NATIONAL & KAPODISTRIAN

UNIVERSITY OF ATHENS

MSc in ”Control & Computing”
Departments of Physics and
Informatics & Telecommunicanions

Master Thesis

Convolutional Neural Network Accelerator on System on a Chip: Application at the Intel/Movidius Myriad2

Bezaitis Charalampos
2019508

Supervisor:
Dionysios Reisis, Professor

Examination Committee:
Hector E. Nistazakis, Associate Professor

Anna Tzanakaki, Associate Professor

July 2021

Περίληψη

Η παρούσα διπλωματική εργασία ασχολείται με την επιτάχνσυη Συνελεκτικών Νευρωρινκών Δικτύων(CNN) στην Myriad2.
Η διπλωματική αρχίζει με μία εισαγωγή στα CNN εστιάζοντας στις πράξεις των στρωμάτων από τα οποία αποτελούνται.
Η Myriad2, μία μονάδας επεξεργάσιας οράσεως, και μερικές από τις εφαρμογές της παρουσιάζονται στο δεύτερο κεφάλαιο.
Τέλος, ένας πρωτότυπος σχεδιασμός εφαρμοσμένος στην Myriad2 επιταχύνει την CNN επίλυση του προβλήματος της κατηγοριοποίησης χειρόγραφων ψηφίων και τέλος, ο σχεδιασμός αυτός συγκρίνεται και με αλλά συστήματα.

Abstract

This thesis considers an acceleration of the inference of a Convolutional Neural Network at Myriad 2.
CNNs are introduced with a focus on their inference calculations.
Myriad 2 Vision Processing Unit and its applications are presented next.
A novel VPU design that accelerates a CNN solution of classifying handwritten numbers is developed and is compared with other systems and designs.

Subject Area: System on a chip, Machine Learning, Computer Vision, Parallel Systems, Field Programmable Gate Array, Vision Processing Unit

Keywords: CNN, SoC, Myriad2, Neural Networks, Accelerator, Edge Computing, Intel/Movidius Myriad 2, FPGA, VPU, Myriad2

Contents

Chapter 1

Introduction

During the past decades the progress of machine learning algorithmic techniques has led to a significant improvement of different tasks. Though,a lot of applications impose restrictions on the execution time of the task and hardware accelerators were introduced to improve the time efficiency of the tasks. Consequently, researchers and engineers are involved in an ongoing effort to upgrade the performance by proposing novel accelerator designs while paying attention to the low energy consumption requirements of edge devices[20] [9].

Aiming at contributing to this effort the objective of thesis is to develop a CNN accelerator based on Myriad 2 System on a Chip. Myriad 2 Multi-processor System has proven capable of running machine learning tasks on previous work, thus was chosen for the image recognition task [12]. The classifying handwritten digits parallel software of this thesis contributed to a conference paper [8].

The thesis is structured as follows. In the second chapter, Convolutional Neural Networks are introduced with focus on the exact calculations of their inference in the second chapter. In the third chapter, Myriad 2 VPU is presented with its various applications. In the fourth chapter, a solution to classifying handwritten numbers is given resulting to the inference C program. In the fifth chapter, the porting of the inference to the parallel software running in Myriad 2 is depicted. In the sixth chapter, the results of the

parallel VPU software are presented and compared with other systems and implementations.

Chapter 2

Convolutional Neural Networks

1. Introduction

Neural networks were inspired by the first attempts of scientists to understand how the human brain works and how what we call intelligence is formed. The most basic building element of the brain is the **neuron** and a similar building block is the perceptron in the Artificial Neural Networks [19].

A learning machine that learns from a set of training data can be built by borrowing the brain neuron. A basic operation consist of one single neuron that following certain rules can learn to distinguish two linearly separable classes. This operation is called **perceptron**. A perceptron has ”synaptic” weights that are updated in order to perform different operations. The weights are updated, when the perceptron ”learns” the output for each of the input patterns. Memory neurons in the brain work in a similar way For example, in Fig. 2.1 a perceptron can be used to classify black and white dots.

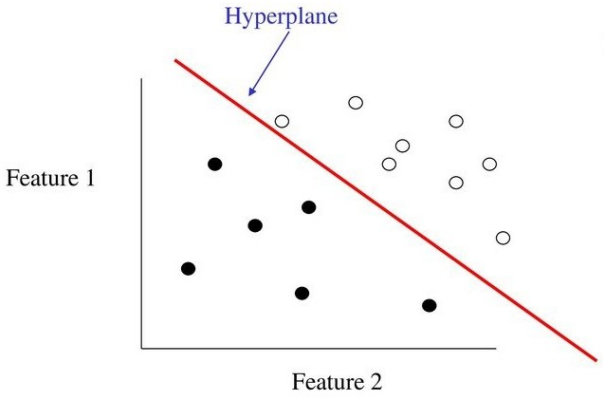

Figure 2.1: Linearly separable classes (black and white) can be separated by a single perceptron

By combing a large number of perceptrons, which are connected by different layers, the architecture of a **Artificial Neural Network**(ANN) is made. ANNs are able to classify multiple non-linear classes and achieve high results of accuracy.

Learning is achieved by adjusting the synaptic weights to minimize a preselected cost function. This adjusting of the weights became possible and efficient with the use of the backpropagation algorithm. Backpropagation train neural networks based on the set of input-output training samples.

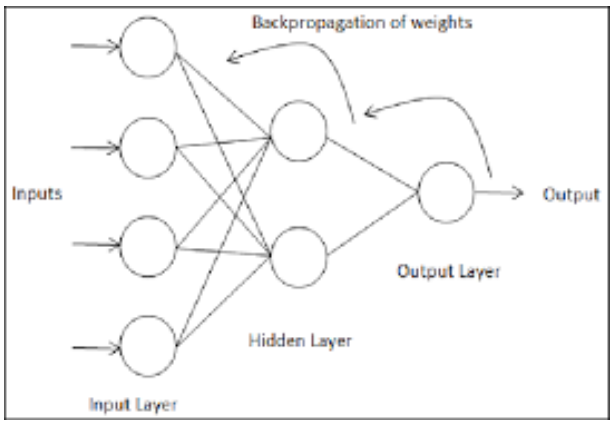

Figure 2.2: Simple depiction of how backpropagation updates the synaptic weights

Though, ANNs are very useful but in certain tasks such as image classification lack accuracy and efficiency. For example, vectorized image arrays result in a huge increase to the number of input parameters but information of the position of each pixel is lost. Also, in limited hardware like embedded/edge devices such number of parameters can not be sustained.

Convolutional Neural Networks(CNNs) offer a solution to both of this problems[18]. The first layers perform convolution sense the name. After the convolution layer, pooling and activation functions pre-processing is done before the input of the ANN and better results are achieved.

The next sections expand on each of the blocks of the CNN some of which were already refereed but not explained consistently. All these blocks are used to build the **inference** of a CNN.

2. Simple Perceptron

In the basic perceptron architecture the input features are applied to the input nodes and are weighted by the respective weights that define the synapses. The bias term is then added on their linear combination and the result is pushed through a nonlinear activation function.

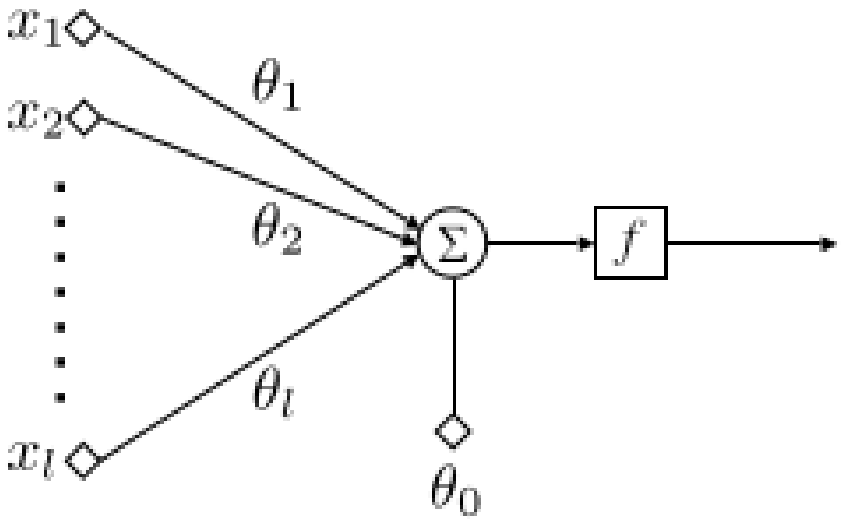

Figure 2.3: Perceptron - Neuron

For example, a simple neuron is able to classify patterns to 2 classes. It would output a 1 for patterns of the first class or a zero for patterns of the second class

3. Fully Connected Network

The feed-forward networks that have been introduced before as ANN are also known as fully connected networks.

This name is to stress out that each one of the neurons/nodes in any layer is directly connected to every node of the previous layer. The nodes of the first hidden layer are fully connected to those of the input layer. In other words, each neuron is associated with a vector of parameters, whose dimension is equal to the number of nodes of the previous (input) layer. The algebraic operations which are performed are inner products.

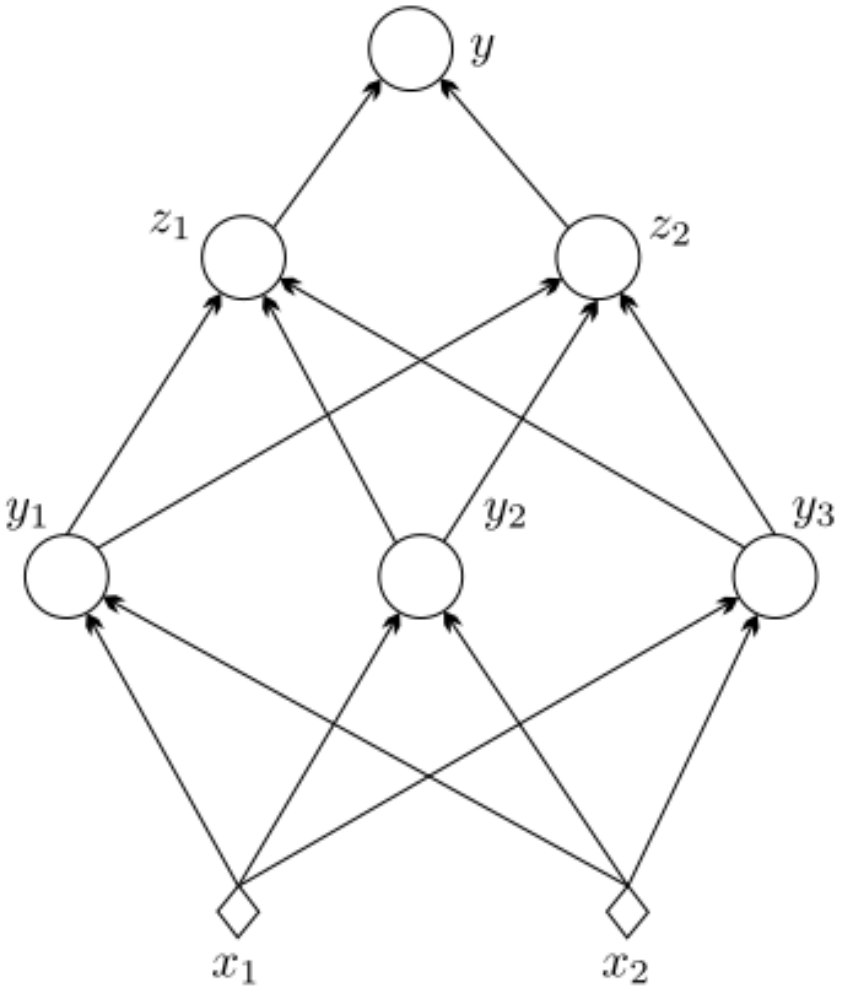

Figure 2.4: Fully Connected Network

In the Fig. 2.4 a small network is displayed consisting of two layers. The first one has 3 neurons and the second one 2 neurons. It is easily seen that each node is connected to every node of the ”forward” layer.

4. Convolution

Convolution in Neural Nets resembles matrix multiplication element by element. A kernel is multiplied element by element with the bigger image array. The output is stored to a new array and then the operation is repeated by ”sliding” to cover the rest of the input matrix.

An example helps to deepen the understanding of the operations used in convolution: We start by defining the two matrices as the Input Matrix (Fig . 2.5) and the kernel matrix (Fig. 2.6)

$$\begin{bmatrix} IM(1,1) & IM(1,2) & IM(1,3) \\ IM(2,1) & IM(2,2) & IM(2,3) \\ IM(3,1) & IM(3,2) & IM(3,3) \end{bmatrix}$$

Figure 2.5: Input Matrix

$$\begin{bmatrix} K(1,1) & K(1,2) \\ K(2,1) & K(2,2) \end{bmatrix}$$

Figure 2.6: Kernel

We place the kernel at the top left part of the Input Matrix and the first output is derived:

$$O(1,1) = IM(1,1)*K(1,1)+IM(1,2)*K(1,2)+IM(2,1)*K(2,1)+IM(2,2)*K(2,2)$$

The next iteration for the second output is:

$$O(1,2) = K(1,1)*IM(1,2)+K(1,2)*IM(1,3)+K(2,1)*IM(2,2)+K(2,2)*IM(2,3)$$

Hence by following the above pattern a 2X2 array is formed.

$$\begin{bmatrix} O(1,1) & O(1,2) \\ O(2,1) & O(2,2) \end{bmatrix}$$

Figure 2.7: Output Array

Terms used usually with CNNs:

- Depth: The depth of a layer is the number of filter matrices that are employed in this layer. This is not to be confused with the depth of the network, which corresponds to the total number of hidden layers used. Sometimes, we refer to the number of filters as the number of channels.
- Receptive field: Each pixel in an output feature map array results as a weighted average of the pixels within a specific area of the input (or of the output of the previous layer) image array. The specific area that corresponds to a pixel is known as its receptive field.
- Stride: In practice, instead of sliding the filter matrix one pixel at a time(just as the previous example), one can slide it by, say, s pixels. This value is known as the stride. For values of s > 1, feature map arrays that are smaller in size result
- Zero padding: Sometimes, zeros are used to pad the input matrix around the border pixels.
- Bias term: After each convolution operation that generates a feature map pixel, a bias term, b, is added.

5. Activation Function

Once convolutions have been performed and the bias term has been added to all feature map values, the next step is to apply nonlinearity (activation function) to each one of the pixels of every feature map array. A lot of nonlinear functions can be employed such as tanh, sigmoid, hyperbolic tangent and . Currently, the rectified linear activation function, **ReLU**, seems to be the most popular and efficient.
In the context of artificial neural networks, ReLU is an activation function defined as the positive part of its argument:

$$f(x) = max(0, x)$$

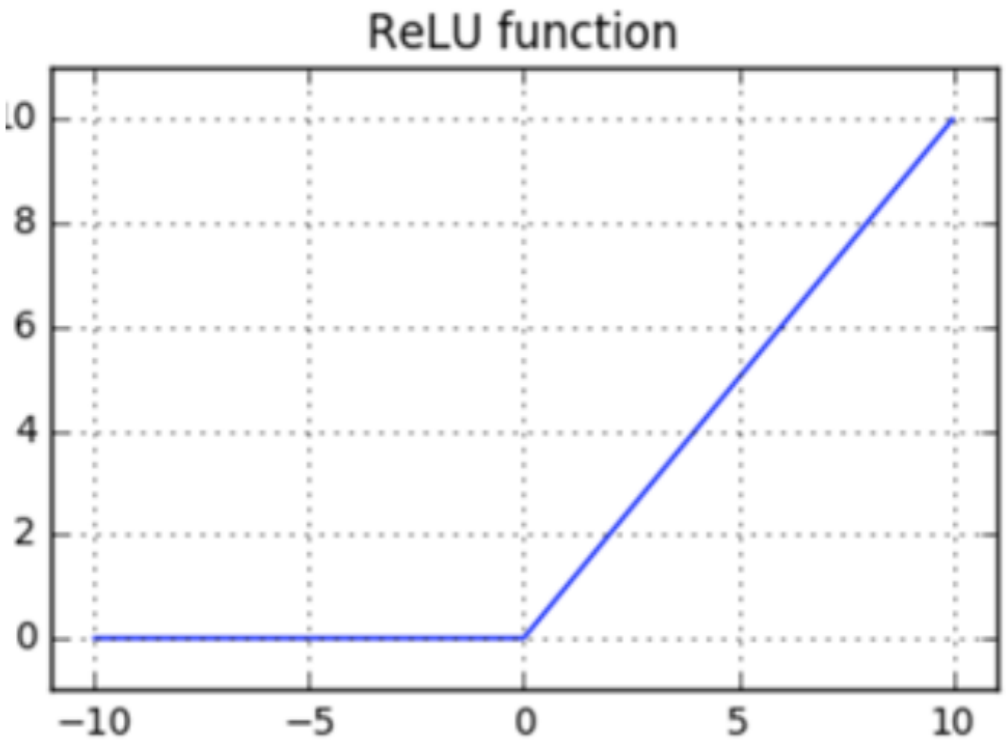

Figure 2.8: ReLU activation function

6. Pooling

The purpose of this step is to reduce the dimensionality of each feature map array. Sometimes, the step is also referred to as spatial pooling. To this end, one defines a window and slides it over the corresponding matrix. Sliding can be done by adopting a value for the respective stride parameter, s. Pooling operation consists of choosing a single value to represent all the pixels that lie within the window. The most commonly used operation is the **max pooling**. Max pooling operates as follows: among all the pixels that lie within the window, the one with the maximum value is selected.

$$\begin{bmatrix} 2 & 1 & 5 & 7 \\ 3 & 2 & 1 & 7 \\ 1 & 3 & 2 & 6 \\ 9 & 4 & 1 & 8 \end{bmatrix}$$

Figure 2.9: Input Matrix

$$\begin{bmatrix} 3 & 7 \\ 9 & 8 \end{bmatrix}$$

Figure 2.10: Matrix after Max Pooling

7. Output layer

After multiple layers of convolution and padding, we would need the output in the form of a class. Convolution and pooling layers would only be able to extract features and reduce the number of parameters from the original images.

However, to produce the final output we need to apply a fully connected layer to generate an output equal to the number of classes we need, and thus the biggest output indicates the predicted class.

8. Training and Inference of a CNN

Training refers to the process of creating the Convolution Neural Network. Training involves the use of a deep-learning framework (e.g., TensorFlow) and a training dataset. During training, by using various techinques such as backpropagation, the model updates the weights, biases and kernels to achieve the optimal results regarding the expected output.

Inference refers to the process of using the previously calculated variables of the CNN model to make a prediction. In this way,

inference is used to predict the results from the previously unseen input data. The prediction could be classification to a category.

Chapter 3

Myriad 2

1. Myriad 2 System-on-Chip Presentation

Myriad 2 is a multicore, always-on System On Chip that supports computational imaging and visual awareness for mobile, wearable, and embedded applications. Myriad 2 is a vision processing unit (**VPU**) solution for devices that both power and thermal dissipation are key issues and need to be kept at a minimum [15].

Myriad 2 uses a combination of low-power very long instruction word (VLIW) processors with vector and Single Instruction Multiple Data (SIMD) operations capable of very high parallelism in a clock cycle, allied with hardware acceleration for key image processing and computer vision kernels, backed by a very high band width on-chip multicore memory subsystem. The Myriad2 VPU aims to provide high performance efficiency, allowing high-performance computer vision systems with very low latency to be built while dissipating less than 1 W[16].

More specifically, Myriad's 2 design in process technology was able to sustain 12 Streaming Hybrid Architecture Vector Engine (SHAVE) vector processors with a clock rate up to 600 MHz. It also combines two RISC processors LEON OS and LEON RT with the same clock rate. The 2 Leon processors support real time operating systems and various peripheral devices.

Figure 3.1: Intel's Myriad 2 Vision Processing Unit during radiation testing for space at CERN [3]

Another integral part of the VPU is the **shared memory**. All the processors and their instructions reside in a shared 2-MByte memory block called Connection Matrix (CMX) memory, which can be configured to accommodate different instruction and data mixes depending on the workload. The CMX block comprises 16 blocks of 128 Kbytes, which in turn comprise four 32-Kbyte RAM instances organized as 4,096 words of 64 bits each, which are independently arbitrated, allowing each RAM block in the memory subsystem to be accessed independently. The 12 SHAVEs acting together can move (theoretical maximum) 12 x 128 bits of code and 24 x 64 bits of data, for an aggregate CMX memory bandwidth of 3,072 bits per cycle (1,536 bits of data).

2. Myriad 2 Architecture

The **SHAVE** processor is a hybrid stream processor architecture combining the best features of GPUs, DSPs, and RISC with both 8-bit, 16-bit, and 32-bit integer and 16- and 32-bit floating-point

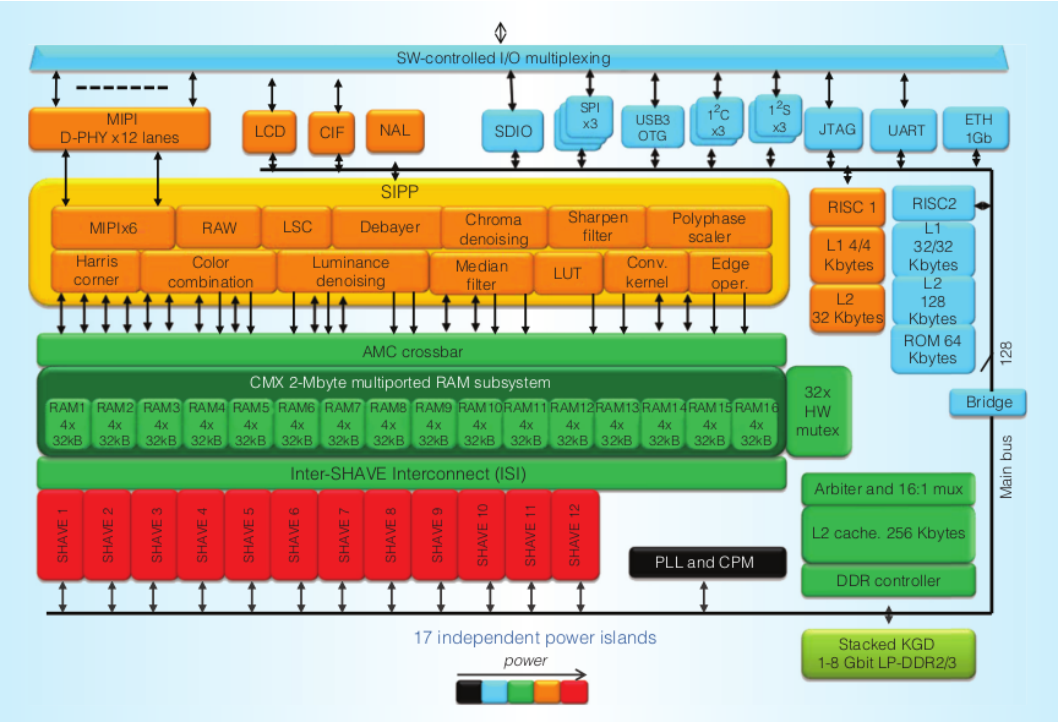

Figure 3.2: Myriad 2 Architecture

arithmetic as well as unique features such as hardware support for sparse data structures. These configurations prove especially useful for designing and using the fast but small CMX memory. It also controls multiple functional units including extensive SIMD capability for high parallelism and throughput at both a functional unit and processor level. This architecture maximizes performance per watt while maintaining ease of programmability, especially in terms of support for design and porting of different software applications.

Because power efficiency is paramount in mobile applications, Myriad 2 provides extensive clock and functional unit gating and support for dynamic clock and voltage scaling for dynamic power reduction. It also contains 17 power islands:

- one for each of the 12 SHAVE processors;
- one for the CMX memory subsystem; one for the media subsystem, including video hardware accelerators and RISC1
- one for RISC2 and peripherals

- one for the clock and power management
- one always-on domain

This allows fine-grained power control in software with minimal latency to return to normal operating mode, including maintenance of the static RAM (SRAM) state that eliminates the need to reboot from external storage. Myriad 2 has been designed to operate at 0.9 V for 600 MHz.

3. Myriad 2 Applications

Myriad 2 VPU has found use in various projects such as Google Project Tango, Google Clips and DJI Drones. It is able to perform in such projects as it runs at between 80 and 150 GFLOPS on little more than 1W of power[22]. These results lead also to exploring the use of Myriad 2 for space applications [11].

3..1 European Space Agency use of Myriad 2

One of the many organizations that use Myriad 2 is the European Space Agency. After extensive testing, Myriad 2 is currently being used on a experimental satellite[4].

One of the most important requirements to test the suitability of a chip to fly is the radiation hardening. Damage or malfunction can be caused by high levels of ionizing radiation (particle radiation and high-energy electromagnetic radiation). This is especially true for environments in outer space where the Earth atmosphere does not exist[2].

In such manner, an ESA-led team subjected Myriad 2 chip to one of the most energetic radiation beams available on Earth. These tests took place at CERN, the European Organization for Nuclear Research. ESA worked with Irish firm Ubotica Technologies to put chips in a path of an experimental beamline fed by the Super Proton Synchrotron (SPS) particle accelerator [21]. Located in a circular tunnel nearly

7 km in circumference, the SPS is CERN's second largest accelerator after the Large Hadron Collider (LHC), which the SPS feeds into in turn.

Myriad 2 could let us overcome the performance bottleneck faced by imaging instruments on CubeSats and other small satellites. Low data downlink bandwidth due to a small antenna size and limited power levels stops us from accessing all the imagery we could acquire [1].

On some extent, this was full-filled by ESA's Φ-sat-1 mission. Φ-sat-1, an enhancement of the Federated Satellite Systems (FSSCat) mission, is one of the first experiment to demonstrate how artificial intelligence can be used for Earth observation – in this case, filtering out less than perfect images so that only usable data are returned to Earth.

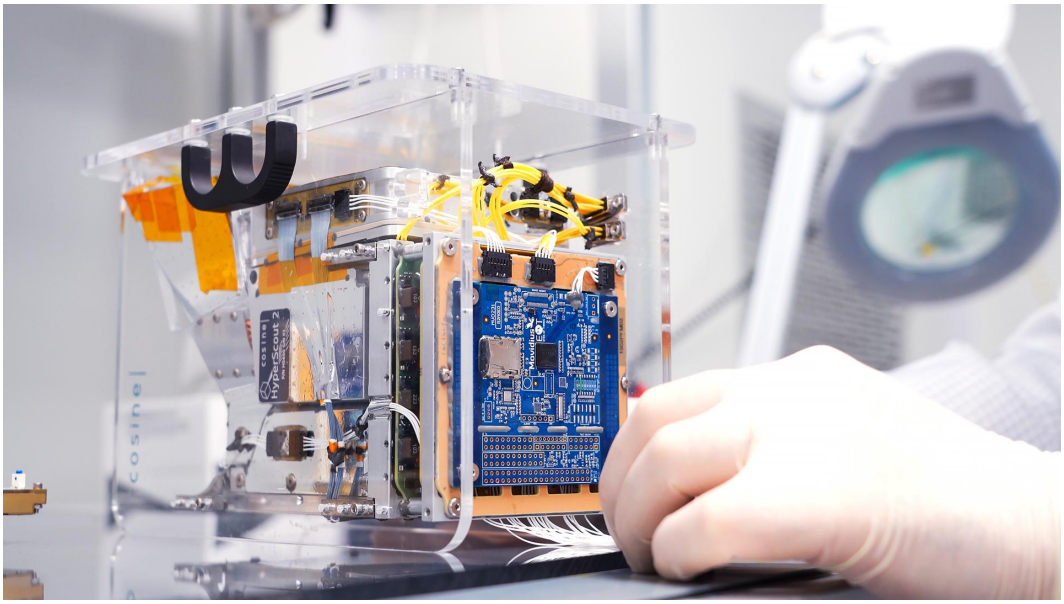

Figure 3.3: Intel's Myriad 2 Vision Processing Unit brings onboard artificial intelligence to satellites on a system build by Ubotica Technologies and paired with a hyperspectral-thermal from cosine measurement systems[6]

On September 2, 2020 the experimental satellite Φ-sat-1 ,about the size of a cereal box, was ejected from Vega rocket's dispenser along with 45 other similarly small satellites. The satellite is now soaring at over 17,000 mph (27,500 kmh) in sun-synchronous orbit about 329 miles (530 km) overhead.

Φ-sat-1 contains a new hyperspectral-thermal camera and onboard AI processing thanks to Myriad 2. For the initial verification, the satellite saved all images and recorded its AI cloud detection decision for each, in order for the team on the ground to be able verify that the Deep CNNs running on Myriad 2 were behaving as expected.

The initial data downlinked from the satellite has shown that the AI-powered automatic cloud detection algorithm has correctly sorted hyperspectral Earth observation imagery from the satellite’s sensor into cloudy and non-cloudy pixels.

In conclusion, Φ-sat-1 has enabled the pre-filtering of Earth observation data. Thus, only relevant part of the image with usable information are downlinked to the ground, thereby improving bandwidth utilisation and significantly reducing aggregated downlink costs.

4. Myriad 2 Interfaces

4..1 CIF

The Camera Interface of Myriad 2 is a single image input interface. With the use of multiple GPIOs Myriad 2 receives images of various sizes and formats accordings to its CIF standard. The frames are transmitted and received by CIF in a bit-parallel format. The GPIOs that implement the interface are the following: vsync, hsync, pixel clock and pixel data. Pixel data bits depend on the CIF format that is being used each time.

4..2 LCD

Liquid Cristal Display supports single image output. Similar to the Camera Interface, with the use of GPIOs Myriad outputs images of different sizes and depths.

4..3 Accelerator

Both interfaces enable Myriad to work as an accelerator for machine learning or to be more specific CNN implementations. Input data in the form of image frames can be received from CIF. After the reception, Myriad2 process the input data (classifying,convolution, rendering etc) and output the results with the use of LCD.

These interfaces can be implemented with any other device capable of having a sufficient number of GPIO pins (with clocks) such as an Arduino or even a FPGA[17]. So, with an established communication Myriad2 can work as accelerator for them[5].

Chapter 4

MNIST CNN solution

1. Model Architecture and Training

The model was trained with the MNIST dataset containing 60K total handwritten digit grayscale images with dimensions 28x28 pixels [10]. Out of the 60K images, 50K where selected for the training process and the remaining 10K were used for the model validation.

Figure 4.1: Mnist Dataset Images

The dataset images were given as input to the CNN depicted on Fig. 4.1, which was implemented via TensorFlow [13].

Estimator API consisting of the following operations:

1. one convolution layer containing 32 filters of 5×5 kernel size each
2. max pooling layer dedicated to downsample the feature maps by extracting the max value of 2 × 2 windows with stride 2
3. a 30-neuron fully connected layer
4. a 10-neuron output layer; this output result represents the final classification task

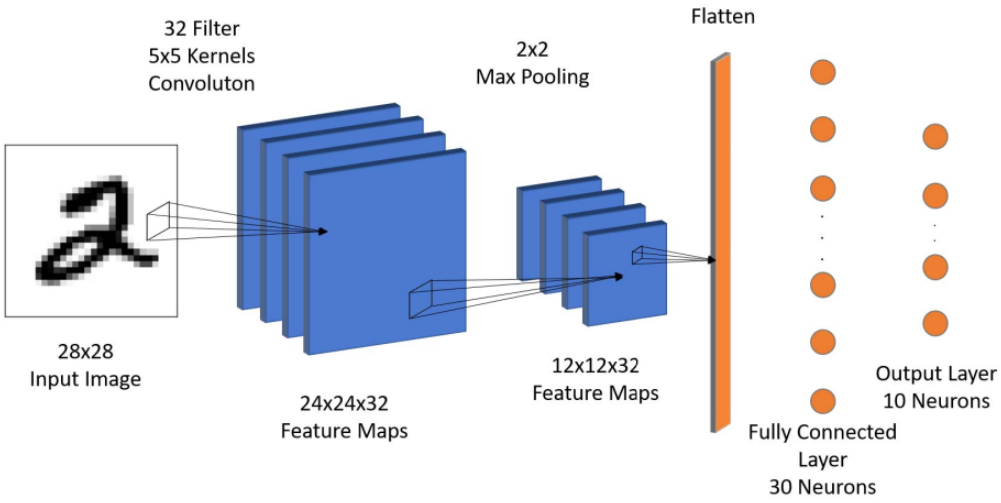

Figure 4.2: Model Architecture

The implementation results showed that the accuracy of the model was not affected by either using or not using padding. Consequently, no padding was perfomed for our model in the convolution layer and thus, the CNN design minimized the unnecessary computations in Myriad 2. The model uses the ReLU activation function in all the layers of the model's architecture apart the output layer. The latter choice was taken because ReLU provides the best performance with respect to the computational cost ratio and it is efficiently implemented in hardware regarding the resources' utilization. The output layer calculates the argmax and softmax of the output layer's

tensors in order to produce the classification. The model uses the Adam Optimization Algorithm with input from the cross-entropy loss function and it achieves 98.66% accuracy after 9 epochs.

2. Inference Software C Implementation

A C software program was developed that calculates the Inference of the CNN model. The C software uses the same parameters of the TensorFlow application. Biases, weights and a random image were extracted from the Tensorflow API and were added to the C implementation by organizing them as header files.

The inference is calculated in the following steps:

1. It calculates the convolution of the input image with the 1st filter's kernel.
2. It adds the corresponding bias and applies the ReLU function.
3. To the resulting array it performs the max pooling operation.
4. It repeats the steps 1, 2, 3 until it calculates the feature map of each filter of the Convolution Layer, by calling the corresponding function in each iteration.
5. It calculates the matrix multiplication of the Fully Connected Layer: one neuron at a time. It calculates the dot product of the feature map array with the weight array of each neuron of the Fully Connected Layer.
6. Finally, it calculates the matrix multiplication of the output layer resulting in the classification of the network. The results are not normalized into a probability distribution.

The code can be seen on the appendix section 1..

Chapter 5

Myriad 2 CNN Accelerator

After the Inference C implementation, the MNIST solution was ported to Myriad Software. The design of the Myriad2 CNN Accelerator is revolved around the parallel architecture of the SoC. The design utilizes all the 12 available SHAVE processors in order to reduce the inference execution time and increase the throughput of the accelerator.

The Myriad2 Accelerator is designed under the bare metal programming paradigm, which allows the use of the Leon processors without any operating system and with minimal schedulers to control the pipeline of applications. To provide extra acceleration, the SHAVEs use Single Instruction Multiple Data routines for the main computations.

The accelerator incorporates one Leon processor (OS) operating as the scheduler of the 12 SHAVEs. The Myriad2 Accelerator omits the use of Real-Time Executive for Multiprocessor Systems (RTEMS) to eliminate the operating system overhead in the cost of integration efforts for the developer.

The Myriad2 CNN Accelerator is efficient regarding the utilization of the available memory resources of t the SoC. Given the fact that the application has low memory requirements, the aim is to use of the CMX memory for storing all the neural network parameters (weights, biases) and the input image. The performance of the accelerator is vastly improved without the degradation caused by the use of the DRAM.

Moreover, the SHAVE code can be stored either in the CMX memory or in the DRAM. The Myriad2 Accelerator takes advantage of this feature, by storing the SHAVE code in the DRAM memory and the application data are stored in the CMX. This division of the memory provides a major performance gain because the program data reside in the fastest possible memory in the SoC. The fact that the program code resides in the DRAM memory proves not to be a bottleneck. because of the caching capabilities of the Myriad2 SoC. Myriad 2 caching reduces the DRAM performance penalty when fetching instructions from it.

The mapping of the C software implementation to the 12 SHAVE processors, operating in parallel, is as follows:

1. The computations of the 32 filters of the Convolution Layer and the corresponding max pooling operations are mapped onto the available processors on a per filter basis (e.g. when the 32 filters are mapped onto 12 SHAVEs: each of the first 8 SHAVEs performs the convolution of 3 filters and each of the last 4 SHAVEs performs the convolution of 2 filters).

2. The completion of the calculations of the Convolution and Pooling Layers constitutes an execution barrier, reaching which, indicates that the input of the Fully Connected Layer is available. Each of the 30 neurons of the Fully Connected Layer is mapped onto a SHAVE processor: each of the first 6 SHAVEs performs 3 dot products and each of the last 6 SHAVEs performs 2.

3. Finally, when all SHAVEs finish the execution of the Fully Connected layer, this constitutes a second barrier, then the calculation of the values of the 10 neurons of the Output Layer will be completed.

   The organization of the SHAVE data in the CMX memory is critical to the performance of the accelerator as well as the power consumption of the accelerator. Targeting the reduction of the data sharing and consequently the memory access clashes among the SHAVEs, data redundancy was essential. For the convolution layer data redundancy is accomplished by storing multiple copies of the convolution parameters as many as the number of SHAVE processors: one copy of the input image, the convolution layer’s weights and biases is stored in each SHAVE’s CMX space.

The weights of the Fully Connected Layer are stored in a common access region of the CMX memory, due to their size that restricts the data redundancy.

In the appendix section 2. the SIMD pseudo-code running on the shaves can be seen.

Chapter 6

Results

Table 6.1 presents the Myriad2 parallel inference implementation results. The CNN design on the Myriad2 shows an almost linear speedup with respect to the number of SHAVE processors employed in the computations, which can be seen on Fig. 6.1. This proves the high parallelization of the design and the ability of the Shaves to run efficiently and concurrently.

Table 6.1: Myriad 2 Parallel Implementation

# of Shaves	**1**	**2**	**4**	**8**	**12**
Execution Time (ms)	1.82	1.04	0.54	0.39	0.35
Speedup	-	1.75	3.37	4.66	5.2
Power Consumption (mW)	510	529	563	636	707

The relation of execution time of the Myriad2 parallel software implementation with respect to the number of SHAVE processors that are utilized in the design is presented in Fig. 6.2.

Moreover, in Fig. 6.3 it is observable that, there is a small increase in the power consumption in relation with the number of SHAVE processors. Each additional SHAVE processor adds 15 mW to the total power consumption of the SoC. Therefore, the Myriad2 CNN accelerator is advantageous because increasing the number of used SHAVEs results in higher speedup at a small penalty in the power consumption.

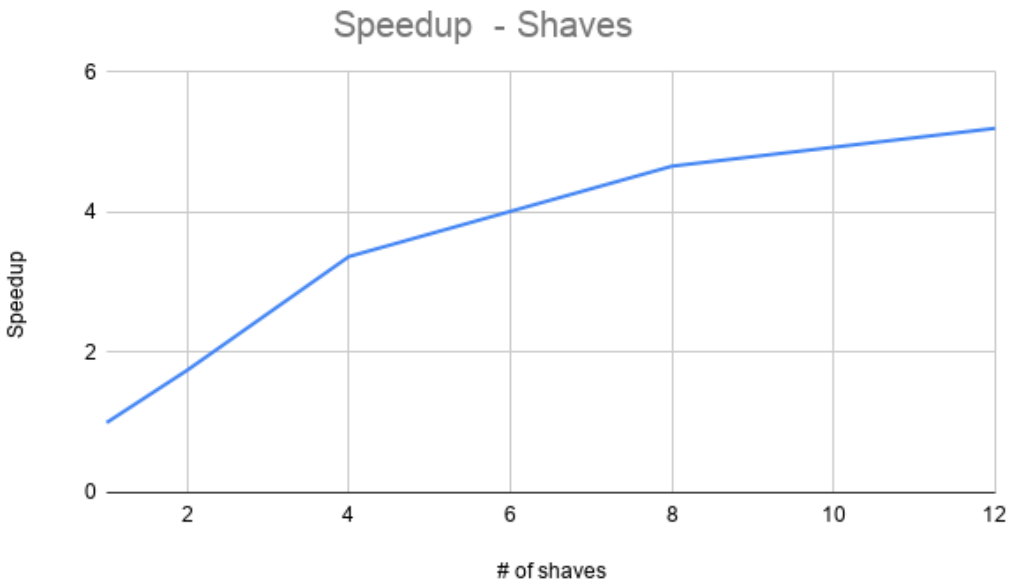

Figure 6.1: Speed up of Myriad 2 Accelerator(without RTEMS) with the number of Shaves

1. Comparison with Intel commercial CPU

On the Intel® Core i7-1065G7 CPU the C software executes at 1.80 ms , as it can be seen on Table 6.2. Intel's CPU executed the sequential implementation of the C code at 1.80 ms, when the power consumption of the CPU is at 15 Watt[7].

Table 6.2: Intel CPU C sequential implementation

Devices	Myriad 2: 12 Shaves	Intel i7-1065G7
Execution time (ms)	0.35	1.80
Power Consumption(W)	0.707	15

Myriad 2, when executing the parallel software was able to achieve performance 5x performance gain of the Intel CPU when also, consuming 20 times less power.

It should be noted that the Intel CPU was running the inference in only one core, but the VPU's model is able to surpass the CPU and proves its capabilities in such a low power environment.

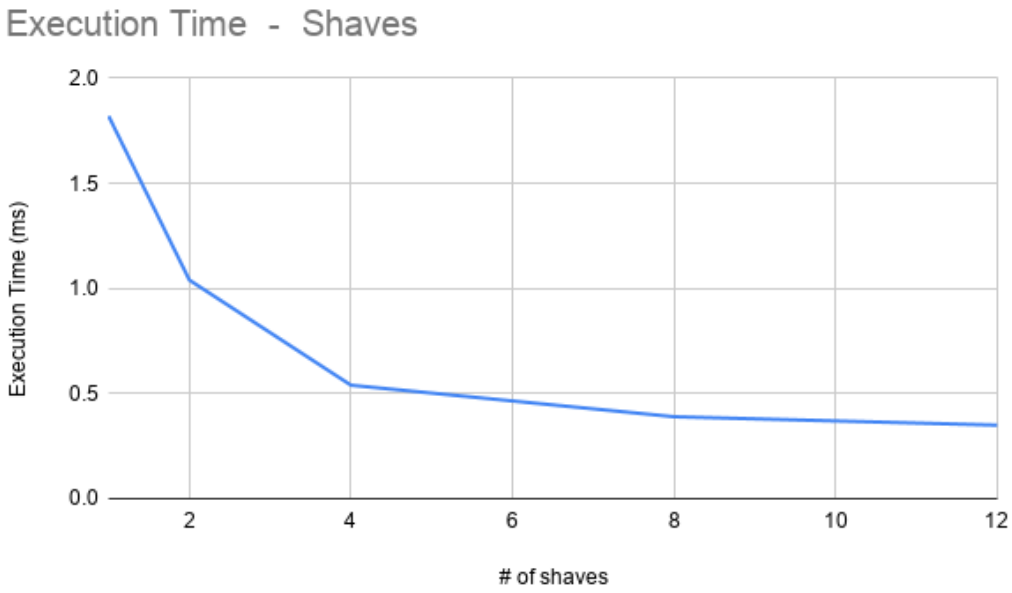

Figure 6.2: Execution time of Myriad 2 Accelerator(without RTEMS) with the number of Shaves

2. Comparison with a FPGA implementation

To continue comparing the Myriad 2 CNN design, a FPGA implementation is used [9]. The FPGA implementation runs the same CNN model with the same layers and weights on a Xilinx Kintex Ultrascale board.

Table 6.3 shows a 16x performance advantage of the FPGACNN accelerator compared to the bare metal implementation (without RTEMS) of the CNN on the Myriad 2 SoC.

Table 6.3: FPGA implementation

Devices	Myriad 2: 12 Shaves	FPGA
Execution time (ms)	0.35	0.021
Power Consumption(W)	0.707	3.631

On the other hand, the Myriad 2 requires 5 times less power than the FPGA accelerator for completing the same CNN calculations.

FPGA implementation proves to be much faster than the Myriad

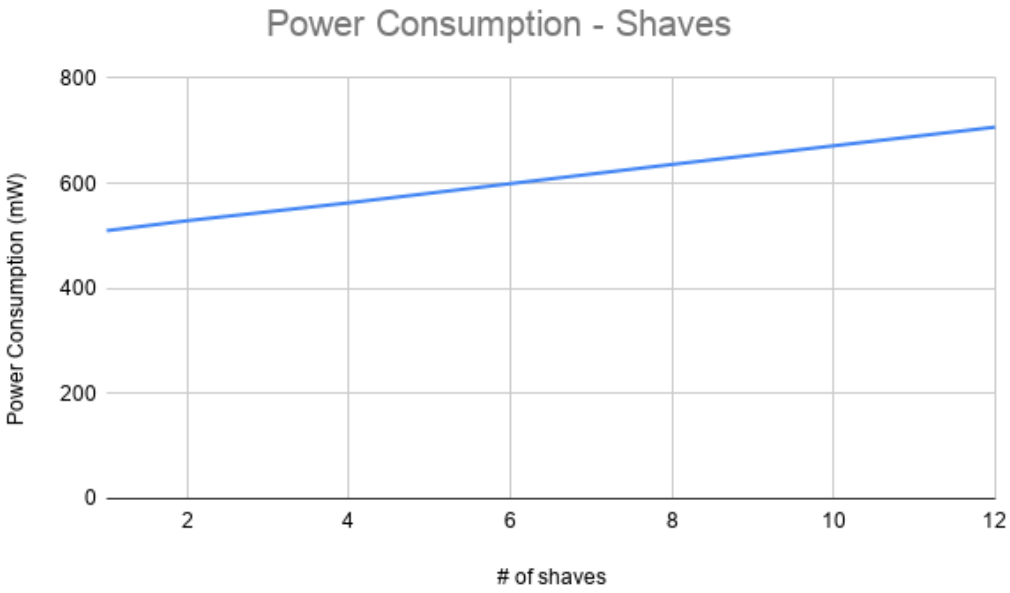

Figure 6.3: Power Consumption of Myriad 2 Accelerator(without RTEMS) with the number of Shaves

implementation but also consuming more power. Thus, Myriad 2 is directly comparable with the FPGA implementation and can even be preferred for low power high performance applications.

3. Comparison with the use of RTEMS and Bare Metal

Table 6.4: RTEMS and Bare Metal paradigm

Paradigms	Execution time (ms)
RTEMS	0.35
Bare Metal	0.707

Table 6.4 shows an almost 0.1 ms performance edge of the bare metal implementation compared to the Real-Time Executive for Multiprocessor Systems(RTEMS) paradigm. The advantage of the bare metal implementation is more evident in the designs utilizing more SHAVE processors, where the execution time of the CNN accelerator is in the same order of magnitude with the RTEMS overhead.

However, it should be noted that in deeper/bigger CNNs the RTEMS overhead may not be remarkably evident. In deeper CNNs, RTEMS could be a valuable asset in optimizing the CNN solution and achieving high performance with less development effort.

Chapter 7

Conclusion and Future Work

The current thesis has presented a CNN accelerator design on the Intel Myriad 2 SoC, which is optimized for low power image processing applications. The VPU accelerator when compared with a commercial CPU, has a significant advantage with respect to the CNN execution time . The VPU accelerator when compared with the FPGA accelerator, has a significant advantage with respect to performance per Watt.

Future work includes the design and implementation of more complicated CNN based applications for the Myriad 2 SoC that gained attention due to the power consumption efficiency.

One such example is a ship detection algorithm. Ship detection CNN ,while it’s output is only a yes or no for the existence of a ship in the input image , requires more computational calculation and it uses more memory. This is because of the input RGB images (instead of black/white MNIST images) and in order to achieve higher accuracy more layers are needed. To deal with such a problem, the current Myriad 2 software design should be enhanced. Effective utilization of DRAM memory would be essential. Another important result will be the use of DMA driver for the communication between the CMX and DRAM. An additional considerably improvement with respect to execution time maybe achieved by including assembly code in the current implemented C programs.

Chapter 8

References

[1] Diego Dantas et al. *Testbed for Connected Artificial Intelligence using Unmanned Aerial Vehicles and Convolutional Pose Machines*. Jan. 2020.

[2] ESA. *Digital Twin Earth, quantum computing and AI take centre stage at ESA's Φ-week*. 2020. url: https://www.esa.int/Applications/Observing_the_Earth/Digital_Twin_Earth_quantum_computing_and_AI_take_centre_stage_at_ESA_s_Ph-week.

[3] ESA. *ESA team blasts Intel's new AI chip with radiation at CERN*. 2018. url: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/ESA_team_blasts_Intel_s_new_AI_chip_with_radiation_at_CERN.

[4] ESA. *Next artificial intelligence mission selected*. 2020. url: https://www.esa.int/Applications/Observing_the_Earth/Ph-sat/Next_artificial_intelligence_mission_selected.

[5] Cobham Gaisler. *GR-HPCB-FMC-M2 Mezzanine Board*. 2020. url: https://www.gaisler.com/index.php/products/boards/gr-vpx-xcku060.

[6] Intel. *Intel Powers First Satellite with AI on Board*. 2020. url: https://www.intel.com/content/www/us/en/newsroom/news/first-satellite-ai.html/#gs.4pihbh.

[7] Intel. *Intel® Core™ i7-1065G7 Processor (8M Cache, up to 3.90 GHz)*. 2019. url: https://www.intel.com/content/www/us/en/products/sku/196597/intel-core-i71065g7-processor-8m-cache-up-to-3-90-ghz/specifications.html.

[8] Angelos Kyriakos et al. “Design and Performance Comparison of CNN Accelerators Based on the Intel Movidius Myriad2 SoC and FPGA Embedded Prototype”. In: *2019 International Conference on Control, Artificial Intelligence, Robotics Optimization (ICCAIRO)*. 2019, pp. 142–147. doi: 10.1109/ICCAIRO47923.2019.00030.

[9] Angelos Kyriakos et al. “High Performance Accelerator for CNN Applications”. In: *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2019, pp. 135–140. doi: 10.1109/PATMOS.2019.8862166.

[10] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). url: http://yann.lecun.com/exdb/mnist/.

[11] Vasileios Leon et al. “Improving Performance-Power-Programmability in Space Avionics with Edge Devices: VBN on Myriad2 SoC”. In: *ACM Transactions on Embedded Computing Systems* 20 (Mar. 2021), pp. 1–23. doi: 10.1145/3440885.

[12] Charalampos Marantos et al. “Efficient support vector machines implementation on Intel/Movidius Myriad 2”. In: *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. 2018, pp. 1–4. doi: 10.1109/MOCAST.2018.8376630.

[13] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. url: http://tensorflow.org/.

[14] Theo J. Mertzimekis. *UoA Thesis Template*. 2016. url: http://mertzimekis.gr/home/links/latex-linux.

[15] D. Moloney et al. “Myriad 2: Eye of the computational vision storm”. In: *2014 IEEE Hot Chips 26 Symposium (HCS)*. 2014, pp. 1–18. doi: 10.1109/HOTCHIPS.2014.7478823.

[16] David Moloney et al. “Myriad 2: Eye of the computational vision storm”. In: Aug. 2014, pp. 1–18. doi: 10.1109/HOTCHIPS.2014.7478823.

[17] Joaquín España Navarro et al. “High-Performance Compute Board - A Fault-Tolerant Module for On-Board Vision Processing”. In: *European Workshop on On-Board Data Processing*. June 2021, pp. 1–7.

[18] AI Research. *Deep Neural Networks for Acoustic Modeling in Speech Recognition*. 2015. url: http://airesearch.com/ai-research-papers/deep-neural-networks-for-acoustic-modeling-in-speech-recognition/.

[19] Sergios Theodoridis. “Machine Learning - A Bayesian and Optimization Perspective”. In: 2nd ed. Academic Press, 2020. Chap. 18, pp. 902–1029.

[20] Foivos Tsimpourlas et al. “A Design Space Exploration Framework for Convolutional Neural Networks Implemented on Edge Devices”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2212–2221. doi: 10.1109/TCAD.2018.2857280.

[21] Ubotica. *UB0100 CubeSat Board*. 2020. url: http://ubotica.com/ub0100/.

[22] Wikipedia. *Vision processing unit*. 2021. url: https://en.wikipedia.org/wiki/Vision_processing_unit.

Chapter 9

Appendix

1. Mnist Software Inference C code

main.c

```
#include<stdio.h>
#include”image.h”
#include”biases.h”
#include”dense_weights_0.h”
#include”dense_weights_1.h”
#include <time.h>


#define N 784

void conv(const float *img, const float *kernels, float conv_bias,
    float *conv_layer_out, int count);
float relu(float x);
float max(float a, float b, float c, float d);
float fully_connected(const float *dense_weights, float *
   conv_layer_out, const float dense_bias_0, int count);
float output_layer(const float *dense_weights, float *
   conv_layer_out, const float dense_bias_1);

int main() {

  clock_t begin=clock();
  int i,j;
  float conv_layer_out[32][12*12];
  float fully_connected_out[30];
  float output_layer_out[30];
```

```
24
25
26
27    for(i=0; i<32; i++) {
28      conv(img, kernels[i], conv_biases[i], conv_layer_out[i], i);
29    }
30
31    for(i=0; i<30; i++) {
32      fully_connected_out[i] = fully_connected(&dense_weights_0[i
       ][0], &conv_layer_out[0][0], dense_bias_0[i], i);
33    }
34
35
36
37    for(i=0; i<10; i++) {
38      output_layer_out[i] = output_layer(dense_weights_1[i],
       fully_connected_out, dense_bias_1[i]);
39    }
40
41    printf("\nOutputLayer\n");
42    for(i=0; i<10; i++) {
43      printf("%f\n", output_layer_out[i]);
44    }
45
46      clock_t end = clock();
47      double time_spent=(double)(end-begin)/CLOCKS_PER_SEC;
48
49      printf("\n Time taken = %.10f \n",time_spent);
50      return 0;
51  }
52
53  void conv(const float *img, const float *kernels, float conv_bias,
       float *conv_layer_out,int count) {
54    float conv_out[24*24];
55    int i,j,k,l;
56    float acc = 0.0;
57    //convolution
58    for(i=0; i<24; i++) {
59      for(j=0; j<24; j++){
60        acc = 0.0;
61        for(k=0; k<5; k++){
62          for(l=0; l<5; l++){
63            acc += *(img + (i+k)*28+(j+l)) * *(kernels + k*5 + l);
64          }
65        }
66      *(conv_out + i*24 + j) = relu(acc + conv_bias);
67      }
68    }
69    //pooling
70    k = 0;
71    l = 0;
```

```
  for(i=0; i<24; i=i+2) {
    l = 0;
    for(j=0; j<24; j=j+2){
      *(conv_layer_out +k*12+l) = max(*(conv_out + i*24 + j), *(conv_out + i*24 + j+1),
                      *(conv_out + (i+1)*24 + j), *(conv_out + (i+1)*24 + (j+1)));
      l++;
    }
    k++;
  }
}

float relu(float x) {
  if(x < 0) {
    return 0;
  } else {
    return x;
  }
}

float max(float a, float b, float c, float d) {
  float e = a > b ? a : b;
    float f = c > d ? c : d;
    return e > f ? e : f;
}


float fully_connected(const float *dense_weights, float *conv_layer_out, const float dense_bias_0, int count) {
  float out = 0;
  int i, j;
  for(i=0; i<32*12*12; i++){
    out += (*(dense_weights + i)) * (*(conv_layer_out + i));
  }
  out += dense_bias_0;
  return relu(out);
}

float output_layer(const float *dense_weights, float *fully_connected_out, const float dense_bias_1) {
  float out = 0;
  int i;
  for(i=0; i<30; i++){
    out += *(dense_weights + i) * *(fully_connected_out + i);
  }
  out += dense_bias_1;
  return out;
}
```

2. Myriad 2 Shave SIMD pseudo-code

main.c

```

#define N 784

// These variables are shared between all shaves
extern volatile u8 shared2[12];
extern volatile u8 shared1[12];
extern volatile float img[784];
extern volatile const float conv_biases[32];
extern volatile const float dense_bias_0[30];
extern volatile const float dense_bias_1[10];
extern volatile float conv_layer_out[32][12*12];
extern volatile const float kernels[32][25];
extern volatile const float dense_weights_0[30][32*12*12];
extern volatile const float dense_weights_1[10][30];
extern volatile float fully_connected_out[30];
extern volatile float output_layer_out[10];


//Function Declaration
void conv(const float *img, const float *kernels,
            float conv_bias, float *conv_layer_out);

float relu(float x);
float max(float a, float b, float c, float d);

float fully_connected(const float *dense_weights,
                        float *conv_layer_out, const float
                        dense_bias_0, int count);

float output_layer(const float *dense_weights,
                    float *conv_layer_out,
                     const float dense_bias_1);


void start(int* idShave, int *out)
{
  int i;
  int j=0;

    conv(img,kernels[*idShave],conv_biases[*idShave],
            conv_layer_out[*idShave]);

    conv(img,kernels[*idShave+12],conv_biases[*idShave+12],
            conv_layer_out[*idShave+12]);

```

```
   if(*idShave<8)
     conv(img,kernels[*idShave+24],conv_biases[*idShave+24],
             conv_layer_out[*idShave+24]);


shared1[*idShave]=1;

//Custom Barrier
//Only when every shave reads the shared arrray as 1
// when can progress
while(1)
{
    if((shared1[0]==1) && (shared1[1]==1) && (shared1[2]==1) &&
  (shared1[3]==1) && (shared1[4]==1)
   && (shared1[5]==1) && (shared1[6]==1) && (shared1[7]==1) && (
  shared1[8]==1) && (shared1[9]==1) && (shared1[10]==1) && (
  shared1[11]==1))
   break;

}

fully_connected_out[*idShave] =
fully_connected(&dense_weights_0[*idShave][0],
                &conv_layer_out[0][0], dense_bias_0[*idShave],
                *idShave);

fully_connected_out[*idShave+12] =
fully_connected(&dense_weights_0[*idShave+12][0],
                 &conv_layer_out[0][0],
                 dense_bias_0[*idShave+12],
                 *idShave+12);

if(*idShave<6)
  fully_connected_out[*idShave+24] =
  fully_connected(&dense_weights_0[*idShave+24][0],
                   &conv_layer_out[0][0],
                   dense_bias_0[*idShave+24],
                   *idShave+24);


shared2[*idShave]=1;

//Second Custom Barrier
while(1)
{
    if((shared2[0]==1) && (shared2[1]==1 ) && (shared2[2]==1) &&
   (shared2[3]==1) && (shared2[4]==1) && (shared2[5]==1) && (
  shared2[6]==1) && (shared2[7]==1) && (shared2[8]==1) && (
  shared2[9]==1) && (shared2[10]==1) && (shared2[11]==1))
    break;

```

```
  }
  if(*idShave<10) output_layer_out[*idShave] =
  output_layer(dense_weights_1[*idShave], fully_connected_out,
                dense_bias_1[*idShave]);


    SHAVE_HALT;

    return;
}
void conv(const float *img, const float *kernels, float conv_bias,
    float *conv_layer_out) {
  float conv_out[24*24];
  int i,j,k,l;
  float acc = 0.0;
  //convolution
  for(i=0; i<24; i++) {
    for(j=0; j<24; j++){
      acc = 0.0;
      for(k=0; k<5; k++){
        for(l=0; l<5; l++){
          acc += *(img + (i+k)*28+(j+l)) * *(kernels + k*5 + l);
        }
      }
    *(conv_out + i*24 + j) = relu(acc + conv_bias);
    }
  }
  //pooling
  k = 0;
  l = 0;
  for(i=0; i<24; i=i+2) {
    l = 0;
    for(j=0; j<24; j=j+2){
      *(conv_layer_out +k*12+l) = max(*(conv_out + i*24 + j), *(
    conv_out + i*24 + j+1),
                      *(conv_out + (i+1)*24 + j), *(conv_out + (i
    +1)*24 + (j+1)));
      l++;
    }
    k++;
  }
}


float fully_connected(const float *dense_weights, float *
    conv_layer_out, const float dense_bias_0, int count) {
  float out = 0;
  int i, j;
  for(i=0; i<32*12*12; i++){

    out += (*(dense_weights + i)) * (*(conv_layer_out + i));
```

```
  }
  out += dense_bias_0;
  return relu(out);
}

float output_layer(const float *dense_weights, float *
    fully_connected_out, const float dense_bias_1) {
  float out = 0;
  int i;
  for(i=0; i<30; i++){
    out += *(dense_weights + i) * *(fully_connected_out + i);
  }
  out += dense_bias_1;
  return out;
}


float relu(float x) {
  if(x < 0) {
    return 0;
  } else {
    return x;
  }
}

float max(float a, float b, float c, float d) {
  float e = a > b ? a : b;
    float f = c > d ? c : d;
    return e > f ? e : f;
}
```