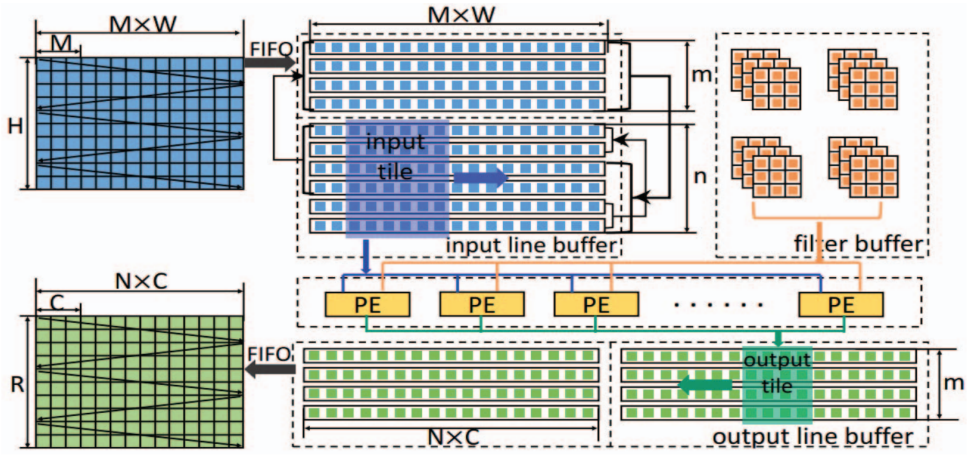


Figure 4.16: Architecture overview of the accelerator from [143].

loops can be parametrized, unrolled or split. These operations, and others, are automatically applied to generated a hardware. The tools can use a high level language for simplicity such as HLS or a lower level VHDL/Verilog implementation for performance. The full system is easily connected at the end since it is just a sequence of layers each of which having an input memory and an output memory.

	Publication	Workload	Throughput	Platform
FPGA	Haddoc2	3.8 Mop	318.48 Gop/s[1]	Cyclone V
		24 Mop	515.78 Gop/s[1]	Cyclone V
		24.8 Mop	437.30 Gop/s[1]	Zynq XC706
	fpgaConvNet	3.8 Mop	185.81 Gop/s[1]	Zynq XC706
		24.8 Mop	166.16 Gop/s[1]	Zynq XC706
	FINN	112.5 Mop	2500 Gop/s[1]	Zynq ZC706
GPU	CudNN R3	1333 Mop	6343 Gop/s	Titan X
ASIC	Yoda NN	24.8 Mop	525.4 Gop/s	UMC 65 nm
		23.4 Mop	454.4 Gop/s	UMC 65 nm
	NeuFlow	350 Mop	1280 Gop/s	IBM 45nm SOI

[1]Performance of the feature extractor

Table 4.2: Comparison of State-of-the-Art Hardware Generators from [3].

In [2], a tool, Haddoc2, was presented to automatically generate a hardware accelerator for a given CNN. FINN [220] is a similar tool the uses the Vivado design suite to produce the final IP. It was aimed to generate binarized network accelerators for faster inference on FPGAs. These generators become quite common recently for two main reasons: 1) CNNs have similar layered structures and atomic building blocks such as matrix-vector multiplication and 2) the loss, usually associated with the usage of high level generation tools, is minimal because the final product is but a replication of the building blocks; issues from place-and-route and scheduling does not apply. Table 4.2 shows a list of these generators as well as metrics on how they compare with other generators for other platforms such as Yoda NN [11], NeuFlow [61] for ASIC, CuDNN [12] for GPU and fpgaConvNet [224], an other FPGA hardware generator.

4.5 ASIC

If the target application is well defined, an ASIC is, naturally, the most efficient way of implementing a hardware accelerator for said application. In [59], authors proposed

an architecture that uses a streaming processor with a grid of connected ALUs. The Streaming processor forwards the inputs to the compute cores which will then writes the output to the external memory.

The efficiency of ASICs could be seen in [144]. In this work, authors present DaDianNao, a CNN accelerator with a 656× speedup over a gpu whilst reducing energy consumption by a factor of 184 using a 28 nm technology. The 28 nm technology was also used in [230]. In this later, authors achieved a throughput of 806 GOPs with less than 4 million gates and 352 KB of on-chip memory. Their design, called Chain-NN, consumes 567 mW which yield a 1.4 TOPs/W power efficiency. In [127], authors addressed the flexibility problem of ASICs by proposing MAERI, an accelerator build using modular and configurable building blocks that would host the network to be accelerated. The energy efficiency of MAERI was achieved by maximizing resource utilization using optimized mapping.

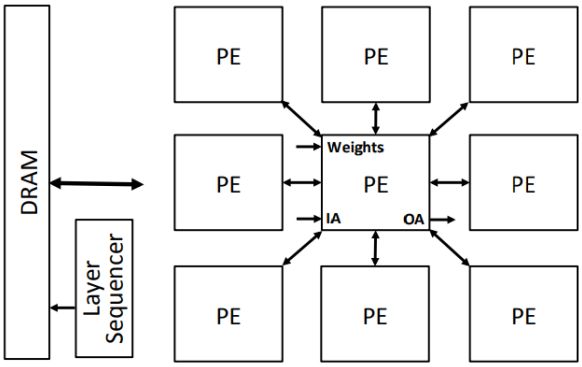


Figure 4.17: Architecture of the ASIC proposed in [175].

Scalability is an issue for many CNN accelerators. The results obtained on a network architecture are not guaranteed if a larger network is deployed. For ASIC, once the hardware is deployed, it is very complicated and costly to replace it. Being able to scale with the size of the network is a valuable asset to have in an ASIC accelerator. The problem of scalability in ASIC accelerators was tackled in [66]. PEs can be reused, however, memory should be large enough to host the target network. By using a 3D memory, the proposed accelerator, TETRIS, optimize the area distribution between PEs and SRAM buffers. The smaller footprint of 3D memories also decreases bandwidth pressure. Combined with a partitioning strategy, the accelerator achieves a 4.1× speedup with a 1.5× decrease in energy consumption over similar accelerators.

In [175], authors propose SCNN, a sparse CNN accelerator. The motivation behind SCNN is that most weights are zeros and zero multiplication could be skipped. Their architecture uses a grid of connected complex PEs. The grid is connected to an off-chip memory that contains network specification and weights as shown in Figure 4.17. Each PE has a coordinate-aware $F \times I$ array multiplier that skips zero-valued weights. When compared to a CNN accelerator, SCNN achieves 2.7× increase in performance with 2.3× drop in energy consumption.

The idea of a grid of connected PEs was also present in Eyeriss [36]. In addition, an efficient dataflow scheme was presented to maximize the use of these PEs while reducing required memory acceses by reusing the data locally. The dataflow, called Row Stationary (RS), is illustrated in Figure 4.18.

The non-reliance on memory by reusing PEs is best seen in their results. They recorded 0.0029 Dynamic Random-Access Memory (DRAM) accesses per each MAC for convolutional layers of AlexNet while requiring 278 mW of power.

YodaNN [11] takes the lead in power efficiency with their ultra-low power accelerator. The circuit was built using 65 nm technology at 0.6 V. The overall power

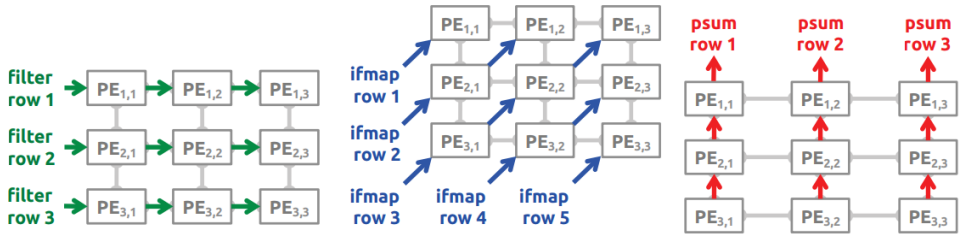


Figure 4.18: Dataflow of a single PE of Eyeriss, the accelerator from [36]. From left to right, in green is the flow of weights, in red the flow of input feature maps and finally in red the flow of partial sums.

consumption is 0.895 mW with an efficiency of 61.2 TOPs/W. With an area of less than 2 mm^2, YodaNN also leads in area efficiency. This was achieved by using binary networks where weights are limited to the values $+1$ and -1.

ASICs does not have a common architecture. Each work has its own design in order to best serve the specific application for which it is targeted. A very successful ASIC accelerator is Google's TPU. Many works are based on the same architecture for the efficiency it offers. For this, we give the architecture of these ASICs in a separate section as well as most notable works based on it.

4.5.1 TPU Architecture

The ubiquity of TPUs was instigated by Google's TPU [111]. The architecture of a TPU is shown in Figure 4.19. It relies on an external memory (a DDR3 in this case) to stream the inputs and weights. The instructions are also streamed from the host via a unified buffer. In order to match the computations of the CNN to the hardware architecture of the compute core, a systolic data setup is put in place to arrange the inputs. Beside this re-arrangement, a TPU architecture is similar to a Floating Point Unit (FPU) co-processor.

In the heart of every TPU, a systolic array is pumping. A systolic array is a grid arrangement of PEs as can be seen in Figure 4.20. This structure allows high throughput and can deal very efficiently with compute bound operations. Each PE perform a multiply-and-accumulate on the inputs and forward its output the the adjacent PEs. Originally, the PE would write-back the output to a register as seen on the left part of Figure 4.21. The next cycle, the same value needs to be read from the memory. The interconnection in the right part of Figure 4.21 shortcuts these "write-back" and "read" phases.

Google's TPU feature a 65,536 8-bit matrix multiplication units with a peak throughput of 92 TOPs. This throughput was reached by connecting ALUs together. In a CPU, time and energy is wasted when accessing registers after each operation. A systolic array is designed to fix this problem. The output of one ALU is passed to the next one as we can see in Figure 4.21. An ALU with such interconnection is usually referred to as a processing engine [211], we do not adopt the same appellation in this thesis. The connections between the ALUs matches the execution tree of a GeMM.

By matching the execution pattern, inputs could be streamed once throughout the systolic array. The output is written once all computations are done with no intermediate write/read operations.

The compute-heavy layers in a CNN are the convolutional layer and the fully-connected layer. Both these layers could be transformed to a matrix-matrix or a matrix-vector multiplication respectively. The systolic array is excellent at both. First, the weights are fed into the ALUs from the weight FIFO. The data setup module from Figure 4.19 is responsible for shaping the data into the correct matrix

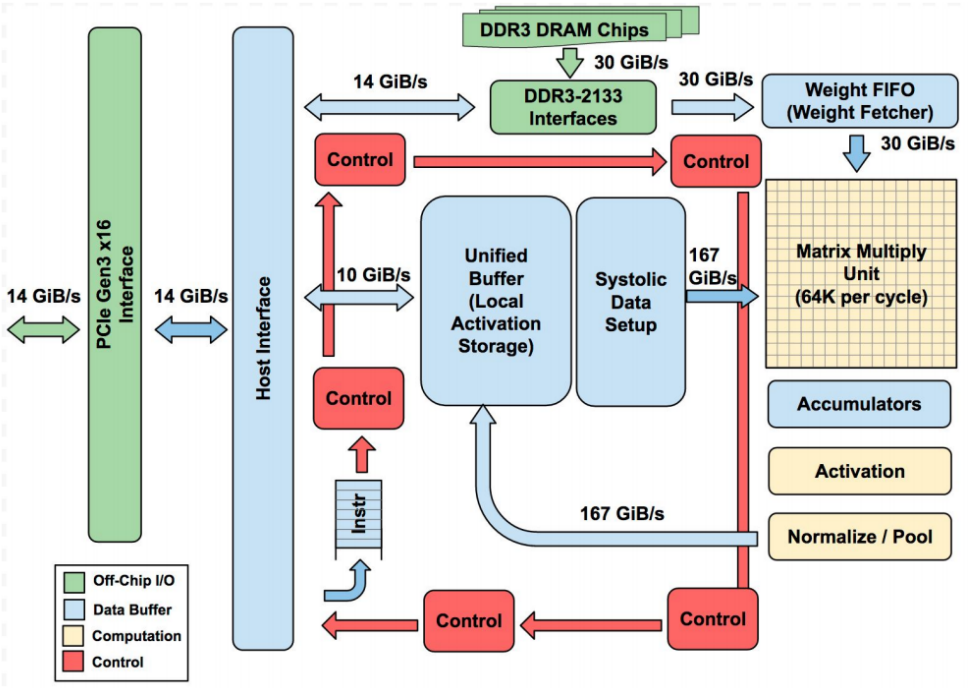


Figure 4.19: Architecture of the TPU from [111].

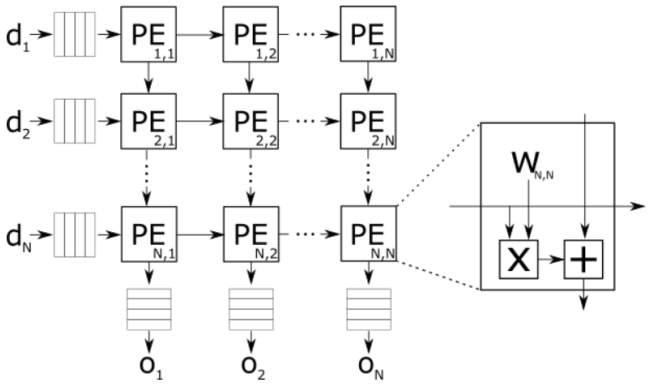


Figure 4.20: Architecture of a Systolic Array

operation. The flow of inputs is shown is Figure 4.22. The final outputs are stored in the accumulators, which are also responsible for summing partial outputs. If an activation or a pooling layer follows the currently executed layer, the two subsequent modules performs the required operations, element per element, on the systolic array output. Finally, the outputs are stored back to the unified buffer.

TPUs are initially designed for data-centers. The main concern in these environments is efficiency and TPUs are, by far, the most energy efficient platforms for CNNs. Google's TPU was compared to a server-class Intel Haswell CPU and an NVIDIA K80 GPU. The TPU was at least 30× more energy efficient[13] than the NVIDIA GPU and 80× more than the Intel CPU. Performance wise, the TPU was from 15× to 30× faster than the two other platforms. These results were obtained on many kinds of ML applications such as CNNs, Multilayer Perceptrons and LSTMs.

[13]Energy efficiency is measured in TOPs/Watt.

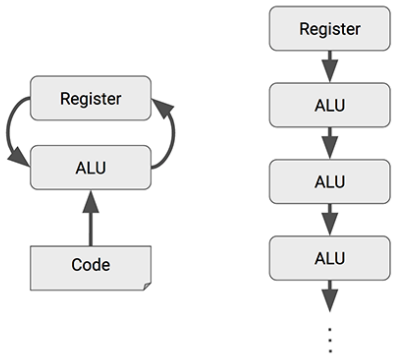


Figure 4.21: The simplified architecture of a CPU core on the left. The connections between ALUs in a TPU is on the right.

The TPU's efficiency depends on the memory bandwidth. If a GDDR5 memory is used in the TPU, the previous numbers would jump to 70× more efficiency against a GPU and 200× agasint a CPU.

Google's TPU was designed to accelerate TensorFlow algorithms by providing primitives that directly map TensorFlow code into the TPU. This is an incredible feat since TensorFlow is a very high level library resulting in shorter time-to-market and easier deployment of new CNN architectures.

4.5.2 TPU-based Accelerators

Based on [111], Stanford's TPU, named ConvAU, was proposed in [118]. The architecture of this TPU is shown in Figure 4.23. ConvAU achieves a 200× more TOPs/W when compared to an NVIDIA K80 GPU and a 1.9× compared to Google's TPU from [111].

A cycle-accurate simulator, Scale-Sim, for systolic arrays was proposed in [196]. it takes the CNN topology and the available hardware configuration as inputs. The hardware configuration consists of the size of memories allocated for inputs, outputs and weights and the dimension of the systolic array (width × height). It generates the expected number of cycles required to process the input architecture as well as other statistics such as the utilization efficiency and the reads/writes traces.

4.5.3 Summary

ASIC-based accelerators dominate other platforms in power efficiency. This was partly achieved by connecting PEs with each other; the alternative being relying on on-/off-chip memory to store intermediate results. This design methodology was adopted in every work in we mentioned in this section.

The, arguably only, downside of ASICs is flexibility. Once a circuit is printed, reusing the element is not a viable option. However with CNNs, this is not an obstacle. Designing a general CNN accelerator on ASIC could process any type of data just by loading the correct network architecture and weights to the memory. This is enabled by the uniqueness of CNNs as feature extractors.

4.6 General Conclusion

In this chapter, we surveyed the existing acceleration methods of machine learning algorithms, namely, CNNs. We classified the accelerators by platform. In Figure 4.24,

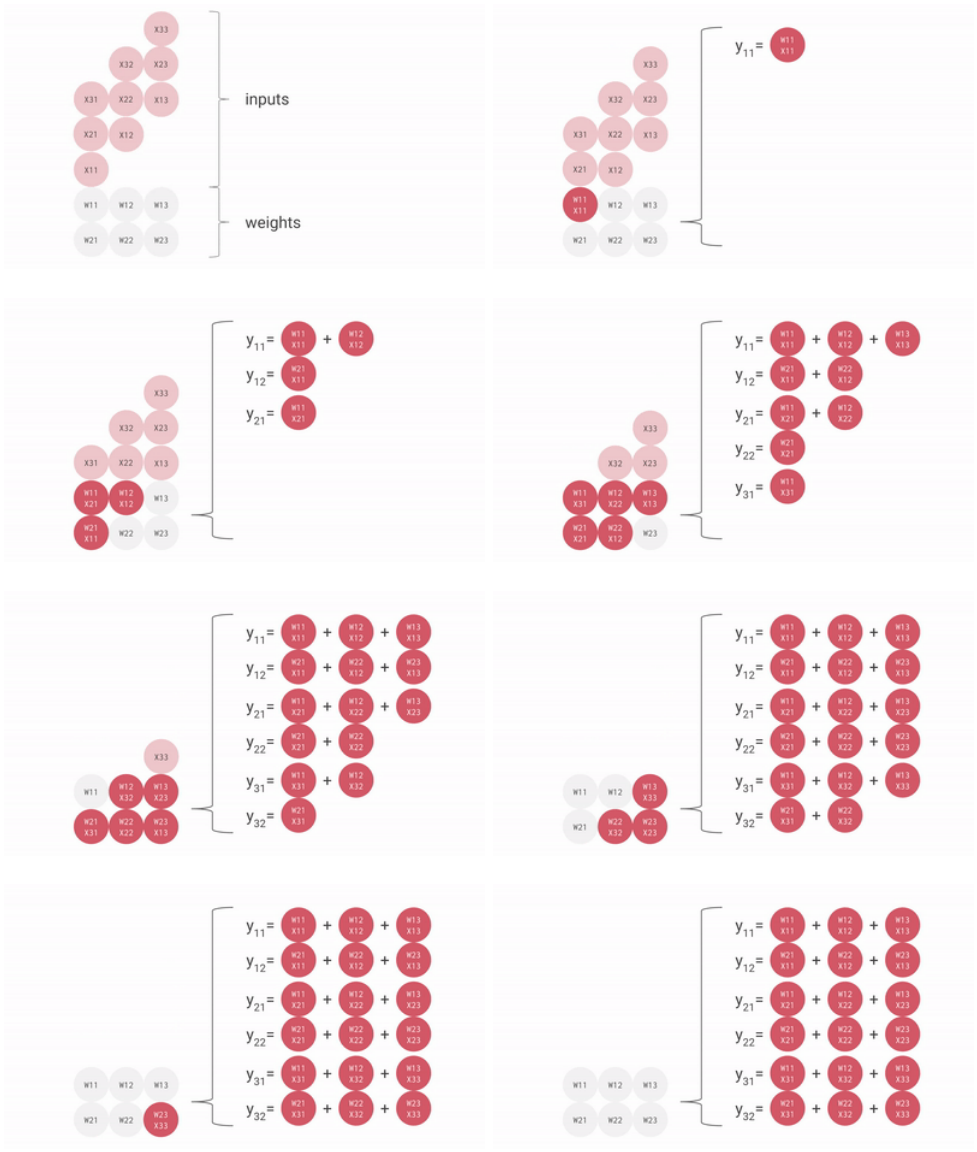


Figure 4.22: From top left to bottom right, the execution flow of a 3×3 input feature map and a 3×2 weight matrix. Weights are passed to the execution units colored in gray. The input, in light red, is then streamed through the systolic array. The computed outputs is shown on the right of each figure.

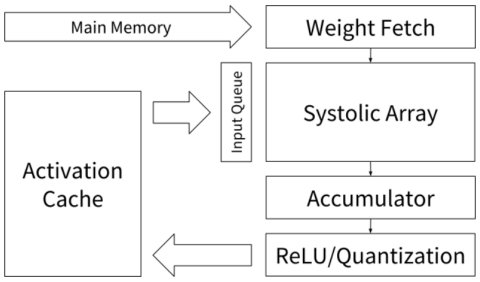


Figure 4.23: Overall architecture of the accelerator from [118].

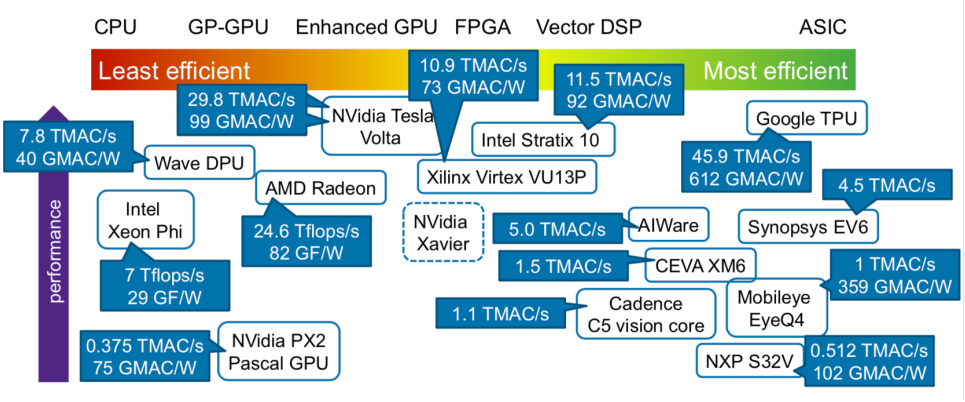


Figure 4.24: Comparison from [137] of various hardware accelerators in terms of energy efficiency, measured as GMACs per Watts, and performance, measured as TMACs/Tflops per second.

we summarize the most notable of these accelerations in an energy *vs* performance comparison.

While CPUs are the most ubiquitous, their performance is questionable. It is possible to deploy CPUs for IoTs, however, for an ADS, it is not possible to rely on their performance to execute the ML component. A solid proposition to the performance issue is the usage of GPUs. In terms of sheer performance, only TPUs can compare, which are usually cloud components. The downside however is that this performance comes from batching. In an autonomous vehicle scenario, the latency is more important than the throughput. High end GPUs, such as the NVIDIA Tesla Volta can satisfy both components, but their cost is immense relative to the price of the car. Moreover, their energy inefficiency is problematic, albeit, tolerable for non-electric cars. As for FPGAs, they can compete with other platforms in terms of performance, however, the peak performances are usually the result of a major tradeoff in accuracy, which is not tolerable for autonomous vehicles.

Chapter 5

Design for Performance: On Associative Processors

Contents

Memory access latency is one of the main problems that prevent processors from achieving high performance. To eliminate the need of loading/storing large sets of data, the In-Memory Computing (IMC) (or associative processor) architecture has been proposed as a solution to the von Neumann bottleneck. In IMC, logic and memory structures are combined together to allow in-memory computations. In this chapter, we propose a new algorithm to compute the matrix multiplication inside the memory that exploits the benefits of IMC. Many algorithms can be boiled down to a GeMM problem, notably, CNNs which are heavily present in the autonomous driving field. In this chapter, we will focus on one of the most efficient implementations of IMC architectures, namely the ReAP. ReAP is a processor architecture made possible with the invention of memristors. In [123], it was stated that memristors would become the future of AI. The proposed approach is based on the Cannon algorithm and uses a series of rotations without duplicating the data. It runs in $\mathcal{O}(n)$, where n is the dimension of the matrix. The method also applies to a large set of row by column matrix-based applications. Experimental results show several orders of magnitude increase in performance and reduction in energy and area when compared to the latest FPGA and CPU implementations. Work in this chapter has been partially presented in [163].

5.1 Motivations

With the recent decline in Moore's law, parallelism becomes the only resort for the increasing demand of computing power. With multiple compute cores, the memory needs to be faster than ever to provide the required data in time without augmenting the speed. This difference in speed between the memory and the computing unit is called the von Neumann Bottleneck or the memory wall.

Many applications suffer from this architectural limitation. Matrix multiplication (MM) is one of the most important building blocks in many applications like machine learning (as shown in Sections 2.3.3 and 2.3.2), image processing, etc. Although the logic of this operation is very simple (multiply and accumulate), the data needs to be massively transferred between the compute cores and memory.

IMC is a promising solution to this bottleneck. It consists of combining memory and logic in the same physical core. There is no need to transfer data back and forth since the computations are done in the same location where data is stored. Associative Processors (AP) are an IMC model of the conventional Single Instruction Multiple Data (SIMD) processor. Values that need to be processed are stored inside the processor cells, and an operation is performed on all these cells at the same time. Hence, associative processors do not need a separate memory module to store the data.

In this chapter, we use IMC to solve the memory bottleneck for matrix multiplication. The resistive implementation of associative processors (ReAP) is used for its superior density. We propose a vectorized implementation of Cannon's algorithm [30] that suits the ReAP architecture. We also generalized the algorithm for other applications like All-Pairs Shortest Paths (APSP). Our approach is able to reduce the execution time by an order of magnitude when compared to one of the most efficient FPGA implementations for large size matrices. Area consumption reduction is even more important.

The algorithm we present consists of storing elements inside the ReAP for efficient execution. We also discuss an architecture that helps moving the data inside the ReAP. This technique offers a very efficient data storage compared to similar methods. Existing approaches replicate values to perform multiple operations on the same value at the same time. In our approach, the values are kept uniquely inside the ReAP without any duplication which reduces area consumption. The approach we propose can be generalized to a set of matrix applications that are executed in a row per column format. Our approach is able to reduce the execution time by an order of

magnitude when compared to one of the most efficient FPGA implementations for large size matrices. Area consumption reduction is even more interesting.

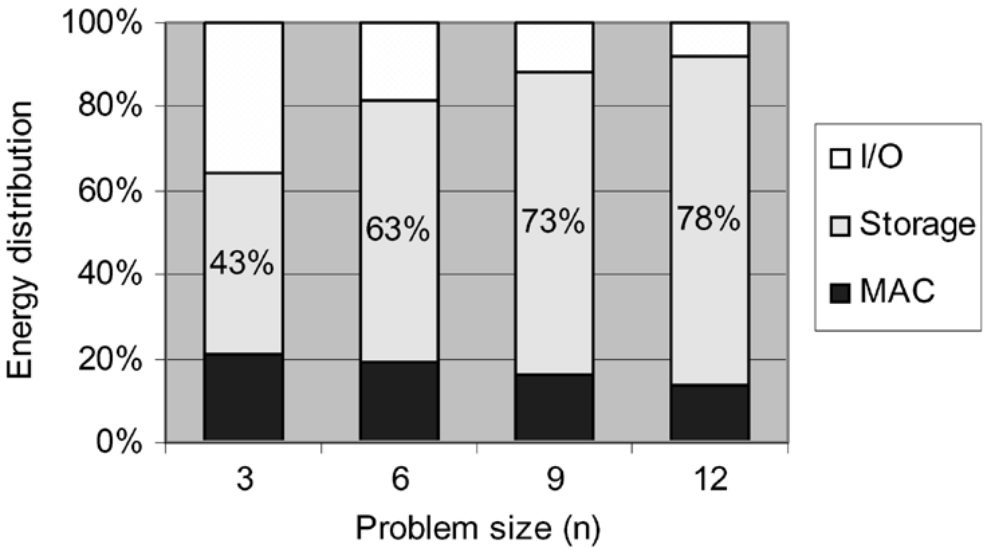


Figure 5.1: Energy distribution of a matrix multiplication design proposed in [125]. With large instances, storage tends to dominate energy consumption.

In Figure 5.1, the energy distribution of a matrix multiplication implementation is presented. As far as energy is concerned, almost 80% of an implementation is consumed in storage [113]. By embedding computing inside the memory, this energy is reduced since no transfers are needed.

5.2 Hardware Accelerators for Matrix Multiplication

GeMM is used in a variety of fields like neural networks in machine learning, image and signal processing, etc. In most applications, matrix multiplication is the performance bottleneck. Numerous works targeted this problem. We classified these methods in two major categories:

5.2.1 Software Implementations

Many works tend to reduce the complexity of the naive i-j-k algorithm. Strassen's algorithm in [208] drops the overall complexity from $O(n^3)$ to $O(n^{2.807})$. Williams' algorithm of $O(n^{2.373})$ in [238] is the best current performance for MM. However, the complexity of the operations makes it very difficult to use and today's hardware is unable to benefit from its performance since it requires very large matrices to show a noticeable advantage. Also, Bshouty [25] proved the existence of a lower bound of $O(n^2)$. This lower bound limits the capabilities of software-only improvements.

Bshouty [25] proved the existence of a lower bound of $O(n^2)$. This lower bound limits the capabilities of software-only improvements. Although this limit can be theoretically reached, numerical stability and low-rank complexity factors make it impractical to use the algorithm.

5.2.2 Hardware Improvements

5.2.2.1 CPU

One of the major issues of computing a matrix product is the memory organization and the non-sequential access. For this reason, cache enhancements were proposed to cope with this issue. These improvements profit from cache spatial locality to reduce slow memory accesses. Optimized SW libraries such as Intel's MKL also proposes efficient implementations. However, due to its limited parallelism, CPUs cannot cope with the increasing size of the problem.

5.2.2.2 FPGA

Jang et. al. [104] proposed an energy-time efficient approach for MM by storing only one of the input matrices. In another work, Jiang et. al. [107] proposed a Scalable Macro-pipelined architecture (SMPA). This architecture uses the temporal parallelism to maximize the use of processing elements (PEs) in order to get better throughput.

These works and others ([112] and [263]) profit from the highly parallel architecture of reconfigurable devices to maximize the performance or the resource usage. The downside of these methods is resource utilization since input matrices need to be stored in the programmable logic (PL). In [104], Jang et. al. reported 78% energy dissipation on storage for a 12×12 GeMM. With the growing size of matrices, the energy dissipation increases proportionally. This energy should be used for computation rather than storage.

5.2.2.3 Dedicated Hardware

Many application specific hardware (ASIC) was developed for matrix multiplication. They generally uses SUMMA [221] for its efficient memory management. However, a dedicated ASIC can only execute the targeted function. In the other side, our approach runs on an associative processor and can execute a variety of application compared to those solutions.

In a similar fashion, Google recently presented their TPU which uses systolic arrays to perform matrix multiplication. It is a CISC processor designed for machine learning applications. It targets data centers [117] and is not available for general public.

Many dedicated circuits use SUMMA [221]. Similarly to systolic arrays, computations can be easily mapped to different computing units. Since memory accesses are predetermined, data can be efficiently transferred between compute units.

Haron et. al. [86] used the IMPLY logic to perform GeMM. In their 3D method, input matrices are duplicated and all the multiplications are done in a single stage. The additions are done using a binary adder tree which gives the overall algorithm an $O(\log n)$ complexity. As far as we know, this is the most efficient algorithm. However, duplicating matrices n times can be costly with the matrix size growth.

Morad et. al. [155] proposed a dense and sparse MM using a General Purpose-Single Instruction Multiple Data processor and a sequential processor. Associative memory array is used by the SIMD processor to perform the parallel computations. In a similar manner, Yavits et. al. [248] proposed an efficient implementation of sparse MM on an associative processor.

Their approach, which can be seen in Figure 5.2, consists of three steps:

- *Broadcast*: distributes the elements of row $A_{j,*}$ over the the corresponding elements of column $B_{*,j}$.
- *Multiply*: performs the multiplication between each two pairs.
- *Reduce*: sums all the values to obtain elements of the resulting matrix.

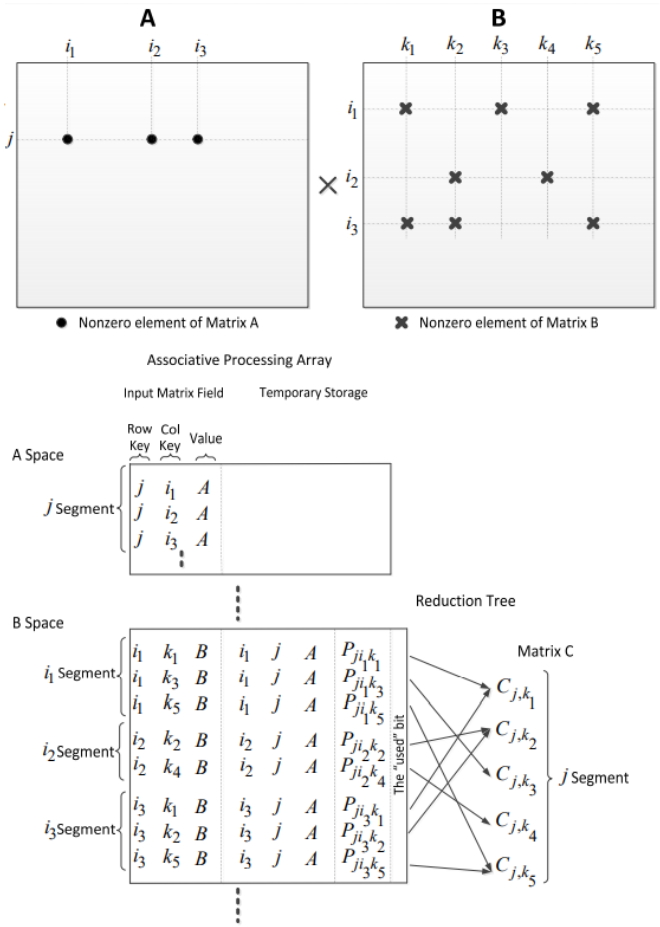


Figure 5.2: Sparse matrix multiplication on associative processor from [248]. Matrices A and B, shown in the top half, are stored in the CAM. Each row stores a non zero value as a triplet of the row index, the column index and the value.

Elements are then copied inside the CAM to perform desired operations per row. Elements of the jth row of A are copied to corresponding locations of ith column of B. The results are finally summed to obtain the final output.

At the end the operations are done in parallel and hence, the row is multiplied by the whole matrix.

When compared to existing methods, our approach profits from IMC to perform the operation. This gives a huge leap in performance and area compared to FPGA implementations since no data storage or loading is required. For implementations based on AP, our storage policy is very efficient since we do not store indexes which is common for associative processors. We also implement all the operations inside the CAM, therefore, there is no need for a reduction tree.

5.3 Resistive Associative Processor

Von Neumann Architectures suffer from the speed gap between the main compute unit and the main memory. This is called the von Neumann Bottleneck. This bottleneck prevents the compute unit from reaching its full potential. In Memory Computing gives a promising solution to this issue. In our work, we used the ReAP to accelerate matrix operations. For this we explain the architecture we used in this section.

In this section, we describe the required knowledge on Associative Processors in

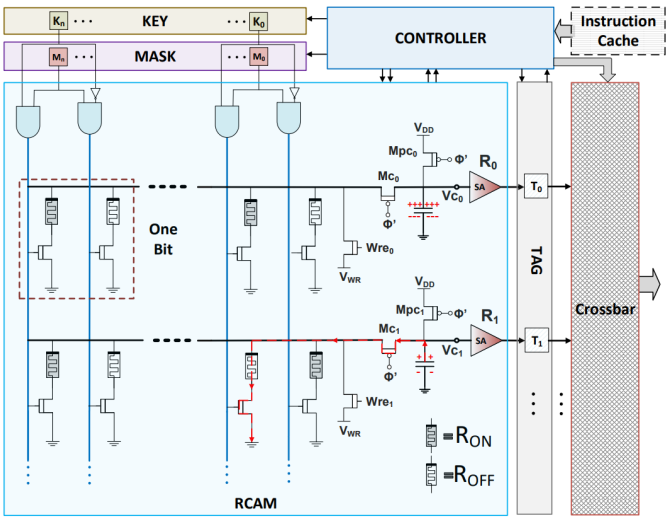


Figure 5.3: Architecture of an associative processor.

order to understand our approach. We give details about the overall architecture and how to exploit it to perform logic operations. Since we used a specific architecture, we only show how our processor works, later in Section 5, we describe the method used to reconfigure the AP for more complex operations as well as how others used memristor technology to accelerate the same application as us (GeMM).

5.3.1 ReAP Architecture

Figure 5.3 shows the architecture of an AP. It consists of a CAM, a controller, an instruction cache, an interconnection circuit and some specific registers (*Mask*, *Key* and *Tag*). We used a resistive implementation of the CAM memory, the RCAM. Each bit is represented using two memristors and two transistors.

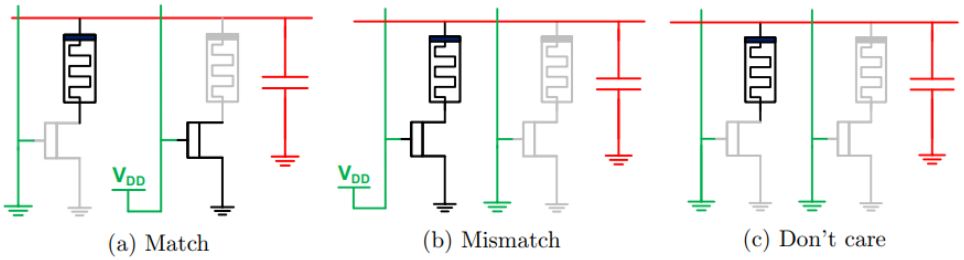
The CAM holds the data in a column format. A program is a set of instructions to be performed on that data. An instruction on AP consists of consecutive compare and write phases. A Look-Up Table (LUT) is used for each instruction. The execution of an instruction is the application of the LUT on the CAM. Operations in an AP are performed on data inside the CAM in a column per column fashion. Therefore, the number of rows does not affect the execution time of an operation.

The idea behind associative processing is not a new topic. It was introduced by Foster in [64]. However, it did not gained much attention due to CMOS technology limitations. With the upraise of memristors, CAMs can be implemented using a small number of transistors. The 2T2R cell architecture [249] only uses 2 transistors compared to 10 used in CMOS.

5.3.2 Logic execution

During the compare phase, columns to be compared against are marked with "1" in the *Mask*. Only values inside these columns are compared with the *Key*. If the value inside the CAM matches the *Key*, the row will be tagged "1" using the *Tag* register. For example, if the *Mask* is "101" and the *Key* is "100". Only the first and the third columns are considered. These columns should be compared with the first (from right to left) and third columns of the *Key* register and should be "0" and "1" respectively.

Architecturally, by setting a high voltage in the *Tag* register cell, a capacitor connected to each row would either discharge or maintain it's charge depending on the values of the row. If at least the value in one cell mismatch the value of the



(a) Match (b) Mismatch (c) Don't care

Figure 5.4: Typical evaluation phases of a ReAP cell for the match (a), mismatch (b), and don't care (c) states.

Table 5.1: LUT for Multiplication

Cr	R	B	A	Cr	R	Comment
0	0	0	0	0	0	NC
0	0	0	1	0	0	NC
0	0	1	0	0	0	NC
0	0	1	1	0	1	2nd Pass
0	1	0	0	0	1	NC
0	1	0	1	0	1	NC
0	1	1	0	0	1	NC
0	1	1	1	1	0	1st Pass
1	0	0	0	0	1	NP
1	0	0	1	0	1	3rd Pass
1	0	1	0	0	1	NP
1	0	1	1	1	0	NC
1	1	0	0	1	0	NP
1	1	0	1	1	0	4th Pass
1	1	1	0	1	0	NP
1	1	1	1	1	1	NC

key, the capacitor would discharge. This is similar to writing a zero in the *Tag* register at that line. Figure 5.4 shows the three possible cases during the compare phase. A match occurs if the looked-up values is the same stored in the cell. In this case, the capacitor in the right of Figure 5.4 can not find a way to a ground and therefore, remain charged for the active cell. If in the same cell a mismatch occurs, the resistance of the memristor and the activation of the transistor would create a path to the ground which will discharge the capacitor. In the case where the *Mask* is set to "0", the transistor is not activated and no discharge is possible which is equivalent to ignoring all the cells in that column.

In the write phase, only the rows that are tagged "1" are considered. Depending on the LUT of the current instruction, a value should be written to columns with a mask of "1".

Table 5.1 shows an example of the LUT for unsigned multiplication. "A", "B" and "R" are the masked bits of the corresponding variable. "Cr" is the carry bit and is always masked. In the compare phase, rows that correspond to the quadruple (A, B, R, Cr) in the left column of the table are tagged. If a line matches these values, the result "R" and carry "Cr" are updated using the right column of the LUT. Other quadruples are omitted since they do not affect the two outputs. The order of operations is important and is shown in the "Comment" tab in the right column. The quadruples need to be processed in the given order to prevent a row from being processed twice.

Multiplication cannot be done in place so we need a result column "R". Other combinations of "A", "B", "Cr" and "R" are omitted since they do not affect the current values of the two outputs "Cr" and "R", which are the active bits of the carry and the result respectively. The mask of the "Cr" column is always set. For

Table 5.2: Running time of various operations on AP. IP (resp. OOP) stands for In-Place (resp. Out-of-Place) execution. S and U stands for signed and unsigned representations.

Inst.	**Runtime**	**Area/Row**
NOT	$2m$	$2m$
AND	$2m$	$3m$
OR	$6m$	$3m$
Addition (IP, S/U)	$10m$	$2m+1$
Addition (OOP, S/U)	$11m$	$3m+1$
Subtraction (IP, S/U)	$10m$	$2m+1$
Subtraction (OOP, S/U)	$11m$	$3m+1$
2's Complement	$6m$	$2m+1$
Abs	$8m$	$2m+1$
Min (IP, S/U)	$10m$	$2m+2$
Multiplication (OOP, U)	$10m^2$	$4m$
MAC (OOP, U)	$10m^2+10m$	$4m$
Multiplication (OOP, S)	$10m^2+4m-14$	$8m+4$

each cycle, the mask will be set to "1" for a bit of A, B and R. The compare phase will look for the combinations as ordered in the table. The first combination to be searched for is "0111". For rows that match these values, the corresponding "Cr" and "R" columns will be set to "10".

We implemented a set of other operations that we needed for the MM problem. We present in Table 5.2 the different time-space complexities of the implemented operations. "m" is the number of bits of the input (operands). IP/OOP stands for In-Place/Out-of-Place execution respectively. Signed and unsigned operations are marked "U" and "S" respectively.

5.3.3 Circuit Implementation

In [132], a 2-transistor-2-memristor CAM cell is presented. A memristor can change its resistance. We only consider two levels, Ron and Roff. In the experiments, Ron is set to 500 Ω and Roff to 10k Ω. Depending on their order, the two memristors inside a cell can store a value of "1" (Ron - Roff) or "0" (Roff - Ron). Writing consists of flipping the resistance of these two memristors to match the required order of the desired value.

Before each compare phase, a pre-charge step is required. A capacitor connected to each CAM row is charged. If the value inside the *Key* does not correspond to the value inside the cell, a line connecting the capacitor to the ground can be found and a discharge will take effect (row 2 in Figure 5.3). Otherwise, the capacitor will maintain its charge if the searched word corresponds to the stored value in the row (row 1 in Figure 5.3). After that, a sense amplifier decides on the logical value of the tag register by comparing the voltage across the capacitor to the threshold voltage (V_{th}).

5.3.4 Data Management

Data inside the CAM can be moved using a Switching Matrix (SM). Each CAM cell is connected to other cells via a crossbar like in Figure 5.5. The SM can be implemented using memristors. We can consider the Ron as a short circuit and the

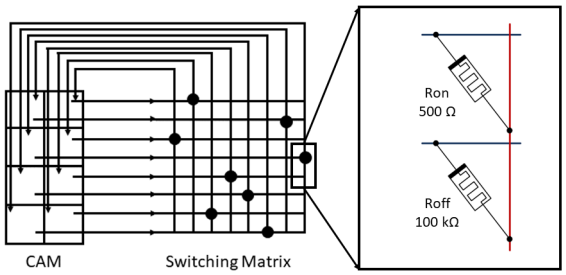


Figure 5.5: Connection between the CAM and the SM. The black dots stands for a short circuit which means that the input line connected to this dot will be redirected to the output line connected to the CAM.

Roff as an open one. With this in mind, only one memristor connecting a horizontal line with a vertical line in the SM can play the role of a simple switch like in Figure 5.5. Memristors also allow reconfigurability since we can change the values and hence the interconnection scheme.

5.4 Matrix Multiplication Algorithm

In this section we present an efficient MM implementation of Cannon's algorithm [30] on ReAP. We first discuss how we store the data inside the CAM. The operations needed for the operation are then explained.

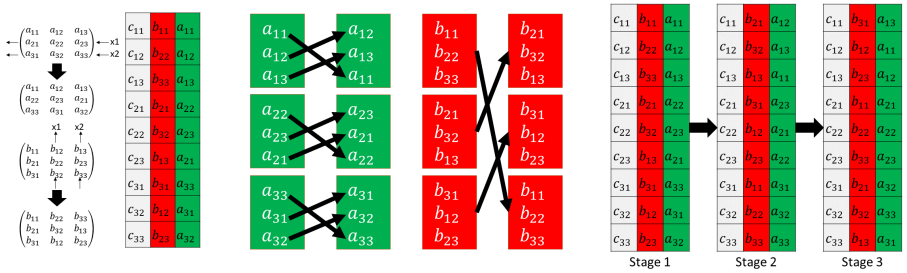
5.4.1 Cannon's Algorithm on ReAP

Given a general matrix multiplication $C = A * B$. Cannon's algorithm as shown in Algorithm 2 performs a series of rotations followed by a Multiply-and-Accumulate (MAC) to compute the output C. A, B and are square matrices of n rows. A grid of n by n processing elements (PEs) can efficiently performs the required operations in n compute stages. Each processing element PE_X holds a value of A: $PE_{X,A}$, a value of B: $PE_{X,B}$ and computes a value of C: $PE_{X,C}$. After each stage, elements of A and B are circularly transferred between processors to satisfy the matrix product formula for each value of C: $c_{i,j} = \sum_{k=1}^{k=n} a_{i,k} * b_{k,j}$

Algorithm 2 Cannon's Algorithm

procedure MM(**Mat**: A, **Mat**: B, **Mat**: C)
  for $i = 0 \dots n$ **do**
    $A_{i,\bullet} \leftarrow CircularShiftLeft(i)$
  end for
  for $i = 0 \dots n$ **do**
    $B\bullet, i \leftarrow CircularShiftUp(i)$
  end for
  for $k = 0 \dots n$ **do**
    $PE_{X,C} \leftarrow PE_{X,C} + PE_{X,A} * PE_{X,B}$
    $Ak, \bullet \leftarrow CircularShiftLeft(1)$
    $B\bullet, k \leftarrow CircularShiftUp(1)$
  end for
end procedure

In Algorithm 2, $A_{i,\bullet}$ represents the i-th row of matrix A and $B_{\bullet,k}$ represents the k-th column of matrix B. The *CircularShift* operations rotates the elements of a vector such that the first element is stored at the end and all the other values are



(a) CAM state after loading input matrices.

(b) Inner and outer rotations.

(c) CAM state after each stage rotation.

Figure 5.6: The execution scheme of MM on ReAP with the loading phase in 5.6a and the rotations after each stage in 5.6b. Intermediate CAM states after rotations in 5.6c.

shifted. If a parameter i different than one is passed, the first element would be stored at $n - i$-th position while shifting the other values similarly.

5.4.2 Cannon Algorithm on CAM

In a ReAP, the CAM cells are the compute units. Each row is able to perform one operation on values residing in its cells. For this reason, we store matrices in vectorized format. Three columns are needed to store A, B and C. The execution consists of consecutive execute-and-rotate stages. At each stage, we multiply columns A and B and add the results to column C. Values of C are initially set to "0". Then, we rotate the data inside the CAM to match Cannon's circular shifts.

In Figure 5.6, we see the execution stages on ReAP. After storing the two matrices, we perform a first MAC operation. Once the MAC operation is finished, we should prepare the data for the next stage. We have computed one partial sum of each one of the C values. Since matrices are stored in a vector format (1D) inside the CAM, Cannon's circular shifts are replaced by inner and outer rotations as illustrated in Figure 5.6b. Algorithm 3 illustrate the multiplication of two matrices stored in columns A and B inside the CAM.

The execution is a loop of n stages. In each stage, we compute one multiplication for each element in C. The MAC operation performs a multiply-and-accumulate operation between columns A and B and stores the temporary result in the column C. The rotations in 5.6b are performed on the SM with the two instructions $InnerRot$ and $OuterRot$. These two operations run simultaneously on the SM. The dots in Figure 5.5 corresponds to all the required rotations in the algorithm. The time needed to perform these rotations is constant. Since MAC operations run also in a constant time, the loop in Algorithm 3 performs a constant number of operation for n times, which proves the linear complexity of our algorithm.

In Figure 5.6c we can see the results of these rotations. The MAC operation is performed between each two rotations. Considering MAC-and-Rotate for the first row, the value computed is: $b_{11} * a_{11} + b_{21} * a_{12} + b_{31} * a_{13}$ which is actually c_{11}.

5.4.3 Generalization

All operations between two matrices, that can be performed in an ijk format (row-per-column) can be accelerated by this approach i.e. algorithms of the form:

$$c_{i,j} = \otimes_k(\oplus(a_{i,k}, b_{k,j}))$$

Algorithm 3 MM on ReAP Execution

```
procedure MM(Column: A, Column: B, Column: C)
    for i = 1...n do
        C ← MAC(A, B)                          ▷ Perform MAC on the CAM
        {                                      ▷ Parallel rotations inside the SM
            A ← InnerRot(A)                    ▷ Rotations for column A
            B ← OuterRot(B)                    ▷ Rotations for column B
        }
    end for
end procedure
```

Where $\otimes_k$ is an operation over a set and $\oplus$ is an operation between two values $a_{i,k}, b_{k,j}$. For MM, $\otimes_k$ is the *sum over k elements* and $\oplus$ is a *multiplication*.

Algorithms of this class can be found in a variety of linear algebra computations. Dominance product [223] is one of these algorithms. We denote it as $C = A \circledcirc B$. It is defined as:

$$c_{i,j} = |\{k/a_{i,k} \leq b_{k,j}\}|$$

In this product, the two operators are $||$ (which denotes cardinal of a set) and $\leq$. The implementation of the dominance product using our method also yields a linear time and is performed as follows. LUTs for $||$ and $\leq$ are first implemented in the ReAP. Then A and B are stored inside the CAM as shown in subsection 5.6a. The results are then computed in parallel inside the ReAP. For a given stage, we compare the value of A with the value of B. If the value of A is less than or equal to the value of B, we store "1" in a temporary column T. At the end of each stage, the cardinal can be implemented as an *Increment If* operation when the value in T is "1".

Another matrix processing that can be accelerated by our approach is the All-Pairs Shortest Paths (APSP) algorithm [63]. This algorithm also knows as the Floyd Warshall algorithm consists of computing the shortest paths between every two nodes in a graph.

For a given graph $G(V, E)$, we can define the adjacency matrix A which is a $|V|$ square matrix. Each element $a_{i,j}$ is the distance between vertices i and j in G. The APSP is defined as:

$$c_{i,j} = \min_k(a_{i,k} + a_{k,j})$$

In this case, operators are $\min_k$ and $+$. Similar to MM, an atomic operation, in this case, is the sum of the two elements residing on the same row. The aggregation operator is the min. This can be implemented using an in-place minimum operator.

The complexity of operations used in APSP and the dominance product are similar in ReAP. For this, and also for the sake of clearness, we will only be including APSP results in the experiments.

5.5 Experimental Results

In this section, we present experimental values obtained during simulation. We will first explain the experimental setup. Results are discussed later in this section.

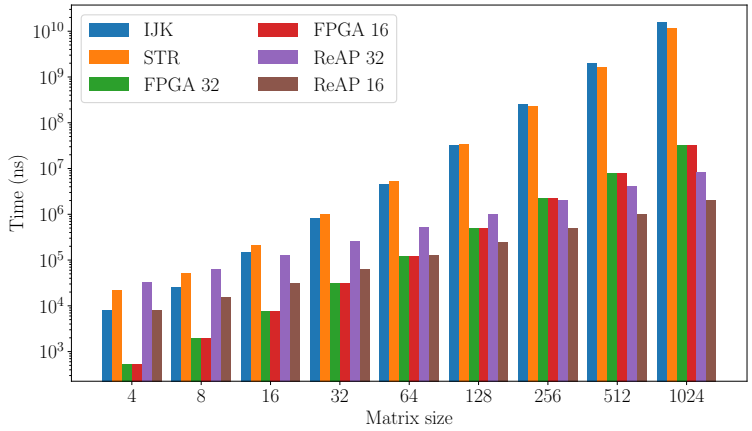
For performance evaluation, we used the cycle-accurate associative processor simulator in [247]. For energy comparisons, HSpice is used to emulate the behavioral part of the ReAP. We generate the number of compares/writes for each row/cell, and using a ReAP netlist, we compute the energy consumption for a given cycle.

We compare the results of this design to FPGA. For the FPGA part, we used Vivado HLS to create an IP core for GeMM [17]. The generated design uses pipelining

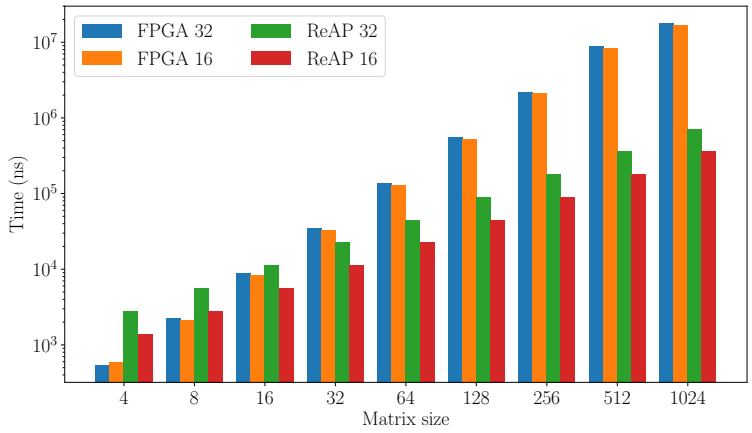
for the outer loop in the matrix product and unrolling/flattening for the inner loops. Input matrices are also partitioned along the correspondent dimension (1 for first matrix and 2 for the second). The partitioning factor is equal to the number of elements in each row/column. With these directives, the generated IP will create as much multiply and accumulators as it can to reach maximum parallel computations with a pipeline interval of 1. We tested the obtained IP for latency and energy results on a Virtex-7 XC7V485T.

5.5.1 Performance

We compared the execution time of same instance sizes and compared the results in Figure 5.7a. This figure compares execution times for 6 implementations: on an i7 CPU using the conventional IJK algorithm and with the Strassen's (STR) implementation, on a FPGA using 16-bits and 32-bits implementation [17] and finally, using our approach on a ReAP using 16-bits and 32-bits implementations. With relatively big matrices (256x256), the FPGA design provides very poor performance compared to the proposed ReAP method. We can also see that the time needed to compute 1024 matrix product is lower on ReAP than the 512 MM on FPGA.



(a) Matrix multiplication execution time (ns).



(b) All pairs shortest paths execution time (ns).

Figure 5.7: Execution time in nsec for MM (left) and APSP (right) on an i7 CPU (IJK, Strassen), an FPGA and a ReAP for 16-bit and 32-bit data width.

These measurements were conducted without including the time needed to transfer inputs to the compute unit (the IP core); If we consider the communication overhead, which is a major issue when dealing with matrix multiplication, the time can be very important. Also, if we deal with recurrent data, ReAP can hold the redundant matrix since the 2T2R cells can be used as memory at the same time as compute units. In this case, we only transfer the modified data, re-perform the computations and read the new output.

The poor performance of MM on ReAP for small instances is caused mainly by two factors:

- Level of parallelism: on ReAP, the number of operations to be done in parallel depends on the input size. For small matrices, a low number of operations is performed at each stage.
- Data width: the number of cycles to perform the multiplication on ReAP depends on the number of stages i.e. input size n and the data width m. For very small values of n, the execution time is driven by the multiplication operation complexity. We measured the execution time of 16 bit inputs on the two cores and compared the data to the previous results in Figure 5.7. Beyond a size of 64x64 element, our ReAP implementation outperforms the Xilinx design.

It is possible to fix this performance issue by allowing data duplication for small instances since it will not consume much resources. Another solution is the usage of high frequency memristor devices like the crossbar used in [86] with 5 Ghz frequency compared to ours (500 Mhz).

As mentioned in Subsection 5.4.3, the proposed approach for MM can be generalized to any row per column application. For the APSP example, the execution time is dramatically reduced as shown in Figure 5.7b. The + and *min* operations can be performed in a linear time depending on the word size on ReAP. For a word length of m=32-bits, one stage will only take 64 passes to complete. The final results are two orders of magnitude better than the FPGA design.

In addition, the ReAP is also able to store the two matrices for future usage and no time is needed to load them. Matrices can represent weights in a neural network, hence, the inference part can be done easily with no data transfers.

CPUs perform very badly against accelerators as shown in Figure 5.7a. For this, we decided to exclude it from area and energy comparisons.

5.5.2 Area

Since memristor layer can be built on top of CMOS layer [209], we used the number of transistors as a comparison metric for area. This metric also gives a small insight about the cost of the two designs.

The ReAP approach requires storing the two matrices in separate columns. One column is needed for the result matrix. For MM, and for a word size of m, we need $4m$ cells per line; each cell contains 2 transistors and the number of rows is equal to the matrix size (n^2). The overall number of transistors is calculated using the equation:

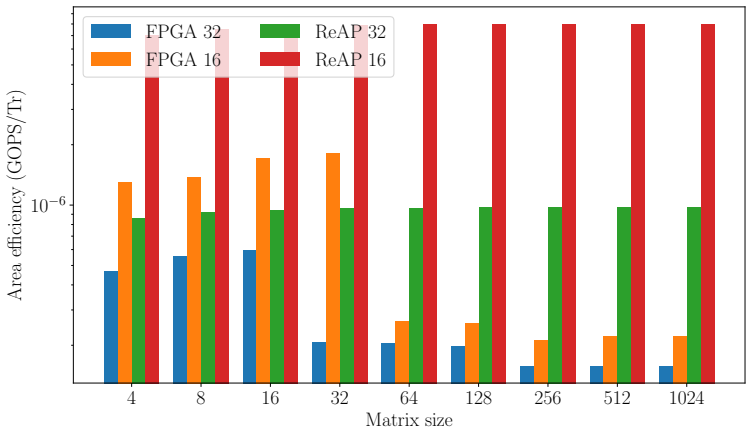
$$Tr_{ReAP} = 2 * (4m) * n^2$$

For FPGA, since the resource report does not contain the transistor count, we estimated this number using the formula:

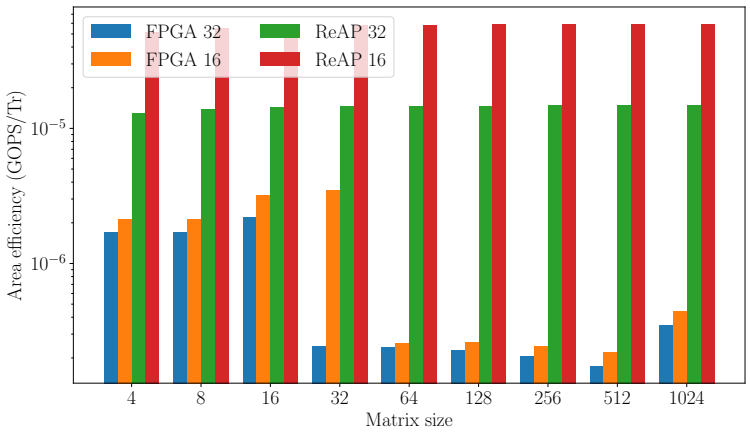
$$Tr_{FPGA} = Tr_{BRAM} * N_{BRAM} + Tr_{FF} * N_{FF} \\ + Tr_{LUT} * N_{LUT} + Tr_{DSP} * N_{DSP}$$

With $N_{Component}$ and $Tr_{Component}$ the number of instances of the component and the minimum number of transistors to realize it[1].

[1]For an 18K BRAM we need 18*1024*8*6 transistors. 18*1024 is the number of words, 8 is the bit-width and 6 is the minimum number of transistors for a memory cell.



(a) Efficiency in area usage for matrix multiplication.



(b) Efficiency in area usage for APSP.

Figure 5.8: Area efficiency in GOPS per transistor for MM (left) and APSP (right).

In Figure 5.8 we computed the number of transistors obtained by the previous formula with the corresponding estimated count on FPGA. Area efficiency is then calculated as performance per transistor. The results show an order of magnitude better efficiency for MM and two orders for the all-pairs shortest path. This proves a very small cost for the same operation. For FPGA, area depends on the type of resources used. For matrices bigger than 32, FPGA starts using BRAMs to store matrices which leads to area deficiency.

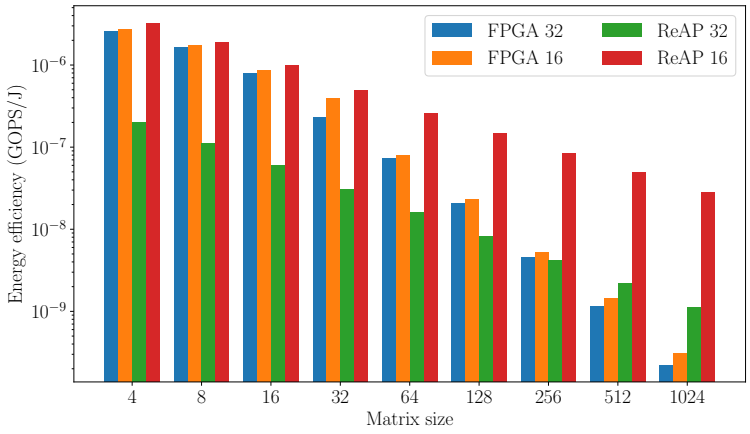
The huge difference in transistor count is mainly due to the simplicity of the 2T2R cell design [132], and the number of cells is exactly the size of the matrices times the data width. For FPGA, resources are spread between storage, logic, and communication which explains the efficiency of our approach compared to FPGA. The reduced number of transistor also affects indirectly the chip size. Since halving chip size reduces chip cost by roughly a factor of 8 (2^3) [91, 117], ReAPs will be relatively cheap compared to other similar embedded cores.

To measure the energy consumption of our approach, we have created an equivalent netlist for each cycle and simulated the generated design on HSpice. We have used a frequency of 500Mhz (2ns for each cycle) and we obtained the results shown in Figure 5.9.

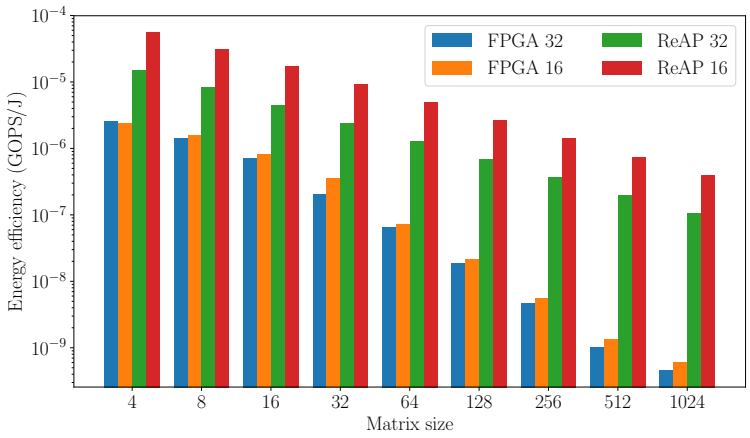
During the compare cycle, only 0.007 pJ were needed for each row on average.

One write operation costs 1.074 pJ for each cell. This huge difference affects poorly the overall consumption results which we discussed earlier. This energy can also be reduced with better memristors since they are still in their debut.

In Figure 5.9a we compared energy efficiency. For the 32-bit MM, we see an advantage for the FPGA design in small designs. This is caused by the large number of required writes. For 16-bit implementation, the small number of resources plays a big role in giving to the ReAP approach better results.



(a) Efficiency in energy consumption for matrix multiplication.



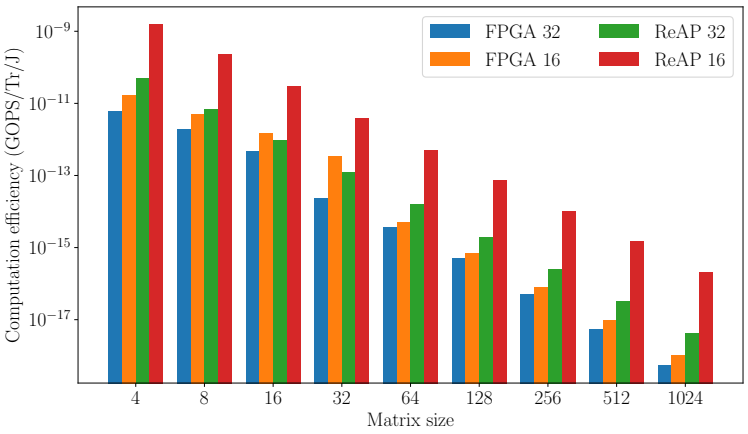
(b) Efficiency in energy consumption for APSP.

Figure 5.9: Energy efficiency in GOPS per Joule for MM (left) and APSP (right).

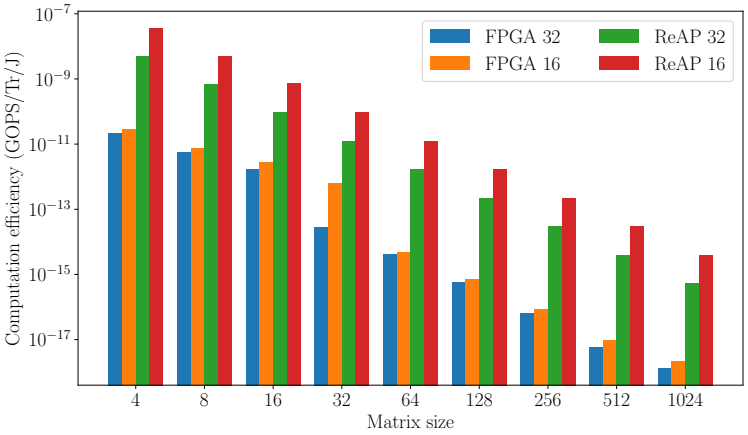
An interesting observation is the low energy consumption of the design for the 32-bit APSP compared the 32-bit MM in 5.9b even though they use the same number of resources. This difference is caused by the hight write energy. APSP changes a value only if this node offers better shortest path that the current value. On average, this value is not altered a lot. This results in low number of write operations and hence, low energy consumption.

5.6 Contributions and Guidelines

In this section, we presented a novel implementation of matrix multiplication on an associative processor. The method runs on a Resistive Associative Processor. The



(a) Matrix multiplication computation efficiency.



(b) All pairs shortest paths computation efficiency.

Figure 5.10: Computation efficiency in GOPS per transistor per Joule for MM (left) and APSP (right).

proposed technique shows better timing results compared to other efficient implementations on CPU and FPGA.

Since the memristor layer can be placed on top of the CMOS layer, resulting area is very efficient compared to FPGA based techniques. The number of transistors is an order of magnitude less in ReAP than in FPGA. This offers very high density and hence, very high computing power.

The only downside of the ReAP compared to other implementations can be the energy consumption. When the number of writes in the ReAP becomes important, power consumption can be problematic. For very big designs, this issue becomes a barrier that needs to be addressed. However, it is possible to reduce the number of write operations by tagging fewer rows each cycle but this can be application-dependent. Proposing heterogeneous cell architectures such as read-only cells and read-write cells can be handy in some cases.

Although current memristors faces variability[2] issues and short life cycles, it is

[2]With the current state of technology, memristors have very high variabilities which may reduce their precision. This is, very confidently, going to change with future implementations. Since we only considered two levels of resistance (Ron and Roff), This problem is not as critical as our implementations such as memristor-based neuromorphic circuits.

very plausible to discuss the integration of memristor-based components in today's FPGA SoCs. The associative computing power combined with the non-volatile in-memory capabilities can be very useful in the hands of every SoCs developer. Matrix and vector operations constitute a huge portion of signal and image processing applications. The linear time for the first and constant for the latter on the ReAP are two very attractive characteristics.

Chapter 6

Design for Performance: On Constant Multiplication

Contents

In autonomous driving scenarios, CNNs are used for inference. A fully operational version is deployed and, in most cases, no training is needed on the vehicle. In this chapter, we tackle the problem of CNNs' inference by optimizing resource utilization, power consumption and delay through a low level over-squeezing of the multiplication circuit. We exploit the constant aspect of trained CNNs weights to reduce the required resources to implement multipliers. Our experiments show that with over-squeezing the multiplier, we achieve considerable speedup in the convolution and fully connected layers. Results also show that with squeezing multipliers at circuit level, overall multipliers energy consumption drops by more than 20× with no accuracy loss.

6.1 Motivations

Deep learning systems such as CNNs have shown remarkable success during the last few years thanks to empirical achievements on a wide range of complex real life problems. From handwritten digit and speech recognition to environment perception for autonomous cars, these systems have demonstrated their ability to train robust feature extractors that can be successfully exploited by a classifier.

CNNs are complex in nature and require massive amounts of data. This fact leads to costly hardware implementations in embedded systems. The *de-facto* used accelerators in the industry are GPUs. Since these latter are power hungry, they are not suitable for low power-budget embedded systems. In the KITTI challenge [67], CNNs' runtime ranges between 0.2 seconds to 4 seconds which corresponds to 5 to 0.25 frames-per-second respectively. As these networks may be used in critical real time applications such as autonomous vehicle, or power-limited systems such as autonomous drones, it is significantly challenging to meet both time and energy specifications. For this reason, a plethora of research work has been focusing on optimizing the inference of CNNs and a variety of techniques have been proposed for this purpose [3].

Earlier implementations, especially GPU-based CNNs, were designed in an accuracy-oriented manner without much focus on the implementation constraints. However, it led to heavy architectures that are power and resource-hungry and thereby challenging to deploy. This sparked a prolific area of research, resulting in a variety of techniques for optimizing their implementation at different levels.

In [3], authors enumerate 4 optimization categories: First, algorithmic optimizations focus on the software implementation. Second, datapath optimizations address the memory bottleneck. Compiler-based techniques rely on high-level tools and libraries such as HLS and CUDA to optimize the implementation. Finally, model optimization techniques rely on the approximate computing paradigm, pruning and compressing CNN architectures.

Going from software optimization, to memory management or model optimization, we notice that all these techniques result in different trade-offs between power consumption and resource utilization on the one hand and accuracy on the other hand. However, to the best of our knowledge, none of them took profit from intrinsic features of CNNs to optimize the network ***at circuit level***. In fact, once the training phase is achieved, CNN weights remain constant and during the inference only inputs are variable.

We tackle the problem from a new angle by suggesting drastic optimization in the multiplier circuit design. Our work is motivated by the observation that in CNN-dedicated multipliers, one of the two operands that corresponds to the synapse weight is permanently constant. This is due to the fact that inputs are multiplied by *constant weights* identified off-line during the learning phase. Hence, a fine grain optimization opportunity in multiplication circuits can be achieved.

The main scientific contributions of this chapter are summarized as follows:

- We suggest an exhaustive RTL-level and Circuit-level optimization of multiply

operations for both convolution layers and fully connected layers to reduce FPGA and ASIC-oriented CNNs resource utilization and power efficiency.

- Our approach allows the implementation of CNNs **without storing network weights**, which saves considerable memory space **with no loss of accuracy**.
- We publish an open source tool that generates HDL source of a convolution layer with compressed multipliers based on the layer weights.

6.2 Background

Artificial Neural Networks (ANNs) in general and CNNs in particular consist of processing a given input through a set of layers. Each layer performs operations between the input I and a set of precomputed coefficients referred to as weights (W). In conventional architectures, the output of each layer is forwarded as an input to the subsequent layer. The overall network results are the outputs of the final layer.

The weights used inside each layer are obtained at the training phase of the network. The training consists of reducing the error between the network output and a desired output by continuously updating weights until reaching the desired accuracy. This phase requires a dataset with sufficient number of samples and is performed in general at off-line. Once the weights are identified, no training is required and the network is ready for classification, detection or prediction. In this chapter, we only focus on this later part, the inference part, for real time applications.

CNNs are particular deep neural networks where the main particularity is the introduction of convolution layers. In these layers, a set of convolutions is performed between the input and a set of kernels which represents the layer weights. In CNNs, the role of the convolution layers is to perform automatic features extraction. Once features are extracted, a set of fully connected layers act as a classifier and outputs the final result. The number of layers and parameters per layer is an empiric design choice.

The output of a convolutional layer is computed as follows:

$$O[c][x][y] = B[c] + \sum_{k=0}^{C-1} \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} I[k][Sx+i][Sy+j] \times W[c][k][i][j] \qquad (6.1)$$

where O is the output feature map, B is the bias vector, C is the number of input channels, R is the size of the kernel, S is the stride and W is the weight matrix.

In Figure 6.1, we show the number of operations performed during one inference run for six different networks. It is clear that the majority of workload is dedicated to MACC operations. In other words, one can focus on optimizing MACC hungry layers thereby having a considerable impact on the overall CNN performance and power consumption.

Besides multiplications, Equation 6.1 also requires two memory accesses for reading the input I and the weight W for each multiplication.

6.3 Related works on complexity reduction

A plethora of optimizations have been proposed to reduce processing and memory requirements of CNNs on embedded platforms. Model optimization techniques rely on the approximate computing paradigm, pruning and compression of CNN architectures. We notice that all the proposed techniques result in inevitable trade-offs between power consumption and resource utilization on the one hand and accuracy on the other hand. These approaches can be divided into two main categories: Reducing precision and Network compression.

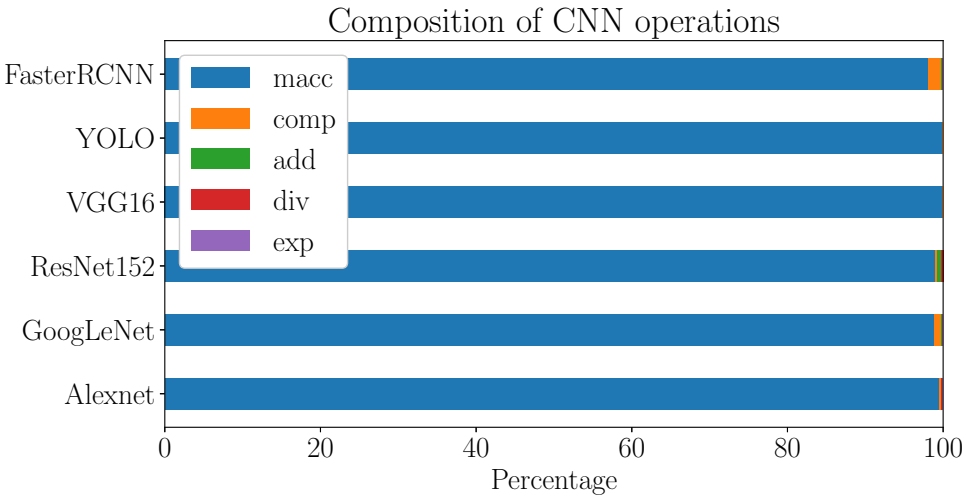


Figure 6.1: Percentage of Multiply and ACCumulate (MACC), comparison (comp), addition (add), division (div) and exponent (exp) operations for different networks

6.3.1 Reducing Precision

Reducing operands' precision includes using fixed point instead of floating point, data quantization and weight sharing. The idea is to transpose data into a smaller space of quantization levels. Practically, the process aims at minimizing the error between the quantized and the initial data. The precision is correlated to the number of quantization levels and consequently to the number of bits required to represent the data.

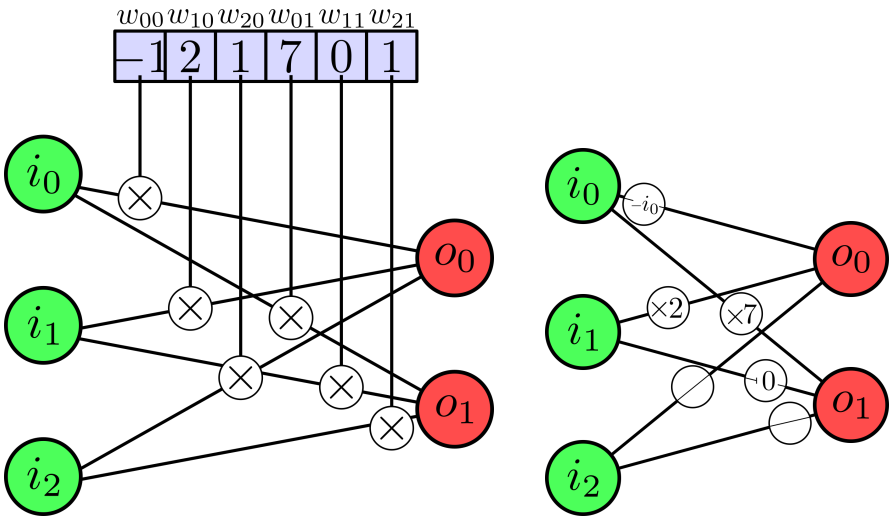
The most straightforward quantization approach consists of a linear mapping with uniform distance between each quantization level. It usually consists of converting values from floating point to an N-bit fixed point number. Authors in [147] reduce the weights bitwidth to 8 bits and the activations to 10 bits. In [79], both weights and activations can reach 8-bits with fine-tuning. In [49], authors manage to reduce even more aggressively the data bitwidth. By introducing the concept of binary weights (-1 and 1), the multiply operation is reduced to addition and subtraction only. The same idea is extended in [46] by using binary weights and activations, thereby reducing the MAC operation to an XNOR. However, these two approaches have a dramatic accuracy loss of 19% and 29.8%, respectively [190].

While these works rely on linear quantization with uniformly spaced out values, the weights and activations distributions are not uniform [83, 152].

For example, in [79, 152], weights are quantized to powers of two. Consequently, the multiply operation is substituted by a bitshift operation.

In [35], authors suggest weight sharing. The approach consists of assigning a single value to different weights in order to reduce the number of unique weights by filter and thereby reduce the memory size.

In [89], authors noticed that the same weight occurs multiple times in or across weight vectors. They proposed a new CNN architecture, named *Unique Weight CNN Accelerator* (UCNN), to exploit this weight repetition. Due the to different applied compression and quantization techniques, the number of unique weights in each filter has decreased dramatically. Using a memorization technique and three fixed and unique weights par layer, they obtained a 3.1× performance increase.



(a) Portion of a standard fully connected layer.

(b) Constant weight impact on the architecture.

Figure 6.2: Illustration of our idea with 6.2a, the network before constant-multiplication compression and 6.2b the result after customized multiplication compression.

6.3.2 Network Compression

Besides the approach of tuning the data precision, a plethora of works in literature has focused on reducing the network size and the number of performed operations. Reducing the network size and optimizing the number of operations includes techniques such as compression, pruning and compact network architectures.

The sparsity of the rectified linear unit (ReLU) output activation is exploited in [38] to reduce memory access, particularly to costly off-chip memory access. The proposed reconfigurable hardware skips reading the weights and performing the MAC for zero-valued activation thereby reducing energy cost by 45%. Authors in [7] go even further and instead of just gating the read and MAC operation for zero-valued weights, they suggest to skip the cycle to increase the throughput by 1.37×.

Networks are usually over-parameterized and a large amount of redundancy exists within their weights. Network pruning techniques such those proposed in [84,174,246] aim at removing the redundancy. To maintain the primary accuracy level, aggressive network pruning techniques may require weights fine-tuning.

To the best of our knowledge, none of the optimization techniques took profit from intrinsic features of CNNs to optimize the network implementation.

Efficient algorithms for constant multiplication have been discussed in [225]. These algorithms have been implemented in diverse applications such as FFT [186], matrix-vector multiplication [4], FIR filters [250] and graph applications [110].

A multiplierless implementation of CNN was proposed in [74]. However, authors only considered power of two weights. A more general implementation of a multiplierless neuron for artificial neural networks (no convolutional layer) is presented in [198]. In their work, the authors changed the architecture of a single multiplier to perform an approximate operation for the given input. Since their version is not exact, a trade-off between accuracy and resource efficiency is inevitable.

In [176] the CORDIC algorithm was used to replace multiplications. By choosing weights as trigonometric functions and constraining them to line in $[-1, +1]$ range, they are able to perform convolutions without implementing multiplications. While this technique achieves interesting results in terms of resource efficiency, the implementation still requires high memory size to store weights.

Exact multiplier-less implementations of CNNs have not been discussed in pre-

vious works. In this chapter, we present a novel approach that relies on constant multiplication within CNN operations. In fact, once the training phase is achieved, weights remain constant in most common CNNs. During the inference phase, only inputs are variable unless a new training is performed. Since only one operand is variable for each multiplier in a given architecture, it is profitable to replace the conventional multiplier with a version customized to the constant value. Therefore, the proposed compression technique does not require any weight storage. Considering that memory bottleneck is a major problem in CNN implementation, eliminating weight memory halves the required accesses for multiplications. A simplified illustration of this idea is presented in Figure 6.2.

6.4 Proposed Approach

6.4.1 The impact of operands value on the operation

Figure 6.3 illustrates the architecture of a 4x4 array multiplier. While the first row is composed of AND gates, the subsequent lines are based on building blocks implemented by AND gates along with full adders.

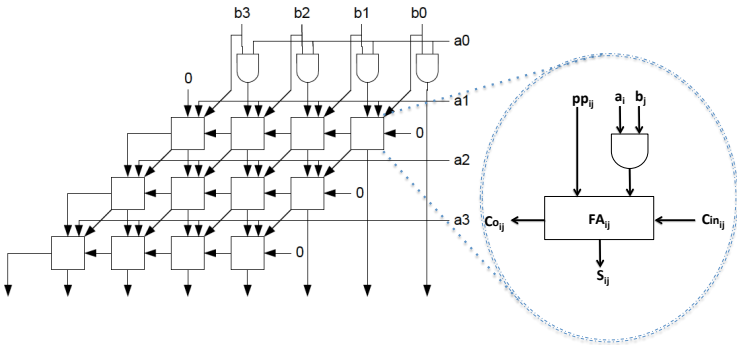


Figure 6.3: A 4x4 array multiplier architecture.

We consider the notation presented in Figure 6.3. The building block parameters are expressed as follows:

$$S_{ij} = (a_i.b_j) \oplus pp_{ij} \oplus Cin_{ij} \tag{6.2}$$

$$Co_{ij} = (a_i.b_j).pp_{ij} + Cin_{ij}.(a_i.b_j \oplus pp_{ij}) \tag{6.3}$$

Where S_{ij} is the output of individual full adder blocks and Co_{ij} is the carry of the same block. Moreover, as shown in Figure 6.3, the array multiplier architecture is given such that:

$$\forall i, Cin_{i0} = 0 \tag{6.4}$$

And

$$\forall j \geq 1, Cin_{ij+1} = Co_{ij} \tag{6.5}$$

In the following, we study the cases where operand A, i.e. $a_3a_2a_1a_0$, is constant. For this we consider the cases where a_i is set to 0 and 1.

6.4.1.1 Bit 0 case

Let's consider the case where a bit $a_i = 0$. Hence, the parameters are expressed as follows:

$$S_{ij}^{0} = pp_{ij} \oplus Cin_{ij} \tag{6.6}$$

And

$$Co_{ij}^{0} = Cin_{ij}.pp_{ij} \tag{6.7}$$

Moreover, given Equations 6.4 and 6.5, Equation 6.7 becomes:

$$\forall i, j \ while \ a_i = 0, Co_{ij}^{0} = 0 \tag{6.8}$$

Consequently, Equation 6.6 becomes:

$$\forall i, j \ while \ a_i = 0, S_{ij}^{0} = pp_{ij} \tag{6.9}$$

Architecturally, Equations 6.9 and 6.8 correspond to deleting the whole row of building blocks or first row of AND gates corresponding to the bit equal to 0 and forwarding the previous signals to the subsequent row.

6.4.1.2 Bit 1 case

If we suppose $a_i = 1$, The parameters are expressed as follows:

$$S_{ij}^{1} = b_j \oplus pp_{ij} \oplus Cin_{ij} \tag{6.10}$$

$$Co_{ij} = Cin_{ij}.pp_{ij} \tag{6.11}$$

Architecturally, Equations 6.10 and 6.11 correspond to deleting the whole row of AND gates corresponding to the bit equal to 1.

To summarize, using a compressed version of multipliers requires less resources for the same operation. Moreover, 0 bits have more impact on compression rate. The more zeros in a network weights, the less resources it requires. This is validated later in subsection 6.7.1.

6.4.2 Fixed operands in CNNs

The second observation is a trivial aspect of neural networks. In fact, once CNNs are trained, their weights are identified and fixed once and for all. The deployed architecture is thereby fixed. The inputs of each layer are going to be multiplied by the same weights for the CNN lifetime. Hence, either in convolutions or within fully connected layers, multiplications occur with fixed operands.

Based on the first observation, we exploit the constant-operand multiplication to optimize the CNN implementation. The multiplier circuit is squeezed at design time to a new fitted circuit with fewer stages, thereby less resources. Consequently the circuit takes less time performing the multiply operation and consumes less power than a conventional multiplier. It is worth mentioning that, while this aspect is intrinsic to CNNs, to the best of our knowledge, it has not been considered for optimization purposes before.

An illustration of the circuit fitting is shown in Figure 6.4. In this input case where one operand is fixed to "1010", a whole row of AND gates as well as a row of FA-based building blocks are deleted from the circuit. The resulting multiplier is hence implemented using 100% less AND gates (outside fulls adders) and 1/3 less full adders compared to a conventional multiplier.

For FPGA-oriented inference, the impact of this observation on the RTL implementation is drastic. In fact, we not only save FPGA fabric resources, but also memory elements. As the multiplier is weight-customized, one does not need to store the weights on board and the only convolution layer inputs are sufficient to achieve the MACC operations. Such impact helps not only reducing resource utilization and power consumption but also memory access bottlenecks. In fact, one could notice

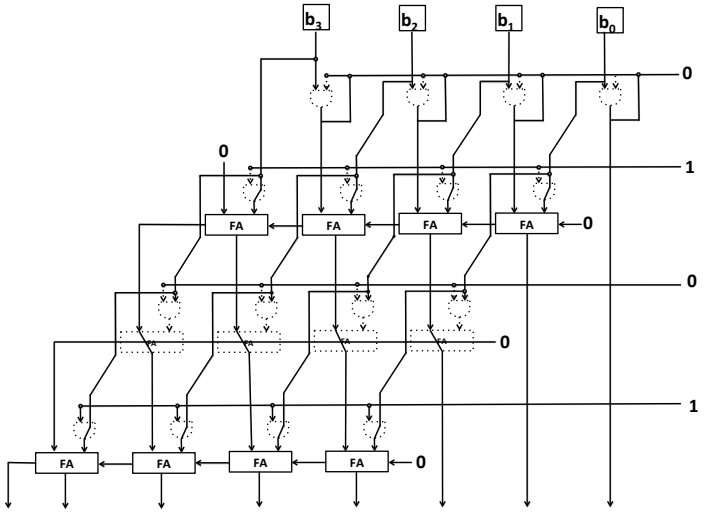


Figure 6.4: Illustration of a multiplier architecture with fixed input = "1010".

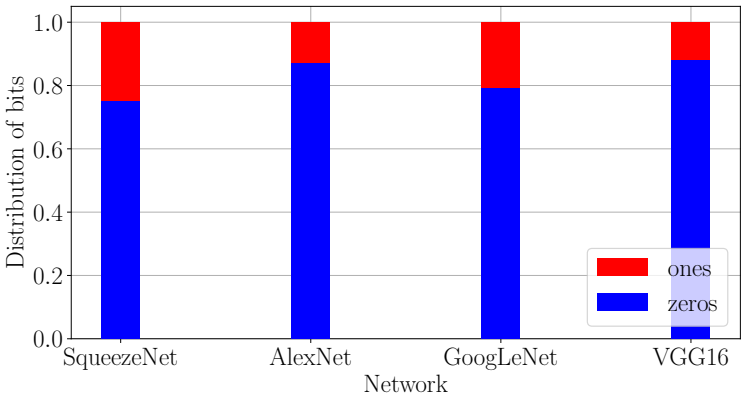


Figure 6.5: Number of 'zero' bits compared to 'one' bits (normalized) in different quantized CNNs using a 7 bits fixed point representation.

that the multiplier takes only one parameters instead of two. Therefore, since the first operations to execute in a convolution are multiplications, **the convolution layer does no longer need to access weights**; streaming inputs suffice to compute the output.

6.4.3 Weight-driven multiplier squeezing in CNNs

In order to estimate the expected impact of multiplication over-squeezing on performance, energy and resource utilization, we profiled seven fine-tuned networks, namely SqueezeNet, AlexNet, GoogLeNet and VGG. Figure 6.5 show the number of zeros and ones in the bits of the tested networks' weights.

As shown in Figure 6.5, about 80% of weight bits are equal to zero in most networks. We have already demonstrated that a 0 input to the multiplier results in removing the whole row and forwarding the previous intermediate results to the next row. As for the 20% remaining rows, and since the input is 1, we save this amount

of AND gates by skipping multiplications.

In the next subsection, we propose an experimental setup to validate these claims and quantify the gains in terms of resources, energy and delay.

6.5 Theoretical Performance Study

In this section, we propose an analytic study on the performance of the proposed implementation using the Roofline model [237]. An other analytical study of the memory requirements after applying our compression scheme is discussed earlier in this section.

6.5.1 Performance study

Computation and communication are two principal constraints in system performance optimization. An implementation can be either computation-bounded or memory-bounded. In [237], a roofline performance model is developed to relate system performance to off-chip memory traffic on one hand and the peak performance provided by the hardware platform on the other.

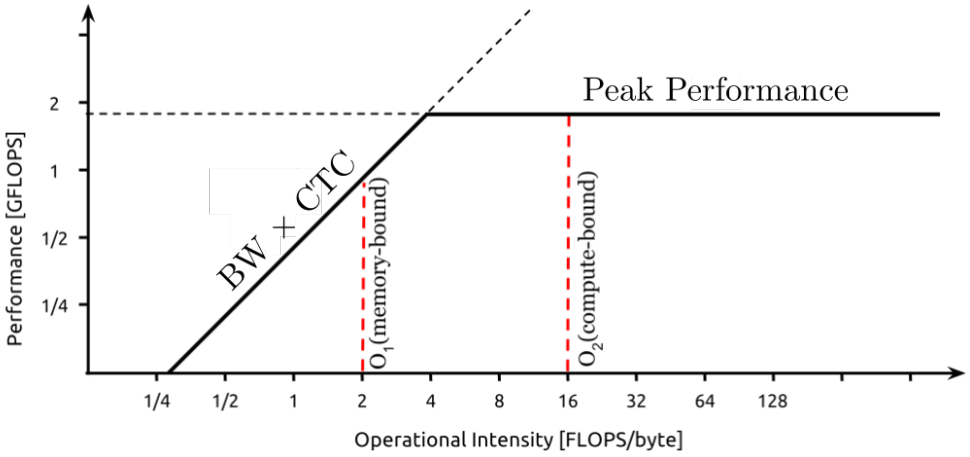


Figure 6.6: Roofline model for two applications O_1 and O_2. O_1 has less compute intensity and is memory bound since it has

In Figure 6.6, we show the relation between performance and compute intensity. We distinguish two types of applications, compute-bound and memory-bound applications. If an application (or an algorithm) reads one byte from off-chip memory for each operation, it has an operational intensity of 1. The lower this value, the more reliant the application on memory. A memory-bound application has very low operational intensity and is limited to the memory bandwidth, which is slower than that of the compute units.

Equation 6.12 formulates the attainable throughput of an application on a specific hardware platform. The number of operations per second GOPS is used as the metric of throughput. The actual performance of an application kernel can be no higher than the minimum value of two terms.

$$Attainable\_Perf = \min(Perf_{peak}, BW * CTC) \tag{6.12}$$

The first term, $Perf_{peak}$, describes the peak possible performance provided by all available computation resources in a hardware platform. Memory bandwidth is given by BW and the intensity of computing i.e. computation to communication (CTC)

Algorithm 4 Standard Convolutional Layer Algorithm

procedure CONV(**Mat**: $I[C, IH, IW]$, **Mat**: $W[K, C, R, R]$, **Mat**: $O[K, OW, OH]$)
  for $n = 0 \dots K$ **do** ▷ For each filter
    for $c = 0 \dots C$ **do** ▷ For each input channel
      for $i = 0 \dots OW$ **do** ▷ For each row
        for $j = 0 \dots OH$ **do** ▷ For each column
          $I_W = I[c, i \rightarrow i + R, j \rightarrow j + R]$ ▷ Extract an input window
          $F = W[n, c]$ ▷ Read the corresponding filter
          $O_{n,i,j} += I_W \circledast F$ ▷ Multiply and accumulate
        end for
      end for
    end for
  end for
end procedure

ratio, features the memory traffic required by a kernel in a specific system implementation. The second term bounds the maximum performance that the memory system can support for a given computation to communication ratio.

Notice that the CTC is given by Equation 6.13.

$$CTC = \frac{\#OPs}{\#Bytes} \tag{6.13}$$

Where OPs is the total number of operations to perform and $Bytes$ is the total number of bytes read that need to be read from the off-chip memory.

In this subsection, we use Equation 6.12 to compare the performance of a conventional implementation of a convolution layer with and without the proposed optimization. The pseudo-code of a baseline implementation of a convolution layer (from Equation 6.1) is given by Algorithm 4 where I is the input feature map, W the layer filters and O the output feature map. The operation $\circledast$ represent a dot-multiplication between two matrices followed by an addition.

We first compare the impact the convolution loop unrolling factor, and consequently the parallelism level, on the CTC ratio with and without our method. To avoid the data communication bottleneck, the data input is forwarded to different kernels to perform parallel convolutions. We consider the most commonly used kernel size in recent CNNs which is 3×3. It results in 17 operations, 9 multiplications and 8 additions per convolution operation. Hence, if p is the number of parallel kernels, the number of operations by iteration is equal to $17 \times p$. The number of accessed data is equal to 9 inputs, $9 \times p$ weights and p convolution outputs ($10 \times p + 9$ in total). The CTC ratio of a baseline convolution is hence given by Equation 6.14.

$$CTC_{Reference} = \frac{17 \times p}{9 + 10 \times p} \tag{6.14}$$

Since no weights communication is required in our method, the CTC ratio is then given by Equation 6.15.

$$CTC_{Compressed} = \frac{17 \times p}{9 + p} \tag{6.15}$$

Figure 6.7 visualize the above Equations.

We consider a real time inference where the CNN input is collected from PCIe link. Moreover, given the size of weights file, they are stored in DDR. Figure 6.8 visualizes the roofline model with computational roof and I/O bandwidth roof of a conventional CNN. In the same Figure we show the roofline model of the same CNN loop with our technique. The model is implemented for the Zynq UltraScale+ ZU9CG board and

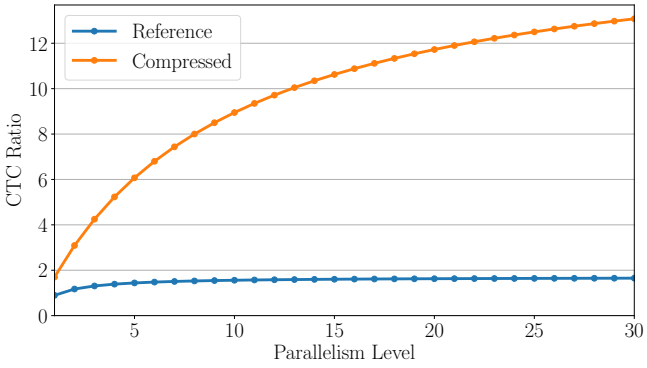


Figure 6.7: Comparison of the parallelism level on the CTC ratio with and without our method.

the operations are considered to be 8-bit fixed point. Since the memory bandwidth (BW) considered in the roofline model is the system slowest storage device [237], the slope corresponds to the DDR bandwidth for the baseline implementation. It corresponds to the PCIe bandwidth in our method since no weights access is required.

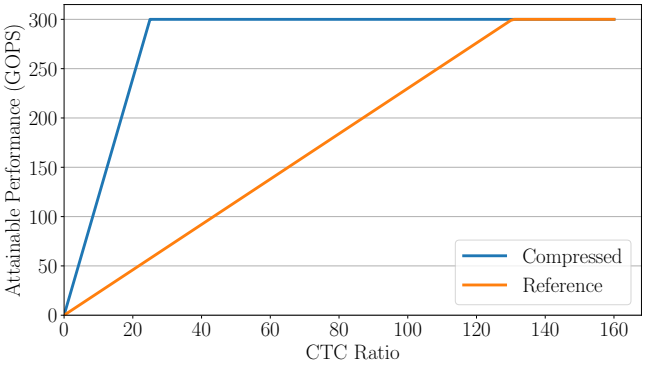


Figure 6.8: The roofline model comparison with and without our method.

An example to illustrate this difference is as follows. Let us consider an implementation with a $CTC < 20$ in the roof-line shown in Figure 6.8. By using compressed multipliers, the design would no longer rely on the DDR which holds the weights but will be limited by the PCIe from which the inputs are read. With higher bandwith, the slope at which the theoretical roof reaches the peak performance will change which allows more designs to reach peak performance. Secondly, for the same number of operations we require less reads and therefore shifts the new CTC to the right gaining a higher roof which results in a better performance roof.

6.5.2 Memory Resource Utilization

Recent optimizations ([143], [38]) spends the majority of their resources in storing and managing weights. In a CNN, weights are only used for multiplication. Since our multiplier implicitly store weights, we do not required any memory. When compared to other works on the literature, our work outperforms any others that rely on a weight storage memory. From an other perspective, memory is considered the main bottleneck of a system. By embedding weights into the compute units, a design using

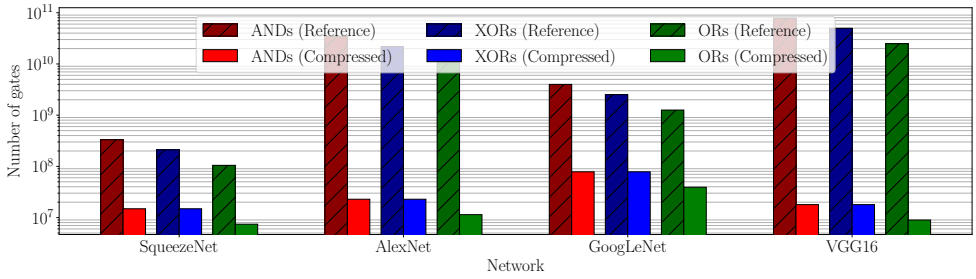


Figure 6.9: Total number of logic gates required before and after applying our squeezing scheme.

our compression strategy is not prone to this bottleneck. A memory is still needed to store intermediate results (activations).

6.6 Circuit-Level Experimental Results

As seen in Figure 6.1, the multiply accumulate operator is the computation bottleneck of the forward phase in CNN architectures. In hardware accelerators, the de-facto standard is using fixed-point implementations due to the huge reduction in delay, energy and resources compared to the minimal loss in accuracy [3].

In a given fixed-point multiplication between $A = a_3a_2a_1a_0$ and $B = b_3b_2b_1b_0$, the output is the result of the integer multiplication between A and B. The only difference is the radix point which is not represented in the number but can be computed in offline knowing the radix points of A and B. As an example, if $A = 11.01_2$ (3.25_{10}) and $B = 10.10_2$ (2.5_{10})[1], the output $C = A \times B = 1000.0010_2$ (8.125_{10}) can be obtained as the result of the integer multiplication between A and B and setting the radix at the fourth bit based on the input radix points $(2 + 2)$. The signed version is similar with an additional XOR operation between the two sign bits.

Hence, we conduct our experiments on fixed point networks based on an integer multiplier architecture such as the one presented in Figure 6.3. We evaluate the impact on resource utilization and delay by comparing a conventional implementation with the proposed compressed constant multiplier.

6.6.1 Experimental Setup

For the profiling results, we used the Caffe [105] description of SqueezeNet [98], AlexNet, GoogLeNet and VGG16.

For each of these networks, we used the already generated weights from Model Zoo [29]. Model Zoo is an online collection of already trained networks on famous datasets. We then used the Ristretto framework [80] to generate the optimal quantization parameters using the fixed-point representation. We used the fine-tuned network to compute the number of zero bits in weights and the required resources for each network. After this step, we noticed that the optimal parameter width in all of these networks is 7 bits.

We designed a 7×7 and a 7×15 multipliers using Orcad Capture and simulated them using PSpice. These multipliers are used to estimate the energy and delay of a single multiplication. For very accurate results, all input combinations are tested exhaustively in terms of delay and energy and the worst case is recorded.

[1] In this example, we ignore the sign bit since it does not impact the multiplication. Hence, the most-significant bit of each input is part of the number.

6.6.2 Results

6.6.2.1 Resource Utilization

In the first experiment, we compare the number of logic gates that are required to perform the multiplications of a given layer. By moving forward in the network layer-wise, we count the number of required resources in the multiplier and compare this number to the required resources given the actual weight after compression. Figure 6.9 shows the total number of gates required to implement the multiplication operations of the different networks. We notice that our approach achieves between 1 and 4 *order*s of magnitude of gain in terms of utilized resources. The relative gain differs from a network to another depending on its corresponding size as well as the respective weights distribution (see Figure 6.9).

6.6.2.2 Impact on delay

To estimate the impact on the delay of one multiplier, we vary the number of zeros within the fixed operand bits and track the propagation average delay. When applying our squeezing scheme, for each zero-bit[2] in the right operand in Figure 6.3, we save a row of full-adders. As for one-bits, we remove the AND gates used for multiplication which will save us at least the delay of 1 AND operation each time. When one or none of the bits are equal to 1, no carry propagation is required. Moreover, since ANDs are also not required, the theoretical delay is equal to 0 ns.

Given the average number n of zeros within 8-bit-weights, we define the average multiplier as the squeezed circuit that corresponds to a weight containing n zero bits.

In Figure 6.10 we show the estimated delay of the compressed multipliers. The number of zeros in the fixed operand dictates the circuit and is enough to estimate the delay of the multiplier. We compare the delay of each version to the conventional array multiplier. At worst, which is the case with all 'ones', the compressed multiplier is one AND gate faster since no multiplications are performed. When at most one bit is set to 1, the circuit contain no gates and the estimated delay is 0 ns. The difference of delays between the SqueezeNet multipliers and the VGG16 multipliers is due to the inputs size.

Figure 6.10 also shows the impact of our approach on these two corner networks. We highlight the average number of zeros per weights by considering the networks profiling results shown in Figure 6.5. As for SqueezeNet, the average number of zeros per weight is equal to 5 while it is equal to 6 for VGG16. Consequently, the most common multipliers are 1.8× faster for SqueezeNet and have 0 ns delay for VGG16. On average, this compressed version is almost 2× faster for SqueezeNet and more than 4× for VGG16.

6.6.2.3 Impact on energy consumption

We followed the delay experiments with a similar approach for energy. In Figure 6.11, we show the impact of the proposed approach on the multipliers energy consumption. For visualization purposes, circuits with 0 energy consumption (no gates, i.e. at most one bit is set to 1) are shown to have 10^{-2} nJ. We present the results of the two corner networks, SqueezeNet and VGG-16. As expected, the results are coherent with the number of resources afforded. In fact, for SqueezeNet, the energy consumption of the average multiplier is more than 20× less than the conventional circuit. More interestingly, the energy consumption of the average multipliers in VGG16 is equal to 0 nJ.

The average compressed multiplier in SqueezeNet is estimated to consume 23.85nJ per use as pointed by the arrow in Figure 6.11. Taking into account the number of multiplications in SqueezeNet, the estimated energy consumed by multiplications is 9.25 J per image.

[2] A bit that is set to 0. Similarly, a one-bit is a bit that is set to 1.

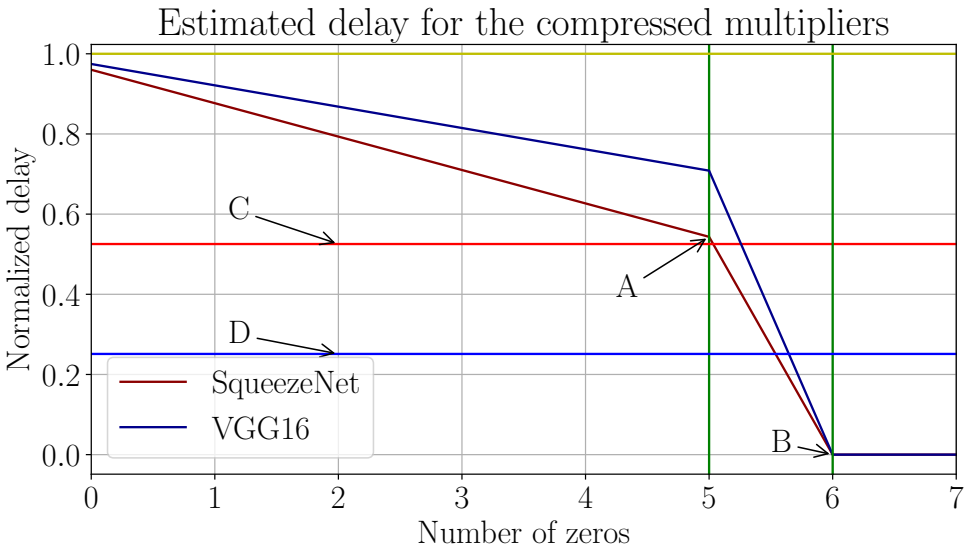


Figure 6.10: The impact of the number of zero bits in network weights on the multiplier delay. The two series represent the delay when using our compressed version (compressed) while varying the number of zeros in the constant operand. The vertical green lines are the average number of zeros from Figure 6.5 for the two networks. The intersubsection gives the delay of the most common multiplier. This is point A for SqueezeNet and B for VGG16. The lines C and D shows the averaged delay over the parameters of SqueezeNet and VGG16 respectively.

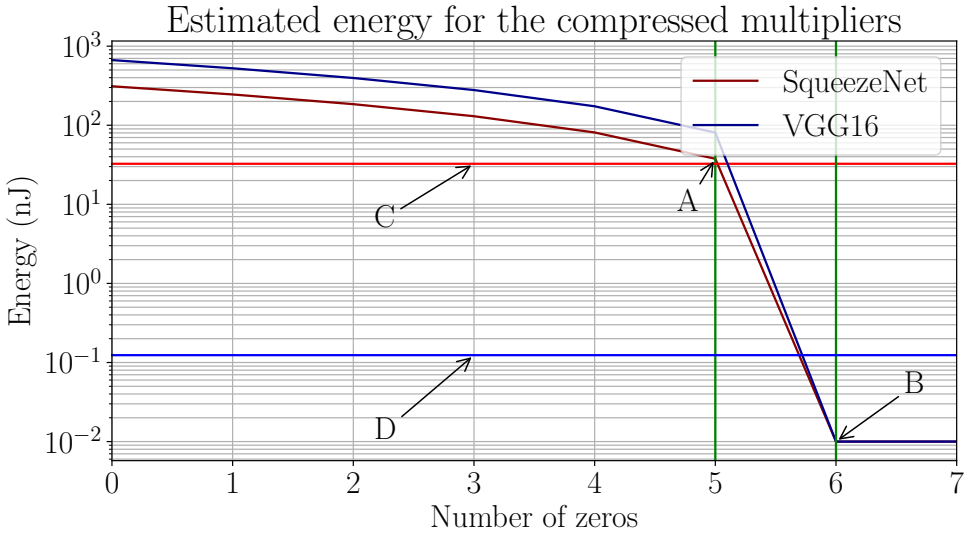


Figure 6.11: The impact of the number of zeros in network parameters on the multiplier energy consumption compared to the conventional multiplier. Similar to the delay, the two series present the decrease in energy consumption for each zero in the constant operand. The C and D lines shows the average energy consumption over the parameters of SqueezeNet and VGG16 respectively. A and B illustrate the energy of the most common multiplier for these two networks respectively. These values are obtained at the intersubsection between the average number of zeros (green lines) and the delay of a compressed multiplier with that number of zeros.

6.7 RTL-Level Experimental Results

In this subsection, we present experimental results of the proposed multiplier architecture in terms of energy and resource savings in FPGA. As established earlier, our idea only tackles multipliers, we do not propose a full-system architecture. Although a CNN inference run is dominated by MACCs, an implementation requires other types of operations that we did not discussed in this work. The difference with the previous subsection is in the implementation. A multiplier in FPGA is implemented either in DSP (hardwired) or using Look-Up Tables (LUTs). Gains from the squeezing scheme discussed in subsection 6.4 may be different.

Many production CNNs are quantized. This increases performance at a small accuracy cost. We used Ristretto [80] to generate fixed-point weights of 4 CNNs, namely, AlexNet, VGG16, GoogleNet and Squeezenet. Ristretto uses the dynamic fixed point representation from [47]. The fixed point multiplication is similar to integer multiplication with a simple XOR operation for the sign bit. For the 4 networks, we obtained weights quantized with 7 bits each an 1 sign bit.

We developed an automatic code generator[3]. In it, We generate a multiplier code that takes one input and performs multiplication with a fixed constant. We also create a TCL script that calls Xilinx's Vivado on the generated multiplier in order to synthesize and map it on a chosen Xilinx chip. In our study, we used the results for Xilinx's xc7z020clg484-1 FPGA.

The generator takes the bit-width generated by Ristretto [80] as parameter. For all *integer* values of the same bit-width, it generates a multiplier taking one input and performing multiplication with that value. For each generated multiplier, we record the number of used resources as well as energy. For fair comparison, we took the default Vivado implementation of the multiplication operator (*) in VHDL as reference. The reference multiplier only uses LUT slices since DSPs are not efficient[4] for low bit-width arithmetic.

It is worth to mention that our comparisons are performed at multiplier level. We only consider resources used and energy consumed between the input pins and the output pins. E.g, for a multiplication by 0, we need no resources and no energy since output pins are directly connected to the ground ; but we do need the same number of resources in the reference implementation since no information about the inputs are present at design, hence, all the components should be present.

6.7.1 Resource Utilization

The implemented multiplier uses only Look-Up Tables (LUT). Hence, we quantify resource utilization as the number of used LUT slices after implementation. We propose two comparisons. First, we compare the required resources for each constant value in one hand with the required resources for a conventional multiplication on the other hand. The conventional multiplication is the (*) operation in VHDL. This is achieved by generating all possible multipliers for a given bit width using our tool.

This study is performed at multiplier-level. It gives insight on how the compressed multipliers compare to the reference. The second study focuses on the impact at network level. We propose a fully-unrolled CNN implementation. In this configuration, we design as many multipliers as possible so that multiplications can be all performed in parallel. For FC layers, we implement as many multipliers as weights. However, since CONV layers have resource sharing, we duplicate weights in order to fully paralellize the execution.

Figure 6.12 shows the result of the first study. We take the case of a 7 bit multiplication and we generate the possible 128 possible multipliers. We present the distribution of used resources. At worst, 15 LUT slices are required per multiplication.

[3]The link is omitted for blind review purposes.

[4]Xilinx's DSPs of the Ultrascale have a 27×18-bit multiplier. For 7-bit multiplication, it is inefficient to use such multiplier. LUTs are more adapted resource-wise.

Special cases, such as 0, 1 and 2's powers require no resources and can be done by forwarding input bits to corresponding output bits which will also results in null delay an no energy. A conventional multiplication using Vivado's implementation requires 51 LUT slices since it at least needs to multiply each bit of the multiplier with each bit of the multiplicand.

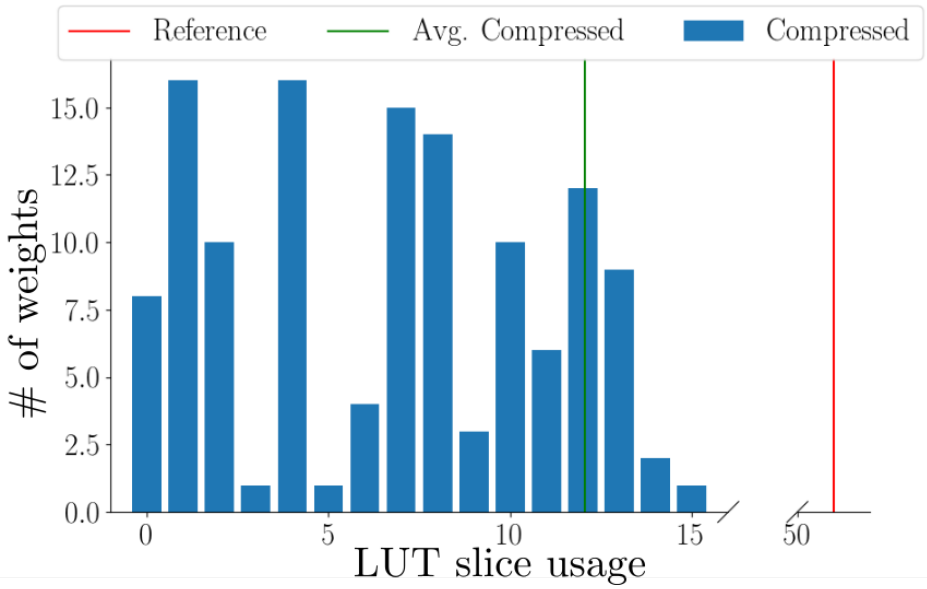


Figure 6.12: Distribution of resource utilization for different compressed multipliers. A comparison between the average of utilization over all possible128 multipliers (green line) and the utilization of the conventional implementation (red line).

In Figure 6.13 we show the impact at network level. VGG16 is the heaviest network in terms of weights. For this network, we managed to save 64.39× fewer LUT Slices when using the compressed version. For Squeezenet, the lightest network, we saved 11.44× slices. As mentioned before, we considered a fully unrolled implementation where all multiplications are done in parallel. In real life, this is achieved by creating as many multipliers as weights for fully connected layers and by duplicating the filters for each output and instancing a multiplier for each duplicated filter.

In our case, resource are related to the number of zeros in a network's weights. In Figure 6.13, we see this behaviour where the more zeros are present in a network, more savings are possible, hence, less resources are used.

Although the reference design is not obliged to duplicate as many instance as possible, the huge resource savings in our compressed version can cover this. The results shows more than an order of magnitude less resources than the reference implementation. This is enough to compensate for the inflexibility and the need to instantiate as many multipliers as operations. The advantage in this case becomes performance since our design will still be able to process all multiplication in parallel while a *not-fully unrolled* implementation have to wait between two usages of the same multiplier.

6.7.2 Power Efficiency

Power consumption is linearly correlated to resources used, hence, the tendencies in power consumption are almost identical to resources. In Table 6.1, we present normalized power consumption for all multipliers in each layer type (fully connected and convolution). We record more than 10000× less power when only considering multiplications for fully connected layers in VGG16 and Alexnet. This is caused by

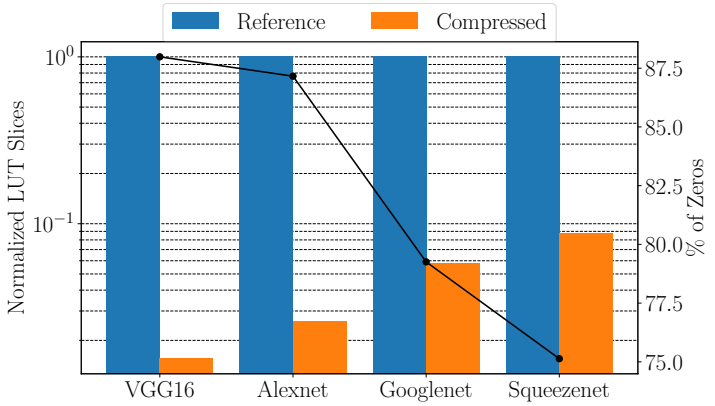


Figure 6.13: LUT slice utilization (left axis) for the reference and the compressed implementation per network. The correlation with the percentage of zeros (left axis) is shown with the black line for each network.

	VGG16	Alexnet	Googlenet	Squeezenet
FC	18656.94×	1960.99×	21.77×	/
CONV	70.71×	39.75×	18.70×	12.43×

Table 6.1: Energy savings relative to the reference implementation

the massive number of weights in these two networks. Gains also depend on the network architecture. CNNs like VGG16 and AlexNet are not optimized. GoogleNet and Squeezenet on the other hand have fewer weights. They have been subject to extensive architectural optimizations which narrows the compression space. This is visible in Figure 6.1 where our compression technique only achieves an order of magnitude in these latters. Notice that, Squeezenet have no fully connected layers by design[5].

6.8 Discussion

Our idea exploits an inherent aspect of CNNs which is constant weights to optimize CNN-dedicated multipliers implementation. While promising results have been shown, some limitations need to be addressed. In fact, the proposed approach is suitable for ASIC design as it implies optimization aspects at circuit level. This aspect makes our approach unsuitable for some reinforcement online learning neural networks that continuously adapt the network weights at runtime. While these networks are mainly used for prediction purposes, offline learning algorithms remain the most widely used in classification and recognition problems. As for FPGA platforms, their reconfigurable aspect allows coping with dynamic requirements of some machine learning algorithms or applications.

The technique is also a viable candidate for In-Memory Computing architectures such as Resistive Associative Processors [163]. The dedicated multipliers could be seen as a memory that is able to perform computations. This open the doors to a plethora of possible target platforms.

[5] 1×1 convolutions are introduced to play the role of fully connected layers.

6.9 Contributions and Guidelines

A novel way of compressing CNN accelerators is presented in this chapter. We exploit the constant nature of trained CNNs' weights to design an optimized implementation. Our approach is orthogonal to all previously proposed compression approaches as it operates at RTL and circuit level. Hence, it can be coupled with any other high level pruning or compression technique. The experimental study shows that the proposed approach achieves more than 90% saving in utilized resources within multipliers and at least 20× less energy consumption. Even though our approach is unsuitable for ASIC-oriented continuously trained CNNs such as reinforcement learning-based [14], a wide range of CNNs can benefit from our compression technique. Generalizing our technique to different hardware platforms is our future focus. We are making our VHDL code generation tool publicly available[6] for FPGA-based CNN accelerator implementations.

[6] https://github.com/267-OS822/ACM-TACO-2019

Chapter 7

Design for Reliability: A Fault Injection Study

Contents

Deep learning architectures for autonomous driving requires both reliability and performance [232]. Therefore, we focus our second aspect of the study on reliability for machine learning accelerators. In this chapter we propose an exploratory study on the reliability of CNN accelerators. First, we give a background on reliability and existing works that targeted reliability in ML algorithms. Then we propose two studies in which we answer two questions: Are CNNs reliable for safety-critical systems? and if so, What are the vulnerabilities of a CNN implementations?

7.1 Motivations

Because of the massive amount of data handled by CNNs, their implementation in embedded systems requires high performance Hardware platforms. While the literature is focusing on strategies for increasing the accuracy of CNNs and adapting them to new tasks, the reliability of CNN-dedicated hardware still remains under-explored. New hardware generations successively shrink transistors dimensions, thereby increasing circuits sensitivity to external events which can negatively affect their accuracy. One of the major sources of these errors in modern embedded systems are soft errors such as Single Event Upset (SEU) and Single Event Transient (SET). These events cause bit flips in the target chip. Depending on their duration, they can be upsets if they permanently change the bit or transient if they affect the bit for a short duration. These errors are typically caused by high energy particles striking electronic devices. These events can lead to bit flips in sequential parts and memory cells. This situation often leads to system level failures and violations of safety specifications. In safety-critical systems, incorrect values being unreliably computed represent a serious issue, as these systems must comply with strict safety standards [100]. These standards constrain the manufacturers with less than 10 failures in time (FIT) caused by soft errors. This translates into less than 10 failures over a billion hours of operation.

Deep learning architectures are known to be inherently resilient to faults [121]. While this tolerance is enough for everyday systems such as spam classifiers and general image classification, their usage in safety-critical systems such as autonomous vehicles may be more exacting. The extent to which these errors may affect the system must be carefully modeled and identified.

The impact of faults on layer activations can be compared to approximate computing. Chen et. al [40] have shown this effect by using approximate memories. Other approximating techniques involves fixed-point quantization of a CNN [134] which plays on bit-width of operators (activations and weights). However, faults have a random aspect that cannot be compared to approximate computing techniques where possible changes are defined.

Recently, the resilience to errors issue has been addressed in [201] where authors propose a method for predicting the error resilience of neurons in a deep neural network. However authors didn't consider multiple events neither bit flips within memory. Fault characterization in CNN-based embedded systems is important. Because of resource constraints, hardening approaches for these systems must incur a small overhead in terms of area and energy.

In the first work in Section 7.4, we study the impact of single and multiple soft errors on the accuracy of CNNs. By simulating errors in memory and in the outcome of hardware's calculations, we characterize the behavior of CNN models deployed on embedded hardware. In this way, we provide a holistic overview on the susceptibility of CNNs to transient errors.

Intentional attacks are an other potential source of faults. The widespread usage of CNNs led to the development of sophisticated attacks. Adversarial attacks are among these attacks. Malicious users could intentionally tamper with processed data to fool the network. While these attacks are limited to the input, they can be easily generalized to other parameters of the system such as the CNN' weights [139].

In the second work in Section 7.5, we consider random errors resulting from the

environment. These errors are simulated as bit-flips. Redundancy is a common solution to reliability issues. However, a systematic redundancy has high resource and energy overheads and is not suitable for limited-budget systems [180].

7.2 Soft Errors Primer

As sub-micron technology dimensions sharply decreased to a few nanometer range in commercialized ICs, the sensitivity of electronic circuits increased drastically. Hence, embedded systems are becoming remarkably sensitive to the environmental working conditions and thereby vulnerable to soft errors. These errors result from a voltage transient event induced by alpha particles from packaging material or neutron particles from cosmic rays [272]. This event is created due to the collection of charge at a p-n junction after a track of electron-hole pairs is generated. A sufficient amount of accumulated charge in the struck node may invert the state of one or multiple sequential elements, such as latches and static SRAM cells, thereby resulting in SEUs or multiple bit upsets (MBUs) [58]. The current pulses induced by the event may also corrupt combinatorial circuits thereby leading to one or more transient pulses being generated and propagated [58]. In past technologies, this issue was considered in a limited range of applications in which the circuits are operating under aggressive environmental conditions like aero-space applications. Nevertheless, shrinking transistor size and reducing supply voltage in new hardware platforms brings soft errors concern up to mainstream applications.

7.3 Fault Injection in Embedded Systems

Two types of fault injection were presented in [141]. The authors managed to achieve miss-classification after a series of careful bit-flipping. They report the loss in accuracy for the target class only. In our work, we study the impact on the overall accuracy. Furthermore, the authors assumed the injections are carefully selected whereas in our experimental setup, injections are performed randomly to simulate environment faults.

In [23], a physical fault injection on DNNs was discussed. In their results, authors reduced the accuracy of the DNN after injecting faults in the activation functions of hidden layers.

In [180], a method for estimating fault tolerance in ANNs is proposed. This method exploits redundancy of hidden units to increase the network's fault tolerance. In their results, a very high number of replications (more than 7) is needed to achieve complete fault tolerance. Our study locates the most vulnerable parts to reduce this overhead when redundancy techniques are employed.

In [6] authors discussed the inherent Partial Fault Tolerance (PFT) of neural networks. They proposed a modified neural network that offers complete fault tolerance. Their proposal handles fault injections in output layer bias, in weights between hidden-output layer, in hidden layer bias and in weights between input-hidden layer. The overhead of their modification exerts an order of magnitude more area and delay to reach complete fault tolerance.

The PFT of ANNs during training was discussed in [216]. The authors considered replication to enhance the PFT of a network. In [189], it was shown that only 17 bit-flips are required to corrupt a network such as Alexnet. Authors carefully selected the target bits to be flipped. In this work, we focus on random error injections on different levels: data representation, position in the representation and position in the architecture.

The inherent fault tolerance of networks has also been studied in [184]. However, the authors focused on relatively small CNNs. Their methodology is based on stuck-at

faults. Stuck-ats in feed-forward neural nets was also discussed in [181]. Replication was proposed as a solution to achieve fault tolerance.

Stuck-ats were also discussed in [259]. Authors studied the impact of faulty MAC units in the TPU's grid-like architecture. Their results show that with less than 0.006% fault rate, accuracy degrades dramatically. They also proposed two solutions by pruning and retraining. Their study only considers permanent errors in activations since they claim that memory errors could be mitigated by ECCs.

The reliability of object detection networks on GPUs has been studied in [57]. Their study was based on fault injection and exploited the error leaking potential between GPU threads. Our study is platform independent and the result could be projected to other embedded systems.

Common adversarial attacks have been discussed in [253]. Authors surveys the different input bit-flipping maneuvers to corrupt a network's output. The countermeasures to these attacks are also hovered and explained in their paper.

Adversarial attacks could be considered sources of faults. Authors in [139] presented an analysis on adversarial attacks on input, weights, activation functions and other network parameters. In their attack design, faults are injected to fool the network and reduce its accuracy. The authors also proposed a hash function to increase the resilience of a given DNN.

To the best of our knowledge, this is the first study that explores random fault injections in CNNs' weight memory considering the different quantization parameters, different data representations, the bit position and the layer of occurring faults.

7.4 Impact of Soft Errors on CNN's Accuracy

7.4.1 Proposed Methodology

Figure 7.1 outlines our error injection method. When an input I is received, an element $I_{i,j} \in I$ is selected at random following a non-uniform distribution. A single bit index is selected from the IEEE-745 single-precision floating-point format representation of the value $I_{i,j}$ (described in detail in Figure 7.2) by drawing from a uniform distribution $\sim \mathcal{U}(0, 32)$. Finally, this bit is flipped and the value of $I_{i,j}$ is substituted with its altered version $I'_{i,j}$. This process is repeated for n iterations.

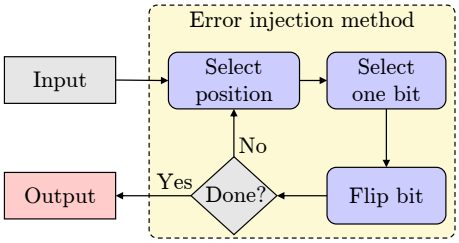


Figure 7.1: Overview of the proposed method for soft error injection.

It is important to observe that an error will not appear on every subsection of a Float32 with the same probability. There is a 71.87% chance for an error to appear in the fraction subsection, whereas in the case of the exponent, there is a 25% chance. The sign bit only has a 3.12% of possibilities to be affected. The relative impact of an error depends on the subsection where it's present, and also in the significance of this bit with respect to its subsection. Changes in the most significant bits of the exponent might be dramatic, while changes in the least significant bits of the fraction can be negligible.

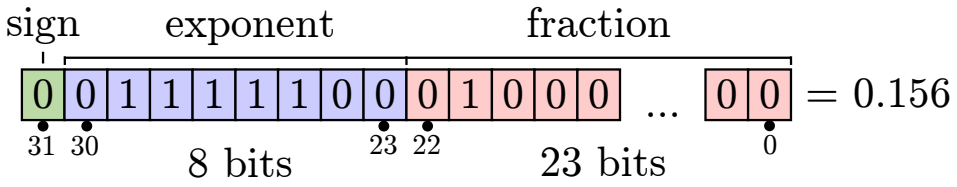


Figure 7.2: Description of the IEEE-745 single-precision floating-point format.

7.4.1.1 Error injection in layer activations

Let $\mathcal{A}_0$ denote the *golden run*—referring to an activation which doesn't contain any injected error—and let $\mathcal{A}_k$ denote the *k-th dirty run*—referring to an activation with k injected errors. It is possible to build a 2-layer model with only two levels deep to collect a large enough sample to characterize the distribution of the errors in $\mathcal{A}_k$ with respect to $\mathcal{A}_0$. This model is composed by two layers: an input layer and a KCWH convolutional layer with K number of filters, C channels, W filter width and H filter height.

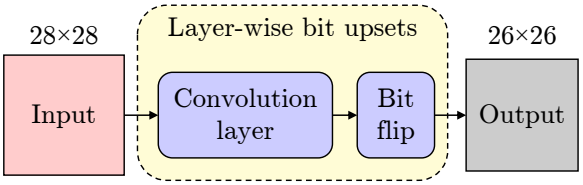


Figure 7.3: Our 2-layer model for simulating soft errors in activations is composed by a convolutional layer and our custom bit flip layer. The custom bit flip layer comprises a convolutional layer and a bit flip module applied to the outputs of this convolutional layer.

Figure 7.3 describes our 2-layer model for fault injection in layer activations. In it, an input tensor is subjected to a padded convolution to which a bit flip layer is appended. The output dimensions of the bit flip layer match $1:1$ the output of previous the convolution, with the only effect being the one described in Figure 7.1. It is possible then to parametrize the amount k of bit flips to be performed by our custom layer, and to generate different versions of $\mathcal{A}_k$.

7.4.1.2 Full model fault injection

The aim of this experiment is to quantify errors on a fully trained model. We study the drop of accuracy after fault injection, which is defined as the ratio between correct and incorrect classifications (or predictions).

We distinguish two targets of soft-errors: memory errors hitting the stored weights, and compute errors targeting the combinatorial circuit and corrupting the layer outputs. In both scenarios, injections may occur in two locations: memory, and compute units. In this study, we excluded non *von Neumann* architectures such as In-Memory Computing; for this type of architectures, errors can be more dramatic since they can affect both storage and logic at the same time.

For CNNs, weights are usually stored in memory. During the inference phase, compute units load an input and the respective weights of consecutive layers. This behavior is successively repeated for each new input. Based on this, we simulate memory errors as a random weight bit-flip event. In the case of compute errors,

we consider partial outputs (activations) as the event target (since each circuit has its own architecture), and low-level fault injection can not be performed unless an architecture is defined.

In the case of events affecting compute units, the impact is transient. In our model, training is done in a safe environment i.e. the generated weight file is considered perfect. Events happen during the inference part. In this phase, operations only depend on a given input and the network parameters[1]. Future behavior is not impacted since a new input is given each time. In our simulation model, given a probability, events will only affect the results (multi-class classification accuracy) of that given run with no history on previous events.

Memory events are permanent[2]. If a fault caused by a given event manages to flip a bit in memory, corrupted data will be stored for future use. In order to simulate this behavior in our model, we consider the impact of successive events each of which causes a bit-flip that should be seen by future events. Hence, our simulation results for this type of errors focuses on the cumulative aspect. It is worth to mention that applications used for space operations can run for years in *very* aggressive environments and therefore this impact should be anticipated.

7.4.2 Experimental Setup

Initially, three different experiments are created as part of an ablation study, in order to understand the effect that error injection can have on the performance of full DNN models. The first experiment consists on characterizing the effect of logical errors in the activation of a convolutional layer. With this purpose, a 2-layer model composed by a convolutional and our custom bit flip layer is assembled, and a number of activations are computed in order to characterize the distribution of the error defined as $\mathcal{A}_k - \mathcal{A}_0$. In the second and third experiments, we track the impact of transient errors on the overall performance of the DNN. We consider single and multiple event transients on combinational circuits as well as single and multiple event upsets in memories. This is translated in the CNN context by bit corruptions in the activation functions on one hand, and bit flips within neuron weights on the other.

7.4.2.1 Simulation environment

Our method for fault injection was implemented in Python 3.6 with NumPy, and DNN were built with PyTorch 0.4 over CUDA 9.0 and CuDNN 7.1.

Experiments on layer activations were run on Intel i7-6850K (15M Cache, 3.80 GHz) with 32GB RAM and NVIDIA Titan X Ultimate Pascal GPU 12GB GDDR5X. We used a fixed input size of 28×28, and the convolutional layer had 1 filter of size 3×3 with stride 1. Output activation was of size 26×26 and a rectified linear unit (ReLU) was used as activation function. Results were collected from 1 million sample activations with different parameterizations for k.

Full-model study was run on an Intel i7-6500U with 8GB of RAM running Python 3.5.2 with PyTorch 0.4. Experiments were performed on LeNet network trained for 16 epochs on the MNIST dataset. We used the Adam with a learning rate of 0.002. For backwards error propagation, we used cross entropy loss as criterion.

7.4.2.2 Dataset

In full-model fault injection experiments, we focus on characterizing the performance of the network on multi-class classification tasks. For this, we use the well known MINST dataset [130]. MNIST is a dataset of handwritten digits consisting of 70,000

[1]Network parameters (weights) are stored in memory and are not affected by the event since it hit the compute unit.

[2]We suppose that no safety measure are present during the simulation. If memory is operated with an error correcting code, this can be mitigated and no errors are permanent.

gray-scale images (28×28 pixels) divided into 10 classes, with approximately 7,000 images per class (Figure 7.4). There are 60,000 training images, and 10,000 test images (sets are balanced).

Figure 7.4: Example images from the MNIST dataset.

7.4.2.3 DNN model

In our full-model experiments, we study the effect of noise on the well known LeNet5 architecture [129], when performing multi-class classification over the MNIST dataset. Our choice of network was motivated by the following reasons:

- Available benchmarks: As one of the most widely studied networks, its performance is well documented and understood. Many variants improved on it but these modification are minimal and does not change much in the general structure.
- Possibility of generalization: LeNet is a modern predecessor to contemporary CNN architectures. The layers it contains are used almost every CNN for multi-class classification (VGG, GoogLeNet, ResNet, etc). Thus, it is possible to generalize the obtained results to other topologies.

7.4.3 Experimental Results

7.4.3.1 Layer-wise Analysis of Multiple Event Transients

In this experiment, we focus on characterizing the distribution of of the error between golden activations and faulty activations, defined as the difference $\mathcal{A}_k - \mathcal{A}_0$. For this, we build a 2-layer model that takes 28×28 input tensors and produces a 26×26 output resulting from an unpadded KCHW convolution, where $K = 1$, $C = 1$ and $H = W = 3$, terminated by a bit flip layer. This model is described in Figure 7.3. For this experiment, $k = \{1, 2, 3\}$ are selected in order to observe how increasing levels of corruption can affect the produced activation.

It is worth noticing that the soft error rate is estimated once in 10^9 hours of operation. Hence, scenarios with two or three bit flips are relatively rare. Nevertheless, this issue is getting more frequent with new technology nodes especially when deploying DNNs in aggressive operating conditions such as high-radiation environments, as multiple bit-flips can happen with a single particle strike [56].

Figure 7.5 displays the distribution of $\mathcal{A}_1$ over 1 million activations under single bit flip injections. We observe that the activation errors are visibly centered around zero, with 90% of activations being of small size. This is consistent with the statistical expectation of the error, as comes defined by the disposition of the bits with respect to the IEEE-754 format (as seen in Figure 7.2). In 10% of the cases, the error produces a very high and distinctive activation in both positive and negative direction.

Figure 7.6 displays a similar pattern to Figure 7.5, without any distinct feature indicating an increased accumulation of large errors. In this case, the difference leans towards errors greater than zero, but this may be considered an effect of statistical noise. Again, the proportions of the errors are the same with 90% concentrated around the boundaries of zero, and with a 10% clearly showing remarkably high and low alterations.

Figure 7.7 exhibits a similar behavior to the previously described experiments. In this case, we see a larger accumulation of negative errors but 90% are still around zero. It is interesting that in this case, the error doesn't seem to grow out of control

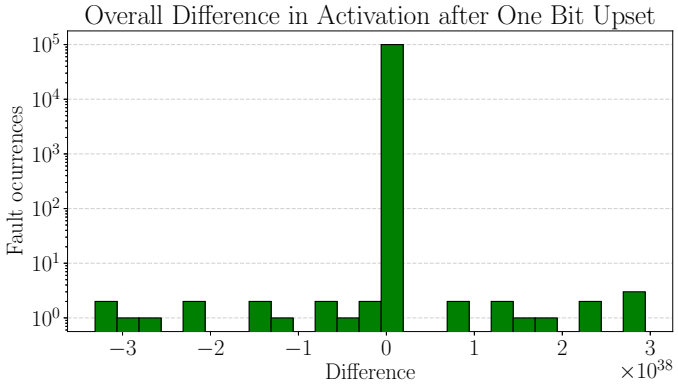


Figure 7.5: Distribution of $\mathcal{A}_1$ over 1 million activations in logarithmic scale.

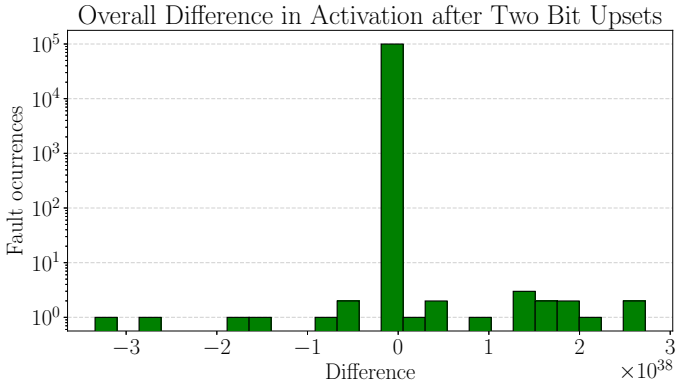


Figure 7.6: Distribution of $\mathcal{A}_2$ over 1 million activations in logarithmic scale.

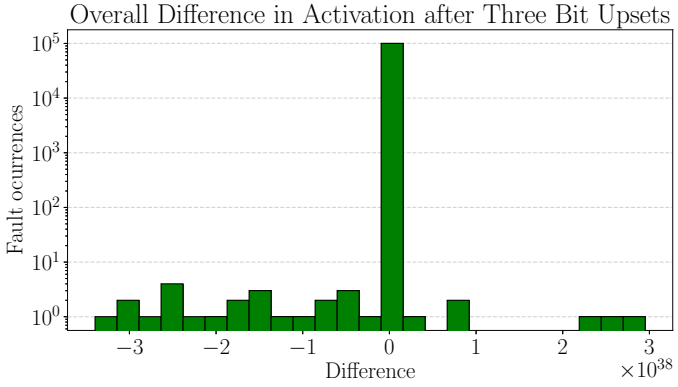


Figure 7.7: Distribution of $\mathcal{A}_3$ over 1 million activations in logarithmic scale.

even with 3 bit flips by occurrence. This is clearly due to the fact that alterations in the fraction part of a 32-bit float might carry a low impact in the final error.

7.4.3.2 Impact of soft-errors on model performance

This experiment evaluates the performance degradation caused by soft-errors on a trained DNN while performing inference. At first, we train LeNet5 obtaining an accuracy of 0.987% in the test set. The resulting weights are used for subsequent tests. We simulate soft-errors hitting compute units (SET) and memory (SEU).

7.4.3.2.1 Single/multiple event transients We consider the intermediate output of each layer in order to simulate bit-flips inside logic-units (PEs). We insert the previously defined injection layer at each step and record the obtained accuracy while increasing the fault injection probability. For each probability, we measure the average accuracy on the test set of 10k samples.

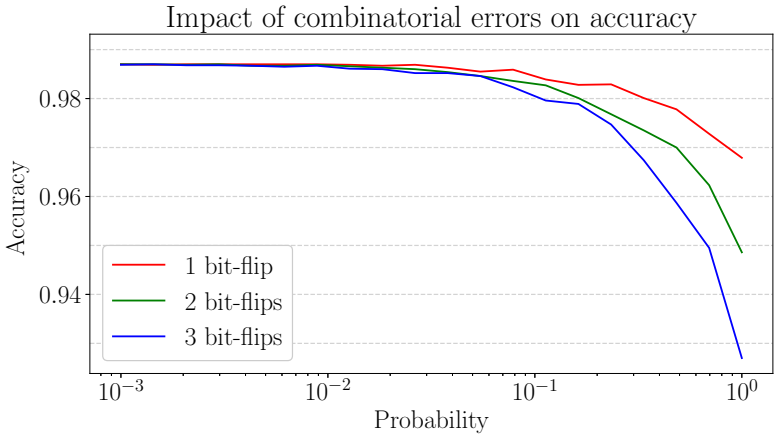


Figure 7.8: Comparison between single and multiple event transients and their impact on overall LeNet-5 performance.

Figure 7.8 illustrates the drop of accuracy of the aforementioned network (Section 7.4.2.3). As expected, the negative impact of injecting multiple errors is higher than single error injection. The accuracy loss starts at a fault injection probability of 10^{-2} and accelerates exponentially. However, even with multiple fault injections in combinational circuits, the overall accuracy degradation is not dramatic. In fact, with a probability of 1.0, and for 3 bit-flips per event, we noticed a drop of 5% in accuracy. This is not only due to the alterations in the fraction part of a 32-bit float, but also to the CNN properties. In fact, the complex structure of the layers help masking errors and preventing them from causing overall system failure. Practically, these capabilities of neural networks make them inherently robust to SETs.

7.4.3.2.2 Single/multiple event upsets This subsection explores the impact of single and multiple event upsets on the overall network accuracy. Although both SETs and SEUs are caused by the same *transient* phenomenon, there is a fundamental difference between them. In fact, while glitches caused by SETs are volatile in time, SRAM cells are bistable elements and a bit flip leads the cell to a new stable corrupted state. This means that the event is rather irreversible and by consequence the error lasts in time. This is translated by a cumulative aspect of the errors that needs to be considered in simulations. Therefore, to assess the impact of memory bit flips, we inject faults cumulatively within network weights and track the model accuracy. Injections are simulated as random bit-flips in the weight file (after training). Since memory errors are permanent, we keep previous errors after each injection. We measure the overall accuracy on the test set after each injection. This experiment is repeated multiple times for more accurate results.

Figure 7.9 shows the effects of memory errors that correspond to alterations in weights on the network accuracy. We notice a relatively slow decrease over cumulated errors. The main impact can be viewed in extreme cases. As we can notice, the best case scenario (maximum recorded accuracy) stagnates at almost perfect accuracy. However, the worst case scenario can transform the network from an almost always right to an almost random number generator. Before the 10 error step, we notice that the network can instantly drop below acceptable accuracy. This is explained by the

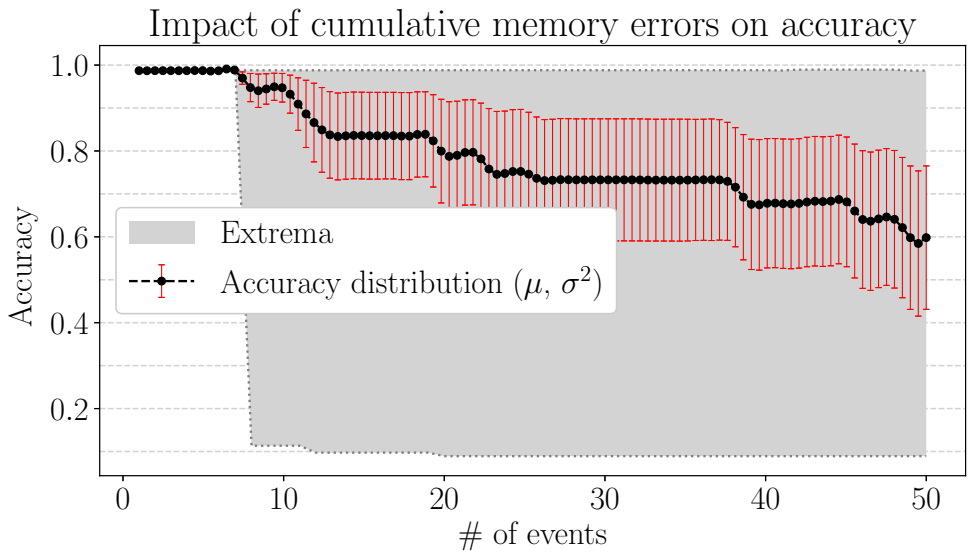


Figure 7.9: Evolution of average accuracy (in black) when the number of errors augments. The dispersion and the extreme cases are shown in red and gray respectively.

nature of weights and their relative importance when determine the network's final decision. For a given layer, the order of magnitude of weights is relatively similar. A random error can easily disorient this order which gives the neuron concerned by this weight a huge impact on the output.

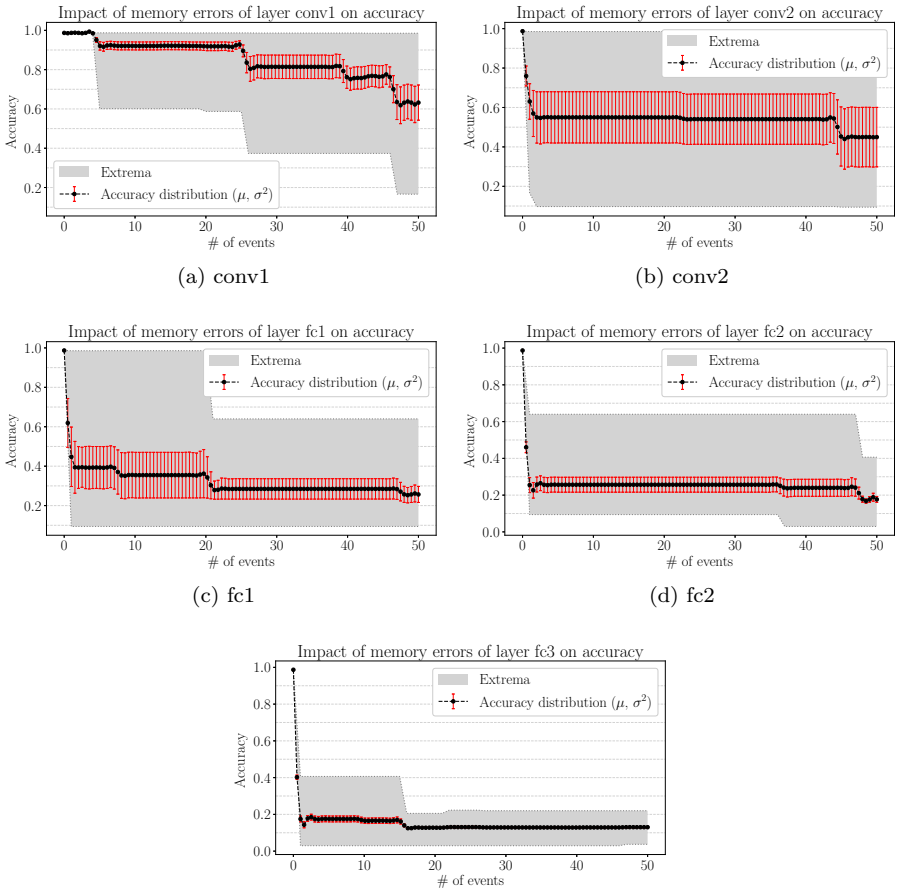
The drop of accuracy continues until reaching the level of 10%. This was reached very rapidly in the worst case scenario. At this level, the network is either outputting the same number each time or outputting random numbers. The first happens if one output neuron get affected and receives a huge boost. This will manifest as a neuron giving the maximum confidence each run. If many weights are altered, which is the case after a long time time of execution, the network output will be random giving it a chance to correctly predict the number 10% of the times.

The impact of memory errors is not equal between layers. Some layers are more critical if exposed to the same amount of faults. In the next experiment we evaluate the responsibility of each layer on the overall network performance. Injections are performed on weights like the previous experiments, however, we isolate a single layer from the network as a target for errors. Other layers use the trained weights and outputs correct values for given inputs.

The difference of singular impact can be viewed in Figure 7.10. The top-left Figure 7.10a shows that the first layer is the most resilient when compared to others. The average accuracy drops very slowly over time while errors corrupt the memory. Convolutional layers (Figure 7.10a, 7.10b) does not have much impact when compared to fully connected layers (Figure 7.10c, 7.10d, 7.10e). This is due to two main reasons:

- Fully connected layers classifies the extracted features into most suitable classes. Errors in in these layers leads to direct impact on the output. On the other hand, convolutional layer weights are feature extractors. The output of these layers is then classified by fully connected layers. Errors can be masked during the classifying mitigating their impact.

- Convolutional layers are followed by maximum-pooling layers in LeNet. If an error reduces the value of a number which is not the local maximum, this has no effect on subsequent layers and the error is masked. Fully connected layers lacks this follow-up and therefore suffers from this error.



(a) conv1 (b) conv2

(c) fc1 (d) fc2

(e) fc3

Figure 7.10: Layer-wise analysis of memory soft-errors with independent responsibility. Average accuracy is in black circles with the standard deviation in red. The interval of extreme cases (minimum and maximum) is illustrated with the gray region.

Extreme cases confirm these same hypothesis. We can see the severe impact of errors in the last layers where in less than 10 errors, the best recorded accuracy is below 40%.

7.4.3.3 Discussion

Progress in approximate computing shows that CNNs are, to some extent, immune to small error-driven deviations. We can exploit this fact to optimize the protection of floating point data. In this setup, errors affecting the mantissa should get less interest than exponent since their impact will not change the order of magnitude. On the other hand, limiting the output of each layer[3] can mitigate the effect of errors on the exponent. Errors in sign bits have a direct impact on the data integrity and should be implemented on hardened memory cells.

The conducted experiments on errors targeting the combinatorial part (SET/Multiple Event Transient (MET)) shows a drop in accuracy. However, for a very exaggerated probabilities, the network kept a very high average accuracy. As explained, the reason is the short lifetime of the error. The impact is momentary and only limited to the current inference; and while it has a chance of totally giving a false classification

[3]The upper bound can be fixed by maximizing the activation values of each layer obtained during the test or the validation phase.

or prediction, .

In contrary to errors striking the compute part, memory faults are more dramatic. One memory error affects all future computations. In a CNN, weights constitutes the most important data and, if not protected, may cause severe consequences.

Based on the last two observations, designing reliable CNNs for aggressive environments should focus on protecting the weights memory. Storing weights in secure or redundant memories or implementing an ECC is a design must. Moreover, it is worth noticing that latter layers have the biggest impact and should be specifically protected.

CNNs fed by a continuous source of data such as a video stream outputs a classification for each frame. Therefore, having occasional miss-classifications is not much problematic due to the possible recovery in subsequent runs. In this case, compute units require minimal attention since rare events are masked.

7.4.4 Summary and Design Guidelines

In this work, an exhaustive exploration of the transient errors impact on deep neural networks, notably CNNs, was presented. While deep learning is getting drastic interest in mainstream and critical applications, reliability issues have not been sufficiently addressed. Since these algorithms may replace humans in systems dedicated to operate in aggressive environments, having a deep insight on their reliability is an essential concern.

Although the regularization abilities of deep neural networks provide them with an added degree of robustness against external disturbance, our empirical findings add nuances to this widely admitted claim. In fact, while errors in processing elements do not have a huge impact on the overall system accuracy, a drastic degradation was noticed with faults occurring in memory.

A corruption in classification layers is more accuracy degrading than feature extraction layers, as their relative impact in the final decision of the network is more significant. Therefore, reliability enhancement techniques should be deployed since some errors can dramatically and irreversibly corrupt the network behavior.

In future work, the focus on developing new strategies to increase the robustness of DNNs in aggressive environments, as well as gaining an understanding on design topologies that can be deployed in autonomous vehicles stationed in low Earth orbit, where increased radiation levels are a very significant concern.

7.5 CNN Fault Injector

7.5.1 Experimental Methodology

In this subsection we present our setup and methodology to evaluate the reliability. We use the same methodology from [164] with different experimental setup. When compared to the previous work in Section 7.4, we evaluate more variables and confirm the obtained results on other networks.

7.5.1.1 Methodology

Without considering physical damage, soft errors compromise system functionality by causing bit-flips in memory or in computational elements. Since memory errors are more critical and durable [164], we only focus on bit-flips in memory. In most machine learning accelerator designs, two memories are present: 1) the weights memory ($\mathcal{M}_w$) which stores trained network parameters and 2) intermediate output memory ($\mathcal{M}_i$) which stores the output of hidden layers.

$\mathcal{M}_i$ receives new values for each input. A bit-flip in this memory will only affect the current run, and, only if it occurs before the subsequent layer starts processing.

This is similar to errors in computational parts which we will not study. On the other hand, a bit-flip in $\mathcal{M}_w$ will remain active until a new network is deployed. We focus on this kind of errors.

To reproduce this behaviour, we simulate a soft error in $\mathcal{M}_w$ by a number of bit-flips in a random weight once the network is trained. Multiple studies are conducted based on this simulation hypothesis. In each study we evaluate a robustness variable of a CNN based system. The position of the flip is decided by the study and the evaluated variable.

7.5.1.1.1 Networks CNNs can perform a variety of tasks. Whether it is for images, voice, text or other input types, classification is the most common task performed by CNNs. Other tasks, such as detection, uses a classification sub-network. In the experimental setup we propose, we only consider classification networks. Consequently, the study can be projected to other variants.

7.5.1.1.2 Dataset Measuring a CNN's accuracy requires a labeled testset. Since testsets are usually not labeled, we use the validation set of ImageNet as used in the challenge. The set contains 50000 image with the correspondent class of each image. The set contained 1000 classes.

7.5.1.1.3 Data Representation We consider two data representations:

- IEEE-754's 32-bit float: This is the standard representation format for floating point format. It is the dominant representation in CPU and GPU architectures. Many GPUs are optimized to deal with floating point multiplications. For simplicity we refer to this representation as $\mathcal{F}$ in the rest of this section.
- X-bit fixed point: We used the format from [48]. Trading accuracy for high performance by using low bit-widths is a common practice in CNN acceleration. This representation uses two parameters: bit-width and fractional length. Negative fractional lengths can be used to represent powers of two. This representation is referred to as $\mathcal{Q}$ (for quantized) in the rest of this section.

7.5.1.1.4 Injection Algorithm Based on the fault-injection model in [164], we create a fault-injection engine. The engine takes a trained network, a dataset and a test type. The test type dictates the execution flow and the parameters to vary during the test. Multiple test types are developed, more details are provided later this subsection. Depending on the selected test type, a series of bit-flips are performed in the network's weights. After each test, the engine reports the measured accuracy on the dataset after the injection.

We consider three test types: *full-network*, *index-wise* and *layer-wise* tests.

- *Full*-network injection: the engine generates a list of errors that are identified by their layer and the position in the layer. The engine incrementally injects errors in the network. After each injection, we measure the accuracy on the whole dataset. As a result, we aim to compare the two data representations in terms of inherent resilience. This comparison is useful to decide which data representation is more suitable when faults are present.
- *Indexed* injection: in this test, the generated errors are injected in a fixed bit significance. The engine then loops over every possible position from the least to the most significant bit[4]. The result of this test type extends on the result of the *full* test. After comparing the two representations, we use this study to explain the difference, if any, between the obtained accuracies. Furthermore, this helps localizing the most vulnerable bits to protect.

[4]In the case of a trained network represented as 32-bit floating point, the engine loops over the 32 bit positions.

- *Layer*-wise injection: in this test, errors are generated in the same layer with different positions. This test is repeated for each layer whilst reporting the accuracy after each run. The number of errors injected is proportional to the number of parameters of each layer with a single injection in the layer with the fewest parameters. This is similar to the real world where the soft-error rate is proportional to the surface of the chip. This study allows us to understand the inherent tolerance of CNNs layers. Finding the most vulnerable layers will assist in creating comprehensive reliability enhancement strategies.

These tests are repeated 60 times to reduce probabilistic variations. In each run, the engine generates a new set of errors and the injection of the generated errors is performed each run. We then present the mean of the 60 runs as well as the maximum, minimum and the standard deviation of the test.

A single soft error can cause multiple bit-flips. Furthermore, memory errors are cumulative. We fix the number of errors to be injected when varying the index of the bit-flip to 50[5]. For layers, we inject errors proportional to the number of parameters with at least 1 injected error[6]. An extensive study, with variable number of errors, is possible, however, the same tendency reappears.

7.5.1.2 Experimental Setup

The engine was developed on python. For CNN inference, we used the framework Caffe. The code is made publicly available[7]. As part of our study, we perform injections on quantized (low-precision) CNNs. The weights were obtained using Ristretto.

The experiments were performed on an Nvidia Quadro P5000 GPU with an Intel(R) Xeon(R) W-2123 CPU with 3.60GHz frequency.

We used four network architectures: GoogleNet, Alexnet, VGG16 and SqueezeNet. These networks were selected for their wide usage, diversity, various sizes and high accuracy. Their convolutional layers are widely reproduced as feature extractor in other models. This facilitates generalizing the obtained results to other networks.

For the 32-bit floating point represented weights, we used trained instances from Caffe's Model Zoo[8]. We used Ristretto to quantize and fine-tune the four networks into 8-bit fixed point networks without huge loss in accuracy.

7.5.2 Experimental Results

The results we collected from the engine are presented in this subsection. For each test type (*full*, *layer* and *index*) we show the obtained results separately.

7.5.2.1 Impact of Data Representation and Quantization

The results were obtained on weights represented as 32-bit floating point. We present a comparison between the impact of different data representations on the accuracy of the different networks.

Figure 7.11 illustrates the result of comparing the two representations $\mathcal{F}$ and $\mathcal{Q}$. The $\mathcal{Q}$ representation is clearly more resilient than it's counterpart. This tendency is present for the four networks with different rates. The theoretical reason behind this resilience is explained by the overall difference after injection denoted by $\mathcal{A}$ in [164]. For instance, the $\mathcal{Q}$ representation with 7 decimal bits and 1 integer bit will differ from the original value by at most ± 1. For the $\mathcal{F}$ representation, the difference on activations can reach 3×10^{38} [164].

[5] It is worth repeating that event upsets are not only originated by the environment but can be intentionally provoked.

[6] Scales with the size of the target layer.

[7] https://github.com/cypox/CNN-Fault-Injector

[8] Publicly available on: https://github.com/BVLC/caffe/wiki/Model-Zoo

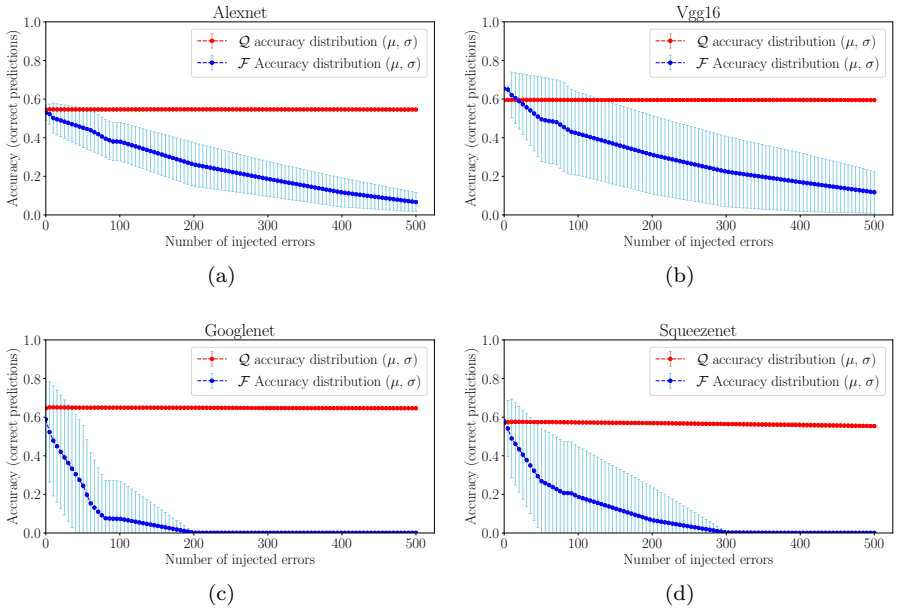


Figure 7.11: Comparison between the 8-bit fixed point representation ($\mathcal{Q}$) of weights and the 32-bit IEEE-754 representation ($\mathcal{F}$). The results of different runs are presented as the mean and the standard deviation of the top-1 accuracy.

Network	Alexnet	VGG16	Googlenet	Squeezenet
Weights ($\times 10^6$)	60.97	138.36	7	1.25

Table 7.1: Number of weights (in millions) per network.

The decrease in accuracy in VGG16 and Alexnet is not as fast as the decrease for the same number of errors in Googlenet and Squeezenet. The main reason for this phenomenon is the number of weights as shown in Table 7.1. The same number of errors have less impact if the number of weights is important.

7.5.2.2 Significance of Bits

To further explore this decrease in accuracy, we investigate the individual impact of the bit position. The injections are performed at the same position on the four networks each run. The only difference being the index of the bit-flip on the binary representation of the weight. We performed this study only on the $\mathcal{F}$ representation. The $\mathcal{Q}$ representation in invulnerable to bit-flips as shown in the previous results in Figure 7.11.

In Figure 7.12, the four networks have the same tendency. Unless bits are injected in the exponent's most significant bit, almost no impact on the accuracy is perceived.

7.5.2.3 Layer Tolerance

The impact of injected faults may depend on its location within the network architecture. This subsection explores the layers tolerance aspect. Similar to subsection 7.5.2.2, we isolate the target layer in the fault injection process. This isolation allows to track the individual impact of the chosen layer on the overall accuracy.

Googlenet and Squeezenet have a special architecture. They are built on top of two modules, *inception* for the former and *fire* modules for the latter. These modules regroup a set of convolutional layers working in parallel on the same input. The output of the module is obtained by concatenating the outputs of each execution

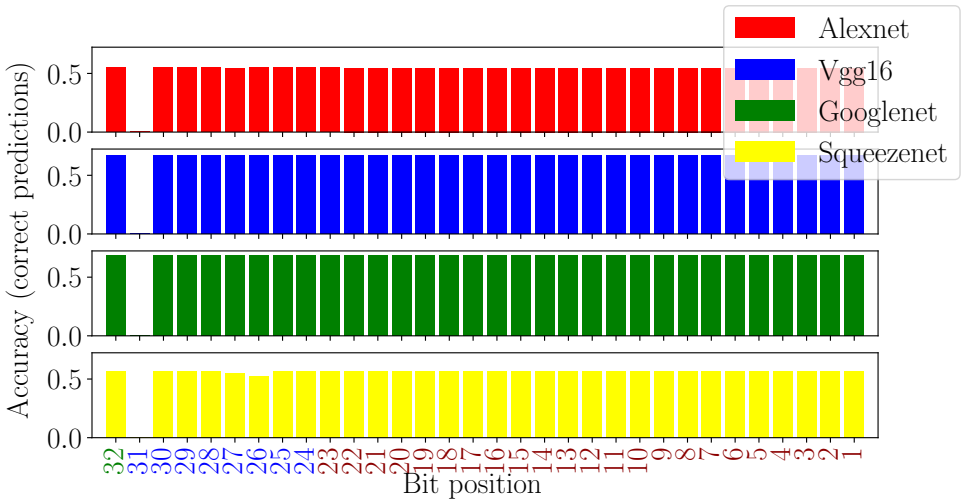


Figure 7.12: Position of bit-flips in the $\mathcal{F}$ representation and its impact on the accuracy. In the X-axis, red labels represent the mantissa, blue labels represent the exponent and the sign bit is in green.

branch. For clarity, we reduced the individual layers into the corresponding modules. For each module, we take the average accuracy of its individual layers.

Figure 7.13 presents the results of this study. The four networks tend to lose more accuracy when injections occur in advanced layers. This is correlated to our previous results in [164]. While CNNs have a sequential structure, error propagation is not problematic in CNNs. Errors in early layers have, in general, less impact on the accuracy. This shows the implicit characteristic of CNNs to maintain a sane behaviour when incorrect values are forwarded. This is explained by the implicit redundancy in CNN weights. After training, many weight clusters are repeated. A small number of errors is algorithmically masked if it occurs in the first layers. Techniques such as pruning can greatly affect this study. It explores weight redundancy to reduce computations, hence, augment performance. Errors that can be previously masked by redundancy would no longer be harmless. While it achieves high throughput with acceptable accuracy, reliability can be greatly compromised [202]. This is because reducing the number of redundant weights increases the *importance* of each individual non-pruned weight. This trade-off should be considered to evaluate CNN acceleration in aggressive environments.

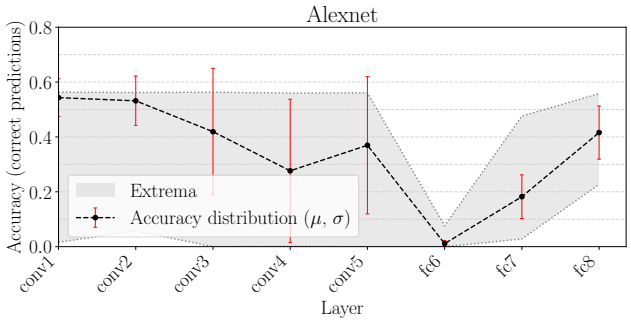
It is worth to mention that, although the mean value is at a comfortable accuracy, the minimum accuracy reported is almost always $\approx 0\%$[9]. This means that in some runs, the injected errors were able to fully compromise the network. As rare as it could be, anticipating these cases by studying the network should precede any deployment. Also, the fact that a few number of errors can damage a network this far is an other motivation to deeply study the impact of faults.

7.5.3 Discussion

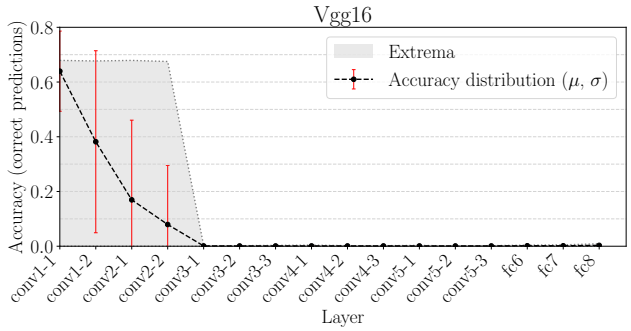
7.5.3.1 Floating Point and Fixed Point

In contradiction to common belief, the $\mathcal{F}$ representation is more vulnerable to injections even though it has more bits. The individual impact of a bit in a short representation (8-bit fixed point) is greater than its counter part in the $\mathcal{F}$ representation. However, the divergence from the correct value is greater in the later due to the nature of the representation. The exponent is not represented in the fixed point

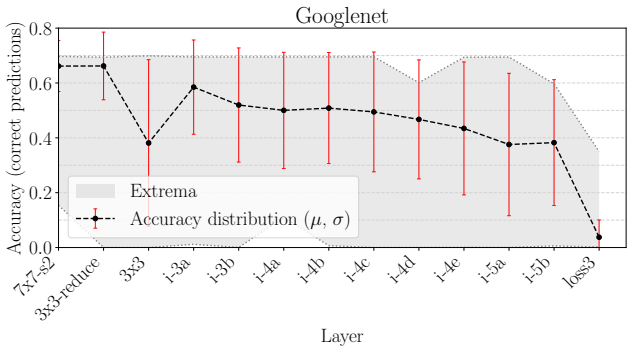
[9]The worst case is 0.001 which is equal to randomly guessing the class over the 1000 possibilities.



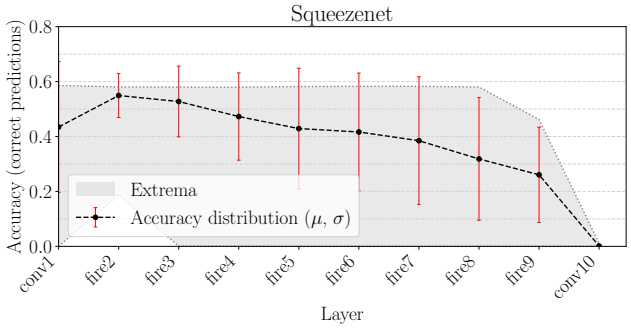
(a)



(b)



(c)



(d)

Figure 7.13: Impact of faults layer-wise for the four networks. Each series is represented as the mean top-1 accuracy (black dots), the standard deviation (red error-bars) and the minimum/maximum (gray fill).

representation. A bit-flip in any position is similar to adding or subtracting a power of 2. Since all weights range from -1 to $+1$ [139], the value added or subtracted is minuscule. Hence, it's impact can be logically masked.

7.5.3.2 Bit Position

In the $\mathcal{F}$ representation, not all the bits in the exponent are important. The impact of bits is not linear to the bit position but constant except for the most-significant bit as could be seen in Figure 7.12. This is partly due to the distribution of CNN weights. Weights range from -1 to $+1$. High values in the exponent are always accompanied with a negative exponent sign. Having a bit-flip will decrease the value even more, making it close to 0, which is not a big difference considering the range of weights. The only important change is the most-significant bit of the exponent. If changed from 0 to 1, the new value will be orders of magnitude higher than the others. Combined with the maximum pooling, this leads to catastrophic results in the subsequent layers.

7.5.3.3 Layer Index

Although the general tendency shows that the last layers are more vulnerable, no conclusions could be drawn. In other words, vulnerable layers of a new CNN can not be located without a simulation. A fault injection engine such as the one we presented in this work should be used to evaluate the individual layer vulnerability. This is an extension to Netscope [10], a neural network visualizer and analyzer. We introduced the resilience parameter which is computed from the accuracy reported by the fault-injection engine. The modified version allows to extract the most vulnerable layers visually. The analyzer is made publicly available with the engine source code[11].

Figure 7.14 shows the output of the extended visualizer. It is a different representation of the *layer* results. The percentage of correct predictions is shows for each layer individually. Depending on the complexity of the network, the visual representation helps identifying the vulnerable layers. In the example of Googlenet, the *3x3* layer in the inception modules seems to be very fragile. Errors are more critical when injected in this layer. A higher level of protection of this particular layer (in all modules) should increase reliability with minimal overhead.

7.5.3.4 Suggestions and Guidelines

The first study shows a bit difference between the storage formats. Floating point representation should be used with caution in critical systems. The fixed point representation would result in less memory and computation[12] overhead with higher reliability. System designers should consider this aspect when dealing with aggressive environments.

It was shown that the most significant bit in the exponent is the vulnerable part of the floating point representation. Using this conclusion, the overhead of redundancy techniques could be reduced. Techniques such as TMR will Replicate the whole number three times. This will triple the memory requirements of Googlenet for instance, whose number of weights is $6,996,452$, thereby adding 427 megabytes of required storage. If applied exclusively to the vulnerable bits, only 13 megabytes ($\frac{427}{32}$) would be necessary. This result can also be extended to the *layer* test with variable degrees of protection depending on the resilience of each layer as outputted by the injection engine.

[10] https://github.com/ethereon/netscope

[11] https://www.github.com/cypox/CNN-Fault-Injector

[12] Fixed point representation is usually coupled with low precision arithmetic. This allows for better efficiency with comparable accuracy.

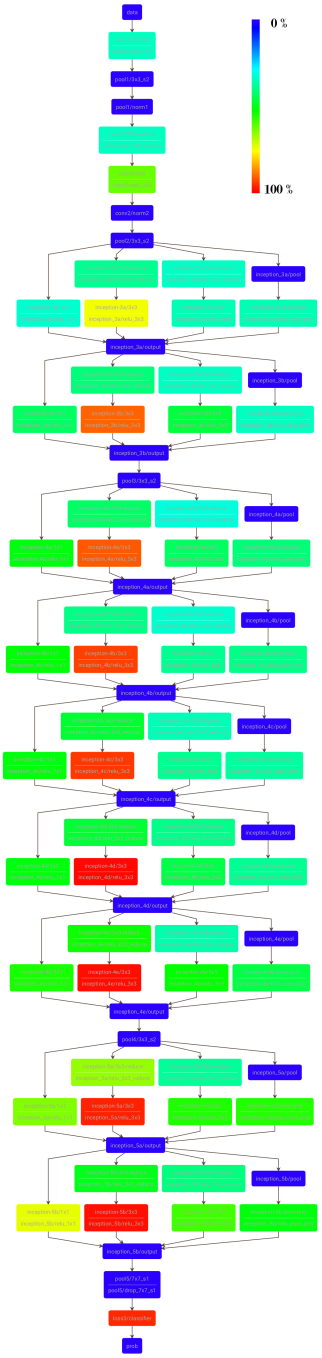


Figure 7.14: Vulnerable layers in Googlenet represented with the same order of precedence used during execution. The background color of each layer represent its vulnerability.

7.5.4 Summary

An extensive analysis of inherent fault tolerance of CNNs is proposed. We show that quantization have a positive impact on reliability. The issue with non-quantized networks is the data representation. For the IEEE-754 format, the most significant bit of the exponent is crucial to reliability for CNNs. A layer-wise analysis is then performed. For complex networks, the reliability of the overall network should be studied based on the architecture. The framework we developed to analyse reliability is made publicly available. This study is useful in localizing the vulnerable parts of CNNs and helps designing comprehensive low-overhead reliability enhancement techniques.

Chapter 8

General Conclusion and Perspectives

Contents

8.1 Conclusion

The rapid pace of machine learning development is a key contributor to the recent advances in ADSs. These advanced systems are the future of transportation. Hardware and software components are at the brain of these autonomous systems. While the challenge of accurate decision making was partly solved by machine learning, their usage is quite problematic. The hardware requirement of these algorithms are colossal. Moreover, the safety measures that accompanies these systems need to be addressed and solved explicitly. In next generation ADSs, designers will be faced with a dilemma. On the one hand, ADSs are demanding high performance platforms and highly reliable systems; systems that are expected to swallow an immense quantity of data coming from an increasing number of sensors of different types. On the other hand, these systems must be deployed on low cost platforms using recent technologies and have a short Time-To-Market.

To solve the issue of performance, many architectures have been proposed. In this thesis, we surveyed the most recent accelerator designs and we classified them by platform. Each platform presents its own strengths and weaknesses. A heterogeneous system is inevitable in the design of driverless vehicles. The ubiquity of high performance embedded devices, albeit inefficient, eased the integration of machine learning algorithms in ADSs.

We also presented two accelerator designs. First, we proposed to solve the memory bottleneck problem present in various von Neumann architectures by using the In-Memory Computing paradigm. Our first design performs computation and storage in the same unit using a ReAP, hence, improve performance and reduce requirements over a state-of-the-art design on FPGA.

The second accelerator uses a novel acceleration technique. It hard-codes weights in the multipliers of the accelerator. With this idea, no storage is required for weights, moreover, the complexity of multipliers is reduced which yield better performance. While many accelerators on FPGAs trades accuracy for performance, this acceleration technique enables a new trade off, that of flexibility, for safety critical systems. This acceleration scheme is orthogonal to all others and can be applied in conjunction with others.

Finally, we tackled the problem of hardware accelerators from a reliability perspective. We simulated environment errors and intentional attacks as random bit-flips. We used this assumption to create our fault injection model. Errors targeting the memory units appear to be more dramatic on accuracy when compared to their combinatorial units counterpart. We also concluded that quantization, counter-intuitively, reduces this impact to a huge extent. As for whether machine learning should be trusted, in it's current state, and without any protection mechanism, for autonomous driving, if a short answer needs to be given based on the results presented in this thesis, it would be no.

In this thesis, we also presented a study on system design for autonomous driving. In this study, we showed how a profiling could be performed offline to choose the best input size to use in a system. The resized images are then streamed and used in an FPGA accelerator that outputs final detection results. While this study only focus on detection, this system could be completed with other components.

8.2 Perspectives

The various tasks that needs to be performed in timely manner can only be solved using heterogeneous platforms. Associative processors and FPGAs are dedicated to specific types of applications. They are not capable of managing a full stacked system on their own. Although efficient, their integration with a full system may engender communication problems. As a follow up to our study, the usage of a ReAP as an

acceleration module need to be studied. Weights, originally stored inside the CAM, need to be shared with other compute units.

The integration of embedded devices as acceleration modules in system designs also raises the question on schedulability. Tasks can be performed by various platforms, more efficiently on some than others. A single platform can never be capable enough of managing the whole system without being overwhelmingly expensive. An offline study needs to be performed on what should be executed on which platform.

From a system point-of-view, the implication of other road users is essential to achieve full self driving. Vehicle-to-vehicle communication and vehicle-to-infrastructure can provide more information to the vehicle on the state of the environment and even the state of the vehicle itself. These external sources of information can also be used as processing units. Resources deployed in the infrastructure or in the cloud are capable of remotely process the vehicle state and take decisions if the network delays allows it. The scheduling and partitioning problem mentioned earlier can only be more necessary. A multi-level scheduling of tasks need to be performed to pick the best execution configuration with the hardware deployed in the car, that on other vehicles and the hardware on the cloud.

In their current state, these systems are prone to errors. While we focused on hardware failures and intentional bit-flip attacks on network parameters and activations, the nature of errors can be adversarial, targeting input data, for instance, images. Neural networks have shown a particular weakness towards these attacks [72]. Since their usage can not be excluded from autonomous system design, a fault-injection study that includes the input, i.e. images, signals and other types of data, need to be performed to model the full extent of these inconsistencies. Later on, hardware security mechanism needs to be deployed to parry with the increasing security challenges, especially with human life in the cycle.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 49, 64

[2] K. Abdelouahab, M. Pelcat, J. Sérot, C. Bourrasset, and F. Berry. Tactics to directly map cnn graphs on embedded fpgas. *IEEE Embedded Systems Letters*, 9(4):113–116, Dec 2017. 77

[3] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot, and François Berry. Accelerating cnn inference on fpgas: A survey. *arXiv preprint arXiv:1806.01683*, 2018. 8, 11, 74, 75, 77, 104, 114

[4] L. Aksoy, P. Flores, and J. Monteiro. A novel method for the approximation of multiplierless constant matrix vector multiplication. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 98–105, Oct 2015. 107

[5] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana

Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. 64, 66

[6] Manaar Alam, Arnab Bag, Debapriya Basu Roy, Dirmanto Jap, Jakub Breier, Shivam Bhasin, and Debdeep Mukhopadhyay. Enhancing fault tolerance of neural networks for security-critical applications. *CoRR*, abs/1902.04560, 2019. 123

[7] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, June 2016. 107

[8] Amjad Almahairi, Nicolas Ballas, Tim Cooijmans, Yin Zheng, Hugo Larochelle, and Aaron Courville. Dynamic capacity networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2549–2558. JMLR.org, 2016. 74

[9] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016. 75

[10] Alexander Amini, Guy Rosman, Sertac Karaman, and Daniela Rus. Variational end-to-end navigation and localization. In *International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24, 2019*, pages 8958–8964. IEEE, 2019. 7, 41

[11] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. Yodann: An architecture for ultralow power binary-weight cnn acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):48–60, Jan 2018. 77, 78

[12] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. Optimizing performance of recurrent neural networks on gpus. *CoRR*, abs/1604.01946, 2016. 70, 77

[13] ARM. Arm expects vehicle compute performance to increase 100x in next decade. https://www.arm.com/company/news/2015/04/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade. Accessed: 2020-03-11. 39

[14] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017. 120

[15] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. An opencl™deep learning accelerator on arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 55–64, New York, NY, USA, 2017. ACM. 50, 61, 74, 76

[16] Taiwo Oladipupo Ayodele. Types of machine learning algorithms. In Yagang Zhang, editor, *New Advances in Machine Learning*, chapter 3. IntechOpen, Rijeka, 2010. 24

[17] Daniele Bagni, A. Di Fresco, J. Noguera, and F. M. Vallina. A zynq accelerator for floating point vivado hls, January 2016. 95, 96

[18] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *CoRR*, abs/1511.06297, 2015. 74

[19] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432, 2013. 74

[20] Mohamed Benmimoun, Andreas Pütz, Adrian Zlocki, and Lutz Eckstein. eurofot: Field operational test and impact assessment of advanced driver assistance systems: Final results. In *Proceedings of the FISITA 2012 World Automotive Congress*, pages 537–547, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 44

[21] Jonathan Binas, Daniel Neil, Shih-Chii Liu, and Tobi Delbrück. DDD17: end-to-end DAVIS driving dataset. *CoRR*, abs/1711.01458, 2017. 44

[22] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. 47

[23] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. Practical fault attack on deep neural networks. *CoRR*, abs/1806.05859, 2018. 123

[24] Karel A Brookhuis, Dick De Waard, and Wiel H Janssen. Behavioural impacts of advanced driver assistance systems–an overview. *European Journal of Transport and Infrastructure Research*, 1(3), 2019. 38

[25] Nader H. Bshouty. A lower bound for matrix multiplication. *SIAM Journal on Computing*, 18(4):759–765, 1989. 87

[26] David M. Budden, Alexander Matveev, Shibani Santurkar, Shraman Ray Chaudhuri, and Nir Shavit. Deep tensor convolution on multicores. *CoRR*, abs/1611.06565, 2016. 65

[27] Martin Buehler, Karl Iagnemma, and Sanjiv Singh. *The 2005 DARPA grand challenge: the great robot race*, volume 36. Springer, 2007. 7, 38, 46, 47

[28] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. *CoRR*, abs/1903.11027, 2019. 44

[29] Caffe. Caffe model zoo. http://caffe.berkeleyvision.org/model_zoo.html. Accessed: 2018-09-05. 114

[30] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozeman, MT, USA, 1969. AAI7010025. 86, 93

[31] Srimat T. Chakradhar, Murugan Sankaradass, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ISCA '10*, 2010. 75, 76

[32] Casey Chan. What facebook deals with everyday. https://gizmodo.com/what-facebook-deals-with-everyday-2-7-billion-likes-3-5937143, 2012. Gizmodo. 25

[33] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. 65, 67

[34] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. 67

[35] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 2285–2294. JMLR.org, 2015. 106

[36] Y. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016. 8, 78, 79

[37] Y. Chen, J. Emer, and V. Sze. Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro*, 37(3):12–21, 2017. 67

[38] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, Jan 2017. 107, 113

[39] Y. Chen, J. Wang, J. Li, C. Lu, Z. Luo, H. Xue, and C. Wang. Lidar-video driving dataset: Learning driving policies effectively. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5870–5878, June 2018. 44

[40] Yuanchang Chen, Yizhe Zhu, Fei Qiao, Jie Han, Yuansheng Liu, and Huazhong Yang. Evaluating data resilience in cnns from an approximate memory perspective. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, pages 89–94, New York, NY, USA, 2017. ACM. 122

[41] Y. Choi, N. Kim, S. Hwang, K. Park, J. S. Yoon, K. An, and I. S. Kweon. Kaist multi-spectral day/night data set for autonomous and assisted driving. *IEEE Transactions on Intelligent Transportation Systems*, 19(3):934–948, March 2018. 44

[42] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015. 64

[43] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1337–III–1345. JMLR.org, 2013. 71

[44] EUROPEAN COMMISSION. Eu road safety policy framework 2021-2030 - next steps towards "vision zero", 2019. 14

[45] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 44

[46] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016. 106

[47] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014. 56, 57, 117

[48] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014. 133

[49] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 3123–3131. Curran Associates, Inc., 2015. 106

[50] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *CoRR*, abs/1801.08058, 2018. 67

[51] Electric Vehicle Database. Energy consumption of full electric vehicles. https://ev-database.org/cheatsheet/energy-consumption-electric-car. 45

[52] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 31

[53] Nina Dethlefs and Ken Hawick. Define: A fluent interface dsl for deep learning applications. In *Proceedings of the 2Nd International Workshop on Real World Domain Specific Languages*, RWDSL17, pages 3:1–3:10, New York, NY, USA, 2017. ACM. 61

[54] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham W. Taylor, and Shawki Areibi. Caffeinated fpgas: FPGA framework for convolutional neural networks. *CoRR*, abs/1609.09671, 2016. 59

[55] Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, Diogo Moitinho de Almeida, Brian McFee, Hendrik Weideman, Gábor Takács, Peter de Rivaz, Jon Crall, Gregory Sanders, Kashif Rasul, Cong Liu, Geoffrey French, and Jonas Degrave. Lasagne: First release., August 2015. 65

[56] Anand Dixit and Alan Wood. Impact of new technology on soft error rates. *Reliability Physics Symposim (IRPS)*, pages 486–492, 2011. 127

[57] Fernando dos Santos, Pedro Foletto Pimenta, Caio Lunardi, Lucas Draghetti, Luigi Carro, David Kaeli, and P Rech. Analyzing and increasing the reliability of convolutional neural networks on gpus. *IEEE Transactions on Reliability*, PP:1–15, 11 2018. 124

[58] I. Chatterjee et al. Impact of technology scaling on sram soft error rates. *IEEE Trans. Nucl. Sci.*, 61(6):3512–3518, December 2014. 123

[59] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 257–260, May 2010. 77

[60] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. In *2009 International Conference on Field Programmable Logic and Applications*, pages 32–37, Aug 2009. 75, 76

[61] Clément Farabet, Berin Martini, B. Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. *CVPR 2011 WORKSHOPS*, pages 109–116, 2011. 75, 76, 77

[62] Kayvon Fatahalian. From shader code to a teraflop: How gpu shader cores work. In *SIGGRAPH'2009*, SIGGRAPH '10. ACM, 2010. 67, 68

[63] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962. 95

[64] Caxton C. Foster. *Content Addressable Parallel Processors*. John Wiley & Sons, Inc., New York, NY, USA, 1976. 90

[65] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. Deltarnn: A power-efficient recurrent neural network accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 21–30, New York, NY, USA, 2018. Association for Computing Machinery. 74

[66] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. *SIGARCH Comput. Archit. News*, 45(1):751–764, April 2017. 78

[67] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012. 43, 44, 51, 104

[68] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of high-performance deep learning convolutions on simd architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 66:1–66:12, Piscataway, NJ, USA, 2018. IEEE Press. 65

[69] Zoubin Ghahramani. Unsupervised learning. In *Summer School on Machine Learning*, pages 72–112. Springer, 2003. 24

[70] Worldwide global automobile database. Car sales figures: Number of cars sold in united states, europe, china, breakdown by make and model. https://www.teoalida.com/cardatabase/. 45

[71] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 696–701, June 2014. 75, 76

[72] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, 2014. 143

[73] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159, April 2017. 74

[74] Denis A. Gudovskiy and Luca Rigazio. Shiftcnn: Generalized low-precision architecture for inference of convolutional neural networks. *CoRR*, abs/1706.02393, 2017. 107

[75] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, Jan 2018. 8, 74

[76] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of fpga based neural network accelerator, 2017. 50

[77] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of fpga-based neural network inference accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 12(1), March 2019. 73, 74, 75

[78] John L. Gustafson. *Brent's Theorem*, pages 182–185. Springer US, Boston, MA, 2011. 65

[79] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *CoRR*, abs/1604.03168, 2016. 106

[80] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2018. 50, 114, 117

[81] Ouarnoughi Hamza, Neggaz Mohamed Ayoub, E Gulcane, Ozcan Ozturk, and Niar Smail. Hierarchical collborative platform for autonomous driving. In *Workshop on INTelligent Embedded Systems Architectures and Applications (ESWEEK'2019)*, ESWEEK'2019, 2019. 18

[82] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, and et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 75–84, New York, NY, USA, 2017. Association for Computing Machinery. 74

[83] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *ICLR*, 2016. 8, 74, 106

[84] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pages 1135–1143, Cambridge, MA, USA, 2015. MIT Press. 107

[85] Song Han, Yu Wang, Huazhong Yang, William (Bill) J. Dally, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, and et al. Ese. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, 2017. 50

[86] A. Haron, J. Yu, R. Nane, M. Taouil, S. Hamdioui, and K. Bertels. Parallel matrix multiplication on memristor-based computation-in-memory architecture. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 759–766, July 2016. 88, 97

[87] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998. 26

[88] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. 65, 67

[89] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, 2018. 106

[90] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. Libxsmm: Accelerating small matrix multiplications by runtime code generation. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 981–991, Nov 2016. 64

[91] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. 98

[92] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017. 11, 62, 63

[93] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 41:1–41:16, New York, NY, USA, 2019. ACM. 70

[94] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. 67

[95] Xinyu Huang, Xinjing Cheng, Qichuan Geng, Binbin Cao, Dingfu Zhou, Peng Wang, Yuanqing Lin, and Ruigang Yang. The apolloscape dataset for autonomous driving. *CoRR*, abs/1803.06184, 2018. 44

[96] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, Aug 2016. 74, 75

[97] Loc Nguyen Huynh, Rajesh Krishna Balan, and Youngki Lee. Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices. In *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, WearSys '16, pages 25–30, New York, NY, USA, 2016. ACM. 71, 72

[98] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016. 33, 114

[99] M. Imani, D. Peroni, Y. Kim, A. Rahimi, and T. Rosing. Efficient neural network acceleration on gpgpu using content addressable memory. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1026–1031, March 2017. 71

[100] ISO. Road vehicles – Functional safety, 2011. 16, 122

[101] ISO/PAS. Road vehicles — Safety of the intended functionality, 2019. 16

[102] Benoît Jacob and Gaël Guennebaud. Eigen: a c++ linear algebra library. http://eigen.tuxfamily.org. Accessed: 2019-09-10. 64

[103] Vishal Jain. Hls based deep neural network accelerator library for xilinx ultrascale+ mpsocs. https://github.com/Xilinx/CHaiDNN, 2013. 50, 52

[104] Ju-Wook Jang, S. B. Choi, and V. K. Prasanna. Energy- and time-efficient matrix multiplication on fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(11):1305–1319, Nov 2005. 88

[105] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 49, 58, 64, 66, 114

[106] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. Optimizing n-dimensional, winograd-based convolution for manycore cpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 109–123, New York, NY, USA, 2018. ACM. 65

[107] J. Jiang, V. Mirian, K. P. Tang, P. Chow, and Z. Xing. Matrix multiplication based on scalable macro-pipelined fpga accelerator architecture. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 48–53, Dec 2009. 88

[108] Ziheng Jiang, Tianqi Chen, and Mu Li. Efficient deep learning inference on edge devices. In *SysML*, 2018. 67

[109] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang. Accelerating low bit-width convolutional neural networks with embedded fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2017. 74

[110] Georgina Binoy Joseph and R Devanathan. Algorithms for multiplierless multiple constant multiplication in online arithmetic. *Circuits, Systems, and Signal Processing*, 37(11):5127–5142, 2018. 107

[111] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM. 8, 79, 80, 81

[112] Z. Jovanovic and V. Milutinovic. Fpga accelerator for floating-point matrix multiplication. *IET Computers Digital Techniques*, 6(4):249–256, July 2012. 88

[113] Ju-Wook Jang, S. B. Choi, and V. K. Prasanna. Energy- and time-efficient matrix multiplication on fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(11):1305–1319, Nov 2005. 75, 87

[114] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *SIGOPS Oper. Syst. Rev.*, 51(2):615–629, April 2017. 71

[115] Nick Kanopoulos, Nagesh Vasanthavada, and Robert L Baker. Design of an image edge detection filter using the sobel operator. *IEEE Journal of solid-state circuits*, 23(2):358–367, 1988. 30

[116] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, June 2014. 74

[117] Sato Kaz, Young Cliff, and Patterson David. An in-depth look at google's first tensor processing unit (tpu). https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, 2017. 60, 88, 98

[118] Kevin Kiningham, Michael Graczyk, Athul Ramkumar, and SCPD Stanford. Design and analysis of a hardware cnn accelerator. *Stanford University*, 27:6, 2017. 8, 81, 83

[119] Sheila Klauer, Feng Guo, Jeremy Sudweeks, and Thomas Dingus. An analysis of driver inattention using a case-crossover approach on 100-car data. National Highway Traffic Safety Administration (NHTSA, 01 2010. 44

[120] Philip Koopman, Uma Ferrell, Frank Fratrik, and Michael Wagner. A safety standard approach for fully autonomous vehicles. In Alexander Romanovsky, Elena Troubitsyna, Ilir Gashi, Erwin Schoitsch, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security*, pages 326–332, Cham, 2019. Springer International Publishing. 16

[121] Skanda Koppula, Lois Orosa, A. Giray Yaundefinedlundefinedkçundefined, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. Eden: Enabling energy-efficient, high-performance deep neural network inference using approximate dram. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 166–181, New York, NY, USA, 2019. Association for Computing Machinery. 122

[122] A. Kouris, S. I. Venieris, and C. Bouganis. Cascadecnn: Pushing the performance limits of quantisation in convolutional neural networks. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 155–1557, Aug 2018. 74

[123] Robert Kozma, Robinson E. Pino, and Giovanni E. Pazienza. *Are Memristors the Future of AI?*, pages 9–14. Springer Netherlands, Dordrecht, 2012. 86

[124] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. 32

[125] V. K. P. Kumar and Y. . Tsai. On synthesizing optimal family of linear systolic arrays for matrix multiplication. *IEEE Transactions on Computers*, 40(6):770–774, June 1991. 8, 87

[126] Hsiang-Tsung Kung. Why systolic architectures? *Computer*, 15(1):37–46, Jan 1982. 60

[127] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 461–475, New York, NY, USA, 2018. ACM. 78

[128] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, June 2016. 58

[129] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 27, 31, 127

[130] Yann LeCun et al. Lenet-5, convolutional neural networks. 126

[131] Fengfu Li and Bin Liu. Ternary weight networks. *CoRR*, abs/1605.04711, 2016. 8, 74

[132] J. Li, R. Montoye, M. Ishii, K. Stawiasz, T. Nishida, K. Maloney, G. Ditlow, S. Lewis, T. Maffitt, R. Jordan, L. Chang, and P. Song. 1mb 0.41 m2 2t-2r cell nonvolatile tcam with two-bit encoding and clocked self-referenced sensing. In *2013 Symposium on VLSI Technology*, pages C104–C105, June 2013. 92, 98

[133] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. A 7.663-tops 8.2-w energy-efficient fpga accelerator for binary convolutional neural networks (abstract only). In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 290–291, New York, NY, USA, 2017. Association for Computing Machinery. 74

[134] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. *CoRR*, abs/1511.06393, 2015. 122

[135] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 2178–2188, Red Hook, NY, USA, 2017. Curran Associates Inc. 74

[136] X. Lin, S. Yin, F. Tu, L. Liu, X. Li, and S. Wei. Lcp: a layer clusters paralleling mapping method for accelerating inception and residual networks on fpga. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2018. 74

[137] Mike Demler Linley Gwennap and Loyd Case. *A Guide to Processors for Deep Learning*, volume 1. The Linley Group, 2017. 8, 83

[138] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. *CoRR*, abs/1701.00299, 2017. 74

[139] Qi Liu, Tao Liu, Zihao Liu, Yanzhi Wang, Yier Jin, and Wujie Wen. Security analysis and enhancement of model compressed deep learning systems under adversarial attacks. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, ASPDAC '18, pages 721–726, Piscataway, NJ, USA, 2018. IEEE Press. 122, 124, 138

[140] S. Liu, L. Li, J. Tang, S. Wu, and J. Gaudiot. *Creating Autonomous Vehicle Systems*. Morgan & Claypool, 2017. 7, 39, 49

[141] Y. Liu, L. Wei, B. Luo, and Q. Xu. Fault injection attack on deep neural network. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 131–138, Nov 2017. 123

[142] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on cpus. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1025–1040, 2019. 61, 67

[143] L. Lu, Y. Liang, Q. Xiao, and S. Yan. Evaluating fast algorithms for convolutional neural networks on fpgas. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 101–108, April 2017. 8, 50, 61, 74, 76, 77, 113

[144] T. Luo, S. Liu, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam, and Y. Chen. Dadiannao: A neural network supercomputer. *IEEE Transactions on Computers*, 66(1):73–88, Jan 2017. 78

[145] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J. Seo. End-to-end scalable fpga accelerator for deep residual networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017. 75

[146] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–54. Association for Computing Machinery, Inc, 2 2017. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017 ; Conference date: 22-02-2017 Through 24-02-2017. 74, 75

[147] Yufei Ma, N. Suda, Yu Cao, J. Seo, and S. Vrudhula. Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Aug 2016. 75, 106

[148] Will Maddern, Geoffrey Pascoe, Chris Linegar, and Paul Newman. 1 year, 1000 km: The oxford robotcar dataset. *The International Journal of Robotics Research*, 36(1):3–15, 2017. 44

[149] Janarbek Matai, Dustin Richmond, Dajung Lee, and Ryan Kastner. Enabling fpgas for the masses. *CoRR*, abs/1408.5870, 2014. 8, 60

[150] Gareth Mitchell. How much data is on the internet? https://www.sciencefocus.com/future-technology/how-much-data-is-on-the-internet/, 2018. Science Focus. 25

[151] Sparsh Mittal. A survey on optimized implementation of deep learning models on the nvidia jetson platform. *Journal of Systems Architecture*, 97:428 – 442, 2019. 71

[152] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *CoRR*, abs/1603.01025, 2016. 106

[153] Neggaz Mohamed Ayoub, Alouani Ihsen, Niar Smail, and Kurdahi Fadi. A new storage-less hardware compression technique for cnns using constant multiplication. In *WAICA: Workshop on Artificial Intelligence and Computer Architecture Co-located with the 15th HiPEAC Conference on High Performance and Embedded Architectures and Compilation*, WAICA2020, 2020. 17

[154] Farinaz Koushanfar Mohammad Ghasemzadeh, Mohammad Samragh. Rebnet: Residual binarized neural network. In *Proceedings of the 26th IEEE International Symposium on Field-Programmable Custom Computing Machines*, FCCM '18, 2018. 74

[155] Amir Morad, Leonid Yavits, and Ran Ginosar. Efficient dense and sparse matrix multiplication on GP-SIMD. In *2014 24th International Workshop on Power and Timing Modeling, Optimization and Simulation, PATMOS 2014*, pages 1–8. IEEE, sep 2014. 88

[156] Duncan J. M. Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit K. Mishra, Debbie Marr, Suchit Subhaschandra, and Philip Heng Wai Leong. High performance binary neural networks on the xeon+fpga™ platform. *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017. 50, 74

[157] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 575–580, Jan 2016. 74, 75

[158] Mohammad Motamedi, Philipp Gysel, and Soheil Ghiasi. Placid: A platform for fpga-based accelerator creation for dcnns. *ACM Trans. Multimedia Comput. Commun. Appl.*, 13(4):62:1–62:21, September 2017. 61

[159] H. Nakahara, T. Fujii, and S. Sato. A fully connected layer elimination for a binarizec convolutional neural network on an fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2017. 74

[160] Hiroki Nakahara, Haruyoshi Yonekawa, Hisashi Iwamoto, and Masato Motomura. A batch normalization free binarized convolutional deep neural network on an fpga (abstract only). In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 290, New York, NY, USA, 2017. Association for Computing Machinery. 74

[161] M. A. Neggaz, I. Alouani, S. Niar, and F. Kurdahi. Are cnns reliable enough for critical applications? an exploratory study. *IEEE Design Test*, pages 1–1, 2019. 17

[162] M. A. Neggaz, S. Niar, and F. Kurdahi. Computational and communication reduction technique in machine learning based near sensor applications. In *2018 30th International Conference on Microelectronics (ICM)*, pages 68–71, Dec 2018. 18, 39, 49

[163] M. A. Neggaz, H. E. Yantir, S. Niar, A. Eltawil, and F. Kurdahi. Rapid in-memory matrix multiplication using associative processor. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 985–990, March 2018. 17, 86, 119

[164] Mohamed A Neggaz, Ihsen Alouani, Pablo R Lorenzo, and Smail Niar. A reliability study on cnns for critical embedded systems. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 476–479. IEEE, 2018. 132, 133, 134, 136

[165] Mohamed Ayoub Neggaz, Pablo Ribalta Lorenzo, Ihsen Alouani, and Smail Niar. A reliability study on cnns for critical embedded systems. In *2018 International Conference on Computer Design*, 2018. 17, 51

[166] G. Neuhold, T. Ollmann, S. R. Bulò, and P. Kontschieder. The mapillary vistas dataset for semantic understanding of street scenes. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 5000–5009, Oct 2017. 44

[167] Andrew NG. *Machine Learning Yearning*. deeplearning.ai, 2018. 25, 31, 42

[168] Smail Niar, Mohamed Ayoub Lachachi, Yazid abd Neggaz, Ihsen Alouani, Hasan Yantir, Fadi Kurdahi, and Ahmed Eltawil. Road extraction and object recognition for autonomous cars. In *Workshop on New Platforms for Future Cars: Current and Emerging Trends (in conjunction with DATE'2018)*, DATE'2018, 2018. 18

[169] Dan Noyes. The top 20 valuable facebook statistics – updated november 2019. https://zephoria.com/top-15-valuable-facebook-statistics/, 2019. Zephoria - Digital Marketing. 25

[170] E. Nurvitadhi, Jaewoong Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016. 71

[171] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84, Dec 2016. 74

[172] NVIDIA. https://developer.nvidia.com/drive/drive-software. https://developer.nvidia.com/drive. 48

[173] University of Illinois. The llvm compiler infrastructure. https://llvm.org/. Accessed: 2019-09-16. 70

[174] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40, June 2017. 107

[175] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 27–40, New York, NY, USA, 2017. ACM. 8, 78

[176] Y. Parmar and K. Sridharan. A resource-efficient multiplierless systolic array architecture for convolutions in deep networks. *IEEE Transactions on Circuits and Systems II: Express Briefs*, pages 1–1, 2019. 107

[177] Abhishek Patil, Srikanth Malla, Haiming Gang, and Yi-Ting Chen. The H3D dataset for full-surround 3d multi-object detection and tracking in crowded urban scenes. *CoRR*, abs/1903.01568, 2019. 44

[178] Dan W Patterson. *Artificial neural networks: theory and applications*. Prentice Hall PTR, 1998. 26

[179] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 13–19, Oct 2013. 75

[180] D. S. Phatak and I. Koren. Fault tolerance of feedforward neural nets for classification tasks. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 2, pages 386–391 vol.2, June 1992. 123

[181] D. S. Phatak and I. Koren. Complete and partial fault tolerance of feedforward neural nets. *IEEE Transactions on Neural Networks*, 6(2):446–456, March 1995. 124

[182] A. Podili, C. Zhang, and V. Prasanna. Fast and efficient implementation of convolutional neural networks on fpga. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 11–18, July 2017. 74

[183] A. Prost-Boucle, A. Bourge, F. Pétrot, H. Alemdar, N. Caldwell, and V. Leroy. Scalable high-performance architecture for convolutional ternary neural networks on fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, Sep. 2017. 74

[184] P. W. Protzel, D. L. Palumbo, and M. K. Arras. Performance and fault-tolerance of neural networks for optimization. *IEEE Transactions on Neural Networks*, 4(4):600–614, July 1993. 123

[185] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, and et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 26–35, New York, NY, USA, 2016. Association for Computing Machinery. 8, 74, 75

[186] Fahad Qureshi, Anastasia Volkova, Thibault Hilaire, and Jarmo Takala. Multiplierless processing element for non-power-of-two ffts. *hal-INRIA*, 2018. 107

[187] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013. 67

[188] A. Rahman, J. Lee, and K. Choi. Efficient fpga acceleration of convolutional neural networks using logical-3d compute array. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1393–1398, March 2016. 75

[189] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network withprogressive bit search. *arXiv preprint arXiv:1903.12269*, 2019. 123

[190] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016. 106

[191] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, Jun 2017. 51

[192] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018. 66

[193] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar. Deep3: Leveraging three levels of parallelism for efficient deep learning. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017. 74

[194] Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar. Delight: Adding energy dimension to deep neural networks. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, page 112–117, New York, NY, USA, 2016. Association for Computing Machinery. 74

[195] SAE. Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems, 2014. 38

[196] Ananda Samajdar, Yuhao Zhu, Paul N. Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic CNN accelerator. *CoRR*, abs/1811.02883, 2018. 81

[197] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf. A massively parallel coprocessor for convolutional neural networks. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 53–60, July 2009. 75, 76

[198] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy. Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 145–150, March 2016. 107

[199] Harald Schafer, Eder Santana, Andrew Haden, and Riccardo Biasini. A commute in data: The comma2k19 dataset. *CoRR*, abs/1812.05752, 2018. 44

[200] Brandon Schoettle. Sensor fusion: A comparison of sensing capabilities of human drivers and highly automated vehicles. *Sustain. Worldw. Transp.*, pages 1–42, August 2017. 11, 40

[201] C. Schorn, A. Guntoro, and G. Ascheid. Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 979–984, March 2018. 122

[202] B. E. Segee and M. J. Carter. Fault tolerance of pruned multilayer networks. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume ii, pages 447–452 vol.2, July 1991. 136

[203] Y. Shen, M. Ferdman, and P. Milder. Overcoming resource underutilization in spatial cnn accelerators. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016. 74

[204] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 33

[205] S Singh. Critical reasons for crashes investigated in the national motor vehicle crash causation survey. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812506, 2018. 14

[206] A Smiley and KA Brookhuis. Alcohol, drugs and traffic safety. road users and traffic safety. *Publication of: VAN GORCUM & COMP BV*, 1987. 38

[207] M. Song, Y. Hu, H. Chen, and T. Li. Towards pervasive and user satisfactory cnn across gpu microarchitectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2017. 71

[208] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, August 1969. 87

[209] Dmitri B Strukov, Duncan R Stewart, Julien Borghetti, Xuema Li, M Pickett, G Medeiros Ribeiro, Warren Robinett, G Snider, John Paul Strachan, Wei Wu, et al. Hybrid cmos/memristor circuits. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 1967–1970. IEEE, 2010. 97

[210] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 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*, FPGA '16, pages 16–25, New York, NY, USA, 2016. ACM. 61, 75

[211] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017. 79

[212] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. 33

[213] K. Takeda, J. H. L. Hansen, P. Boyraz, L. Malta, C. Miyajima, and H. Abut. International large-scale vehicle corpora for research on driver behavior on the road. *IEEE Transactions on Intelligent Transportation Systems*, 12(4):1609–1623, Dec 2011. 44

[214] J. Tan and X. Fu. Chapter 23 - addressing hardware reliability challenges in general-purpose gpus. In Hamid Sarbazi-Azad, editor, *Advances in GPU Research and Practice*, Emerging Trends in Computer Science and Applied Computing, pages 649 – 705. Morgan Kaufmann, Boston, 2017. 72

[215] Linpeng Tang, Yida Wang, Theodore L. Willke, and Kai Li. Scheduling computation graphs of deep learning models on manycore cpus. *CoRR*, abs/1807.09667, 2018. 66

[216] E. B. Tchernev, R. G. Mulvaney, and D. S. Phatak. Investigating the fault tolerance of neural networks. *Neural Computation*, 17(7):1646–1664, July 2005. 123

[217] Kertu Toompea. Simulations for training machine learning models for autonomous vehicles. Master's thesis, UUNIVERSITY OF TARTU, Institute of Computer Science, Computer Science Curriculum, 2019. 46

[218] Transportation Research Board and National Academies of Sciences, Engineering, and Medicine. *Naturalistic Driving Study: Field Data Collection*. The National Academies Press, Washington, DC, 2014. 44

[219] Udacity. Udacity dataset. https://github.com/udacity/self-driving-car/tree/master/datasets, January 2020. 44

[220] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017*

ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17, pages 65–74, New York, NY, USA, 2017. ACM. 61, 74, 77

[221] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997. 88

[222] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018. 70

[223] Virginia Vassilevska, Ryan Williams, and Raphael Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 585–589. ACM, 2007. 95

[224] S. I. Venieris and C. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, May 2016. 61, 77

[225] Yevgen Voronenko and Markus Püschel. Multiplierless multiple constant multiplication. *ACM Transactions on Algorithms (TALG)*, 3(2):11, 2007. 107

[226] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1058–1066, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. 32

[227] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. *Intel Math Kernel Library*, pages 167–188. Springer International Publishing, Cham, 2014. 61, 64, 67

[228] Erwei Wang, James J. Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter Y. K. Cheung, and George A. Constantinides. Deep neural network approximation for custom hardware: Where we've been, where we're going. *ACM Comput. Surv.*, 52(2):40:1–40:39, May 2019. 74

[229] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, Mu Li, and Yida Wang. A unified optimization approach for cnn model inference on integrated gpus. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 99:1–99:10, New York, NY, USA, 2019. ACM. 71, 72

[230] S. Wang, D. Zhou, X. Han, and T. Yoshimura. Chain-nn: An energy-efficient 1d chain architecture for accelerating deep convolutional neural networks. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1032–1037, March 2017. 74, 78

[231] Shenlong Wang, Min Bai, Gellért Máttyus, Hang Chu, Wenjie Luo, Bin Yang, Justin Liang, Joel Cheverie, Sanja Fidler, and Raquel Urtasun. Torontocity: Seeing the world with a million eyes. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 3028–3036, 2016. 44

[232] Richard Wei, Vikram S. Adve, and Lane Schwartz. DLVM: A modern compiler infrastructure for deep learning systems. *CoRR*, abs/1711.03016, 2017. 70, 122

[233] Xuechao Wei, Yun Liang, and Jason Cong. Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery. 75

[234] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 29:1–29:6, New York, NY, USA, 2017. ACM. 74, 76

[235] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. 64

[236] Wikipedia. Core i7 - intel. https://en.wikichip.org/wiki/intel/core_i7. Accessed: 2019-09-25. 62

[237] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. 74, 75, 111, 113

[238] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 887–898, New York, NY, USA, 2012. ACM. 87

[239] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980. 65, 76

[240] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, Feb 2019. 67

[241] E. Wu, X. Zhang, D. Berman, and I. Cho. A high-throughput reconfigurable processing array for neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2017. 74

[242] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876, 2015. Withdrawn. 71

[243] Qian Wang Xianyi Zhang and Zaheer Chothia. Openblas. https://github.com/xianyi/OpenBLAS. Accessed: 2019-09-10. 64

[244] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Yu-Wing Tai. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017. 74

[245] Li Yang, Zhezhi He, and Deliang Fan. A fully onchip binarized convolutional neural network fpga impelmentation with accurate inference. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '18, New York, NY, USA, 2018. Association for Computing Machinery. 74

[246] T. Yang, Y. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6071–6079, July 2017. 107

[247] H. E. Yantır, M. E. Fouda, A. M. Eltawil, and F. J. Kurdahi. Process variations-aware resistive associative processor design. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 49–55, Oct 2016. 95

[248] L Yavits, A Morad, and R Ginosar. Sparse Matrix Multiplication On An Associative Processor. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):3175–3183, nov 2015. 8, 88, 89

[249] Leonid Yavits, Shahar Kvatinsky, Amir Morad, and Ran Ginosar. Resistive associative processor. *IEEE Comput. Archit. Lett.*, 14(2):148–151, July 2015. 90

[250] J. Yli-Kaakinen and T. Saramaki. A systematic algorithm for the design of multiplierless fir filters. In *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, volume 2, pages 185–188 vol. 2, May 2001. 107

[251] Guillaume Girardin Yohann Tschudi. Road to robots, sensors and computing for autonomous vehicle. https://www.autonomoustechconf.com/sites/autosensorsconf/files/assets/1%20Data%20Analysts%20Yole%20Development_Girardin_Tschudi.pdf, 2018. 7, 16

[252] Fisher Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan, and Trevor Darrell. BDD100K: A diverse driving video database with scalable annotation tooling. *CoRR*, abs/1805.04687, 2018. 44

[253] Xiaoyong Yuan, Pan He, Qile Zhu, Rajendra Rana Bhat, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *CoRR*, abs/1712.07107, 2017. 124

[254] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *arXiv preprint arXiv:1906.05113*, 2019. 11, 38, 44

[255] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *CoRR*, abs/1906.05113, 2019. 14

[256] C. Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2016. 74

[257] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM. 60, 61, 74, 75

[258] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, pages 326–331, New York, NY, USA, 2016. ACM. 50, 75

[259] Jeff Jun Zhang, Tianyu Gu, Kanad Basu, and Siddharth Garg. Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator. In *2018 IEEE 36th VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2018. 124

[260] 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*, FPGA '17, pages 25–34, New York, NY, USA, 2017. ACM. 50, 61

[261] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 951–965, Berkeley, CA, USA, 2018. USENIX Association. 66

[262] Qianru Zhang, Meng Zhang, Tinghuan Chen, Zhifei Sun, Yuzhe Ma, and Bei Yu. Recent advances in convolutional neural network acceleration. *Neurocomputing*, 323:37 – 51, 2019. 7, 49, 56

[263] Ting Zhang, Cheng Xu, Tao Li, Yunchuan Qin, and Min Nie. An optimized floating-point matrix multiplication on fpga. *Information Technology Journal*, 12(9):1832, 2013. 88

[264] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. Hwu, and D. Chen. Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2018. 74

[265] X. Zhang, C. Xie, J. Wang, W. Zhang, and X. Fu. Towards memory friendly long-short term memory networks (lstms) on mobile gpus. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 162–174, Oct 2018. 71

[266] R. Zhao, H. Ng, W. Luk, and X. Niu. Towards efficient convolutional neural network for domain-specific applications on fpga. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 147–1477, Aug 2018. 75

[267] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 15–24, New York, NY, USA, 2017. Association for Computing Machinery. 74

[268] Tian Zhao, Xiaobing Huang, and Yu Cao. Deepdsl: A compilation-based domain-specific language for deep learning. *ArXiv*, abs/1701.02284, 2017. 61

[269] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. Automatic code generation of convolutional neural networks in fpga implementation. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 61–68, Dec 2016. 74

[270] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016. 8, 74

[271] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. *CoRR*, abs/1612.01064, 2016. 8, 74

[272] J. Ziegler and al. Ibm experiments in soft fails in computer electronics. *IBM Journal of Research and Development*, 40(1), 1996. 123

[273] A. Zlateski, K. Lee, and H. S. Seung. Znn – a fast and scalable algorithm for training 3d convolutional networks on multi-core and many-core shared memory machines. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 801–811, May 2016. 64, 66