HELLENIC REPUBLIC
National and Kapodistrian
University of Athens
EST. 1837

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΤΜΗΜΑ ΦΥΣΙΚΗΣ

ΔΠΜΣ ΗΛΕΚΤΡΟΝΙΚΟΣ ΑΥΤΟΜΑΤΙΣΜΟΣ

Διπλωματική Εργασία

Hardware Accelerator for Convolutional Neural Networks

Επιταχυντής Υλικού για Συνελικτικά Νευρωνικά Δίκτυα

Άγγελος Ψημίτης-Χριστοδουλόπουλος

ΑΜ: 2019-513

Αθήνα, Νοέμβριος 2022

SUPERVISOR

Dr. Nikolaos Vlassopoulos, Research Associate

EVALUATION COMMITEE

Dionysios Reisis, Professor

Ektoras Nistazakis, Professor

Dr. Nikolaos Vlassopoulos, Research Associate

Abstract

In this thesis we attempt to discuss the concept of hardware accelerators for convolutional neural networks which can be perceived as a linkage between two different areas of computer science. The design of parallel systems and the design of machine learning algorithms. The former exploits ways of mapping high level programs into hardware structures in order to increase the speed of computations and has been growing since the rise of VLSI engineering in the last 40 years. The latter addresses the idea of creating dynamic algorithms that proceed in an iterative fashion, based on some task and some quantity of experience in order to solve problems which are extremelly difficult (or even impossible) to be solved by using hard-coded programs. In the last 10 years machine learning has opened a vast world filled with endless possibilities and countless applications, especially through Deep Learning and Deep Neural Networks (DNNs). Convolutional Neural Networks (CNNs) are a special case of DNNs which are used for image recognition in various computer vision related tasks. A hardware accelerator is a special-purpose system dedicated to increase the speed of the computationally intensive parts of the CNN algorithm for results to be given both fast and efficiently regarding energy consumption. In what follows we will approach the theoretical background behind efficient parallel computations and neural networks and we will present the FPGA design of the CNN Accelerator. All VHDL code written for this work can be found in `https://github.com/AggelosPsimitis/FPGA-hardware-accelerator-for-CNN/tree/master`.

Περίληψη

Στην παρούσα εργασία επιχειρούμε να συζητήσουμε το θέμα των επιταχυντών για νευρωνικά δίκτυα που γίνεται αντιληπτό ως συνδετικός κρίκος ανάμεσα σε δύο πεδία της επιστήμης των υπολογιστών: τη σχεδίαση παράλληλων συστημάτων και τη σχεδίαση αλγορίθμων μηχανικής μάθησης. Το πρώτο πεδίο ερευνά τους τρόπους με τους οποίους προγράμματα υψηλού επιπέδου αντιστοιχίζονται σε δομές υλικού με σκοπό την επιτάχυνση των υπολογισμών. Το δεύτερο ερευνά τη σχεδίαση δυναμικών αλγορίθμων που έχουν ως στόχο την προσέγγιση της λύσης προβλημάτων που είναι εξαιρετικά δύσκολο (ή ακόμη και αδύνατο) να επιλυθούν από προγράμματα γραμμένα εξ' ολοκλήρου από το χρήστη. Ο κλάδος της μηχανικής μάθησης έχει αναδείξει την τελευταία δεκαετία πλήθος εφαρμογών και δυνατοτήτων κυρίως μέσω της βαθιάς μάθησης και των νευρωνικών δικτύων. Τα συνελικτικά νευρωνικά δίκτυα είναι μια ειδική κατηγορία βαθιών νευρωνικών δικτύων που χρησιμοποιούνται για ανίχνευση εικόνων καθώς και σε πολλές άλλες εφαρμογές που σχετίζονται με την υπολογιστική όραση. Ένας επιταχυντής υλικού είναι ένα σύστημα ειδικού σκοπού εξειδικευμένο στην αύξηση της ταχύτητας των εντατικών υπολογισμών που αποτελούν μέρος του αλγορίθμου ενός νευρωνικού δικτύου. Σκοπός του είναι τα αποτελέσματα των υπολογισμών να παράγονται γρήγορα και αποδοτικά όσον αφορά την κατανάλωση ενέργειας. Στα επόμενα κεφάλαια θα προσεγγίσουμε το θεωρητικό υπόβαθρο των παράλληλων υπολογισμών καθώς και εκείνο των νευρωνικών δικτύων και θα παρουσιάσουμε τη σχεδίαση ενός επιταχυντή συνελικτικού νευρωνικού δικτύου σε FPGA. Η υλοποίηση πραγματοποιήθηκε σε γλώσσα VHDL και βρίσκεται στο σύνδεσμο: `https://github.com/AggelosPsimitis/FPGA-hardware-accelerator-for-CNN/tree/master`.

Contents

1 Introduction **1**

2 Parallel Computation Theoretical Background **3**

2.1 Systolic Architectures 3

2.1.1 Matrix-vector product 5

2.1.2 N-tap FIR filter 5

2.2 Measuring the performance of parallel algorithms 7

2.3 Space-time mapping 10

2.4 Unfolding 16

3 Deep Neural Networks Theoretical Background **22**

3.1 Parametric modeling 23

3.2 Neural Networks 25

3.3 Convolutional Networks 28

4 Accelerating Inference for DNNs **30**

4.1 Training vs Inference 31

4.2 Key Metrics 31

4.3 Taxonomy of accelerator architectures 36

4.4 Hardware architectures for kernel computations in DNN processing 37

5 Design of Hardware Accelerator for CNN on FPGA **40**

5.1 The CNN model 40

5.2 Architectures for efficient 2D convolution 41
5.3 Overall architecture 45
5.4 Input Layer 45
5.5 Convolution Layer 46
5.6 Pooling Layer 47
5.7 Fully Connected and Output Layers 50

6 Tests and Results 52

7 Appendix: Tables 57

1 Introduction

Most neural network software applications are executed on CPUs and result in very long training times. Recent research in robotics and autonomous driving has featured the needs for increasingly larger data sets and high precision real time results. This means that neural networks are increasing in size by introducing many hidden layers, leading to large scale architecture models consisting of millions of parameters and performing billions of operations resulting in programs of high complexity and high usage of computational resources. GoogleNet for example receives 224x224x3 (RGB) images as input and is constructed of 57 convolutional (hidden) layers with 6 million weight values (connections) and 1,43 giga Multiply-Accumulate (MAC) operations. These contemporary demands present a significant challenge for general puprose processors and research has moved towards developing techniques for designing efficient dedicated hardware for accelerating the computations of these models utilizing GPUs, ASICS and FPGAs. GPUs are very popular hardware accelerators for CNNs due to their high bandwidth in memory, high throughput and efficiency in floating point arithmetic. They consist of hundreds of small processing cores which can be used in parallel to enhance image processing. GPUs though lack in power consumption and FPGAs usually outperform them in implementation of vision kernels [20]which is the main theme of this work. FPGAs are programmable reconfigurable devices with low power consumption. They consist of Configurable Logic Blocks (CLBs), I/O cells and DSP blocks. CLBs are comprised of Look Up Tables (LUTs) and Flip FLops (FFs). FPGAs are fine-grained devices. A program can be divided in a large number of small tasks executed in parallel from different blocks of the device. Due to their low power consumption they are great accelerators for battery-driven devices or cloud services in large data servers. Another advantage of FPGAs is their capability of reconfiguring parts of the system's structure while

the rest is still used. On the other hand FPGAs have small available on-chip memory, low I/O bandwidth and limited resources.

This thesis is organized as follows: In chapter 2 we revise on the theory behind parallel computations and the basic metrics used to measure the performance of parallel algorithms. We see how the systolic architectures and the linear array idea can be used to improve the performance of the matrix-vector multiplication and the N-tap FIR operation. These examples will help us to better understand the design of the digital signal processing (DSP) and fully-connected (neuron) blocks in chapter 5. We proceed with the analysis of space-time transformation which is a tool that produces the $N-1$-dimension systolic representation of a DSP circuit given its N-dimension space representation, and finally we conclude this chapter with the concept of unfolding. In chapter 3 we briefly discuss the elementary theory behind Deep Neural Networks. In chapter 4 we present some key points in the literature of DNN hardware accelerators, the most important of them being the metrics used to evaluate an architecture. In chapter 5 we discuss the proposed FPGA architecture which is implemented in VHDL and developed in Vivado 2018.3. We introduce the CNN software model and present the implementation details of every model's individual component. Finally, in chapter 6 we present the simulation and synthesis results for the proposed design.

2 Parallel Computation Theoretical Background

2.1 Systolic Architectures

High performance, special-purpose computer systems are used to meet specific application requirements or to off-load computations that are especially taxing to general-purpose computers. For this reason the concept of systolic architectures has been deployed as a general methodology for mapping high-level computations into hardware structures [11]. A systolic system is composed of a large number of processing elements (PEs) designed to work in parallel and perform simple tasks (fig. 2.1). This network most commonly is used as a co-processor in combination with a host computer which feeds the network data in a regural flow [19] and the PEs rhythmically compute and pass the data through the system in a way that resembles the functionality of the heart. The degree of the system's concurrency and thus the system's capability of performing fast computations is determined not only by the number of PEs but also by the underlying algorithm. Massive parallelism can be achieved if the algorithm is designed to introduce high degrees of pipelining and multiprocessing. Computational tasks can be conceptually classified into two families, compute-bound and I/O-bound computations. If the total number of operations is larger than the total number of input and output elements, then the computation is compute-bound, otherwise it is I/O-bound [11]. In this sense, matrix-matrix multiplication is a compute-bound task because every entry in a matrix is multiplied by all entries in some row or column of the other matrix but matrix-matrix addition is I/O-bound because the total number of additions is not larger than the total number of entries in the two matrices. In order to increase an I/O-bound computation, one would need to increase the memory bandwidth (size of data that can be read in one clock cycle) leading to the use of either expensive fast components or

complicated interleaved memories. On the other hand, compute-bound computations's speedup can be achieved by the relatively simple and inexpensive systolic approach.

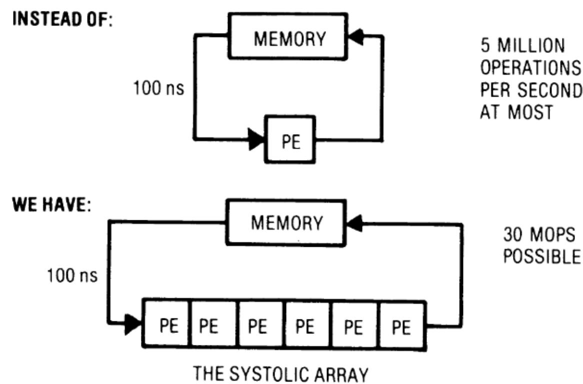


Figure 2.1: Basic principle of a systolic system [11].

When a large number of PEs work simultaneously, coordination and communication become essential, especially with VLSI technology where routing costs dominate the power, time and area required to implement a computation. Area and time play a critical role in the design of parallel systems. Lower bounds have been obtained for well-studied problems such as matrix-matrix multiplication, sorting and discrete Fourier transform which give rise to a trade-off between space and time. Results show that if A is the area used by a VLSI circuit to compute one of the n-input, n-output functions refered above and T is the time required for the computation, then the bound

$$AT^2 > \Omega(n^2)$$

must hold [15]. This inequality most importantly states that if an improved version of the algorithm takes less time to complete the computation - i.e: T' = T/2, then the area required by the circuit would increase - i.e: A' = 4A. This is due to the more complex routing required for the communication/coordination of the PEs.

2.1.1 Matrix-vector product

As a first example we will discuss the computation of a matrix-vector product using systolic design methodologies. Given an NxN matrix A and an N-vector $\vec{x}$, the goal is to compute the product $\vec{y} = A\vec{x}$ where each element of the result vector $\vec{y}$ is

$$y_i = \sum_{j=0}^{N-1} a_{ij} x_j$$

for $0 \leq i \leq N-1$. The simplest sequential method to complete the above task is the following:

1: **for** $i = 0 : N-1$ **do**
2: $\quad y_i \leftarrow 0$
3: $\quad$ **for** $j = 0 : N-1$ **do**
4: $\quad\quad y_i \leftarrow y_i + a_{ij} x_j$
5: $\quad$ **end for**
6: **end for**

which takes $O(N^2)$ time and uses N multiplications and N-1 additions for each y_i. This complexity may lead to unwanted delays and cause various problems on a real-time system, especially if the number of entries N is very large. To overcome this, one could use an N-cell linear array of PEs (fig. 2.2) and calculate the entire product in 2N-1 multiply/add steps ($O(N)$), thus achieving a reasonably efficient speedup over the naive sequential algorithm [14].

2.1.2 N-tap FIR filter

As an exercise we will utilize the above linear-array idea to design a parallel algorithm that implements an N-tap FIR filter. A finite impulse response (FIR) filter takes as input a stream of data x_1, x_2,..., and outputs a stream of data y_N, y_{N+1},..., where

$$y_t = \sum_{k=0}^{N-1} a_k x_{t-k}$$

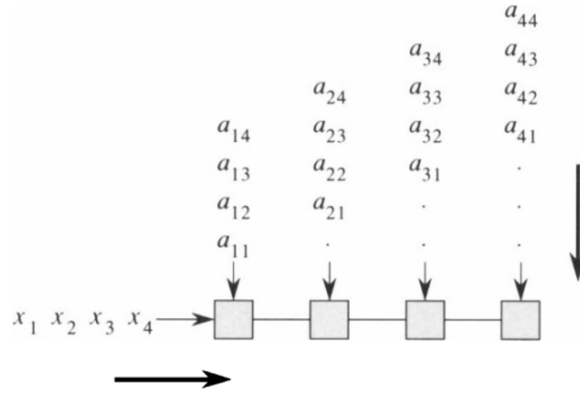


Figure 2.2: Computing the matrix-vector product $\vec{y} = A\vec{x}$ on an N-cell linear array for N=4. The ith cell computes y_i by adding the product $a_{ij}x_j$ to its memory at step i+j-1. [14].

for $t \geq N$ and $a_0, a_1, ..., a_{N-1}$ are the filter's coefficients. The algorithm works as follows. At each time step, every PE receives a value of input stream x from the left and a filter coefficient from the top. It multiplies these two values and adds the result with the content of its local memory. Then it updates its local memory with the result of the addition and passes the x value to its right neighbor (fig. 2.3). In short, every PE works as a Multiply and Accumulate (MAC) unit. For this algorithm to work however, we need a suitable permutation for the set of coefficients. More specifically, let $M = \{a_0, a_1, ..., a_{N-1}\}$ be the set of the filter's coefficients and let $C(M)$ denote the set of cyclic permutations of length N on the set M [21]. These permutations are of the form

$$f_p = \begin{pmatrix} a_0 & a_1 & \cdots & a_{N-1} \\ a_{(p+N-1)modN} & a_{(p+N-2)modN} & \cdots & a_{(p+0)modN} \end{pmatrix} \tag{2.1}$$

where $p = 0, 1, ..., N-1$. For the 4-tap FIR of fig. 2.3, where $N = 4$, we can use the following sets of permutations for the coefficients of each processing element

$$f_1 = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_0 & a_3 & a_2 & a_1 \end{pmatrix}$$

$$f_2 = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_1 & a_0 & a_3 & a_2 \end{pmatrix}$$

$$f_3 = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_2 & a_1 & a_0 & a_3 \end{pmatrix}$$

$$f_4 = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix}$$

In this way, Processing Element p outputs $y_{p+(j-1)N}$ at time $t = (j-1)N + 2p - 1$, where $p = 1, 2, 3, ..., N$ and $j = 1, 2, 3, ...$ is the local output of PE p.

Notice that the algorithm requires an overhead in order to feed each processing element with the proper sequence of filter coefficients. The naive way to do this would be to store the set $M = \{a_0, a_1, ..., a_{N-1}\}$ as rows into an NxN matrix, then reverse every row and finally rotate right each row by p ($p = 1, 2, ..., N$) which would require a total complexity of $O(N^3)$. Using the permutations of the form 2.1 however, we can reduce the overhead's complexity to $O(N^2)$. For more details regarding techniques for generating conflict free parallel address accessing please refer to [21].

2.2 Measuring the performance of parallel algorithms

The main purpose of writing parallel algorithms is to increase performance. Metrics such as speedup, work, efficiency and scalability are introduced in order to evaluate the proposed algorithms. The idea behind parallel algorithm design is to divide the work to be done among many processing cores as equally as possible. We define T_{seq} to be the run-time of a single processing core and $T_{parallel}$ to be the run-time of a multi-core system. Then **Speedup** is defined as

$$S = \frac{T_{seq}}{T_{parallel}} \tag{2.2}$$

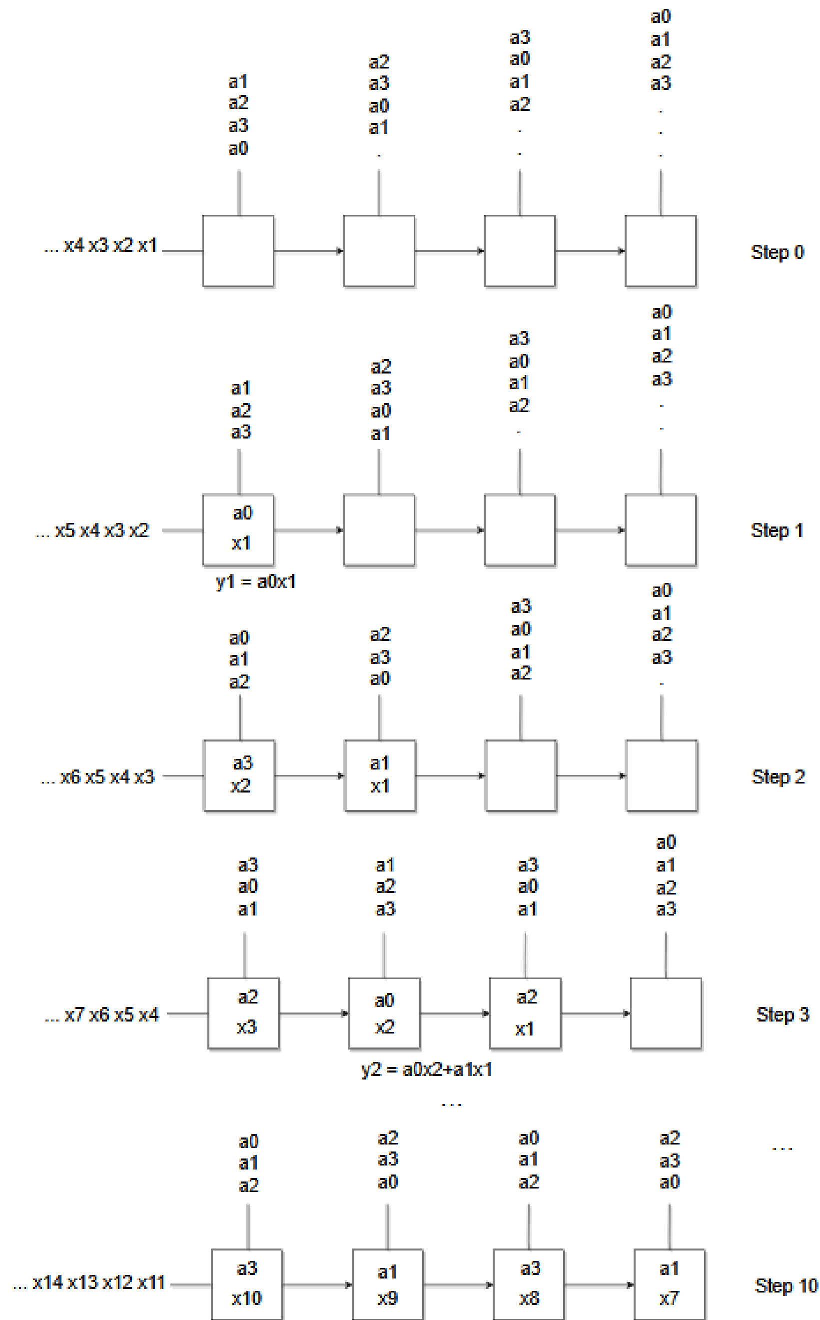


Figure 2.3: 4-tap FIR linear array. At step 10 processor p = 1 calculates its j = 4th (y_{13}) output which completes at step $t = 13$. Processor p = 2 calculates its j = 3rd (y_{10}) output which completes at step $t = 11$ and so on.

and measures how many times faster the execution of a program using multiple processing cores is compared to the same program being executed by a single processing core. We would

like to design algorithms that have as much speedup as possible. Given p processing cores we would like our parallel algorithm to be p times faster than the best sequential one. When $T_{seq} = p \cdot T_{parallel}$ happens we say that we achieved linear speedup and this is the best that we can hope for. In practice though it is unlikely to get linear speedup because the use of many processors/threads introduces overhead. For example in a shared-memory system mutual exclusion mechanisms are needed for protecting critical sections [18]. The transmition of data between processing cores and the synchronization between them is another example of overhead in a large distributed system. More formally we would define the running time of a parallel algorithm as

$$T_{parallel} = T_{serial}/P + T_{overhead} \tag{2.3}$$

Another important measure of parallel algorithms performance is the work performed by the algorithm. The work W is defined as

$$W = T_{parallel} \cdot P \tag{2.4}$$

Where $T_{parallel}$ is the run time of the parallel algorithm and P is the number of processors used. The notion of work measures the total processing effort needed for an algorithm and it accounts for inefficiencies caused by one or more processors being idle (or not performing a usefull task) during the computation. For example the work of the linear array parallel algorithm solving the matrix-vector problem discussed in the previous section has work $W = \Theta(N^2)$. Work can be defined alternatively as

$$W = N_1 + N_2 + ... + N_T \tag{2.5}$$

where N_i is the number of actively used processors during ith step. Using the notion of work we can measure the efficiency with which the processors are utilized. Efficiency E of a parallel algorithm is defined as

$$E = \Gamma/W = \frac{\Gamma}{T_{parallel} \cdot P} = \frac{S}{P} \tag{2.6}$$

where Γ is the running time of the best sequential algorithm and W is the work of the parallel algorithm. Given the best sequential algorithm for a specific program, efficiency can be increased if we decrease the work performed by the parallel algorithm. We would like to design algorithms that are both fast and efficient which means that we want the running time $T_{parallel}$ to be as small as popssible and the efficiency E to be as close to 1 as possible. The question of whether speed or efficiency is more important depends on many factors and varies widely among applications [14]. It is important to note that that running times, speedup and efficiency all depend on the problem size.

2.3 Space-time mapping

A formal methodology for designing systolic architectures is presented in [19]. A regular dependence graph (DG) is defined for every compute-bound computation and the edges of that graph represent precedence constraints. The DG is said to be regular if the presence of an edge in a certain direction at any node in the DG represents the presence of an edge in the same direction at all nodes in the DG. The DG is a space representation where no time instance is assigned to any computation. The goal is to map the DG into a space-time representation where each node corresponds to a certain processing element which is scheduled to operate at a certain time instance. This systolic design methodology efficiently maps an N-dimensional DG to a an (N-1)-dimensional systolic array.

The basic vectors involved in a DG $\rightarrow$ systolic array mapping are :

- Projection vector $d = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$
- Processor space vector $p^T = \begin{pmatrix} p_1 & p_2 \end{pmatrix}$
- Scheduling vector $s^T = \begin{pmatrix} s_1 & s_2 \end{pmatrix}$

- Hardware Utilization Efficiency, HUE = 1/|s^Td|

and the following holds true:

- Two nodes that are displaced by **d** or multiples of **d** are executed by the same processor.
- Any node with index $I^T = (i, j)$ would be executed by processor

$$p^T I = \begin{pmatrix} p_1 & p_2 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

- Any node with index I would be executed at time $s^T I$.
- Two tasks executed by the same processor are spaced $|s^T d|$ time units apart.

These vectors must satisfy some feasibility constraints:

- The processor space vector **p** and the projection vector **d** must be orthogonal. If points A and B in the graph, differ by the projection vector **d**, ie, $I_A - I_B$ = **d**, then they must be executed by the same processor, ie,

$$p^T I_A = p^T I_B \rightarrow p^T (I_A - I_B) = 0 \rightarrow \mathbf{p}^T \mathbf{d} = 0$$

- If A and B are mapped to the same processor, then they cannot be executed at the same time, ie,

$$\mathbf{s}^T I_A \neq \mathbf{s}^T I_B \rightarrow s^T (I_A - I_B) = 0 \rightarrow \mathbf{s}^T \mathbf{d} \neq 0$$

- Edge mapping: If an edge **e** exists in the DG, then an edge $\mathbf{p}^T \mathbf{e}$ is introduced in the systolic array with $\mathbf{s}^T \mathbf{e}$ delays.

Finally, with the above vectors and constraints defined, the space representation (DG) can be transformed into a space-time representation by interpreting one of the spatial dimensions as a temporal (time) dimension and using the following relation:

$$\begin{pmatrix} j' \\ t' \end{pmatrix} = \begin{pmatrix} \mathbf{p}^T \\ \mathbf{s}^T \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

As an example we will transform the DG of fig. 2.4 into a space-time mapping. The space representation in fig. 2.4 descibes a 3-tap FIR filter:

$$y(n) = w_0 x(n) + w_1 x(n-1) + w_2 x(n-2)$$

To implement the 3-tap FIR operation using the systolic design R1 (Results Stay, Inputs and Weights move in opposite directions) [11] we choose the following vectors:

$$\mathbf{d} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \mathbf{p}^T = \begin{pmatrix} 1 & 1 \end{pmatrix}, \mathbf{s}^T = \begin{pmatrix} 1 & -1 \end{pmatrix}$$

Then the space-time transformation is:

$$\begin{pmatrix} j' \\ t' \end{pmatrix} = \begin{pmatrix} p^T \\ s^T \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i+j \\ i-j \end{pmatrix}$$

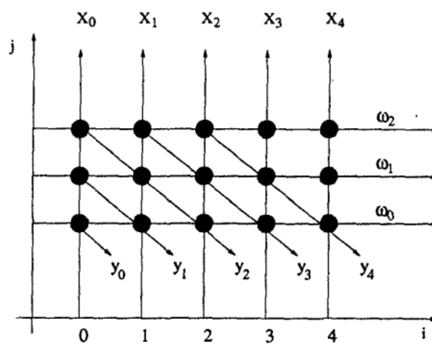


Figure 2.4: Space representation for 3-tap FIR filter.

In fig. 2.5 we see that all nodes that align to the (1, -1) direction group together as one processing element due to the projection vector **d**. Then, using the above result $\begin{pmatrix} j' \\ t' \end{pmatrix} = \begin{pmatrix} i+j \\ i-j \end{pmatrix}$ we can

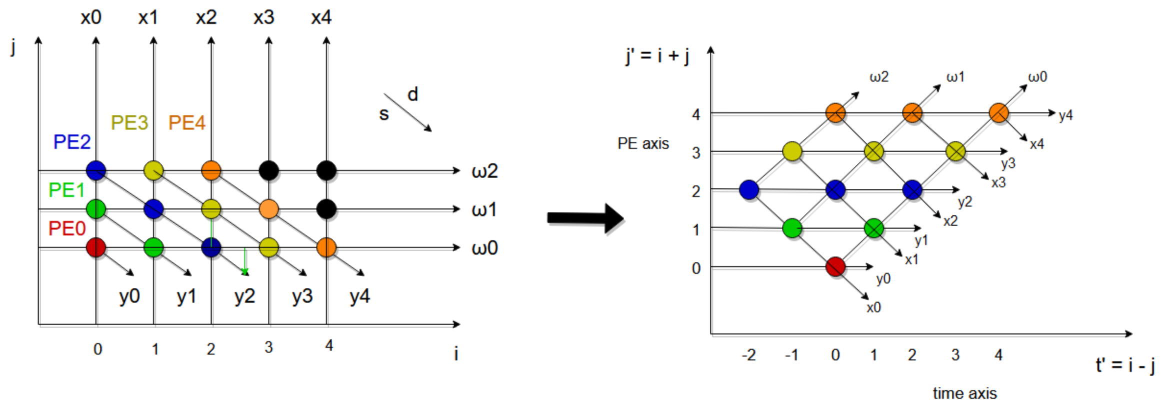


Figure 2.5: Space-time transformation for 3-tap FIR.

easily produce the space-time mapping of the 3-tap FIR's DG. It is also worth noting that each PE will be working every two clock cycles which we can verify from HUE = $1/|s^T d| = 1/2$ for the given values of $\mathbf{s}^T$ and $\mathbf{d}$.

Now let's return to the matrix-vector product which is described by a DG in a 2-D plane as in fig. 2.6 and thus it will be mapped to a 1-D systolic array of PEs.

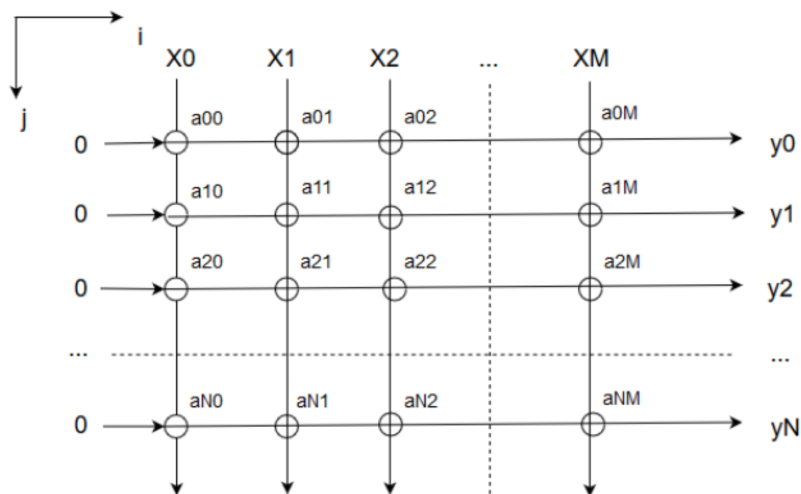


Figure 2.6: Dependence graph for matrix-vector product.

Assuming that each node (i, j) stores a matrix coefficient a_{ij} in its local memory we can express

the matrix-vector product in a standard Regular Iterative Algorithm (RIA) form :

$$a(i, j) = a(i, j)$$

$$X(i, j) = X(i, j - 1)$$

$$Y(i, j) = Y(i - 1, j) + X(i, j)a(i, j)$$

and then deduce the Reduced Dependence Graph of fig. 2.7.

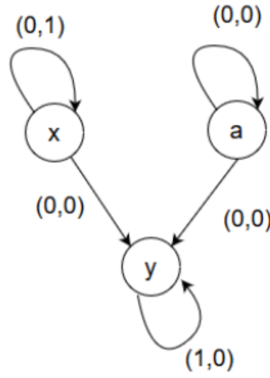


Figure 2.7: Reduced dependence graph for matrix-vector product.

The scheduling inequality that must hold for one edge X $\rightarrow$ Y is defined as

$$s^T I_Y + \gamma_Y \geq s^T I_X + \gamma_x + T_X$$

where $\mathbf{s}^T = \begin{pmatrix} s_1 & s_2 \end{pmatrix}$. Assuming that $T_{mult} = 1$, $T_{add} = 1$, $T_{comm} = 0$, the scheduling inequalities for the edges of fig. 2.7 are:

Edge X $\rightarrow$ X':

$$s^T I_{X'} + \gamma_{X'} \geq s^T I_X + \gamma_X + T_X \rightarrow s^T (I_{X'} - I_X) + \gamma_{X'} - \gamma_X \geq 0 \rightarrow$$

$$\rightarrow s^T e + \gamma_{X'} - \gamma_X \geq 0 \rightarrow \begin{pmatrix} s_1 & s_2 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \gamma_{X'} - \gamma_X \geq 0 \rightarrow$$

$$\rightarrow s_2 + \gamma_{X'} - \gamma_X \geq 0 \tag{2.7}$$

Edge a $\rightarrow$ a':

$$s^T I_{a'} + \gamma_{a'} \geq s^T I_a + \gamma_a + T_a \rightarrow s^T (I_{a'} - I_a) + \gamma_{a'} - \gamma_a \geq 0 \rightarrow$$

$$\rightarrow s^T e + \gamma_{a'} - \gamma_a \geq 0 \rightarrow \begin{pmatrix} s_1 & s_2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \gamma_{a'} - \gamma_a \geq 0 \rightarrow$$

$$\rightarrow \gamma_{a'} - \gamma_a \geq 0 \tag{2.8}$$

Edge a $\rightarrow$ Y':

$$s^T I_Y + \gamma_Y \geq s^T I_a + \gamma_a + T_a \rightarrow s^T (I_Y - I_a) + \gamma_Y - \gamma_a \geq 0 \rightarrow$$

$$\rightarrow s^T e + \gamma_Y - \gamma_a \geq 0 \rightarrow \begin{pmatrix} s_1 & s_2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \gamma_Y - \gamma_a \geq 0 \rightarrow$$

$$\rightarrow \gamma_Y - \gamma_a \geq 0 \tag{2.9}$$

Edge Y $\rightarrow$ Y':

$$s^T I_{Y'} + \gamma_{Y'} \geq s^T I_Y + \gamma_Y + T_Y \rightarrow s^T (I_{Y'} - I_Y) + \gamma_{Y'} - \gamma_Y \geq 1 \rightarrow$$

$$\rightarrow s^T e + \gamma_{Y'} - \gamma_Y \geq 0 \rightarrow \begin{pmatrix} s_1 & s_2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \gamma_{Y'} - \gamma_Y \geq 1 \rightarrow$$

$$\rightarrow s_1 + \gamma_{Y'} - \gamma_Y \geq 1 \tag{2.10}$$

Because the scheduling is linear we have $\gamma_X = \gamma_Y = \gamma_a = 0$. So we are searching solutions for the inequalities $s_2 \geq 0$ and $s_1 \geq 1$. One solution is $\mathbf{s}^T = \begin{pmatrix} 1 & 0 \end{pmatrix}$ which subject to

$$s^T d \neq 0 \rightarrow \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \neq 0 \rightarrow d_1 \neq 0$$

leads to $d^T = \begin{pmatrix} 1 & 0 \end{pmatrix}$. Also we need

$$p^T d = 0 \rightarrow \begin{pmatrix} p_1 & p_2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 0 \rightarrow p_1 = 0$$

so we choose $p^T = \begin{pmatrix} 0 & 1 \end{pmatrix}$.

Based on these vector values we construct the edge mapping array from which the final projected systolic array of fig. 2.8 is derived.

e	$\mathrm{p}^T\mathrm{e}$	$\mathrm{s}^T\mathrm{e}$
$\mathrm{e}_{(a)} = (00)^T$	0	0
$\mathrm{e}_{(X)} = (01)^T$	1	0
$\mathrm{e}_{(Y)} = (10)^T$	0	1

Table 2.1: Edge mapping for the systolic design that is produced by the space-time transformation of the matrix-vector problem

2.4 Unfolding

Unfolding is a transformation technique descibed in [19]. It can be applied to a DSP program in order to create a new one that computes more than one iterations of the original program

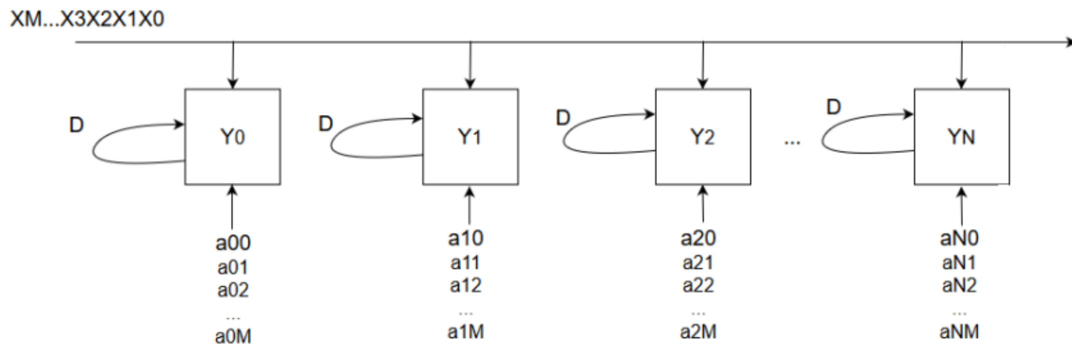


Figure 2.8: Systolic Architecture derived from table 2.1

concurrently. The Unfolding algorithm presented below when applied to a Data Flow Graph that describes the original DSP program produces its unfolded version. Unfolding is sometimes referred to also as loop-unrolling.

1: **Input:** Data Flow Graph of DSP program
2: **Output:** Unfolded Data Flow Graph of DSP program
3: **for** each node U of original DFG **do**
4: draw J nodes $U_0, U_1, ..., U_{J-1}$
5: **end for**
6: **for** each edge $U \rightarrow V$ with w delays in the original DFG **do**
7: draw J edges $U_i \rightarrow V_{(i+w)\%J}$ with $\lfloor \frac{i+w}{j} \rfloor$ delays for i = 0, 1, ..., J-1
8: **end for**

For example consider the program that implements the following equation (fig. 2.9):

$$y(n) = ay(n-9) + x(n) \tag{2.11}$$

If we replace index n with 2k and then with 2k+1 we get the following two equations:

$$y(2k) = ay(2k-9) + x(2k) \tag{2.12}$$

$$y(2k+1) = ay(2k-8) + x(2k+1) \tag{2.13}$$

which are two subsequent iterations of the original problem.

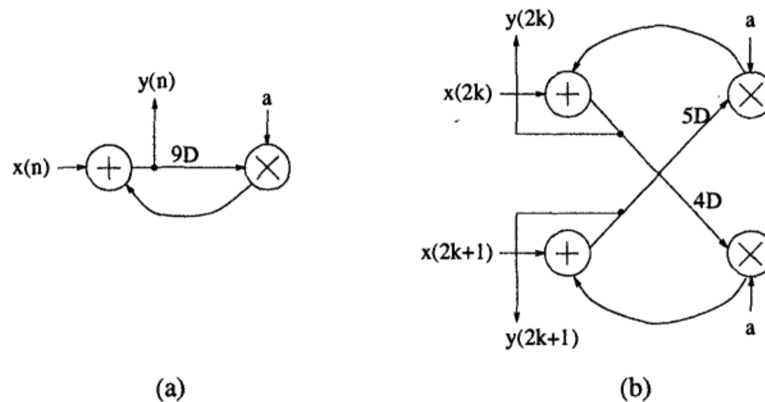


Figure 2.9: (a) Original DSP program describing eq. 2.10 for n=0 to ∞. (b) The 2-unfold DSP program describing eq. 2.11, 2.12 for k=0 to ∞.

Now we will construct a DFG for the above DSP program. Nodes A and B in fig. 2.10(a) represent input and output, respectively, and the nodes C and D represent addition and multiplication by a, respectively. After implementing the two steps of algorithm **??** with J=2 and the DFG of fig. 2.10(a) as input, we derive the 2-unfolded DFG of the program as in fig. 2.10(b).

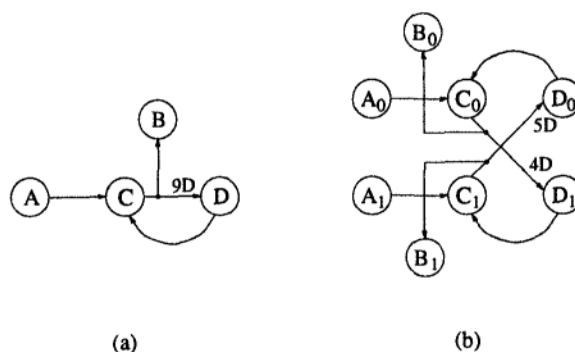


Figure 2.10: (a) DFG corresponding to the DSP program of fig. 2.9(a). (b) The 2-unfolded DFG corresponding to the 2-unfolded DSP program in fig. 2.9(b)

Now let us consider a problem in which we approach loop-unrolling in software. We want to

design a DFG that implements a 7-tap FIR filter and uses the architecture of fig. 2.11. The problem is that in each clock cycle, only one input sample can be read from the RAM, leading to the usage of only one MAC unit whereas the other two remain idle. To overcome this waste of resources, unfolding will be used in order to introduce the necessary level of parallelism.

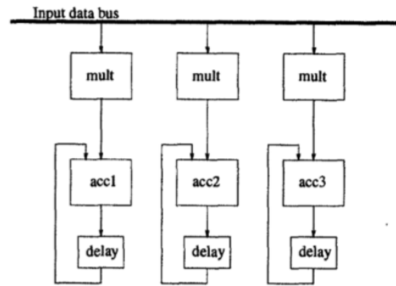


Figure 2.11: Programmable DSP with 3 MAC units.

Our goal is to implement the following equation :

$$y(n) = ax(n)+bx(n-1)+cx(n-2)+dx(n-3)+ex(n-4)+fx(n-5)+gx(n-6) \quad (2.14)$$

which describes the circuit of fig. 2.12, but we need all 3 MAC units to work concurrently. For this reason we first draw the DFG that describes the 7-tap FIR filter and then we apply the unfold algorithm with J=3.

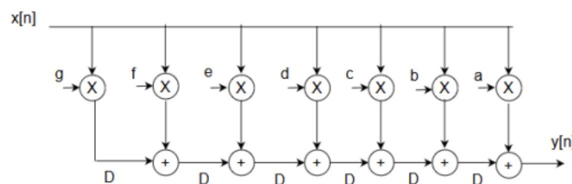


Figure 2.12: 7-tap FIR filer in transposed form.

The DFG of fig. 2.14 is derived after application of the unfolding algorithm. Notice that the total number of delays remains equal (as it should be) to the number of delays in the original DFG. After the unfold-transformation, three iterations of the program can be performed in the

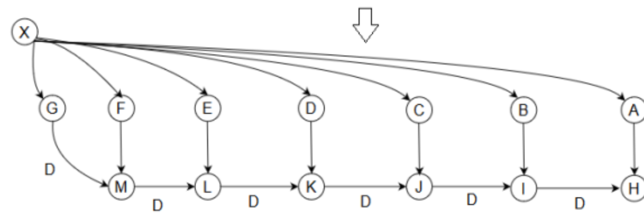


Figure 2.13: DFG of 7-tap FIR filter.

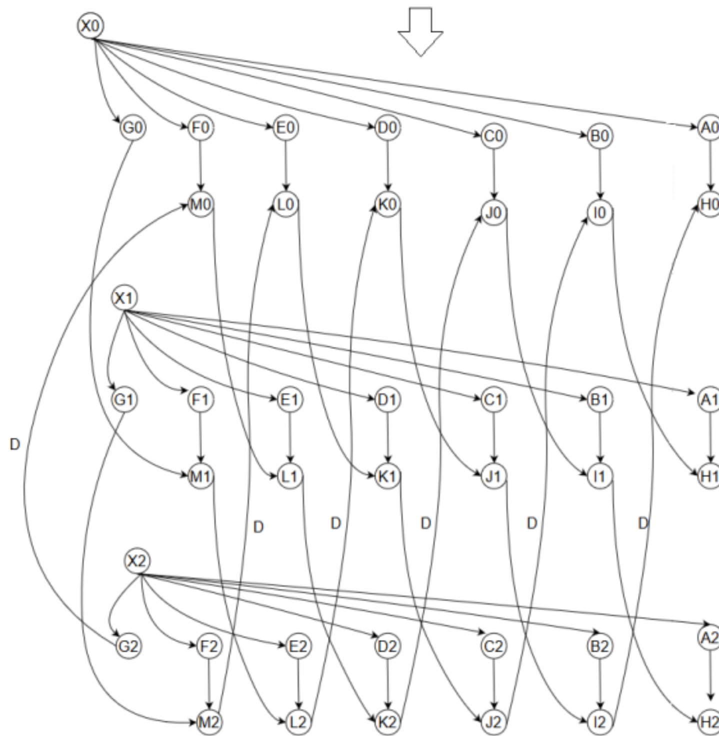


Figure 2.14: Unfolded DFG of 7-tap FIR with J=3.

same clock cycle, i.e, x[3k], x[3k+1], x[3k+2], but still, the three MAC units can not receive more than one input sample/cycle through the data bus with the current architecture. To proceed, we construct the acyclic precedence graph of fig. 2.14 by removing all edges with delays and finally we define an overlapped schedule in which all MACs are utilized in every clock cycle. Assuming that multiplication nodes A, B, C, D, E, F, G need time $T_{mult} = 3$ u.t, addition nodes H, I, J, K, L, M $T_{add} = 1$ u.t and input nodes X $T_{in} = 1$ u.t, the critical path is $T_{crit} = 6$ u.t. The overlap schedule of fig. 3.1 is rather complicated though, but it is the best one can achieve

given that only one data bus is present in the current architecture.

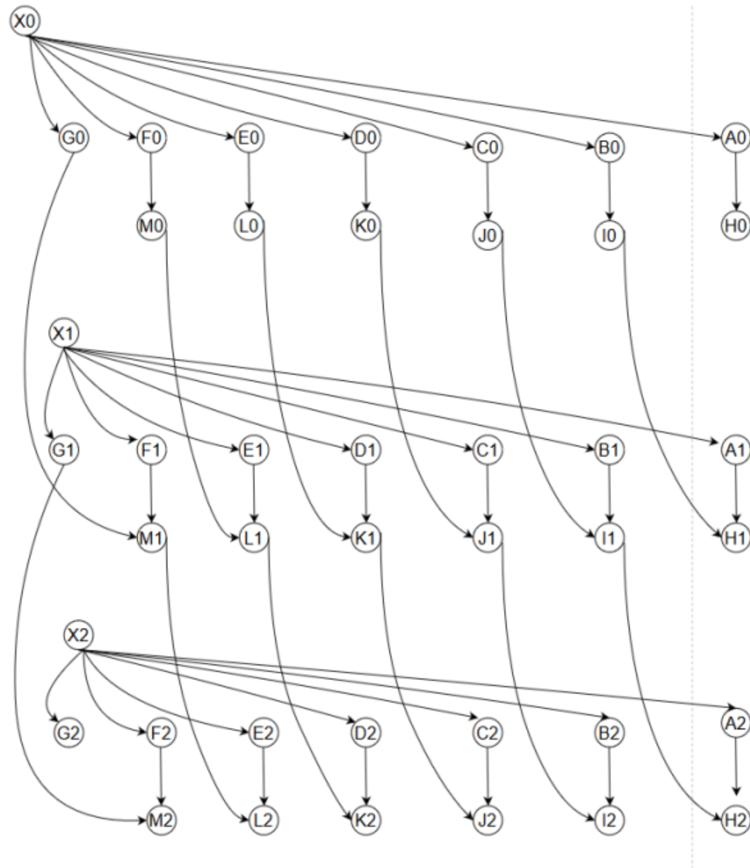


Figure 2.15: Acyclic precedence graph of 3-unfolded DFG.

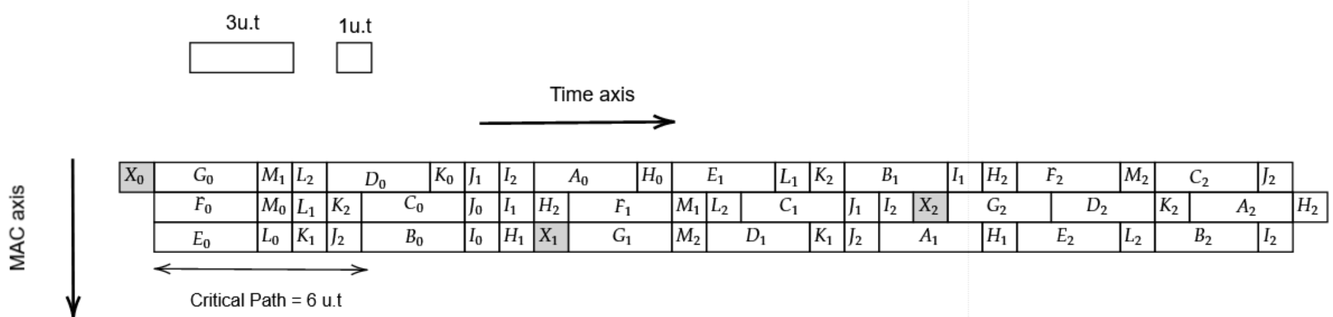


Figure 2.16: Overlap schedule for the acyclic precedence graph of fig. 2.15.

3 Deep Neural Networks Theoretical Background

Deep learning is a specific kind of machine learning. Machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions.

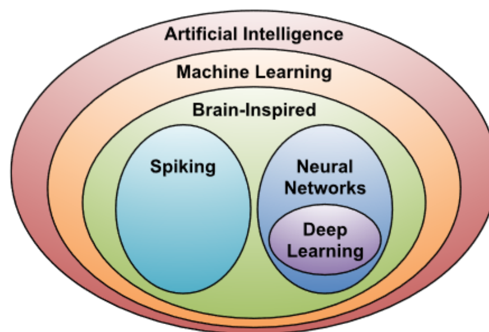


Figure 3.1: Deep learning in the context of artificial intelligence.

A definition of machine learning is the following: "A computer program is said to learn from experience E with respect to some class of tasks T and a performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."[7]. Following below are some of the classes that a task T might belong to.

- **Classification**: The computer program is asked to specify in which among k different categories some input belongs to.
- **Regression**: The computer is asked to predict a numerical value given some input.
- **Transcription**: The machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe the information in to discrete textual

form.

- **Machine translation**: The input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language (used in Natural Language Processing - NLP)
- **Anomaly detection**: The computer program shifts through a set of events or objects and flags some of them as being unusual or atypical. A typical example of this task is credit card fraud detection.
- **Denoising**: The machine learning algorithm is given as input a corrupted example $\tilde{x} \in R^n$ obtained by an unknown corruption process from a clean example **x** $\in R^n$ and the learner must predict the clean example from the corrupted version.

Machine learning technology is applied to many aspects of modern life: from web searches to content filtering on social networks to recommendations on e-commerce websites, object identification and various other computer vision related applications.

3.1 Parametric modeling

A large class of machine learning problems can be thought of as equivalent to a function estimation/approximation task. This idea goes way back to 1795 when Lagrange published the polynomial interpolation theorem which states the following. Given a set of coordinate pairs (x_i, y_i) with $0 \leq i \leq k$, where the x_i are called nodes and the y_i are called values, there exists a polynomial $L(x)$ that has degree $\leq k$ such that $L(x_i) = y_i$ for every $0 \leq i \leq k$. This task is more generally refered to as curve fitting and it is of great importance in machine learning. In parametric modeling, the functional dependence between the input x_i and the output y_i is defined via a set of unknown parameters θ whose number is fixed. A system is learning to estimate these parameters by digging in the information that resides in the available data set during the training phase. The usual path to follow is to adopt a functional form such as a linear or quadratic function, and try to estimate the associated unknown coefficients so that the graph of the function "passes through" the data and follows their deployment in space as close

as possible. The adopted functional form for the curve corresponding to fig 3.2.a for example is

$$\hat{y} = f_\theta(x) = \theta_0 + \theta_1 x \tag{3.1}$$

whilst for the curve of fig 3.2.b is

$$\hat{y} = f_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \tag{3.2}$$

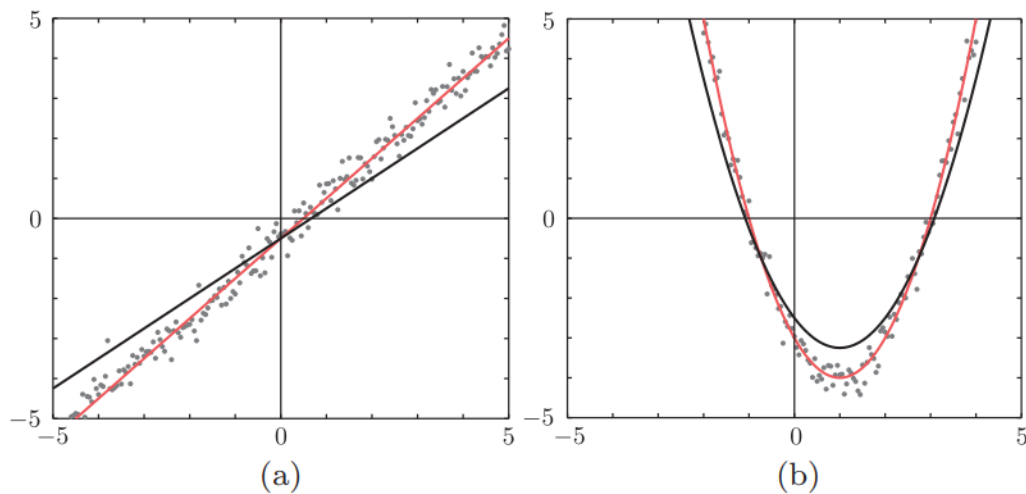


Figure 3.2: Fitting (a) a linear function and (b) a quadratic one. The red lines are the optimized ones.[24]

Given a training set $D = \{x_1, x_2, ..., x_N\} := \{y_1, y_2, ..., y_N\}$ and having adopted a parametric family of functions, one has to estimate for the unknown set of parameters. The more usual approach is to adopt a loss function that quantifies how much the predicted values $\hat{y}$ deviate from the measured values y that are known to the system through the data set. We adopt a nonnegative loss function $L(y, f_\theta(x))$ and define the total loss (cost) over all data points as

$$J(\theta) := \sum_{n=1}^{N} L(y_n, f_\theta(x_n)) \tag{3.3}$$

Assuming that $J(\theta)$ has a minimum, we iteratively update the values of the parameter vector $\vec{\theta}$

according to the following equation

$$\vec{\theta}' = \vec{\theta} - \epsilon \frac{\partial J}{\partial \vec{\theta}} \tag{3.4}$$

where ϵ is learning factor. The idea behind the above equation is that in order to find the values $\vec{\theta}_*$ that minimize J, we follow the gradient of J with respect to $\vec{\theta}$ in the opposite direction and we stop when we find a minimum. This optimization method is called gradient descent and it is the backbone of the back-propagation algorithm which is implemented by a neural network during the training phase.

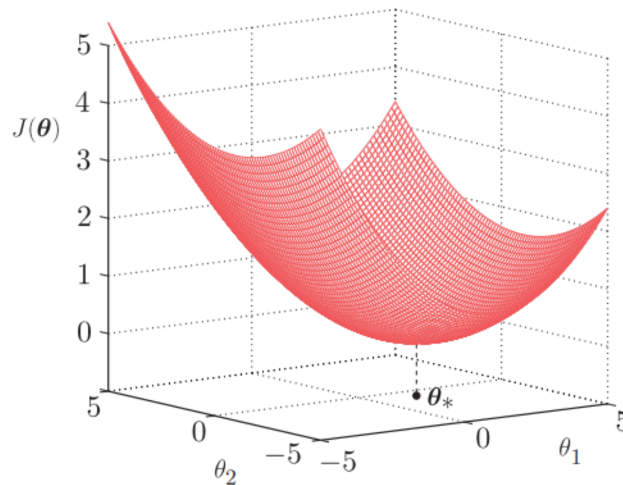


Figure 3.3: The least squares function is most commonly used as a cost function and it has a unique minimum at point θ_*.[24]

3.2 Neural Networks

Neural networks are learning machines, composed of a large number of neurons connected in a layered fashion. Every neuron in each layer is connected to every neuron in the next layer and the connections between them are called $synapses$. Every synapse has a value called $weight$ which indictates the importance of the specific connection on all paths that the information uses to flow forward. The $weights$ and $biases$ of all neurons are the $parameters$ of the network. Learning is achieved by adjusting the unknown parameters so as to minimize a pre-selected cost

function. Deep learning refers to learning networks with many layers of neurons capable of extracting specific features from raw input data that reside in multi-dimensional spaces in order to perform regression/classification tasks.

The fundamental block of a neural network is called the $perceptron$ (neuron) and it's structure is depicted in fig 3.4.

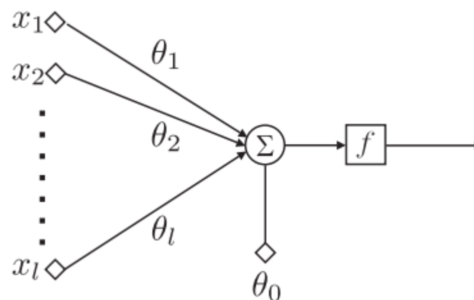


Figure 3.4: In the perceptron architecture all values x_i of the input vector are weighted by the respective synapses θ_i. The bias term θ_0 is added on the linear combination of the inputs and weights and finally the result is passed through a non linear function.

Every value x_i that is given as input to the perceptron is adjusted to a weight connection θ_i. The perceptron computes the dot product of the inputs and weights, then adds a bias value θ_0 and finally passes the result through a non linear function $f(\cdot)$.

$$y = f(\sum_{i=1}^{l} \theta_i x_i + \theta_0) \tag{3.5}$$

The non-linear function $f(\cdot)$ is usually one of the following

Rectified Linear Unit

$$f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{3.6}$$

Logistic Sigmoid

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.7}$$

Hyperbolic Tangent

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.8}$$

A system that is made of many perceptrons is called a multi-layered perceptron (neural network). The network architecture shown in fig 3.5 is the most commonly used in practice and it is easily generalized by considering additional hidden layers.

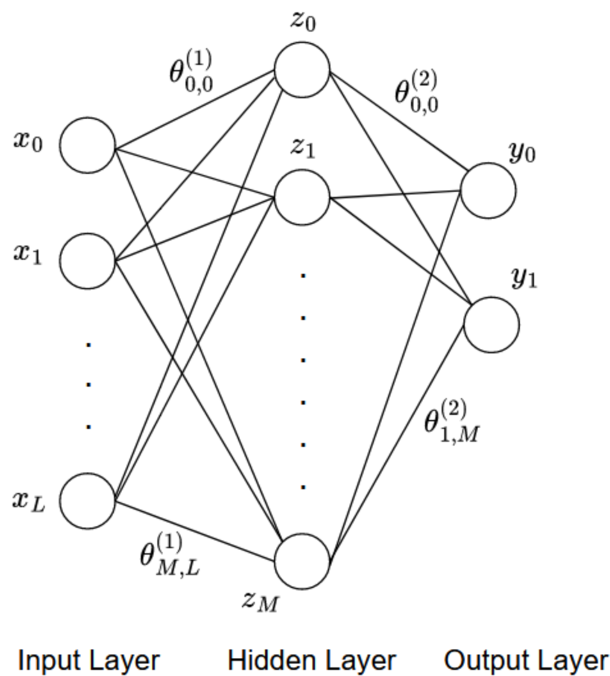


Figure 3.5: 2-layered fully connected neural network. The reason that this is described as a 2-layer is because there are two layers of weights (parameters) that need to be adjusted during the training phase.

In this architecture information flows from input to output and thus the network is called feed-forward. The system receives vector $\vec{x}$ as input and outputs vector $\vec{y}$ where each y_i is computed as:

$$y_i = f(\sum_{j=0}^{M} \theta^{(2)}_{i,j} \cdot f(\sum_{k=0}^{L} \theta^{(1)}_{j,k} x_k)) \tag{3.9}$$

One of the most important properties of feedforward networks with hidden layers (deep networks)

is that they provide a universal approximation framework [7]. This means that every feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid) can approximate any Borel measurable function from one finite-dimension (input) to another (output) with any desired nonzero amount of error, provided that the network is given enough hidden units. In this property lies the capability of deep neural networks to extract the functional relations that give rise to various patterns hidden inside the input space of the training data sets.

3.3 Convolutional Networks

A common type of deep neural networks is convolutional neural networks (CNNs). In these networks, each layer generates a successively higher level abstraction of the input data, called a feature map, which preserves essential yet unique information[23]. CNNs are widely used in a variety of applications including image processing, speech recognition, robotics etc.

Data are subject to distortion and various deteriorations due to noise. Before they are sent to the fixed-size input layer of a neural net, data must be approximately size-normalized and centered in the input field. Unfortunately, no such pre-processing can be perfect. Convolutional networks combine three architectural ideas to ensure some degree of shift and distortion invariance: local receptive fields [17], shared weights and spatial or temporal subsampling. Using local receptive fields, neurons can extract elementary visual features such as oriented edges and corners in images which are then combined by the higher layers to extract more complicated patterns[2]. This idea addresses a key property of images, which is that nearby pixels are more strongly correlated than more distant ones. Modern approaches to computer vision exploit this property by extracting local features that depend only on small subregions of the image. Information from such features can then be merged in later stages of processing in order to detect higher-order features and ultimately yield information about the image as a whole [3]. Hidden layers scan the input image with a neuron that has a local receptive field (filter-kernel) and the state of this neuron is stored in an output feature map. This operation is equivalent to a convolution with a small size kernel, followed by a "squashing" function. A convolutional layer is usually

composed of several feature maps with different weight filter-kernels in order to extract different elementary patterns from the input data.

The architecture of a convolutional network is shown in fig 3.6 and is structured in a series of stages. The first stages include two types of layers: convolutional and pooling layers. Units in convolutional layers scan the data using their small receptive field kernels and pass their local weighted sum through a non-linear function such as ReLU. The following pooling layer downsamples the output of the convLayer in order to merge semantically similar features into one. A typical pooling unit computes the maximum of a local path of units in one feature map. The basic idea behind CNNs and DNNs in general is to exploit the property that many natural signals are compositional hierarchies, in which higher-level features are obtained by composing lower-level ones [13]. At the last stages, the feature map produced by the final Convolution-Pooling layer pair will be a pattern composed of a combination of many low level features. This final feature map is flattened into a $n \times 1$ vector and then it is passed to the input of a fully connected layer. Finally the k-node output layer neurons computes their weighted sums and pass the results in a $softmax$ function. The purpose of a function such as softmax is to represent the outputs as probabilities that the input image belongs in one of k different classes.

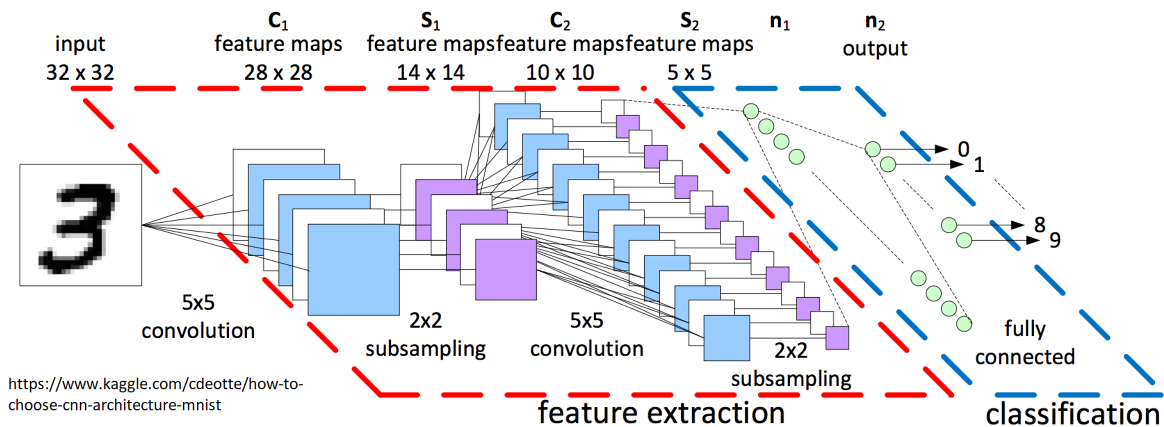


Figure 3.6: CNN for classification of 32x32 2D grayscale images of hand-written digits.

4 Accelerating Inference for DNNs

Deep Neural Networks (DNNs) are employed in a myriad of applications such as self-driving cars, cancer detection, complex gaming, information-retrieval search engines, mobile communication and more. In many of these domains, DNNs are now able to exceed human accuracy due to their ability to extract high level features from raw sensory data by using statistical learning in order to obtain an effective representation of an input space [22]. The constant improvement in classification accuracy though is achieved at the expense of higher computational and storage complexity. For example ResNet which scored 76.1% at ImageNet's top-1 classification accuracy in 2015 raised the number of GOPS (Giga Operations Per Second) to process a single 224x224 image to 22.6 in comparison to the 1.4 GOPS needed by the 2012 ImageNet winner AlexNet which scored a top-1 accuracy of 57.2%[26]. At the same time, many applications require DNNs to be deployed on mobile and embedded devices where memory, runtime budgets and energy reserves are strictly limited. In addition to the above constraints, the inherent paralelization ability of DNN models has led to the realization of FPGA platforms for real-time DNN processing in contrast to the traditional CPUs or the strong in parallel computations but power-hungry GPUs. FPGA's are a great candidate for accelerating DNN processing because they can achieve both high levels of parallelism and low power consumption but they also face challenges in performance and flexibility, the most important of those being:

- FGPA's support working frequency at 100-300MHz which is much less than CPU and GPU.
- Implementation of DNNs on FPGAs is much harder than that on CPUs or GPUs and development times are significantly larger because there are no development frameworks

such as TensorFlow, Keras, PyTorch or Caffe for hardware.

4.1 Training vs Inference

Since DNNs are part of machine learning algorithms they are deployed in two phases: the training phase and inference. During the training stage the algorithm implements the back-propagation algorithm which iteratively updates the network's parameters (connection-weights and biases) towards the direction of minimizing a cost function by using extensive chain-rule calculus, trying to find the function's local (or global) minimum, an optimization process known as gradient-descent. In this way the network model improves its predictive power. During the second phase, known as inference or feed-forward phase, the network uses the learned model's parameters to predict (regression) or classify (classification) new data samples never before seen by the network. For training there are some important considerations one should take when designing a neural processor. Backpropagation requires intermediate outputs of the network to be preserved for the backward computations and thus training has increased storage requirements and due to the gradient descent optimization technique, the precision requirement is generally higher than inference. A variety of techniques are used to improve the efficiency and robustness of training with the most common among them being batching and pruning. In batching, the loss is computed from multiple inputs before a single pass of weight updates is performed and in this way helps to speed up and stabilize the process. In prunning, weights (connections) that are not so important are cutted-of leading to a more sparse network. In a typical setup, DNNs are trained only once, on large GPU/FPGA clusters, but the inference is implemented each time a new data sample arrives as input. As a consequence, the literature mostly focuses on accelerating the inference phase [1], which is extremely important for real-time applications where new input data are received at high frequencies.

4.2 Key Metrics

Efficient processing of DNNs has been a significant research area over the past years. When one compares and evaluatets the strengths and weaknesses of different designs and proposed

techniques, one should consider a set of key metrics such as accuracy, throughput, latency, energy consumption, power consumption, cost, flexibility and scalability [22].

- **Accuracy** is used to indicate the quality of the result for a given task. For image classification it is measured as the percentage of correctly classified images.
- **Throughput** is an indication of the amount of data that can be processed or the number of executions of a task that can be completed in a given time period and is often critical to an application.
- **Latency** measures the time between the arrival of data in the system's input and the generated result at its output. For real-time interactive applications such as autonomous navigation and robotics, low latency is key.

  Throughput and latency are affected by several factors. One way of understanding this is by measuring the rate of inferences per second (feed-forward computations per second) and expand it in the context of a parallel system.

$$\frac{\text{inferences}}{\text{second}} = \frac{\text{operations}}{\text{second}} \times \frac{1}{\frac{\text{operations}}{\text{inference}}} \tag{4.1}$$

  The number of operations per second is dictated by both the hardware and the DNN model, but the number of operations per inference is determined strictly by the DNN model. For example a model with one hidden layer has less operations per inference than a deeper architecture. Interestingly, eq. 4.1 suggests that the throughput can be increased by careful co-design of software and hardware. The hardware design space is tightly coupled with the model's architecture space, i.e, the best neural architecture depends on the hardware and vice versa [9]. In the context of accelerators, many PEs are utilized to compute the feed-forward phase and thus the number of operations per second can be factorized even more as

$$\frac{\text{operations}}{\text{second}} = \left(\frac{1}{\frac{\text{cycles}}{\text{operation}}} \times \frac{\text{cycles}}{\text{second}}\right) \times \text{number of PEs} \times \text{utilization of PEs} \tag{4.2}$$

The first term refers to the peak throughput of a single PE while the second and third terms refer to the amount of parallelism and the ability of the design to effectively utilize this level of parallelism respectively. One can increase the peak throughput of a single PE by increasing the number of cycles per second, which corresponds to a higher clock frequency, by reducing the critical path at the circuit level, or the cycles per operations by introducing pipeline. The overall throughput could then be increased by increasing the number of PEs and thus the maximum number of MAC operations that can be performed in parallel. One must be careful though of the corresponding increase in chip area and the costs of complex wiring, or the decrease in on-chip storage area which gives rise to more off-chip memory reads which are expensive in terms of energy. Another important consideration is the hardware utilization efficiency.

$$\text{utilization of PEs} = \frac{\text{number of active PEs}}{\text{number of PEs}} \times \text{utilization of active PEs} \tag{4.3}$$

The first term of eq. 4.3 reflects the ability to distribute the workload to PEs, while the second term reflects how efficiently those active PEs are processing the workload. The goal is to distribute the workload to as many PEs as possible while at the same time design a dataflow such that the active PEs do not become idle while waiting for new data to arrive.

- **Energy efficiency** is used to indicate the amount of data that can be processed or the number of executions of a task that can be completed for a given amount of energy. High energy efficiency is important when processing DNNs at the edge in embedded devices with limited battery capacity (smartphones, smart sensors). Edge (online) processing may be preferred over the cloud (offline) for certain applications due to latency, privacy, or communication bandwidth limitations.

- **Power consumption** is used to indicate the amount of energy consumed per unit time.

  Power consumption and energy efficiency limit the throughput of the system as follows:

$$\frac{\text{inferences}}{\text{second}} \leq Max(\frac{\text{joules}}{\text{second}}) \times \frac{\text{inferences}}{\text{joule}} \tag{4.4}$$

Where inferences per joule can be factorized as

$$\frac{\text{inferences}}{\text{joule}} = \frac{\text{operations}}{\text{joule}} \times \frac{1}{\frac{\text{operations}}{\text{inference}}} \tag{4.5}$$

The amount of energy per operation can be broken down to the sum of energy required to transfer input/output data and the energy required by the MAC operation, ie,

$$E_{total} = E_{transfer} + E_{MAC} \tag{4.6}$$

With the use of pipeline and parallel processing, not only we can achieve higher computational speeds but also achieve low power consumption. For a 1st-order approximation analysis, the power consumption of a CMOS circuit can be estimated as

$$P = C_{total} V_0^2 f \tag{4.7}$$

where C_{total} is the capacitance of the circuit, V_0 is the supply voltage and f is the clock frequency, and the propagation delay can be written as:

$$T_{pd} = \frac{C_{charge} V_0}{k(V_0 - V_t)^2} \tag{4.8}$$

where C_{charge} is the capacitance that is charged/discharged in a single clock cycle (the capacitance along the critical path), V_0 is the sypply voltage, V_t is the threshold voltage and k is a function of the parameters related with the technology used such as $\frac{W}{L}$ MOSFET's gate width over length and C_{ox} the capacitance of the oxide.

By introducing M=3 level pipeline as in fig. 4.1 we can reduce the critical path of the system to 1/3 and also the capacitance to be charged/discharged in each clock cycle to $\frac{C_{charge}}{3}$. The total capacitance remains the same, but if the same clock speed f is maintained, only a fraction of the original capacitance is being charged/discharged in the same amount of

time which implies that the supply voltage can be reduced to βV_0, with $0 \leq \beta \leq 1$. Hence the power consumption of the pipelined system will be

$$P_{pip} = C_{total}\beta^2 V_0^2 f = \beta^2 P_{seq} \tag{4.9}$$

and its propagation delay is given by

$$T_{pip} = \frac{\frac{C_{charge}}{M}\beta V_0}{k(\beta V_0 - V_t)^2} \tag{4.10}$$

Usually the clock period of a circuit is set equal to the maximum propagation delay, and since we assumed that the pipeline circuit preserves the clock speed we can solve the following quadratic equation to find β

$$T_{pd} = T_{pip} \rightarrow M(\beta V_0 - V_t)^2 = \beta(V_0 - V_t)^2 \tag{4.11}$$

The energy consumption is dominated by the data movement whose capacitance tends to be much higher than the capacitance for arithmetic operations. The further the data needs to travel, the larger the capacitance and thus the switching activity, resulting to increased energy consumption. For this reason it is really important to design workload flows where data is reused as much as possible, in order to reduce the number of off-chip memory accesses, which are the most expensive operation as we see in fig. 4.3

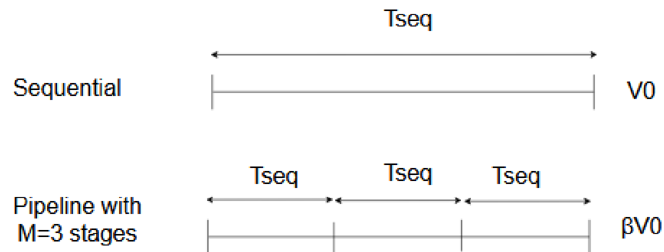
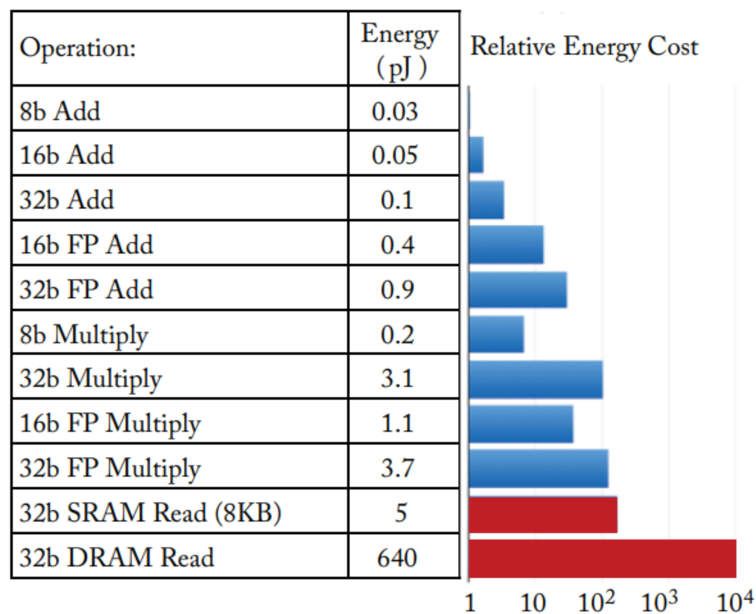


Figure 4.1: Critical path length for sequential and 3-level pipelined systems

- **Hardware cost** is used to evaluate whether the system is financially viable and it is mostly determined by the chip area in conjuction with the process technology (e.g BiCMOS).
- **Flexibility** refers to the range of models that the dedicated DNN processor can support and also to the ability of the software environment that maps the model, to exploit the full capabilities of the hardware.
- **Scalability** refers to how well a design can be scaled up to achieve higher throughput and energy efficiency when increasing the amount of resources (number of PEs and on-chip storage). Ideally the throughput would scale linearly and proportionally with the number of PEs and similarly the energy efficiency would improve with more on-chip storage.



Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Multiply	0.2
32b Multiply	3.1
16b FP Multiply	1.1
32b FP Multiply	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640

Figure 4.2: Energy consumption for various arithmetic operations and memory accesses in a 45 nm technology [8]

4.3 Taxonomy of accelerator architectures

Today's systems are a hierarchical composition of compute cores organized as nodes that are connected to memory and I/O devices through a coherent system bus. Conceptually, accelerators can be attached at all levels of this hierarchy [4]. The goal is to increase the system's performance, but as we saw in the previous section, there are various metrics that play a significant role in

defining the granularity and integration of the accelerator. Some popular categories that have emerged through the years at the system level are the following:

- **Integrated specialized units** such as floating point units (FPUs) and vector units that offload the CPU from computationally intensive operations.
- **Specialized ASICs** such as Google's Tensor Processing Unit (TPU) [10] that provide special-purpose processing, typically targeted to a set of algorithms in a particular domain.
- **Generated accelerators**, which are reconfigurable devices such as FPGAs allowing applications written in high-level languages (VHDL-Verilog) to be directly compiled into hardware.

The DNN architectures can be broadly divided based on the area in which the architecture has been primarily optimized. *ALU* category is the one where the MAC units are modified such that the accelerator can have large computing resources and flexibility to achieve optimal performance with variable bit precision. In the *Dataflow* category, the parameters (weights, activations, partial sums) are managed such that the overall data movement energy is reduced, and in *Sparsity*, the data is managed such that the matrix multiplication units can avoid zero multiplications effectively [16].

4.4 Hardware architectures for kernel computations in DNN processing

Multiply-and-accumulate (MAC) operation is the basic component of matrix-matrix and matrix-vector multiplication. Since the heaviest processing in a DNN often maps to a matrix multiplication, the fundamental computation of both the convolutional and fully-connected layers are MAC operations which can be easily parallelized. Highly parallel compute paradigms can be used for matrix multiplication. They are categorized as being either temporal or spatial [22]. In $temporal$ architectures DNN algorithms can be mapped and optimized on CPUs and GPUs in a way that the number of multiplications is reduced whilst the kernel computations can be ordered in tiles to improve memory subsystem behavior. In $spatial$ architectures, carefully designed dataflows can increase data reuse by utilizing low cost memories in the memory hierarchy

leading to reduced energy consumption and also optimized data movement.

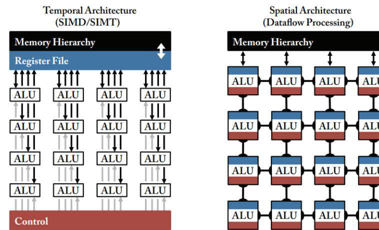


Figure 4.3: Two main paradigms of highly parallel compute paradigms [22]

In *temporal* architectures all the ALUs share the same control and register file (memory) in order to perform parallel MAC operations. Well studied techniques that utilize this scheme are Toelpiz matrix multiplication [22] and Winograd transform [19]. Many software libraries such as Open BLAS, IntelMKL (for CPU) and cuBLAS, cuDNN (for GPU) have been designed for optimized implementation of matrix multiplication. General matrix multiplication (GEMM) algorithms consider a granular partition of matrices and the computations are performed in a layered fashion[5]. The goal is to find a high-performance implementation of the smallest kernel in terms of data movement between memory hierarchy layers, and proceed in a bottom to top approach.

In *spatial* architectures the ALUs are connected with each other forming a mesh-array of processing elements. Each processng element has its own control logic and memory and thus is capable of executing different instructions on the data. Data are loaded on these distributed systems by higher layers of the memory hierarchy.

DNNs often use the same piece of data for multiple MAC operations. This property results in three basic forms of data reuse that reduce data movement costs (fig 4.4):

- Input feature map reuse : where different filters (different convolution neurons) are applied to the same input feature map in order to generate multiple output feature maps.
- Filter reuse: where a batch of input feature maps is processed by the same filter.
- Convolutional reuse: where the filter slides across different (often overlapping) positions in the input feature map to produce an output feature map.

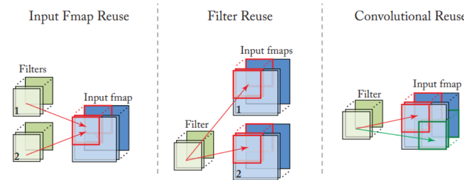


Figure 4.4: Data reuse opportunities in a convolutional layer [22]

There are several commonly used design patterns that exploit data reuse and can be categorized into a taxonomy of dataflows. Weight stationary, Output stationary, Input stationary and Row stationary dataflows. The weight-stationary dataflow is designed to minimize the energy consumption of reading weights by maximizing the reuse of weights from the local register file at each processing element. The output-stationary dataflow is designed to minimze the energy consumption of reading and writing the partial sums keeping the accumulation of partial sums for the same output activation value local in the register file. Similarly, the input-stationary dataflow minimizes the energy consumption of reading input activations. Each input activation runs through as many MACs as possible in the processing element. Finally the row-stationary dataflow aims to maximize the reuse and accumulation at the register file level of all types of data (weights, input activations and partial sums) for overall energy efficiency [22].

In this work, the weight-stationary dataflow depicted in fig 4.5 is chosen as the design-pattern for the basic component of the accelerator's convolutional layer, the DSP block. More details regarding the implementation and the adjustments made for the chosen CNN model are given in chapter 5.

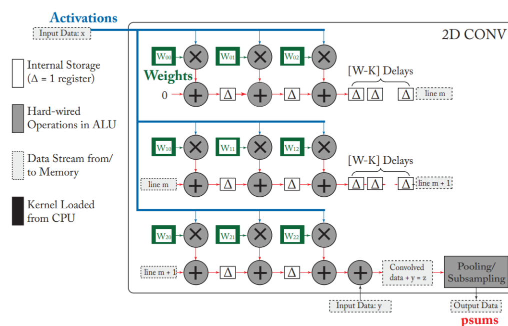


Figure 4.5: Weight-stationary dataflow as implemented in nn-X or neuFlow [6]

5 Design of Hardware Accelerator for CNN on FPGA

5.1 The CNN model

The model was trained with the MNSIT dataset which contains 60K handwritten digit grayscale images. From this dataset 50K images were used for the training and the rest 10K were used for testing the model's accuracy. The model was created in google's Colab using the Keras python Deep Learning API. The model's architecture consists of :

- one convolution layer containing 32 filters of 5x5 kernel size each. The convolution layer takes as input the 28x28x1 input image and outputs a 24x24x32 feature map,
- a ReLU activation layer,
- a max pooling layer which extracts the max value of 2x2 windows with stride 2 in order to downsample the convolutional output to a 12x12x32 array,
- a ReLU activation layer,
- a 30-node fully connected layer which takes as input the flattened (4608x1) output of the max pooling layer and
- a 10-node output layer with softmax activation that produces the probabilities that a given input image belongs in one of the 10 different classes (digits).

The model was compiled using the Adam optimizer and the categorical cross-entropy as the loss function [24]. Training was performed with a 33% validation split [25] and after 8 epochs the model achieved 97.73% accuracy on the test set. One of the issues that this thesis does not address is the fact that the model's parameters are in float-32 bit numerical representation whilst

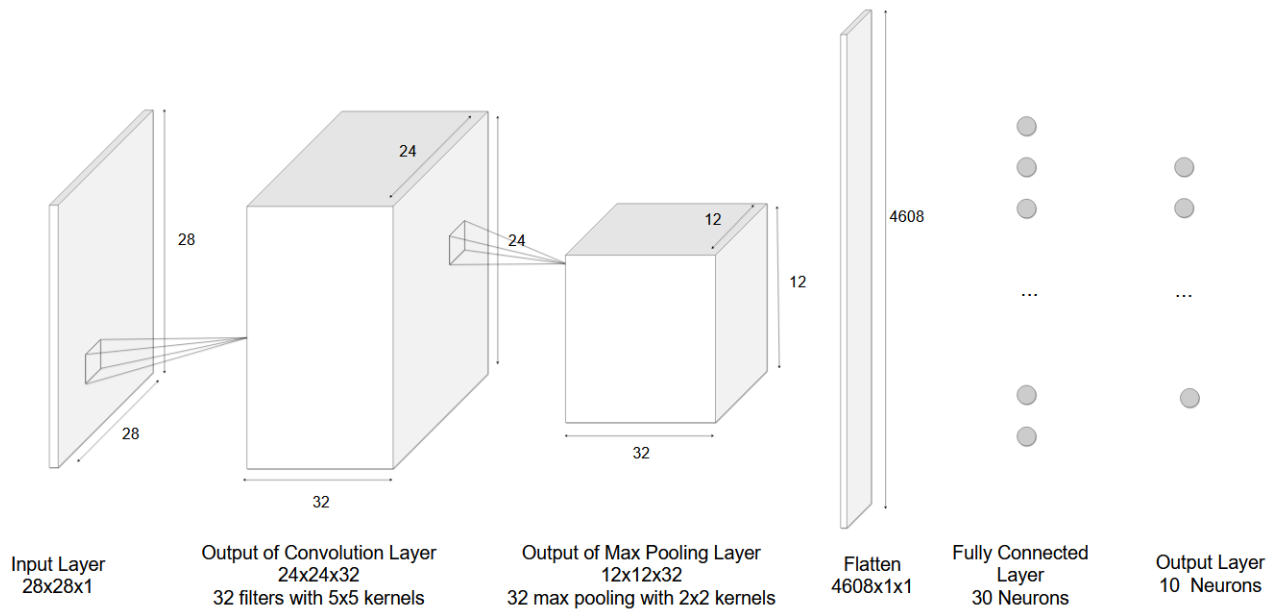


Figure 5.1: CNN model architecture

the FPGA uses fixed-point arithmetic. In a real use-case scenario where one would want to load the trained neural network into an embedded device, one should convert the neural network algorithm to a fixed-point using Matlab and C-code. The floating to fixed-point conversion is not expected to degrade the prediction accuracy of the classifier as it is shown in [12]. In the following sections we describe the FPGA architecture and we represent every pixel of the 28x28 input image and hence the weights of the filter kernels as 16-bit integers. The results of the convolution layer and the pooling layers are 32-bit integers (due to integer multiplications). The results of the Fully connected layer are 64-bit integers which are truncated into 32-bit integers before they enter to the Output layer which finally produces ten 64-bit integer results. Many improvements can be made as far as concerning the reduction of the number of bits used at the final output.

5.2 Architectures for efficient 2D convolution

During inference, the most intensive computations are performed by the convolutional and the fully connected layers. The 2D convolution operation is basically a sliding of a filter-kernel

window over a 2D image with a particular stride (sliding step) while at the same time multiplying each filter value with the corresponding pixel and accumulating the result.

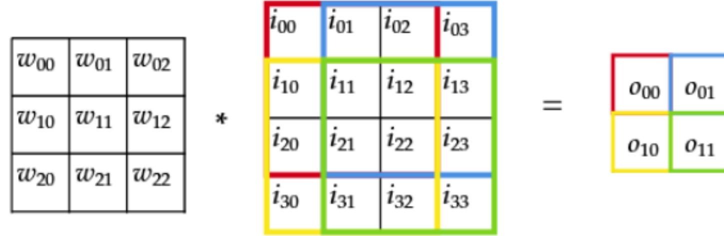


Figure 5.2: 2D convolution on a 4x4 pixel image using a 3x3 filter-kernel with stride 1 and zero padding.

In fig. 5.2 the convolution is performed with a 3x3 kernel on a 4x4 image using zero padding and stride S = 1 for both dimensions resulting in a 2x2 output. More generally the output dimensionality can be computed by the formula :

$$O_H \times O_W = (N_H - W_H + S_H)/S_H \times (N_W - W_W + S_W)/S_W \tag{5.1}$$

where N_H, N_W are the height and width of the input image and W_H and W_W are the height and width of the filter-kernel. The convolution operation is defined as :

$$o_{m,n} = \sum_{i=0}^{W_H-1} \sum_{j=0}^{W_W-1} i_{m+i,n+j} w_{i,j} \tag{5.2}$$

To compute the convolution of fig. 5.2 we need 4x9 = 36 multiplications and 4x8 = 32 additions as shown in the following equations:

$$o_{0,0} = i_{0,0} \cdot w_{0,0} + i_{0,1} \cdot w_{0,1} + i_{0,2} \cdot w_{0,2} + i_{1,0} \cdot w_{1,0} + i_{1,1} \cdot w_{1,1} + i_{1,2} \cdot w_{1,2} + i_{2,0} \cdot w_{2,0} + i_{2,1} \cdot w_{2,1} + i_{2,2} \cdot w_{2,2} \tag{5.3}$$

$$o_{0,1} = i_{0,1} \cdot w_{0,0} + i_{0,2} \cdot w_{0,1} + i_{0,3} \cdot w_{0,2} + i_{1,1} \cdot w_{1,0} + i_{1,2} \cdot w_{1,1} + i_{1,3} \cdot w_{1,2} + i_{2,1} \cdot w_{2,0} + i_{2,2} \cdot w_{2,1} + i_{2,3} \cdot w_{2,2} \tag{5.4}$$

$$o_{1,0} = i_{1,0}{\cdot}w_{0,0}{+}i_{1,1}{\cdot}w_{0,1}{+}i_{1,2}{\cdot}w_{0,2}{+}i_{2,0}{\cdot}w_{1,0}{+}i_{2,1}{\cdot}w_{1,1}{+}i_{2,2}{\cdot}w_{1,2}{+}i_{3,0}{\cdot}w_{2,0}{+}i_{3,1}{\cdot}w_{2,1}{+}i_{3,2}{\cdot}w_{2,2} \quad (5.5)$$

$$o_{1,1} = i_{1,1}{\cdot}w_{0,0}{+}i_{1,2}{\cdot}w_{0,1}{+}i_{1,3}{\cdot}w_{0,2}{+}i_{2,1}{\cdot}w_{1,0}{+}i_{2,2}{\cdot}w_{1,1}{+}i_{2,3}{\cdot}w_{1,2}{+}i_{3,1}{\cdot}w_{2,0}{+}i_{3,2}{\cdot}w_{2,1}{+}i_{3,3}{\cdot}w_{2,2} \quad (5.6)$$

Assuming that each multiplier takes 3 u.t (unit time) and each adder takes 1 u.t the sequential approach needs 140 u.t to compute the convolution. At this step we can improve the performance by using the weight stationary dataflow architecture of fig. 5.3[22].

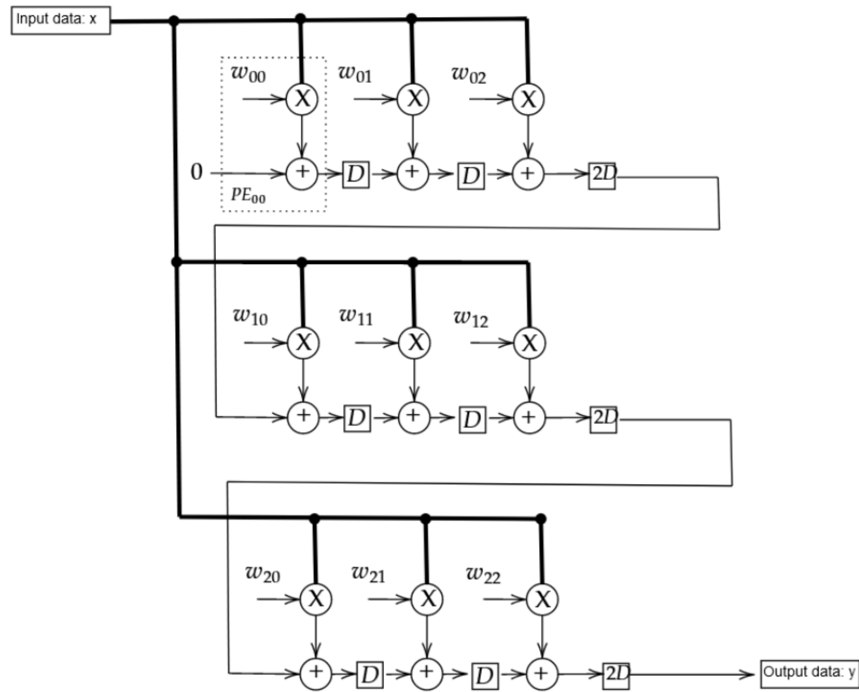


Figure 5.3: Weight Stationary Dataflow architecture

The architecture of fig. 5.3 is pipelined and has $T_{crit} = 4u.t$ critical path (1 adder + 1 multiplier). The lower bound for the clock period T is $T \geq 4$ u.t. With this clock rate the sequential algorithm completes its calculations after 140/4 = 35 clock cycles, whilst the pipelined approach completes its calculations after 16 clock cycles as we see in table 7.3 and achieves a speedup $S = \lfloor \frac{T_S}{T_P} \rfloor = \lfloor \frac{35}{16} \rfloor = 2$. This dataflow not only increases the throughput but also reduces energy consumption because all weight kernel values can be stored locally and the expensive off-chip memory accesses are minimized. Each multiplier/adder pair in fig. 5.3 is perceived as

a Processing Element (PE) or a Multiply and Accumulate unit (MAC). Every PE multiplies its input with a weight value that is stored locally and accumulates the result by adding the partial sum it receives from the preceding PE as depicted in fig. 5.6. If we further modify the circuit of fig. 5.3 such that every row of PEs receives values from different input rows we can reduce the number of clock cycles even more. The architecture of fig. 5.5 is the one that we use as the basic DSP block of the convolution layer in the FPGA accelerator. With this dataflow the convolution is completed after 11 clock cycles as we see in table 7.7. Full parallelization is exploited in [12] where each DSP block receives a whole window of input pixels per clock cycle and an adder tree sums the results of the multiplications in $O(log(N))$.

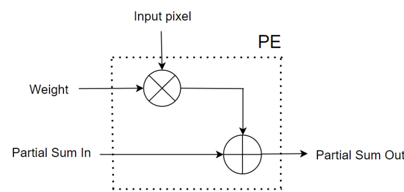


Figure 5.4: Processing Element (PE) performing MAC operation

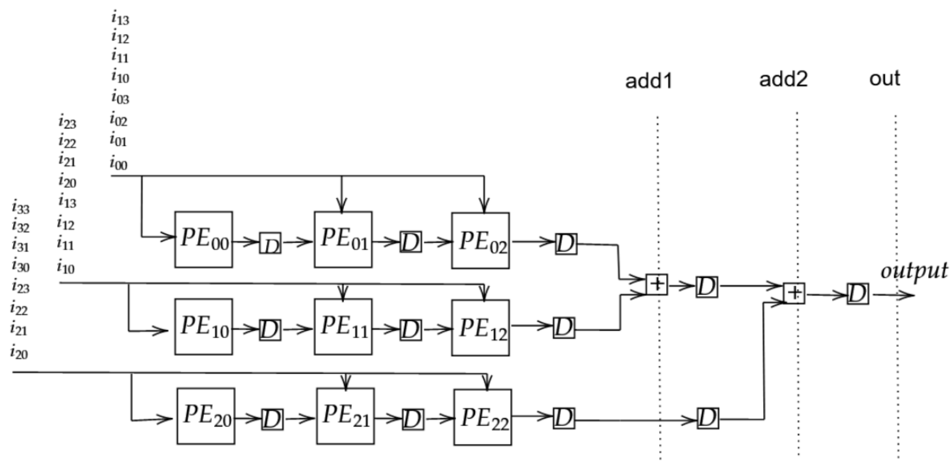


Figure 5.5: Weight Stationary Dataflow architecture unfolded.

5.3 Overall architecture

The FPGA architecture of the CNN accelerator is based on [12] and is divided in four fundamental blocks that correspond to the software model: a) the Input Layer b) the Convolution Layer c) the Pooling Layer and d) the Fully Connected Layer. The main difference in our approach is the DSP block presented in the previous section, which also leads to some adjustments in the functionality of the Input, Pooling and Fully connected Layers. The advantages of the FPGA architecture are a) the exploitation of parallelism for the CNN model which is achieved by dividing the 32 filter-kernel convolutions into 32 DSP blocks that will compute the calculations taking place within each filter in parallel and b) it is a highly pipelined design that improves the throughput and the power consumption.

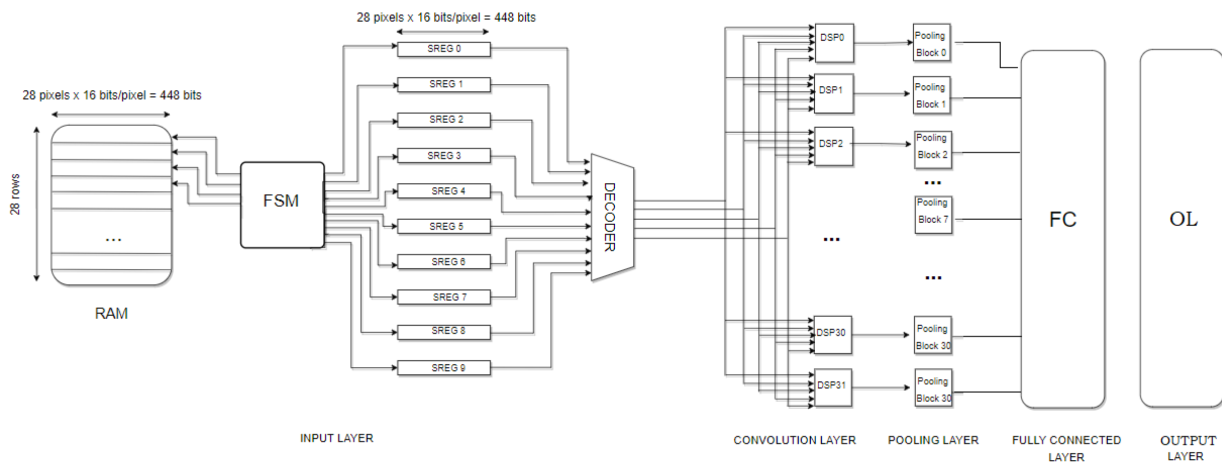


Figure 5.6: Hardware Accelerator FPGA Architecture

5.4 Input Layer

The input layer receives the input image of the CNN and stores it to the on-chip RAM memory. Then it creates the input that is fed to the Convolution Layer. Input Layer stores the 28x28 input image row by row in a RAM so that the Finite State Machine (FSM) can read a whole row (28 pixels) in a single clock cycle. The FSM is responsible for reading one image row from the RAM and then writing it into a shift register. Then the shift register outputs 1 pixel per clock cycle

and feeds one row of PEs in every DSP block of the convolutional layer. We need 5 registers because each DSP block comprises of 5 rows with 5 PEs each. There are two sets of registers. When all 5 registers of the 1st set output the final pixel of their load, the FSM needs to read the RAM again and load the 5 registers with subsequent rows for the computation to be continued. This means that the DSP blocks will remain idle for 2 clock cycles each time we need to feed the Convolution Layer with new rows. To improve this we utilize a 2nd set of registers which are loaded with subsequent image rows and remain idle until the 1st set has finished its shifting. At this point the registers of the 2nd set are enabled and start shifting their inputs. The FSM starts reading the RAM and then writes the registers of the 1st set with the next image rows and so on. A decoder is used to choose which set of registers is forwarded to the convolutional layer each time. With this schema the DSP blocks remain idle for 1 clock cycle. When a register of one set finishes shifting row i, it updates its input with row i+2 because the row i+1 is already loaded to the symmetric register of the other set (tables 7.8, 7.9). The required time from the moment that the first input image row is received until the shift registers starts sending pixels to the convolution layer is 38 clock cycles. The input image needs to be transmitted by a host computer into the external interfaces of the FPGA through methods such as PCIe, Ethernet or USB but this is not implemented in this project.

5.5 Convolution Layer

The main processing elements of the convolution layer are the DSP blocks presented in fig. 5.7. The convolution layer includes a total of 32 such DSP blocks. Each DSP block computes the convolution of the 28x28 input image with the 5x5 kernel of one of the 32 filters that are defined in the software model. It consists of 25 PEs divided into 5 rows, an adder tree which sums the result of PE row computations in O(log(N)), an adder for the bias and a multiplexer that implements the ReLU function. Every row of PEs in the design receives pixels from different input image rows and the result is accumulated. The output of the convolution layer is a 24x24x32 array. Every DSP block produces 24x24=576 output elements (convolution of the filer kernel with 576 image windows). Accumulation through the PE rows has a delay of 5 clock cycles whereas the delay of the adder tree summation is 4 clock cycles, resulting in a total latency of 9 clock cycles,

after which the output elements are produced in every clock cycle. Every time the last element of an output row $o_{i,23}$ is received though, the next row's first element $o_{i+1,0}$ is received after 5 clock cycles. Intermediate elements that reach the output during that time period are redundant data and are not part of the DSP's 24x24 output array and thus should not be written in the FIFO memories of the Pooling Layer's.

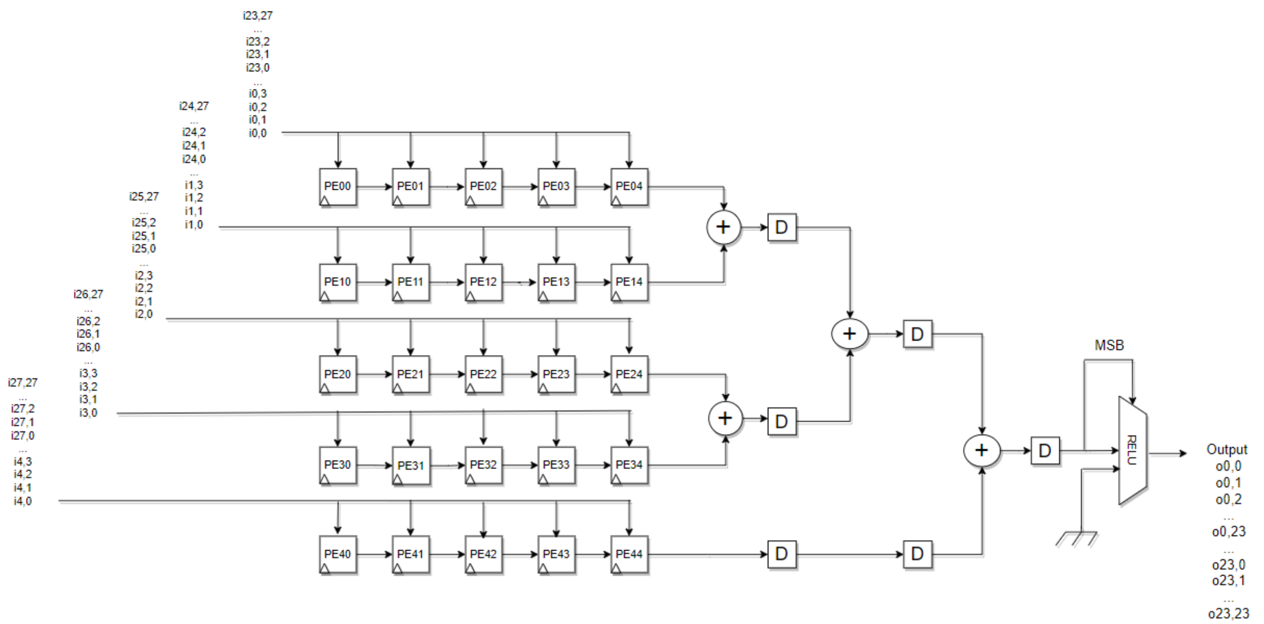


Figure 5.7: DSP block for 5x5 convolution on a 28x28 image

5.6 Pooling Layer

The pooling layer consists of 32 blocks corresponding to each of the 32 DSP blocks of the convolution layer. The main role of the pooling layer is to downsample the 24x24x32 feature map into a 12x12x32 feature map. For this purpose, each pooling block chooses the maximum value of 2x2 window with stride 2. The pooling layer receives entire rows of data (one pixel per clock cycle) and thus plays a critical role in the latency of the overall design. The architecture of the Pooling block is depicted in fig. 5.8. The difference with the design of [12] is that since the DSP blocks output pixels of one row at a time, we are unable to fill the FIFOs with pixels from 2 consecutive rows of the 24x24 map and create subsequent 2x2 windows that feed the max

pooling FSM. To outcome this we use the following scheme. We introduce 2 sets of 2 FIFOs. Each set is controlled by a Demux FSM. When the block receives pixels from row i, counter_1 triggers Demux_FSM_1 to start alternate writing between the first and second FIFO of the 1st set. When the block starts receiving pixels from row i+1, counter_2 triggers Demux_FSM_2 to start alternate writing between the first and second FIFO of the 2nd set. At the same time both FIFOs of 1st set begin to empty, feeding the max pool intra_1 which outputs the maximum between the contents of the two FIFOs. The outputs of max_pool_intra_1 unit must be delayed by 29 clock cycles to meet the proper outputs from max_pool_intra_2 in order to be compared with each other and produce the final result. With this design the pooling block has a latency of 57 clock cycles.

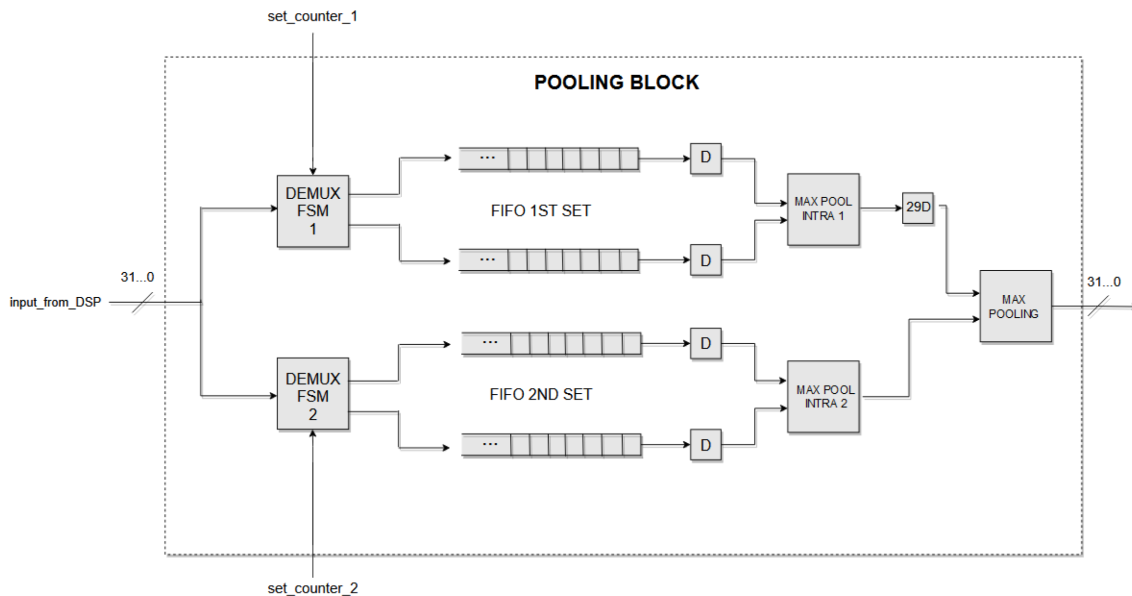


Figure 5.8: Pooling Block Architecture

Figure 5.9 describes the way that the pooling block operates given a specific 24x24 feature map produced by the presceding DSP block. Cyan boxes denote the values stored in FIFOs of the 1st set and green boxes denote the values stored in FIFOs of the 2nd set. Numbers with bolt are the maximum of each comparison. The two Demux_FSMs are transitioning between states IDLE, WRITE_FIFOs and READ_FIFOs with the help of two counters which are synchronizing the Input, Convolution and Pooling layers. These counters are part of the top level design. Finally the creation of the 12x12x32 output created by the whole Pooling Layer is depicted in fig 5.10

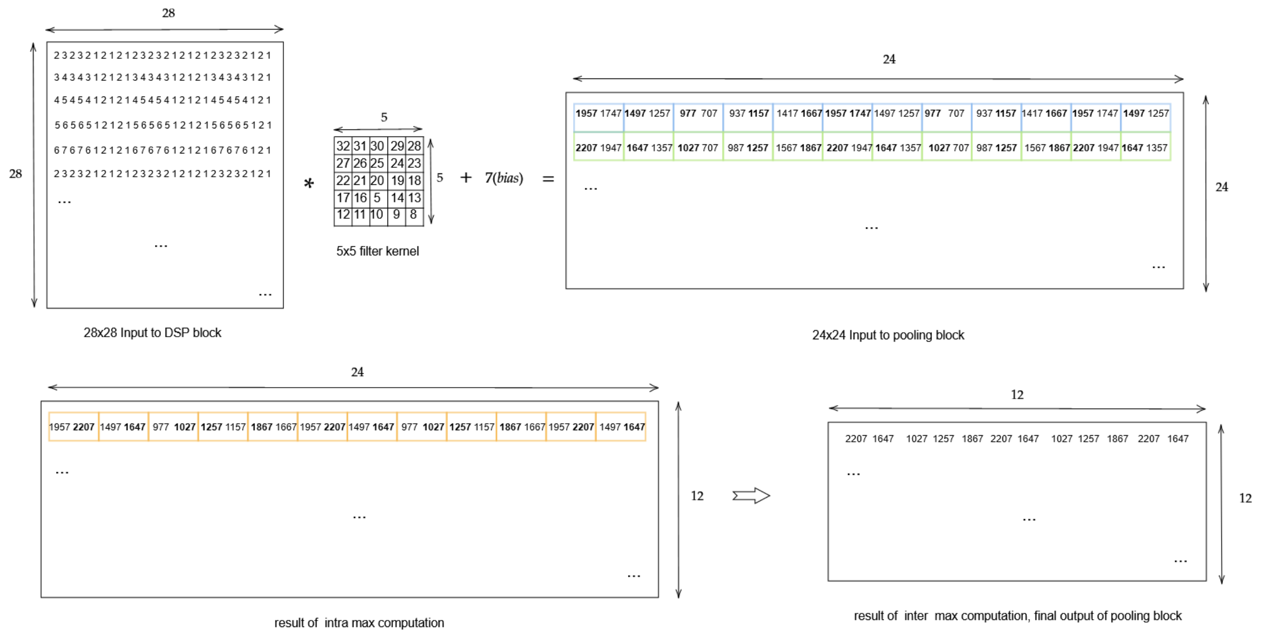


Figure 5.9: Example of DSP block and Pooling block operations for the proposed architectures. The 28x28 input image and the 5x5 kernel show the values that are used in the simulations of chapter 6.

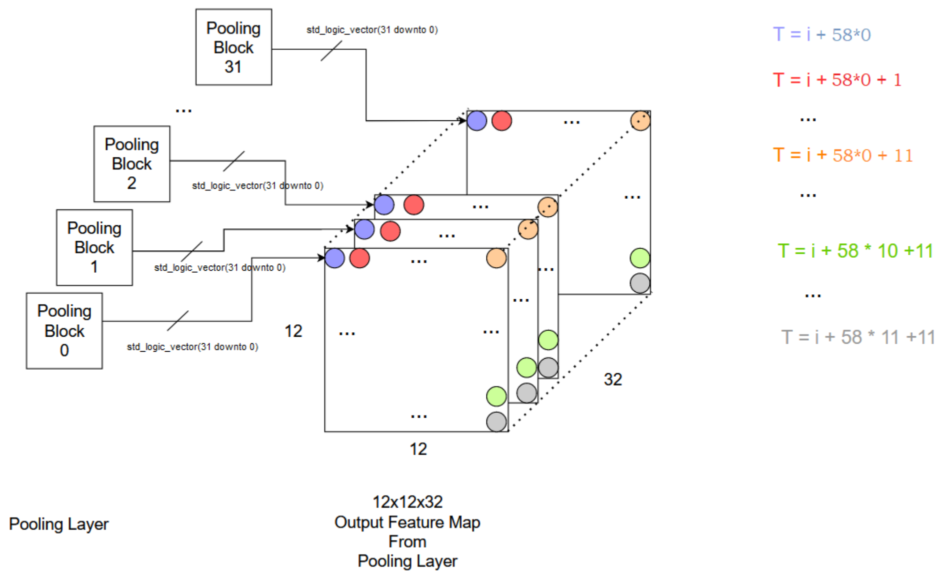


Figure 5.10: All 32 pooling blocks outputs are received at the same time in parallel. Each pooling block outputs the first value of every other row after 58 clock cycles due to its latency. The whole 12x12x32 map is produced in 649 clock cycles.

5.7 Fully Connected and Output Layers

The Fully Connected Layer is comprised of 30 neurons. The main task of this layer is to perform the multiplication of the 4608×1 input vector I^0 and the 30×4608 weight matrix W^0, then add the 30×1 bias vector B^0 and finally perform the ReLU operation to produce the 30×1 vector result O^0. All 30 neuron-multipliers operate in parallel and each one has a local ROM memory where one entire row of the W^0 weight matrix is stored (30 ROM memories in total). The operation of each neuron is pipelined and is depicted in fig. 5.11. When the 32 outputs from the previous pooling layer arrive, every neuron is set to operate with the help of an external (top level) counter and remains set for 11 consecutive clock cycles. Then the neurons are not set anymore and remain idle until the next 32 outputs from the pooling layer arrive and so forth. The neurons perform the dot product

$$\sum_{i=0}^{31} \alpha_i \beta_{i+k\cdot 32} \tag{5.7}$$

where $\alpha = [\alpha_0, \alpha_1, ..., \alpha_{31}]$ is the output of the presceding pooling layer, $\beta = [\beta_0, \beta_1, ..., \beta_{4607}]$ is a row of the W^0 weight matrix and $k = 0, 1, 2, ..., 143$. Each neuron computes the above dot product and accumulates the result with the help of an internal register as shown in fig. 5.11. All neurons start operating when elements of a new row of the 12x12x32 output matrix arrive as input. The neurons stop operating (close) after 11 clock cycles until they re-open again when elements of a subsequent row arrive and so forth. When all neurons complete their computations the B^0 bias vector is added to the 30×1 result vector and finally each entry is passed through the ReLU multiplexer which implements the following non-linearity

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{5.8}$$

The Output Layer is constructed with 10 neurons of the same architecture. It performs the multiplication of the 30×1 I^1 input vector received from the fully connected layer and the 30×10 W^1 weight matrix stored inside ROM memories. Then it adds the 10×1 B^1 bias vector

to the product of the multiplication. The Output Layer starts computing when the I^1 is received. Every one of the ten output layer's results represents the probability that the input image belongs in one of the ten different digits that the classifier is trained to recognize.

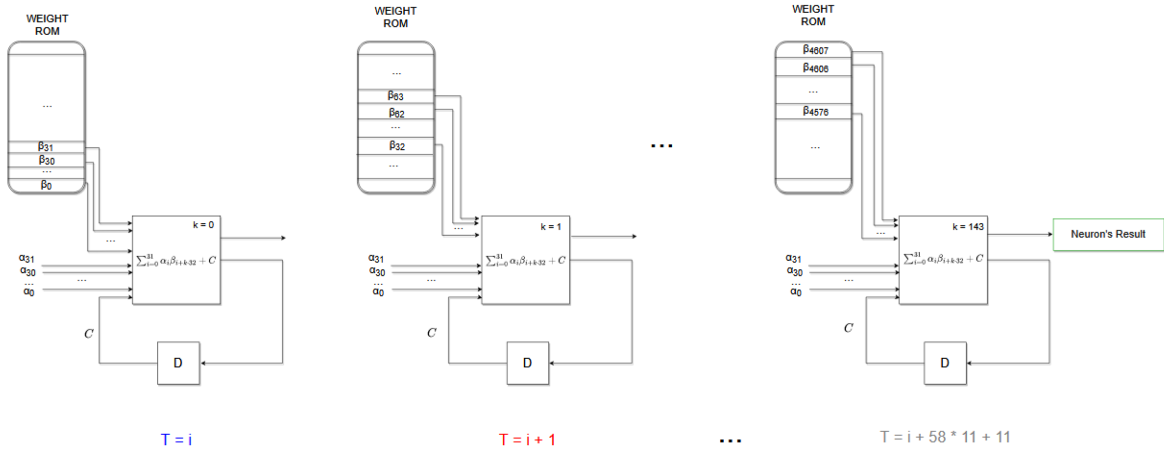


Figure 5.11: Architecture and operation of a fully connected neuron.

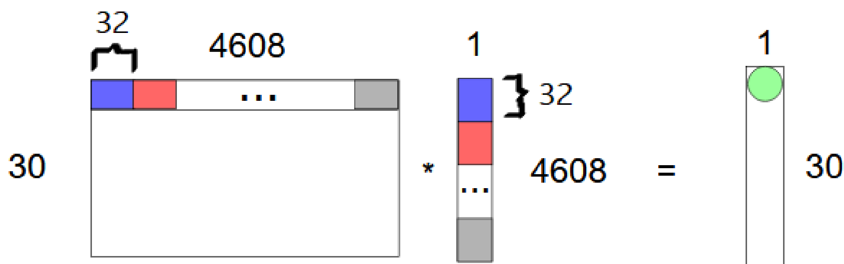


Figure 5.12: Matrix - vector multiplication $O^0 = W^0 \cdot I^0$. Every row of the 30×4608 W^0 weight matrix is stored inside a ROM memory. The vector I^0 arrives as input in chunks of 32 elements at 144 different time instances, so the dot product operation is computed in 144 steps. All 30 neurons compute their dot products in parallel.

6 Tests and Results

The development of the FPGA CNN accelerator was performed in Vivado 2018.3 using VHDL and targeting xc7z030fbv676-1 part of the Zynq-7000 product family. The design was synthesized succesfully and validated with testbenches for every entity and finally for the top level module. Unfortunately the implementation failed because placement could not complete due to overutilization of I/O ports. The design contains 448+1+1+5+10$\times$64=1096 I/O ports whilst the target board has only 380 I/O ports available. To overcome this one could use parallel to serial converters as buffers and serialize the data output hence using less I/O ports.

The functionality of the accelerator was tested using the 28x28 matrix 7.10 as input to the accelerator. For the filter kernel we used the 5x5 matrix 7.11. This matrix is stored inside each DSP block's ROM memory. The bias value that each DSP adds to its result before the ReLU operation is 7_{10} and it is also stored inside every DSP's ROM memory. For convenience in the tests, every DSP block uses the same filter kernel and bias. In a real use-case scenario each one of the 32 filters and biases would have different values in order to support the CNN model which uses 32 different filters to extract different features from the input image. Every one of the 30 blocks (neurons) inside the Fully Connected layer reads a total of 4608 weight values from its weight ROM memory in 144 different (non-consecutive) time steps. In that way each neuron performs the dot product with the $(4608\text{x}1)$ input vector which also arives in 144 different time steps from the Pooling Layer. To keep the computations easy for debugging we have instantiated every weight ROM memory and bias with the same values equal to one (1_{10} = x00000001_{16}). We used the same approach for testing every block (neuron) of the Output Layer. In a real use-case scenario, the weight values and biases of the fully connected and output layers would have different values produced by the back-propagation algorithm performed in a host computer

during the training phase and then the host would load these values into the device. With the above simplifications every block of the Output Layer gives a final result equal to 236135071_{10} at its output. The required processing time for a single input image is 726 clock cycles as we see in the following behavioral simulation.

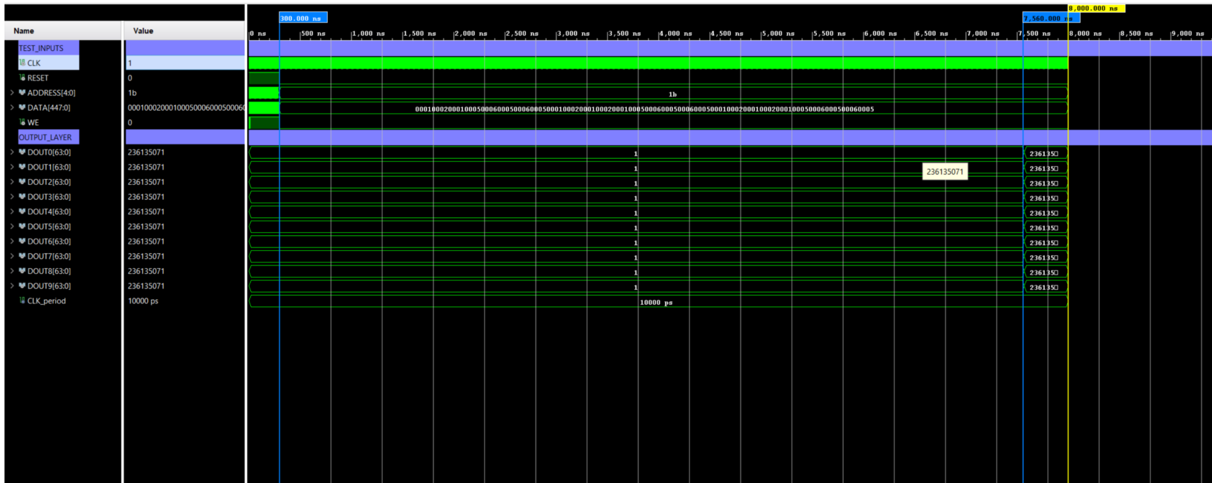


Figure 6.1: Top level behavioral simulation. Clock cycle's duration is 10 ns so the total processing time for one image is 726 clock cycles.

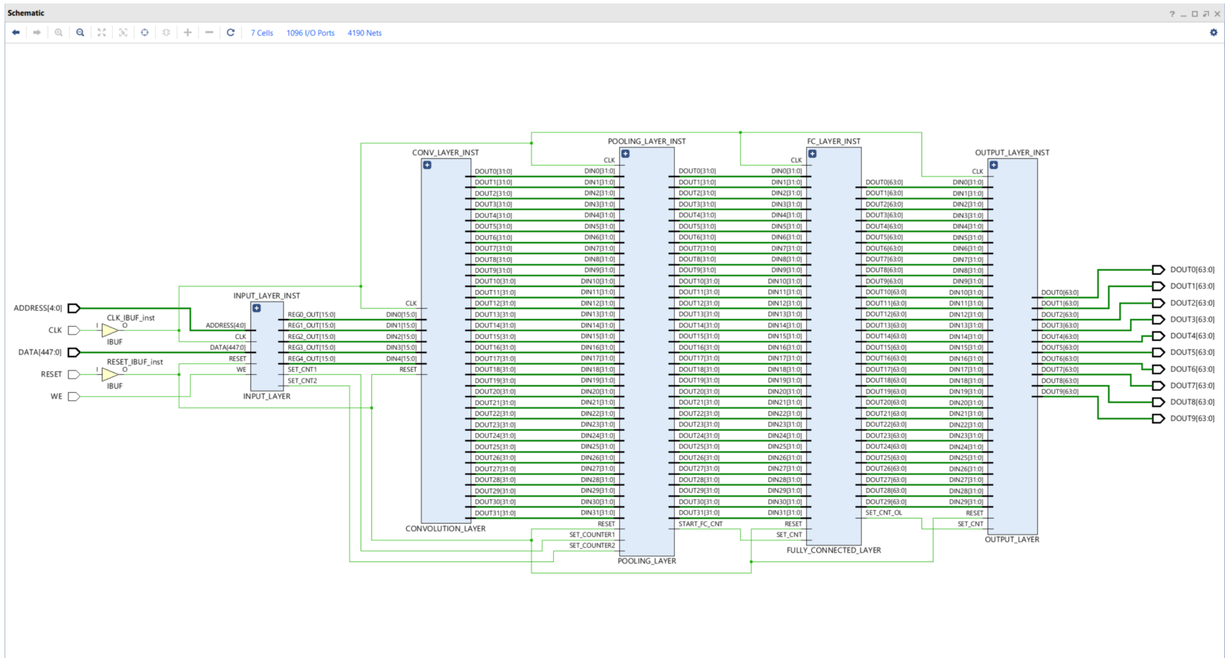


Figure 6.2: Register Transfer Level schematic of the architecture.

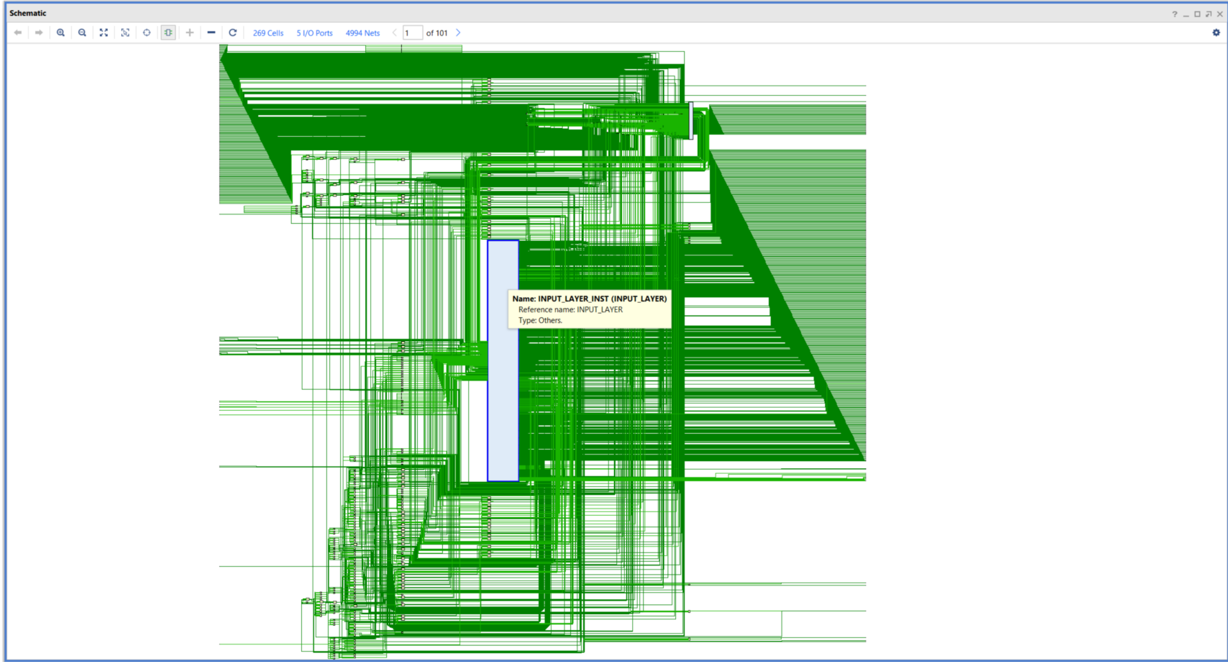


Figure 6.3: Input Layer Synthesis Result.

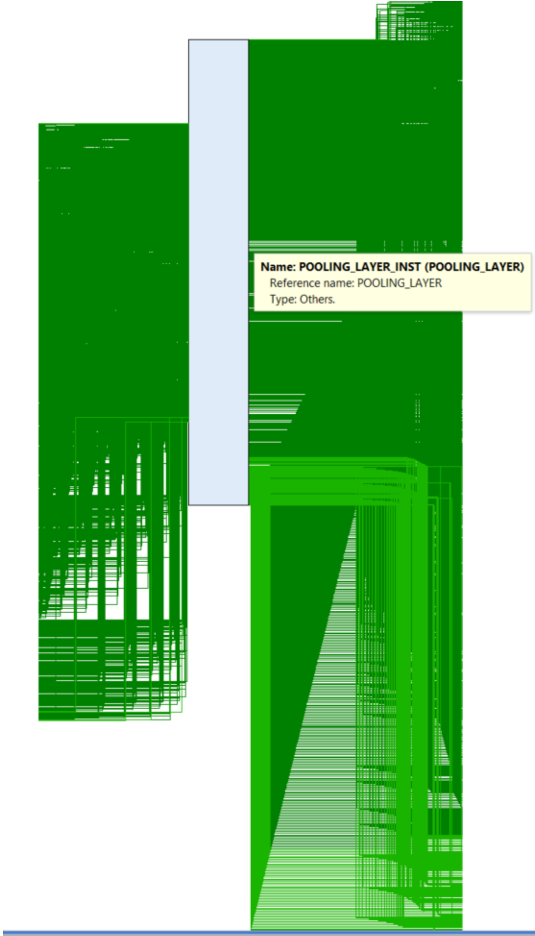


Figure 6.4: Pooling Layer Synthesis Result.

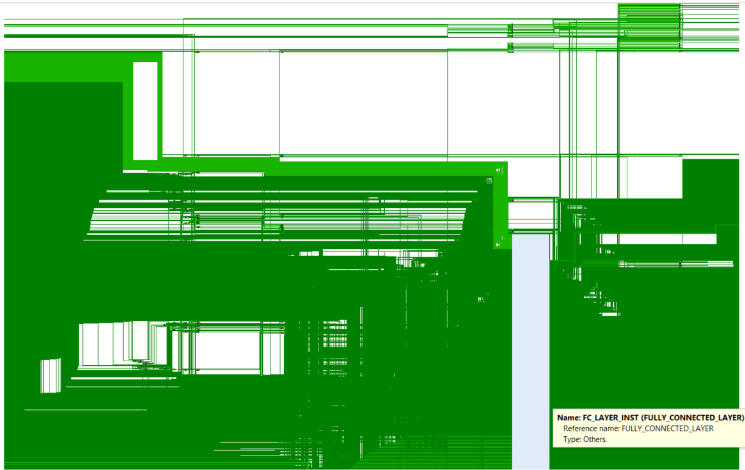


Figure 6.5: FUlly connected Layer Synthesis Result.

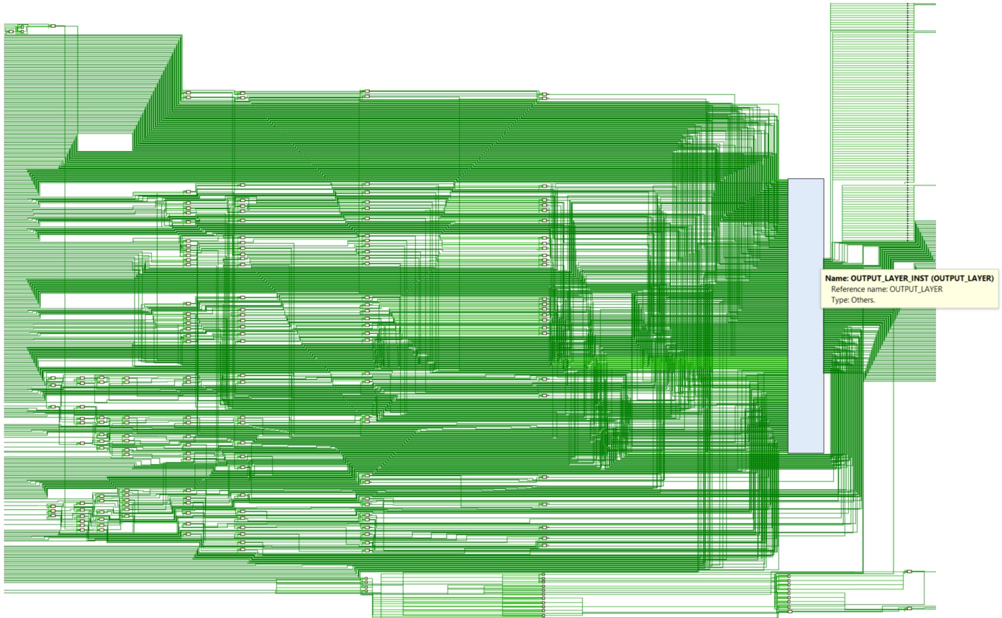


Figure 6.6: Output Layer Synthesis Result.

7 Appendix: Tables

T(time)	PE00	PE01	PE02
1	$w_{00}i_{00}$	$w_{01}i_{00}$	$w_{02}i_{00}$
2	$w_{00}i_{01}$	$w_{00}i_{00} + w_{01}i_{01}$	$w_{01}i_{00} + w_{02}i_{01}$
3	$w_{00}i_{02}$	$w_{00}i_{01} + w_{01}i_{02}$	$w_{00}i_{00} + w_{01}i_{01} + w_{02}i_{02} = []$
4	$w_{00}i_{03}$	$w_{00}i_{02} + w_{01}i_{03}$	$w_{00}i_{01} + w_{01}i_{02} + w_{02}i_{03} = ()$
5	$w_{00}i_{10}$	$w_{00}i_{03} + w_{01}i_{10}$	$w_{00}i_{02} + w_{01}i_{03} + w_{02}i_{10}$
6	$w_{00}i_{11}$	$w_{00}i_{10} + w_{01}i_{11}$	$w_{00}i_{03} + w_{01}i_{10} + w_{02}i_{11}$
7	$w_{00}i_{12}$	$w_{00}i_{11} + w_{01}i_{12}$	$w_{00}i_{10} + w_{01}i_{11} + w_{02}i_{12} =<>$
8	$w_{00}i_{13}$	$w_{00}i_{12} + w_{01}i_{13}$	$w_{00}i_{11} + w_{01}i_{12} + w_{02}i_{13} = //$
9	$w_{00}i_{20}$	$w_{00}i_{13} + w_{01}i_{20}$	$w_{00}i_{12} + w_{01}i_{13} + w_{02}i_{20}$
10	$w_{00}i_{21}$	$w_{00}i_{20} + w_{01}i_{21}$	$w_{00}i_{13} + w_{01}i_{20} + w_{02}i_{21}$
11	$w_{00}i_{22}$	$w_{00}i_{21} + w_{01}i_{22}$	$w_{00}i_{20} + w_{01}i_{21} + w_{02}i_{22}$
12	$w_{00}i_{23}$	$w_{00}i_{22} + w_{01}i_{23}$	$w_{00}i_{21} + w_{01}i_{22} + w_{02}i_{23}$
13	$w_{00}i_{30}$	$w_{00}i_{23} + w_{01}i_{30}$	$w_{00}i_{22} + w_{01}i_{23} + w_{02}i_{30}$
14	$w_{00}i_{31}$	$w_{00}i_{30} + w_{01}i_{31}$	$w_{00}i_{23} + w_{01}i_{30} + w_{02}i_{31}$
15	$w_{00}i_{32}$	$w_{00}i_{31} + w_{01}i_{32}$	$w_{00}i_{30} + w_{01}i_{31} + w_{02}i_{32}$
16	$w_{00}i_{33}$	$w_{00}i_{32} + w_{01}i_{33}$	$w_{00}i_{31} + w_{01}i_{32} + w_{02}i_{33}$

Table 7.1: PE computations using the stationary weight dataflow architecture of fig. 5.3(I).

T(time)	PE10	PE11	P12
1	$w_{10}i_{00}$	$w_{11}i_{00}$	$w_{12}i_{00}$
2	$w_{10}i_{01}$	$w_{10}i_{00} + w_{11}i_{01}$	$w_{11}i_{00} + w_{12}i_{01}$
3	$w_{10}i_{02}$	$w_{10}i_{01} + w_{11}i_{02}$	$w_{10}i_{00} + w_{11}i_{01} + w_{12}i_{02}$
4	$w_{10}i_{03}$	$w_{10}i_{02} + w_{11}i_{03}$	$w_{10}i_{01} + w_{11}i_{02} + w_{12}i_{03}$
5	$[] + w_{10}i_{10}$	$w_{10}i_{03} + w_{11}i_{10}$	$w_{10}i_{02} + w_{11}i_{03} + w_{12}i_{10}$
6	$() + w_{10}i_{11}$	$[] + w_{10}i_{10} + w_{11}i_{11}$	$w_{10}i_{03} + w_{11}i_{10} + w_{12}i_{11}$
7	$w_{10}i_{12}$	$() + w_{10}i_{11} + w_{11}i_{12}$	$[] + w_{10}i_{10} + w_{11}i_{11} + w_{12}i_{12} = []$
8	$w_{10}i_{13}$	$w_{10}i_{12} + w_{11}i_{13}$	$() + w_{10}i_{11} + w_{11}i_{12} + w_{12}i_{13} = ()$
9	$<> +w_{10}i_{20}$	$w_{10}i_{13} + w_{11}i_{20}$	$w_{10}i_{12} + w_{11}i_{13} + w_{12}i_{20}$
10	$// + w_{10}i_{21}$	$<> +w_{10}i_{20} + w_{11}i_{21}$	$w_{10}i_{13} + w_{11}i_{20} + w_{12}i_{21}$
11	$w_{10}i_{22}$	$// + w_{10}i_{21} + w_{11}i_{22}$	$<> +w_{10}i_{20} + w_{11}i_{21} + w_{12}i_{22} =<>$
12	$w_{10}i_{23}$	$w_{10}i_{22} + w_{11}i_{23}$	$// + w_{10}i_{21} + w_{11}i_{22} + w_{12}i_{23} = //$
13	$w_{10}i_{30}$	$w_{10}i_{23} + w_{11}i_{30}$	$w_{10}i_{22} + w_{11}i_{23} + w_{12}i_{30}$
14	$w_{10}i_{31}$	$w_{10}i_{30} + w_{11}i_{31}$	$w_{10}i_{23} + w_{11}i_{30} + w_{12}i_{31}$
15	$w_{10}i_{32}$	$w_{10}i_{31} + w_{11}i_{32}$	$w_{10}i_{30} + w_{11}i_{31} + w_{12}i_{32}$
16	$w_{10}i_{33}$	$w_{10}i_{32} + w_{11}i_{33}$	$w_{10}i_{31} + w_{11}i_{32} + w_{12}i_{33}$

Table 7.2: PE computations using the stationary weight dataflow architecture of fig. 5.3(II).

T(time)	PE20	PE21	P22
1	$w_{20}i_{00}$	$w_{21}i_{00}$	$w_{22}i_{00}$
2	$w_{20}i_{01}$	$w_{20}i_{00} + w_{21}i_{01}$	$w_{21}i_{00} + w_{22}i_{01}$
3	$w_{20}i_{02}$	$w_{20}i_{01} + w_{21}i_{02}$	$w_{20}i_{00} + w_{21}i_{01} + w_{22}i_{02}$
4	$w_{20}i_{03}$	$w_{20}i_{02} + w_{21}i_{03}$	$w_{20}i_{01} + w_{21}i_{02} + w_{22}i_{03}$
5	$w_{20}i_{10}$	$w_{20}i_{03} + w_{21}i_{10}$	$w_{20}i_{02} + w_{21}i_{03} + w_{22}i_{10}$
6	$w_{20}i_{11}$	$w_{20}i_{10} + w_{21}i_{11}$	$w_{20}i_{03} + w_{21}i_{10} + w_{22}i_{11}$
7	$w_{20}i_{12}$	$w_{20}i_{11} + w_{21}i_{12}$	$w_{20}i_{10} + w_{21}i_{11} + w_{22}i_{12} = []$
8	$w_{20}i_{13}$	$w_{20}i_{12} + w_{21}i_{13}$	$w_{20}i_{11} + w_{21}i_{12} + w_{22}i_{13} = ()$
9	$[] + w_{20}i_{20}$	$w_{20}i_{13} + w_{21}i_{20}$	$w_{20}i_{12} + w_{21}i_{13} + w_{22}i_{20}$
10	$() + w_{20}i_{21}$	$[] + w_{20}i_{20} + w_{21}i_{21}$	$w_{20}i_{13} + w_{21}i_{20} + w_{22}i_{21}$
11	$w_{20}i_{22}$	$() + w_{20}i_{21} + w_{21}i_{22}$	$[] + w_{20}i_{20} + w_{21}i_{21} + w_{22}i_{22} = o_{00}$
12	$w_{20}i_{23}$	$w_{20}i_{22} + w_{21}i_{23}$	$() + w_{20}i_{21} + w_{21}i_{22} + w_{22}i_{23} = o_{01}$
13	$<> +w_{20}i_{30}$	$w_{20}i_{23} + w_{21}i_{30}$	$w_{20}i_{22} + w_{21}i_{23} + w_{22}i_{30}$
14	$// + w_{20}i_{31}$	$<> +w_{20}i_{30} + w_{21}i_{31}$	$w_{20}i_{23} + w_{21}i_{30} + w_{22}i_{31}$
15	$w_{20}i_{32}$	$// + w_{20}i_{31} + w_{21}i_{32}$	$<> +w_{20}i_{30} + w_{21}i_{31} + w_{22}i_{32} = o_{10}$
16	$w_{20}i_{33}$	$w_{20}i_{32} + w_{21}i_{33}$	$// + w_{20}i_{31} + w_{21}i_{32} + w_{22}i_{33} = o_{11}$

Table 7.3: PE computations using the stationary weight dataflow architecture of fig. 5.3(III).

T(time)	PE00	PE01	PE02
1	$w_{00}i_{00}$	$w_{01}i_{00}$	$w_{02}i_{00}$
2	$w_{00}i_{01}$	$w_{00}i_{00} + w_{01}i_{01}$	$w_{01}i_{00} + w_{02}i_{01}$
3	$w_{00}i_{02}$	$w_{00}i_{01} + w_{01}i_{02}$	$w_{00}i_{00} + w_{01}i_{01} + w_{02}i_{02} = []$
4	$w_{00}i_{03}$	$w_{00}i_{02} + w_{01}i_{03}$	$w_{00}i_{01} + w_{01}i_{02} + w_{02}i_{03} = ()$
5	$w_{00}i_{10}$	$w_{00}i_{03} + w_{01}i_{10}$	$w_{00}i_{02} + w_{01}i_{03} + w_{02}i_{10}$
6	$w_{00}i_{11}$	$w_{00}i_{10} + w_{01}i_{11}$	$w_{00}i_{03} + w_{01}i_{10} + w_{02}i_{11}$
7	$w_{00}i_{12}$	$w_{00}i_{11} + w_{01}i_{12}$	$w_{00}i_{10} + w_{01}i_{11} + w_{02}i_{12} =<>$
8	$w_{00}i_{13}$	$w_{00}i_{12} + w_{01}i_{13}$	$w_{00}i_{11} + w_{01}i_{12} + w_{02}i_{13} = //$

Table 7.4: PE computations using the unrolled stationary weight dataflow architecture of fig. 5.5(I).

T(time)	PE10	PE11	P12
1	$w_{10}i_{10}$	$w_{11}i_{10}$	$w_{12}i_{10}$
2	$w_{10}i_{11}$	$w_{10}i_{10} + w_{11}i_{11}$	$w_{11}i_{10} + w_{12}i_{11}$
3	$w_{10}i_{12}$	$w_{10}i_{11} + w_{11}i_{12}$	$w_{10}i_{10} + w_{11}i_{11} + w_{12}i_{12} = [[]]$
4	$w_{10}i_{13}$	$w_{10}i_{12} + w_{11}i_{13}$	$w_{10}i_{11} + w_{11}i_{12} + w_{12}i_{13} = (())$
5	$[] + w_{10}i_{20}$	$w_{10}i_{13} + w_{11}i_{20}$	$w_{10}i_{12} + w_{11}i_{13} + w_{12}i_{20}$
6	$() + w_{10}i_{21}$	$[] + w_{10}i_{20} + w_{11}i_{21}$	$w_{10}i_{23} + w_{11}i_{20} + w_{12}i_{21}$
7	$w_{10}i_{22}$	$() + w_{10}i_{21} + w_{11}i_{22}$	$[] + w_{10}i_{20} + w_{11}i_{21} + w_{12}i_{22} =<<>>$
8	$w_{10}i_{23}$	$w_{10}i_{22} + w_{11}i_{23}$	$() + w_{10}i_{21} + w_{11}i_{22} + w_{12}i_{23} = ///$

Table 7.5: PE computations using the unrolled stationary weight dataflow architecture of fig. 5.5(II).

T(time)	PE20	PE21	P22
1	$w_{20}i_{20}$	$w_{21}i_{20}$	$w_{22}i_{20}$
2	$w_{20}i_{21}$	$w_{20}i_{20} + w_{21}i_{21}$	$w_{21}i_{20} + w_{22}i_{21}$
3	$w_{20}i_{22}$	$w_{20}i_{21} + w_{21}i_{22}$	$w_{20}i_{20} + w_{21}i_{21} + w_{22}i_{22} = [[[]]]$
4	$w_{20}i_{23}$	$w_{20}i_{22} + w_{21}i_{23}$	$w_{20}i_{21} + w_{21}i_{22} + w_{22}i_{23} = ((()))$
5	$w_{20}i_{30}$	$w_{20}i_{23} + w_{21}i_{30}$	$w_{20}i_{22} + w_{21}i_{23} + w_{22}i_{30}$
6	$w_{20}i_{31}$	$w_{20}i_{30} + w_{21}i_{31}$	$w_{20}i_{23} + w_{21}i_{30} + w_{22}i_{31}$
7	$w_{20}i_{32}$	$w_{20}i_{31} + w_{21}i_{32}$	$w_{20}i_{30} + w_{21}i_{31} + w_{22}i_{32} =<<<>>>$
8	$w_{20}i_{33}$	$w_{20}i_{32} + w_{21}i_{33}$	$w_{20}i_{31} + w_{21}i_{32} + w_{22}i_{33} = ////$

Table 7.6: PE computations using the unrolled stationary weight dataflow architecture of fig. 5.5(III).

T(time)	Adder1	Adder2	Out
1	-	-	-
2	-	-	-
3	-	-	-
4	[] + [[]] = #	-	-
5	() + (()) = ##	# + [[[]]] = o_{00}	-
6	-	## + ((())) = o_{01}	o_{00}
7	-	-	o_{01}
8	<> + <<>>= ###	-	-
9	// + /// = ####	###+ <<<>>>= o_{10}	-
10	-	#### + //// = o_{11}	o_{10}
11	-	-	o_{11}

Table 7.7: PE computations using the unrolled stationary weight dataflow architecture of fig. 5.5(IV).

SREG0	SREG1	SREG2	SREG3	SREG4
row 0	row 1	row 2	row 3	row 4
row 2	row 3	row 4	row 5	row 6
row 4	row 5	row 6	row 7	row 8
row 6	row 7	row 8	row 9	row 10
row 8	row 9	row 10	row 11	row 12
row 10	row 11	row 12	row 13	row 14
row 12	row 13	row 14	row 15	row 16
row 14	row 15	row 16	row 17	row 18
row 16	row 17	row 18	row 19	row 20
row 18	row 19	row 20	row 21	row 22
row 20	row 21	row 22	row 23	row 24
row 22	row 23	row 24	row 25	row 26

Table 7.8: The order in which input image rows are loaded to the registers of the 1st set (Input Layer).

SREG0	SREG1	SREG2	SREG3	SREG4
row 1	row 2	row 3	row 4	row 5
row 3	row 4	row 5	row 6	row 7
row 5	row 6	row 7	row 8	row 9
row 7	row 8	row 9	row 10	row 11
row 9	row 10	row 11	row 12	row 13
row 11	row 12	row 13	row 14	row 15
row 13	row 14	row 15	row 16	row 17
row 15	row 16	row 17	row 18	row 19
row 17	row 18	row 19	row 20	row 21
row 19	row 20	row 21	row 22	row 23
row 21	row 22	row 23	row 24	row 25
row 23	row 24	row 25	row 26	row 27

Table 7.9: The order in which input image rows are loaded to the registers of the 2nd set (Input Layer).

Table 7.10: 28x28 pixels input image used for testbench simulation.

$$
\begin{vmatrix}
2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 \\
3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 2 & 1 & 2 & 1 \\
4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 \\
5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 \\
6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 \\
2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 \\
3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 2 & 1 & 2 & 1 \\
4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 \\
5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 \\
6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 \\
2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 \\
3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 2 & 1 & 2 & 1 \\
4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 \\
5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 \\
6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 \\
2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 \\
3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 2 & 1 & 2 & 1 \\
4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 \\
5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 \\
6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 \\
2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 \\
3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 2 & 1 & 2 & 1 \\
4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 \\
5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 & 2 & 1 & 5 & 6 & 5 & 6 & 5 & 1 & 2 & 1 \\
6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 & 2 & 1 & 6 & 7 & 6 & 7 & 6 & 1 & 2 & 1 \\
2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 1 & 2 & 1 \\
3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 3 & 1 & 2 & 1 & 2 & 1 & 3 & 4 & 3 & 4 & 2 & 1 & 2 & 1 \\
4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1 & 2 & 1 & 4 & 5 & 4 & 5 & 4 & 1 & 2 & 1
\end{vmatrix}
$$

Table 7.11: 5x5 filter kernel used for all DSP blocks during testbench simulation.

$$
\begin{bmatrix}
32 & 31 & 30 & 29 & 28 \\
27 & 26 & 25 & 24 & 23 \\
22 & 21 & 20 & 19 & 18 \\
17 & 16 & 15 & 14 & 13 \\
12 & 11 & 10 & 9 & 8
\end{bmatrix}
$$

Resource	Utilization	Utilization %
LUT	51906	66.04
LUTRAM	1024	3.85
FF	19647	12.5
BRAM	7	2.45
DSP	15	3.75
BUFG	12	37.5

Table 7.12: Resource Utilization

References

[1] Kamel Abdelouahab et al. *Accelerating CNN inference on FPGAs: A Survey*. 2018. DOI: 10.48550/ARXIV.1806.01683. URL: https://arxiv.org/abs/1806.01683.

[2] Y. Bengio and Yann Lecun. "Convolutional Networks for Images, Speech, and Time-Series". In: (Nov. 1997).

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. 1st ed. Springer, 2007. ISBN: 0387310738. URL: http://www.amazon.com/Pattern-Recognition-Learning-Information-Statistics/dp/0387310738%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0387310738.

[4] Calin Cascaval et al. "A taxonomy of accelerator architectures and their programming models". In: *IBM J. Res. Dev.* 54 (2010), p. 5.

[5] Robert van de Geijn and Kazushige Goto. "Anatomy of high-performance matrix multiplication Kazushige Goto, Robert A. van de Geijn ACM Transactions on Mathematical Software (TOMS), 2008". In: *ACM Transactions on Mathematical Software* 34 (May 2008), Article 12. DOI: 10.1145/1356052.1356053.

[6] Vinayak Gokhale et al. "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2014, pp. 696–701. DOI: 10.1109/CVPRW.2014.106.

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

[8] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[9] Weiwen Jiang et al. *Hardware/Software Co-Exploration of Neural Architectures*. 2019. DOI: 10.48550/ARXIV.1907.04650. URL: https://arxiv.org/abs/1907.04650.

[10] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. 2017. DOI: 10.48550/ARXIV.1704.04760. URL: https://arxiv.org/abs/1704.04760.

[11] Kung. "Why systolic architectures?" In: *Computer* 15.1 (1982), pp. 37–46. DOI: 10.1109/MC.1982.1653825.

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

[13] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539.

[14] F. THOMSON LEIGHTON. "CHAPTER 1 - ARRAYS AND TREES". In: *Introduction to Parallel Algorithms and Architectures*. Ed. by F. THOMSON LEIGHTON. Morgan Kaufmann, 1992, pp. 1–276. ISBN: 978-1-4832-0772-8. DOI: https://doi.org/10.1016/B978-1-4832-0772-8.50005-4. URL: https://www.sciencedirect.com/science/article/pii/B9781483207728500054.

[15] Richard J. Lipton and Robert Sedgewick. "Lower bounds for VLSI". In: *STOC '81*. 1981.

[16] Raju Machupalli, Masum Hossain, and Mrinal Mandal. "Review of ASIC accelerators for deep neural network". In: *Microprocessors and Microsystems* 89 (2022), p. 104441. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2022.104441. URL: https://www.sciencedirect.com/science/article/pii/S0141933122000163.

[17] Bruno Olshausen and David Field. "Emergence of simple-cell receptive field properties by learning a sparse code for natural images". In: *Nature* 381 (July 1996), pp. 607–9. DOI: 10.1038/381607a0.

[18] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011. ISBN: 9780123742605 0123742609.

[19] K.K. Parhi. *VLSI DIGITAL SIGNAL PROCESSING SYSTEMS: DESIGN AND IMPLEMENTATION*. Wiley India Pvt. Limited, 2007. ISBN: 9788126510986. URL: https://books.google.gr/books?id=APFRHFkMqG8C.

[20] Murad Qasaimeh et al. "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels". In: May 2019. DOI: 10.1109/ICESS.2019.8782524.

[21] D. Reisis and N. Vlassopoulos. "Address Generation Techniques for Conflict Free Parallel Memory Accessing in FFT Architectures". In: *2006 13th IEEE International Conference on Electronics, Circuits and Systems*. 2006, pp. 1188–1191. DOI: 10.1109/ICECS.2006.379653.

[22] Vivienne Sze et al. "Designing DNN Accelerators". In: *Efficient Processing of Deep Neural Networks*. Cham: Springer International Publishing, 2020, pp. 73–118. ISBN: 978-3-031-01766-7. DOI: 10.1007/978-3-031-01766-7_5. URL: https://doi.org/10.1007/978-3-031-01766-7_5.

[23] Vivienne Sze et al. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. 2017. DOI: 10.48550/ARXIV.1703.09039. URL: https://arxiv.org/abs/1703.09039.

[24] Sergios Theodoridis. *Machine Learning: A Bayesian and Optimization Perspective*. 1st. USA: Academic Press, Inc., 2015. ISBN: 0128015225.

[25] Yun Xu and Royston Goodacre. "On Splitting Training and Validation Set: A Comparative Study of Cross-Validation, Bootstrap and Systematic Sampling for Estimating the Generalization Performance of Supervised Learning". In: *Journal of Analysis and Testing* 2 (Oct. 2018). DOI: 10.1007/s41664-018-0068-2.

[26] Min Zhang et al. "Optimized Compression for Implementing Convolutional Neural Networks on FPGA". In: *Electronics* 8.3 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8030295. URL: https://www.mdpi.com/2079-9292/8/3/295.