A CNN FRAMEWORK ACCELERATED BY AN FPGA IN SYNTHESIZED C

A Thesis
by
Eleftheria Chatzidaki

In Partial Fulfillment of the
Requirements for the
Diploma of
Electrical and Computer Engineering

Thesis Committee:
Professor Apostolos Dollas (Supervisor)
Associate Professor Ioannis Papaefstathiou (AUTH)
Professor Michalis Zervakis

TECHNICAL UNIVERSITY OF CRETE
Chania, Greece

2019
Defended November 13

ABSTRACT

These days, Convolutional Neural Networks are popular for image classification and recognition. We prefer to utilize them because they achieve high accuracy by exploiting the inherent properties of images. A major disadvantage of CNN is that they perform many and complex calculations that cost a lot of time, energy and resources. The best solution that we can suggest is to take advantage of the properties of Field Programmable Gate Arrays. FPGAs are specialized in the acceleration of calculations and they consume less energy than Graphics Processing Units or Central Processing Units.

We introduce a framework written in C++ that can adopt FPGA kernels to accelerate calculations as matrix multiplications. We connected an available matrix multiplication with addition implementation to the framework and we tested it on a Trenz Platform. Apart from that, we implemented a fast and cache-aware framework applying OpenMP, GCC option flags, OpenMP environment variables and features from C++17. Our CNN framework is tested on a LeNet-5 architecture using the MNIST dataset containing L1, L2 Regularizations, Vanilla, Momentum, Momentum with Nesterov Updates, He-et-al weight initialization, Fisher-Yates shuffle, Stochastic Gradient Descent techniques that are all implemented from scratch. Furthermore, we implemented 3 ways of load MNIST dataset, as well as, naive, cache blocking, OpenMP and Hybrid cache blocking with OpenMP in matrix multiplication, transpose and copy algorithms, that we tested and investigated their behavior among the mini-batch sizes and the number of used threads.

Besides all the aforementioned that was made from scratch, we used the Xilinx Vivado SDK to make a bare-metal C++ project with the appropriate cache size linker script and adjusted the matrix multiplication with addition code to our framework. Afterwards, we programmed the Trenz Platform that contains an ARM CPU and an FPGA accelerator. As a result, we achieved a 4.3x-8.5x better performance using an FPGA to accelerate matrix multiplication with addition than using a naive or a cache blocking single thread implementations and in specific unfair(lack of multi-threading on Trenz) cases, depending on the mini-batch size of multi-threading OpenMP(up to 1.27x) or Hybrid algorithms(up to 2.27x) on a CPU.

ACKNOWLEDGEMENTS

There are many people, throughout the writing of this dissertation, that I want to thank for this thesis.

Firstly, I want to thank my professors Ioannis Papaefstathiou and Apostolos Dollas for the inspiration, as well as, their support and their time, until the completion of my thesis. I would also like to thank my professor Michalis Zervakis, who introduced me to some prior aspects of computer vision at his courses and serving as my committee member and Dionisios Pnevmatikatos who gave me access to the MHL Lab and its equipment (Trenz Platform, Kronos server).

I would like to acknowledge Antonios Nikitakis for his guidance for the thesis's subject and the enlightening meetings that helped me to understand better my thesis even at the very early stages. Moreover, I want to thank Vasileios Amourgianos Lorentzos for his contribution in this thesis that inspired me further, his time and patient to answer my repeated questions for his implementation that I adjusted.

Furthermore, I want to acknowledge some of my friends and MHL lab members. Especially, I want to thank Periklis Chrysogelos, for his time, his bibliography suggestions as answers to my questions and the enjoyable and enlightening discussions. Also, I want to thank Pavlos Malakonakis and Andreas Brokalakis for their support, their time to discuss with me and to answer my questions and the inspiration that they gave to me through the last year. As well as, I want to thank Athanasios Vasileios Grammatopoulos for his time at the very early stages of my thesis, who guided me to find answers about advanced coding issues or requirements that I had faced. I also want to thank Nikolaos Tampouratzis who answered my questions and informed me about his work that it will be considered in my future work.

Last but not least, I want to thank my family and friends for their unconditionally support throughout my life.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS AND ABBREVIATIONS

CNN Convolutional Neural Network

FPGA Field Programmable Bate Array

CPU Central Processing Unit

GPU Graphics Processing Unit

TPU Tensor Processing Unit

FC Fully Connected

CONV Convolutional

DNN Deep Neural Network

SGD Stochastic Gradient Descent

2D 2 Dimensions

3D 3 Dimensions

4D 4 Dimensions

ReLU Rectified Linear Unit

CNTK Cognitive Toolkit

RNN Recurrent Neural Network

ML Machine Learning

AI Artificial Intelligence

CUDA Compute Unified Device Architecture

AWS Amazon Web Services

SDK Software Development Kit

MNIST Modified National Institute of Standards and Technology database

IP Intellectual Property

HLS High Level Synthesis

MLP Multi-Layer Perceptron

RGB Red-Green-Blue

ILSVRC ImageNet Large Scale Visual Recognition Competition

VHDL VHSIC-HDL

VHSIC Very High Speed Integrated Circuit

VHLS Vivado High-Level Synthesis

HDL Hardware Description Language

RTL Register Transfer Level

ASIC Application-Specific Integrated Circuit

IC Integrated Circuit

SRAM Static Random-Access Memory

GNU GNU's Not Unix

OpenMP Open Multi-Processing

FNN Feed-forward Neural Network

FATFS File Allocation Table File System

MSB Most Significant Bit

LSB Least Significant Bit

MLE Maximum Likelihood Estimation

CM Classical Momentum

NAG Nesterovs Accelerated Gradient

NAN Not A Number

MPSoC Multi-Processor System-On-Chip

SoC System-On-Chip

APU Application Processing Unit

RPU Real-time Processing Unit

PL Programmable Logic

PS Processing System

BRAM Block Random-Access Memory

LUT Look-Up Table

HP High-Performance

DSP Digital Signal Processing

FSBL First Stage Boot Loader

ARM Acorn RISC Machines

BSP Board Support Package

DDR Double Data Rate

SD Secure Digital

Batchnorm Batch normalization

trans transpose

matmul matrix multiplication

QPI Quick-Path Interconnect

IPC Instructions Per Cycle

TDP Thermal Design Power

AVX Advanced Vector Extensions

(G)FLOPS (Giga) Floating Point Operations Per Second

PC Power Consumption

Chapter 1

Introduction

1.1 Overview

The goal of this thesis was to implement a framework to train Neural Networks using the advantages of an FPGA. The framework is based on a Convolutional Neural Network, which is a popular and successful deep neural network model. Deep Neural Networks require computational power, that technologies as GPU, TPU or FPGA can handle faster than general-purpose processors. FPGAs are designed to reprogram and to consume low power per instruction. For the aforementioned reasons, we focus on FPGA technology for the acceleration of the computations.

The purposed CNN is designed in a hybrid CPU/FPGA system, where CPU acts as the controller and FPGA perform the computations. At the moment, we want to adjust a matrix multiplication with addition computation function in the Fully Connected layer and as future work to expand the FPGA usage and implement and adjust computation functions that slow down the training process. The kernels that we have programmed use the generated bitfiles from Vivado Xilinx Tools. Apart from that, we make research in software to investigate the behavior of matrix multiplication, transpose and copy functions in more than one algorithm implementations that help us to improve our framework.

1.2 Related Work

There are many frameworks that built-in for different purposes and problems. The framework that we guided and tried to translate in C++ was an implementation from C231n course - Convolutional Neural Networks for Visual Recognition from Stanford [2] and Paras Dahal's Deepnet [8] that are both written in Python and are used to introduce someone into the Neural Networks.

Another interesting implementation of a CNN framework that has the same logic and we take into consideration while we build ours is Darknet, written in C and Cuda for CPU and GPU computation support [9].

Some other significant and widely used frameworks that we can refer as related work, considering that a lot of our work focus on CNN in general, are:

- **TensorFlow**
  The most popular framework at the moment is the TensorFlow . It is adopted by many researchers and large companies that use AI in their services due to flexible architecture and the computational graph abstraction usage. It is available in C++, Python, and R and it has 2 helpful and widely used tools, the TensorBoard and TensorFlow Serving [10].

  TensorBoard is the interface that visualizes effectively the network to optimize, debug and understand the model. TensorFlow Serving is claimed that is a high-performance, flexible serving system for machine learning models and it is designed for production environments [11, 12].

  The major disadvantages of TensorFlow are that it is not completely open-source, the Python is not a high-level language hence it is slow and the lack of many pre-trained models [11, 12].

- **The Microsoft Cognitive Toolkit/CNTK**
  It is another open-source deep learning framework to train deep learning models created by Microsoft . It is similar to Caffe framework and its interface can support Python, C++, C#/.NET, and Java. Microsoft Cognitive Toolkit supports CNN and RNN to solve efficiently handwriting, speech and image recognition problems. It is flexible and it supports distributed training. Therefore, it lacks visualizations and it does not support ARM architecture that mobile devices mainly supports [11, 13, 12].

- **Caffe**
  Caffe is a powerful deep learning framework which is developed by Berkeley AI Research (BAIR) and by community contributors . It is fast, efficient and supports C, C++, Python, MATLAB. It is claimed that it is easy to build a CNN for image classification and generally for deep learning research. Caffe works in CPU and GPU and allows the training of a model without writing code. Therefore, it is not great for new architectures and RNNs [13].

- **Torch/PyTorch**
  Torch is a framework for deep learning based on Lua which is used by many industry giants and researchers . It uses C/C++ libraries and CUDA. Moreover,

it is an open-source, flexible, high-level speed and efficiency library that has many available pre-trained models. Although its documentation is unclear, Lua is not popular nowadays and it lacks plug-and-play code for immediate use [13].

PyTorch is an open-source and also supported from Facebook, rapidly grown framework . Furthermore, PyTorch is a port to Torch framework, therefore it runs on Python which is one of the most popular, preferred and easy to use language today. It can execute high complexity computations of tensors. The PyTorch framework has a simpler modeling process in comparison to Torch framework [12].

- **Theano**
  Theano is the oldest deep learning framework in Python and it was developed at the University of Montreal in 2007 . It is claimed a powerful library that allows for numerical operations involving multi-dimensional arrays with a high level of efficiency. The intensive computations performed properly optimized in GPU instead of CPU with high efficiency in its operations and numerical tasks. Therefore, it is a bit buggy in Amazon Web Services(AWS) and in case we want to gain a high level of abstraction, it needs to be used with other libraries [13].

- **Keras**
  Keras is a user-friendly, easily extensible framework. It runs seamlessly on both CPU and GPU while it works with Theano and TensorFlow. A major disadvantage is that it can not be efficiently used as an independent framework [12]. Keras supports Python language and both CNN and RNN. It is a nice start for the users that have some experience in Python and want to delve further into deep neural networks [13].

There are many more frameworks, but we focus on the most significant to get a better idea on frameworks, their purpose, general functionalities and why they are preferred as they are the most popular nowadays.

FPGA accelerated Neural Networks

We found some interesting frameworks that are accelerated by FPGAs that are worthy to mention:

- **TABLA**
  TABLA is a template-based framework that focuses on training acceleration of statistical machine learning . It automatically generates accelerators in Verilog for stochastic gradient descent [14].

- **F-CNN**
  The F-CNN is an FPGA-based Framework for training CNNs, that has modules implemented for acceleration using FPGAs. It is claimed that is energy efficient than GPU and has higher performance than a CPU, both implemented on Caffe[15].

- **FINN**
  FINN is an experimental framework from Xilinx Research Labs that use a flexible heterogeneous streaming architecture to build fast and flexible BNN inference FPGA accelerators. The generated accelerators can are great for embedded applications such as autonomous driving because they have submicrosecond latency while executing millions of classifications per second. It is claimed to has the fastest classification rates [16].

- **FPDeep**
  FPDeep is a framework that trains DNNs using layer parallelism and a hybrid of model, in order to configure distributed reconfigurable clusters . Its policy is to balance the workload among FPGAs, achieving better utilization. The whole system is fine-grained pipelined and reduces the storage request. It is claimed to have good scalability to numerous FPGAs, limited only by FPGA's bandwidth connection to other FPGAs [17].

There are many more and we should investigate them further to improve our framework in the future, using their techniques.

1.3 Contributions and Outline

Initially, this thesis aimed to understand, explore, implement and optimize a basic CNN framework that can easily adjust IP cores that accelerate the computation functions on an FPGA. FPGAs are known for fast computation operations with low power consumption, hence the neural networks can take advantage of them at their heavy, multiple and complex computational functions within the layers, Many vendors and researchers have implemented, announced frameworks, libraries, systems for deep learning. Therefore, we wanted to understand in-depth and evaluate or even optimize a CNN framework that could easily be imported in Vivado SDK to accelerate via IP cores the complex calculation and we could use it as a basic

framework to focus further on FPGAs and IP cores or separately in framework than to implement every time the whole system.

In **Chapter 2** we provide relevant theoretical and basic mathematical background for neural networks, backpropagation, optimization, neural networks, common layers and we refereed to often used architectural design patterns for processing images and text, especially for convolutional and recurrent neural networks. Furthermore, we make a generic and basic introduction of the High-Level Synthesis and FPGAs technology.

In **Chapter 3**, we develop a CNN model that can classify fast grey-scale images from MNIST dataset. We set some goals and we perform many optimizations and explore methods that we compare and finally make some choices that lead to a specific CNN framework that we use, although it is configurable if we want to make changes. Moreover, we used some profiling tools to determine possible errors or weaknesses of our framework that we deal or we choose to examine further in the future.

In **Chapter 4**, we introduce in general the existing IP core of matrix multiplication and addition computation function that it was available and our attempt to adjust it in our system. We describe the changes that we should do to compile and run in the platform and the process that we follow to achieve it.

In **Chapter 5**, we sum up the results of the software and the hardware implementations and compare them.
Finally, in **Chapter 6**, we conclude to the outcome of this thesis, identify the remaining challenges and discuss the path forward.

Chapter 2

Background and Concepts

This chapter provides the necessary technical background details on neural networks, more specifically in Convolutional Neural Networks. Additionally, the second section makes an introduction of Field-Programmable Gate Arrays and High-Level Synthesis.

2.1 Convolutional Neural Networks

This section will deal with the basic theory of Neural Networks in general and Convolutional Neural Networks. There is a hierarchy from abstract concepts like a neuron to more specific like the CNN layers that are developed in our architecture.

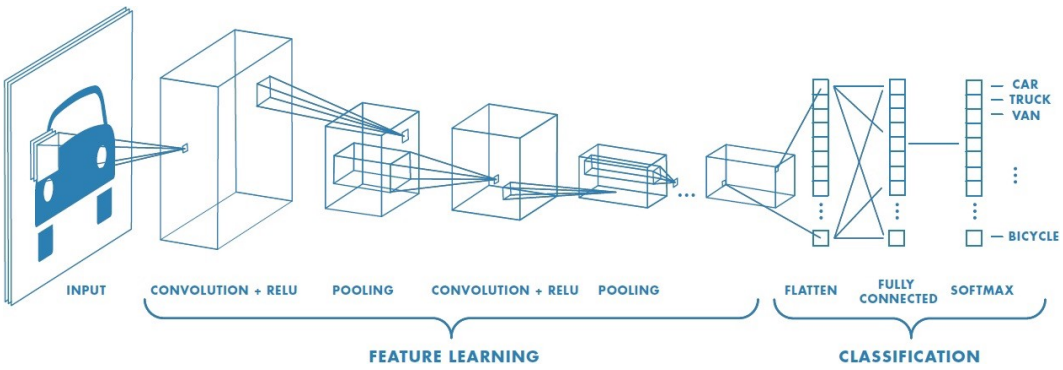


Figure 2.1: Example of a CNN[1].

2.1.1 Introduction to Neural Networks

Neural networks are popular nowadays. There is an enormous amount of datasets and information that need to be classified. In this case, neural networks are trying to do this task, operating as human's neurons, and make decisions with good accuracy.

Neuron

Neurons are inspired by the biological neurons in a human nervous system. The way of their connection and their anatomy tried to match with mathematical formulas. Hence, we assume that a neuron is the basic computational unit that is

connected with the others via synapses. Signal input is transferred among the dendrites of neurons and the single output fires/exits from axon which is connected with dendrites via synapses [2].

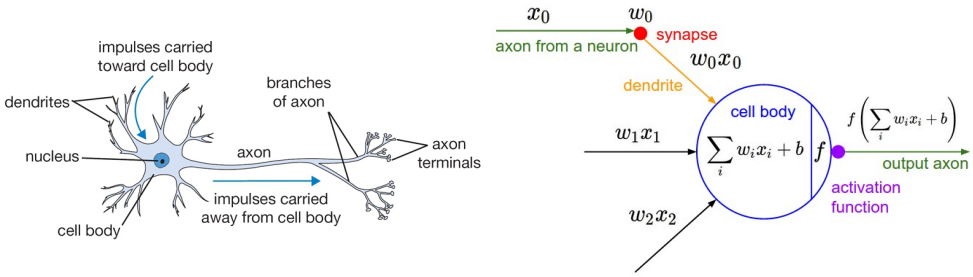


Figure 2.2: A basic scheme of a neuron and it's general/typical mathematical match[2].

A mathematical neuron is a computational node that has one or more inputs x and a single output f*(x)(see Figure 2.1). Inside neuron, we do the computations depending on the layer's functionality and learn new features keeping them as weights (memory) to identify future inputs.

Feedforward Neural Network

Feedforward neural networks or Deep feedforward networks or multi-layer perceptrons(MLPs) are artificial neural network and that is claimed as the most essential parts of deep learning models. The goal of a feedforward network is to learn features and find an approximate function for the whole network that can identify specific information like numbers, objects in an image, etc. Their name suggests the flow of information through the function of the neuron or layer of neurons that it is inserted to the output. There is not feedback process from the neuron to itself, with connections from the output back to input forming a cycle, like in recurrent neural networks. Hence, they composing many different functions describing an acyclic graph combining layers connected in a chain structure. Each perceptron in a layer is connected to every perceptron on the next layer and there is no connection among perceptrons in the same layer. The overall length of the layer's chain depends on the depth of the model, hence they are named "deep" feedforward networks. The first layer is often named as the input layer, the last output layer and the layers between them are commonly referred to as hidden layers because they do not connect with the external world[18].

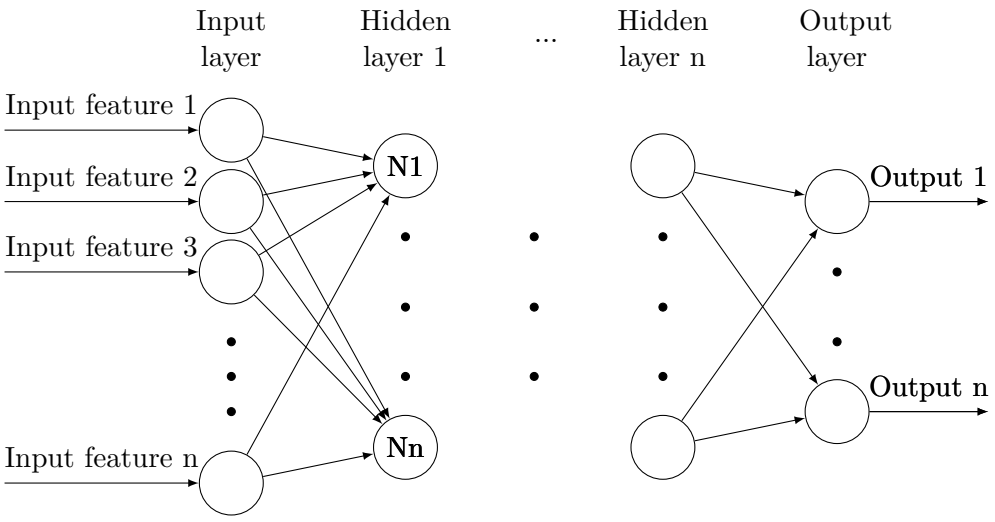


Figure 2.3: Deep Neural Network structure.

A single layer of nodes can consist of a single-layer perceptron network which is the simplest neural network, nevertheless, we focus on deep neural networks that have much more than one layer of output nodes.

Backpropagation

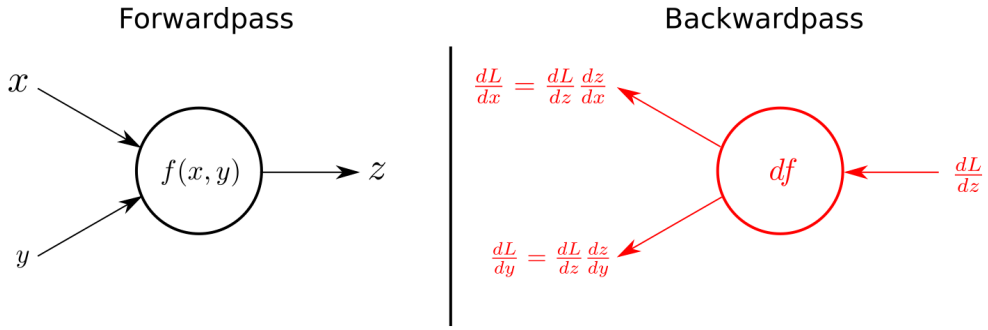


Figure 2.4: The use of chain rule for the backward pass in computational graphs.[3]

The output of the feedforward network becomes the input of the backpropagation process. This process is developed to optimize the weights and bias of a neural network by updating them to a value closer to the target output. Hence, the backpropagation algorithm is claimed to be an inexpensive procedure that can minimize the error of each layer and the whole network's error. The chain rule is also applied in backpropagation, but it is in reversed flow. Furthermore, the layers that are affected within by performing computations are the Convolutional and Fully Connected as they update weight, bias arrays calculate it with a λ hyperparameter.

The changes will be explained later by calculating the derivatives of these layers. The loss of the epoch or mini-batch can be calculated by the absolute difference of forward and backward function during training. The backpropagation algorithm is an inexpensive procedure to eliminate cost function[18, 19].

2.1.2 Introduction to Convolutional Neural Networks

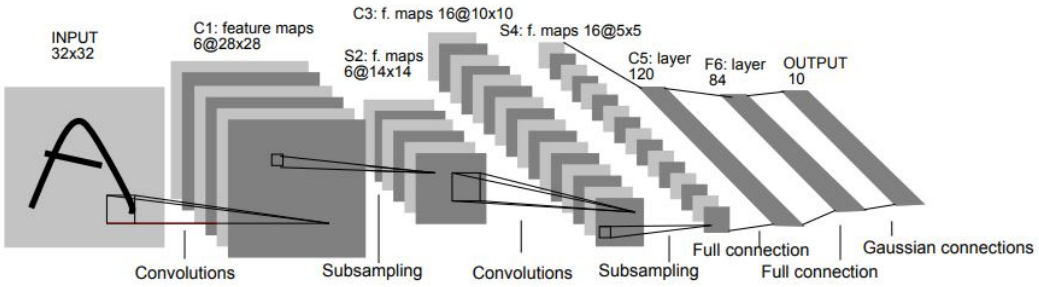


Figure 2.5: LeNet-5 Architecture as a CNN example [4].

Convolutional Neural Networks or CNN or ConvNets are neural network architectures specialized to handle data with some spatial topology, commonly applied to analyze images, videos, text, etc[20]. Inputs of data are described as 3D arrays as (depth,height,width) of image. We will focus on images where depth consists of the number of colors that an image have. More specifically, 1 refers to grayscale and 3 to color channels RGB (Red-Green-Blue). Every image has 28x28 pixels height and width, the so-called feature maps[20].

2.1.3 Layers

As it is aforementioned, the neurons are grouped in structures and referred to as layers. Moreover, they categorized as input, output or hidden layers while the latter are able to interpret non-linearities in the input data. [20]

Our framework, at this moment, has implemented the following layers that will be analyzed afterward:

- Convolutional
- Maxpool
- Batch Normalization
- Fully Connected
- Flatten
- Dropout
- Activation: ReLU
- Activation: Tanh
- Activation: Sigmoid

Convolutional Layer

The Convolutional layer is used for the most demanding computational work of a Convolutional Neural Network. It's the special layer of Convolutional Neural Networks. The Convolutional layer use im2col(image to columns) and col2im(columns to image) methods and slide 2D kernels/filters/windows over width and height of the padded image to get a 2D activation map as it is illustrated:

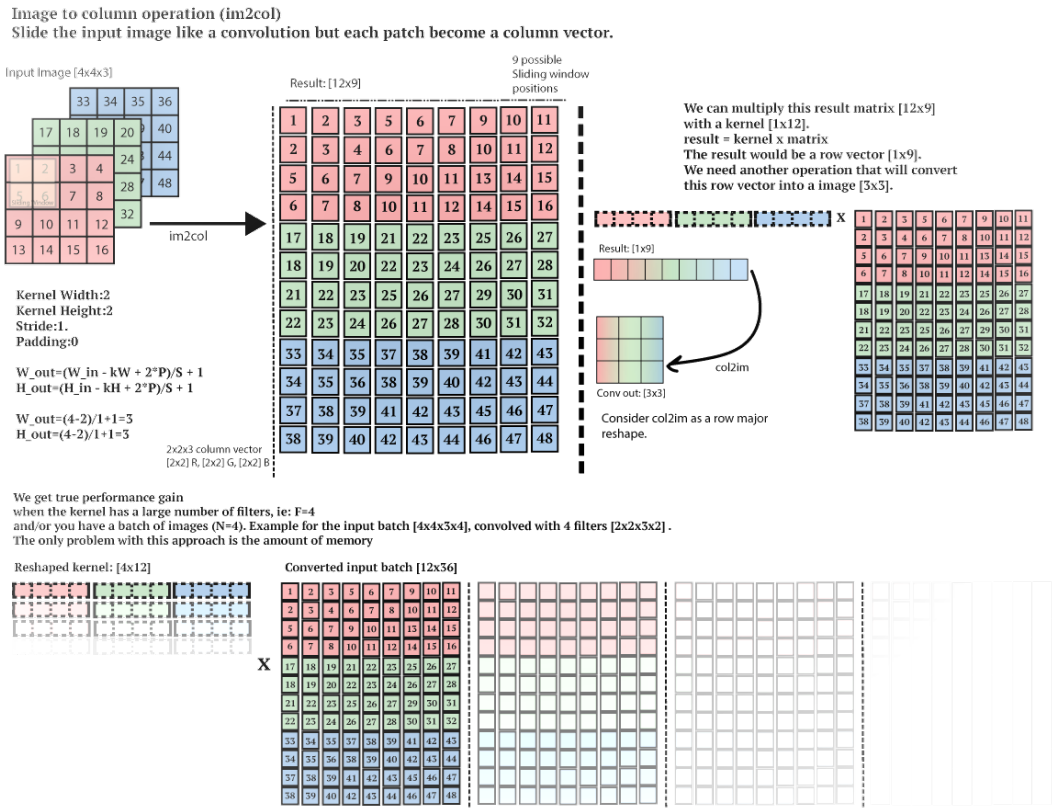


Figure 2.6: The input image takes the height and width of the filter to become a column vector. On the other hand, the col2im function takes the resulted matrix with the width and height of the filter to become a row vector and afterward, we convert it back to image. It's a confusing process of multiple convert ions etc but we can gain better performance than an only for-loop convolution process. The disadvantage of im2col and col2im implementation is the amount of memory that we should have and use to perform the necessary calculations and conversions/transformations [21].

At forward pass, we convolve at the 2D dimensions(height x width) of the input image creating an activation map that detects features and computes dot products between every input's position and the entries(learned variables) of the filter. Each region of the input image will connect to only one neuron as a hyperparameter and

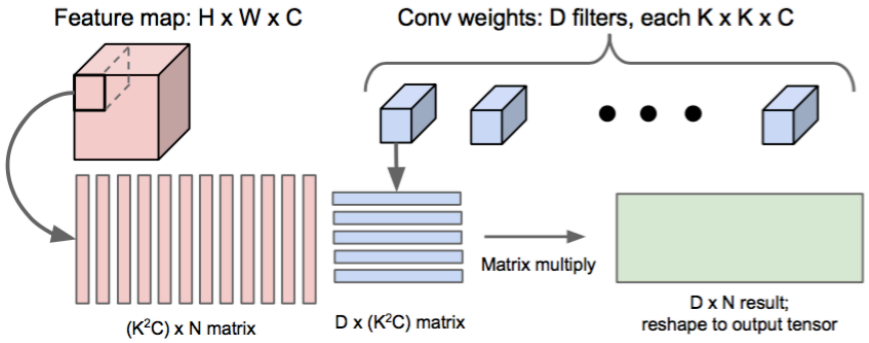


Figure 2.7: The feature map reshape from 3D to 1D for every picture like the Convolutional weights to perform matrix multiplication for the forwarding [5].

it is called the filter size or receptive field of the neuron.

$$Z_{ij}^{l} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} a_{(i+a)(j+b)}^{l-1} \tag{2.1}$$

The Convolutional layer's input is an image of $D(depth)xW(width)xH(height)$ (3D dimension array), in which the depth is 1 for grayscale images like the MNIST dataset has and we use it or 3(RGB) for colorful images. Moreover, we need three hyperparameters to get an optimized output volume. These are the number of the filters that will become the depth of the output volume and they are trained to seek for something different in the input, although each time a set of neurons look to the same region of the input as a depth column (fibre). Furthermore, the other two are the stride that specifies the slide of the filter of the window/filters with spatial extent (F) and the zero-padding that fulfill the input volume with zeros depending the divisions of sliding filters to fit while sliding. There is a formula for it: $(W - F + 2xP)/S + 1$ and $(H - F + 2xP)/S + 1$ [2].

Parameter sharing scheme is useful in convolutional layer to control the number of parameters assuming that region features are useful to compute at more than one spatial region. Hence, the restriction of each depth slice(activation map) within the output volume to the same weight and bias will lead us to a reduction of the number of parameters.

The backpropagation is affected by updating only a single set of weights instead of every single one because each neuron in the output volume represents the overall gradient of which can be totaled across the depth [22].

Following the theory:

$$\frac{\partial C}{\partial W_{ab}^{l}} = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial C}{\partial Z_{ij^{l}}} \times a_{(i+a)(j+b)}^{l-1} \tag{2.2}$$

$$\frac{\partial C}{\partial b^{l}} = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial C}{\partial Z_{ij}^{l}} \tag{2.3}$$

$$\frac{\partial C}{\partial a_{ij}^{l-1}} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \frac{\partial C}{\partial Z_{(i-a)(j-b)}} \times W_{ab} \tag{2.4}$$

Maxpool Layer

Maxpool layer is often used to reduce the computations and the number of parameters that the Convolutional layer produces. It is periodically used after a couple of Convolutional layers to cut down the spatial size and control over-fitting. As its name suggests it operates Max to every depth slice of input volume independently and down-samples it. While the output height ($H_{out} = (H_{in} - F)/S + 1$) and width ($W_{out} = (W_{in} - F)/S + 1$)are depend on the stride (S) and spatial extent (F) parameters, the depth remain the same.

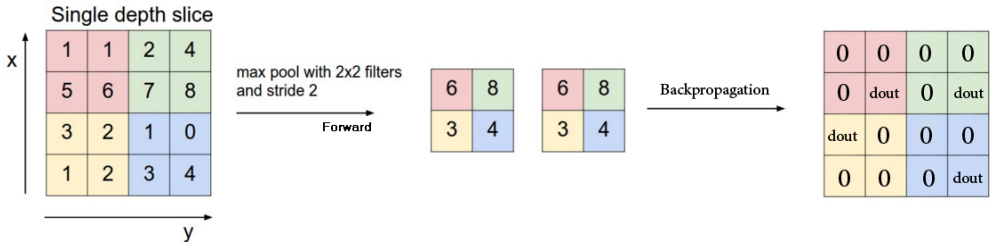


Figure 2.8: Left: Maxpool forward stage sliding window chose max values inside. Right: The backward stage returns the max value in the position of the window that it was stored from forwarding while the rest of them remain zero [23].

Nevertheless, it is possible future architectures to contain few or any of pooling layers, because they suggest worse generative models in training than the models without pooling [2].

Fully-Connected Layer

Fully connected layer have full connections of its inputs, as they perform matrix multiplication with a weight matrix and add a bias offset. Any change of its input

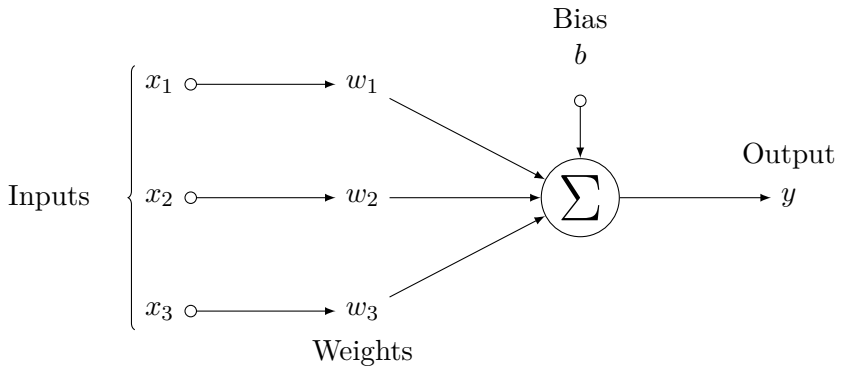


Figure 2.9: Matrix multiplication in Fully connected layerś forward process for one neuron.

values affect the output values. The inputs are the 3D gray images flatten in 1D. The result of Flatten layer is a 2D array because we process batches and/or mini-batches of images, instead of single images. It is defined by

$$Y_{ij} = X_{ik} * W_{kj} + b_i \tag{2.5}$$

$$\begin{bmatrix} X_{11} & \cdots & X_{1j} \\ \vdots & \ddots & \vdots \\ X_{i1} & .\cdots & X_{ij} \end{bmatrix} * \begin{bmatrix} W_{11} & \cdots & W_{1j} \\ \vdots & \ddots & \vdots \\ W_{i1} & . & W_{ij} \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_i \end{bmatrix} = \begin{bmatrix} Y_{11} & \cdots & Y_{1j} \\ \vdots & \ddots & \vdots \\ Y_{i1} & \cdots & Y_{ij} \end{bmatrix} \tag{2.6}$$

where W is the weight array, X is the input array of features, Y is the output array, b is the bias vector and i,j,k the dimensions of the arrays/vector.

The aforementioned (Convolutional) layer differs from Fully Connected layer, as its neurons share parameters and they are locally connected in the input. Hence, we assume that it is possible to convert a Fully Connected layer to a Convolutional layer and conversely, as their functional form is exactly the same [2].

Fully Connected layer is used in the classification stage. The weight matrix is initialized following the He et al.[24] technique that will be explained in detail later. Biaś offset is set to zero, according to the Python model that we are consulting.

Flatten Layer

Flatten layer is the connection between the training procedure and the classification of our Convolution Neural Network. The pooled featured map flatten in a column.

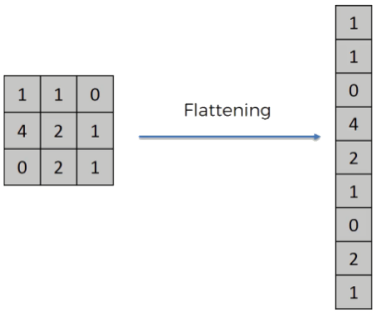


Figure 2.10: Flatten layer reshape 2D images to 1D -vector- arrays before Fully-Connected layer [25].

The output of the Flatten layer is the input of the Fully-Connected layer, afterward. Following some convolutional layers, it is necessary to flatten the array to use the fully connected layer for their connection [26].

Batch Normalization Layer

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.7}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2.8}$$

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.9}$$

$$y_i = \gamma x_i + \beta \tag{2.10}$$

A batch normalization layer normalizes each input channel to have 0-mean and/or unit (1) variance across a mini-batch. Hence, they reduce the sensitivity to network initialization and speed up the training procedure between Convolutional and activation (ReLU) layers. Each input channel subtracts the mini-batch mean and after it divides it by the mini-batch standard deviation, while the layer shifts the input data by a learnable offset β and scales it by a learnable scale factor γ [2, 27].

The equations for backpropagationś γ and β [8]:

$$\frac{\partial C}{\partial \gamma \hat{x_i}} = \frac{\partial C}{\partial y_i} \tag{2.11}$$

The output volume of backpropagation is calculated by[8]:

$$\frac{\partial C}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial C}{\partial y_i} \tag{2.12}$$

$$\frac{\partial C}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial C}{\partial y_i} \times \hat{x_i} \tag{2.13}$$

$$\frac{\partial C}{\partial x_i} = \frac{\partial C}{\partial \hat{x_i}} \times \frac{1}{\sqrt{\sigma_B^2 + \beta}} + \frac{\partial C}{\partial \mu_B} \times \frac{1}{m} + \frac{\partial C}{\partial \sigma_B^2} \times \frac{2}{m}(x_i - \mu_B) \tag{2.14}$$

Dropout Layer

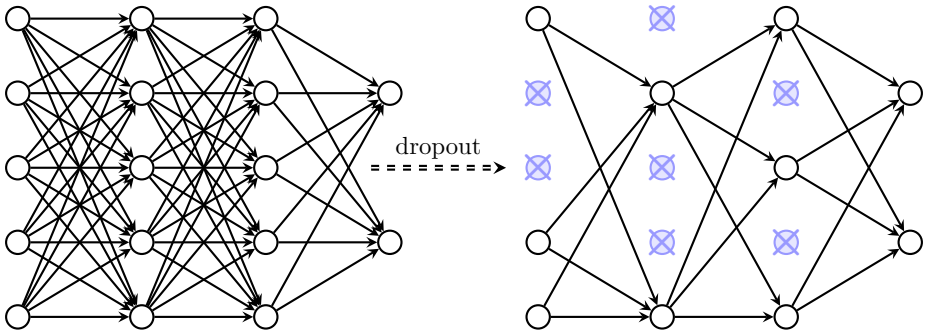


Figure 2.11: Dropout example with pruned nodes.

Dropout is a recent effective method to prevent a neural network from overfitting and provides an efficient way of approximately combining exponentially many different neural network architectures. While the network is training, it keeps a large number of parameters and complex that restrict the network to generalize to new data. During the forward propagation, the input amount or neurons n is multiplied by probability p. The p random fraction will decide the percentage of neurons that will be ignored by the network. Therefore, Dropout force the network to learn the most robust features that are related to other different subsets of other neurons and are less susceptible to noise. Although training time is reduced for each epoch, the network will need an almost double number of iterations until convergence [28]. The default probability is $p = 0.5$.

$$b = mask/(1-p) \tag{2.15}$$

While, the derivative for the back-propagation is

$$\partial b = \frac{mask}{1-p} \tag{2.16}$$

Dropout belongs to regularization methods of a Neural Network among the L1 (Laplacian) and L2 (Gaussian) regularization techniques. Furthermore, Dropout is preferred to be applied after a Fully-Connected, because a Convolutional layer has significant resistant to overfitting due to shared weights of filters[8].

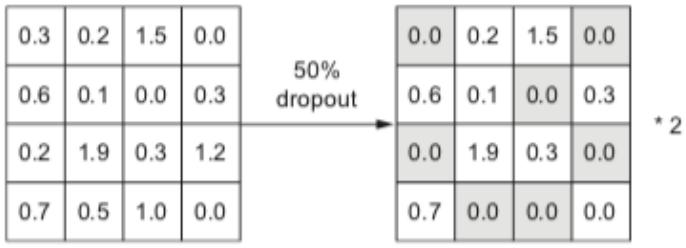


Figure 2.12: Dropout with 0.5(probability) cut out factor keep value over 0.5 and the rest become zero(pruned)[29].

Activation Layers

There are a plethora of activation functions, which introduce non-linearity into the network. In our implementation, we handle these functions as layers. Each layer of a neural network performs a linear transformation of the input. However, activation layers perform a scalar, non-linear, fixed mathematical operation in every single number of its input. The output of the activation layers helps the network to find and extract complex features to pass them in the next input[30].

As previously mentioned there are several activation functions, therefore, we currently have implemented only ReLU, Sigmoid, and Tanh that are widely known and used. Further implementation of activation layers/functions is part of our future work.

Activation Layer: ReLU

Rectified Linear Unit (ReLU) and many alternatives are used successfully in the last few years in deep neural networks. The ReLU activation function is the most used and it is used in almost all the CNN or deep learning[31]. It was introduced by Hahnloser et al. in 2000 [32] and it is defined by the function $f(x) = \max(0, x)$ or

$$ReLU(x) = \begin{cases} x, & if \text{ x} \geq 0 \\ 0, & if \text{ x} < 0 \end{cases} \tag{2.17}$$

ReLU behaves as a ramp function[33]. Input positive values $x \geq 0$ yield a linear output of the input while negative ones $x < 0$ forces them to zero, providing a good performance[34].

As the activation is thresholded to zero and sparsely activated, it is considered to have a cheap operation compared to Tanh or Sigmoid activation functions. Furthermore, ReLU is claimed to accelerate the convergence of Stochastic Gradient Descent-SGD better than the Sigmoid and Tanh functions [35]. Therefore, in case of many ReLU units become or are zero, then the gradient becomes zero which is known as a "dead" unit and it is an irreversible condition from that point on while learning rate is too high[2].

The backward function of ReLU computes the value $z = y * f'(x)$, where y is the output of the previous layer's backward function and [36]

$$f'(x) = ReLU'(x) = \begin{cases} 1, & if \text{ x} > 0 \\ 0, & if \text{ x} \leq 0 \ . \end{cases} \tag{2.18}$$

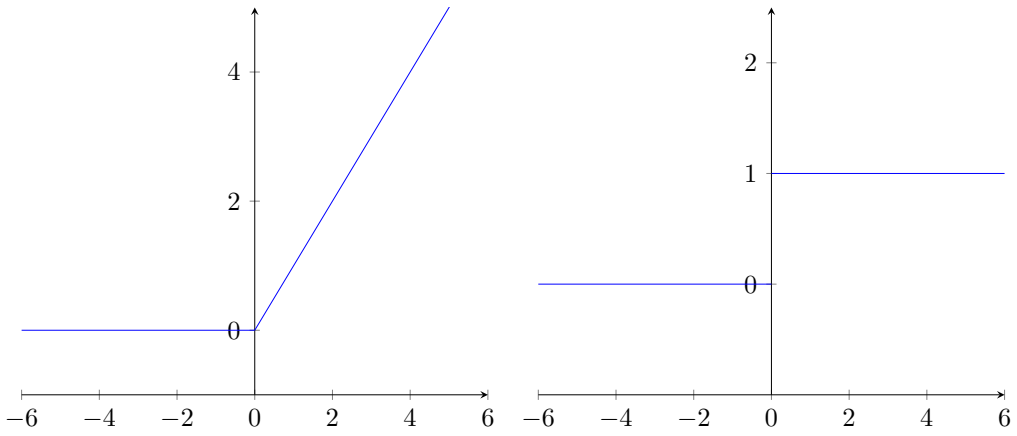


Figure 2.13: Left: Plot for ReLU Forward. Right: Plot for ReLU Backward

Activation Layer: Sigmoid

The sigmoid or logistic function or sigmoidal curve named by its characteristic S-shape. It is a special case of logistic function and its non-linearity function is defined by

$$Sigmoid(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{2.19}$$

The sigmoid function exists into a range between 0 and 1. More specifically, large positive numbers become 0, while large positives become 1. It is used for models that have to predict the output and it is used for multiclass classification as it is a more generalized logistic activation function. Moreover, it is differentiable and monotonic whereas it's derivative is not (monotonic).

Major drawbacks are that sigmoid outputs are not zero-centered that can cause undesirable zig-zag movement in weight gradient updates. Therefore, it could be eliminated or reduced in the final update of the batch, because of the gradient's addition across it. As well as, the most significant problem is that sigmoid kills gradients and cause saturation. In case of, 0 or 1 the gradient is almost zero, which means that it deactivates the local gradient while the too large initial weights will saturate and the network will almost stop learn. For the aforementioned reasons, sigmoid is rarely used [2].

The derivative for the back-propagation of a sigmoid function is

$$\frac{d}{dx}Sigmoid(x) = \frac{1}{1+\mathrm{e}^{-x}} \cdot (1 - \frac{1}{1+\mathrm{e}^{-x}}) \tag{2.20}$$

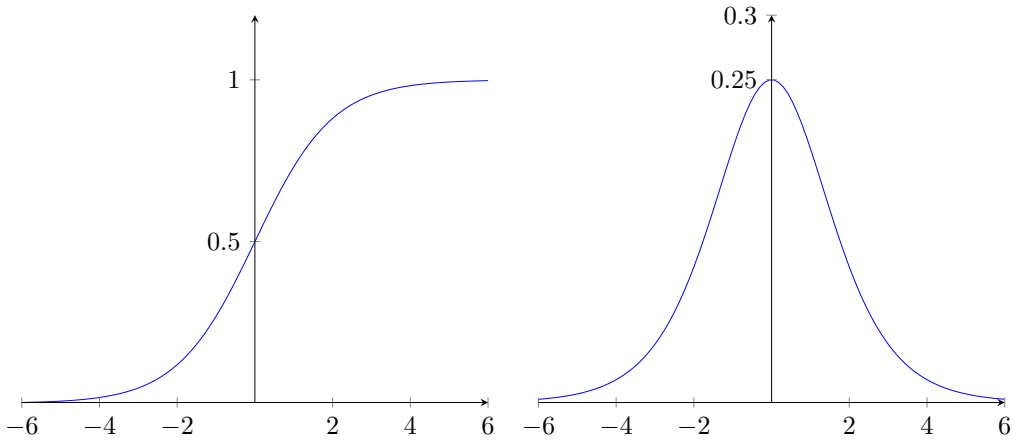


Figure 2.14: Left: Plot for Sigmoid Forward. Right: Plot for Sigmoid Backward

Activation Layer: Tanh

The Tanh non-linearity is another S-shaped function with range [-1, 1]. Furthermore, this function is differentiable and monotonic, whereas its derivative is not monotonic. Tanh function's output is zero-centered unlike a sigmoid, but their activations both saturate which means that suffers from vanishing gradient problem. As well as, both Tanh and logistic sigmoid activation functions are commonly used in feed-forward networks. Tanh is more preferable than sigmoid, because is stepper and referred to as the scaled version of the sigmoid [31]. However, the choice between sigmoid or Tanh is depending on the requirement of gradient strength [2]

$$Tanh(x) = \frac{2}{1 + \mathrm{e}^{-2x}} - 1 \tag{2.21}$$

The derivative for the back-propagation of a Tanh function is

$$\frac{d}{dx}Tanh(x) = 1 - (\frac{2}{1 + \mathrm{e}^{-2x}} - 1)^2 \tag{2.22}$$

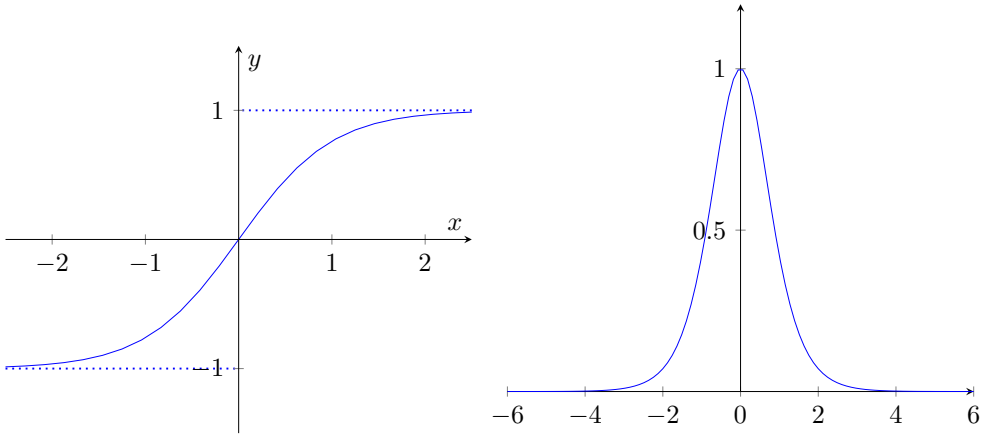


Figure 2.15: Left: Plot for Tanh Forward. Right: Plot for Tanh Backward

2.1.4 Convolutional Neural Network Architectures

Many CNN architectures were introduced at ImageNet Large Scale Visual Recognition Competition and they are named [37]. The most popular are:

- **LeNet-5**
  LeNet-5 first introduced and developed by Yann LeCun in 1998. LeNet-5 is designed for handwritten and machine-printed character recognition [38]. We also have implemented a slightly different version of it as it is a basic

architecture to test our framework. In theory, LeNet-5 has only 7 hidden layers as it is illustrated in Figure 2.5 and described in detail in the LeCun's paper [38]. The basic differences of our implementation are the max-pool layer instead of average-pool and an extra layer to flatten 4D arrays to 2D.

- **AlexNet**
  It was introduced in ImageNet ILSVRC challenge in 2012 by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton[39]. AlexNet is similar to LeNet architecture, but it is deeper and it uses ReLU activation, more filters per layer, dropout to prevent overfitting and Maxpool layers instead of Average-Pool.

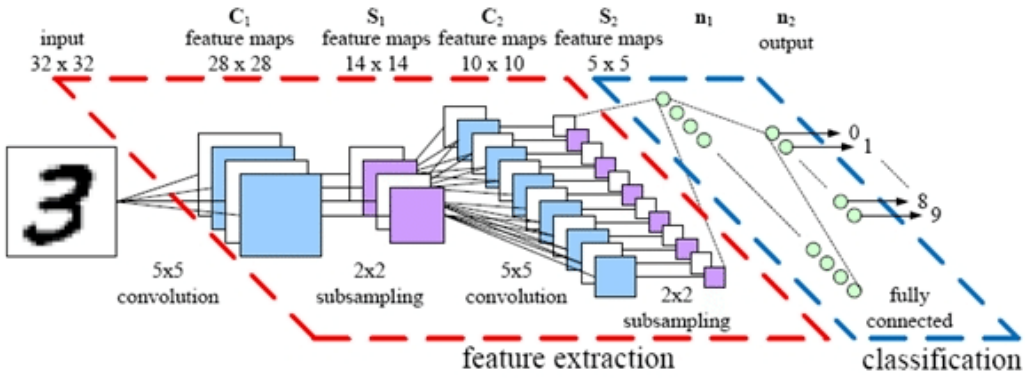


Figure 2.16: AlexNet CNN architecture layers [40].

- **VGG-16**

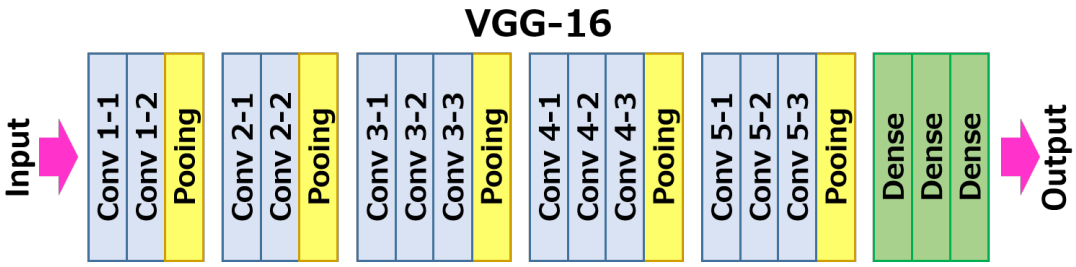


Figure 2.17: VGG-16 CNN architecture layers [41] .

VGG-16 is a simpler architecture model than the previous as it uses a few hyperparameters. It was introduced in ILSVRC 2014, where it got the second place (1st was GoogleNet -2014-) by Karen Simonyan and Andrew Zisserman. The most important outcome from this paper was the claim that the depth of

the network is a critical component to get a good performance. Therefore, it is expensive as it uses a lot of parameters and memory [42, 2].

- **GoogleNet**
  It was the winner of ILSVRC 2014 implemented by Szegedy et al. from Google. Inception Module was its main contribution. This module helped to dramatically reduce the number of parameters in the network if we compare it with AlexNet. As well as, the Fully Connected layers at the top of the ConvNet replaced by Average Pooling, hence a large number of unimportant parameters eliminated. The latest introduced version of GoogleNet is Inception-v4 [43, 2].

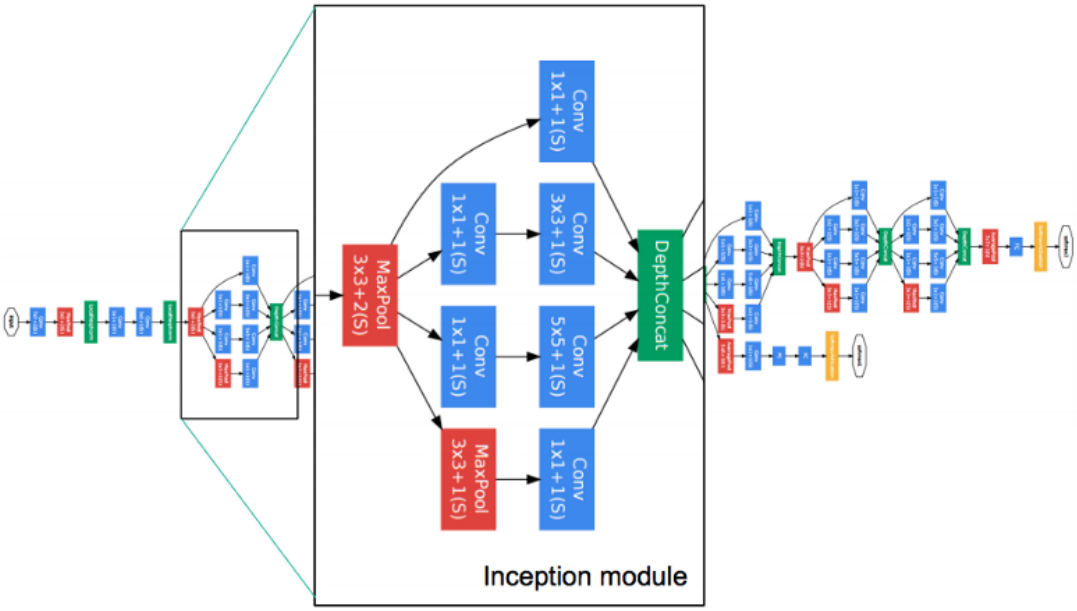


Figure 2.18: GoogleNet CNN architecture layers [41].

- **ResNet**
  ResNet was the winner of ILSVRC 2015 and was developed by Kaiming He et al. The Fully Connected layers are missed from the end of the network while heavy use of batch normalization exists [44].

2.2 High-Level Synthesis and Field-Programmable Gate Arrays

This section provides the necessary theory of Field-Programmable Gate Arrays (FPGAs). The first part focuses on High-Level Synthesis(HLS), which is considered as a programming methodology for FPGAs in high-level languages such as C and

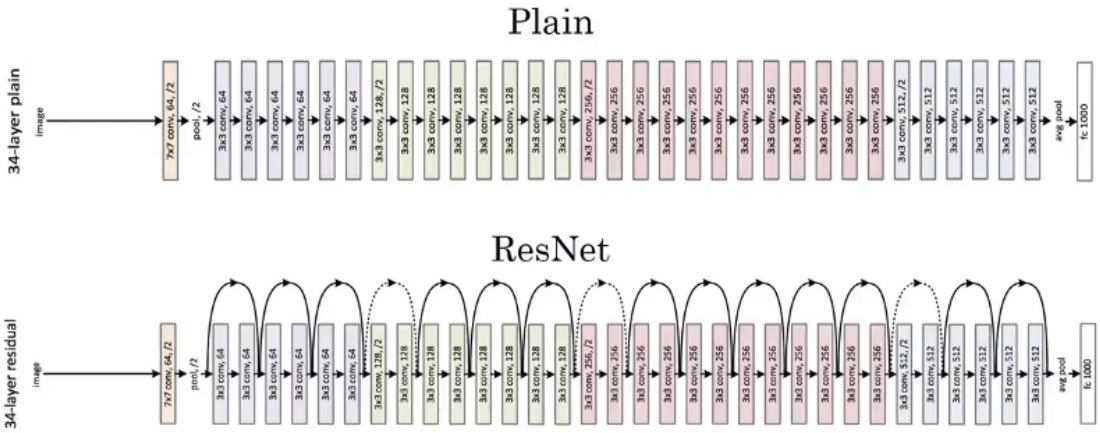


Figure 2.19: ResNet CNN architecture layers [41].

C++ and the second part highlights characteristics and basic details of the FPGA technology.

2.2.1 Introduction to High-Level Synthesis

Latest generation High-Level Synthesis tools and their support of language like C, C++, OpenCL, etc has increased the programmer's productivity and provide a friendlier environment for the developers that are not familiar with Hardware Description Language (HDL), such as VHDL or Verilog in depth. Register Transfer Level (RTL) describe combinational logic, basic arithmetic operations as well as registers. Moreover, RTL becomes active at either the rising edge, falling edge of a clock signal and they are very close to the logic gates and wires of circuits that consists the FPGA or ASIC technology. Hence, RTL synthesis can control the resulted hardware, in which a programmer should make design decisions that later any change would be difficult and costly to happen. Furthermore, HLS tools encourage programmers to move from Register transfer level to more abstract and optimized designs and algorithms that can follow along with the rapid development of systems-on-chip(SoC) without being experts or experienced in depth. The implementation in HLS for application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs) optimized considering the power, performance and cost requirements of a certain system [45, 46].

A major problem of these days is the complexity of high-level code and the ratio of the VHDL developer population in comparison to C/C++. Moreover, the programmers can separate functions from architecture made the design and components more portable and easier specified using C and similar languages. However, the hardware is potentially slower and inefficient, as it is fundamentally concurrent.

As well as, C language does not support parallelism, so we should use properly libraries to avoid sequential execution. Therefore, it is still easier to use C/C++ language as there is an extensive infrastructure of compilers, books, standards, etc and many programmers who know and it is more easy to integrate their skills in C/C++ than learn a new language or to use correctly and efficiently complex ones like VHDL or Verilog.

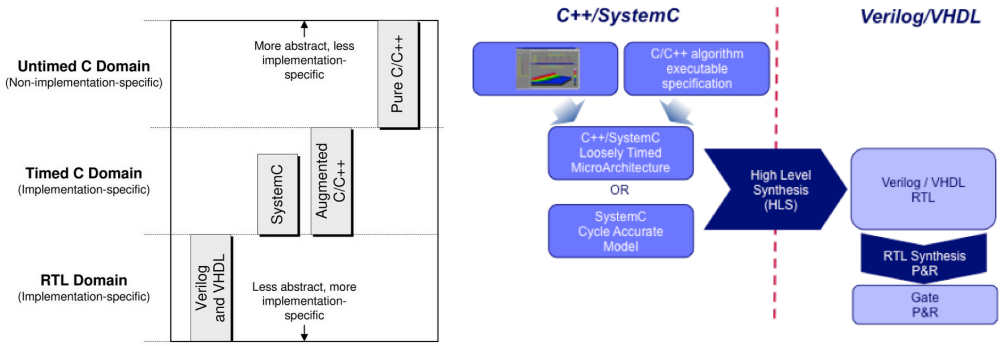


Figure 2.20: Left:HLS high-level to low-level code[47]. Right: A typical C++/SystemC HLS flow[48].

One of the most popular HLS compilers is Vivado High-Level Synthesis (VHLS) by Xilinx. Designers can use most features of C/C++ language as loops, arrays, structs, floats, etc. All these are automatically converted into counters, memories, computation cores, handshake protocols, leading state machines and schedules. The compilation is depending on the system libraries and the choices that the user will decide to use, for example, C++17 version, meta-instructions as "pragma" for parallel operations and many more [46].

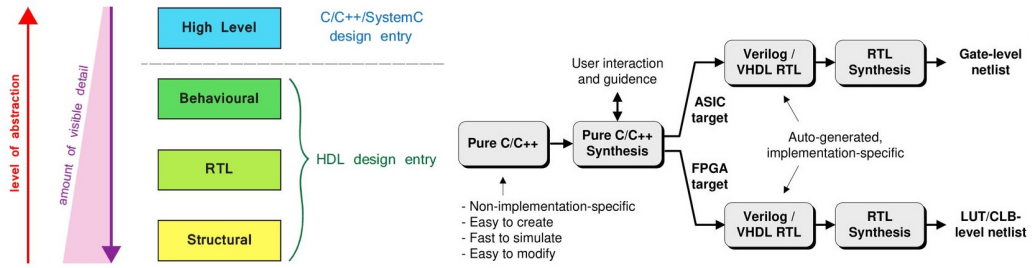


Figure 2.21: Left:Levels of abstraction in FPGA Design[49]. Right: Pure Untimed C/C++ Design Flow [47]. Writing RTL that works smoothly on both FPGA and ASIC implementations is possible using HLS technology

2.2.2 Introduction to Field-Programmable Gate Arrays

FPGAs are integrated circuits (ICs) that can be programmed for different algorithms and configured by a developer to implement digital circuits. Logic cells of an FPGA can be configured to implement software algorithms that are cost and performance efficient in many computational problems of common functions. FPGAs contains a two-dimensional array of logic blocks and programmable switches that are connected via programmable interconnects. Furthermore, FPGAs provide the ability of dynamical reconfiguration which can affect part (partial reconfiguration) or all of the resources that are available in the FPGA (full reconfiguration). Programming information for the logic blocks, the interconnects and the function units are contained in a bitstream that is commonly held in SRAM. This process, which is the same as loading a program in a processor, can affect part or all of the resources available in the FPGA fabric.[50, 51]

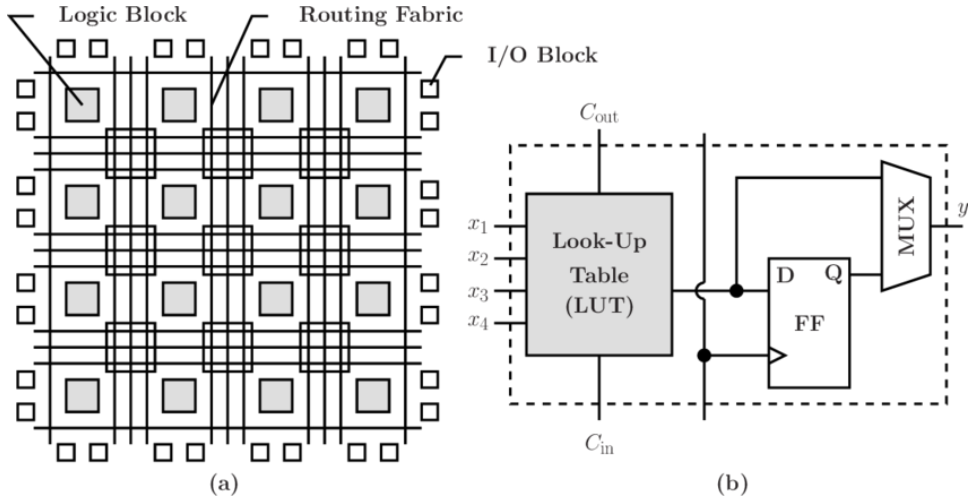


Figure 2.22: (a) : Sketch of the FPGA architecture and (b): a diagram of a simple logic block FF, flip-flop[52].

Differences between FPGAs and ASICs

Application-Specific Integrated Circuits (ASICs) are custom-tailored semiconductor devices or microchips that are designed for a special application. There are many differences between FPGAs and ASICs, as well as, many advantages or disadvantages in both of them. The major differences referred to tool availability, performance, power consumption, design flexibility and the cost of production. More specifically, ASICs can no longer be altered after they get out of the production line, they are not reusable as FPGAs and they demand lengthier development cycles that turn out to be costlier. However, they are smaller, faster and more energy-efficient as they do not suffer from timing overhead from generic intercon-

nects and configuration logic in contrast to FPGAs. We tend to prefer ASICs for high volume, complex, high-cost applications and it is easier to develop software on them while programmable FPGAs are more suitable for prototyping, lower speed, lower complexity, and lower volume designs [46].

Differences between FPGAs and GPUs

Graphics Processing Units (GPUs) are specialized for arithmetic intensive, highly parallel computations while the same program will be executed repeatedly in parallel structure. It is possible to achieve high speedup when the program is partitioned into smaller, independent blocks of data elements that each execute the same code with a lower requirement for sophisticated flow control, and perform minimal conditional branching. Hence, the memory access latency can be hidden with calculations and instruction/thread-level parallelism, although huge data caches. The languages that are commonly used to program a GPU are CUDA (for NVIDIA GPUs) and OpenCL (general-purpose). The advantage of an FPGA over a GPU is the availability of freely programmable general-purpose logic blocks in which they can have heavily specialized accelerators for specific tasks, resulting in advanced processing speed, higher throughput, and energy savings [53, 54, 55].

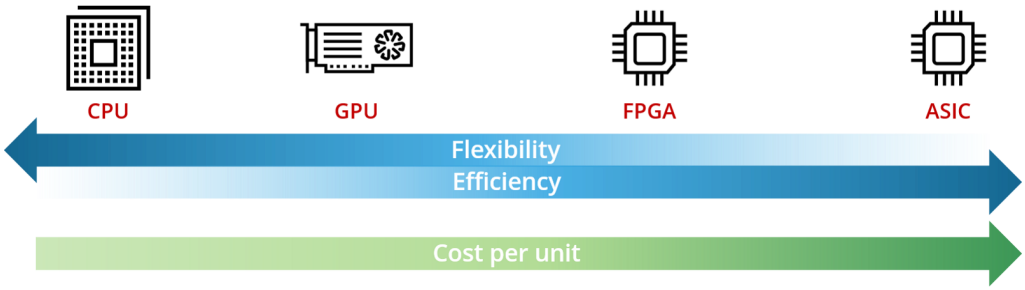


Figure 2.23: Tradeoffs between CPU, GPU, FPGA and ASIC [56].

Chapter 3

CNN Implementation, Training and Optimizations

3.1 Introduction

The previous chapter introduced the goals of this thesis: The implementation of a CNN Framework for training from scratch and the acceleration of the Fully Connected using an FPGA as accelerator.

This chapter is dealing with the first aspect that is the implementation of a CNN Framework, especially the training process and some optimizations for better performance. Firstly, at the following section -3.2-, includes a summary list of the goals and specifications of our implementation. In section 3.3, it is presented an overview of the framework, illustrating the hierarchy of framework files (Figure 3.1) and then the top-level procedures in a basic flowchart (Figure 3.2). To be more precise, in Section 3.4 there is a synoptic outline of the content of the layers that we have implemented, so far. Afterward, there is the presentation of the dataset that it is used and tested, which is known as MNIST (Section 3.5). The following section (3.6) discusses the processing of data before the training, such as weight initialization, normalization, and regularization of the data and the parameters. Section 3.7 and 3.8 introduce the training techniques, such as softmax classifier which use cross-entropy loss, regressions and the SGD Momentum, that are currently implemented for the CNN Framework. Following that, there is a whole section(3.9) to explain the learning process, that the CNN Framework follows, of the present design. The last section 3.10 finally discusses some GNU and OpenMP optimizations and techniques that seem to have better time performance in the custom-tailored Convolutional Neural Network architecture than in naive implementation.

3.2 Goals and Specifications

The aforementioned theory motivated us to implement a CNN Framework that will use the advantages of an FPGA Ultrascale board for the training procedure. The board has been used for IP core development was Xilinx ZYNQ UltraScale+

XCZU9EG MPSoC due to the significant computational load of a CNN. Our approach was to design and develop a system that optimizes matrix multiplication of Fully Connected layer. This layers will have adapted, fully functional FPGA-based kernels as future work. The accelerator will be flexible for any CNN topology as the Fully Connected bitstream and the dataset will be in an SDcard [46].

The main goals are :

1. Develop and implement a CNN framework.
2. Accelerate matrix multiplication and addition (X*W +b) of Fully Connected layer
3. MNIST dataset
4. Optimize training procedure
    a) OpenMP
    b) Fisher-Yates algorithm
    c) -O3 technique
    d) an FPGA accelerator
5. The FPGA-based architecture will be developed on a Zynq UltraScale+ MPSoC (ZU9EG, 64-Bit DDR4, 8 GByte [57]).
6. Adjust HLS code and Vivado Design to Framework in Vivado SDK tool.
7. Test and verify the whole system
8. Set new goals/next steps for future work development and optimization.

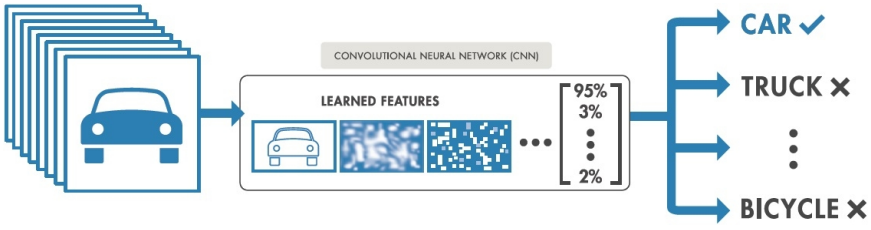


Figure 3.1: CNN learns features from input images and classifies them, as it is illustrated. [1]

3.3 Framework Overview

Our design was based on a Python implementation [58] for Stanford's CS231n course project[2]. First of all, we tested and made corrections to the model-Python code. Then, the initial Python code was modified to update, regularize and get weights and biases in each layer, instead of keeping and modify them at the end of every minibatch, using lists.

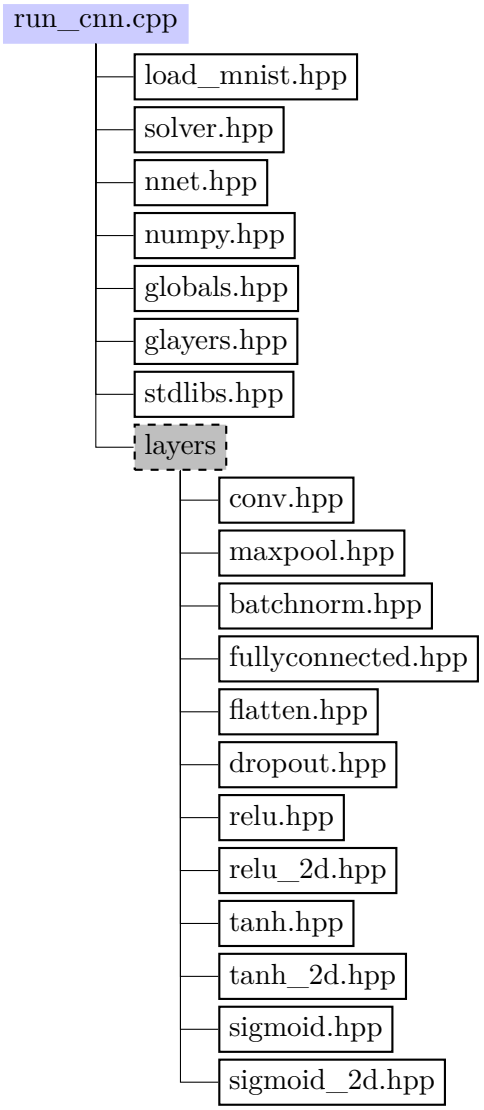


Figure 3.2: The structure of C++ code files. The main function is in .cpp file or more specifically in run_cnn.cpp, which is the top-level file of our framework.

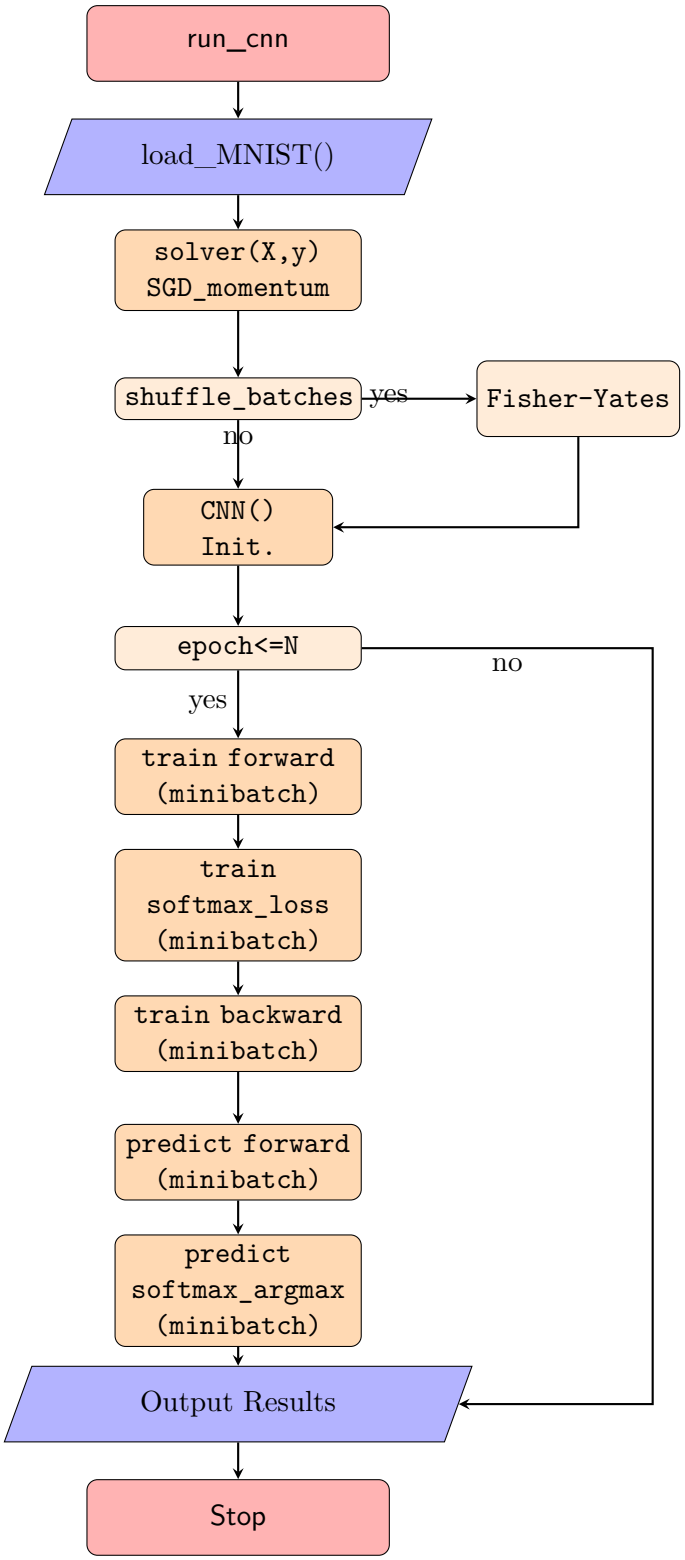


Figure 3.3: Top-Level flowchart of CNN Framework with MNIST dataset. Layers have constructors(for initialization), forward and backward routines. Layers are connected via variadic templates which is a feature of C++17 for the recursive call of functions. The output of the previous layer becomes the input of the next layer.

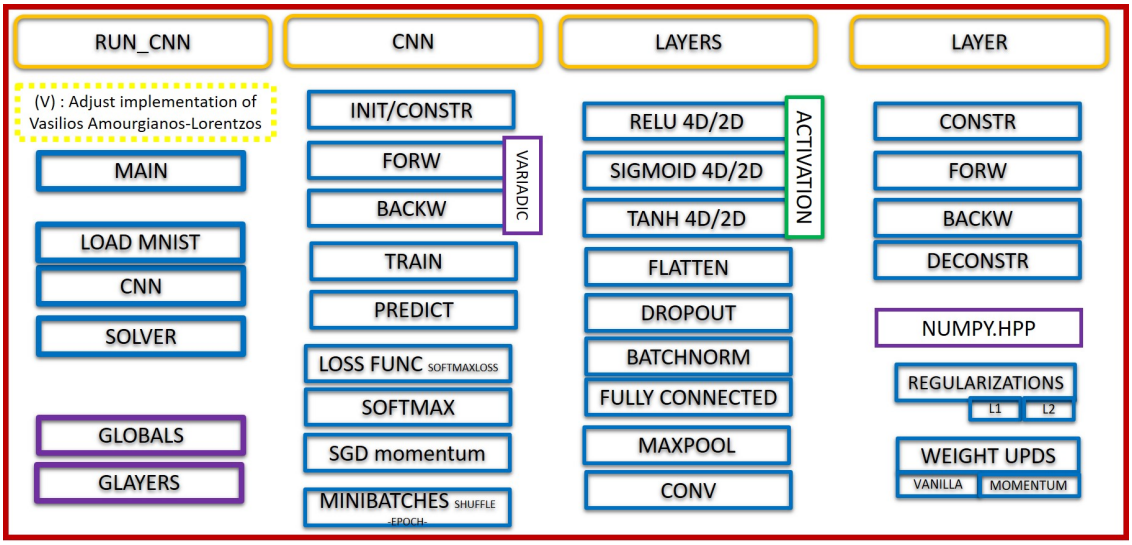


Figure 3.4: An object-oriented view of the developing framework.

The C++ Framework uses classes in each layer that are similar to the Python code. The parameters are kept in namespaces and they can be modified from globals.hpp. The parameters of layers should be properly used as in Python code, which means that a user should calculate correctly the inputs for the padding, height, width, etc, of the network that she/he wants to construct. The layers and its templates can be modified from glayers.hpp, in a similar way as in the Python code. Although the majority of layers is based on the Python code, the Maxpool Layer is implemented differently, in a more naive solution, without im2col function. The "#define LAYERS" should be filled with the sequence of the network that the user wants, using the feature learning(optionally) and the classification(mandatory) parts of a CNN/FNN. Hence, this sequence will be used from the variadic template, which is a feature of C++17, and recursively will call the arguments-layers- in the forward and backward function call to perform the training of the network. Moreover, the activation functions that a user will use, are implemented for 4D and 2D input/output static arrays for the feature learning and the classification independently.

The network uses Stochastic Gradient Descent (SGD) with momentum update and shuffle of the batches. We also implement vanilla update and both L1 and L2 regularization. Furthermore, we implement the read/load of MNIST dataset to train our network and test it. Further implementation of the framework will be part of our future work.

The initial parts of code from Paras Dahal [58] and the changed ones of the Python code are:

- **The CNN class**
- **The SGD solver function**
- **The momentum update function**
- **The regularization functions, l1, l2 and their derivatives delta_l1, delta_l2**

that transform list of arrays to single lists that are better handled from C++, because of our platform's memory limitations.

3.4 Implementation of the Layers

Following the theory above and a Python implementation [2, 58], the classes contain the following functions that they are made from scratch:

Convolutional Layer

We construct a class for Convolutional layer using the im2col process:

- **Conv()** : the Class constructor
  - **random_randn_4d(W)** : He-et-al initialization for 4D Weight array
  - **zeros_bias(b)** : zero initialization for bias array
- **forward(X)** : X is a 4D array with image features
  - **im2col_indices** : the 4D array become a 2D column vector for the mini-batch of images
  - **ravel** : ravel reshape Weight array from 4D to 2D
  - **matmul_bias_conv** : matrix multiplication with cache blocking. The output array is initialized by getting the bias array values
  - **unravel** : reshape the output of matrix multiplication from 2D to 4D
  - **transpose_4d_conv_fw** : re-order 4D array to get batch-size at first dimension as the X feature (input) array
  - **l1_regularization_conv** or **l2_regularization_conv**: compute the regularization loss (details in Section 3.4.1)
- **backward(dout)** : dout is the derivative of the 4D feature array

- **transpose_4d_conv_bw** : transpose 4D array to get the number of filters at the first dimension
- **ravel** : the transposed array becomes 2D from 4D
- **transpose** : the 2D output array of im2col_indices (forward function) transposes before matrix multiplication
- **matmul** : perform matrix multiplication to get derivative of Weight array
- **unravel** : reshape 2D array of previous matrix multiplication to get 4D derivative Weight array
- **sum_4d_axis** : sum all the values to 2nd dimension of the transposed dout input array
- **ravel** : reshape 4D Weight array (forward function) to 2D
- **transpose** : transpose the 2D Weight array
- **matmul** : matrix multiplication of the 2D transposed Weight array and the 2D dX input array
- **col2im_indices** : perform col2im process to get a 4D array of features as it is described above
- **delta_l1_regularization_conv** or **delta_l2_regularization_conv**: compare Weight and bias with their derivatives to find the difference and regularize it (details in Section 3.4.1)
- **vanilla_update_conv** or **momentum_update_conv** : compare the Weight and the bias arrays with their derivatives to find the difference (details in Section 3.6)

Maxpool Layer

- **Maxpool()** : the Class constructor
- **forward(X)** : X is a 4D array with image features
  - **maxpool_fw** : find the max values of the X (4D input array) of the given size kernel/filter/window and rest of them become zeros.we also keep their positions
- **backward(dout)** : dout is the derivative of the 4D feature array

- **maxpool_bw** : keeping the positions from forwarding process and replace the same positions of with the values with the ones of derivative array dout

Batch Normalization Layer

- **Batchnorm()** : the Class constructor
  - **ones_bn** : fill beta array with 1
  - **zeros_bn** : fill gamma array with 0
- **forward(X)** : X is a 4D array with image features
  - **ravel** : reshape array from 4D to 2D
  - **mean_axis_0** : find mean value for μ_B
  - **var_axis_0** : compute σ_B^2
  - **norm_up**, **norm_down** and **norm_x** : compute $\hat{x_i}$
  - **mul_bn** : calculate y_i output array with 2D shape
  - **unravel** : reshape array from 2D to 4D
- **backward(dout)** : dout is the derivative of the 4D feature array
  - **ravel** : reshape array from 4D to 2D
  - **subtr_bn** : subtract X input array from forward function with μ_B array to get the derivative of μ_B
  - **sum_2d_bn_axis_0** : sum by concatenate zero axis of 2D dout array to get the derivative of β
  - **mul_bn_sd** and **sum_2d_bn_axis_0** : multiply 2D dout array and $\hat{x_i}$ array and then sum by concatenate zero axis of the multiplication's 2D output to get the derivative of γ
  - **mul_bn_dout_bn** : calculate derivative of $\hat{x_i}$
  - **dvar_bn** : compute the derivative of σ_B^2
  - **dmu_bn** : calculate the derivative of μ_B
  - **dout_bn** : calculate the output array as 2D
  - **unravel** : reshape array from 2D to 4D

- **vanilla_update_bn** or **momentum_update_bn** : compare the Weight and the bias arrays with their derivatives to find the difference (details in Section 3.6)

Flatten Layer

- **Flatten()** : the Class constructor
- **forward(X)** : X is a 4D array with image features
  - **ravel** : reshape array from 4D to 2D
- **backward(dout)** : dout is the derivative of the 2D feature array
  - **unravel** : reshape array from 2D to 4D

Fully Connected Layer

- **FullyConnected()** : the Class constructor
  - **random_randn** : He-et-al initialization for 2D Weight array
  - **zeros_bias** : zero initialization for bias array
- **forward(X)** : X is a 2D array with image features
  - **copy_2d_array** : make a copy of input array to use it in the backward function
  - **ravel_2d1** : This function currently help us to pass 2D arrays as 1D in accelerator
  - **matmul_bias** : matrix multiplication with cache blocking. The output array is initialized by getting the values of bias array
  - **MulAdd_start** : call FPGA accelerator instead of matmul_bias if the second dimension of output array is $> n * 32$, for $n >= 0$
  - **unravel_1d2** : This function currently help us to transform the accelerator's 1D output array to 2D
  - **l1_regularization_fc** or **l2_regularization_fc** : compute the regularization loss (details in Section 3.4.1)
- **backward(dout)** : dout is the derivative of the 2D feature array

- **transpose** : transpose the forward's 2D input array before matrix multiplication
- **matmul_f** : perform matrix multiplication to get derivative of Weight array
- **sum_2d_zero_axis** : sum the values of 2D array into 1d to get the derivative array of bias
- **transpose** : transpose the Weight array
- **matmul** : perform matrix multiplication between the dX input array and transposed Weight array to get the output array
- **delta_l1_regularization_fc** or **delta_l2_regularization_fc**: compare Weight and bias with their derivatives to find the difference and regularize it (details in Section 3.4.1)
- **vanilla_update_fc** or **momentum_update_fc** : compare the Weight and the bias arrays with their derivatives to find the difference (details in Section 3.6)

Dropout Layer

- **Dropout()** : the Class constructor
- **forward(X)** : X is a 2D array with image features
  - **random_binomial** : fill mask array with random values that are chosen by the binomial distribution of a given probability value
  - **mul_sd** :multiply the X array with the mask(includes values chosen to keep or pruned by a given cut-off factor) array
- **backward(dout)** : dout is the derivative of the 2D feature array
  - **mul_sd** : multiply the dout array with the mask(includes values chosen to keep or pruned by a given cut-off factor) array

ReLU Layer

- **Relu()** : the Class constructor
- **forward(X)** : X is a 4D/2D array with image features

  - **copy_2d_array** : make a copy of input array to use it in the backward function
  - **relu_fw** : if X input array is less than zero it prunes it
- **backward(dout)** : dout is the derivative of the 4D/2D feature array
  - **relu_bw** : if X input array is less than zero it prunes it, otherwise it keeps the value of dout aray

Sigmoid Layer

- **Sigmoid()** : the Class constructor
- **forward(X)** : X is a 4D/2D array with image features
  - **sig_fw** : calculate the sigmoid of input array
- **backward(dout)** : dout is the derivative of the 4D/2D feature array
  - **sig_bw** : calculate the derivative sigmoid of input array

Tanh Layer

- **Tanh()** : the Class constructor
- **forward(X)** : X is a 4D/2D array with image features
  - **tanh_fw** : calculate the tanh of input array
- **backward(dout)** : dout is the derivative of the 4D/2D feature array
  - **tanh_bw** : calculate the tanh of input array

The inner calculations of the aforementioned functions and their functionality are described in the previous chapter (Section 2.3.1), as well as in [2, 8].

3.5 MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology database) of handwritten digits in gray-scale is a subset of US NIST (National Institute of Standards and Technology), which is a larger set. It is commonly used for training and testing in the machine learning field [59]. The training set consists of handwritten digits from 125 high school students (50%) and 125 employees (50%) (totally 250 different people - 100%) from the Census Bureau. Also, the test set contains handwritten digits from different people like the aforementioned description.

There are four files available, which contain separately train and test, and images and labels. The training set contains 60000 examples and the test set 10000 examples. The first 5000 examples of the test set are taken from the original NIST training set, whereas the last 5000 examples are taken from the original NIST test set. Moreover, the first 5000 examples are cleaner and easier than the last 5000 examples. It is important to know that, all the integers in the files are stored in the MSB (Most Significant Bit) first (high endian) format used by most non-Intel processors. Hence, users of Intel processors and other low-endian machines must flip the bytes of the header [60]. The aforementioned information from MNIST's website was necessary to understand how to load and use MNIST properly.

For the MNIST dataset, we have 3 partially different implementations. One of them uses the ff.h library to read the sdcard via FATFs for the to run the code in the platform and the other two uses the ifstream to read the dataset. The last two differs in the way that they read the data. One of them uses the reinterpret_cast of C++ while the other uses a buffer to do the same process and adjust them in the array.

Method	ELOAD (reinterpret_cast)	VLOAD	FATFs bare-metal
Avg. Time (*sec*)	0.041024960	0.0419951540	40.5107269

Table 3.1: Load **whole** MNISTs time for each of the 3 ways (VLOAD and ELOAD (Kronos CPU-20 threads OMP), FATFs in bare-metal(Trenz ARM-1 thread)

3.6 Design choices

Before the learning process, we should make some assumptions for the model that we will follow as it will affect the whole learning process.

Some of the files have the parameters (globals.hpp) and the layers (glayers.hpp or lenet5.hpp) or even the libraries (stdlibs.hpp) that can be modified easily. The

user is not necessary to know in-depth what the whole CNN do to make a simple modification. It can be further parameterized in a future release.

Data Preprocessing

The range of MNIST images is from 0 to 255, hence we normalize it by dividing each cell by 255 to get a new range from 0 to 1.

Parameters Initialization

Weights will store some values that they learn during the training, but before that they should initialize in a value that we will decide and help the network to learn quickly and efficiently.Firstly, for many reasons that are explained in [2], it's recommended to initialize the weights with He-et-al [24].In this case, especially for ReLU activations that we have in our network with $2.0/n$ variance and standard deviation $\sqrt{2.0/n}$, so the He-et-al initialization will be

$$W = \frac{\text{np.random.randn}(n)}{\sqrt{(2.0/n)}} \tag{3.1}$$

If we would have Tanh activation functions we would use Xavier initialization which is

$$W = \frac{\text{np.random.randn}(n)}{\sqrt{(1.0/n)}} \tag{3.2}$$

and differs only in divisor of the square root. However, the implementation of Xavier initialization is part of future work. At this moment, the user should change it by hand.

These techniques warrant that gradients do not vanish or explode too quickly, while they avoid slow convergence and also ensure that we do not keep oscillating off the minima and minimize the variance of the parameters. However, the bias simply will be initialized to zero [2, 24].

Batch Normalization

Batch normalization is implemented in Batchnorm Layer (subsection 2.3.1 and 3.4). It is common to insert it after Fully Connected or Convolutional layers or before ReLU layers. ReLU activation layers of the network adopt a unit Gaussian distribution at the beginning of the training while properly initializing neural networks[2]. There is a detailed explanation in the paper of Ioffe and Szegedy [27].

3.6.1 Regularization

There are many techniques to increase the test accuracy while the algorithms we want to use should adjust the changes of new inputs. These techniques are referred to as regularization techniques. We perform regularization techniques to improve performance and perform generalization. Additionally, regularization can prevent our model from overfitting or underfitting phenomena that we should avoid.

L1 Regularization

L1 or Lasso or L1 norm adds $\lambda * \sum |W|$(absolute value of magnitude) of coefficient as penalty term to the loss function multiplied by λ a positive hyperparameter, controlling the strength of the regularization. Furthermore, L1 leads the weight arrays to become sparse, which means too close to absolute zero, during optimization, and it has multiple solutions. L1 can not learn complex patterns and it is robust to outliers, in other words, the "noisy" inputs almost do not affect neurons with L1 regularization. However, it has a built-in feature selection mechanism, while it generates a model that is simple and interpretable. The backpropagation call the L1 function that adds $\lambda * \frac{W}{|W|+1e^{-8}}$ to the Weight gradient [2, 61, 18].The $1e^{-8}$ is added in case the dividend ($|W|$) is zero and we want to avoid it.

Algorithm 1 L1 Regularization

Input: W: Weight array
Output: out: calculation result

1: *#pragma omp parallel for schedule(static, prmts::sch_static) private(i, j, k, l)*
2: **for** all dimensions of W array **do**
3: $sum = sum\_all(abs(W))$
4: **end for**
5: // default lam = 0.001
6: $out = lam * sum$
7: return *out*

Algorithm 2 Derivative of L1 Regularization

Input: W: Weight array
Output: dW: derivative of the Weight array

1: // default L1_REG_CON = 1e-8 and lam = 0.001
2: *#pragma omp parallel for schedule(static, prmts::sch_static) private(i, j, k, l)*
3: **for** all dimensions of W array **do**
4: $dW+ = prmts :: lam * W/(abs(W) + L1\_REG\_CON)$
5: **end for**

L2 Regularization

L2 (Gaussian) or Ridge or L2 norm or Tikhonov regularization is the most common regularization method. Contrary to L1, L2 has one non-sparse solution, has not to feature selection and it is not to robust to outliers. Moreover, L2 can learn complex data patterns, while it can give better prediction when the output variable is a function of all input features. The L2 regularization diffuse weight array while it heavily penalizing peaky weight arrays. At forwarding, the L2 regression adds $\frac{1}{2}\lambda * \sum W^2$(squared magnitude) of coefficient as penalty term to the loss function multiplied by λ parameter. At the backward procedure, the L2 method adds $lambda * |W|$ to the Weight gradient.

Algorithm 3 L2 Regularization

Input: W: Weight array
Output: out: calculation result

1: // default lam = 0.001
2: $sum = sum\_all(pow(W, 2))$
3: $out = (1/2) * lam * sum$
4: return *out*

Algorithm 4 Derivative of L2 Regularization

Input: W: Weight array
Output: dW: derivative of the Weight array

1: // default lam = 0.001
2: *#pragma omp parallel for schedule(static, prmts::sch_static) private(i, j, k, l)*
3: **for** all dimensions of W array **do**
4: $dW+ = prmts :: lam * W$
5: **end for**

Usually, L2 is used combined with a Dropout at the end of the layer's sequence. The default is $p = 0.5$, but it can be tested in range 0.2-0.8 depending on the accuracy-time needs. Bias regularization is not common, as it gives a worse performance to Neural Network. A combination of both L1 and L2 regularization is named Elastic net regularization [2, 61, 18].

Our Network use by default the L2 regularization technique which can be changed from globals.hpp which is set as a global parameter. At the moment the user can choose between L1, L2 or both. We get the estimated regularization loss at the end of the forward function of Convolutional and Fully Connected Layers that use Weight and bias parameters. Respectively, at the same layers, at the end of the backward function, before the update of weight and bias arrays, we perform the

delta L1, L2 or both regularization methods to "correct" the weight gradient arrays for the next batch or epoch.

Dropout

Apart from the techniques above, Dropout is a technique to avoid overfitting in a Neural Network. However, Dropout is implemented as a Layer (subsection 2.3.1 and 3.4).

There many more techniques apart from the above to prevent overfitting or underfitting of a Neural Network. Some of them are :

- Dropout
- Regularization (L1, L2)
- Data augmentation
- DropConnect
- Feature scale clipping
- Global average pooling
- Early stopping
- Cross-validation
- Injecting Noise
- Ensembling
- ...

We have implemented the first two techniques (Dropout and Regularization (L1, L2)). The others are considered as future work.

3.7 Loss functions

During of training process, the classification part contains a loss function of every forward pass, also known as cost function or objective. At backpropagation, the extracted results from loss function will be the input data of the last layer of the network. The network will keep a loss value that will be increased/decreased during the passes of mini-batches. The quality of the parameters of a particular set is expressed by the loss function [2].

3.7.1 Softmax classifier

Softmax function takes as input the real values from an array and transforms them into real values that are in the range (0,1). The sum of all values of the array is up to 1. The equation of the softmax function is:

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}, \qquad \forall j \in 1...N \tag{3.3}$$

The output values are analogous to input values but in a changing range. The maximal input value will get the largest portion of the distribution in the new range. More specifically, the Softmax class classifier has a probabilistic interpretation mapping "softly" the values of the input array to probabilities, meaning that the output values will have the new range (0,1). Full Softmax which is the variant that we describe is the one that we will use to calculate the probabilities for every possible class. When we do not have many classes the Full Softmax is cheap, hence we use it for the MNIST dataset which contains 10 classes of single-digit numbers (0-9). The alternative variant of Softmax, which we can use for a huge number of classes is Candidate sampling, therefore we will not deal with it at the moment, so we will focus on Full Softmax exclusively [8, 62, 63].

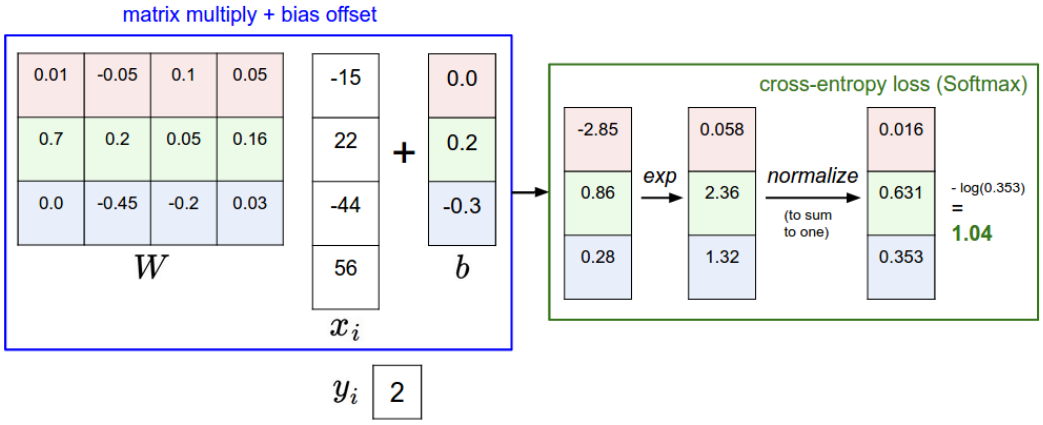


Figure 3.5: Softmax Classifier example for one data point. The final loss of the calculations for Softmax classifier is $-ln(0.353) = 1.04$ [2].

Cross-Entropy

A common loss function that we are currently using is cross-entropy . As we know, Softmax classifier uses the Cross-Entropy loss. The equation of cross-entropy is formed as:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \qquad \text{or} \qquad L_i = -f_{y_i} + \log\sum_j e^{f_j} \tag{3.4}$$

It is used often in regression or classification problems. Cross-entropy has an interesting probabilistic interpretation and information theory view. We can define it for two discrete probability distributions q(estimated distribution) and p(true distribution) as:

$$H(p,q) = -\sum_{x} p(x) \log q(x), \quad where \quad q = e^{f_{y_i}} / \sum_{j} e^{f_j} \tag{3.5}$$

and p is a vector containing only one chosen value "1" in a vector position as a flag. For example, when we detect in an image of MNIST dataset the number 2 then we get $p = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$.

The Probabilistic interpretation minimizes the negative log-likelihood performing Maximum Likelihood Estimation (MLE) in the proper class and can be expressed as [2]:

$$P(y_i \mid x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \tag{3.6}$$

The derivative of cross-entropy is:

$$\frac{\partial L_i}{\partial f_k} = p_k - 1(y_i = k), \quad where \quad p_k = \frac{e^{f_k}}{\sum_j e^{f_j}}, \quad L_i = -\log\left(p_{y_i}\right) \tag{3.7}$$

[2] and a further detailed computing explanation there is in [62].

The classification function that we implemented following Cross-Entropy and Softmax methods:

Algorithm 5 Loss Function - Cross-Entropy and Softmax

Input: X: output array from forward function
y: labels' array

Output: out: output array transformed in range (0, 1)
loss_out : loss value

1: $max\_2d\_single\_axis(X, X1d)$
2: **for** all dimensions of X array **do**
3: $Xexp = exp(X - X1d)$
4: **end for**
5: $sum\_2d\_single\_axis(Xexp, Xexp1d)$
6: **for** all dimensions of Xexp array **do**
7: $p = Xexp/Xexp1d$
8: **end for**
9: $sum\_2d\_single\_axis(Xexp, Xexp1d)$
10: **for** all dimensions of p array **do**
11: // `(of p[i][j])`
12: **if** $y == j$ **then**
13: $p\_likelihood = p$
14: **end if**
15: **end for**
16: **for** all dimensions of p_likelihood array **do**
17: $log\_likelihood = (-1) * log(p\_likelihood)$
18: **end for**
19: **for** all dimensions of log_likelihood array **do**
20: $loss+ = log\_likelihood$
21: **end for**
22: $loss_out = loss/minibatch\_size$
23: **for** all dimensions of p array **do**
24: // `(of p[i][j])`
25: **if** y == j **then**
26: $out = p - 1$
27: **else**
28: $out = p$
29: **end if**
30: **end for**
31: **for** all dimensions of out array **do**
32: $out/ = minibatch\_size$
33: **end for**

3.7.2 Regression

Regression is the squared difference of target and predicted value of the weights Convolutional or Fully Connected Layers. It is performed exclusively in these two layers because only they use weight arrays. The derivative of weight arrays is updated accordingly. Regression is defined as derivative function/delta of L1 and L2

Regularization as it is performed during the backpropagation (for more details see 3.4.1) [2].

3.8 Optimization algorithm

An optimization technique connects train(forward-backpropagate) and predict /test(forward) processes. Optimization algorithm attempts to improve loss, while it updates the parameters of the network to achieve it. More specifically, we try to find a set of weights that approaches as close as possible to the optimal, but also it is necessary to minimize the loss function. There are several optimization algorithms to train our network. Therefore, we chose the Stochastic Gradient Descent method with momentum update and Nesterov to implement at the moment [64, 18, 2].

3.8.1 Gradient Descent

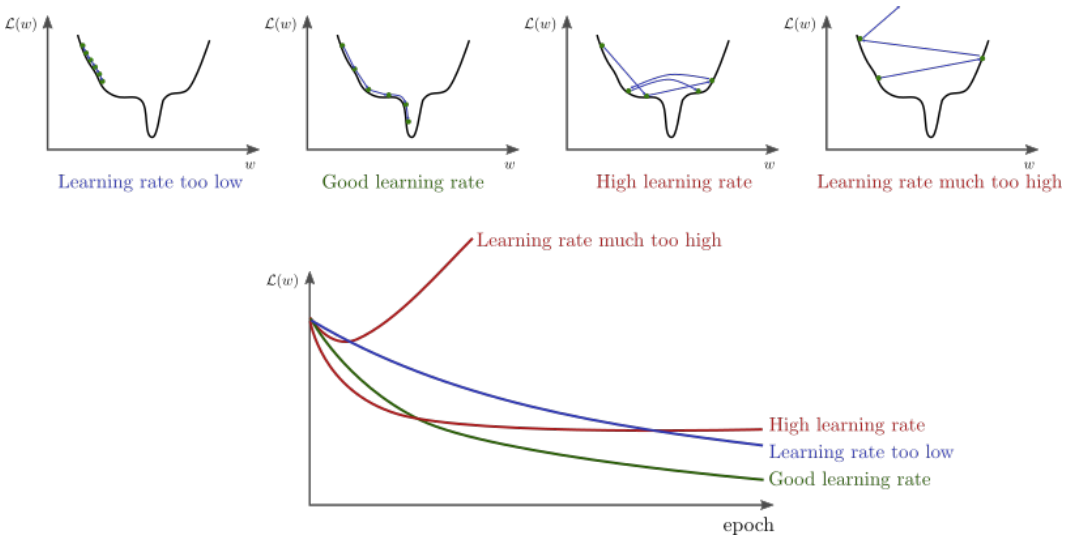


Figure 3.6: Cases of low, good, high and too high learning rate in contrast to the given model [65].

It is the most popular, simple and common method to optimize neural networks. There are many variations of Gradient Descent that evaluate the gradient and update the parameters of the neural network. During the training process, these optimization algorithms learn by performing small local steps using the gradient of the loss function to seek for the global minima. Gradient Descent along with Backpropagation algorithm is a popular way to perform the learning process in many

neural networks. Furthermore, Gradient Descent needs only first-order derivatives of parameters relating to the loss function. The size of the steps is referred to as Learning rate and is essential to be chosen carefully as they to determine the next point:[66, 67, 2, 8, 68].

3.8.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is very efficient, but also a simple technique that is usually used for deep learning. Stochastic Gradient Descent often is used for sparse and large-scale machine learning problems in natural language processing and text classification. It attempts to avoid stuck into a local minimum by sampling data in a stochastic manner and updating parameters repeatedly making small perturbations with the sampling data. Stochastic methods perform hill-climbing methods that effectively "guess" using a learning rate and afterward "check the "guess", hence they can allow faster convergence to the optimum. During training, the gradient descent is actually slow to converge. Hence, the variation that often is used is Stochastic Gradient Descent which estimates the gradient from mini-batches. Mini-batch is called a small sample of randomly chosen training input data in every iteration and its size is a hyperparameter for SGD. Performing many close estimated updates give us better results. Mini-batches are randomly shuffled training MNIST images using the Fisher-Yates algorithm (see 3.8.1) [2, 8, 18, 69, 70].

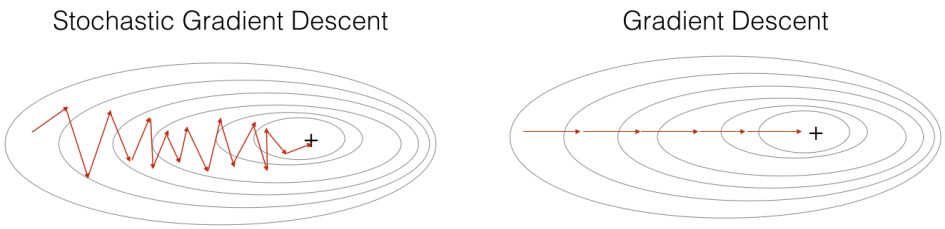


Figure 3.7: + refers to a minimum of the cost. SGD makes some small perturbations until convergence. Therefore, it is faster to compute SGD than GD, as GD uses the whole batch instead of one training example [71].

An epoch is a complete pass of mini-batches through a given dataset. Mini-batches are separated and shuffled at the get_minibatches function in python. In our implementation, during the solver, we get the stepping the size of data we need to pass to CNN class and we perform the shuffle of the training data in a function separately [2, 8].

Stochastic gradient descent with momentum performs a parameter update for

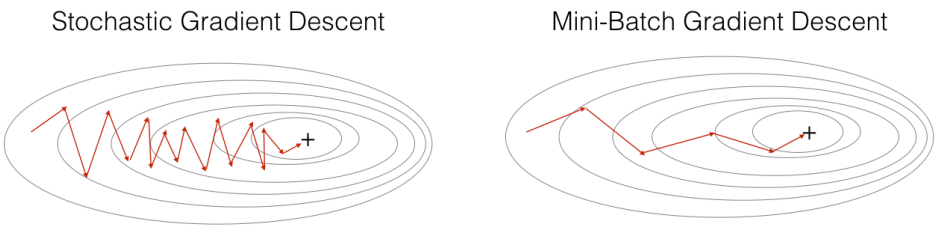


Figure 3.8: + refers to a minimum of the cost. SGD with mini-batches often leads to reaching faster convergence [71].

each training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{3.8}$$

Algorithm 6 Stochastic Gradient Descent with minibatches

Input: train_X: 4D MNIST train images, train_y: 1D MNIST train labels, test_X: 4D MNIST train images, test_y: 1D MNIST test labels

for `epochs` **do**
  for `mini_batches of train_X and train_y` **do**
    $cnn.train(train\_X, train\_y)$
  end for
  / ret is the array have stored inside the predicted values */*
  for `mini_batches of train_X and train_y` **do**
    $cnn.predict(train\_X)$
    $accuracy\_func(train\_y, cnn.ret)$
  end for
  for `mini_batches of test_X and test_y` **do**
    $cnn.predict(test\_X)$
    $accuracy\_func(test\_y, cnn.ret)$
  end for
end for

3.8.3 Vanilla Update

SGD uses the learnable parameters(Weight and bias arrays) of forwarding, as well as the gradients("dW" derivative of Weights and "db" derivative of bias arrays) of backpropagation for each layer(Conv and FC). Then, it updates parameters along the negative gradient, multiplied with learning rate [8, 2]. The equation for Vanilla gradient descent computes (for the entire training dataset) the gradient of the cost function to the parameters [67]:

$$\theta = \theta - \eta \nabla_\theta J(\theta) \tag{3.9}$$

Vanilla Update is part of the backward function of Convolutional, Batch Normalization and Fully Connected Layers of our implementation. The other layers do not have Weight, bias array parameters or in case of Batch Normalization layer gamma and beta array parameters.

Algorithm 7 Vanilla Update

Input: W: Weight array, dW: derivative Weight array
b: bias array, db: derivative bias array
Output: W: Weights array, b: bias array

1: *#pragma omp parallel for*
2: **for loop**
3: W -= $learning\_rate * dW$
4: b -= $learning\_rate * db$
5: **end loop**

3.8.4 SGD with Momentum

Momentum update or classical momentum (CM) is another technique to approach a better convergence in a deep neural network that Polyak introduces on 1964 [72]. It is motivated by the physical perspective of optimization problems. This method suggests an improvement from the SGD above. Furthermore, SGD with momentum update is claimed that the gradient directly integrates the position of the standard SGD and presents the velocity v. Velocity is the parameter that is initialized at zero using the hyperparameter mu(momentum update) which is typically 0.9 and it is stored to make the proper updates of the parameters. Hyperparameter mu can control the velocity and allow faster descent, while it is used to exclude overshooting to a specific area. SGD advantage a little of the momentum schedules, but it speedups the learning in later stages [8, 2, 20]. SGD with momentum is computed by multiplying the momentum term γ of the update vector of the previous step/layer to the current update vector and the learning rate η to the gradient of the objective function $\nabla_\theta J(\theta)$ [67]:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{3.10}$$

Nesterovs accelerated gradient

Nesterov momentum or NAG (Nesterovs Accelerated Gradient by Nesterov, 1983) operates slightly better than standard momentum as it is an improved version with stronger theoretical converge. More specifically, as we see above in Figure 3.8., NAG has better convergence rate than gradient descent in most of the cases, like the momentum method. It can be explained as a calculation of the gradient in a "lookahead" evaluation position and then perform a correction [73, 8]. The effective lookahead can be calculated by adding in the SGD momentum equation 3.10 the momentum term of NAG γv_{t-1} to move the parameters θ. The momentum term of NAG is an approximation of the next position of the parameters[67]:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\ \theta &= \theta - v_t \end{aligned} \tag{3.11}$$

Momentum Update, as well as Vanilla Update, is part of the backward function of Convolutional, Batch Normalization and Fully Connected Layers that update their Weight, bias arrays or in case of Batch Normalization layer the gamma and beta arrays of parameters.

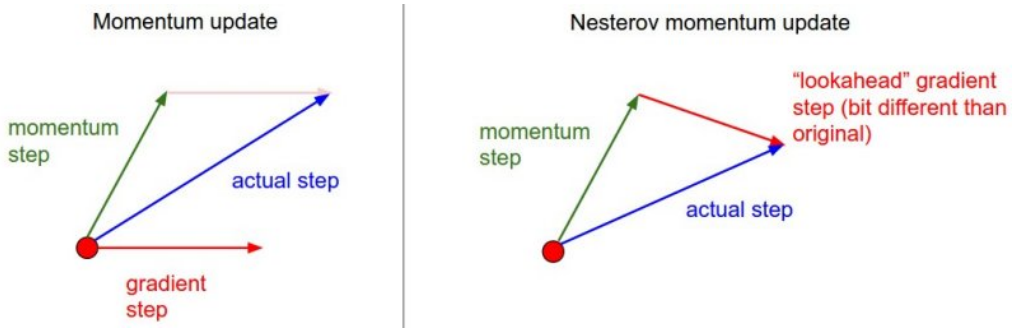


Figure 3.9: Using momentum with gradient update, we can reach out to the "looked-ahead" position [2].

Algorithm 8 Momentum Update and Nesterov

Input: W: Weight array, dW: derivative Weight array,
b: bias array, db: derivative bias array,
velW: look-ahead Weight array, velb: look-ahead bias array
Output: W: Weights array, b: bias array

```
1: /* mu = 0.9 */
2: if nesterov
3: #pragma omp parallel for schedule(static, prmts::sch_static)
4: for loop
5:    W += mu * velW
6:    b += mu * velb
7: end loop
8: end if
9:
10: /* default learning rate: l_rate= 0.01 */
11: #pragma omp parallel for
12: for loop
13:    velW = velW * mu + l_rate * dW
14:    velb = velb * mu + l_rate * db
15:    W -= velW
16:    b -= velb
17: end loop
```

3.9 Optimizations

There are a couple of optimizations that we have implemented to get a better performance in the CPU. More specifically, we tried to take advantage of the locality of cache, parallel performance, and efficient randomness. There are described in details afterward.

3.9.1 Data Processing - Shuffle: Fisher-Yates algorithm

The Fisher-Yates shuffle algorithm published in 1938 from Ronald Fisher and Frank Yates, named by their surnames. It is used to randomly shuffle a given input(list of elements). The algorithm permutes each element of the list with the same probability using a random number generator. They first describe it in their book [74]. Statistical tables for biological, agricultural and medical research. As it is described, the user writes down the finite sequence of elements. After that, she/he picks randomly a number from space $[1, k]$, which is a list of non-picked elements yet. This process is repeated with $\mathcal{O}(n^2)$ asymptotic complexity.

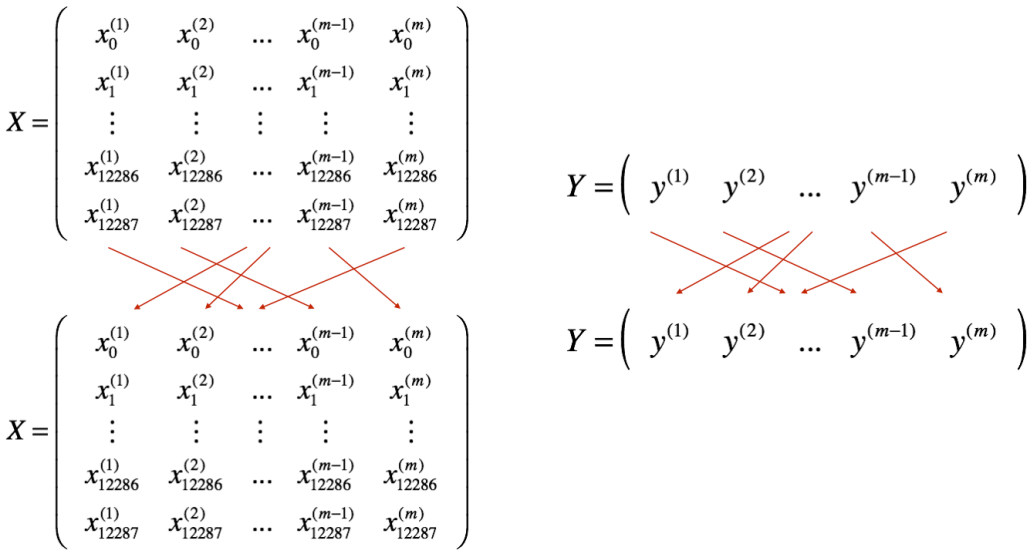


Figure 3.10: An illustration of shuffling images along with the labels [71].

Algorithm 9 Fisher-Yates shuffle Algorithm[7].

Input:X: 4D array of images, y: 2D array of labels
Output:input arrays (X, y) shuffled in any random sequence

```
int j;
srand(time(NULL));
for i = (minibatch_size − 1) downto 1 do
    j = rand()%(i+1);
    std::swap(input_X[i], input_X[j]);
    std::swap(input_y[i], input_y[j]);
end for
```

3.9.2 Durstenfeld shuffle algorithm

In 1964, a better approach of the shuffle algorithm introduced by Richard Durstenfeld and later in 1969 Donal Knuth, but it is often called the Fisher-Yates algorithm. Neither of them acknowledged Fisher-Yates work, hence it is believed that they did not know about it [75]. However, it is an improved version of the Fisher-Yates algorithm and it has linear $\mathcal{O}(n)$ complexity. The algorithm is slightly different. More specifically, Durstenfeld moves, at each iteration, the struck elements to the end of the list by swapping them with unstruck ones.

3.9.3 Optimizing compilation using GCC's -Ox

The default option is -OO which is used to reduce the compilation time. In case we use -O3 the compiler tries to reduce execution time, the execution speed and

the code size, whereas it increases the performance of the generated code and the compilation time. -O3 uses all the implementations-turn on of flags- of the -O1 and the -O2.[76]

Some of the extra flags that we use at the compile step are (explanations have mainly taken by online GCC Option manuals [76]):

- -funroll-all-loops
  At compile time or upon the entry of a loop it can detect the number of iteration of loops that can be unrolled. If it is used correctly it can make our code larger but faster, otherwise, the code will run slower than the naive one.

- -fsched-pressure
  It can improve code and reduce its size by preventing register pressure increase above the number of available hard registers and subsequent spills in register allocation. Scheduling should be enabled before register allocation's activation.

- -fselective-scheduling, -fselective-scheduling2, -fsel-sched-pipelining, -fsel-sched-pipelining-outer-loops
  The -fselective-scheduling or -fselective-scheduling2 must be turned on, otherwise, it has no effect to the code. As well as, if -fsel-sched-pipelining is not turned on the -fsel-sched-pipelining-outer-loops that pipeline outer loops has also no effect.

- -ftree-vectorize, -ffast-math, -fassociative-math , funsafe-math-optimizations
  It is turned on under -O3 and it performs vectorization on trees. As well as, to enable vectorization of floating point reductions we used -ffast-math or -fassociative-math.
  The -ffast-math sets -fno-math-errno, -funsafe-math-optimizations, -ffinite-math-only, -fno-rounding-math, -fno-signaling-nans and -fcx-limited-range and it yields a faster code.
  The -fassociative-math allows the re-association of operands in a series of floating-point operations. Hence, it may also reorder floating-point comparisons and thus may not be used when ordered comparisons are required. This option requires that both -fno-signed-zeros and -fno-trapping-math be in effect. Moreover, it doesn't make much sense with -frounding-math.

- -frename-registers
  It is enabled by default with -funroll-loops. It attempts to avoid false dependencies in scheduled code by making use of registers left over after register allocation. The processors with many registers are the ones that benefit at most.

- -fprefetch-loop-arrays

  If it is supported by the target machine, it generates instructions to prefetch memory to improve the performance of loops that access large arrays. Therefore, this option it is possible to generate better or worse code. In other words, the results are highly dependent on the structure of loops within the source code.

- -march=native

  It is an option tells GCC that it should produce code for a certain kind of CPU. This flag can provide massive speedups for numerically-intensive code, but it is specialized for the hardware architecture that it was compiled.

- -flto

  It creates an output that can be optimized later, at link-time. If we specify the optional n, the optimization and code generation done at link time is executed in parallel using n parallel jobs by utilizing an installed make program.

- -D_GLIBCXX_PARALLEL

  It is used after the flags to convert all use of the standard (sequential) algorithms to the appropriate parallel equivalents. It also may change the sizes and behavior of standard class templates that are included in detail in gcc manual.

- -Wno-write-strings

  Do not warn the user when compiling C, give string constants the type const char[length] so that copying the address of one into a non-const char * pointer which normally produces a warning.

- -fomit-frame-pointer

  It is enabled by -03. Using this option, we don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions.

- -fsingle-precision-constant

  We treat the floating point constant as single precision constant instead of implicitly converting it to double precision constant.

- -frounding-math

  It is an experimental option that disables all GCC optimizations that are affected by rounding mode. It will disable the transformations and optimizations that assume default floating-point rounding behavior, As it is referred in GCC Option's manual, this option disables constant folding of floating-point

expressions at compile-time (which may be affected by rounding mode) and arithmetic transformations that are unsafe in the presence of sign-dependent rounding modes. This is round-to-zero for all floating-point to integer conversions and round-to-nearest for all other arithmetic truncations.

- -ffinite-math-only
  It yield faster code while it allows optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or +-Infs.

- -fno-trapping-math
  While we compile code, we can assume that floating-point operations cannot generate user-visible traps, like division by zero, overflow, underflow, inexact result and invalid operation

- -fno-signed-zeros
  It allows optimizations for floating-point arithmetic that ignore the signedness of zero and implies that the sign of a zero result isn't significant.

- -freciprocal-math
  It allows the reciprocal of a value to be used instead of dividing by the value.

- -fopenmp, -lpthread
  They are used to enable the OpenMP librariy functionalities

- -fvpt, -fprofile-values
  They are used combining with others to profile the code.

There are a plethora of flags that the user can use and all of them are listed in GNU's GCC guide, however, their usage should be carefully considered because they can affect negatively our results.

Instead of, it is not a flag/GCC Option to consider it in this subsection, but a Linux resource value, it is necessary before we run our application to increase stack memory via "ulimit -s". There are 2 options. On the one hand, there is the command "ulimit -s unlimited" to set the stack size unlimited (unlimited value depends on the operating system), while the program allocates more and more space to maintain the inner values/arrays. On the other hand, you can determine the amount of stack you need, using a tool like Valgrind[77] and set it by hand (ie. "ulimit -s 16777216" which is nearly 2 GB). If it will not increase the program throws Segmentation fault, because it runs out of memory.

3.9.4 Cache Blocking

Cache Blocking is an efficient and helpful optimization that we used mainly in transpose, copy and matrix multiplication in order to reduce cache misses and the total time of each epoch and generally the whole program. Furthermore, it is a way to increase spatial and temporal locality of reference (i.e. exploit full cache lines, to improve data reuse). Hence, it is more beneficial when we deal with multi-dimensional arrays and it may be faster than naive one observe it by testing it or by cache analysis. While we load a small subset cache block of a much larger data set, the memory bandwidth pressure reduces, because of the spatial locality and we avoid possibly memory bandwidth bottlenecks. Blocking (tiling) is an approach that many matrix algorithms can adopt efficiently [78]. Matrix multiplication and some transpositions made according to cache blocking theory that improves the run time of out CNN.

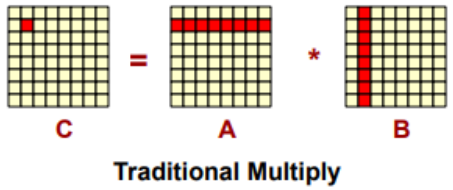


Figure 3.11: Matrix Multiplication process without(/before) Cache Blocking [79].

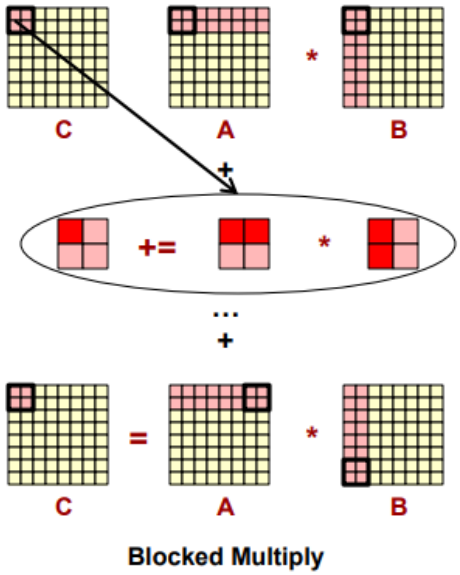


Figure 3.12: Illustration of Matrix Multiplication process, using Cache Blocking to take advantages of cache locality [79].

Algorithm 10 (Full) Cache Blocking for Matrix Multiplication and addition

Input:A[i_dim][j_dim] , B[j_dim][k_dim], C[i_dim]
Output:O[i_dim][k_dim]

```
1: for int ii=0, ii < i_dim, ii+ = block_size do
2:    max_i2 = ii + block_size < i_dim ? ii + block_size : i_dim;
3:    for int i=ii, i < max_i2, i + + do
4:       for int kk=0, kk < k_dim, kk+ = block_size do
5:          max_k2 = kk + block_size < k_dim ? kk + block_size : k_dim;
6:          for int k=kk, k < max_k2, kk + + do
7:             muladd = C[i];
8:             for int jj=0, jj < j_dim, jj+ = block_size do
9:                max_j2 = jj + block_size < j_dim ? jj + block_size : j_dim;
10:               for int j=jj, j < max_j2, jj + + do
11:                  muladd += A [i][j] * B [j][k];
12:               end for
13:               O[i][k] = muladd;
14:            end for
15:         end for
16:      end for
17:   end for
18: end for
```

Another interesting function that we attempted to implement using cache blocking was the transpose of a 4D array. The general idea of it sums up in the algorithm below. The block size was chosen to be 64, which is equal with the cache coherence size of our server.

Algorithm 11 (Full) Cache Blocking to transpose 4D array

Input:IN[i_dim][j_dim],[k_dim][l_dim]
Output:OUT[l_dim][i_dim],[j_dim][k_dim]

```
1: for int ii=0, ii < i_dim, ii+ = block_size do
2:   max_i2 = ii + block_size < i_dim ? ii + block_size : i_dim;
3:   for int i=ii, i < max_i2, i + + do
4:     for int jj=0, jj < j_dim, jj+ = block_size do
5:       max_j2 = jj + block_size < j_dim ? jj + block_size : j_dim;
6:       for int j=jj, j < max_j2, j + + do
7:         for int kk=0, kk < k_dim, kk+ = block_size do
8:           max_k2 = kk + block_size < k_dim ? kk + block_size : k_dim;
9:           for int k=kk, k < max_k2, k + + do
10:            for int ll=0, ll < l_dim, ll+ = block_size do
11:              max_l2 = ll + block_size < l_dim ? ll + block_size : l_dim;
12:              for int l=ll, l < max_l2, l + + do
13:                OUT[l][i][j][k] = IN[i][j][k][l] ;
14:              end for
15:            end for
16:          end for
17:        end for
18:      end for
19:    end for
20:  end for
21: end for
```

We also implemented cache blocking in functions that copy 4D arrays to get faster results, exploited the locality of cache blocking. Block size is set to 64 like the cache coherence size.

Algorithm 12 (Full) Cache Blocking to copy 4D array

Input:IN[i_dim][j_dim],[k_dim][l_dim]
Output:OUT[i_dim][j_dim],[k_dim][l_dim]

```
1: for int ii=0, ii < i_dim, ii+ = block_size do
2:   max_i2 = ii + block_size < i_dim ? ii + block_size : i_dim;
3:   for int i=ii, i < max_i2, i + + do
4:     for int jj=0, jj < j_dim, jj+ = block_size do
5:       max_j2 = jj + block_size < j_dim ? jj + block_size : j_dim;
6:       for int j=jj, j < max_j2, j + + do
7:         for int kk=0, kk < k_dim, kk+ = block_size do
8:           max_k2 = kk + block_size < k_dim ? kk + block_size : k_dim;
9:           for int k=kk, k < max_k2, k + + do
10:            for int ll=0, ll < l_dim, ll+ = block_size do
11:              max_l2 = ll + block_size < l_dim ? ll + block_size : l_dim;
12:              for int l=ll, l < max_l2, l + + do
13:                OUT[i][j][k][l] = IN[i][j][k][l] ;
14:              end for
15:            end for
16:          end for
17:        end for
18:      end for
19:    end for
20:  end for
21: end for
```

3.9.5 OpenMP

OpenMP (Open Multi-Processing) is a library (omp.h) that supports multi-platform shared-memory parallel programming in C/C++ and Fortran. [80] Using OpenMP all threads share memory and data. There are sections in an OpenMP program that are sequential and sections that are parallel. More specifically, at run-time, an OpenMP program will use one thread for the sequential sections, and several threads for the parallel sections. In terminology, there are master and slave threads. Slave threads are the parallel sections that cause additional threads to fork, while the master thread runs from the start to the end. The reason that we use OpenMP is that it allows the programmer to transform her/his code to a lower-level parallel and more efficient from a naive version.

There are some environment variables that we set to maximize performance [81]:

- OMP_WAIT_POLICY=active
  When it is active, it encourages idle threads to spin rather than sleep

- OMP_DYNAMIC=false
  At false state, we use it to do not let the run-time deliver fewer threads than we asked for.

- OMP_STATIC=true
  At true state, we use it to let the run-time deliver a specific number of threads to run our program.

- OMP_PROC_BIND=spread
  It specifies whether threads may be moved between processors and by choosing SPREAD option, a sparse distribution across the place partitions are used.

- OMP_NESTED=true
  It enables nested parallelism.

- OMP_SCHEDULE=static
  It allows specifying schedule type and chunk size. We decide to choose static allocation as it is measured as more efficient.

- OMP_PLACES=sockets
  Automatically generate masks binding tasks on the sockets of the CPUs that we have been allocated to the job. We may have sub-optimal binding when the number of tasks differs from the number of allocated sockets.

- OMP_NUM_THREAD=N
  We set the number of threads we wish. N is depending on the system's specifications. We have a machine with 40 threads, so we will use the maximum possible threads that will give us faster run execution time.

- GOMP_SPINCOUNT=infinite
  Always threads wait actively with consuming CPU power. It defines the number of busy cycles.

- OMP_STACKSIZE=300M
  We set the default thread stack size to 300M(Megabyte) of threads created by the OpenMP run time.

Algorithm 13 Matrix Multiplication and addition with OpenMP

Input:A[i_dim][j_dim] , B[j_dim][k_dim], C[i_dim]
Output:O[i_dim][k_dim]

```
1: #pragma omp parallel for default(none) shared(A, B, C, O)
   schedule (static, prmts::sch_static) private( muladd)
2: for int i=0, i < i_size, i++ do
3:    for int k=0, k < k_size, k++ do
4:       muladd = C[i];
5:       for int j=0, j < j_size, j++ do
6:          muladd += A [i][j] * B [j][k];
7:       end for
8:       O[i][k] = muladd;
9:    end for
10: end for
```

We implemented the matrix multiplication and we tested many versions of it, until we choose the fastest one. Chunk size in scheduling was decided to set as 16, because we have float(4-bytes) values and the cache line is 64bytes (4*16), hence it is a way to prevent false sharing due to multi-threading. The fastest was an hybrid-version of cache blocking and OpenMP implementation that in general follows the GEBP logic[82] :

Algorithm 14 (Partial) Cache Blocking for Matrix Multiplication with OpenMP

Input:A[i_dim][j_dim] , B[j_dim][k_dim], C[i_dim]
Output:O[i_dim][k_dim]

```
1: for int ii=0, ii < i_dim, ii+ = block_size do
2:    for int i=ii, i < min(ii + block_size, i_dim), i++ do
3:       #pragma omp parallel for default(none) shared(A, B, C, O, i)
         private( muladd)
4:       for int k=0, k < k_size, k++ do
5:          muladd = C[i];
6:          for int j=0, j < j_size, j++ do
7:             muladd += A [i][j] * B [j][k];
8:          end for
9:          O[i][k] = muladd;
10:      end for
11:   end for
12: end for
```

We also tested this technique to functions that implemented transpose of the arrays. Their results were depended on the size of the arrays and the read/writes of the memory blocks. Hence, for 4D and 3D arrays, we have better performance in time following the same pattern of one cache blocking dimension and the rest of them speeded-up with OpenMP. Although the cache blocking dimension is not static, it is depending on the current array we want to transpose every time.

Algorithm 15 Transpose with OpenMP (ie. transpose 4D array in forward function of Convolutional layer)

Input:IN[i_dim][j_dim][k_dim][l_dim]
Output:OUT[l_dim][i_dim][j_dim][k_dim]

```
1: #pragma omp parallel for default(none) shared(IN, OUT)
2: for int i=0, i < i_size, i++ do
3:    for int j=0, j < j_size, j++ do
4:       for int k=0, k < k_size, k++ do
5:          for int l=0, l < l_size, l++ do
6:             OUT[l][i][j][k] = IN[i][j][k][l];
7:          end for
8:       end for
9:    end for
10: end for
```

Algorithm 16 (Partial) Cache Blocking for Transpose with OMP (ie. transpose 4D array in forward function of Convolutional layer)

Input:IN[i_dim][j_dim][k_dim][l_dim]
Output:OUT[l_dim][i_dim][j_dim][k_dim]

```
1: for int ii=0, ii < i_dim, ii+ = block_size do
2:    max_i2 = ii + block_size < i_dim?ii + block_size : i_dim;
3:    for int i=ii, i < max_i2, i++ do
4:       #pragma omp parallel for default(none) shared(i, IN, OUT)
5:       for int j=0, j < j_dim, j++ do
6:          for int k=0, k < k_dim, k++ do
7:             for int l=0, l < l_dim, l++ do
8:                OUT[l][i][j][k] = IN[i][j][k][l];
9:             end for
10:         end for
11:      end for
12:   end for
13: end for
```

3.9.6 Limitations and Problems

During the implementation of the framework we dealt with various problems, the majority of them resolved, while some of them will be fixed in the future.

Limitations

A major limitation that we faced, was the change of stack size. We had to set "**ulimit -s unlimited**" or "**ulimit -s 16777216**" in order to get results instead of Segmentation fault at run. Our implementation creates a sequence of layers inside a namespace that it is used to pass the layers(arguments) to variadic templates. The purpose of that namespace is that the input and output of the layers are called by variadic templates in a recursive manner that facilitates their connection and collects them in one place to make some minor changes in a ("kind-of") user-friendly network layer's (architecture) representation. Hence, the namespace allocates a lot of stack space. The initial stack size is depended on the system and can be changed by the commands above, otherwise, it will keep gobbling up RAM for the program's needs in stack size until the system runs out of memory entirely and it will throw Segmentation fault. Therefore, we could increase more the stack size, but it was not necessary for the current implementation and for our tests.

Problems

There were many problems, during the implementation of the framework. We wasted a lot of time with various problems as the aforementioned stack size, a close to Python implementation layer's connection way that dealt with variadic templates, the C++17 support from system's GCC while Vivado SDK uses the system's compiler, hence it's version that was not supported in Server, the Python vs C++ loss in precision for calculations near-zero (while we verified one-to-one the results inside the arrays for specific values), etc. Most of the problems encountered, while false sharing, further hot-spots and bottlenecks, and some minor problems will be fixed in a future version of the framework. False sharing observed while we wanted to use more than half of server's cores and we got worse results. We will try to fix it with padding and spacing of the results in association with the system's size of cache line (future work).

3.9.7 Profiling

The last subsection of the second chapter includes frameworks and tools that analyses and profile our code, in order to find possible problems or bugs or to take some measurements of our implementation.

- **Perf tool**

  Perf is a tool in Linux which provides us an analysis of a program's performance. Using Perf we could mine various information that helped us further improve our code and detect problems. The last version of our code get the following statistics:

```
747.06user 2.06system 0:37.49elapsed 1998%CPU (0avgtext+0avgdata 356304maxresident)k
0inputs+16outputs (0major+80950minor)pagefaults 0swaps

 Performance counter stats for 'time ./test_cnn +RTS -s':

     749098.325420      task-clock (msec)         #   19.979 CPUs utilized
           12,543      context-switches          #    0.017 K/sec
               46      cpu-migrations            #    0.000 K/sec
           81,100      page-faults               #    0.108 K/sec
1,783,575,220,826      cycles                    #    2.381 GHz                      (30.77%)
  982,121,365,046      instructions              #    0.55  insn per cycle           (38.46%)
  139,866,301,334      branches                  #  186.713 M/sec                    (38.46%)
    1,272,476,486      branch-misses             #    0.91% of all branches          (38.46%)
  330,162,016,954      L1-dcache-loads           #  440.746 M/sec                    (38.46%)
  139,683,180,749      L1-dcache-load-misses     #   42.31% of all L1-dcache hits    (38.46%)
   52,772,965,878      LLC-loads                 #   70.449 M/sec                    (30.77%)
      538,406,746      LLC-load-misses           #    2.04% of all LL-cache hits     (30.78%)
   <not supported>     L1-icache-loads
      178,402,004      L1-icache-load-misses                                        (30.77%)
  330,147,821,878      dTLB-loads                #  440.727 M/sec                    (30.77%)
    3,667,775,059      dTLB-load-misses          #    1.11% of all dTLB cache hits   (30.77%)
        1,379,377      iTLB-loads                #    0.002 M/sec                    (30.77%)
        4,094,761      iTLB-load-misses          #  296.86% of all iTLB cache hits   (30.77%)
   <not supported>     L1-dcache-prefetches
   <not supported>     L1-dcache-prefetch-misses

     37.495157598 seconds time elapsed
```

Figure 3.13: Perf Tool

- **Valgrind**

  Valgrind is a framework for building tools. Valgrind has various analysis tools to profile and check our program. We use its tools to detect memory leaks, memory debugging and profile the memory usage of our program.

  Cachegrind

  We used Cachegrind to get some statistics of I1, D1, and LL (last-level) caches (read and write) and investigate if we can improve further our code, decreasing

the miss rates, especially the read ones. It also informs us about the stack limit that we can use and further details of the cache like the associative sizes of the cache and the general cache configuration. The current version of our code get the following statistics:

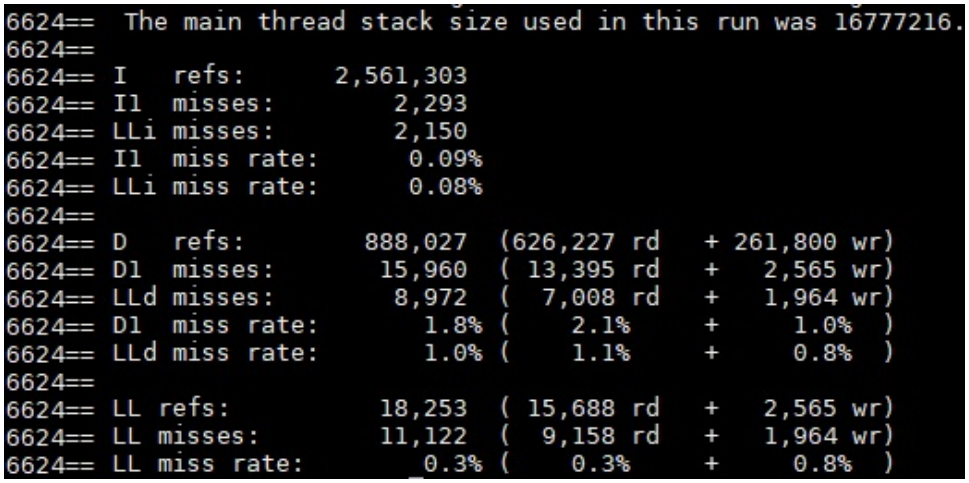


Figure 3.14: Cachegrind Tool

Memcheck

The most used and default tool of Valgrind is Memcheck. It can detect memory errors like incorrect free of the heap, mismatches of malloc-free pattern, memory leaks, undefined values, illegally access of memory, etc. The last version of our code informs us that we do not have any (common) error:

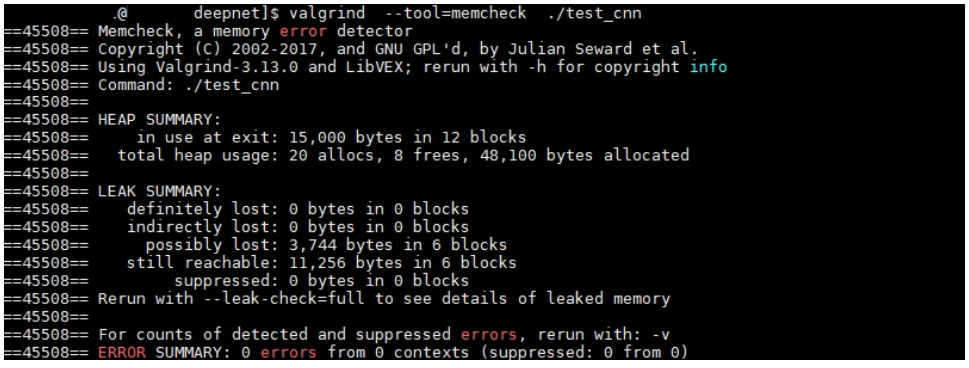


Figure 3.15: Memcheck Tool

Chapter 4

FPGA Accelerator Design and Implementation

4.1 Introduction

The fourth Chapter describe the specifications and some significant details of the Kronos Server that we used to develop our Framework (Section 4.1.1) and the Trenz Ultrascale+ platform that we used (Section 4.1.2). The next section (Section 4.2) refers to the Xilinx Vivado HLS tool and more specifically about the implementation of Matrix Multiplication with Addition, IP core, in Vivado HLS (Section 4.2). The design of the hardware application is produced by Vivado Design tool (Section 4.3). Following that, the CNN framework adjusts the IP Core's functionality in SDK (Section 4.4) There are some Optimizations, suggestions, and notes for the implementation of Matrix Multiplication (for the forward's Matrix Multiplication call of the Fully Connected layer) IP core. Finally, Section 4.5 describes some major problems and limitations of the final project.

4.1.1 Kronos Server Overview

Our implementation was tested and we took our measurements from the lab's server named Kronos. Kronos is a Dell Server with 40 threads of logical cores in CPUs(64-bit), x86_64 architecture, 2 threads of 10 Cores per socket. The CPU model is E5-2630 v4@ 2.20GHz (3.1 max GHz CPU). The cache memory of both L1d and L1i corresponds to 32KB, L2 to 256KB and L3 to 25600KB.

4.1.2 Trenz Platform Overview

An available and suitable for our project platform was a Trenz Electronic Starter Kit TE0808-04-09-2IE-S. There is a ZU9EG chip on the TEBF0808-04 module(Figure 4.3) that is connected to the carrier board and inside a black Core V1 Mini-ITX Enclosure, it contains a pre-assembled heatsink [6]. We are using the Zynq UltraScale+ Multi-Processor system-on-chip (MPSoC) that its family is based on the Xilinx UltraScale MPSoC architecture [83]. It is designed for the evaluation of the embedded applications with graphics, video, waveform, and packet processing

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
Stepping:              1
CPU MHz:               2198.281
CPU max MHz:           3100.0000
CPU min MHz:           1200.0000
BogoMIPS:              4394.93
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39
```

Figure 4.1: Intel CPU architecture information

providing significant power savings, programmable acceleration, and heterogeneous processing.

The Application Processing Unit (APU) of the Zynq UltraScale+ MPSoC includes a quad-core Arm Cortex-A53 MPCore processor(up to 1.5GHz, 64-bit, L1i 32KB, L1d 32KB, L2 1MB). Moreover, the Real-Time Processing Unit (RPU) contain a dual-core Arm Cortex-R5 (up to 600MHz, 32-bit Arm v7-R, L1i 32KB, L1d 32KB) based processing system (PS). There is also a Mali-400MP2 (64-bit, L2 64KB) GPU and Xilinx programmable logic (PL) UltraScale architecture with the programmable FPGA fabric in a single device. Furthermore, the board has an external memory of 4 GByte 64-Bit DDR4 SDRAM and 256KB On-Chip Memory. More details about the overview of the Zynq UltraScale+ MPSoC, there are in [83] of Xilinx documentation.

(a) Steel enclosure (b) Carrier board (c) TEBF0808-04 module

Figure 4.2: Trenz Platform [6]

4.2 High-Level Synthesis Design

The Vivado HLS tool synthesizes the matrix multiplication and addition within computation function written in C. The synthesized C is transformed into an IP block that can be used as an integrated module into the hardware system. The whole current hardware design and IP block made by Vasileios Amourgianos-Lorentzos.

The input/output data made from custom axi streams of 128-bit data by using the hls_stream library. The hls_stream is described as a multi-rate dataflow of 8-bit I/O and 32-bit data processing and decimation design using hls::stream in [84]. More specifically, the custom "AXI_STREAM128_KEEP" is used in 4 unrolled arrays of 4X4 2D dimension arrays (4x4 = 16 intermediate accumulations). Each unroll pass the 32-bit of data to/from an array and compute/process them to an output buffer before or after the matrix multiplication and addition computation that keeps the 16-bit (0xFFFF). The last 16-bits of write stream are dummy bits to latency the procedure and get the right result.

The synthesis reports for the matrix multiplication with addition functionality give us the following estimates:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	7.00	6.47	0.88

Latency (clock cycles)

Summary

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- BATCH_LOOP	?	?	?	-	-	?	no
+ OUT_LOOP	?	?	?	-	-	?	no
++ INIT_B_LAT_U	4	4	2	1	1	4	yes
++ INIT_B_LAT_U	4	4	2	1	1	4	yes
++ INIT_B_LAT_U	4	4	2	1	1	4	yes
++ INIT_B_LAT_U	4	4	2	1	1	4	yes
++ IN_PIPE	?	?	45	16	1	?	yes
++ OUT_S	16	16	5	4	1	4	yes
++ OUT_S	16	16	5	4	1	4	yes
++ OUT_S	16	16	5	4	1	4	yes
++ OUT_S	16	16	5	4	1	4	yes

Figure 4.3: HLS Report-Performance Results

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	6353	-
FIFO	-	-	-	-	-
Instance	0	352	25702	27016	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	25693	-
Register	0	-	79804	32	-
Total	0	352	105506	59094	0
Available	1824	2520	548160	274080	0
Utilization (%)	0	13	19	21	0

Figure 4.4: HLS Report-Utilization Results

4.3 Vivado Design

The design of the current implementation that has attached and uses the aforementioned IP core. They have been used 6 HP slave ports of 128bits width(every port has 2GB/s read bandwidth), therefore the total bandwidth was 10GB/s, not 12GB/s, possibly because it is the limit of the RAM's bandwidth. It is also implemented and configured by Vasileios Amourgianos Lorentzos and it is the following:

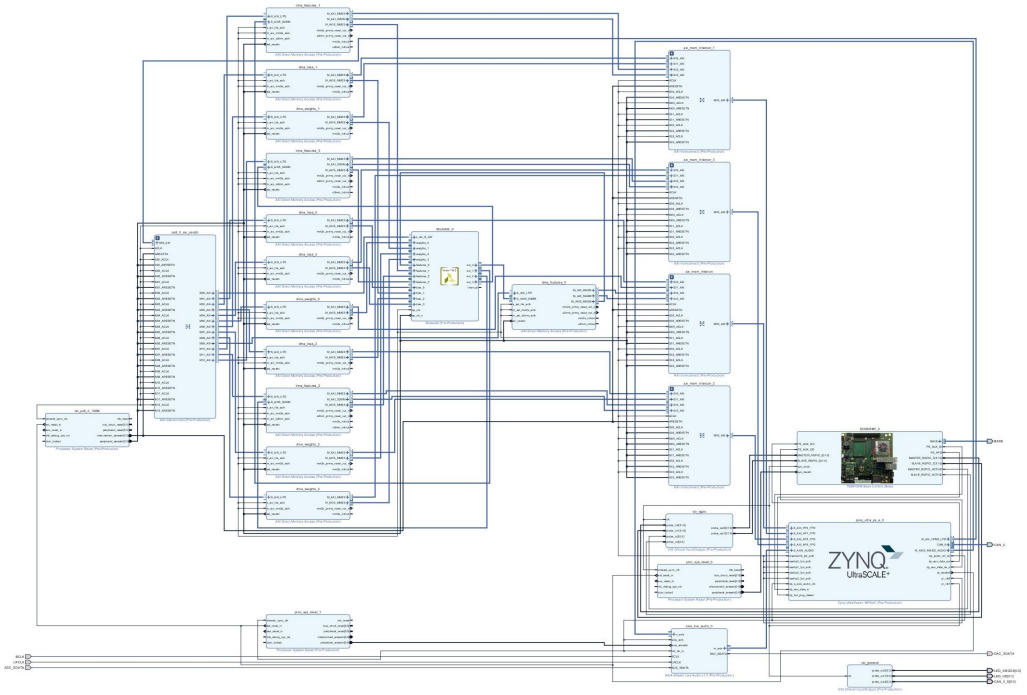


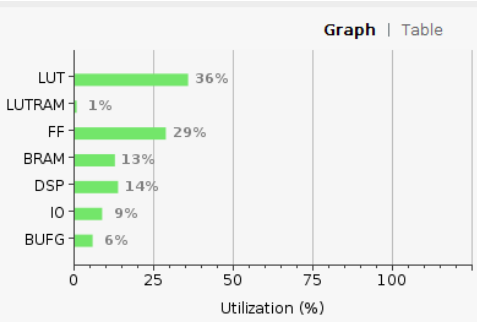
Figure 4.5: Block Design from Vivado Design

The reports of the implementation:

Graph | **Table**

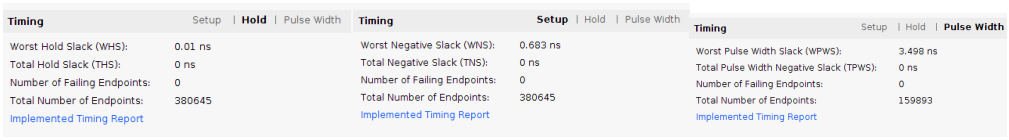
Resource	Utilization	Available	Utilization %
LUT	98987	274080	36.12
LUTRAM	1300	144000	0.90
FF	157677	548160	28.76
BRAM	120	912	13.16
DSP	352	2520	13.97
IO	18	204	8.82
BUFG	25	404	6.19

(a) LUTs Table



(b) Graph LUTs

Figure 4.6: LUTs



(a) Hold (b) Setup (c) Pulse Width

Figure 4.7: Timing

4.4 Vivado SDK

In this section, we used the hardware implementation, the IP core and the produced files (i.e. bitstream) to created our C++ project. Firstly, we create a C project that includes the First Stage Boot Loader (FSBL) for the ARM Cortex-A53 64-bit quad-core processor unit (APU). It has its own Board Support Package Settings (BSP) project that they are linked and includes the necessary libraries in C to boot. We were going to implement a bare-metal application, so we enabled the Xilinx secure key library, Xilinx Secure library, and Generic Fat File System Library in order to read from SD card, from Board Support Package Settings.

Then, we made the C++ project with its own BSP project that included also the aforementioned libraries and it was linked to hardware design. The Vivado SDK application uses the system's GCC compiler, that was useful because we need the C++17 version or above that, we already had installed from CYGWIN packages. After that, we import the code files for CNN. We also include the needed libraries from BSP to stdlibs.hpp file.

Before we compile the project, we should generate the linker script of the project. We made the following changes and use the whole DDR memory:

Available Memory Regions

Name	Base Address	Size
psu_ocm_ram_0_S_AXI_BASEADDR	0xFFFC0000	0x00029D00
psu_ocm_ram_1_S_AXI_BASEADDR	0xFFFE9E00	0x00000200
psu_ocm_ram_2_S_AXI_BASEADDR	0xFFFF0040	0x0000FDC0
DDR_BASEADDR	0x0	0x7fffffff

Stack and Heap Sizes

Stack Size 0x2000

Heap Size 0x7fff0000

Figure 4.8: Linker Script changes for FSBL

Available Memory Regions

Name	Base Address	Size
psu_ddr_0_MEM_0	0x0	0x80000000
psu_ocm_ram_0_MEM_0	0xFFFC0000	0x40000
psu_qspi_linear_0_MEM_0	0xC0000000	0x20000000
DDR_BASEADDR	0x0	0x7fffffff

Stack and Heap Sizes

Stack Size 0x65000000

Heap Size 0x10000000

Figure 4.9: Linker Script changes for CNN project

Available Memory Regions

Name	Base Address	Size
psu_ddr_0_MEM_0	0x0	0x80000000
psu_ocm_ram_0_MEM_0	0xFFFC0000	0x40000
psu_qspi_linear_0_MEM_0	0xC0000000	0x20000000
DDR_BASEADDR	0x0	0x7fffffff

Stack and Heap Sizes

Stack Size 0x70000000

Heap Size 0x0ffffff

Section to Memory Region Mapping

Figure 4.10: Linker Script changes for CNN project (up to 256 batch size)

We increased the stack, in accordance with the server's limits of stack size to be able to run our project without problems.

Optimizations

We made various optimization changes in the setting adding the following compiling flags:"-funroll-all-loops -fsched-pressure -fselective-scheduling -fsel-sched-pipelining -fsel-sched-pipelining-outer-loops -fprefetch-loop-arrays -ftree-vectorize -frename-registers -march=native -flto -Wno-write-strings -fvpt -fprofile-values -freciprocal-math -fomit-frame-pointer -fsingle-precision-constant -frounding-math -fno-trapping-math -fno-signed-zeros -ffinite-math-only -ffast-math -fno-signed-zeros -fno-trapping-math -fassociative-math -freciprocal-math -funsafe-math-optimizations -std=c++17 " and the optimization level to switched to -O3.

Apart from them, we also use most of the optimizations that we made for the software implementation.

Read Files

In order to read dataset, we made a lot of changes in Load_MNIST class and functions, but we keep the arrays as they were declared. Hence, the rest code that used the dataset's arrays remained unchanged. The functions within Load_MNIST class acquired the C-logic of Vasileios Amourgianos-Lorentzos code that was linked with the "ff.h" library.

Stream functions

The Stream functions did not change, therefore we use activation layers instead of activation functions so they were disabled and the stream functions moved to "numpy.hpp" file. The current HLS code will be initialized and get values/connects with the FPGA with AXI streams inside the appropriate functions.

Matrix Multiplication

The Fully-Connected Layer uses 3 fast ways to perform matrix multiplication. Firstly, we attached properly the FPGA accelerated function that needs 32*n size at the second dimension of the output array. Hence, we determine if it is less than 32 and we call the simple matrix multiplication function with partial cache blocking technique. If we do not use the FPGA acceleration we perform the full cache blocking method. The choices that we refer made by testing and observe the time that we get on every occasion.

Therefore, the 2D arrays had to be reshaped because the initial implementation of the accelerator was for 1D arrays. Hence, the total time is significantly increased, but it will be treated and eliminated appropriately in the immediate future.

Compile and Run

We build and compile the system with the C++ options that we referred to previously. Both of the FSBL and the CNN projects created their .elf file. Then, we create the boot file for SD Card to run our CNN in FPGA. The bootloader file is the fsbl.elf, the bitstream for the programmable logic portion (zynq_wrapper.bit), the system hardware project .hdf file (zynq_wrapper.hdf) and the last one is our CNN project named cnn.elf. The BOOT.ini and the dataset will be saved afterward in SD Card.

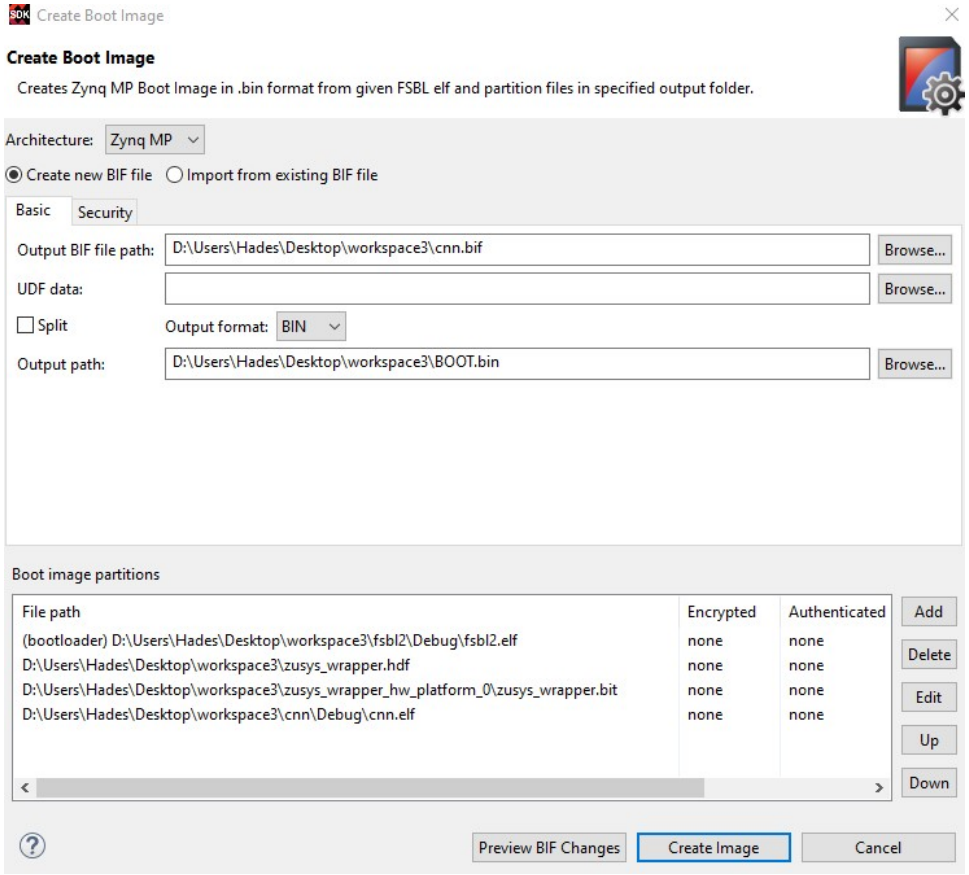


Figure 4.11: Create boot image for SD Card

After, we inserted the SD Card in the appropriate input slot and open the Trenz. We connect via PuTTY to the serial port while we start the FPGA and clear the memory from the buttons on the side of the Trenz enclosure box.

4.5 Limitations and Problems

We made had a lot of problems. Many of them were dealt with, but some are still under construction or will be encountered in the future. The main problem was the size of the memory of the FPGA. Heap and Stack's memory is increased, but it is tricky how to specify them in Linker Scripts or how to use them correctly in a bare-metal application.

Our limit is 2GB of DRR memory, Our implementation affords to load the whole dataset and keep it for further processing.

The matrix multiplication from HLS was not fully compatible. The best accuracy that we could get was about 0.5 which lead us to understand that was not implemented "correctly". It has a great speedup and it is implemented to get the maximum speedup for the current Trenz board, therefore it is not compatible neither in forwarding or backward function. Hence, we could get an estimated time of matrix multiplication function that it is calculated in the FPGA, resizing the weight array to get transposed array's sizes in forwarding function. Then we can decide if it is worthy to implement in the future (future work) again HLS code for the matrix multiplication function getting some measurements in the next chapter.

Update Weights and Regularization

The current IP core does not support weight update or weight regularization. This is a major drawback because it is faster to handle inside the FPGA the calculation instead of on the CPU. Therefore, it is part of the future work to make further development and optimizations.

Chapter 5

Evaluation and Results

5.1 Introduction

The fifth Chapter includes the measurements and some observations about the results. More specifically, we take some measurements, using the LeNet-5 architecture as it is described in Section 5.1 and we observe the behaviour of the CNN on the server-CPU, given a variety of mini-batch sizes and thread number. Following that, the next Section 5.2 contains some optimization and research results. More precisely, Subsection 5.2.1 have time performances of Matrix Multiplication with Addition on the server's CPU, the Trenz's CPU(ARM) and the Trenz's FPGA(connected with an ARM CPU). The other two Subsections have the measurements from transpose (Subsection 5.2.2) and copy (Subsection 5.2.3) that performed only on CPU's. Last Section 5.3 contains measurements about power consumption and generic measurements/calculations of the system's performance on Server and on Trenz Platform.

5.2 CNN Performance

Layers

We attempt to take some measurements given the following parameters and LeNet-5 architecture.

Layers	Feature Map	Size	Kernel Size	Stride	Padding
Convolutional	6	28x28	5x5	1	2
ReLU	-	-	-	-	-
(/Tanh)	-	-	-	-	-
Maxpool	6	14x14	2x2	2	-
ReLU	-	-	-	-	-
(/Tanh)	-	-	-	-	-
Convolutional	16	10x10	5x5	1	2
ReLU	-	-	-	-	-
(/Tanh)	-	-	-	-	-
Maxpool	16	5x5	2x2	2	-
ReLU	-	-	-	-	-
(/Tanh)	-	-	-	-	-
Flatten	-	-	-	-	-
Fully Connected	-	128	-	-	-
ReLU_2d	-	-	-	-	-
(/Tanh_2d)	-	-	-	-	-
Fully Connected	-	64	-	-	-
ReLU_2d	-	-	-	-	-
(/Tanh_2d)	-	-	-	-	-
Fully Connected	-	10	-	-	-

Table 5.1: LeNet-5 Architecture Table with Parameters

Time Performance of Threads - Layers

Our code uses OpenMP and Cache Blocking (Hybrid version), as we referred to in the third Chapter. We tested our code in multiple threads for a LeNet-5 with Tanh activation function for multiple threads.

Threads Time	1T (ms)	2T (ms)	4T (ms)	8T (ms)
train	189.940577	108.701242	61.325151	38.949878
train inference	61.594659	33.156489	18.549237	11.797975
test inference	10.17491	5.409514	3.025038	1.926279
Epoch Time	261.710266	147.267325	82.899524	52.674219

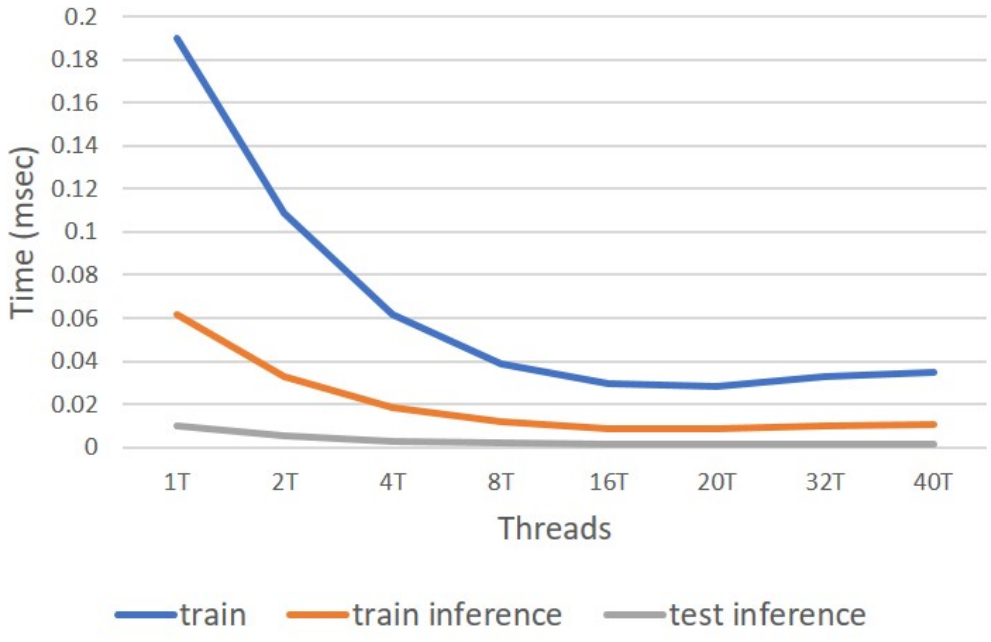
Table 5.2: Time performance of one run of LeNet-5 (Threads : 1, 2, 4, 8)- 1 epoch, 96 mini-batch size, Kronos CPU

Threads Time	16T (ms)	20T (ms)	32T (ms)	40T (ms)
train	29.67968	28.340208	33.017988	35.03828
train inference	8.713907	8.405577	9.663663	10.334649
test inference	1.454152	1.401746	1.569302	1.722189
Epoch Time	39.847824	38.147611	44.251044	47.095271

Table 5.3: Time performance of one run of LeNet-5 (Threads : 16, 20, 32, 40)- 1 epoch, 96 mini-batch size, Kronos CPU

We get the best performance in 20 Threads, possibly, due to Intel QuickPath Interconnect (QPI) overhead bus and maybe due False Sharing (multi-threading) in memory or even hotspots that we will investigate further in the future. Therefore, we notice that using 20 threads instead of one we perform 6.8x faster the training process, and 7.3x faster the inference processes. In total, we get results 6.9x faster (1 epoch) of the whole MNIST dataset.

We can see the reduction of time from the table, as illustrations, for different number of threads below :



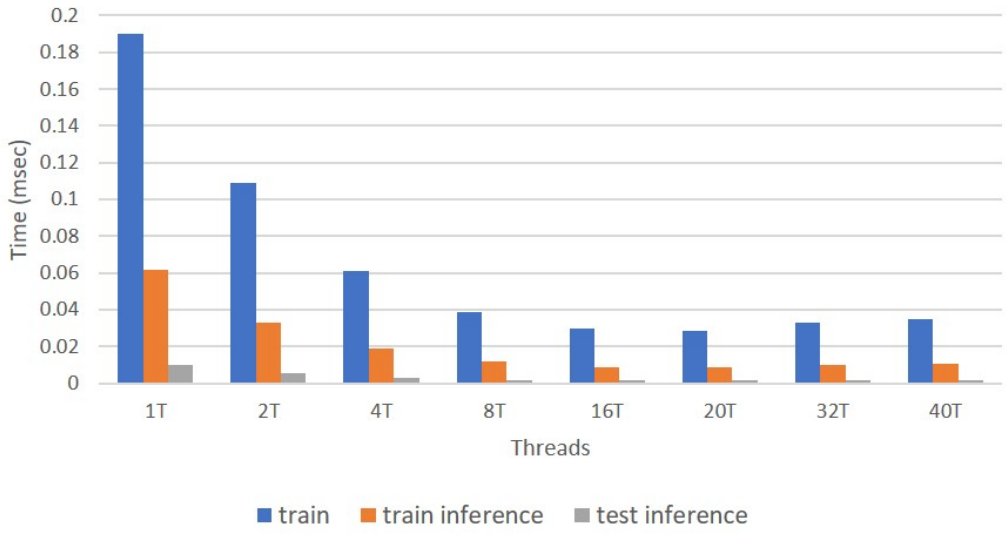


Figure 5.2: Time performance for T number of threads - Kronos CPU. Using 20 threads have the best time performance (fastest).

We can see from the above the summarized time performances for multiple threads that our CNN for LeNet-5 gets the fastest results using 20 threads (1 CPU).

While the graphs above show separately the train and inference times, we sum them as epoch times and we get the following graph :

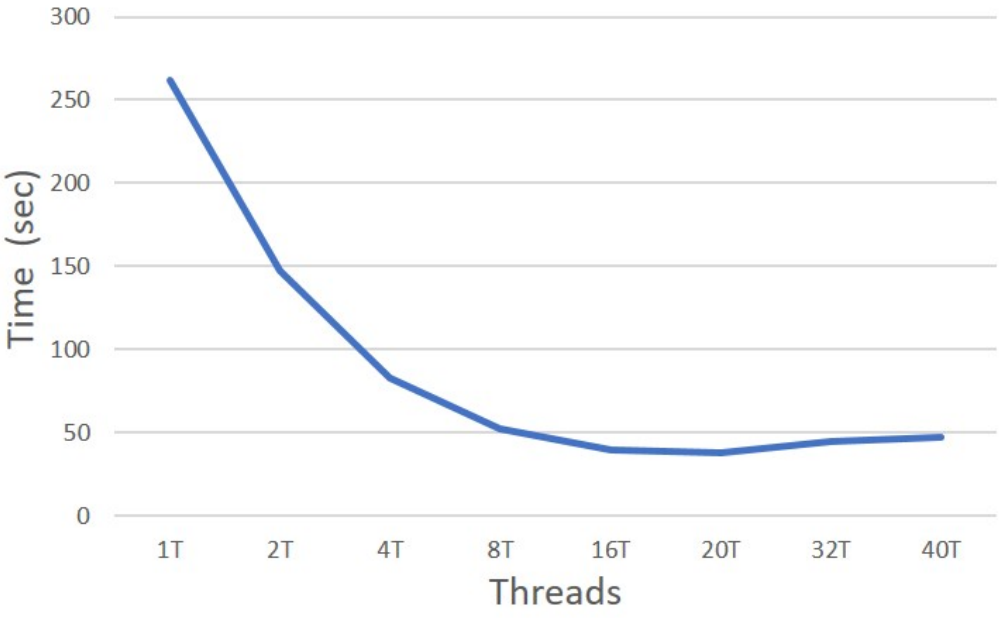


Figure 5.3: Epoch time for N number of threads - Kronos CPU, 96 mini-batch size

Until now, we have investigated the time performance for the train, train inference, test inference separately. Besides that, we want correct results, hence we collect the train and test accuracy, as well as the loss in each epoch and for different numbers of threads. The following table has the results from the run of the first epoch:

Threads	1T	2T	4T	8T
Loss	0.335403	0.427342	0.471495	0.431629
Training Accuracy	0.894568	0.882	0.891417	0.889917
Test Accuracy	0.8998	0.8848	0.9002	0.8968

Table 5.4: Loss and accuracy of one run of LeNet-5 (Threads : 1, 2, 4, 8)- 1 epoch, Kronos CPU

Threads	16T	20T	32T	40T
Loss	0.543014	0.438953	0.290006	0.314198
Training Accuracy	0.87955	0.886151	0.909484	0.892301
Test Accuracy	0.8847	0.893	0.9162	0.8996

Table 5.5: Loss and accuracy of one run of LeNet-5 (Threads : 16, 20, 32, 40)- 1 epoch, 96 mini-batch size

The aforementioned results can confirm to us that we have a great implementation and our system works correctly, as the accuracy is high, the loss reduces, hence the CNN learns during the epochs. We can not take into consideration the difference between the thread number because we use a shuffle of batches and each full run has a different "pack"/batch of images. The array can inform us that on a single thread or on multiple threads, our framework works fine.

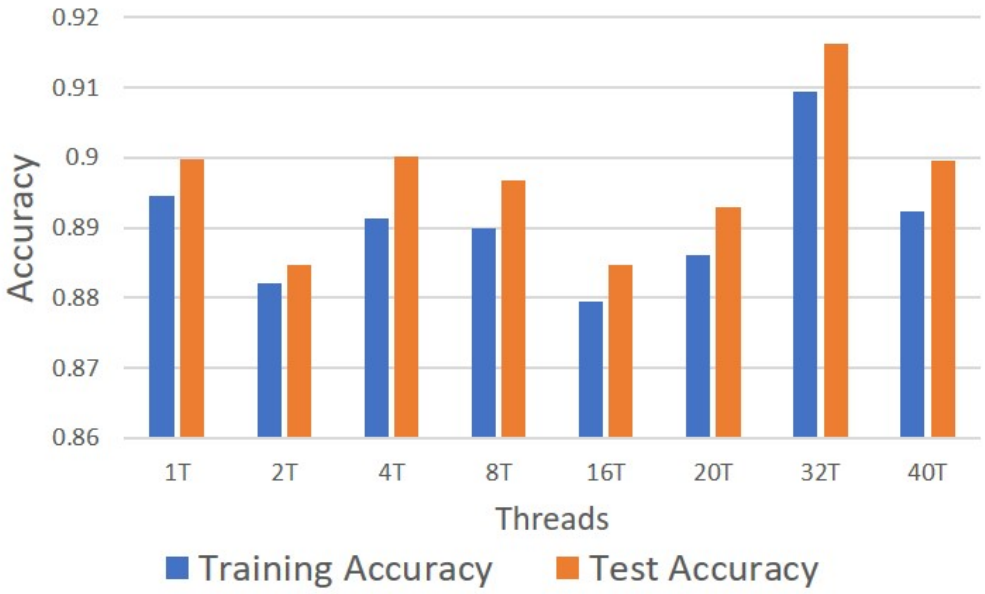


Figure 5.4: Train and Test accuracy for T number of threads - CPU

Using the implementation of 20 threads, we get the graph of the train and test accuracy and we can claim that the framework can be trained and learn efficiently. Each epoch contributes to the accuracy using the updated weights of the previous epoch and finally the weights regularize and update again, during the training procedure.

Figure 5.5: Train and Test accuracy in 15 epochs - Kronos CPU, 96 mini-batch size, 20 threads

The Loss and Loss reduction for every epoch for the learning/training process behaves normally, as expected in a CNN:

Figure 5.6: Loss and loss reduction in 15 epochs - Kronos CPU, 96 mini-batch size, 20 threads

The differences in times between the epochs and the training process, the training inference and the test inference are minor as it was supposed to. We have the same dataset size, same parameters and same architecture, so there is only a small difference in milliseconds that are caused from the calculations on floating point numbers, the round decisions in maths of the language and the processor, as well as, the usage of the GCC optimisation flags that we have chosen:

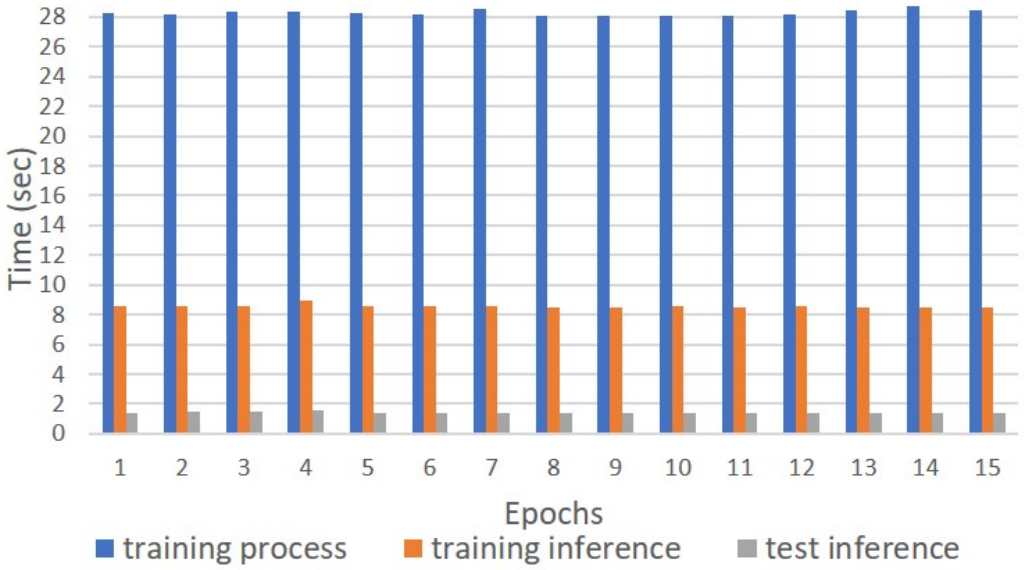


Figure 5.7: Train, train inference and test inference time in 15 epochs - Kronos CPU, 96 mini-batch size, 20 threads

Perf tool reports are helpful to understand our implementation better and investigate further ones. While the number of threads increases, GHz and instructions per cycle reduces. We have a deep network that can be translated into possible hotspots. The reduction of GHz can affect instructions per cycle if our instruction needs a longer clock cycle and the task-clock, increased instructions and cycle measurements can confirm it from their division.

Threads	1T	2T	4T	8T
task-clock (ms)	261370.5899	294294.0939	331530.8038	421872.6854
cycles	779,972,883,994	879,324,468,228	965,693,227,275	1,118,564,101,390
instructions	722,398,991,807	735,888,812,969	761,885,709,091	800,405,627,118
CPUs utilized	0.998	1.996	3.992	7.984
GHz	2.984	2.988	2.913	2.651
insn per cycle	0.93	0.84	0.79	0.72

Table 5.6: LeNet-5 Architecture Summary Table from Perf tool (1T, 2T, 4T, 8T), 96 mini-batch size

Threads	16T	20T	32T	40T
task-clock (ms)	639032.8252	765152.7211	1421466.091	1894648.708
cycles	1,521,813,247,524	1,821,955,797,444	3,378,137,285,778	4,503,765,226,912
instructions	915,177,025,022	974,813,552,887	1,301,284,247,229	1,551,440,241,224
CPUs utilized	15.977	19.971	31.949	39.895
GHz	2.381	2.381	2.377	2.377
insn per cycle	0.6	0.54	0.39	0.34

Table 5.7: LeNet-5 Architecture Summary Table from Perf Tool (16T,20T,32T,40T), Kronos CPU, 96 mini-batch size

While we increase the threads, we use we can assume that the dependencies/sharing in memory, the overhead of the bus and the reduced GHz that give us the multi-threading instead of a single thread usage, all of them affect our system by reducing the amount of instructions per cycle that we can perform. The aforementioned notices are shown below, clearly illustrated in graphs:

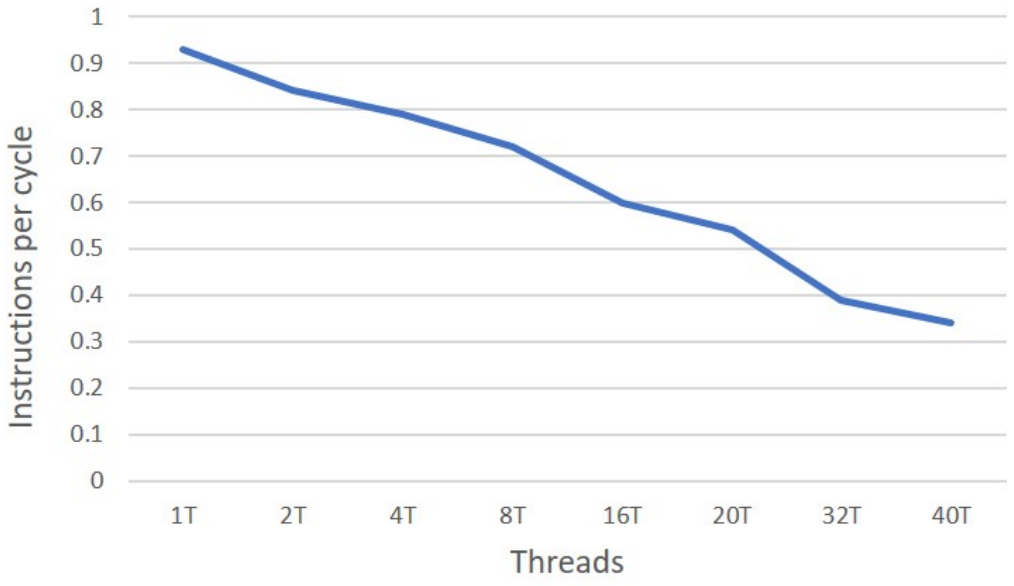


Figure 5.8: Instructions per cycle for N number of threads - Kronos CPU, 96 mini-batch size

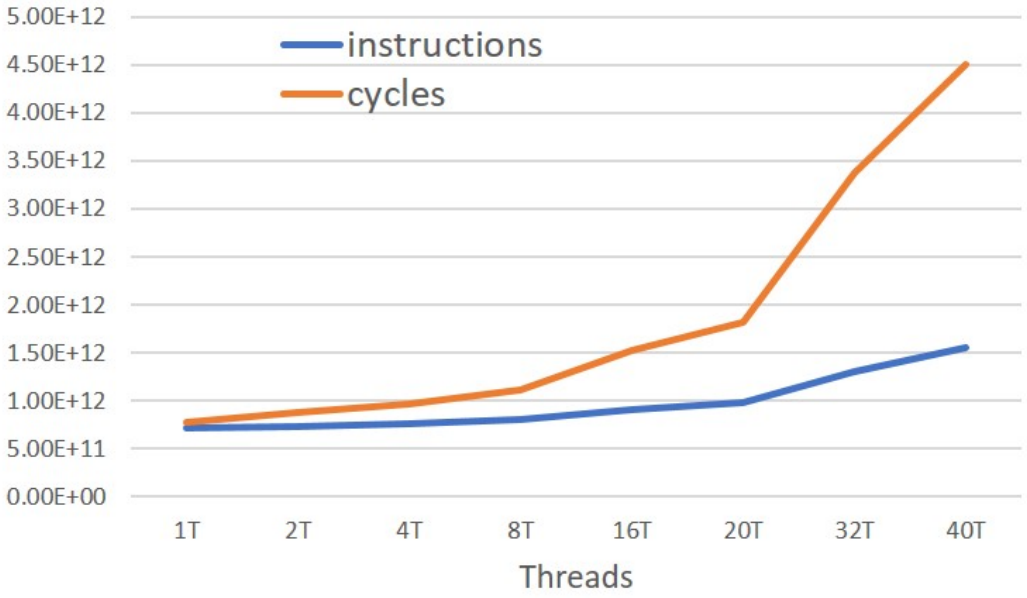


Figure 5.9: Instructions and clock cycles for N number of threads - Kronos CPU, 96 mini-batch size

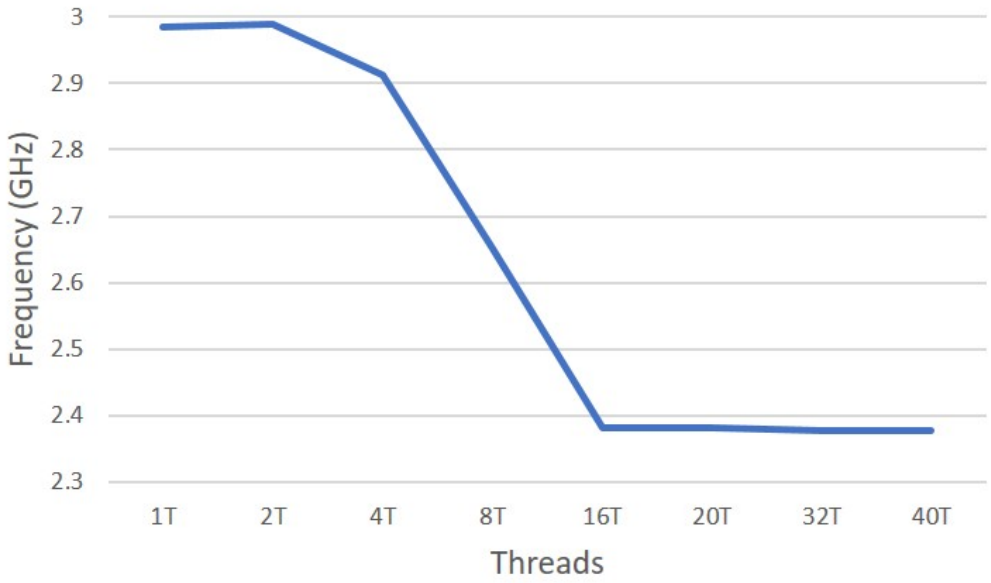


Figure 5.10: Clock rate for N number of threads - Kronos CPU, 96 mini-batch size

From the Valgrind framework, using the cachegrind tool, we can extract a report of cache misses. We want to reduce misses, as low as possible, especially read ones. We can claim that we have a system which can be improved at the future possibly, therefore all the miss percentages are less than 2.1% that indicates a cache optimized framework with great miss rates even at the current version.

I refs:	2,541,477			
I1 misses:	2,271			
LLi misses:	2,133			
I1 miss rate:	0.09%			
LLi miss rate:	0.08%			
D refs:	883,755	(622,613 rd	+	261,142 wr)
D1 misses:	15,837	(13,330 rd	+	2,507 wr)
LLd misses:	8,929	(7,002 rd	+	1,927 wr)
D1 miss rate:	1.80%	(2.1%	+	1.0%)
LLd miss rate:	1.00%	(1.1%	+	0.7%)
LL refs:	18,108	(15,601 rd	+	2,507 wr)
LL misses:	11,062	(9,135 rd	+	1,927 wr)
LL miss rate:	0.30%	(0.3%	+	0.7%)
cache-references:	77,475,922,687			
cache-misses:	3,590,753,132		4.635%	of all cache refs
instructions:	522,566,193,949			

Table 5.8: Valgrind (tool: cachegrind) report for LeNet-5 Architecture with ReLU activation - Kronos CPU, 96 mini-batch size

5.3 Optimizations and research

For the current section, we decided to improve time and examine ways to reduce "slow" in time and "heavy" in memory functions. Some of them where few matrix multiplication calls, transpose and copy functions and see their behavior using few/a lot of memory and their speedup in time using the algorithms from Section 3.9 or some variants.

We explored/observed the time performance of each chosen function, changing the number of threads and batch-sizes to find the best practice.

5.3.1 Matrix Multiplication

Time Performance of Matrix Multiplication with addition - Kronos CPU, 96 mini-batch size

We tested many versions of the Matrix Multiplication with Addition and we sum-up the time from the algorithms of the third Chapter (3.9) in the table below:

Matrix Mult. with addition	Naive	Cache-Blocking Algorithm	OpenMP Algorithm	Hybrid Algorithm
1st FC layer at LeNet-5 mini-batch size	Time (ms)	Time (ms)	Time (ms)	Time (ms)
128 images	26.520995	16.063599	4.020237	2.246671
96 images	20.234731	12.054891	2.808729	1.960531
64 images	13.493708	8.065859	2.056212	1.644697
32 images	6.942636	4.099342	1.005933	1.277977
16 images	3.640126	2.034923	0.479782	1.064858

Table 5.9: Execution time: Naive and Cache Blocking (1 thread), OpenMP and OpenMP with Cache Blocking/ Hybrid (20 threads) - Kronos CPU

We notice that OpenMP Algorithm, which is implemented with OpenMP without Cache blocking performs better in small batch sizes. The fastest train of a batch numerically have nearly 100 images, hence we prefer Hybrid Algorithm. Therefore, we could handle cases that modulo of mini-batches is less than 64 with OpenMP Algorithm to reduce (a little bit) the total time of the training process.

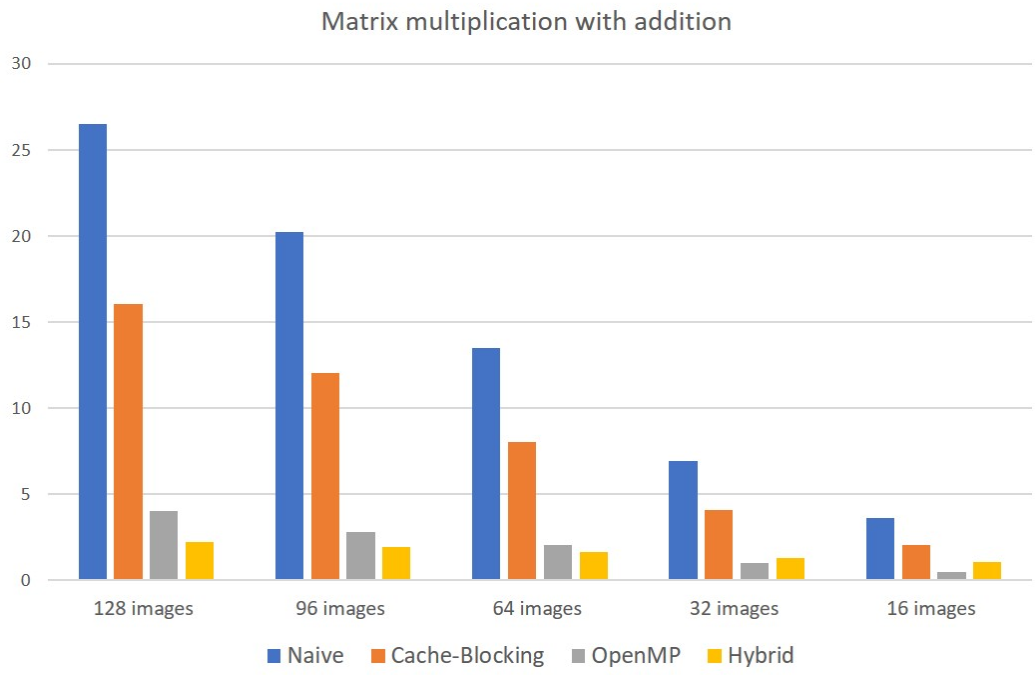


Figure 5.11: Execution time Matrix Multiplication with addition: Naive and Cache Blocking (1 thread), OpenMP and Hybrid (20 threads) - Kronos CPU

Furthermore, we made the graphs for all (3 Fully Connected Layers - FC) and for the rest of them as their numeric difference from the first one lead us to make a separate graph to understand their behavior in time changing the batch size of the number of the threads.

Even though, the Fc1 has the slowest processing time it has the best time in the FC3 for 256 batch size. The FC1 for a lot of data seems to handle better them in the smallest batch size, here for 16 batch size. Also, the FC3 fits better in 256 batch size, while the FC2 has the best time performance in 64 batch size. The size of input data should be considered, as well as, the average time of all the FC layers to get the best time performance. This behavior should be investigated further in the future.

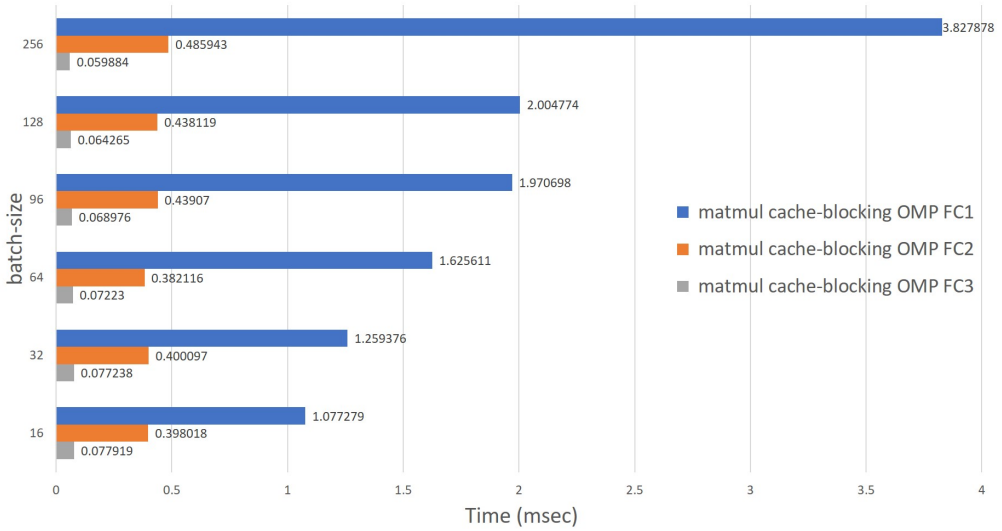


Figure 5.12: Matrix Multiplication with addition in FC1, FC2, FC3 layer for multiple batch-sizes - Kronos CPU (20 threads)

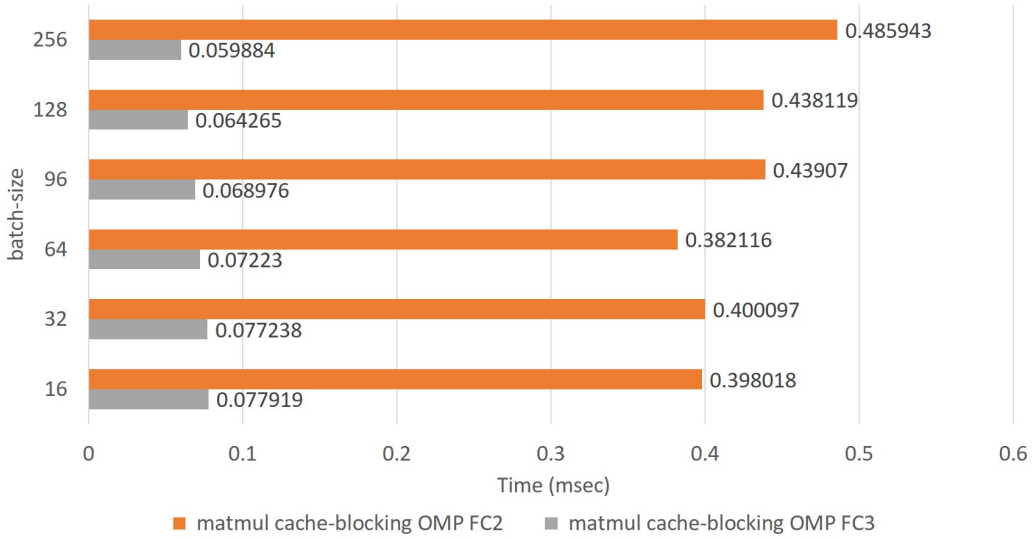


Figure 5.13: Matrix Multiplication with addition in FC2, FC3 layer for multiple batch-sizes - Kronos CPU (20 threads)

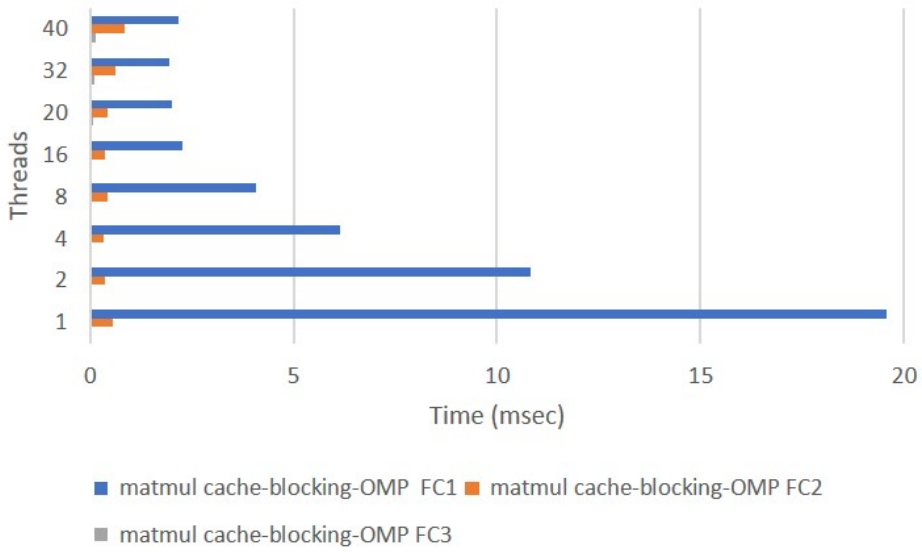


Figure 5.14: Matrix Multiplication with addition in FC1, FC2, FC3 layer for multiple number of threads - Kronos CPU (96 mini-batch size)

If we focous more in FC2 and FC3, we can clearer see the behaviour in time of the increasing usage of the server's threads:

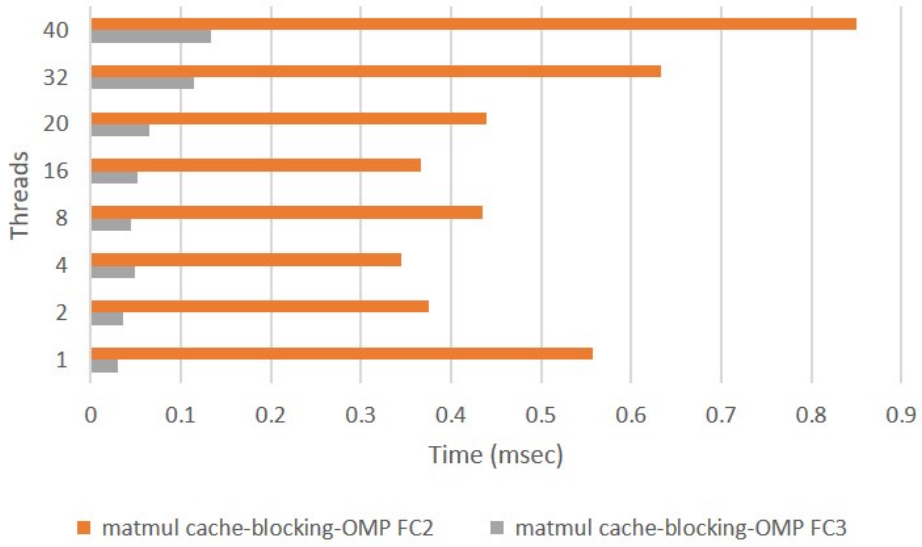


Figure 5.15: Matrix Multiplication with addition in FC2, FC3 layer for multiple number of threads - Kronos CPU (96 mini-batch size)

The choice of 20 threads gave us the best time performance. Therefore, if we look closely in FC2 and FC3 graph above, we can see that the FC layers need less data and using more than the needed number of threads lead us to worse time performance than we can get if we can not control them inside the Neural Network.

Time Performance of Matrix Multiplication with addition - FPGA

Using SDK tool and programming the FPGA via the SD card we can get some time results. All the following results of the FPGA have I/O time. More specifically, we refer as I/O the time of AXI DMA: RAM(/DDR) to FPGA and from FPGA back to RAM(/DDR). The dimensions of the arrays that the kernel has and the frameworks are not the same, so we reshape them to pass/get in/of the FPGA. We can see that changing the dimensions of the arrays, it requires a great amount of extra time, so we are planning to change 2D to 1D arrays in the future and it will also improve the cache locality.

Matrix Multiplication with addition	FPGA call	FPGA with ravel/unravel
1st FC layer at LeNet-5 mini-batch size	Time (ms)	Time (ms)
128 images	11.587082	18.528843
96 images	8.700692	15.554193
64 images	5.809801	12.579602
32 images	2.914501	12.408542
16 images	1.466760	12.410402

Table 5.10: Time performance for single thread Trenz FPGA without and with rave/unravel time

There is a huge different in time as we can observe. The ravel and unravel functions add to our system a lot of milliseconds in time and slow down the time performance of our system.

The following table includes the time differences in the Trenz's ARM CPU working in bare-metal. We can compare the Naive algorithm, the cache blocking algorithm and the Hybrid algorithm without the OpenMP support in the current version.

Matrix Multiplication with addition	Naive Algorithm	Cache-Blocking Algorithm	Hybrid (w.o. OMP)
3rd FC layer at LeNet-5 mini-batch size	Time (ms)	Time (ms)	Time (ms)
128 images	3.209881	4.077781	4.077631
96 images	3.210061	4.078081	4.077931
64 images	3.210121	4.078051	4.078321
32 images	3.210301	4.078321	4.078411
16 images	3.210601	4.078621	4.078501

Table 5.11: Time performance of matrix multiplication with addition in ARM CPU - single thread - 3rd FC

Naive algorithm, on the table above, seems to have better performance in the bare-metal, single thread ARM CPU, for small sizes of arrays.

Time Performance of Matrix Multiplication with addition - CPU - FPGA

We sum up the aforementioned times of the FPGA and CPU in a Table to show the differences in time:

Matrix Mult. with addition	Naive (w.o. OpenMP)	Cache-Blocking Algorithm	OpenMP Algorithm	Hybrid Algorithm	FPGA
1st FC layer mini-batch size	Time (ms)	Time (ms)	Time (ms)	Time (ms)	Time (ms)
128 images	26.520995	16.063599	4.020237	2.246671	3.131191
96 images	20.234731	12.054891	2.808729	1.960531	2.370420
64 images	13.493708	8.065859	2.056212	1.644697	1.612620
32 images	6.942636	4.099342	1.005933	1.277977	0.893010
16 images	3.640126	2.034923	0.479782	1.064858	0.468540

Table 5.12: Execution time: Naive and Cache Blocking (1 thread), OpenMP and OpenMP with Cache Blocking/ Hybrid (20 threads) - Kronos CPU - and bare-metal single thread ARM-FPGA

Matrix Mult. with addition	Naive (w.o. OpenMP)	Cache-Blocking Algorithm	OpenMP Algorithm	Hybrid Algorithm
1st FC layer mini-batch size	Speedup	Speedup	Speedup	Speedup
128 images	8.5	5.1	1.2	0.72
96 images	8.5	5.1	1.18	0.82
64 images	8.4	5	1.26	1.02
32 images	7.8	4.6	1.27	1.43
16 images	7.8	4.3	1.02	2.27

Table 5.13: Speed-up FPGA vs CPU: Naive and Cache Blocking (1 thread), OpenMP and OpenMP with Cache Blocking/ Hybrid (20 threads) - Kronos CPU- and bare-metal single thread ARM-FPGA

The framework in the bare-metal use not OpenMP, so we can compare the naive and cache block algorithms with the FPGA implementations. We attempted to compare it also with the ones with OpenMP, but it will be more accurate if we were using OpenMP and its environmental variables. We can assume that FPGAs have the best performance over Naive and Cache-blocking algorithm. Nevertheless, Hybrid Algorithm is faster than FPGA when we use batch-sizes with over 64 images with OpenMP using only one thread theoretically if we accept the aforementioned assumption. FPGAs have better results for batch-sizes of less than 32 images, OpenMP has an almost equal performance with FPGA for small batch sizes of less than 32 image, while Hybrid algorithm has the best speedup for batches over 64.

Claiming that we can not have an accurate comparison, we conclude that FPGA is 4.3x-5.1x times faster when we use cache blocking in CPU and 7.8x-8.5x times faster than a naive matrix multiplication code.

5.3.2 Transpose

Time Performance of Transpose - CPU

We tested some 4D and 2D versions of the Transpose functions, we chose the best/fastest of them and we take some measurements as shown below:

Transpose 4D	without OpenMP	OpenMP Algorithm	Cache-Blocking Algorithm	Hybrid Algorithm
mini-batch size	Time (ms)	Time (ms)	Time (ms)	Time (ms)
96 images	0.341460	0.091139	0.557041	0.236604

Table 5.14: Transpose execution time of 4D: Naive and Cache Blocking (1 thread), OpenMP (20 threads) - Kronos CPU

Transpose 2D	without OpenMP	OpenMP Algorithm	Cache-Blocking Algorithm
mini-batch size	Time (ms)	Time (ms)	Time (ms)
96 images FC bw	0.00033	0.010891	0.000735
96 images Conv bw	0.247517	0.006066	0.000424

Table 5.15: Transpose execution time of 2D array: Naive and Cache Blocking (1 thread), OpenMP (20 threads) - Kronos CPU

The transpose of a 2D array within Fully Connected Layer seems to have a 14.8x difference using cache blocking in a small array, while the transpose in Convolutional Layer has 14.3x, instead of using OpenMP. Without OpenMP for the small size of arrays, the speedup is even better 330x than time with OpenMP and 22x times faster than the cache blocking implementation. Therefore, OpenMP can handle better larger arrays, as we notice in transpose of the 4D array in Table 5.15. OpenMP is 2.5x times faster than Hybrid algorithm, 3.7x than without OpenMP implementation and 6.1x times faster than cache blocking which is expected, considering the L-cache size. The aforementioned observations are illustrated below:

The graphs below illustrate better the results that we get from all the cases except the one without OpenMP that it was to slow to take into consideration.

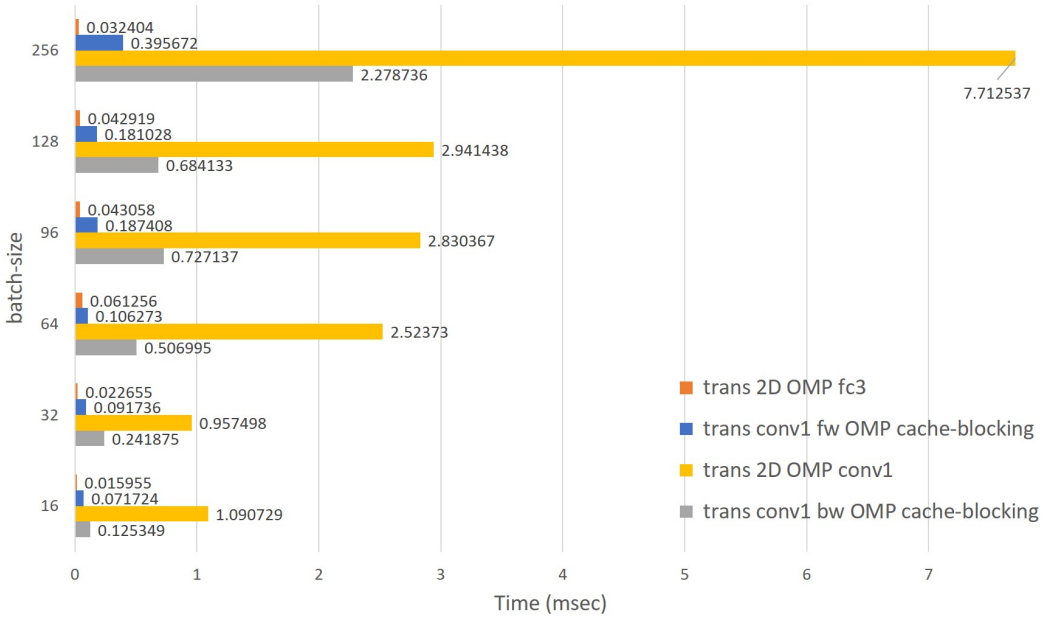


Figure 5.16: Transpose in 4D and 2D arrays for multiple batch-sizes- Kronos CPU (20 threads)

The graph above shows us that small batch sizes lead to faster transpose. Most of the times we have more than x2 time to perform a transpose in larger batches. Therefore, we can notice that for batch-size 32 the transpose of a large array with OMP is faster than that with 16 batch-size. Also, the transpose with hybrid algorithm is faster for 128 than 96 batch-size and if we multiply (128/16 = 8) 8 times x 0.125349 msec(16 batch size) ~= 1 that is slower than 0.684133 msec of 128 batch-size's measurement. The same happens with the other's if we compare them with x-times of the rest batch-sizes.

At the following graph, we notice that for large arrays with OpenMP(yellow bars) we get faster results at transpose, while small arrays with OpenMP(orange bars) are faster for just one thread (max GHz). We also can claim that 8 threads (all for 96 batch size) have the best performance in time for the hybrid algorithm in 4D arrays, as the handle better the cache of that amount of data (blue and gray bars).

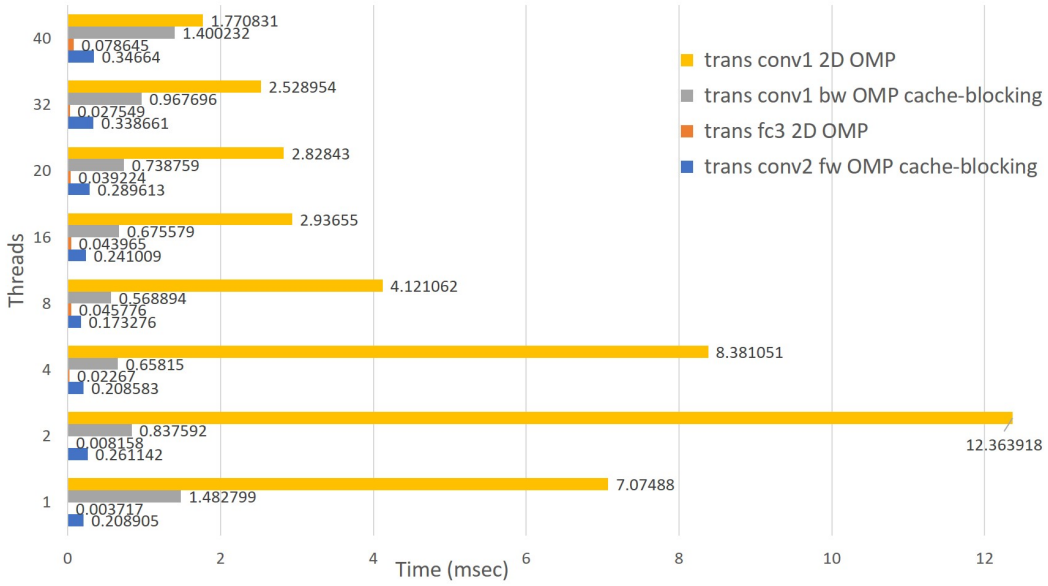


Figure 5.17: Transpose in 4D and 2D for multiple number of threads - Kronos CPU (96 mini-batch size)

A closer look of the chart below, it shows us a zoomed and clearer view of the transpose time from the chart above without the transpose in the first Convolutional layer between 2D arrays.

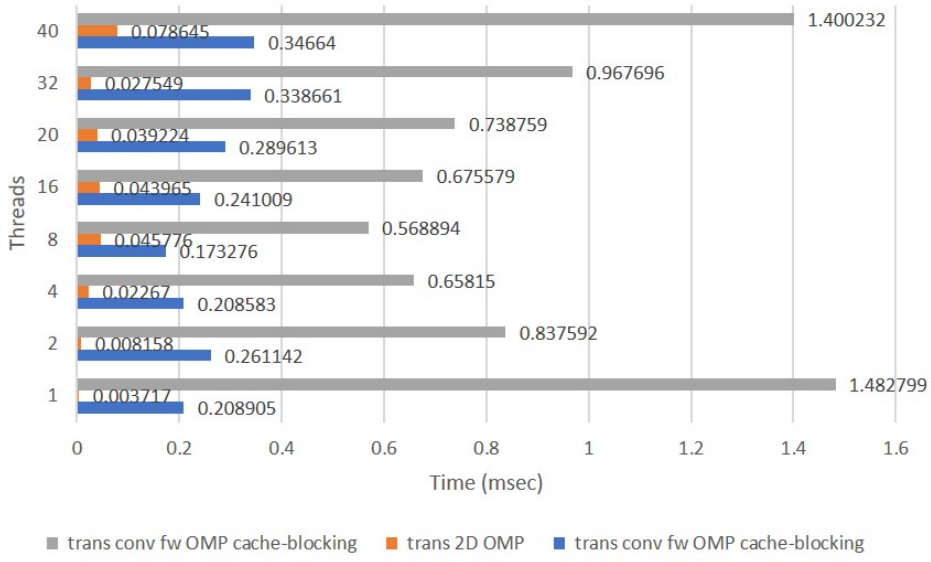


Figure 5.18: A closer look of Transpose in 4D and 2D for multiple number of threads - Kronos CPU (96 mini-batch size)

The following graph shows all the cases that transpose of 2D and 4D arrays support, using 20 threads and 96 mini-batch size images. We can assume that in this case, naive transpose is the best choice for large/medium 2D arrays, while simple OpenMP implementation can handle better 4D arrays.

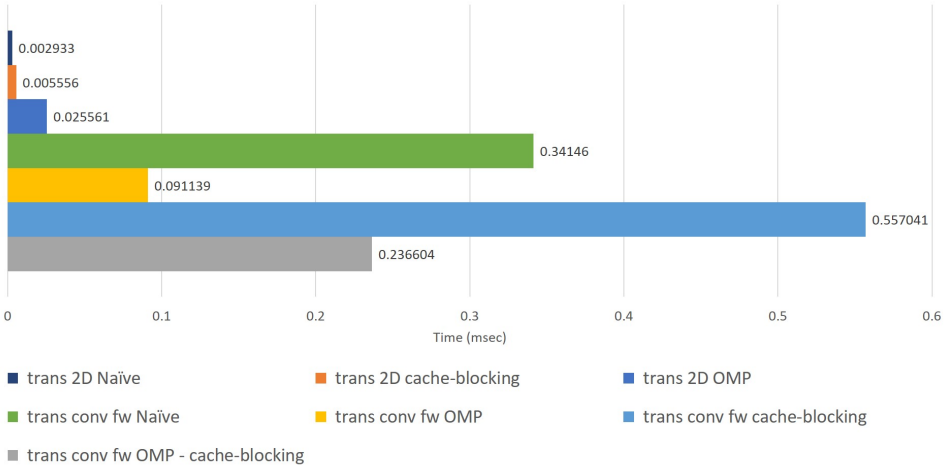


Figure 5.19: Transpose 4D and 2D: Naive and Cache Blocking (1 thread), OpenMP and OpenMP with Cache Blocking/Hybrid (20 threads) for large arrays - Kronos CPU (96 mini-batch size)

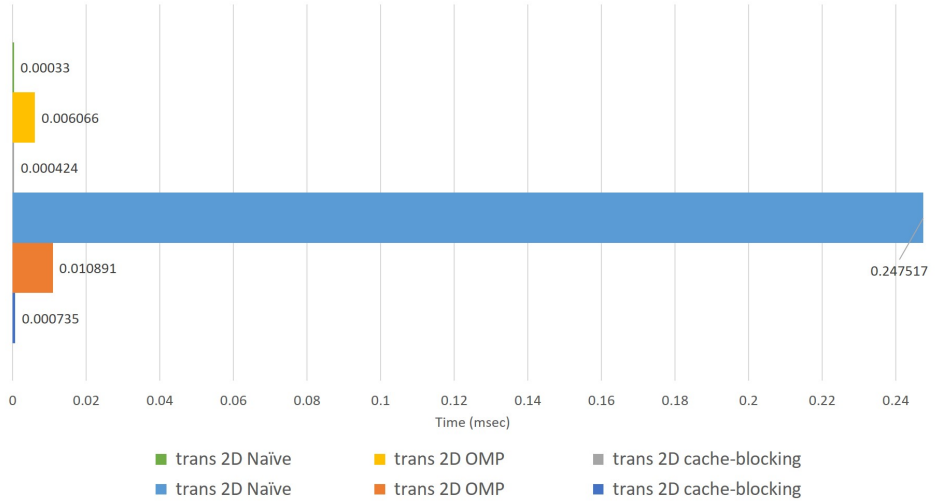


Figure 5.20: A closer look to Transpose 2D: Naive and Cache Blocking (1 thread), OpenMP(20 threads) - Kronos CPU (96 mini-batch size)

We can claim that from the graph above we can observe that as small as the 2D array is the Naive transpose is the best solution while the size of 2D arrays "grows" a bit the cache-blocking seems to be the best solution.

We can also notice here that the transpose of an array should be treated considering and applying the best method for its size to get the best time performance.

5.3.3 Copy

Time Performance of Copy - CPU

We tested some 4D and 2D versions of the Copy functions as we show below:

Copy 4D	without OpenMP	OpenMP Algorithm	Cache-Blocking Algorithm
mini-batch size	Time (ms)	Time (ms)	Time (ms)
96 images	0.043593	0.027361	0.066874

Table 5.16: Copy 4D execution time: Naive and Cache Blocking (1 thread), OpenMP and OpenMP with Cache Blocking/ Hybrid (20 threads) implementations - Kronos CPU

Copy 2D	without OpenMP	OpenMP Algorithm	Cache-Blocking Algorithm
mini-batch size	Time (ms)	Time (ms)	Time (ms)
96 images FC1	0.043409	0.015513	0.231779
96 images FC3	0.003769	0.006633	0.005159

Table 5.17: Copy 2D execution time: Naive and Cache Blocking (1 thread), OpenMP and OpenMP with Cache Blocking/ Hybrid (20 threads) implementations - Kronos CPU

Copying a 4D array is 2.4x faster using OpenMP due to size of the data, while an FC1 2D array is almost 15x faster than cache blocking. Therefore, copying an FC3 2D array that is way too small than an FC1 array is preferable to use cache blocking which is 1.2x faster than OpenMP. Even better, without OpenMP in FC3, the time performance is 1.3x better than the OpenMP and 1.75x than cache blocking.

We illustrate in the following graphs the aforementioned results:

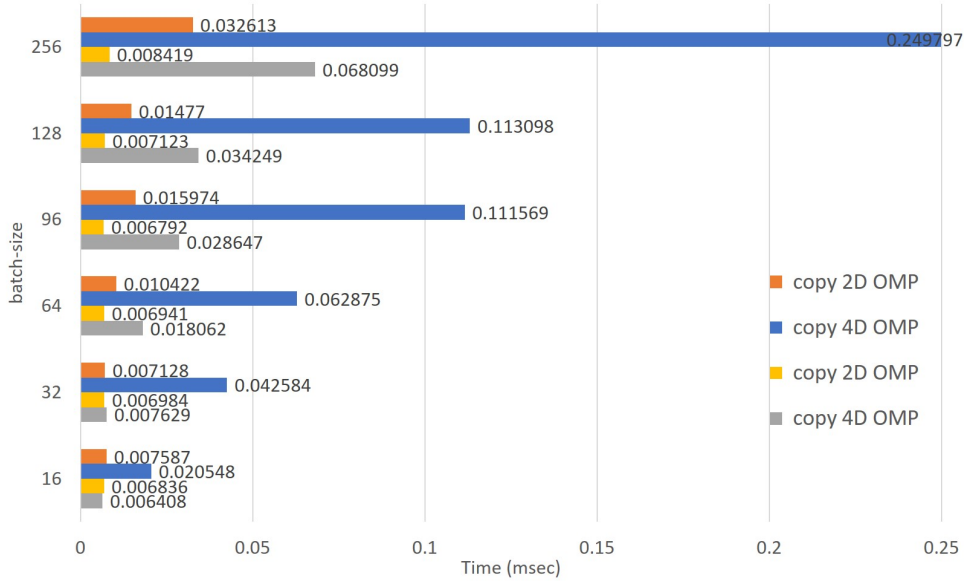


Figure 5.21: Copy 4D arrays for multiple batch-sizes - Kronos CPU (20 threads)

Looking at the graph below we can claim that the time performance and the optimal choice for the number of threads that we can use are depending on the size of the array that we want to copy.

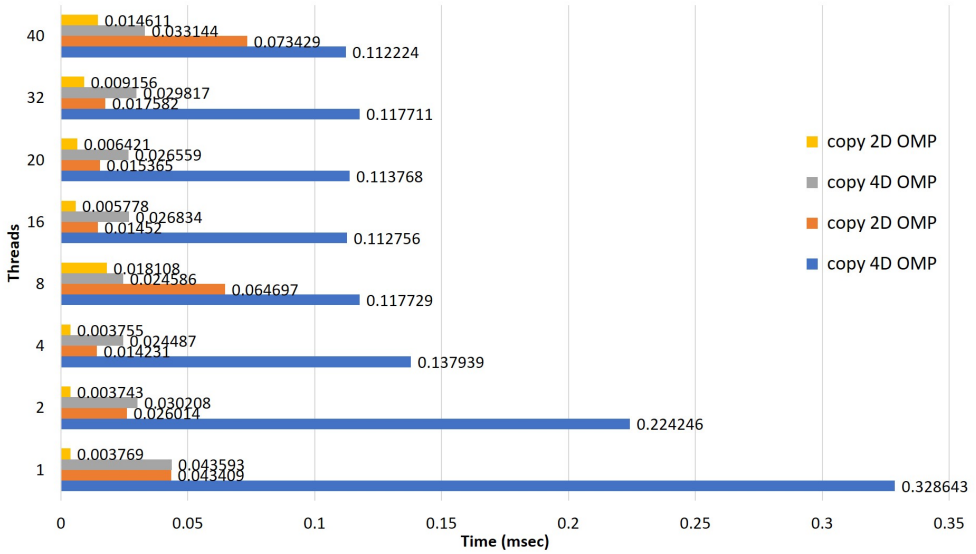


Figure 5.22: Copy 4D and 2D arrays execution time multiple number of threads - Kronos CPU (96 mini-batch size)

We can observe the behavior of 2D and 4D arrays at the following illustration (96 mini-batch size, 16 Threads). It is obvious that copying an array with OpenMP has the best time performance in both cases (2D and 4D).

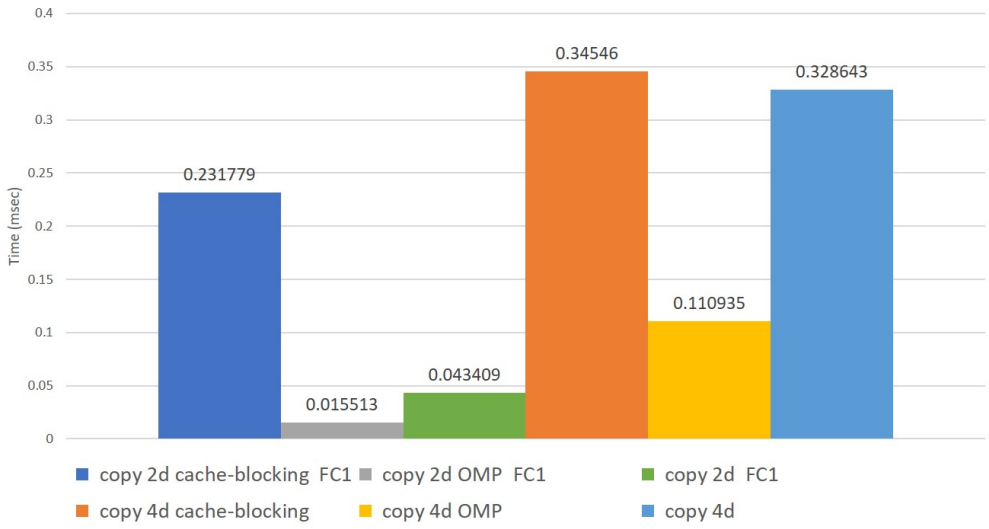


Figure 5.23: Copy 4D and 2D array's execution time: Naive and Cache Blocking (1 thread), OpenMP(20 threads) implementations for large arrays - Kronos CPU

Smaller 2D arrays seem to be handled better from naive algorithm while 4D arrays continue to have better performance using the OpenMP algorithm (96 mini-batch size, 16 Threads).

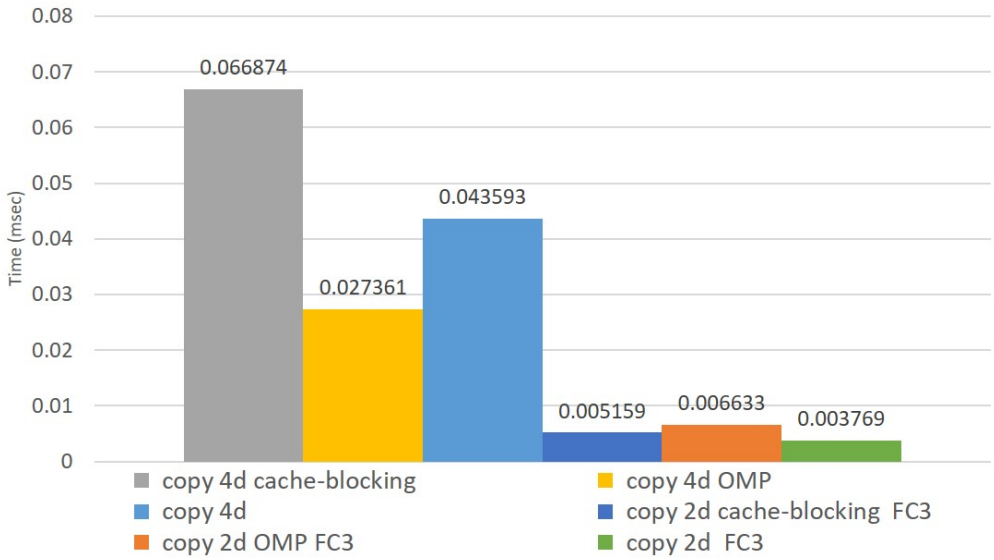


Figure 5.24: A closer look to Copy 2D and 4D array's execution time: Naive and Cache Blocking (1 thread), OpenMP(20 threads) implementations for small arrays - Kronos CPU

To sum up, it is important to determine the size of an array and choose the solution with the best performance. Multi-threading in small arrays should not be used, because we do not get the maximum performance of the process.

5.4 Energy Management and Efficiency

According to the theory [85] we can take some theoretical measurements of the Theoretical Peak Floating-Point of the Server's CPU. We will use:

$$Clock\ Frequency\ [\,TDP|AVX\,] * Cores * 16 DP\ FLOPs\ per\ cycle \tag{5.1}$$

to find the Thermal Design Power (TDP) frequency of the CPU:

$$2.2(GHz) * 10(Cores) * 16(FLOPs)\ =\ \textbf{352 GFLOPs}\ per\ socket$$

$$and\ \textbf{704 GFLOPs}\ per\ dual\ socket\ system$$

AVX instructions can be executed using more power, but the processor will reduce its frequency about 300-500MHz, in our server we observe a 400MHz (2.2 GHz to 1.8 GHz) decrease from the base frequency.

$$1.8(GHz) * 10(Cores) * 16(FLOPs) \; = \; \textbf{288 GFLOPs} \textit{ per socket}$$

and **576 GFLOPs** *per dual socket system*

CPU (Server)			
Avg. Time	Time (ms)	insn per cycle	GHz
Naive	1184224.697874	2.48	2.998
1 Thread	1245757.804671	2.38	3.014
GCC options	256071.725750	0.69	2.970
GCC options, 1 Thread	262707.041271	0.75	2.990
GCC options, 20 Threads	37624.735573	0.48	2.381

Table 5.18: Framework's time in ms, CPU's frequency and instructions per cycle - kronos CPU

Platform	CPU (Server)
Implementations	Speedup
Naive	31.5
1 Thread	33.1
GCC options	6.8
GCC options, 1 Thread	6.98

Table 5.19: Whole Kronos CPU framework's speedup comparing with the best time performance of 20 threads with GCC options

Platform	CPU (Server)	ARM-FPGA (Trenz)
Avg. Epoch Time	ms	ms
GCC options	256071.725750	4595801.757812
Speedup	17.95	1
Avg. Frequency	GHz	GHz
GCC options	2.97	1.2 (ARM) 0.3 (FPGA)
Avg. FLOPS	GFLOPS	GFLOPS
GCC options	(TDP) 704 (AVX) 576	(P.S.) 4.8 (P.L.) 250-1k
Energy	W	W
Average	145	20.35(19.5-21.1)
Energy	J	J
Average	0.037	0.094
Efficiency	2.55	1
Avg. FLOPS/W	GFLOPS/W	GFLOPS/W
GCC options	4.1	12.5–50

Table 5.20: Time, GFLOPs, frequency and speedup for 1 epoch in Server and Trenz [85, 86, 87, 88]

The Table 5.13 have the results using the Amdahl's law to calculate the Speedup[89].

$$\textbf{Speedup} = \frac{time_{\textbf{without Enhancement}}}{time_{\textbf{with Enhancement}}}$$

The Speedup equation[90] will help us to determine if using an FPGA will give us fast/faster results consuming less energy. Multiplying the Energy of each Platform and Algorithm with Matrix Multiplication with Addition time (W*ms), we get the following results. It is a confirmation that an FPGA is the most efficient choice for equal or less than 96 mini-batch of images:

Matrix Mult. with addition	Naive	Cache-Blocking Algorithm	OpenMP Algorithm	Hybrid Algorithm	FPGA
1st FC layer mini-batch size	J	J	J	J	J
128 images	3845.4	2328.7	582.9	326.25	531.5
96 images	2933.4	1748.7	407.45	284.2	164.2
64 images	1957.5	1170.2	304.5	237.8	118.2
32 images	1006.3	594.5	146.5	184.2	59.2
16 images	443.7	294.4	69.6	153.7	30.25

Table 5.21: Energy Consumption comparison between bare-metal ARM-FPGA and CPU - LeNet-5

We can notice that the FPGA implementation has the best power performance for batch sizes less than 96. For larger batch sizes, it is preferred to use the CPU hybrid algorithm as our resources in the current board and design are limited. Therefore all other implementations remain higher than the FPGA one. Furthermore, we can observe that for more than 32 images in a batch size Hybrid algorithm consumes less power than the others in the CPU, but for less than 32 we should prefer the simple OpenMP one than the other CPU implementations for faster and more energy-efficient results.

Assuming that S stands for the ratio of the time in one processor T_1 divided by the time in N processors T_N and f is the fraction of serial code in our system [91]:

$$S = \frac{T_1}{T_N} < \frac{T_1}{(f + \frac{1-f}{N})T_1} < \frac{1}{f}$$

The last part of the equation uses unlimited resources-threads- that is an ideal, often unrealistic case. Using it in our system where N: 20 threads, T_1, T_{20} we find the f and afterwards the theoretical maximum speedup:

$$S = \frac{T_1}{T_{20}} = \frac{262707.041271}{37624.735573} = 6.982$$

The serial code in our system is equal to:

$$\frac{1}{(f + \frac{1-f}{20})} = 6.982$$

$$f = 0.0932$$

Hence, the best theoretical speedup, if we claim that we have infinitive threads and the fraction of the non-parellelized code is 9.32% of total computation time is:

$$S = 6.982 < \frac{1}{0.0981} = 10.19$$

The parallel efficiency is for the best theoretical speedup

$$E = \frac{S}{N} = \frac{6.982}{20} = 0.35$$

which means that each of the cores is idle about 65% of the time and we can further improve our code in the future.

In case of fully parallelized code or nearly 100%, the program could have parallel efficiency 1(max).

Amdahl's Law is a simple method to find the theoretical upper bound that is not ordinary to surpassed or achieved in an application with serial code/tasks. However, it is criticized because it ignores a lot of real-world overheads. To be precise, while we were testing our system, we faced QPI bus overhead issues, using more than 20 cores, that Amdahl's Law fails to detect as it does to many other dependencies. The minimum time performance achieved at 20 threads instead of 40 that our system can theoretically use. Therefore, the percentage of overall serial execution time will remain the same which help us to make an estimated approach of the upper bound[92].

Chapter 6

Conclusion and Future Work

Conclusion

In this thesis, I have implemented a CNN framework almost from scratch in C++ (v. C++17) that supports grey-scale images from MNIST dataset and also I adjusted an IP core that performs fast matrix multiplication and addition (of the bias array), using AXI streams. The project is implemented in a Red Hat server and also in the Vivado SDK Suite with the appropriate changes and needs of the FPGA platform. The CNN framework performs fast and high-accurate Stochastic Gradient Descent to train and test inference, fully supported by CPUs. The CNN framework can train a whole MNIST dataset with the LeNet-5 architecture in 28 seconds (20-core CPU) and separately accelerate the matrix multiplication and addition process on an FPGA ~4.3x-5.1x times faster than a CPU implementation using cache blocking and an implementation without OpenMP ~7.5x-8.5x, using one thread, at this moment, while the energy (Watt*hour) for 1 thread is 6.9x less than a server with CPU. Nevertheless, an epoch last ~256 seconds on a Red Hat server and ~4596 seconds on a Trenz Platform without operating system, while we just call it only twice per mini-batch (Fully-Connected Layer 1 and 2) in forwarding function of the FPGA and it can be even better if we apply it to the third one (incompatible output sizes that should be filled at input and cutted at the output, different dimension sizes of arrays that should be reshaped) accelerate all the computation slow and complex functions using FPGA(s). As well as, Fully Connected layer contains 3 matrix multiplications, however, we currently apply the acceleration to one of them.

Additionally, we should import and take measurements on Trenz using OpenMP and its environment variables to compare them, which is in our future plans. Furthermore, we explore how different types of algorithms/implementations in matrix multiplication, transpose and copy are depended on the cache size, the thread number using OpenMP and the mini-batch size and that we could take an even better performance in a future version if we consider and control them. CNN can support many different architectures as we have implemented many types of layers. Moreover, it is as a user-friendly framework that is similar to the Python code of CS231

Stanford Course [2] and Deepnet [58] that we have been guided that are also easily understood.

The current platform that we tested our system was a Trenz TE0808-04 platform with 2GB DDR. Therefore, neural networks require a lot of memory, hence it would be recommended to use a corresponded platform for large architectures with multiple layers and large datasets that have enough resources to handle them.

Future Work

Many areas need improvement in our implementation. We are planning to add features and functionalities. The areas that we will focus on are the system as Neural Network, the improvement of memory usage and the implementation for specific platforms.

The Neural Network system can change in many areas. More specifically, we are planning to implement more layers and support more architectures and sequences of layers as average pooling, binary input layer, etc. As well as, the system should support more choices of loss functions and weight updates. As a further matter, we plan to support many more datasets, especially RGB ones like CIFAR-10, CIFAR-100, that are often used to train networks. The Adam weight update is under construction and there are many more that are planned to implement. One of our plans is to support also, RNN architectures on the same generic Neural Network system.

We are aware that our system should be improved in many areas. First of all, we are dealing with false sharing while using more cores do not improve our results and while we do not share data of different threads, they are sharing a cache line, which is faulty. Hence, we can remove false sharing by using padding and spacing techniques. There are hotspots and parts of code that should be improved and we can implement code that takes into consideration the size of the arrays to perform the fastest matrix multiplication, transpose or copy technique, number of threads, etc. to get faster results using only the needed resources for the task. Also, we can not get faster/better results, increasing the number of available cores, because of the latency of the QPI bus, therefore we can investigate it further in the future. Moreover, we should improve the OpenMP code and handle special cases, while performing cache blocking and scheduling. Another change we will make is the dimensions of current arrays that are multidimensional, hence we do not take advantage of the locality as we should. Therefore, there are parts of code that can be changed or improved as they are not implemented to run in parallel, due to false results that we got. Apart from the inner changes, we should explore more the GCC optimization flags and

OpenMP environment options that could help us to get faster and yet correct results. Even though we implemented a bare-metal CNN application, we could exploit the multi-core ARM with OpenMP features, etc. in a Petalinux environment with the necessary libraries that need our framework or to make a bare-metal application that can handle the memory, etc. to control/manage the framework's resources and operations.

Although software development is a significant part of our thesis, there is also the hardware code evaluation. There is an implementation in HLS of Convolutional Layer that should be further developed in Vivado Design and SDK tools and then attached to our system. Furthermore, the weight update of the Fully-Connected layer should be also implemented apart from the constructed matrix multiplication accelerated IP. Besides that, it would be interesting to support GPUs to compare our results of the same system.

Appendix A

More Results

A.1 CPU time for each layer

We get the following results using OpenMP and Cache Blocking (Hybrid version). We tested our code in multiple threads for a LeNet-5 with Tanh activation function for multiple threads and we got the following results:

Threads Layers	1T (ms)	2T (ms)	4T (ms)	8T (ms)
Forward				
Convolutional	17171.771	9214.292	49.84946	34.64402
Tanh	8719.567	4243.058	24.83642	13.3892
Maxpool	499.419	272.751	1.533	1.05299
Tanh	2110.084	1052.818	6.11808	3.56738
Convolutional	39694.767	21274.692	122.14591	73.59321
Tanh	6716.072	3455.175	20.10052	10.42761
Maxpool	859.805	460.093	2.94947	1.74959
Tanh	1557.677	781.467	4.8124	2.89336
Flatten	42.898	25.429	0.14971	0.13999
Fully Connected	20857.269	10649.903	60.90863	35.32145
Tanh_2d	293.061	157.376	1.10376	0.58704
Fully Connected	517.68	396.373	3.30439	3.68241
Tanh_2d	150.862	79.784	0.47319	0.36399
Fully Connected	33.676	44.245	0.46707	0.59503
Backward				
Fully Connected	97.938	542.01	6.31538	9.62487
Tanh_2d	4.139	6.047	0.05359	0.0642
Fully Connected	1012.338	1082.829	10.99837	17.68701
Tanh_2d	8.218	8.743	0.07223	0.09045
Fully Connected	7722.833	4139.153	27.64975	21.06681
Flatten	43.326	32.276	0.25896	0.27962
Tanh	78.074	49.004	0.27337	0.24129
Maxpool	258.862	161.709	1.19119	0.89504
Tanh	276.802	161.353	0.96852	0.98859
Convolutional	148381.883	81669.994	441.80117	253.15703
Tanh	118.933	85.799	0.61681	0.62694
Maxpool	431.029	278.007	1.96051	1.90771
Tanh	488.244	305.03	2.48787	1.72461
Convolutional	52262.165	38250.871	231.65433	148.87413

Table A.1: Average time for every layer of LeNet-5 (Threads : 1, 2, 4, 8)

Threads Layers	16T (ms)	20T (ms)	32T (ms)	40T (ms)
Forward				
Convolutional	34.63984	30.96458	32.068	37.00192
Tanh	12.32866	10.63478	9.59495	5.41782
Maxpool	1.28794	0.9517	0.99803	1.47519
Tanh	3.17273	2.88898	2.65597	1.42328
Convolutional	75.68199	63.29355	65.09242	70.28411
Tanh	10.88638	8.34537	7.29103	4.31637
Maxpool	1.91749	1.40178	1.0944	1.12461
Tanh	2.59534	1.99392	1.71611	1.32964
Flatten	0.14797	0.13225	0.16168	0.5035
Fully Connected	36.14988	28.5855	22.25263	23.02115
Tanh_2d	0.61747	0.50133	0.37796	0.5314
Fully Connected	3.45016	3.55332	5.06118	8.86438
Tanh_2d	0.37442	0.30313	0.2292	0.43619
Fully Connected	0.56321	0.67584	1.04107	1.84588
Backward				
Fully Connected	10.07898	9.71828	16.15353	26.20209
Tanh_2d	0.06429	0.06567	0.10866	0.49771
Fully Connected	15.01188	18.37143	20.34919	31.83597
Tanh_2d	0.17913	0.08321	0.13159	0.49868
Fully Connected	21.44933	19.77662	24.6245	33.26432
Flatten	0.26363	0.27801	0.30391	0.52262
Tanh	0.26383	0.2039	0.2168	0.3416
Maxpool	0.83909	0.94946	1.04602	0.90236
Tanh	0.86272	1.06501	1.21812	0.84138
Convolutional	256.74239	209.57183	205.43463	185.71362
Tanh	0.73282	0.61915	0.61738	1.52037
Maxpool	1.97092	1.89875	1.73215	1.9196
Tanh	1.74692	2.09267	1.63038	2.41526
Convolutional	166.50681	134.9584	116.82502	108.6352

Table A.2: Average time for every layer of LeNet-5 (Threads : 16, 20, 32, 40)

A.2 CPU time using 20 threads - Relu and Tanh activation functions

The following results used OpenMP and Cache Blocking (Hybrid version) in 20 threads. We can observe the differences in time for each layer and how the activation layers affect the next layers in time (and their calculations):

Layers	20 Threads (ms)		20 Threads (ms)
Forward			
Convolutional	2.84797	Convolutional	2.731528
Relu	0.148618	Tanh	0.506758
Maxpool	0.05625	Maxpool	0.092748
Relu	0.040487	Tanh	0.184584
Convolutional	6.294795	Convolutional	6.156406
Relu	0.098858	Tanh	0.444596
Maxpool	0.093008	Maxpool	0.092877
Relu	0.032665	Tanh	0.152522
Flatten	0.014641	Flatten	0.013395
Fullyconnected	2.11734	Fullyconnected	2.072485
Relu_2d	0.016058	Tanh_2d	0.033048
Fullyconnected	0.504706	Fullyconnected	0.455869
Relu_2d	0.013699	Tanh_2d	0.019871
Fullyconnected	0.09727	Fullyconnected	0.090727
Backward			
Fullyconnected	1.357518	Fullyconnected	1.405481
Relu_2d	0.007714	Tanh_2d	0.007774
Fullyconnected	1.513596	Fullyconnected	1.479611
Relu_2d	0.009017	Tanh_2d	0.008989
Fullyconnected	1.933841	Fullyconnected	1.917679
Flatten	0.019909	Flatten	0.036375
Relu	0.023098	Tanh	0.016279
Maxpool	0.114904	Maxpool	0.104509
Relu	0.064337	Tanh	0.056975
Convolutional	15.251531	Convolutional	15.302945
Relu	0.067735	Tanh	0.027215
Maxpool	0.158054	Maxpool	0.158729
Relu	0.180032	Tanh	0.268937
Convolutional	10.903264	Convolutional	9.692376

Table A.3: Average time for every layer of LeNet-5 with ReLU and Tanh activation functions

A.3 Memory results from Perf Tool

We could extract some cache results that will be used to further improvements in our future work.

Threads		1T		2T	
task-clock (msec)	CPUs utilized	261370.5899	0.998	294294.0939	1.996
context-switches	K/sec	53,585	0.205	54,333	0.185
cpu-migrations	K/sec	10	0	21	0
page-faults	K/sec	19,394	0.074	18,716	0.064
cycles	GHz	779,972,883,994	2.984	879,324,468,228	2.988
instructions	insn per cycle	722,398,991,807	0.93	735,888,812,969	0.84
branches	M/sec	70,487,553,149	269.684	75,197,528,041	255.518
branch-misses	of all branches	1,072,696,241	1.52%	1,123,325,394	1.49%
L1-dcache-loads	M/sec	289,828,753,024	108.881	289,474,958,306	983.625
L1-dcache-load-misses	of all L1-dcache hits	75,927,384,684	26.20%	75,671,560,772	26.14%
LLC-loads	M/sec	55,157,994,205	211.034	54,277,091,399	184.431
LLC-load-misses	of all LL-cache hits	231,442,636	0.84%	487,959,644	1.80%
L1-icache-load-misses		76,481,029		73,895,089	
dTLB-loads	M/sec	286,339,397,481	1095.53	291,614,746,482	990.896
dTLB-load-misses	of all dTLB cache hits	672,097	0.00%	901,913	0.00%
iTLB-loads	M/sec	2,093,659	0.008	7,820,664	0.027
iTLB-load-misses	of all iTLB cache hits	1,465,785	70.01%	2,886,647	36.91%

Table A.4: LeNet-5 Architecture Summary Table from Perf tool (1T, 2T)

Threads		4T		8T	
task-clock (msec)	CPUs utilized	331530.8038	3.992	421872.6854	7.984
context-switches	K/sec	56,960	0.172	61,729	0.146
cpu-migrations	K/sec	25	0	44	0
page-faults	K/sec	11,484	0.035	7,931	0.019
cycles	GHz	965,693,227,275	2.913	1,118,564,101,390	2.651
instructions	insn per cycle	761,885,709,091	0.79	800,405,627,118	0.72
branches	M/sec	80,847,999,463	243.863	93,356,138,528	221.29
branch-misses	of all branches	1,012,090,300	1.25%	1,122,147,082	1.20%
L1-dcache-loads	M/sec	293,742,895,365	886.02	301,143,470,314	713.825
L1-dcache-load-misses	of all L1-dcache hits	75,879,919,398	25.83%	75,947,068,614	25.22%
LLC-loads	M/sec	53,972,768,746	162.799	53,370,247,036	126.508
LLC-load-misses	of all LL-cache hits	496,463,419	1.84%	508,204,767	1.90%
L1-icache-load-misses		91,501,958		103,295,020	
dTLB-loads	M/sec	293,698,120,001	885.885	300,582,297,213	712.495
dTLB-load-misses	of all dTLB cache hits	938,981	0.00%	995,863	0.00%
iTLB-loads	M/sec	6,947,945	0.021	8,448,452	0.02
iTLB-load-misses	of all iTLB cache hits	1,337,424	19.25%	1,507,614	17.84%

Table A.5: LeNet-5 Architecture Summary Table from Perf tool (4T, 8T)

Threads		16T		20T	
task-clock (msec)	CPUs utilized	639032.8252	15.977	765152.7211	19.971
context-switches	K/sec	65,686	0.103	65,058	0.085
cpu-migrations	K/sec	57	0	49	0
page-faults	K/sec	7,456	0.012	8,724	0.011
cycles	GHz	1,521,813,247,524	2.381	1,821,955,797,444	2.381
instructions	insn per cycle	915,177,025,022	0.6	974,813,552,887	0.54
branches	M/sec	123,941,367,396	193.951	142,015,100,922	185.604
branch-misses	of all branches	1,112,521,138	0.90%	1,238,352,413	0.87%
L1-dcache-loads	M/sec	317,394,760,727	496.68	327,046,660,078	427.427
L1-dcache-load-misses	of all L1-dcache hits	75,452,799,953	23.77%	75,629,372,142	23.12%
LLC-loads	M/sec	53,849,702,303	84.268	53,811,038,157	70.327
LLC-load-misses	of all LL-cache hits	523,868,627	1.95%	542,777,368	2.02%
L1-icache-load-misses		154,487,328		185,555,541	
dTLB-loads	M/sec	316,849,450,495	495.827	327,403,073,475	427.892
dTLB-load-misses	of all dTLB cache hits	1,213,087	0.00%	1,612,050	0.00%
iTLB-loads	M/sec	2,757,485	0.004	3,235,026	0.004
iTLB-load-misses	of all iTLB cache hits	23,701,402	859.53%	3,399,211	105.08%

Table A.6: LeNet-5 Architecture Summary Table from Perf tool (16T, 20T)

Threads		32T		40T	
task-clock (msec)	CPUs utilized	1421466.091	31.949	1894648.708	39.895
context-switches	K/sec	82,988	0.058	112,052	0.059
cpu-migrations	K/sec	1,038	0.001	440	0
page-faults	K/sec	10,189	0.007	11,086	0.006
cycles	GHz	3,378,137,285,778	2.377	4,503,765,226,912	2.377
instructions	insn per cycle	1,301,284,247,229	0.39	1,551,440,241,224	0.34
branches	M/sec	233,572,238,177	164.318	303,328,909,101	160.098
branch-misses	of all branches	1,313,783,749	0.56%	1,330,390,385	0.44%
L1-dcache-loads	M/sec	376,927,297,447	265.168	415,918,766,238	219.523
L1-dcache-load-misses	of all L1-dcache hits	68,067,871,885	18.06%	65,905,756,224	15.85%
LLC-loads	M/sec	46,237,106,374	32.528	43,170,305,685	22.785
LLC-load-misses	of all LL-cache hits	705,001,052	3.05%	735,589,784	3.41%
L1-icache-load-misses		298,382,900		374,326,771	
dTLB-loads	M/sec	376,734,249,727	265.032	416,272,947,627	219.71
dTLB-load-misses	of all dTLB cache hits	4,593,658	0.00%	4,483,663	0.00%
iTLB-loads	M/sec	5,031,204	0.004	2,170,287	0.001
iTLB-load-misses	of all iTLB cache hits	5,559,716	110.50%	6,219,830	286.59%

Table A.7: LeNet-5 Architecture Summary Table from Perf tool (32T, 40T)

Bibliography

[1] Mathworks, "Convolutional Neural Network." https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html.

[2] CS231n, "Convolutional Neural Networks for Visual Recognition." http://cs231n.github.io.

[3] Frederik Kratzert, "Understanding the backward pass through Batch Normalization Layer." http://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html.

[4] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Shape, Contour and Grouping in Computer Vision*, (London, UK, UK), pp. 319–, Springer-Verlag, 1999. http://dl.acm.org/citation.cfm?id=646469.691875.

[5] Namhyuk Ahn, "Collection of PGFTikZ figures.." http://nmhkahn.github.io/assets/CNN-Practice/im2col.png.

[6] Trenz Electronic, "TE0808-04-09-2IE-S Starter Kit." https://shop.trenz-electronic.de/en/TE0808-04-09-2IE-S-TE0808-04-09-2IE-S-Starter-Kit.

[7] GeeksforGeeks, "Shuffle a given array using FisherYates shuffle Algorithm." https://www.geeksforgeeks.org/shuffle-a-given-array-using-fisher-yates-shuffle-algorithm/.

[8] Paras Dahal, "Implementing Convolutional Neural Networks ." https://deepnotes.io.

[9] J. Redmon, "Darknet: Open source neural networks in c." http://pjreddie.com/darknet/, 2013–2016.

[10] Google, "TensorFlow: An end-to-end open source machine learning platform." https://www.tensorflow.org/.

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner,

I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from `https://www.tensorflow.org/`.

[12] MarutiTechLabs, "Top 8 Deep Learning Frameworks." `https://www.marutitech.com/top-8-deep-learning-frameworks/`.

[13] Anton Shaleynikov, "10 Best Frameworks and Libraries for AI." `https://dzone.com/articles/progressive-tools10-best-frameworks-and-libraries/`.

[14] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 14–26, March 2016.

[15] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-cnn: An fpga-based framework for training convolutional neural networks," pp. 107–114, 07 2016.

[16] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," *CoRR*, vol. abs/1612.07119, 2016.

[17] T. Geng, T. Wang, A. Li, X. Jin, and M. C. Herbordt, "A scalable framework for acceleration of CNN training on deeply-pipelined FPGA clusters with weight and workload balancing," *CoRR*, vol. abs/1901.01007, 2019.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[19] P. Murugan, "Feed forward and backward run in deep convolution neural network," *CoRR*, vol. abs/1711.03278, 2017. `http://arxiv.org/abs/1711.03278`.

[20] Andrej Karpathy, "Connecting images and natural language." `https://cs.stanford.edu/people/karpathy/main.pdf`.

[21] Mamy Ratsimbazafy, "Convolution optimisation resources." `https://github.com/numforge/laser/wiki/Convolution-optimisation-resources`.

[22] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *ArXiv e-prints*, 11 2015.

[23] Leonardo Araujo dos Santos, "Artificial Inteligence: Maxpool Layer ." https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/pooling_layer.html.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015. http://arxiv.org/abs/1502.01852.

[25] SuperDataScience Team, "Convolutional Neural Networks (CNN): Step 3 - Flattening ." https://www.superdatascience.com/convolutional-neural-networks-cnn-step-3-flattening/.

[26] S. Moncada , "Step 3: Flattening." https://www.superdatascience.com/convolutional-neural-networks-cnn-step-3-flattening/.

[27] Sergey Ioffe and Christian Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. http://arxiv.org/abs/1502.03167.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[29] Petar Velikovi, "Collection of PGF/TikZ figures.." https://github.com/PetarV-/TikZ.

[30] Leon René Sütfeld and Flemming Brieger and Holger Finger and Sonja Füllhase and Gordon Pipa, "Adaptive blending units: Trainable activation functions for deep neural networks," *CoRR*, vol. abs/1806.10064, 2018. http://arxiv.org/abs/1806.10064.

[31] S. Sharma, "Activation Functions: Neural Networks." https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[32] R.H.R. Hahnloser, R. Sarpeshkar, M.A. Mahowald, R. J. Douglas, H.S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit." https://www.nature.com/articles/35016072.pdf.

[33] Wikipedia, "Rectifier (neural networks)." https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[34] J. Seo, J. Lee, and K. Kim, "Activation functions of deep neural networks for polar decoding applications," in *28th IEEE Annual International Symposium on Personal, Indoor, and Mobile Radio Communications, PIMRC 2017, Montreal,*

QC, Canada, October 8-13, 2017 [34], pp. 1–5. https://dblp.org/rec/bib/conf/pimrc/SeoLK17.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017. http://doi.acm.org/10.1145/3065386.

[36] Developer Guide for Intelő Data Analytics Acceleration Library 2019 Update 1, "ReLU Backward Layer." https://software.intel.com/en-us/daal-programming-guide-relu-backward-layer.

[37] Olga Russakovsky and Jia Deng and Hao Su and Jonathan Krause and Sanjeev Satheesh and Sean Ma and Zhiheng Huang and Andrej Karpathy and Aditya Khosla and Michael Bernstein and Alexander C. Berg and Li Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. https://arxiv.org/pdf/1409.0575.pdf.

[38] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Intelligent Signal Processing*, pp. 306–351, IEEE Press, 2001.

[39] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[40] "Adaptive Convolutional ELM For Concept Drift Handling in Online Stream Data." https://www.researchgate.net/figure/CNN-Architecture-from-AlexNet-43_fig1_308964842.

[41] Medium, "CNN ArchitecturesLeNet, AlexNet, VGG, GoogLeNet and ResNet." https://medium.com/@RaghavPrabhu/cnn-architectures-lenet-alexnet-vgg-googlenet-and-resnet-7c81c017b848.

[42] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[43] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. https://arxiv.org/pdf/1512.03385.pdf.

[45] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, pp. 473–491, April 2011. http://doi.acm.org/10.1109/TCAD.2011.2110592.

[46] David Gschwend, "ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network." https://github.com/dgschwend/zynqnet/zynqnet_report.pdf.

[47] C. Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Newton, MA, USA: Newnes, 1st ed., 2004.

[48] David Kelf, "Linking high-level synthesis with formal verification." https://www.techdesignforums.com/practice/technique/onespin-systemc-hls-formal-verification.

[49] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. UK: Strathclyde Academic Media, 2014.

[50] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 1591–1604, Oct 2016.

[51] Xilinx Inc., "Introduction to fpga design with vivado high-level synthesis." https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf.

[52] Y. Bai, M. Alawad, R. F. DeMara, and M. Lin, "Optimally fortifying logic reliability through criticality ranking," *Electronics*, vol. 4, no. 1, pp. 150–172, 2015. http://www.mdpi.com/2079-9292/4/1/150.

[53] D. B. Thomas, L. Howes, and W. Luk, "A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, (New York, NY, USA), pp. 63–72, ACM, 2009. http://doi.acm.org/10.1145/1508128.1508139.

[54] E. Fykse, "Performance comparison of gpu, dsp and fpga implementations of image processing and computer vision algorithms in embedded systems," 2013.

[55] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2010. https://www.bibsonomy.org/bibtex/2e90a6474d85eac083c921cf5be29f6ef/toevanen.

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

[57] Ali Naseri, John Hartfiel, Trenz Electronic GmbH, "TE0808 TRM." https://wiki.trenz-electronic.de/display/PD/TE0808+TRM#TE0808TRM-KeyFeatures.

[58] Paras Dahal, "Deepnet: Implementations of CNNs, RNNs and cool techniques in deep learning from scratch ." https://github.com/parasdahal/deepnet.

[59] Wikipedia, "MNIST database." https://en.wikipedia.org/wiki/MNIST_database.

[60] "THE MNIST DATABASE of handwritten digits." http://yann.lecun.com/exdb/mnist/.

[61] Zou, Hui and Hastie, Trevor, "Regularization and variable selection via the elastic net (vol b 67, pg 301, 2005)," *Journal of the Royal Statistical Society Series B*, vol. 67, pp. 768–768, 02 2005. https://web.stanford.edu/~hastie/Papers/elasticnet.pdf.

[62] Eli Bendersky, "The Softmax function and its derivative." https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/.

[63] Google Developers, "Multi-Class Neural Networks: Softmax." hhttps://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax.

[64] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014. http://caffe.berkeleyvision.org.

[65] B. D. Hammel, "What learning rate should I use?." http://www.bdhammel.com/learning-rates/.

[66] K. D. Yamada, "Hyperparameter-free optimizer of gradient-descent method incorporating unit correction and moment estimation," *bioRxiv*, 2018. https://www.biorxiv.org/content/early/2018/06/18/348557.full.pdf.

[67] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. https://arxiv.org/pdf/1609.04747.pdf.

[68] Google Developers, "Reducing Loss: Learning Rate." https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate.

[69] Scikit-learn developers, "1.5. Stochastic Gradient Descent." https://scikit-learn.org/stable/modules/sgd.html.

[70] T. Lin, S. U. Stich, and M. Jaggi, "Don't use large mini-batches, use local SGD," *CoRR*, vol. abs/1808.07217, 2018.

[71] loveSnowBest, "Improving Deep Neural Networks Assignment 2." https://lovesnowbest.site/2018/02/16/Improving-Deep-Neural-Networks-Assignment-2/.

[72] B. Polyak, "Some methods of speeding up the convergence of iteration methods," *Ussr Computational Mathematics and Mathematical Physics*, vol. 4, pp. 1–17, 12 1964. http://vsokolov.org/courses/750/files/polyak64.pdf.

[73] Botev, Aleksandar and Lever, Guy and Barber, David, "Nesterov's accelerated gradient and momentum as approximations to regularised update descent," 05 2017. https://arxiv.org/pdf/1607.01981.pdf.

[74] Y. F. Fisher, Ronald A., "Statistical tables for biological, agricultural and medical research (3rd ed.). london: Oliver & boyd," pp. 26–27, 1948[1938].

[75] Wikipedia, "Fisher-Yates shuffle." https://en.wikipedia.org/wiki/FisherYates_shuffle.

[76] GCC, the GNU Compiler Collection, "Options That Control Optimization." https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[77] The Valgrind Developers, "Valgrind." http://www.valgrind.org.

[78] Jim Demmel, "CS 267 Applications of Parallel Computers : Memory Hierarchies and Optimizing Matrix Multiplication ." https://people.eecs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect_02_1999b.pdf.

[79] M. Redekopp, M. Shindler, R. Govindan, "CSCI 350, Veterbi University, USC: Chapter 9, Caching and VM ." http://ee.usc.edu/~redekopp/cs350/slides/Ch9_Caching.pdf.

[80] OpenMP, "Openmp official website." https://www.openmp.org/.

[81] GNU libgomp, "Openmp environment variables." https://gcc.gnu.org/onlinedocs/libgomp.

[82] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, pp. 12:1–12:25, May 2008. http://doi.acm.org/10.1145/1356052.1356053.

[83] Xilinx, "Zynq UltraScale+ MPSoC Data Sheet: Overview ." https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[84] Xilinx Inc., "Vivado design suit user guide high-level synthesis." https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf.

[85] Advanced Clustering Technologies Inc., "Xeon E5-2600 v4 series overview details of the new Intel Xeon Broadwell systems." https://www.advancedclustering.com/wp-content/uploads/2017/02/xeon-e5v4.pdf.

[86] M. Katevenis, N. Chrysos, M. Marazakis, I. Mavroidis, F. Chaix, N. Kallimanis, J. Navaridas, J. Goodacre, P. Vicini, A. Biagioni, P. Paolucci, A. Lonardo, E. Pastorelli, F. Lo Cicero, R. Ammendola, P. Hopton, P. Coates, G. Taffoni, S. Cozzini, and G. Perna, "The exanest project: Interconnects, storage, and packaging for exascale systems," pp. 60–67, 08 2016. http://www.exanest.eu/pub/ExaNeSt_DSD2016_v1.0.pdf.

[87] Intel Corporation, "Intelő Xeonő Processor E5-2690 v4." https://ark.intel.com/content/www/us/en/ark/products/92981/intel-xeon-processor-e5-2630-v4-25m-cache-2-20-ghz.html.

[88] Andriy Berestovskyy, "Xeon what? new intel xeon processors with many integrated core (mic) architecture," 2017. https://www.slideshare.net/AndriyBerestovskyy/xeon-what-new-intel-xeon-processors-with-many-integrated-core-mic-architecture.

[89] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967. http://doi.acm.org/10.1145/1465482.1465560.

[90] MIT Open Courseware, Chris Terman, "Annotations in Computation Structures." https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/c21/c21s1/#15.

[91] Boston University Information Services & Technology, "Speedup Ratio and Parallel Efficiency." https://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/scalability/.

[92] Intel, "Intel Guide for Developing Multithreaded Applications: Predicting and Measuring Parallel Performance." https://software.intel.com/en-us/articles/predicting-and-measuring-parallel-performance.

Index