TECHNICAL UNIVERSITY OF CRETE-SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

MILIADIS PANAGIOTIS

SUPERVISOR: PNEVMATIKATOS DIONISIOS

PERFORMANCE LANDSCAPE OF CNN ACCELERATION TOOLS AND RESOURCE CONSTRAINED PLATFORMS

Acknowledgements

Dedicated to those who supported me in the last months.

Abstract

Over the last years, a rapid growth in the development of applications that are based on Convolutional Neural Networks is observed. Despite of the large advances in processor units, the use of computer vision tasks is still challenging in resource constrained platforms. This thesis will present four toolkits, that accelerate the performance of inference applications by targeting the processor units from the top hardware vendors; Intel, Nvidia, Arm and Xilinx. In order to achieve optimal execution, the toolkits exploit the hardware acceleration that processors provide, as well as special processor units and platforms, which are specially developed for deep learning inference tasks. The most well-known models for each task are described, alongside with the frameworks that the toolkits support and are used for model representation. Last but not least, real-world performance results are collected for different batches of images, in order to achieve a performance landscape of the existing tools.

Contents

Chapter 1

Introduction

1.1 Motivation

Over the last years, a rapid growth in the development of applications that are based on neural networks is observed. Even though the idea of neural networks, and the means to develop and use them for executing various tasks, has been around for several years, their development was limited mainly for research purposes for big labs. That phenomenon was due to the enormous computational demands for their execution, and with the low computational power offered by the resource contained platforms a few years ago, made it impossible to use them efficiently. Therefore, the most important obstacle for developing and using an application that is based on neural networks, was the cost of the hardware that was needed to achieve such a task. Nowadays, the huge technological advances have completely changed the landscape. Central Processor Units (CPUs) with multi physical cores, advanced dedicated Graphic Processor Units (GPUs) with gigabytes of VRAM, and the gigabytes of RAM, the capabilities that can be offered by a single platform exceed the typical systems that were once used for

deep learning inference applications. Moreover, the further need to accelerate the execution of neural networks led to the development of special processor units. These units have very low power consumption and increased performance for the targeting task. However, the execution of applications that based on neural networks remains a challenging task, despite of the huge resources that can be offered by a single system.

The overwhelming majority of build-in inference applications that use neural networks are based on convolutional neural networks (CNNs). Applications for image classification, object detection, image semantic segmentation, super - resolution, face recognition and detection, speech recognition, human pose estimation, text detection etc, are developed on a large scale by many programmers, for different purposes and needs. Yet, these kind of applications require a significant amount of resources that a system can offer in order to execute them efficiently. Furthermore, the applications may have a large set of data to process simultaneously. Thus, despite of the large computational power and big sets of memory that a platform can offer, major questions are left unanswered. How can we develop an application that can take advantage of the hardware acceleration that a system is providing? Are there any other processor units that can execute efficiently a deep learning inference application? What is the performance of deep learning inference applications, which can be achieved in resource contained platforms? In this thesis, i will try to answer these questions by providing a clear image of the existing tools and the associated processor units that can accelerate the execution of deep learning inference applications, by providing a detailed performance analysis and the capabilities that they can offer.

1.2 Context

The performance of the applications that are based on the execution of a convolutional neural network remains a major challenge, despite of the huge technological advancements in resource constrained platforms. Thus, four major hardware manufacturers took advantage of that challenge, and develop their own tools, in order to maximize the performance of deep learning inference applications in their respective platforms and chipsets. Moreover, Intel and Xilinx introduce specialised accelerators that can further accelerate the performance.

Prior to the presentation of the aforementioned tools, one of the objectives of this thesis is to give a background of the neural networks and especially the class of convolutional neural networks. It is well known that this class is a hot topic and many applications that are developed, are importing a plethora of CNN models. While the idea of neural networks is not new, recently frameworks such as TensorFlow and Caffe, allow the construction and the execution of a model, and they make these procedures a simple case for any programmer.

On the other hand, executing a convolutional neural network in a general purpose processor unit is not a simple task. The performance of these applications is based on the quality of the source code of the programmer, and in the most cases this code does not include hardware acceleration, like platform specific commands and functions. Hardware acceleration is a key factor in order to achieve higher performance and to develop more efficient applications. At the same time, many developers use only general purpose processors, ignoring the fact that many companies also construct platforms and processors that are primarily for applications that integrate AI and deep learning algorithms.

The toolkits that are going to be presented by this thesis contain libraries

with optimized functions that can exploit every asset of the target hardware. Thus, they can be used to accelerate the execution of any convolutional neural network model. Moreover, with the advent of hardware accelerators, such as vision processor units and deep learning processor units, the execution of an inference application becomes a much easier procedure.

The first tool is OpenVino and was developed by Intel. OpenVino exploits Intel's hardware, like CPUs and the GPUs that are integrated into them, in order to maximize the performance of deep learning inference. This toolkit includes a set of libraries that can speed up the functions that the developers use for computer vision and deep learning inference applications. Aside from the general processor units, Intel developed a special integrated circuit, that is called Neural Compute Stick that works alongside with OpenVino and offers great advantages in overall performance.

The dedicated GPUs with the gigabytes of Video RAM, that Nvidia has developed all over the years, are well known for their high performance that can provide, when they have to process big sets of data. For that reason, it makes them the perfect unit for executing deep learning inference applications. However, Nvidia developed TensorRT, a platform that can furthermore improve the the performance of deep learning inference applications.

A large amount of edge systems, that are widely used by research teams and by many companies for their products, are based on processor units constructed by Arm. The power efficient CPUs and GPUs that they develop are perfect for executing deep learning inference applications in low-power devices. This balance between executing efficiently deep learning inference applications and keep the power consumption as low as possible is a huge advantage for FPGAs, embedded platforms and mobile devices. For that reason, Arm developed the Neural Network Software Development Kit, a set of software and tools that are

capable of executing efficiently deep learning inference applications from the most known frameworks to any Cortex-A CPUs and Mali GPUs.

The last tool that i am going to examine in this thesis is DNNDK from Xilinx. DNNDK is part of the Edge AI Platform that provides a set of comprehensive tools and models, for efficient deep learning inference developing. The development kit can be used on specific Xilinx Zynq Ultrascale+ Boards. This approach is based on an image that Xilinx provides, in which a special Deep-learning processor Unit is designed for efficient execution of any application that based on deep neural networks. This unit takes full advantage of the Xilinx architecture, in order to achieve the optimal trade-off between latency, power and cost.

1.3 Objectives and Contributions

The primary objective of this thesis is to present a performance landscape of the aforementioned acceleration tools in constrained resource platforms that are widely used for developing various tasks. In order to achieve something like that, applications that are based on image classification, object detection and image segmentation have been developed. These tasks can use a large variety of pre-trained models, and provide a great flexibility by exploring neural networks that are both too deep with many parameters, and others that are not. By providing a full performance analysis of acceleration tools in resource constrained platforms, there is a strong motive to analyse the impact of these tools in the efficiency of deep learning inference applications.

Aside from the performance landscape, a complete description of how a developer can utilize the acceleration tools in his benefit is provided. These toolkits include libraries, pre optimized kernels, optimizers and often many more elements.

All these components cooperate with each other, in order to maximize the performance of a deep learning inference application. In addition, a description of the popular machine learning frameworks that the toolkits support is provided.

Nowadays, the data that an application may have to process each time can exceed the usual small numbers that many benchmarks usually provide. For that reason, i include in the performance analysis the results that can be achieved by processing batches of images simultaneously. Thus, the potential benefits of this approach are shown, based on the aforementioned results. In addition, each platform behaves differently by processing big sets of data in deep learning inference applications.

In order to achieve the goal of the detailed performance analysis, parameters that will help to understand how efficiently each neural network model executes with the help of the acceleration tools are extracted. The inference time of each CNN model is a major parameter is measured, and a detailed discussion in chapter 5 is provided based on the exported results. The number of gigabytes of RAM, that each platform has, is growing over the years, however a deep learning inference program still needs a significant percentage of this resource. Also, the tests provide useful information of how the batch size can affect both inference time and memory consumption, and which tasks cannot be completed due to low RAM in the system. From inference time i can extract throughput, which describes how much images can be processed per second.

Each platform executes the pre-trained models in different precisions like single point floating point (FP32), half-precision (FP16) or INT8. How the precision of the model can affect the aforementioned exported parameters? This is a question that can be discussed by the provided metrics. In addition, these metrics are affected by another factor, which is power consumption. But the main question for each processor unit is how efficiently can execute these tasks according to

power consumption.

Chapter 2

Background

2.1 Convolutional Neural Networks

2.1.1 Introduction to Neural Networks

One of the biggest areas of research in Artificial Intelligence is machine learning. In machine learning, computer systems utilize algorithms and statistical models in order to perform specific tasks without using instructions like normal algorithms do. Instead of instructions, they rely on extracting features from patterns, such as speech and images, in order to perform their tasks. For that reason, each model that was built by machine learning algorithms, is based on a training data set. This training data set determines in which tasks this model can be used on, while the number of the training data determines how accurate the model can be. In this thesis, i deal with a subset of machine learning, the neural networks, and notably with the convolutional neural networks.

Artificial neural networks are computing systems that are inspired by the biological neural networks of brain. Each neuron of biological neural network can be

mapped with a node of the artificial neural network. Each of these nodes are connected with other nodes of the same network, and they form multiple layers. This structure resembles the structure of the actual neurons in brain, so the flow of the data and the communication between the nodes in a artificial neural network can be imagined in the same way as the biological. At the beginning, the main idea of neural networks was to approach and solve real world problems in the same way that a human brain can do. With time, this idea was partially abandoned, and the research was moved on how efficiently can artificial neural networks perform on specific tasks. The most well known task, that is also evaluated in this thesis, is image classification. Other tasks can be speech recognition, medical diagnosis, computer vision, machine translation or even playing video games.

Each neural network is divided into multiple layers, and each layer includes multiple nodes-neurons. The neurons of different layers are connected with each other, but not with the neurons of the same layer. That means, that the output of a node, is the input to a node of a subsequent layer or generally to a layer that is located in a different level in the structure. The output signals that are transmitting over the layers, are usually real numbers, and they are computed by a nonlinear function of the sum of the nodes' inputs. This function can be differed from layer to layer, so the neurons in each layer perform different kind of transformations in their inputs. Within the structure of a neural network, each signal is transmitted from an input layer to an output, after traversing multiple layers. These layers can have either a recurrent format, where a signal traverses the same node multiple times, or a straight format, where each signal traverse each layer one time.

As i described before, each neuron is communicating with another neuron of a different layer through a connection. Like every communication system that is transmitting and receiving a signal, the behaviour of a neural network

is determined by the strength of this connection. This kind of strength in neural networks is called weight. The weights are adjusted during the training phase of the model by a specified learning rule, in order to get to a point where the model can perform the desired task correctly. These weights are playing a major role in the performance of a neural network, and their total number is critical for the resources that are consumed by an application.

2.1.2 Introduction to Convolutional Neural Networks

The most widely used subset of neural networks is convolutional neural networks (CNNs or ConvNets). CNNs are usually applied to applications for analyzing visual imagery. Some examples of these applications are image and video recognition, recommendation systems, image classification, medical image analysis, natural language processing etc.

An overwhelming proportion of CNN models that are developed, including the models that this thesis is going to use for the performance analysis, are based on supervising learning. Supervising learning is a type of machine learning task where learning of a function is based on mapping the input to an output based on example pairs. Each example is a pair consisting of an input vector, usually the input values of an image that are represented by pixels, and a desired output value. The inference function is produced during the training phase, where the initial algorithm tries to map the training data set to the specified labels. When the training phase is over, this inference function can be used by the inference applications for mapping new images in the specified classes.

Even though the desired output values, or the specified output labels, can be differ from network to network, CNN models have a restricted "visual field". The range of the visual field depends on the number and the variety of the specified

labels. If the model can perform exactly the specified task, that it was trained for, then the neurons of the output layer indicate the labels that the input is mapped, and the results can be further processed by the application. Each neuron of the output layer is responding only to small region of the initial visual field, which can also be referred as a receptive field. The receptive field of each neuron is partially overlapped by other receptive fields, in order to cover the entire visual field of the model. When a developer utilize a convolutional neural network model for his application, the accuracy of the mapped label usually never reaches one hundred percent. This happens because of the partially overlapped receptive fields.

CNNs are a special version of multilayer perceptrons. These type of networks are referred as fully connected networks, where each node in one layer is connected to all the nodes of the previous layer. This means that the output values of all nodes are transferred to the next layer. One side effect of this fully communication is that these networks are prone to data overfitting. This problem can be solved by the regularization process. By adding information in input data, usually the weight between the two neurons, or by assembling more complex patterns, the overfitting problem can be prevented.

The full connection between the neurons of subsequent layers does not describe accurately the way that these neurons are connected and transmitting signals in a convolutional neural network. In fact, the neurons of ConvNets are receiving inputs only from a restricted area of the previous layer. This input area is the aforementioned receptive field. So, the receptive field of each neuron in a layer in ConvNets is a

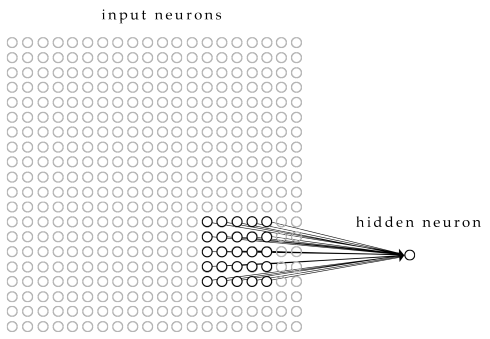

Figure 2.1: Receptive Field in CNN

subarea of the outputs of the previous layer, contrary to what happens in fully connected networks where the receptive field in subsequent layers is every element of the whole output area.

Another major element in ConvNets are weight vectors. Even though i mentioned before how we can adjust them and the significance of their role, the real value of this parameter is much greater. They are used as filters and in fact they represent particular features of the input image that it is being observed. The weight vectors along with biases that are applied, are usually real numbers which also determine the function that will apply to the input vector of each neuron in order to prevent overfitting. Last but not least, many neurons can share the same weight vectors and biases. This handy characteristic has as a result that the amount of the main memory that a program consumes for its execution is significantly reduced.

2.1.3 A brief history of Convolutional Neural Networks

An overwhelming proportion of computer vision systems nowadays are primarily powered by ConvNets, so the history of Convolutional neural networks is quite inseparable with the history of Computer Vision. Furthermore, ConvNets and CV are not new scientific fields, like many think, but they have a deeply connected history of many decades.

All started when Hubel and Wiesel are identified the meaning of receptive field in the late 1950s. In their work [1], they showed that each neuron on the visual cortexes in the cats and monkeys is responded only to a small region of the visual field. This was discovered accidentally, when after many failed tries, as they were slipping new slides into the projector, they noticed a sudden activity in one neuron. After many experiments and research, they concluded that what

got the neuron to be fired was the movement of the line created by the shadow of the sharp edge of the glass slide. The restricted region of each neuron was called receptive field, and neighboring neurons have overlapping and similar receptive fields. Also in their paper, they identified two types of visual cells in the brain,

- simple cells; whose output is maximized by straight edges having particular orientation within their receptive field, and
- complex cells; whose output is insensitive to the exact position of the edges in the fields, which fields are much larger than a normal receptive field.

In the end, they proved that visual processing starts with simpler structures, pretty likely when a new ConvNet model starts its training in our days.

The next highlight in the history was the invention of the first digital scanner[2]. Even though, a digital scanner does not have a direct relation with the ConvNets, the background of this project had played a major role. Russel Kirsch and his group had invented a device, that could transform an image into an array of numbers. With this transformation, the binary language machines could understand and display any image. So, the arrays of pixels that programmers use briefly in order to represent images in their applications came from this invention.

Many years have passed when the origin of the convolutional neural network architecture was proposed. In 1980, Kunihiko Fukushima, inspired by the work of Hubel and Wiesel, built a network of simple and complex cells that could recognize patterns independently by the position shifts. The first artificial neural network, which was consisted by multiple layers in a hierarchically structure, was a fact. Its name was neocognitron[3], which introduced the first two basic types of layers for ConvNets, the convolutional and the downsampling layers. In the time being, downsampling layers implemented functions that performed only average pooling. An alternate way to perform downsampling was introduced by

J. Weng. In this way, it was proposed to implement functions that perform max pooling[4] instead of average. A few years later, the first modern convolutional neural network model was released by LeCun. This model was LeNet-5 [5] and it was trained for recognizing handwritten numbers on checks, while another version of this model was used for reading zip codes. This was the beginning of one of the most well known datasets that developers are using even in our days, the MNIST Dataset[6].

Even though the first convolutional neural network was proposed in 1980, the continuous development of ConvNets, by implementing more convolutional layers and by discovering new types of layers, required faster implementations. This was achieved by using graphic processor units. In 2004, K. S. Oh and K. Jung shown that a simple neural network can be greatly accelerated on GPUs[7], and specifically they proved that the implementation of a neural network on a GPU can be 20 times faster than a similar implementation on CPU. One year after, another paper[8] showed how valuable is for machine learning to implement convolutional neural networks on general purpose GPUs. After that, many researchers are based on this paper and they implement their neural networks on GPU. The most notable moment was in 2010, when Dan Ciresan and his group proved that even deep neural networks can be trained on GPUs by using supervised training, in a much faster scale. Their network outperformed all other approaches that was based on MNIST dataset[9]. One year later, they extended their approach to ConvNets[10].

Another popular dataset in the time being was ImageNet[11], a dataset that was used for benchmarks in many categories like object classification and object detection. Based on this dataset, a well-known competition is held, the ImageNet Large Scale Visual Recognition Competition. At the beginning, all models that was developed for this competition had an error rate that varied around 26%.

Then, in 2012 happened the most breathtaking moment in the modern history of convolutional neural networks. A team from University of Toronto entered the competition with a GPU based convolutional neural network model, that was inspired by LeCun's Lenet-5, and it achieved an error rate of 16.4%. The name of this network was AlexNet, and it inspired many developers to implement their own neural networks by using the ImageNet dataset and the ConvNets approach. As a result, the error rate nowadays has dropped to a few percent, and the overwhelming proportion of the models that are developed for computer vision tasks are based on convolutional neural networks.

2.1.4 The Convolutional Layer

A convolutional neural network is consisted by multiple layers. These layers are divided into three different types; the input layer, the output layer and as well as multiple hidden layers. An example of this hierarchy is shown in figure 2.2. An application can access only to the input layer, in order to feed the input data to the neural network, and the output layer, in order to get the results of either the inference or training. These results, of course can be processed furthermore by the developers in order to develop exactly the task that they wish.

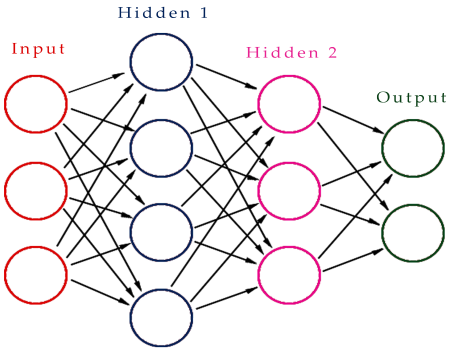

Figure 2.2: Structure of a CNN

In previous years, these hidden layers were limited only to the initial convolutional and downsampling layers. Nowadays, the modern convolutional neural networks are consisted by a variety of hidden layers, that each of them is im-

plemented to perform different tasks. Also, new types of layers are developed in a high rate, so the ConvNet models perform accurately the wishes of their developers. The following two subsections present the basic and the most essential building blocks that the structure of a convolutional neural network model is consisted of.

First of all, each image is represented by three parameters; Height, Width and the number of Channels. The number of channels is referred to the color model that the neural network is trained for. In this thesis, the images that i am going to use for the performance analysis are based on the basic RGB model - Red, Green, Blue -, so the number of the channels are always three. Models that are based on Grayscale, HSV, CMYK or other formats exist, but they are less common.

As its name indicates itself, the main building block of this type of network is the convolutional layers. Each convolutional layer has two main attributes; the input tensor, which has the form of a multidimensional vector, and the weight vectors. The shape of the vector is affected by the number of images, image's width and height and the number of channels. That means that the vector can be 2D, 3D or even 4D, depending on the task that the model was trained and is used for. Equivalent behaviour is existed in weight vectors as well. However, the width and the height are hyper-parameters, while the total number of the weight vectors that are used for each channel may be more than one. The task of a convolutional layer is always to extract features from an input image. These features are passed on the next layers for further processing. Convolution preserves the relationship between pixels by learning image features using sub-arrays of the input data. The sets of data that take part in this mathematical operation are the input tensor of the neuron and the weight vectors, which have smaller dimensions. During this operation, each cell of the weight vector is convolved across with the height

and width of the input tensor, and the result of these convolution is producing a 2D map, which shows the features that are included in the input tensor. That is why, the weight vectors are mentioned by many developers as filters. Because of these filters, a convolution layer can perform operations like edge detection, blur and sharpen.

In figure 2.3 there is example of three 4x4 input vectors of a neuron, and the corresponding three 3x3 weight vectors. Each pair of input and weight vector corresponds to one channel. In convolution, we overlay each weight vector on the top left of the input vector and we perform element-wise multiplication. We sum up the results of the multiplications, and then we sum up the outputs which resulted by the previous procedure for each channel. We apply a potential bias and we get one element of the 2x2 output vector. These procedure is repeated, by moving the filter to the right with a certain stride value, which in my example is one. Moving on, it hops down with the same stride value to the beginning of the image and the procedure is repeated all over again until the whole image is traversed and each element of the output vector is computed. By extending this example, across the multiple neurons of a convolutional layer and by increasing the number or the width and the height of the vectors, the execution of this type of layer is a tough process for any processor unit.

2.1.5 Other Essential Building Blocks

Besides convolutional layers, a convolutional neural network model contains more types of layers that are essential for their construction. In these section, i will present three more layer types which are used by developers in multiple building blocks. Certainly other types of layers are existed or custom layers are implemented, but these are the mandatory elements that a ConvNet should

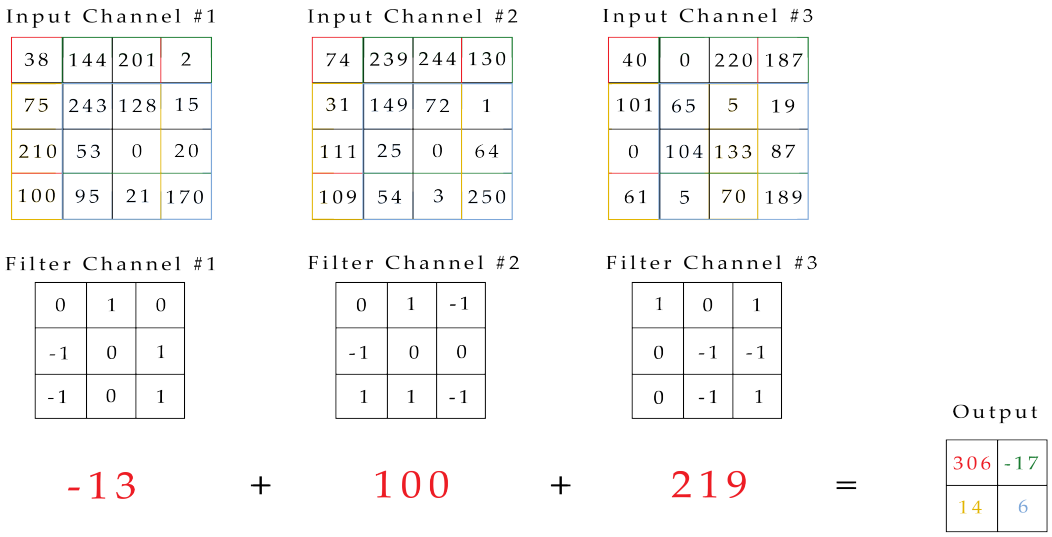

Figure 2.3: Convolution with 3 Channels

contain.

As shown in the previous section, a convolutional layer is a combination of multiplications and sums between different types of arrays. Because of these operations, a cell in the output array can have a negative value, as it seems in figure. Since, the real world data would want our convolutional neural network to learn, the negative values in the cells of the output array would lead to the opposite effect. The solution to prevent this problem is to utilize a Rectified Linear Unit(ReLU), after a convolutional layer. ReLU's purpose is to introduce non-linearity in a convolutional neural network by removing all negative values of the output arrays that a convolutional layer produce. This layer applies the non-saturating function $f(x) = max(0, x)$. By using this function, all negative values are setted to zero, where all positive values remain unchanged. Instead of ReLU, a developer can utilize other non-linear functions such as tanh, $f(x) = \tanh(x)$, or sigmoid, $\sigma(x) = (1 + e^{-x})^{-1}$, however ReLU is preferred due to its performance is way better than the other two, without significant penalty to generalization

accuracy.

One of the two initial layers that were used for the construction of neocognitron was the downsampling layer. Nowadays this layer is most known as pooling layer, but the main task remains unchanged. This kind of layer is used periodically between successive convolutional layers, to downsample an image, when the number of parameters are way too many. Other advantages of this type of layer are the reduction of the main memory that the neural network consumes but as well as the amount of the computations in the network. Nevertheless, the most important is that these advantages can be achieved without losing important information of the input image. There are two ways of pooling that are used commonly in our days; max pooling and average pooling. Max pooling is preferred because it performs better in practice compare to average pooling that was often used until recently. Pooling layers partition the input image into a set of non-overlapping rectangles and for each sub-region that was created, they output the average or the max of the elements that belong to the sub-region. One example of how a pooling layer operates is shown in figure 2.4 .

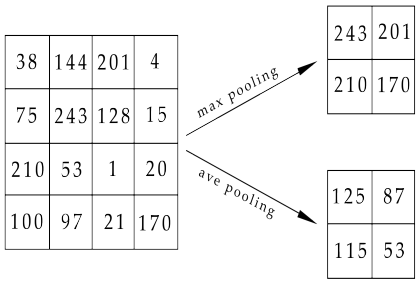

Figure 2.4: Max and Average Pooling

Until now, the input image has been converted into a suitable form, in order to be transmitted and to be processed from neurons throughout the network. Although in order to get the prediction results from the network, an additional layer is commonly used at the end of the structure. The name of this layer is Softmax and within this layer special functions and characteristics are implemented. The

first of them is called flatten, and commonly it is also implemented in a separate layer because of its particularity. This function collapses the spatial dimensions of the input image into the channel dimension. For example if the shape of a vector is 3D, flatten layer collapses it into an 1D array with the same size as the initial. The second characteristic of Softmax layer is the fully-connectedness. Because each neuron of this layer is connected to all the neurons of the previous layer, it is possible to learn the non-linear combinations of the high-level features that are represented by the outputs of the last connected layer, that in the most cases is a convolutional layer. Finally, in order to classify an image in the output labels, each neuron perform a Softmax technique. This softmax technique is a function that computes the probability of each output neuron, and it shows the probability distribution across the specified labels. The function that computes the probability in each neuron is $pr\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_i}}$.

2.2 Hardware Accelerated Data Processing

2.2.1 Hardware Acceleration through the years

Hardware acceleration is not a new concept for increasing the performance of data processing, on the contrary it has a long history of many decades. Hardware acceleration is used to perform functions more efficiently than is possible in a software on a general-purpose CPU. Instead, it takes advantage of the different processors that a system provide by executing some functions of a process more efficiently in the separate units. Processor units in our days can be a graphic processor unit(GPU), a digital signal processor(DSP) or even a processor that is specialized for computer vision tasks.

The first processor units could not execute any floating point operations. In order to perform such operations, special processor units were designed that could work as a coprocessor and expand the capabilities of the system, which in this case was the ability of a system to perform floating point operations. Even though, after a while these external units were integrated to the main central processors units, the concept of a coprocessor would stuck around for many years to come.

When the first computers were available, they were equipped with one single-core, stand-alone central processor unit. The limitations to its capabilities and performance, specially in multimedia applications such as audio, images, video, animation etc, led to the development of the very first coprocessor that was widely used alongside with the CPU, and its name was digital signal processor (DSP). The first stand-alone, complete DSPs were presented in 1980 with NEC μPD7720 and AT&T DSP1, and three years after Texas Instruments produced TMS32010, which was proved an even bigger success. With these new three processors the era of DSPs started, but the most important is that the coprocessors have become an integral part of the structure of any system.

Within digital signal processor architecture, some of the most general principles of computer architecture were first established, that are used till now from the most processor units. First of all, instruction sets that optimize operations like multiply-accumulate was introduced with DSPs. These operations are used extensively in all kinds of matrix operations like convolution for filtering, dot product etc. These type of operations are the key process for the execution of a ConvNet. Where a general purpose CPU would need multiple instructions in order to execute a single MAC operation, DSPs would handle them with only one instruction. For that reason, DSPs were commonly used to deploy the very first artificial neural networks that were developed. Also, DSPs introduced some fundamental instructions in order to increase the data parallelism within a single

processor, like VLIW, SIMD and superscalar architecture. In order to efficiently execute multiple data at the same time, it was needed to use special memory architectures that could fetch multiple data or instructions at the same time. So, various approaches were developed in order to achieve this task, like Harvard Architecture.

At the end of the 90s, systems stopped integrating DSPs. This is due to the fact that the newer central processor units started to integrate most of DSP instructions, and they could perform most of the supported operations by themselves almost as efficient as DSPs could. Also, the appearance of dedicated GPUs allowed to exploit techniques like SIMD in the best possible way. Special processor units are developed nowadays in order to perform very specific operations, like vision algorithms, deep learning functions etc.

2.2.2 Data Processing with Hardware Acceleration

The integration of some instructions that first used to DSPs into CPUs, was a smart move way back in the beginning of the new century. By expanding the capabilities of one processor unit, it has reduced the cost of one complete system and also has made the programmers to feel complacent, because any program can now run efficient into the new CPUs. But is it enough to expand the capabilities of a CPU by simply adding new instructions? The answer to this question came some years later, when the first multi-cores CPUs were introduced. By integrating multiple physical cores into the same silicon, the performance of a CPU is greatly accelerated, as different processes can be executed at the same time. Such CPUs are available by Intel and AMD. Their x86 architecture has based on clever pipelining and vectorization and also in the development of cache memories for fetching multiple data faster in order to run multiple tasks simultaneously and

efficiently. Other architectures also exists for different purposes. The most typical is the reduced instruction set, also known as RISC, that ARM uses for the development of their CPUs. These processors can be found primarily in mobile devices where the low consumption of power is critical.

Nowadays, a system can possess a graphic processor unit that works alongside with the CPU. The principle of the GPU is SIMD architecture, where multiple data can be processed with one single instruction. Aside from the data parallelism that this unit provides, GPUs can act also like a multi-core system, where many tasks can run at the same time. So, GPUs offer a very big advantage, they provide large-scale parallel data processing alongside with task parallelism. Due to the fact that GPUs achieve a large-scale of data parallelism, they mentioned as throughput oriented units. They sacrifice their latency by having slower clock speed, in order to achieve higher throughput on every cycle clock. However, in order to achieve such a high throughput, large sets of data must be fetched by the memory on every cycle clock. Dedicated GPUs, have their memory sets that it is called video RAM (VRAM), and the buses that connect the processor units with VRAM have extremely large bandwidth. This feature provides an important advantage over CPUs for deep learning inference tasks.

Aside from CPUs and GPUs, a system can contain more processing units in order to perform specific tasks. The most typical examples are the aforementioned vision processing unit and digital signal processor, tensor processor unit, neural processor unit, physics processor unit etc. The main difference between the first two units and those that are mentioned, is the flexibility that these units can provide. CPUs and GPUs are often referred as general purpose units, due to the fact that a variety of different applications can be run on them. Meanwhile, most of the other units are designed to perform very specific functions, for very specific tasks. However, the way that these tasks are executed are way more

efficient than if these functions were executed by general purpose units. These units belong to a special category that it is named ASIC, Application-Specific Integrated Circuits. The chips that are developed by this approach are highly customized and they can deliver the best performance amongst all the other approaches. On the other hand, the cost for the development of these chips is very high, while their flexibility is poor.

The final approach in order to accelerate data processing in our days is FPGAs, acronym that refers to Field Programmable Gate Arrays. These boards provide higher memory bandwidth than CPUs and as well as much lower power consumption compared to the aforementioned general purpose processing units. They have more flexibility than the circuits that are based on ASIC approach, due to the fact that they are programmable, and they can be altered after their development. They provide high efficiency, however they are a costly option. One big disadvantage of this approach is that FPGAs may struggle in executing floating point operations. Due to the fact that the models of ConvNets are trained in single precision float point arithmetic (FP32), in this thesis i will present a way of how we can overcome this side-effect.

2.3 Deep Learning Frameworks and Model Representations

While a number of deep learning frameworks exist in our days, such as Caffe, Tensorflow, ONNX , pyTorch, Keras, Deep Learning for Java, MXNet etc, most of them are not supported by the acceleration tools. In this section, i am going to introduce Caffe and Tensorflow, two frameworks that are the most well-known and are usually used for the development of a neural network model. Finally,

i am going to represent ONNX, an open format for representing deep learning models, that are trained from the above mentioned frameworks.

2.3.1 Caffe

Caffe framework [12] is an open source deep learning framework that was originally created in 2013, by Yangqing Jia during his PhD at University of California. Caffe supports a variant of layers of deep learning architectures, but most of them are directed to tasks like image classification, object detection and image segmentation. That's why many convolutional neural networks are developed in this framework, while the development of other types of artificial networks like recurrent is not recommended. Caffe is written in C++, with a python interface. According to Caffe's description, this framework can process over sixty million images per day with a single NVIDIA K40 GPU. That is about 1 millisecond per image for inference, and with more recent versions of libraries as well as newer hardware, this performance can be overcome. Moreover, models and optimization are defined as plaintext schemas instead of code, where the type of layers and their characteristics are defined. After a model is trained, two files are generated; the .caffemodel file contains all the weights that are adjusted during the training phase of the model, and the .prototxt file that contains the multi-layer structure of the model.

Caffe is used for academic research projects, startup prototypes and even large-scale industrial applications in vision, speech and multimedia. In Caffe's github[13] repository exists a custom distribution in order to improve performance when running on CPU, and at the same time a different custom distribution improves the performance on Intel and AMD GPUs and CPUs by using an OpenCL backend. Finally, an experimental Windows Setup is provided within

this repository. A distributed deep learning framework named CaffeOnSpark[14] was created by Yahoo!, by integrating caffe on Apache Spark. Because of the lack of development potential of recurrent neural networks, Facebook released in 2017 the successor of the Caffe framework; Caffe2[15]. Although, it was merged into PyTorch[16] at the end of Match 2018.

Despite of the low use of Caffe for developing new ConvNet models in our days, all hardware vendors support this framework from their corresponding acceleration tools. Thus, the numerous pre-trained models that are designed during the previous years, can be used by the accelerated applications in order to maximize the inference performance.

2.3.2 Tensorflow

Created by Google for research and production, Tensorflow[17] is the most well-known framework for developing applications that based on neural networks. It was released in 2015 and it is an open-source library under the github repository. Tensorflow is used for numerical computation and large-scale machine learning, it uses Python to provide a friendly front-end API application development, meanwhile it is executing these kind of applications in high-performance C++.

Developers can create with Tensorflow dataflow graphs. Each graph describes a multi-layer structure of how data moves through the nodes. The data, that pass through a node, are subject to mathematical operations that are described by the type of the node such as pooling, convolutional, ReLU etc. These operations are applied sequentially to the input data. While the nodes represent mathematical operations, each edge of the graph represents a multidimensional array or tensor. A big advantage of the use of graph is that it can run on multiple CPUs or

GPUs and as well as on mobile devices by executing the same code. In order to achieve high performance, all mathematical operations are written in C++ binaries, because C++ is more hardware friendly and accelerated operations can be achieved. Python is used to fill the gap between C++ binaries and the developer, and also high-level programming abstractions are provided by this programming language to hook them together. After a model is trained under the Tensorflow framework, it can be exported as a .pb graph, and then it can be deployed by any inference application. Because many models are developed and trained by using Tensorflow as its main framework, all manufactures designed their tools to support this framework, without the need of conversion to other framework.

Some new approaches that are currently developed by Google, are Tensorflow in Javascript[18] and Tensorflow Lite[19]. Tensorflow Javascript was created in order to allow a developer to import a pre-trained model to a browser for inference. This can be achieved by converting the existing model to Tensorflow.js format, either from Keras or Tensorflow. Besides that, using Javascript and some high-level layers API, a developer can also define and train neural network models entirely in the browser. On the other hand, Tensorflow Lite was developed in order to achieve low latency and a small binary size in machine learning inference in mobile, embedded and IoT devices. The common characteristic of all these mentioned devices is the low-power consumption, which is usually achieved by using an CPU that comes from ARM. For that reason, the toolkit, provided by ARM, fully supports Tensorflow Lite. In order to deploy a model in Tensorflow Lite, the initial Tensorflow model must be converted from the file format .pb to the the file format .tflite, which indicates the Lite version of the framework.

2.3.3 ONNX

A developer has a variety of frameworks available in order to develop the right model for his desiring task. For example, one of key features in choosing the right framework is whether a developer makes use of static graphs, that are implementing in Tensorflow and Caffe, or dynamic graphs, that are implementing in frameworks like Pytorch. In order to avoid the confusion of choosing the right framework, in September 2017, Facebook and Microsoft introduced a new representation format for deep learning models, that was called Open Neural Network Exchange (ONNX)[20]. The main objective of this new representation was to easily move between frameworks, by using as a middle step, this new format. The idea behind this project was to allow a developer to train his new model in a framework like Pytorch, and to deploy it in another framework like Tensorflow. ONNX format is a serialized representation of the model in a protobuf file. In order to achieve this chain of converting models from a specific format to another, tools have been developed in order to export the model from the initial framework to the ONNX representative format, and then to import it to the new framework that the developer wish. By using the flexibility that ONNX offers, a developer can choose the framework that it is best suited depending on the stage of the development. Other frameworks are optimized for faster training, other for inference in mobile devices and other for complicate network architectures. ONNX offers the chance to use the same model in different situations.

The other goal of ONNX was to create a format that would allow hardware vendors to improve the performance of artificial neural networks, by targeting only one representation. With the large plethora of frameworks that are existed and are often used by developers, it was an impossible task for hardware manufacturers to create tools that can target all frameworks in order to optimize the

execution of their layers. The solution was offered by the flexibility of ONNX, because the tools that are created can now target only one format, and so multiple frameworks are accelerated simultaneously. Intel and Nvidia for example, does not provide support in their acceleration tools for frameworks like PyTorch, but by converting the model in ONNX, it is possible to take advantage of these tools and achieve the maximum performance of deep learning inference applications.

Chapter 3

Acceleration tools

3.1 OpenVino by Intel

3.1.1 Introduction to OpenVino

In 2018, Intel announced the launch of OpenVino[21], or Open Visual Inference & Neural Network Optimization. The goal of this toolkit was the quick development of computer vision algorithms for edge computing in cameras, IoT devices etc. Most of the deep learning inference applications that OpenVino supports are based on Convolutional Neural Networks, and in order to maximize their performance, the toolkit extends the workloads across Intel's hardware. This is feasible by exploiting the accelerators that the corresponding hardware provides. In order to achieve such a task, within this toolkit Intel provides a library of functions and pre optimized kernels, providing maximum performance to inference applications. Moreover, it includes optimized calls for OpenCV, a library of programming functions that are used for the development of computer vision algorithms, and OpenVX, an API that was developed for cross platform

acceleration of computer vision applications. However, in this thesis the tasks that will be developed will be based on the first approach, and not on OpenCV or OpenVX.

According to a research from PassMark[22], the market share of the last generations of Intel CPUs, notably after the 6th generation, has ranged between 68.2% and 82.5%. This means that a large percentage of developers would have as central processor unit, a chip that has been manufactured by Intel. Meanwhile, due to the fact that CPUs are also general purpose processing units means that they can be a universal option for computer vision tasks, like image classification, object detection etc. OpenVino, takes advantage the flexibility of CPUs, and provides to a wide range of developers the ability to accelerate the performance of their inference application, by just integrating the optimized functions that the toolkit contains. Meanwhile, the CPUs from 6th generation and after contain powerful integrated graphic cards in their SoC, so by using OpenVino, deep learning inference at the edge becomes an easier task.

In order to accelerate further the deep learning inference applications, special integrated circuits were developed by Intel, that can be used alongside with OpenVino. Arria 10 GX FPGAs deliver high performance deep learning inferences at low power and latency. There are often used by autonomous vehicles, robotics, IoT and data servers. Meanwhile, a very interesting integrated circuit that is based on ASIC technology was developed by Intel. Intel's Neural Compute Stick[23] is a special vision processing unit (VPU) that enables visual intelligence at a high compute per watt. It supports all typical computer vision tasks, but also deep learning inferences and camera processing. It delivers high performance of inference applications and can be used on systems that low power consumption is a primary requirement.

3.1.2 Deploy an app using OpenVino

The deployment of a deep learning inference application that is based on OpenVino toolkit is divided into two separate parts. The first part is the model optimization, and it occurs before the deployment of the main application. The exported files that are resulted by running the Model Optimizer, are used as inputs in the inference application. So, there is a dependency relationship between the two aforementioned procedures, due to the fact that the maximization of the performance that OpenVino can achieve is feasible only by using the exported files as the input model in the inference application. A complete workflow of these procedures is shown in figure 3.1.

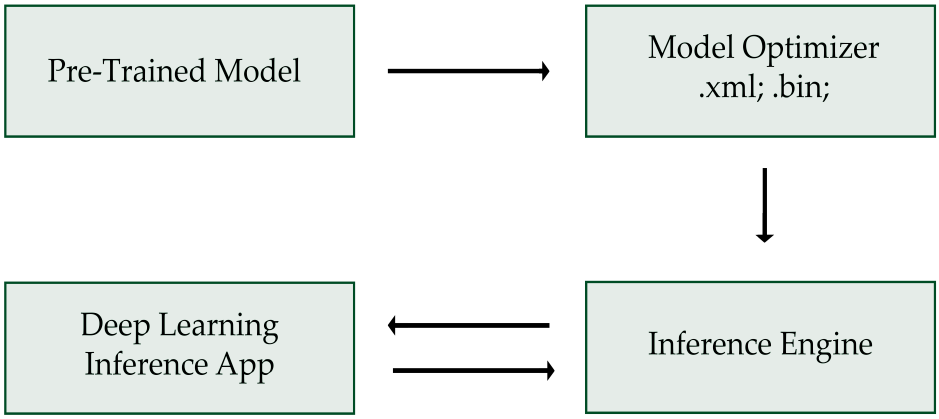

Figure 3.1: Workflow Procedure for OpenVino

Model Optimizer is a cross-platform command line tool, that converts a model from the framework environment that is trained, to the deployment environment of OpenVino. Furthermore, it performs static analysis, and as well as modifies a model for optimal execution on end-point target devices. Model Optimizer con-

verts models that are trained in the most well-known frameworks, and specially in Caffe, Tensorflow, ONNX, MXNet and Kaldi. Although, OpenVino does not support every type of layer that can be implemented in a model by using the aforementioned frameworks, it supports a large variety of them and almost every pre-trained model can be converted to the optimized form. However, if a layer is not supported by the toolkit yet, Intel provides the ability to pass the model through the optimizer by implementing custom plugins for the unsupported layer. By using the Model Optimizer in a model that was developed by one of the supported frameworks, two new files are produced that constitute the Intermediate Representation of the model. This pair of files are

- .xml; which describes the structure and the layers of the network and,
- .bin; which contains the weights vectors and biases binary data.

The Intermediate Representation can be used by any program for load, read and inferred with the help of the Inference Engine that Intel provides within the libraries of OpenVino. Aside from the parser that model optimizer provides in order to check if a model is valid, the conversion procedure discards all layers that are completely useless during inference, but are useful during the training phase of the model and they can't removed from the beginning. One other ability that the Model Optimizer offers is that it can produce the Intermediate Representation of a model in a precision depending on the processor unit that we want to deploy the application. If a CPU is selected then the precision must be FP32, while for Intel Movidius Myriad VPU (Neural Compute Stick), the Intermediate Representation must be in FP16 precision.

The second part of the process is the development of the deep learning inference application. In order to maximize the performance, an inference application must implement the libraries that are provided through OpenVino and use the

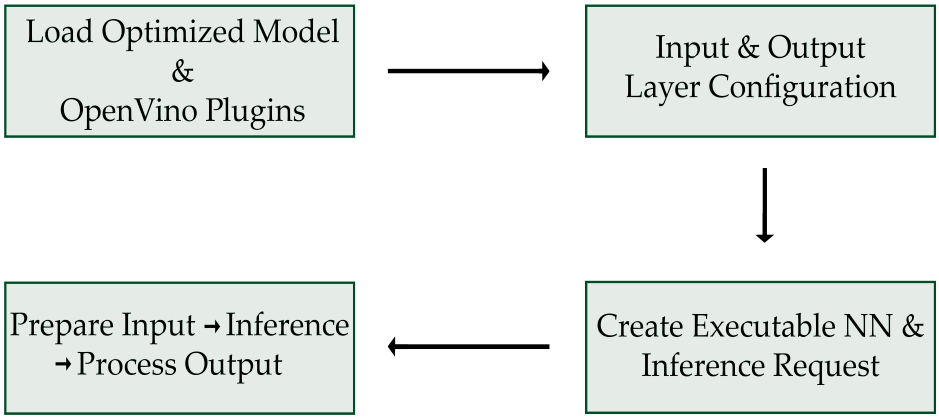

Figure 3.2: Workflow of the application

optimized Intermediate Representation of the model, that occured by Model Optimizer. A workflow of the process and how the inference application can be built is shown in figure 3.2. As shown in the figure, the workflow is divided into the below steps;

- The first step of the process is the load of the plugins that are implemented by OpenVino in order to manage the available devices for the inference process, as well as the network model in its Intermediate Representation form is read by the Inference Application.

- Then, the inference application must request the input's and output's layer information for the model that will be referred. This step provides information about the dimensions of the input and output and as well as the number of channels, in order to feed the data in the right form and also to check that the output layer provides the results that are expected from the model specifications. In this step, some other information can be configured for these two layers such as the precision of the input and output data, the

batch size, a pre-process for the colour format or the layer setup. In the latter two options, the Layer Setup is usually NCHW, - Number of images - Channels - Height-Width- and the pre-process converts an image from an initial colour format like RGB to models supported format, which is BGR. The default options for these choices are FP32 for precision, NCHW format and no pre-process.

- In the third step, the executable neural network is created by providing the converted model and the target processor unit in the appropriate function that OpenVino provides. Furthermore, an inference request must be created by using the executable neural network.

- Last but not least, the developer must prepare the input data, processed to fit the specifications that the model has and he configured during the second stage. Then the inference can start by calling the appropriate function and the produced results can be further processed achieving the desired task.

3.2 TensorRT by Nvidia

3.2.1 Introduction to TensorRT

Over the last years, GPUs are always used in order to accelerate the execution of deep learning inference applications. They exploit the data parallelism through SIMD architecture, and they offer increased throughput compared to other general purpose processor units. However, the utilization of a GPU in the execution of a neural network is not optimal and there is room for big improvements in order to take full advantage of the capabilities of these processor units. Nvidia has based on this approach and introduced TensorRT[24; 25], a platform for

high-performance deep learning inference that provides the tools for an optimal development of an application.

TensorRT includes a deep learning inference optimizer and runtime that delivers low-latency and high-throughput for deep learning inference applications. According to NVIDIA, applications that are developed based on TensorRT for inference, can perform almost 40 times faster than if these applications were implemented on CPU-only platforms. The main reason behind this speedup is the optimization of a neural network model by inserting it in a chain of procedures, before the inference execution takes place. TensorRT is built in CUDA, which is Nvidia's parallel programming model for its GPUs. By exploiting the libraries, development tools and technologies that CUDA offers for deep learning inference, the development of such high-performance tasks is feasible. TensorRT can be used by any GPU that was produced by Nvidia and support CUDA.

Beside GPUs, Nvidia produces a family of boards for the development of Artificial Intelligence Applications. This is the Jetson Family, and the most known model is Jetson TX2[26]. Especially this board is recognized as the fastest and the most power efficient embedded device for AI computing. This includes the deep learning inference applications, where the performance in relation to power consumption is almost optimal. These embedded platforms feature an integrated Nvidia GPU, so the development of inference applications by using TensorRT is feasible.

3.2.2 Deploy an app using TensorRT

Contrary to the two-part process that the developer should follow in Intel's toolkit, the development of a deep learning inference application in TensorRT is much simpler and is limited only in the utilization of functions that this toolkit

offers. The workflow that a developer could follow in order to develop an application using TensorRT is shown in the figure 3.3. Although the frameworks that TensorRT supports are limited to Caffe, Tensorflow and ONNX, a large variety of layers can be supported for the different structure of each model. However, if a layer is not supported by TensorRT, the tool provides the ability to create the unsupported layer and integrate it within the application in order to execute the pre-trained model successfully.

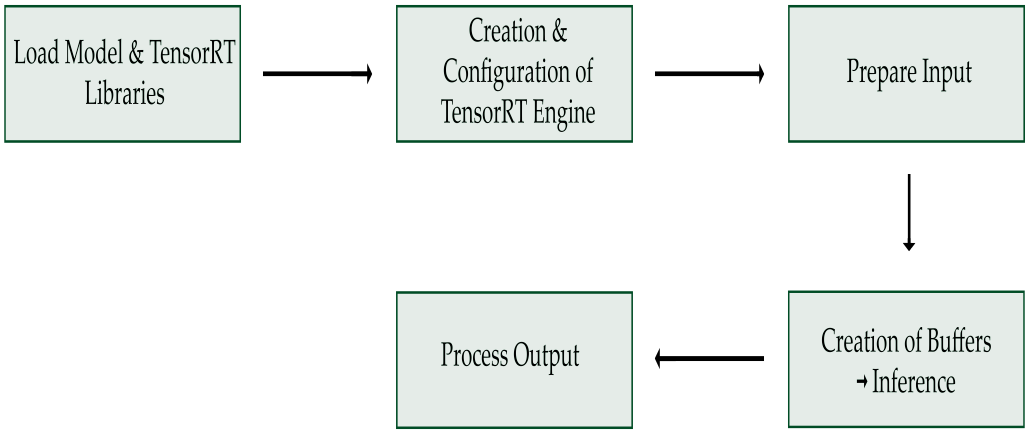

Figure 3.3: Application Workflow with TensorRT

The aforementioned figure shows that an application that is based on TensorRT is also divided into stages, and most of them are similar to OpenVino steps;

- The first stage of the process includes the loading of the libraries that TensorRT implements for the inference process, and as well as the pre-trained model that the developer wishes to accelerate its execution.
- In the next stage, the creation of the engine that will accelerate the execution of the convolutional neural network takes place. The first step is to

check whether the structure of the model is valid and fully-supported by TensorRT or not. If it is valid, then the parser populates a network object in TensorRT. In the second part of this stage, the optimized engine for the target platform is created through a builder, using the network object in TensorRT. The builder is a function, part of the TensorRT library, that creates the engine and the developer can provide critical information about the inference procedure. He can set the precision, the batch size of the inference and the maximum GBs of workspace that the target platform can use.

- After the building of the engine, the inference can be performed. As it happens with OpenVino, the first task of this stage is the preparation of the data. Image resizing, converting the colour format to BGR or other procedures must take place in order to feed the data to the network matching the specifications of the pre-trained model.

- After the preparation of the input data, the inference can be performed. By using CUDA, the developer must create the buffers that feed the data into the network and take the results from the output layer. These buffers must be equal to the dimensions of the input and output layer, otherwise an error might be happen. By executing the appropriate function providing by TensorRT, the inference takes place and the results are exported by the output buffer. Then, the results are at the developer's discretion on how to exploit them.

3.3 NN SDK by ARM

3.3.1 Introduction to Neural Network SDK

A large amount of mobile devices, embedded platforms and as well as FPGAs have as a central processor unit, a processor that it is constructed by ARM. Due to their reduced instruction set compare to the x86 architecture of Intel, these CPUs are perfect for devices that low power consumption is an essential condition. However, these processors nowadays are very powerful and can execute complex functions and algorithms efficiently. That includes deep learning inference applications. For that reason, Arm introduced the Neural Network SDK[27; 28] (NN SDK), a set of software and tools that enables machine learning workloads on the power-efficient devices.

In order to exploit the capabilities of Arm's NN SDK, a certain library must be built into the target platform, the Arm's Compute Library[29]. Compute Library contains a comprehensive collection set of functions for the Cortex-A family of CPUs and for the Mali family of GPUs. Both of processor units were designed by Arm, so by using the low-level optimized functions that the Compute Library provides, the acceleration of algorithms and applications is a feasible task. Although, Arm NN SDK can be used on any system that contains a Cortex-A CPU or a Mali GPU, it can also be used on Cortex-M microcontrollers but with very limited capabilities. The latest version of this tool provides support for Tensorflow and Tensorflow Lite, Caffe and ONNX pre-trained models. However, the supported layers for these frameworks are fewer than the other two tools mentioned, and a big disadvantage is the lack of implementation potential of custom unsupported layers.

Figure 3.4 shows a simple scheme of how a system "communicates" by using

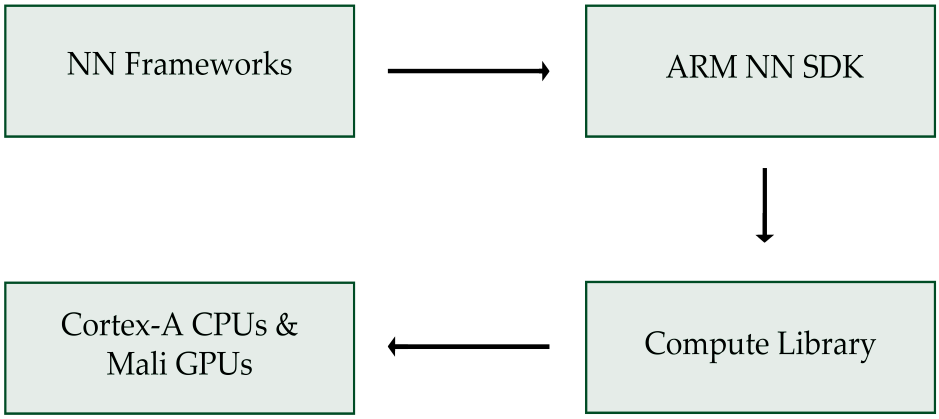

Figure 3.4: ARM Communication System

ARM NN SDK for deep learning inference application. An inference application, that utilizes models from the aforementioned supported frameworks, implements the NN SDK in order to enable the execution of inference tasks across Arm's hardware. The functions from NN SDK API, that enable the execution of a model, utilize the optimized functions that Compute Library provides for the targeted processor unit. Thus, the execution and at the same time the acceleration of the inference application using processor units from Arm is feasible.

3.3.2 Deploy an app using Arm's Development Kit

The integration of NN SDK into an inference application is a simple and easy task. The main stages of the inference application are very similar to the previous two toolkits, so a workflow is needless to be given; In the first stage, the integration of all libraries that the application needs from the SDK is mandatory. Also, a developer must import the model that has file extension of one aforementioned

supported frameworks. It is recommended that these files must be in a binary format, because they are a lot smaller than the text format that NN SDK supports. The debugging of the model is impossible due to the fact that binary files are not readable, but as long as a developer use pre-trained models, he usually do not need this ability.

In the second stage, a parser is used in order to check if the model is valid and full-supported by the tool. If this procedure succeeds, then the network is created for the inference stage. A major difference between this tool and the previous two, is that the developer cannot make any changes in the network as it could happen in the previous two occasions. So, the batch size and any other parameters must be strictly specified into the network structure before the creation of the imported binary file.

The inference stage begins by taking the binding points of the input and output layer, for feeding and getting the results respectively. Then the optimizer is called which takes into account the two parameters, the target platform, CPU or GPU, and the network that was built during the previous stage. Also, it is very important to carefully prepare the input that will be fed into the network, as many faults can occur, especially where none changes can be performed during the network build phase. Finally, the inference can be performed by the application, and the results are exported for further processing.

3.4 Edge AI by Xilinx

3.4.1 Introduction to Edge AI

In subsection 2.2.2 , i described that FPGAs are always used for accelerating data processing by maintaining at the same time low power consumption. Due to the fact that these platforms are programmable, they can customized to perform deep learning inference applications applications. In order to facilitate the development of these applications, Xilinx introduced Edge AI[30], a platform that provides all the comprehensive tools that can enable and accelerate the execution of ConvNet models. Figure 3.6 shows all the components consisting by Edge AI, and the stages that are integrated inside the inference application. This approach can be used on boards that are based on Zynq Ultrascale+, and more specifically for ZCU102, ZCU104 and Ultra96.

In order to enable the deep learning inference on the aforementioned boards, Xilinx has designed a special Deep-learning Processing unit, figure 3.5, in order to support a wide range of edge AI applications with low latency in their execution. Inside the DPU, two engines are implemented in order to accelerate the execution of operations. The first one is the Convolution Engine, that performs convolution calculations, and the second one is the Misc Engine which perform calculations such average or max pooling, ReLU. Furthermore, DPU integrates a high performance scheduler, that performs smart instruction merging or splitting and dynamically adjusts the FIFO depth in order to fit the network requirements. This DPU can execute the most common layer types and operators and by using hardware acceleration the capabilities of the underlying Xilinx FPGA architecture are exploited. Also, the data are fetched by RAM with High Speed Data tubes, in order to decrease the overall inference time. Finally, the CPU and DPU

are used simultaneously in order to execute different parts of the application.

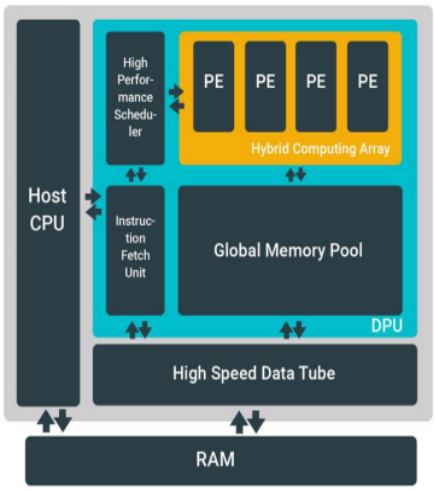

Figure 3.5: DPU's architecture [31]

3.4.2 Deploy an app using DNNDK

The execution of a pre-trained model in the integrated DPU requires a two-step modification. None of these steps can be avoided, as the tools are essential to check the validity of the pre-trained model, and as well for the conversion to a special file format for their execution by the DPU. Moreover, a tensor-based APIs are provided in order to enable easy inference application development. A detailed description is given in this subsection for the hierarchical structure of procedures that must be followed for the deployment of a deep learning inference application.

It is clear that FPGAs struggles to perform floating point executions. All neural networks are trained either to FP32, most common occasion, or to FP16, so the exported pre-trained models have the same precision. Deep Neural Network Development Kit[32] is a set of tools that helps the smooth execution of a pre-trained model in Xilinx FPGAs. The first tool from the chain of procedures that the model must undergo is Decent, or Deep Compression Tool. Decent is a tool

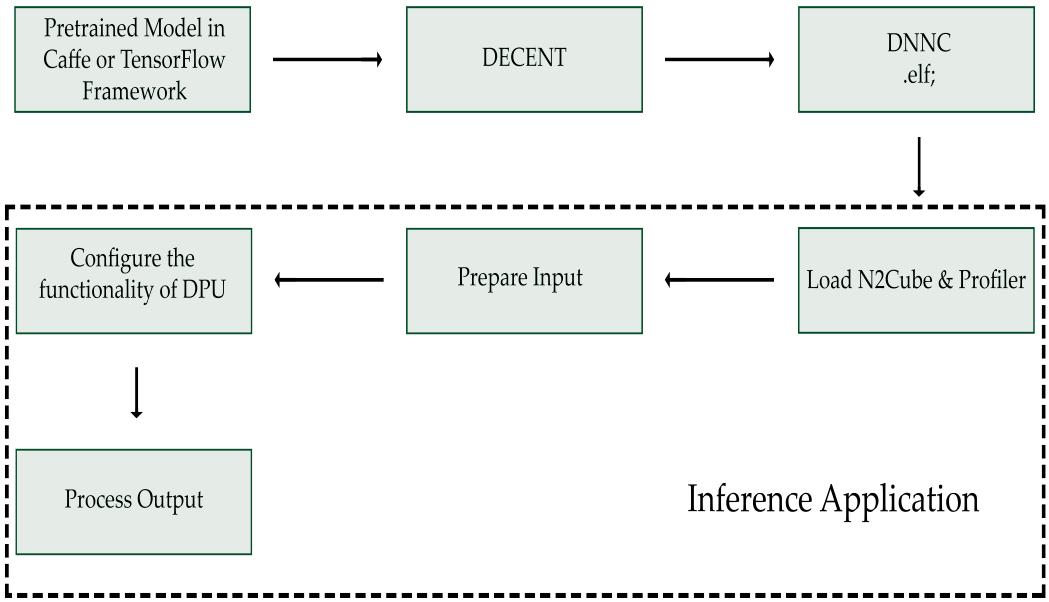

Figure 3.6: DNNDK deployment

that converts a pre-trained model with floating point precision into a model with INT8 precision. This modification is critical in order to achieve the maximum performance on AI inference applications. Even though by compressing the initial model the computing complexity is reduced a lot, the impact of this procedure to the accuracy is minimal. Thus, the new fixed-point network model requires less bandwidth, providing faster speed and higher power efficiency.

The second tool is DNNC, or Deep Neural Network Compiler. As its name indicates, DNNC is a compiler responsible for parsing a model that has a Supported Framework format and generates an Intermediate Representation. The Intermediate Representation is going to be used in the following inference application. DNNC is also responsible to perform sophisticated optimizations on the model such as layer fusion, instruction scheduling and reuses on-chip memory for a more efficient execution by the DPU. Contrary to what happens in the

aforementioned parsers of the other toolkits, DNNC fails when a non-supported layer is implemented into the model, but also indicates the layers that can't be processed by DPU due to floating point operations, like Softmax Layers. The latter layers must be implemented and executed on the CPU instead of DPU, but these parts cannot be accelerated by this procedure. The model or the parts that are successfully compiled by DNNC are converted to special .elf files, with high-efficient instruction set and data flow, that are read by the special DPU kernel for optimal execution. So the exported .elf file or files, depending on whether the whole model was compiled or parts of it by DNNC, are imported by the inference application to perform the wished task.

The development of a deep learning inference application looks pretty much with the above mentioned cases. The procedure is divided into the typical three stages of input data pre-processing in order to match the specifications of the pre-trained models, the execution of the model for the inference and the post-processing of the exported data. The utilization of N^2Cube, a lightweight set of tensor-based APIs, is necessary to achieve the second stage of the procedure. This library is part of DNNDK core library, and implements the functionality of DPU loader and encapsulates the system call to invoke the DPU drive. So task scheduling, monitoring, profiling and resources management is achieved for the DPU by using N^2Cube in the source code of the application. Finally the Performance profiler enables the monitoring in-depth of the efficiency and utilization of the developed AI inference application.

Chapter 4

Benchmark Description

4.1 ConvNet Tasks and Models Description

4.1.1 Image Classification

The first application of the performance analysis is based on Image Classification. Image classification is referred to a process that can classify an image to a class of predefined objects, according to its visual content and features. Even though, image classification seems quite trivial as a concept, for the development of computer vision applications in our days still remains a very challenging task. For that reason, an annual competition has held for evaluating algorithms for image classification, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)[33]. ILSVRC has also an another motivation, to measure the progress of computer vision models. Many models that are used for this task came from implementations that took part in this competition in the previous years. The task, that is developed for this thesis, is based on a typical ImageNet challenge, where the goal is to classify a variety of images into 1000 categories.

- **AlexNet**

Alexnet[34] competed in ILSVRC in 2012, and its participation was a game changer for the future of computer vision models. It achieved a top-5 error of 15.3%, which was almost 11% percentage points lower than the runner up model achieved. From that year, the models that achieved the lowest top-5 error are based on convolutional neural networks. AlexNet is mainly consisted by 11x11, 5x5 and 3x3 convolutional layers, and ReLU activations that are attached after every convolution. It also consists multiple max pooling layers and a single fully-connected layer. 91.23% of total 726.79M multiply-accumulate operations are happened in the convolutional layers. The total size of the pre-trained model is 232.57MB, it contains 60.97M parameters and can achieve 57% accuracy on ImageNet dataset. The image resolution that it is used by this model for this task is 227x227 pixels.

- **SqueezeNet1.1**

By using almost 50x fewer parameters than AlexNet, and in particular 1.24M, Squeezenet1.1 is a small ConvNet model that can achieve the same accuracy of 57% in ImageNet dataset. This version of Squeezenet is an improved successor to the initial model that it was presented by the paper[35] in 2016. The total size of the pre-trained model is only 4.72MB and the image resolution is the same as AlexNet, 227x227 px. Meanwhile, the total multiply-accumulate operations for this model are 387.75M, almost the half that needed for AlexNet to process a single image.

- **GoogleNet/Inception**

GoogleNet or Inception-V1[36] was the winner of ILSVRC competition in 2014. As its name indicates, it was developed by Google and achieved a top-5 error rate of 6.67%. This was a tremendous improvement compared to the AlexNet model that was competed two years earlier. This was the first time that a ConvNet model could challenge the human level performance in image classification. GoogleNet has an accuracy of 68.7%, and the images that are used by this model have resolution 224x224 px. GoogleNet performs convolution with three different sized filters, 1x1 , 3x3 and 5x5. Additionally max pooling is used in order to downsample the image. GoogleNet has reduced the number of parameters by a factor of 8.71x compared to AlexNet, while the pre-trained model is 51.05MB. Furthermore, the total number of multiply-accumulate operations are 1.59G, a number that far exceeds the already mentioned models.

Inspired by Inception-V1, three newer versions have been developed, in order to improve the already existing model. In this thesis, i will also use for the performance analysis the final version; Inception-V4[37] which was introduced in 2016. Inception-V4 achieved a top-5 error rate of 3.8%, an improvement of almost 3% from the initial GoogleNet model, meanwhile the accuracy is reaching at 83.3% in ImageNet dataset. The resolution of the images that are used by this model for this task is 299x299 px. However, in order to achieve the mentioned improvements, Inception-V4 performs 12.27G MAC operations, meanwhile the number of parameters is 42.62M. The size of the pre-trained model is 162.87MB.

- **VGGNet**

The runner-up of the ILSVR competition in 2014 in classification task was VGGNet[38], losing only by the first version of Inception model. VGGNet was developed by the Visual Geometry Group from University of Oxford. That year, was the first time that two models obtained error rate under 10%. Despite of not winning that year's competition, VGGNet is the most prefered choice in the community for extracting features from images. The reason behind this preference is that VGGNet models consist a limited number of convolutional layers with 3x3 filters, and generally its structure is much simpler than other known models. The most known versions of VGGNet are VGG-16, which took part in competition, and VGG-19. Both of them will be used for the performance evaluation.

VGG-16 is consisted by 16 layers that are mainly divided into convolutional layers, max pooling layers and fully connected layers. VGG-16 process images with resolution 224x224 px, reaching an accuracy of 70.5% on ImagetNet dataset and 8.8% in top-5 error rate. Despite of its simple structure and the limited number of layers that was used for its construction, VGG-16 performs 15.47G multiply-accumulate operations and at the same time the number of parameters is reached 138.36M. Almost 99,21% of MAC operations take place in convolutional layers. These characteristics of this model can be a bit challenging to handle even from the accelerated applications that were developed for this thesis. Meanwhile the size of the pretrained model is 527.79 MB.

The other most known model by the VGGNet family is VGG-19. VGG-19 has similar characteristics with VGG-16, but as the name indicates, it is consisted by 19 layers. The very first model that it was created by the VG group, was consisted by only 11 layers. In order to improve its accuracy, they started to add more convolutional layers. When they developed VGG-19, they observed that this model was starting converging with VGG-16, and there was not improvement in accuracy. So they stopped adding more layers. This pre-trained model is slightly

bigger than its predecessor, 548.05MB. Also, VGG-19 performs 19.63G MAC operations, and has 143.67M parameters. That means, the applications that are based on this pre-trained model, have slightly lower performance than apps that based on VGG-16.

- **ResNet**

ResNet, or Residual Neural Network[39], was the model that won ILSVR competition in 2015. It achieved an extremely low top-5 error rate of 3.57% in image classification and it was the first model that managed to win the human level performance in ImageNet dataset. The model that took place in competition was consisted by 152 layers, but there are more variants of this network. The other known ResNet models are those with 50 and 101 layers. In this thesis, all three of them will be used for the performance analysis.

All of the three models accept images that have resolution 224x224 px. The model with the fewer number of layers is ResNet-50. This pre-trained model has size of 97.72 MB, while the number of parameters is 25.56M. The accuracy that can be achieved by this model in ImageNet dataset is about 75.5%, which is much better than AlexNet and Squeezenet, and slightly better than VGG-16. ResNet-50 performs only 3.87G multiply-accumulate operations, a number that is a significant smaller than the previous networks or from the other models of this family.

The other most known model from this family is ResNet-101. ResNet-101 has very similar structure to ResNet-50, but with the difference, as its name indicates, in the number of layers, which is 101. The size of the pre-trained model that i am going to use for the analysis is 170.39MB. Meanwhile the number of parameters is

44.55M, almost 20 million more than ResNet-50. The number of MAC operations is 7.59G, and over 90% of them takes place at convolutional layers, as it happens with the almost every model that based on convolutional neural networks. The accuracy that can be achieved by this model in Imagenet dataset is about 76.4%, which is a bad trade off compare to the lower performance that this model has due to the high number of MAC operations and parameters.

The final model and at the same time the most known by this family of networks is ResNet-152. This model consists 152 layers, and it can achieve the best accuracy compared to the other two models from this family, which is slightly over than 77%. The number of its parameters is 60.19M, while the total number of multiply-accumulate operations is 11.3G. The total size of the pre-trained ResNet-152 model is 230.26MB. As it seems, ResNet-152 has the poorest performance among the models of the Resnet family. Despite of the poor performance that the ResNet models provide compared to other models, many developers are willing to make that "sacrifice", in order to take advantage of the very good accuracy that these models provide for the image classification task.

- **DenseNet**

DenseNet or Densely Connected Convolutional Networks[40] was first mentioned in Computer Vision and Pattern Recognition conference in 2017, where the authors won the best paper award. Despite of not taking part in ILSVR competition, DenseNet networks are commonly used by many developers. These types of networks are so deep, as it happens with the ResNet models, but they require fewer parameters. In this thesis, three different types of DenseNet models will be used which are DenseNet-121, DenseNet-161 and DenseNet-169. The

input dimensions of the images are 224x224 pixels.

DenseNet-121 consists 121 layers, and it is the narrowest of all the three other networks. It is mainly consisted by 1x1 or 3x3 convolutions alongside with ReLU activation and Batch Normalization. DenseNet-121 requires 7.98M parameters and the total number of multiply-accumulate operations is 3.08G. Despite of the fact that this model has more layers than the corresponding narrowest ResNet model, its performance seems to be better, while the accuracy for ImageNet dataset is 75%, slightly worse than ResNet-50. The total size of the pre-trained model is 30.81MB.

The model of this family that can achieve the best accuracy, which is 77.8%, is DenseNet-161. In order to achieve the improvement of 2.8% compared to DenseNet-121, forty more layers was need, while the number of parameters has been increased by a factor of 3.6x. The total size of the pre-trained model is 110.32MB. In addition, because of the more parameters and as well as of the forty more layers, 8.52G MAC operations is needed in order to classify an image. This is a significant trade-off in order to achieve the mentioned improvement.

The final model of the DenseNet family is DenseNet-169. DenseNet-169 has similar structure with the corresponding models of this family. Despite of the eight more layers than the latter model, DenseNet-169 has 14.15M parameters and requires 3.72G multiply-accumulate operations. These two features has as a result the accuracy to drop from 77.8% that the DenseNet-161 has to 76.4%, but the performance of the model is significantly improved. The most important about DenseNet-169 is that it can achieve a similar accuracy as ResNet-101, but the MAC operations that are required and as well as the number of parameters have been greatly reduced. Despite the large number of layers, DenseNet models are usually preferred, because they provide such a good accuracy in image classification task and at the same time their performance is better compared to

ResNet models.

4.1.2 Object Detection

The second task that was developed for the performance analysis is Object Detection. In computer vision and image processing, object detection refers to the process that deals with detecting instances of semantic objects. These objects belong to certain classes, that were defined at the training phase of the model and they also depended by the training dataset. Alongside with image classification, Object Detection is one of the most well-known and challenging tasks for the developers, while a large number of applications and models are developed for this specific task. In this thesis, we will try two different approaches, Single Shot Detection or SSD, and You Only Look Once or YOLO.

- **Single Shot Detection**

Single Shot Detection was first proposed by the paper[41] in 2015. By using SSD, it takes one single shot to detect multiple objects within an image. There are two models that was developed by using this approach. SSD300, which uses images with resolution 300x300 pixels, and SSD512, which uses images with resolution 512x512. Both of them will be used for the Object Detection task.

Both models, were trained under VOC0712 dataset[42] (Visual Object Classes), which came from an annual challenge for object detection that was ended in 2012. In this dataset there are 20 different classes like person, bird, cat, dog, aeroplane, bicycle, television, sofa etc. The main goal of SSD models is to classify the objects of an image in one of those 20 categories. For that reason, object detection

and image classification have a lot in common. SSD300 and SSD512 are based on VGG-16 in order to extract the feature maps, a model that was mentioned and used for image classification. So, in order to extract these features, convolutions with 3x3 filters are used. Moreover, after the extraction of the features, SSD continues to use 3x3 convolution filters for each cell to make predictions. SSD300 requires over 31.37G multiply-accumulate operations to perform object detection task, while an overwhelming amount of these operations are required by convolutions. Also SSD300 has more than 26.28M parameters and the size of the pre-trained model is 100.28MB. The accuracy in object detection is measured as the mean average precision (mAP), which is 75.8.

SSD512 is another type of Single Shot Detection, which take bigger images as an input, compared to SSD300. As a result, this model has an improved mean average precision by 2.7 units. Due to this model makes more predictions in order to classify the objects in the right class, it requires over 90.21G MAC operations in order to perform its task. So the trade-off between an improved mAP by taking bigger pictures, is the low performance compared to SSD300. Meanwhile, the parameters of SSD512 is slightly over 27.19M, and the size of the pre-trained model is 103.73MB.

- **You Only Look Once**

The second approach that i am going to use for the object detection task, is YOLO[43]. Most of the models that are developed for object detection, don't look at the complete image, but they look only to parts where there is a high probability of containing one or more objects. On the other hand, YOLO divides the image into smaller regions, and predicts the bounding boxes and probabilities

for each region. The bounding boxes are weighted with the probabilities, and when a class probability is above a predefined threshold value, the bounding box is selected and used to locate the object within the image. In this thesis, i will use Yolo V3[44], which is the the newest version of this approach and it was trained under the COCO dataset.

COCO dataset[45] is very similar to VOC0712, but it contains eighty different classes. COCO dataset is mainly used for object detection and as well as for image segmentation. Some of the images that provides have 4 or 5 objects, so an accurate mAP can be provided by running images from this dataset. The images, that were taken by this dataset, have been resized to 416x416 pixels. Meanwhile, in this version of YOLO some improvements have been made in order to increase the accuracy compared to the previous two versions.

The mean average precision of YOLO v3 in COCO dataset, is 55.3. This number may seem smaller than the corresponding mAP that was presented in SSD, but the use of a different dataset is a major factor. A typical example is that SSD300 and SSD512 have mAP 43.1 and 48.5, when these models are trained using COCO dataset. Not much is known for YoloV3, only that its structure includes 106 layers of fully convolutional underlying architecture.

4.1.3 Image Segmentation

The third and final task, that was developed for the performance analysis of the acceleration tools is based on image segmentation. Image segmentation is a process of partitioning a digital image into multiple segments. The goal of this analysis, is to simplify the representation of an image and convert it to something more meaningful and easier to analyze. In contrast to image classification, the applications that are developed for image segmentation have as a goal to get a

pixel-level understanding, which means that each pixel of the image has to be classified in one class, accordingly to the classes of the dataset that was used to train the model. Image segmentation is mainly used for medical imaging and traffic system controls. Due to the pixel analysis that it is required by this task, image segmentation is one of the most challenging tasks in computer vision, and even the hardware that it is used nowadays has difficulties executing applications that based on this task. For the analysis, two different kind of models are used, which are voc-fcn8 and Dilation. Each model has been trained to a completely different dataset, as will be shown below.

- **VOC-fcn8**

Voc-fcn8 is a model that was first described in the paper [46], by Jonathan Long in 2015. As its name indicates, this model was trained under VOC0712 dataset, which classes were quoted in the section where i described SSD. Voc-fcn8 has two other variants, voc-fcn16 and voc-fcn32, but according to authors fcn8 achieves the best results in the specific dataset. For that reason, fcn8 pre-trained model will be used in order to perform the image segmentation task.

FCN models have its root in VGG-16 model, as it was used to initialize all the models from this family. Aside from all the characteristics that was inherited by VGG-16, fcn8 uses deconvolution in order to upsample and get the output size larger. This must be done in order to get the original size of the picture in order to calculate the pixelwise output and classify an object to its class. Despite of the fact that deconvolution performs the opposite task in from convolution, it requires only 0.0032% multiply-accumulate operations of the total 181.55G, that it is required of the whole model. Of course, a tremendous portion of these

operations are required by convolution layers, and specifically the 99.68%. The size of the pre-trained model is 513.04MB and it has 134.49M parameters. The mean accuracy of classifying an object to the right class for fcn8 in VOC0712 dataset is 75.9%.

- **Dilation**

The second model, that is used for the image segmentation task, is Dilation. Dilation was introduced by the paper [47] in 2016. Unlike from all the previous models that was mentioned, the dataset that this version of Dilation uses is named Cityscapes. Cityscape Dataset[48] is mainly used for image segmentation, and contains high pixel-level annotations of thousand frames that came from real recorded sequences in street scenes from over 50 different cities. This dataset is often used for applications like traffic system controls or tasks that focus on semantic understanding of urban street scenes, and contains 30 different classes.

For this task, Dilation uses images with resolution 1396x1396, so it is undoubtedly the most challenging model that this thesis uses. Dilation model is based on VGG-16, but the last two pooling layers are removed, and they are replaced by multiple "dilated" convolutions, a custom convolution type. The filters that are used for the convolution are 3x3 and 1x1. There are not formal results for the accuracy that this model can achieve by using the CityScape dataset, but benchmarks results [49] mention that Dilation can outperform fcn8 in PASCAL VOC dataset. The number of multiply-accumulate operations that this model requires is 2.65T, an extremely big number compared to all previous models, while over 99% of these operations take place on convolution. The total number of parameters, that this model has, is 134.46M, while the size of the pre-trained

model is 512.6 MB.

4.2 Technical Description

The models that are used for the performance landscape in the following chapter, were implemented in either Caffe or Tensorflow frameworks. ONNX was not used due to the fact that all of the aforementioned models were available in repository hostings like GitHub, so it was not needed any conversion from an unsupported framework to ONNX. Each tool that was used for the inference acceleration was implemented in two different platforms, except Edge AI from Xilinx, which was implemented only on ZCU102. All the platforms that were used for the performance analysis are listed on table 4.1, and as well as which toolkit was implemented in each platform and which processor is targeted.

System	Processor Unit	Toolkit	Target
PC	CPU	OpenVino	Intel I7-6700
PC	VPU	OpenVino	Neural Compute Stick 2
TegraTX2	GPU	TensorRT	Nvidia Pascal GPU
PC	GPU	TensorRT	GTX1060 6GB
System On Chip	CPU	ARMNN SDK	ARM Cortex A-53
System On Chip	CPU	ARMNN SDK	ARM Cortex A-57
ZCU102	DPU	DNNDK	Xilinx DPU

Table 4.1: List of Processor Units

The benchmarks were resulted by the execution of the neural network models that were described in the previous sub-section. However, the execution of some pre-trained models was not possible, due to the fact that some layers were not supported by the tools. The models that were excluded for each toolkit and the reason that this happened is described in the following list;

- A core element of the models that created for the Image Segmentation task is the utilization of Deconvolution layer, in order to restore the image in its

original dimensions for depiction. Unfortunately, this type of layer is not supported by Arm's toolkit yet, so both of the models that are used for Image Segmentation were excluded for this task.

- Another layer that it is not supported by both Arm NN SDK and Xilinx DNNDK is Normalization. Models that are trained by using this type of layer, like SSD300 and SSD500 are also excluded by the corresponding platforms. Furthermore, DNNC does not support the Crop layer, which is implemented in the FCN8 model in Image Segmentation.

- Yolov3 is a special model that originally was created by using DarkNet framework for its training. Although the conversion from this framework to Caffe or Tensorflow is feasible, some layers are not supported by the parsers of the toolkits. Thus, the utilization of custom plugins is mandatory in order to use this model for the Object Detection task. The toolkits from Nvidia and Intel provide the ability to create custom plugins, on the other hand the other two do not provide this flexibility.

- As it was described in sub-section 3.4.2 , the quantization of a model in INT8 precision is an essential procedure in order to use ZCU102 for inference. However, Decent tool does not support the structure of every model. In addition, the conversion of input data from the initial range [0,255] into the range [0,1] that is needed for DenseNet models and Googlenet-v4 is a questionable procedure for Xilinx.

- Last but not least, the quantization process requires a lot of resources for models like Dilation, so the conversion of this models in INT8 was not possible at the moment.

One of the main features of each platform is the precision of the models that is required in order to execute the networks successfully. CPUs from Intel and

Arm use models in FP32 precision, while the Intel's Neural Compute Stick 2 requires models in FP16 precision. The graphic processor unit from Nvidia can support both precisions, but as the company advises the use of FP16 precision will negatively affect the performance of inference execution, so this case is avoided. On the other hand, Tegra TX2 fully supports both precisions, so it was possible to accelerate the performance on both occasions. Finally, it was already mentioned that the execution of a neural network in Zynq Ultrascale+ evaluation boards requires the conversion of the model into INT8 precision, due to the special DPU that designed.

All images that were used for the inference, were selected randomly from the corresponding datasets that were used to train the models. Also, all images were already resized in the appropriate dimensions that each model requires before they are used as inputs. Another option is to implement OpenCV functions into the application that provide easy modifications in both the image dimensions and the range of input pixels from the image, like the ranges [0,255] and [0,1] that are used by the aforementioned models.

By running each model the following parameters are exported; the total time that the application needs for inference execution and as well as the number of images that the platform can process each second by implementing the corresponding toolkit. The inference time was exported by providing in the inference application different batch sizes of images each time. So, the behaviour of the processor unit could be described for multiple data processing. Furthermore, the throughput is calculated for each model in order to show how many images can be inferred each second for each platform. This parameter can be exported by converting inference time into throughput. This metric can show in which batch size each platform can process the most images per second, and as well as which is the upper limit in performance of processor units.

Chapter 5

Performance Analysis

5.1 Multi-thread in DPU

Before i present the Benchmark Results, it is necessary to show an interesting feature that Xilinx provides for their Deep-Learning Processor Unit. During the inference stage, a developer can set the number of threads that can work in parallel to further accelerate the execution of the task.

Multithreading in widely used in order to provide concurrent execution of multiple tasks. This can happen either by providing multiple processes or by splitting the data of a process in order to execute them concurrently in batches rather than in a sequential order. The latter approach can be used in the inference application for the DPU, where the developer can set the number of images that can be processed concurrently in order to reduce the inference time, by creating more threads.

In figure 5.1, i provide the speed-up factor that can be achieved by using multithreading. In the specific example, i illustrate by using as a basis for comparison the single thread, the acceleration that can be achieved by creating two, four and

eight threads. So, the optimal number of threads can be found where the overall best performance can be achieved. The speed-up factor for each number of thread is calculated as the average speed-ups obtained by all pre-trained models. The inference times for each pre-trained model, which were used to export the speed-up factors for the aforementioned total numbers of threads, are shown in the tables A.1, A.2, A.3, A.4 in Appendix.

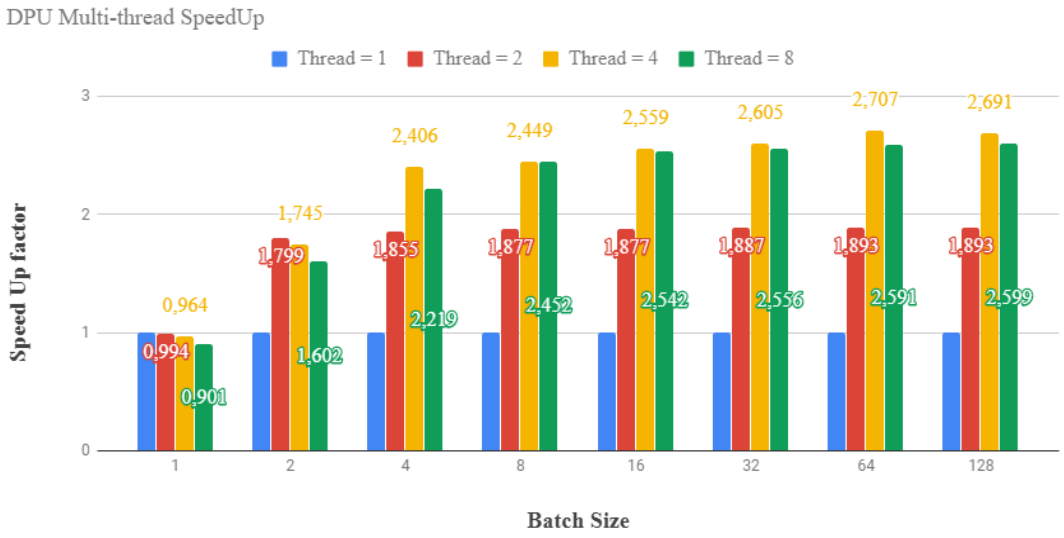

Figure 5.1: Speed-up of Multithreading in DPU

As it seems only when the app has only one image to process, it is better a single thread to be implemented. This happens because the overhead for creating more threads slows down the execution. In all other occasions multithreading is recommended to accelerate the process. By extending the previous approach, when the total number of threads coincides with the total number of images, then the optimal speed up can be achieved. Overall though, in almost every case the best speed-up can be achieved by creating four threads, as it seems in the aforementioned figure. The creation of eight threads or more does not accelerate further the execution of the task, unless when the total images are eight where the

speed-up factor is slightly better. Beyond this situation, it worsens slightly the execution time of the inference task compared to the utilization of four threads in all other batch sizes.

For the performance analysis in the next section, the execution times which will be presented along with the other processor units in their respective platforms, come from the implementation of four threads within the inference application deployed in ZCU102.

5.2 Benchmarks Results

5.2.1 Inference Time and Throughput for pre-trained models

As it was described in section 4.2, inference time was measured by running multiple times the deep learning inference applications, which were created for each platform by using the corresponding toolkit to enable and accelerate the performance of the process. In this section i will discuss about the performance in Image Classification task for three different occasions; when batch size equals to 1 in table 5.1, to 16 in table 5.2, and to 128 in table 5.3. Other instances were run for different batch sizes and the results are imprinted in Appendix A. Furthermore, the performance of Object Detection task is shown in table 5.4, while the table 5.5 shows the results for Image Segmentation.

All the aforementioned tables present the time that a platform with its processor unit needs to perform the three tasks by using a variety of pre-trained models. In 4.1, i explain some very essential features for each model that was imported into the tasks. Two key characteristics are the number of the parameters of a

InferenceTime(ms)	I7 6700	Neural Stick 2	GTX1060 6GB	TX2(FP32)	TX2(FP16)	Cortex A57	Cortex A53	ZCU102
Alexnet	18.315	24.899	2.777	11.427	7.402	141	195	-
InceptionV1	16.107	23.646	3.706	9.833	5.866	213.5	375	11
InceptionV4	94.694	141.978	22.335	83.924	41.413	1170	2078.5	-
SqueezeNet1.1	3.125	10.207	2.146	3.375	2.323	77.5	106	-
DenseNet121	26.998	50.197	13.372	29.736	20.207	339	667	-
DenseNet161	66.532	139.383	22.579	66.029	45.458	773	1455.5	-
DenseNet169	32.083	64.803	17.65	36.927	26.653	425	821	-
ResNet50	33.81	56.976	5.772	21.506	11.955	807.5	1008.5	21.5
ResNet101	63.186	102.302	9.619	38.2	20.919	1441	1890.5	35
ResNet152	92.968	152.952	13.982	55.153	30.18	2028.5	2662.5	48
VGG16	142.578	177.892	10.828	65.881	38.026	1029.5	1595.5	54.5
VGG19	145.241	215.961	12.704	79.324	45.655	1344	2026.5	62.5

Table 5.1: Inference Time(ms) with Batch Size = 1

InferenceTime(ms)	I7 6700	Neural Stick 2	GTX1060 6GB	TX2(FP32)	TX2(FP16)	Cortex A57	Cortex A53	ZCU102
Alexnet	126.438	356.977	8.92	61.263	38.876	808.5	1577	-
InceptionV1	169.422	335.938	19.62	122.849	68.715	-	-	53
InceptionV4	1091.498	2214.96	147.014	1007.323	492.6	-	-	-
SqueezeNet1.1	51.428	129.56	6.71	39.562	23.774	-	-	-
DenseNet121	573.375	761.404	67.166	391.211	265.155	5301.5	11197.5	-
DenseNet161	1279.151	2180.98	156.526	940.54	638.501	11605	23419.5	-
DenseNet169	646.134	995.164	85.45	474.205	355.512	6787	13770.5	-
ResNet50	326.91	869.416	42.071	275.136	145.897	4913	9341	113.5
ResNet101	655.317	1594.85	76.581	502.514	265.505	8686.5	17337.5	193.5
ResNet152	946.292	2389.41	104.043	735.167	385.735	12574	24980	270
VGG16	1323.49	2795.25	105.554	825.091	452.953	10390.5	19100	343.5
VGG19	1629.35	3403.53	134.693	1039.403	572.843	12410	21620	389

Table 5.2: Inference Time(ms) with Batch Size = 16

InferenceTime(ms)	I7 6700	Neural Stick 2	GTX1060 6GB	TX2(FP32)	TX2(FP16)	Cortex A57	Cortex A53	ZCU102
Alexnet	589.305	2842.91	53.475	411.777	239.832	6096.5	12329	-
InceptionV1	1248.01	2670.26	132.038	954.438	532.395	-	-	412.5
InceptionV4	8358.763	17678.8	953.909	7930.395	3877.763	-	-	-
SqueezeNet1.1	408.659	1020.75	40.178	310.393	183.963	-	-	-
DenseNet121	4602.264	6071.11	436.177	2923.055	2110.083	47842.5	96441	-
DenseNet161	10086.689	17420.9	1039.479	7448.316	5092.563	101125.5	OutOfMemory	-
DenseNet169	5338.663	7935.63	555.271	3755.263	2656.821	59102.5	118648.5	-
ResNet50	2407.34	6941.904	271.85	2120.495	1132.359	39122	75424.5	853.5
ResNet101	4823.6	12753.5	478.864	3910.52	2072.454	67986	136776.5	1484
ResNet152	7321.046	19085.8	744.937	5724.997	3032.654	97528.5	OutOfMemory	2087
VGG16	9760.43	22329.2	798.636	6912.875	3784.857	OutOfMemory	OutOfMemory	2553.5
VGG19	12626.8	27192.9	982.612	8551.72	4727.338	OutOfMemory	OutOfMemory	2814.5

Table 5.3: Inference Time(ms) with Batch Size = 128

pre-trained model and the total number of multiply-accumulate operations that a processor unit must perform for a successful execution of the process. The larger these numbers are for a pre-trained model, the more time a processor unit will need to perform its task. This claim can be confirmed by the mentioned tables. In addition, of course the increase in the number of images that are imported in the application and processed during the inference stage, leads to an increase in the

execution time. In some cases, the inference time is doubled each time the number of images is doubled, while the above claim is not true for specific models like AlexNet and ResNet, and for processor units like the GPU. This behaviour can mainly be observed from the throughput tables below for each targeted processor unit, which each value of this metric indicates how many pictures are inferred by the application each second.

The results from inference time and throughput will be analysed in detail in the following section, where they will be used to indicate how well a processor unit can perform the specific tasks. Generally though, the number of images that can be processed each second increases when batch size equals to 16 or 128, compared to applications that have batch size equals to 1. Some platforms have an upper limit on how many images can be processed each second, but this limit depends on the pre-trained model that is used. Meanwhile, some models with many parameters and many MAC operations, such as VGG16 and VGG19, have a slight improvement.

Same behaviour is also observed in the other two tasks that were developed. Especially in Image Segmentation, where occasions with batch sizes greater than one are run, the improvement in throughput is minimal and inference time is roughly doubled, every time the number of images is doubled in each test.

InferenceTime(ms)	I7 6700	Neural Stick 2	GTX1060 6GB	TX2(FP32)	TX2(FP16)
SSD300	184.396	610.547	21.119	118.17	117.644
SSD512	559.325	1383.74	43.394	279.074	278.846
YOLOV3	204.253	596.23	36.854	277	266.603

Table 5.4: Inference Time(ms) for Object Detection

Batch Size = 1	I7 6700	GTX1060 6GB	TX2(FP32)	TX2(FP16)
FCN8	1117.67	91.292	609.016	633.601
Dilation	14377.2	1289.647	10738.42	10534.961
Batch Size = 2				
FCN8	2284.22	178.436	1402.453	1314.215
Dilation	28421.8	2605.868	20887.409	20660.691
Batch Size = 4				
FCN8	4093.35	349.991	2855.754	2832.96
Dilation	58071.5	OutOfMemory	OutOfMemory	OutOfMemory
Batch Size = 8				
FCN8	7791.61	660.071	5584.956	5791.882
Dilation	206525	OutOfMemory	OutOfMemory	OutOfMemory

Table 5.5: Inference Time(ms) for Image Segmentation

5.2.2 Platforms' Results and Discussion

- **Intel**

Even though I7-6700 is a general purpose processor unit, it delivers high performance in execution of deep learning inference applications. The inference times, that it can achieve when batch size equals to 1, are very low and prove that OpenVino can exploit every asset of a CPU. As batch size is increasing, the CPU has a distributed behaviour, due to the pre-trained models. Models like AlexNet and InceptionV1 can be accelerated significantly as batch size is increasing. On the other hand for DenseNet models and Squeezenet the best performance can be achieved when the inference task has to process one image at a time. For all other models, a significant improvement can be achieved when batch size equals to 16, but beyond this value there is not a big progress in performance.

Furthermore, it turned out that Image Segmentation is a very challenging task, as in both models the processor unit can't process more than one image

each second. But both Dilation and FCN8 are based on VGG16, and thus an improvement is observed by increasing the batch size. However, when batch size is equal to or greater than 4, a throughput greater than 1 is achieved in FCN8. This sample shows that an increased batch size can often accelerate the procedure in a CPU. Finally, an object detection task can be performed with the SSD300, SSD512 and YoloV3 pre-trained models in just a few hundred milliseconds.

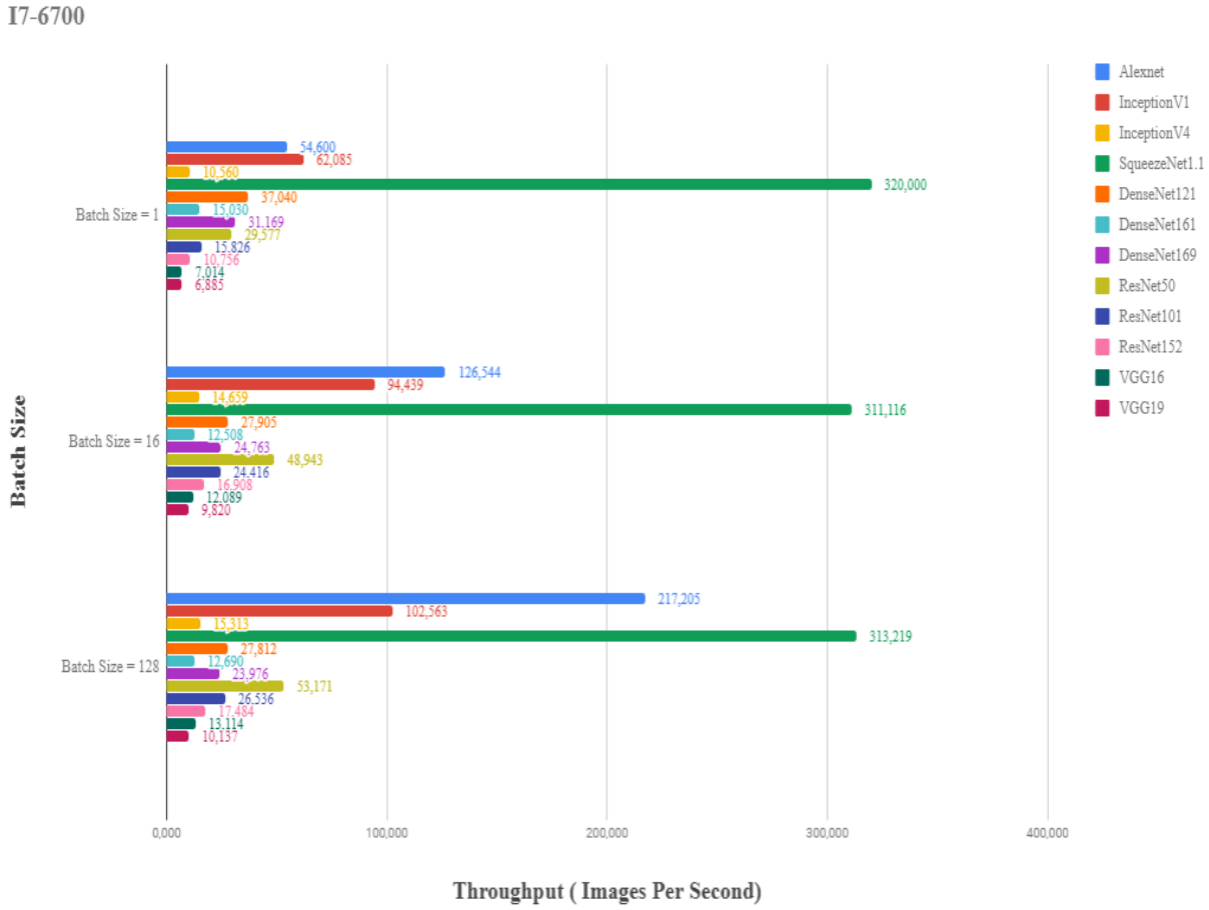

Figure 5.2: Throughput for I7-6700

Intel Movidius Neural Compute Stick 2 delivers high performance in deep learning inference applications by minimizing the power consumption, as it just

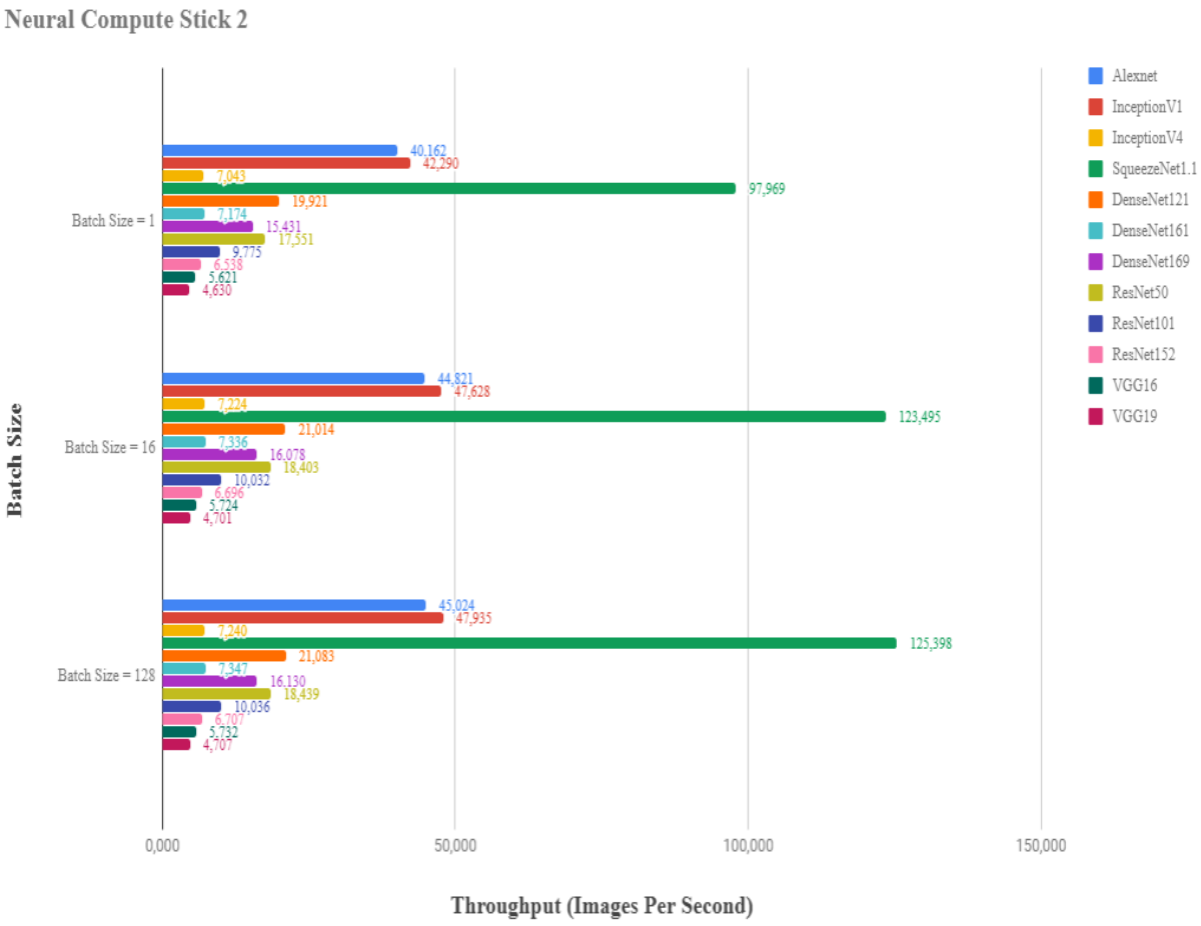

Figure 5.3: Throughput for Neural Compute Stick 2

plugs in a USB3 port of any system with Windows, Ubuntu and CentOS. A big advantage of the Visual Processor Unit, that is contained by the embedded system, is that supports pre-trained models in FP16 precision contrary to Intel's CPU. The NCS2 can't operate any tasks without OpenVino, so the utilization of the toolkit is necessary.

When batch size is equal to one, NCS2 achieves very similar results compared to I7-6700, as it is only few milliseconds slower that the CPU. However, by using batches of 16 or 128 images, the execution does not significantly accelerate fur-

ther. This can be seen from the throughput graph for NCS2. Only pre-trained models with few MAC operations and parameters can be accelerated, and the improvement is limited to a few more images per second. That means, that the inference task for NCS2 is almost optimal from the beginning. The inference time for the image classification task for the AlexNet model when batch size equals to one is 24,899 ms and the respective time when batch size equals to 16 is 356,977 ms. Other example is ResNet50 where the corresponding metrics are 56,976 ms and 869,416. Both of these examples have as common that the inference time, when batch size equals to 16, is approximately sixteen times greater than when batch size equals one. This example enhances the approach that the inference execution is almost optimal from processing just one image at a time.

A disadvantage of NCS2 is that it could not run the Image Segmentation task, as both of the pre-trained models could not run in the embedded system. In addition, NCS2 cannot perform that well the object detection task, as it happened with Image Classification when batch size equals to 1. This can be seen from table 5.4, where NCS2 needs almost three times more time in order to perform successfully a object detection task for each model.

- **Nvidia**

Graphic processor units can deliver high throughput when they have to process multiple data at the same time. Thus, an application that is based on processing multiple data can be accelerated to a great extend by using the specific processor unit. This claim can be seen in the throughput graph for the GTX1060 6GB, where regardless of the pre-trained model the GPU can process more images per second as the number of batch size increases. In tables where inference times

are presented, in the initial test where batch size equals to 1, graphic processor unit can perform image classification task for each pre-trained model in just a few milliseconds. But as the batch size increases, the gap, between this GPU and the other SoC, that are presented in this analysis, keeps increasing and the GPU can accelerate many times the execution of an inference application compared to the other processor units. However, the side-effect of this acceleration is the high power consumption that a GPU has.

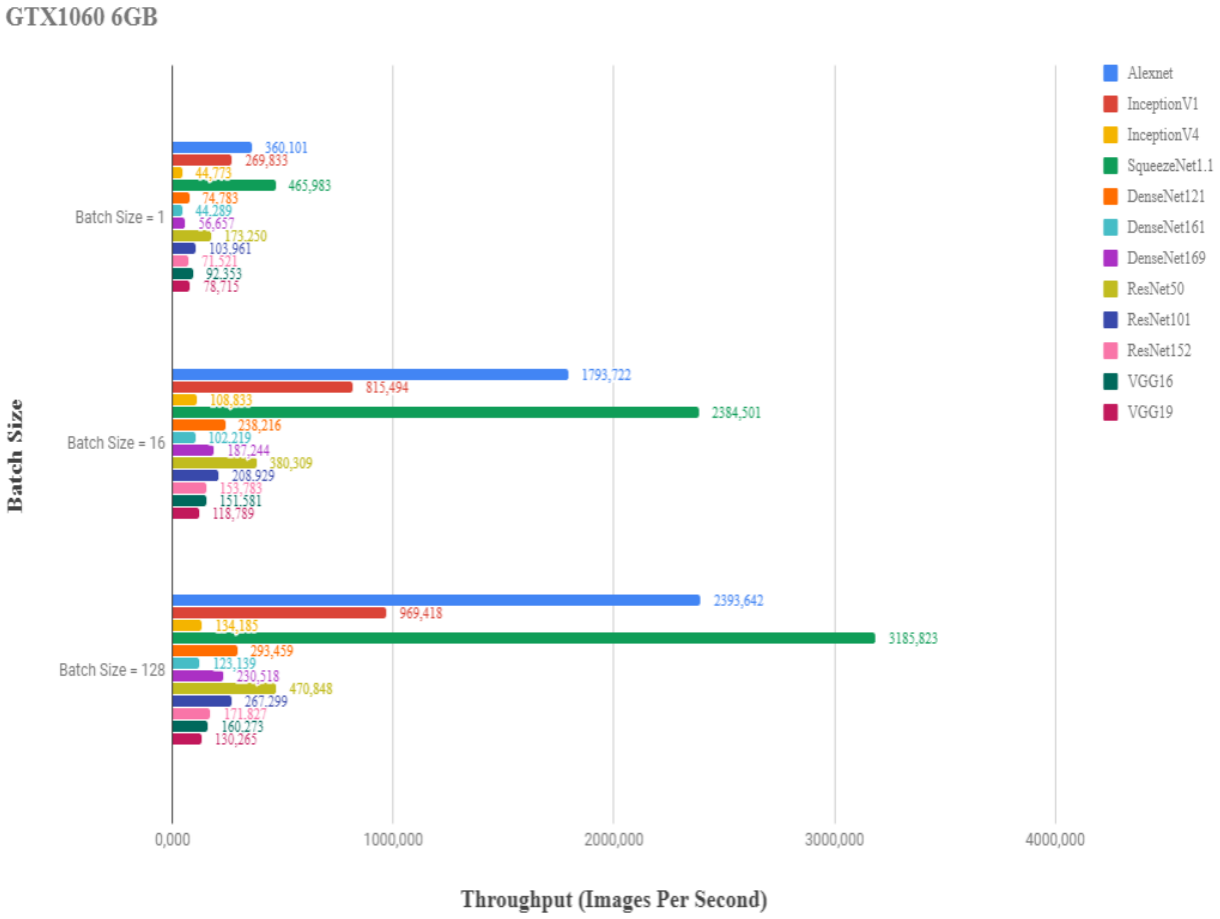

Figure 5.4: Throughput for GTX1060 6GB

On the other hand, the cost of a such high throughput is the big memory con-

sumption for tasks like Image Segmentation. By running the Image Segmentation task for the Dilation model, the process was killed due to the system did not provide enough memory for the execution. This has to do with the Video RAM that a graphic processor unit contains, and not with the simple, RAM which a system provides. Also, the aforementioned model along with FCN8 cannot be further accelerated by inferring more images at the same, as the throughput does not increase. However, object detection task is greatly accelerated compared with the other processor units, as it can perform any pre-trained model in few dozens milliseconds.

Contrary to the GPU, Tegra TX2 can perform deep learning inference application with pre-trained model in both floating point precisions, FP32 and FP16. So, by using FP16 pre-trained models, applications that are developed using TensorRT can have way better performance without losing the quality of the exported data compared to FP32. Tegra TX2 does not have a Video RAM, due to the fact it possesses an integrated graphic card and not a dedicated. However, it provides 8GB RAM that the GPU shares with CPU. So, it is natural that Tegra TX2 cannot provide the same throughput as a GPU can. Furthermore, it has a significantly lower power consumption, as it is recognized as the most power efficient embedded AI computing device.

All the features that were described in the above paragraph can be confirmed from the given tables for inference time and throughput. Processing one image at a time is slower than a GPU as it was expected, but faster than Intel's CPU due to the use of the integrated GPU. Throughput can be increased by providing more data for simultaneous processing, but this acceleration depends on the pre-trained model. For models like AlexNet or SqueezeNet, which have few MAC operations and parameters, throughput can be greatly accelerated. On the other hand, for models like VGG16 and DensenNet161 throughput does not increase

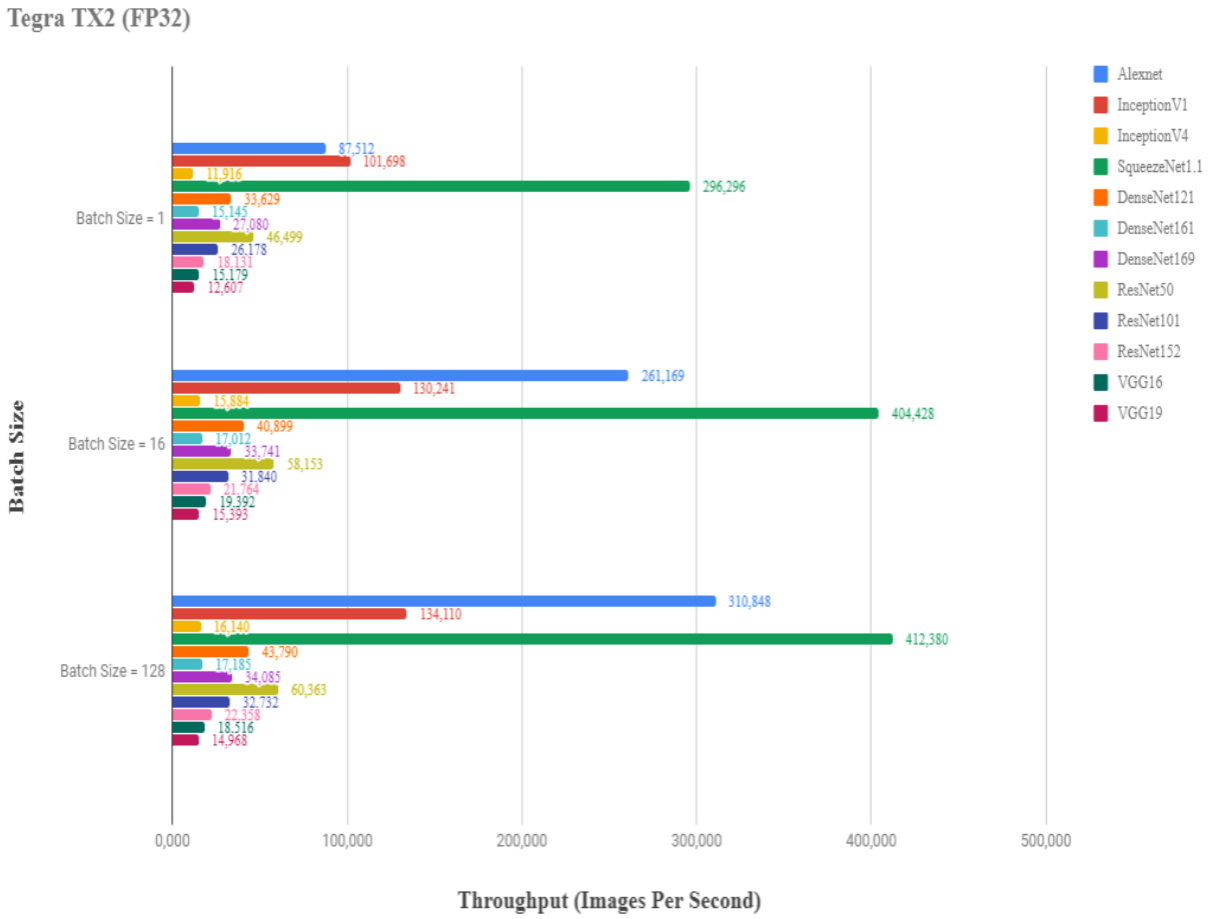

Figure 5.5: Throughput for FP32 in TegraTX2

significantly. Furthermore, the performance of the pre-trained models, that were used for the object detection task, is few times slower for the corresponding models in GPU. Similar behaviour is imprinted in the models that were used for Image Segmentation. Also, by increasing the batch size, the performance does not improve, because as it seems from the table 5.5 as images are doubled, the inference time is also doubled.

In Tegra TX2, i can show how important it is to use pre-trained models in FP16 precision compared to FP32 precision. By using FP16 precision, accuracy

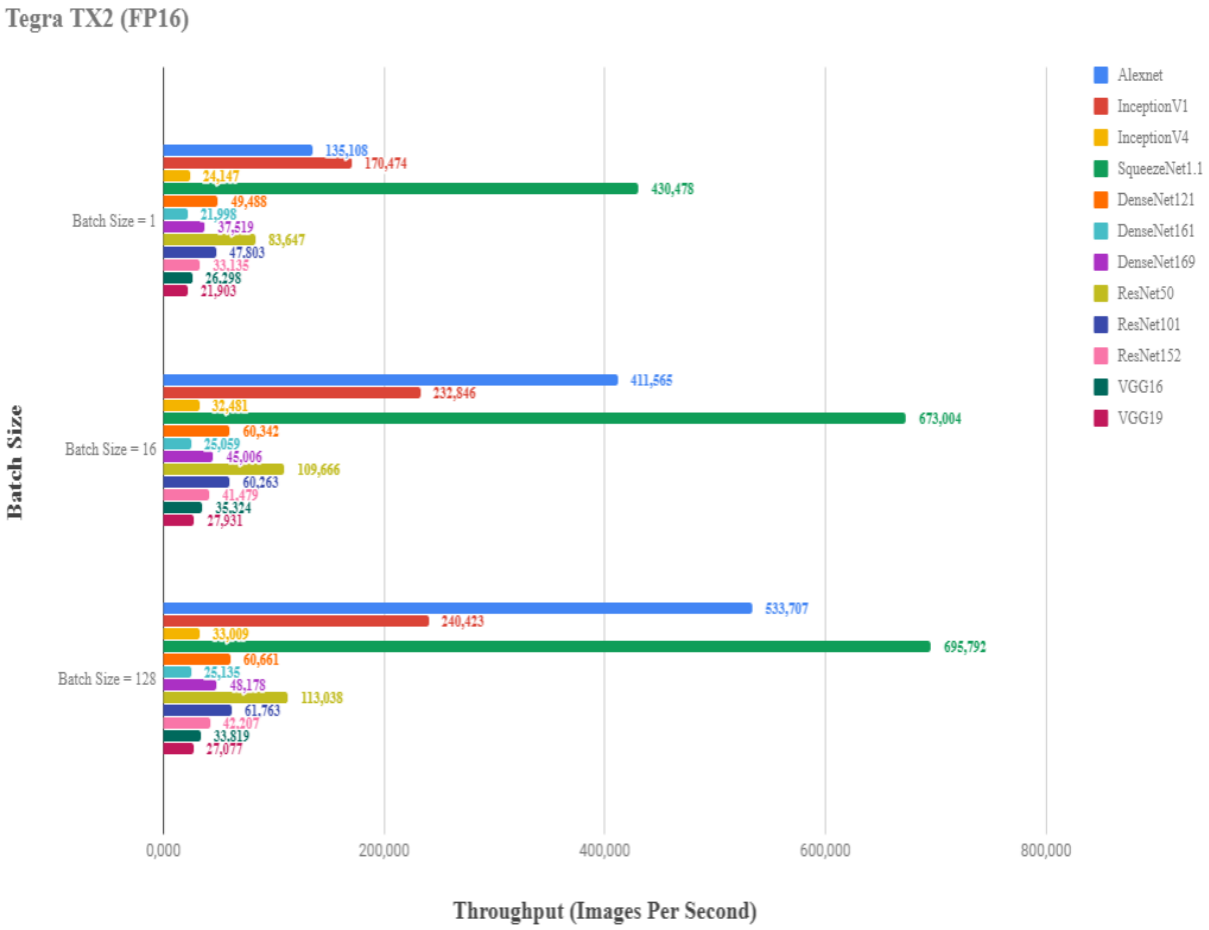

Figure 5.6: Throughput for FP16 in TegraTX2

may be slightly reduced, but the advantages to memory consumption and execution time are huge. In tables that inference times are imprinted, it is proved that the use of a FP16 pre-trained model can cut almost in half the inference time, which is a tremendous improvement. However, this approach is working only for pre-trained models that are used for a image classification task. In tasks like object detection, table 5.4, and image segmentation, table 5.5, where the pre-trained models have more complex structures, the utilization of FP16 or FP32 precision does not play any significant role, as the inference times of the corresponding

procedures converge.

- **Arm**

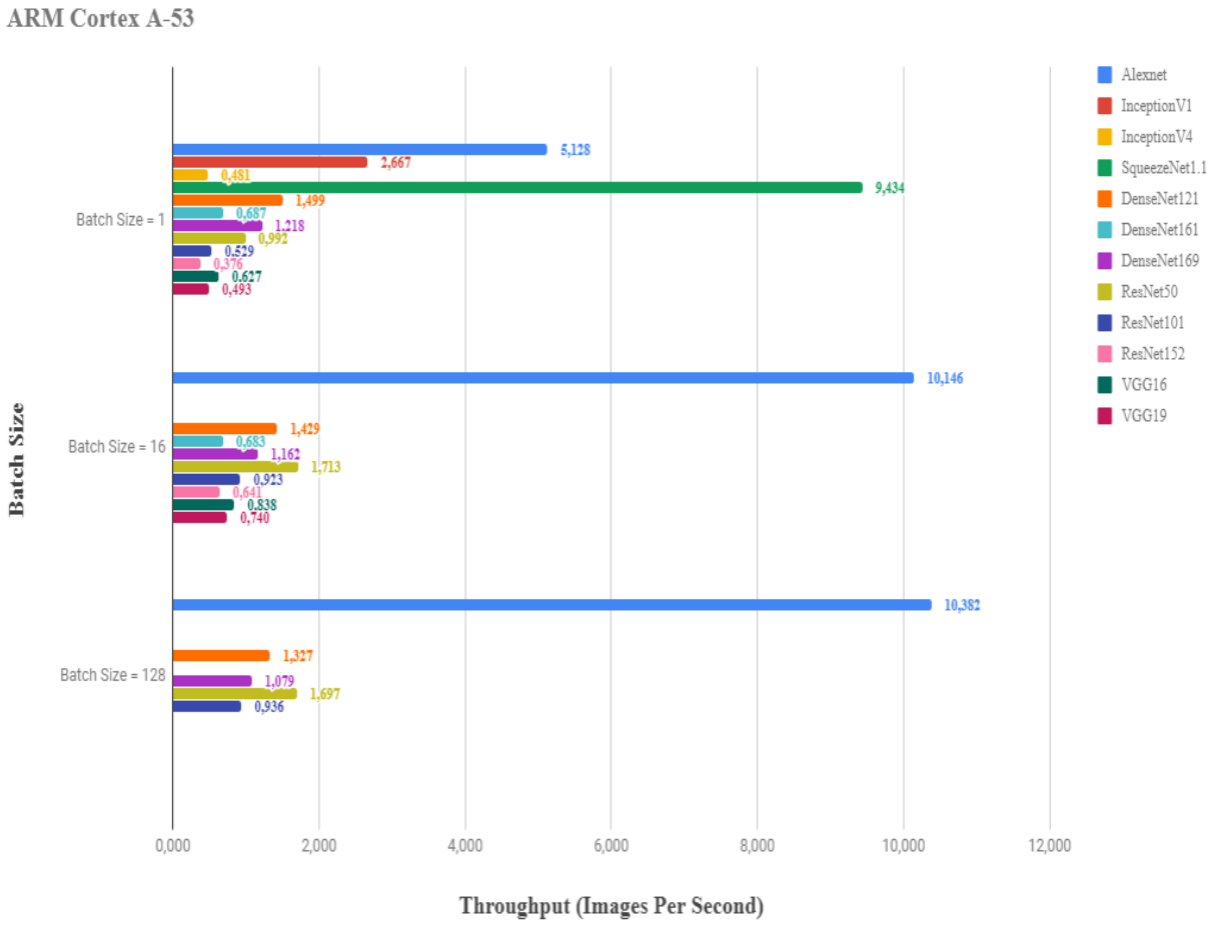

Figure 5.7: Throughput for Cortex A-53

With a reduced instruction set, compared to the x86 architecture that Intel use for their CPUs, the deployment of a deep learning inference application on Arm's CPUs is quite challenging. Arm NN SDK enables this feature and achieves

a performance that is imprinted by the inference time tables. The performance of both Cortex A-53 and Cortex A-57 is poor compared to other processor units, but both of these processors are mainly used for devices that low power consumption is essential. Thus, the inference times that can be achieved by NN SDK is reasonable to be high. Furthermore, by increasing the batch size from 16 to 128 in both processors, the application needs four times more time to perform its task, presenting a linear behaviour after a specific batch size.

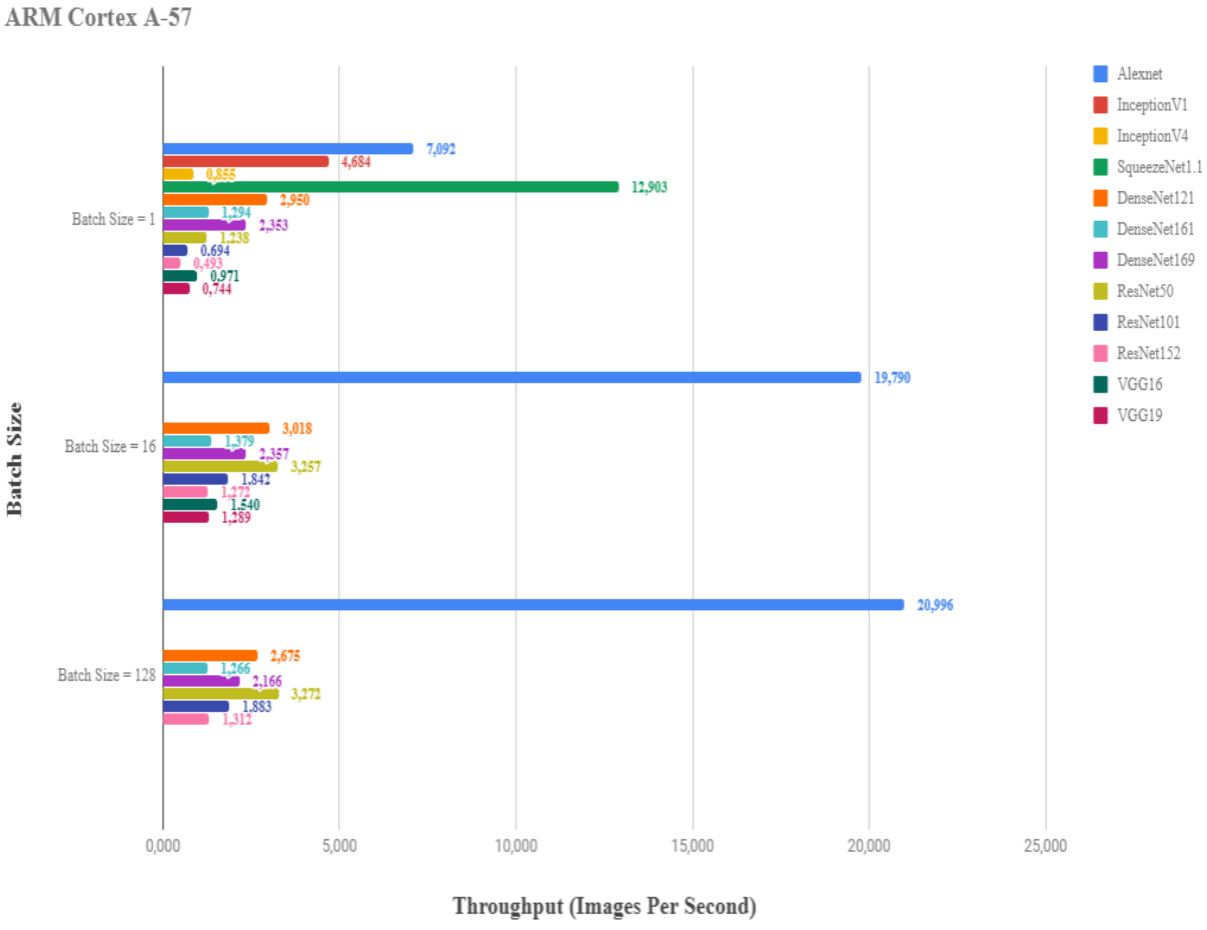

Figure 5.8: Throughput for Cortex A-57

The above approach can be also confirmed by the graphs that imprint through-

put, where the increase in batch size does not improve the mentioned metric. On the other hand, throughput is significantly improved when using batch sizes greater than one. This improvement for the ARM Cortex A-57 can be almost up to three times for models like AlexNet and ResNet, and two times for VGG pre-trained models. For Cortex-A53, these factors are slightly reduced compared to the previous CPU. This happens because Cortex-A53 was designed in order to provide maximum power efficiency, while Cortex-A57 was designed to provide maximum compute performance. Often they are used as a pair in a big.LITTLE configuration. That explains how the latter model can perform a deep learning inference application twice as good as Cortex-A53 when then batch size is equal to or greater than 16.

- **Xilinx**

Even though a limited number of the available pre-trained models could successfully run by using DNNDK on ZCU102, an analysis can be provided for the performance. A major difference between this toolkit and the others is the conversion of the pre-trained model by using a quantization procedure from floating point into INT8. As it is described by Xilinx, this process sacrifices the accuracy to a very small degree that a model provides, in order to achieve maximum performance. Thus, a direct comparison between the DPU and the other processor units cannot be made, but only in accordance with the preceding procedure.

The ability of creating manually the threads that will be executed from DPU was described in section 5.1. In this point i will refer only to an application that utilize four threads, but an easy correlation can be made for the other occasions. FPGAs can provide high throughput with high compute power similar

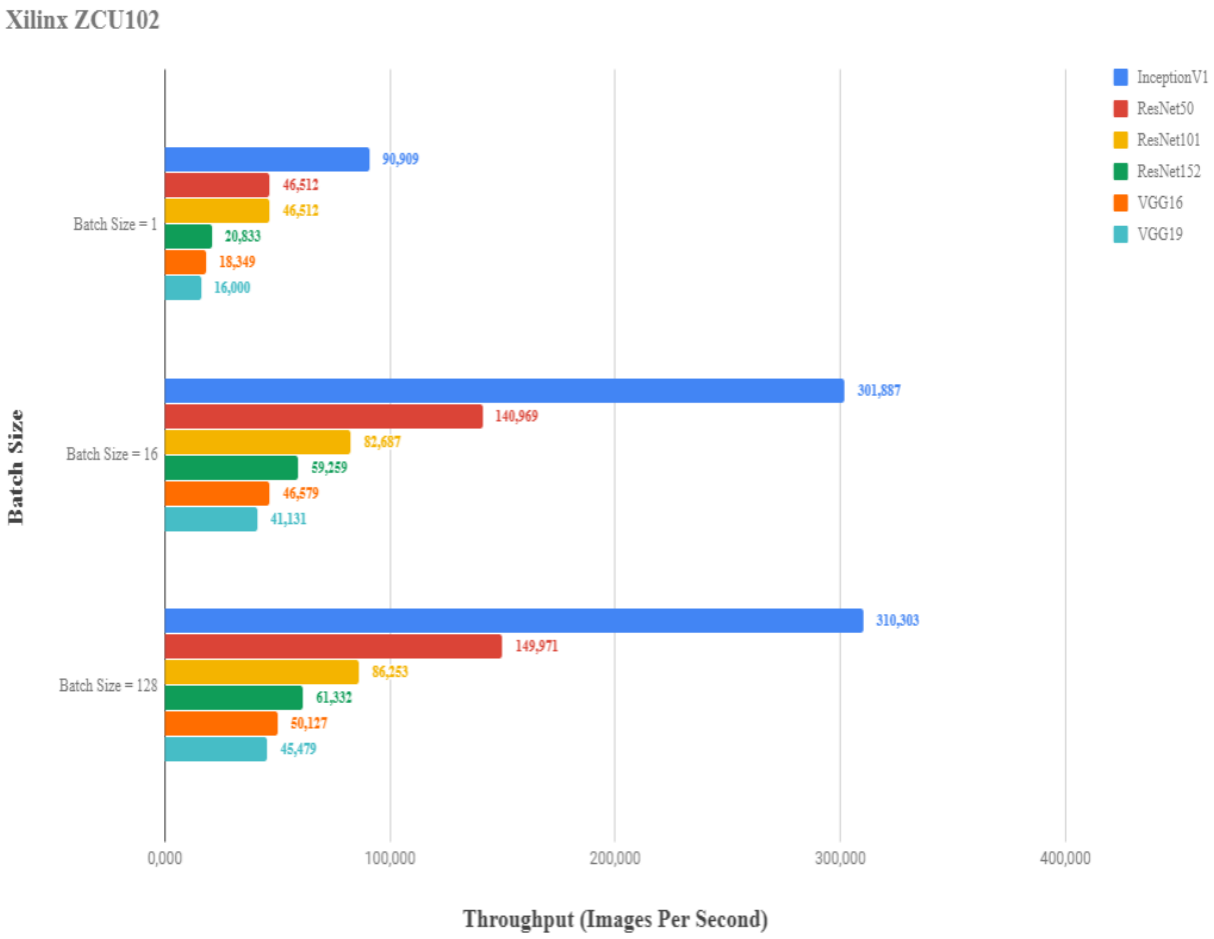

Figure 5.9: Throughput for ZCU102

to GPUs. In addition, they are way more power-efficient. The first characteristic is imprinted in throughput and inference time tables. The conversion of a model into INT8 precision and the execution of it in DPU can achieve a low inference time of dozens milliseconds. A significant improvement is observed when the application processes batches of 16 images, and this is imprinted better by making a comparison between the throughput in graph 5.9. By increasing the batch size from 1 to 16, throughput scales by a factor slightly over than 3. However, by further increasing the batch size, throughput does not improve at all, and the

inference time has a more linear behaviour.

Chapter 6

Conclusions

This thesis discusses the latest achievements in acceleration of inference applications that use CNN models, which are developed for resource constrained platforms. First, the background of CNNs and the means for hardware accelerated data processing are presented. Furthermore, the acceleration toolkits are presented by the most well-known hardware vendors; Intel, Nvidia, Arm and Xilinx, and how these can be integrated within an application. In addition, a description of the supported frameworks is given along with the most preferred pre-trained models. Last but not least, real-world results are obtained by running a variety of pre-trained models for multiple tasks. These results are used for the performance landscape of both the given toolkits, as well as the different processor units and platforms that integrate the aforementioned tools. In order to accelerate the performance of inference applications, the exploitation of the capabilities that a processor unit offers through the toolkits is not sufficient. The pre-trained models have to go through an optimization process before their utilization by the applications. The optimizers check the validity of the pre-trained models and perform configurations in order to convert them into an optimized form for the toolkits and processor units.

Appendix A

Inference Results

InferenceTime(ms)	1	2	4	8	16	32	64	128
VGG16	52	102	200	399	794.5	1583.5	3166.5	6333.5
VGG19	60	116.5	231.5	459	916	1829	3651	7306.6
ResNet50	21	39	76.5	148	295	588	1175	2347.5
ResNet101	33.5	65	127.5	252.5	503.5	1005.5	2007	4014
ResNet152	47	90.5	180	356	713.5	1416	2836	5669
InceptionV1	10.5	20	39.5	78.5	150.5	300.5	601	1194

Table A.1: Inference Time with Num of Threads = 1

InferenceTime(ms)	1	2	4	8	16	32	64	128
VGG16	53	58,5	114	223	441,5	879	1749,5	3493.5
VGG19	61	67	128.5	254	502	1000.5	2001.5	3999
ResNet50	20.5	21.5	41	78.5	155.5	308	613.5	1224
ResNet101	34	35.5	68	132.5	266	523	1044	2087
ResNet152	47.5	49	94.5	186	370	737.5	1467	2936.5
InceptionV1	10.5	11	20.5	40	78.5	155	308	613

Table A.2: Inference Time with Num of Threads = 2

InferenceTime(ms)	1	2	4	8	16	32	64	128
VGG16	54.5	60.5	99	187.5	343.5	645	1257.5	2553.5
VGG19	62.5	68.5	107.5	206	389	737.5	1398	2814.5
ResNet50	21.5	23	31	58	113.5	217	424.5	853.5
ResNet101	35	37	51.5	99.5	193.5	388	740.5	1484
ResNet152	48	50	72	137.5	270	542	1060	2087
InceptionV1	11	11	14	29.5	53	108	203	412.5

Table A.3: Inference Time with Num of Threads = 4

InferenceTime(ms)	1	2	4	8	16	32	64	128
VGG16	59	64	99.5	178	350	671.5	1295.5	2576
VGG19	66	70	115.5	199.5	388.5	757	1464	2865
ResNet50	24.5	26	35	60.5	116.5	224	445.5	906.5
ResNet101	37	40	54.5	101	195	387.5	766	1530
ResNet152	50	55.5	75.5	141.5	272.5	537.5	1086.5	2158
InceptionV1	11.5	12.5	16.5	29	52	111	219.5	434.5

Table A.4: Inference Time with Num of Threads = 8

InferenceTime(ms)	I7 6700	Neural Stick 2	GTX1060 6GB	TX2(FP32)	TX2(FP16)	Cortex A57	Cortex A53	ZCU102
Alexnet	46.352	46.804	3.323	14.432	9.565	190	364.5	-
InceptionV1	23.761	44.366	5.011	17.662	10.37	-	-	11
InceptionV4	166.256	279.653	27.727	137.994	68.197	-	-	-
SqueezeNet1.1	6.325	18.391	2.35	5.962	3.839	-	-	-
DenseNet121	61.555	97.694	16.541	54.013	37.027	673.5	1293	-
DenseNet161	144.509	274.934	31.829	127.752	86.521	1460.1	2794.5	-
DenseNet169	66.26	126.989	22.045	67.726	48.527	808	1606.5	-
ResNet50	57.74	111.586	8.187	37.71	20.64	1073.5	1564	23
ResNet101	109.155	202.011	14.615	70.619	38.287	1923	2868.5	37
ResNet152	162.05	301.716	22.411	103.598	55.718	2700.5	4036.5	50
VGG16	273.431	353.048	18.056	129.46	65.357	1420.5	2599	60.5
VGG19	313.24	428.488	21.445	154.974	80.502	1743.5	3085.5	68.5

Table A.5: Inference Time with Batch Size = 2

InferenceTime(ms)	I7 6700	Neural Stick 2	GTX1060 6GB	TX2(FP32)	TX2(FP16)	Cortex A57	Cortex A53	ZCU102
Alexnet	55.679	90.916	3.9	20.689	12.834	270.5	529.5	-
InceptionV1	47.154	86.424	7.481	32.97	18.769	-	-	14
InceptionV4	293.518	555.558	44.703	262.625	130.128	-	-	-
SqueezeNet1.1	11.956	34.47	3.267	10.654	6.76	-	-	-
DenseNet121	124.794	192.632	22.794	101.401	69.722	1298.5	2603.5	-
DenseNet161	292.713	547.973	50.422	245.371	163.897	2857	5607	-
DenseNet169	148.282	250.689	31.417	124.107	89.56	1651.5	3233	-
ResNet50	115.996	219.59	12.794	73.07	39.022	1621	2672.5	31
ResNet101	195.894	401.19	22.966	134.953	71.589	2862	4921.5	51.5
ResNet152	281.361	600.415	33.443	196.542	104.316	4078	7012.5	72
VGG16	431.44	702.173	31.425	228.165	119.667	2592.5	4756	99
VGG19	524.828	852.969	36.239	281.031	148.356	3532	5558	107.5

Table A.6: Inference Time with Batch Size = 4

InferenceTime(ms)	I7 6700	Neural Stick 2	GTX1060 6GB	TX2(FP32)	TX2(FP16)	Cortex A57	Cortex A53	ZCU102
Alexnet	83.105	180.525	5.988	36.995	23.04	441	878	-
InceptionV1	84.212	170.116	11.553	62.855	35.704	-	-	29.5
InceptionV4	574.724	1108.93	80.427	513.245	251.397	-	-	-
SqueezeNet1.1	25.581	66.606	4.125	20.323	12.435	-	-	-
DenseNet121	271.3	381.785	36.02	196.681	134.998	2659	5441	-
DenseNet161	610.545	1090.98	85.455	475.673	320.366	5712	11501	-
DenseNet169	301.174	498.589	47.568	239.123	171.494	3327	6715.5	-
ResNet50	176.869	436.923	22.988	142.589	76.051	2541	4762.5	58
ResNet101	342.566	799.323	41.901	260.385	137.073	4657.5	9069.5	99.5
ResNet152	639.2	1196.08	61.508	379.767	198.977	6733.5	12984	137.5
VGG16	732.122	1399.67	54.108	415.463	234.934	5076.5	9292.5	187.5
VGG19	892.211	1703.92	69.797	527.602	292.732	6586.5	10813.5	206

Table A.7: Inference Time with Batch Size = 8

InferenceTime(ms)	I7 6700	Neural Stick 2	GTX1060 6GB	TX2(FP32)	TX2(FP16)	Cortex A57	Cortex A53	ZCU102
Alexnet	192.353	710.561	15.576	108.739	64.871	1532.5	3032	-
InceptionV1	331.861	670.594	31.634	241.666	135.35	-	-	108
InceptionV4	2134.261	4423.78	273.569	1996.351	975.686	-	-	-
SqueezeNet1.1	113.185	257.705	11.652	78.245	46.337	-	-	-
DenseNet121	1173.398	1520.31	119.707	776.241	526.636	11228	22780.5	-
DenseNet161	2483.62	4357.54	289.131	1890.146	1274.374	23493	48106.5	-
DenseNet169	1338.134	1985.15	159.594	943.903	666.03	13926	28094	-
ResNet50	618.774	1737.267	77.99	539.929	287.174	9572	18469.5	217
ResNet101	1239.278	3186.88	129.903	992.275	521.747	16827	33834	388
ResNet152	1861.96	4776.46	182.558	1452.57	763.207	24272.5	49043	542
VGG16	2491.54	5584.07	202.768	1699.193	892.454	20534.5	OutOfMemory	645
VGG19	3162.66	6801.52	254.879	2114.848	1136.036	23121	OutOfMemory	737.5

Table A.8: Inference Time with Batch Size = 32

InferenceTime(ms)	I7 6700	Neural Stick 2	GTX1060 6GB	TX2(FP32)	TX2(FP16)	Cortex A57	Cortex A53	ZCU102
Alexnet	337.74	1415.87	29.226	216.866	120.905	3051.5	5846	-
InceptionV1	627.873	1336.84	69.317	480.218	267.303	-	-	203
InceptionV4	4246.603	8842.5	505.978	3972.599	1944.661	-	-	-
SqueezeNet1.1	209.764	512.399	20.78	155.592	92.173	-	-	-
DenseNet121	2312.027	3037.5	229.768	1465.088	1054.037	22805	44836	-
DenseNet161	4966.137	8709.34	545.66	3763.22	2535.478	49798	97312.5	-
DenseNet169	2657.288	3969.91	296.797	1882.349	1330.254	28478	57285	-
ResNet50	1228.037	3471.639	147.684	1070.364	567.887	19214.5	37529.5	424.5
ResNet101	2471.588	6369.17	240.279	1970.356	1039.603	33489	67944.5	740.5
ResNet152	3634.223	9548.59	366.854	2885.364	1516.289	48076.5	97690	1060
VGG16	4926.2	11166.2	394.147	3472.175	1825.857	OutOfMemory	OutOfMemory	1257.5
VGG19	6346.03	13597.5	493.119	4282.261	2302.197	OutOfMemory	OutOfMemory	1398

Table A.9: Inference Time with Batch Size = 64

References

[1] D.H. Hubel, T.N. Wiesel, "Receptive fields of single neurones in the cat's striate cortex," *The Journal of physiology*, vol. 148, pp. 574–591, 1959. 12

[2] "The first digital image." www.nist.gov/node/774341. 13

[3] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 4, pp. 193–202, 1980. 13

[4] J.Weng,N.Ahuja,TS.Huang, "Learning recognition and segmentation of 3-d objects from 2-d images," pp. 121–128, 1993. 14

[5] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. 14

[6] Y. LeCun,C. Cortes,C.J.C. Burges, "The mnist dataset of handwritten digits." yann.lecun.com/exdb/mnist/. 14

[7] KS Oh,K. Jung, "Gpu implementation of neural networks," *Pattern Recognition*, vol. 37, pp. 1311–1314, 2004. 14

[8] D. Steinkraus, P. Simard, I. Buck, "Using gpus for machine learning algorithms." Eighth International Conference on Document Analysis and Recognition (ICDAR'05), 2005. 14

[9] D. Ciresan;U. Meier,L. Gambardella,J. Schmidhuber, "Deep big simple neural nets for handwritten digit recognition." arXiv preprint arXiv:1003.0358, 2010. 14

[10] D. Ciresan,U. Meier,J. Masci,L. Gambardella,J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," pp. 1237–1242, 2011. 14

[11] A. Krizhevsky, I. Sutskever, G.E. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, 2012. 14

[12] Caffe, "caffe.berkeleyvision.org." 25

[13] Caffe. github.com/BVLC/caffe. 25

[14] G. Motroc, "Yahoo enters artificial intelligence race with caffeonspark." https://jaxenter.com/yahoo-enters-artificial-intelligence-race-with-caffeonspark-124324.html. 26

[15] "Caffe2 open source brings cross platform machine learning tools to developers." https://caffe2.ai/blog/2017/04/18/caffe2-open-source-announcement.html. 26

[16] PyTorch. github.com/pytorch/pytorch. 26

[17] Tensorflow. github.com/tensorflow/tensorflow. 26

[18] "Introducing tensorflow.js: Machine learning in javascript." https://medium.com/tensorflow/introducing-tensorflow-js-machine-learning-in-javascript-bf3eab376db. 27

[19] Tensorflow Lite. tensorflow.org/lite/guide. 27

[20] ONNX. github.com/onnx/onnx. 28

[21] OpenVino. software.intel.com/en-us/openvino-toolkit. 30

[22] PassMark Software. https://www.cpubenchmark.net/high_end_cpus.html. 31

[23] Neural Compute Stick 2. software.intel.com/en-us/neural-compute-stick. 31

[24] Nvidia TensorRT. developer.nvidia.com/tensorrt. 35

[25] P. Wojciechowski, P. Mukherjee,S. Sharma , "How to speed up deep learning inference using tensorrt." https://devblogs.nvidia.com/speed-up-inference-tensorrt/. 35

[26] Jetson TX2. developer.nvidia.com/embedded/jetson-tx2. 36

[27] ARMNN. github.com/ARM-software/armnn. 39

[28] ARMNN. developer.arm.com/ip-products/processors/machine-learning/arm-nn. 39

[29] Compute Library. developer.arm.com/ip-products/processors/machine-learning/compute-library. 39

[30] EdgeAI. xilinx.com/products/design-tools/ai-inference/edge-ai-platform.html. 42

[31] Xilinx, "ug1327-dnndk-user-guide," vol. 1.5, p. 33, 2019. 43

[32] DNNDK. https://www.xilinx.com/products/design-tools/ai-inference/edge-ai-platform.html. 43

[33] "Imagenet large scale visual recognition challenge." image-net.org/challenges/LSVRC/. 46

[34] AlexNet. en.wikipedia.org/wiki/AlexNet. 47

[35] F.N. Iandola, S. Han, M.W. Moskewicz, K. Ashraf, W.J. Dally, K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size." arXiv preprint arXiv:1602.07360, 2016. 47

[36] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, "Going deeper with convolutions." arXiv preprint arXiv:1409.4842, 2014. 48

[37] C. Szegedy, S. Ioffe, V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning." arXiv preprint arXiv:1602.07261, 2016. 48

[38] K. Simonyan, A. Zisserman, "Very deep convolution networks for large-scale image recognition." arXiv preprint arXiv:1409.1556, 2014. 49

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Deep residual learning for image recognition." arXiv preprint arXiv:1512.03385, 2015. 50

[40] Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger, "Densely connected convolutional networks." arXiv preprint arXiv:1608.06993, 2016. 51

[41] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg, "Ssd: Single shot multibox detector." arXiv preprint arXiv:1512.02325, 2015. 53

[42] VOC2012. host.robots.ox.ac.uk/pascal/VOC/voc2012/. 53

[43] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, "You only look once:unified, real-time object detection." arXiv preprint arXiv:1506.02640, 2015. 54

[44] YoloV3. https://pjreddie.com/darknet/yolo/. 55

[45] COCO Dataset. http://cocodataset.org. 55

[46] Jonathan Long, Evan Shelhamer, Trevor Darrell, "Fully convolutional networks for semantic segmentation." IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015. 56

[47] Fisher Yu, Vladlen Koltun, "Multi-scale context aggregation by dilated convolutions." arXiv preprint arXiv:1511.07122, 2016. 57

[48] "Cityscapes." https://www.cityscapes-dataset.com/. 57

[49] Sik-Ho Tsang, "Review: Dilatednet — dilated convolution (semantic segmentation)." https://towardsdatascience.com/review-dilated-convolution-semantic-segmentation-9d5a5bd768f5. 57

[50] "Artificial neural network." en.wikipedia.org/wiki/Artificial_neural_network.

[51] "Convolutional neural network." en.wikipedia.org/wiki/Convolutional_neural_network.

[52] R. Demush, "A brief history of computer vision (and convolutional neural networks)." hackernoon.com/a-brief-history-of-computer-vision-and-convolutional-neural-networks-8fe8aacc79f3.

[53] S. Saha, "A comprehensive guide to convolutional neural networks." towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[54] V. Zhou, "An introduction to convolutional neural networks." victorzhou.com/blog/intro-to-cnns-part-1.

[55] R. Prabhu, "Understanding of convolutional neural network." medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148.

[56] "Digital signal processor." en.wikipedia.org/wiki/Digital_signal_processor.

[57] A. Shimoni, "A gentle introduction to hardware accelerated data processing." hackernoon.com/a-gentle-introduction-to-hardware-accelerated-data-processing-81ac79c2105.

[58] Caffe. en.wikipedia.org/wiki/Caffe_(software).

[59] ONNX. ai.facebook.com/tools/onnx/.

[60] ONNX. en.wikipedia.org/wiki/Open_Neural_Network_Exchange.

[61] Intel OpenVino Guide. docs.openvinotoolkit.org/latest/_docs_IE_DG_Introduction.html.

[62] B. Raj, "A simple guide to the versions of the inception network." https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202.

[63] S.-H. Tsang, "Review: Vggnet — 1st runner-up (image classification), winner (localization) in ilsvrc 2014." https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11.

[64] Sik-Ho Tsang, "Review: Resnet — winner of ilsvrc 2015 (image classification, localization, detection)." https://towardsdatascience.com/review-resnet-winner-of-ilsvrc-2015-image-classification-localization-detection-e39402bfa5d8.

[65] Sik-Ho Tsang, "Review: Ssd — single shot detector (object detection)." https://towardsdatascience.com/review-ssd-single-shot-detector-object-detection-851a94607d11.

[66] Ayoosh Kathuria, "What's new in yolo v3?." https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b.

[67] Sik-Ho Tsang, "Review: Fcn — fully convolutional network (semantic segmentation)." https://towardsdatascience.com/review-fcn-semantic-segmentation-eb8c9b50d2d1.