Semester Project Report

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Real-Time Image Classification using Convolutional Neural Networks on Embedded Platforms using SoC / FPGA Convolutional Layer Accelerator

Student:	Patryk Oleniuk
Laboratory:	Processor Architecture Lab (**LAP**)
Supervisors:	Mikhail Asiatici, Rene Beauchat
Tools to train the CNN:	Python, Tensorflow, Jupyter Notebook
Algorithm Analysis:	Python, C/C++
Embedded System:	Xilinx Zynq ZC-7006 on PetaLinux / BareMetal
Tools for Embedded System:	Sigasi, Vivado 2017, Xilinx C/C++ SDK 2017

Abstract:
Cameras are used everywhere as flexible sensors for numerous applications. For mobility and privacy reasons, the required image processing should be local on embedded computer platforms with performance requirements and energy constraints. Dedicated acceleration of Convolutional Neural Networks (CNN) can achieve these targets with enough flexibility to perform multiple vision tasks. A challenging problem is the design of efficient accelerators for that purpose, since some CNNs require a lot of repetitive image processing. In this project, I show and implement solutions to quantize the trained CNN as well as perform and optimize computations on an Embedded System. The design flow is evaluated by implementing the previously trained CNN to recognize facial emotions from face image implemented in python on a PC. The project explains the process of porting the CNN algorithm from python to C/C++ and then executing it on a ZYNQ FPGA board running with PetaLinux. The bottleneck of the CNN is the convolutional layers and that is why different solutions for that accelerator are analysed and some of them implemented on VHDL, connected to the embedded processor and their performance is tested.

Contents

List of figures

List of tables

List of Sample Codes

List of Terms and Abbreviations:

CNN	– Convolutional Neural Network.
Tensorflow	– python toolbox for Machine Learning.
Convolutional Kernel	– feature map, image or images representing a pattern to detect.
FPGA	– Field Programmable Gate Array
SoC	– System on Chip
FFs	– flop flops
LUTs	– Look Up Tables
BRAMs	– internal Block RAM (inside the FPGA)
LSb	– Least Significant bit
MSb	– Most significant bit
ZynQ	– ARM microcontroller used in ZC-06 board
GPIO	– General Purpose Input Output
GPO	– General Purpose Output
GPI	– General Purpose Input
L1- L6	– Indicators of Convolutional Layer Numbers of the CNN

1. Introduction

Cameras are used everywhere as flexible and cheap sensors for numerous applications. For mobility and privacy reasons, the required image processing should be local on embedded computer platforms with performance requirements and energy constraints that would allow to perform certain tasks, e.g. image classification. [1]
Currently the image classification tasks are performed very often by data structures called Convolutional Neural Networks (CNNs). The CNNs show very promising classification accuracy especially for image classification, but also for many other fields. The CNN properties and obtained accuracy are based not only of the chosen internal architecture, but solely on the training data and its amount, i.e. it requires a lot of trained, labelled data to train the model properly. The CNN used in this example was trained before and is designed to classify 6 emotions (happy, sad, surprised, angry, normal, scared) from 48x48 greyscale image containing frontal face. Executing single classification for this CNN on a standard PC on python takes few seconds. It was designed like any other typical CNN hence all the methodology and codes below can be used directly or with minor modifications to port and accelerate any other CNN to classify different objects. The CNN training, or the training acceleration is NOT the topic of this document.
The project shows the analysis of the computations behind a typical CNN classification (forward propagation) and the approach to port it into an Embedded Platform, so that the classification algorithm can be executed there. The process includes:

1. Migration of the trained model from python / tensorflow (the CNN is usually trained in some high-level language like python or Java) to C/C++.
2. Profiling of the C/C++ code to identify bottlenecks of the classification algorithm.
3. Quantisation of the coefficient to decrease computational power but keep good classification precision.
4. Case studies on 2 sample CNN accelerator designs using FPGA present in the Embedded Platform.
5. Implementation of the accelerator on the Embedded System.
6. Acceleration results.

A challenging problem is the design of efficient accelerators of CNN forward propagation, i.e. the “new image” classification task based on the previously trained CNN. The document describes a dedicated accelerator of the CNN that can achieve fast timing targets with enough flexibility to perform multiple vision tasks simultaneously, since some CNNs require a lot of repetitive image processing. In the project, I show and implement efficient solutions to quantize the trained CNN as well as perform and optimize computations on an Embedded System. The design flow is evaluated by implementing the CNN and executing it on a ZYNQ FPGA board running with PetaLinux. The bottleneck of the CNN are the convolutional layers and that is why different solutions for that accelerator are analysed and some of them implemented on VHDL, connected to the embedded processor and the performance of the best accelerator solution is tested.

2. The Trained CNN Model

The purpose of the CNN used in this example is to classify facial images into 6 emotion categories. It uses a lot of images’ convolutions to find specific features in the images as well as non-linear operations to perform the classification task based on the detected features.

The input of the CNN is 48x48 greyscale image (array of pixels), each pixel traditionally represented by 8 bits (0-255). The output of the CNN is the array with floating-point classification probabilities. The overview of the system is shown below.

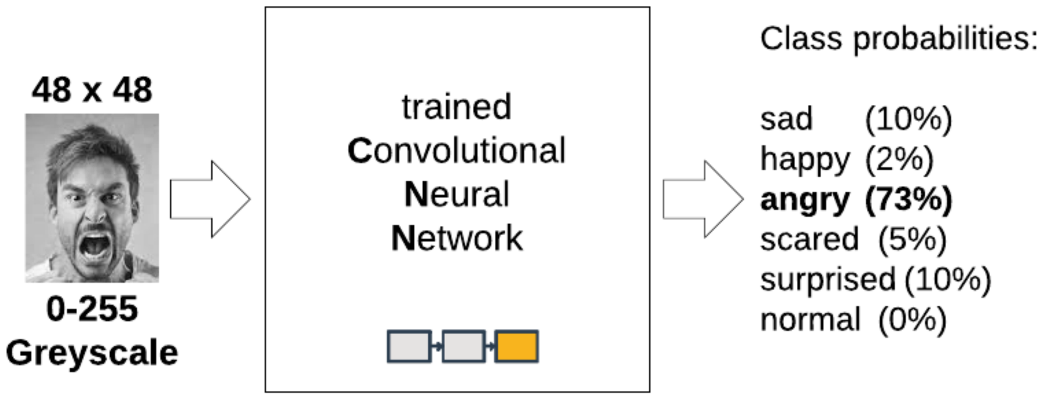


Figure 1. General view of CNN classification.

The CNN computational model contains multiple layers of very similar operations. There are multiple convolutional layers, which detect multiple features by performing the 2D image convolution, and fully connected layer at the end which maps the image pixels to the classified probability by simple matrix-matrix multiplication. The important thing to notice is that during the training period, each feature map and fully-connected matrix changes in order to minimise the error with the training data. The training of the CNN was described in [ADD REFERENCE] and this document is <u>not</u> to describe and/or accelerate the training of the CNN.

2.1 Computational Graph – Details

The CNN introduces 7 similar stages of image processing. Each of the stage is called a layer. In the CNN image processing domain, we could distinguish 2 main types of layers:

- **Convolutional** – the input image is convoluted in 2D with a trained feature image. It is usually much smaller than the image, e.g. input – 48x48 face image, feature map: 8x8 eye → this feature map will detect the position of the certain type of eye in the image). Each layer can have certain number of features (e.g. 22 8x8 feature maps). There can be also more input images (e.g. 22 images from the previous layer). <u>Then usually for each resulting image there is a "transfer" feature map so that each of the resulting images has a feature map that related to any of the input image.</u> As An example, for 22 input images and 22 output images, there are 22*22=484 feature maps. The resulting images for each of the input images are summed together to create the resulting image.
- **Fully connected Layers** – the input image is transformed into a matrix and multiplied to obtain another form of the data, e.g. transforming 22 6x6 images into 6x classification probabilities in the last layer of the aforementioned CNN).

To obtain better classification and training results, different image operations can be performed in between layers, i.e.:

- **ReLU** – Nonlinear Operator which "flattens" the negative part of the image (if the image is defined as signed). The function is applied to all the pixels in the image. The Function of the ReLU is shown below.

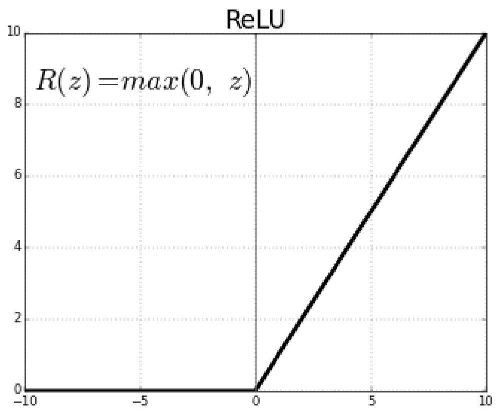


Figure 2. ReLU rectifier function.

- **Max Pooling** – It decreases the size of the image by choosing the pixels with the biggest value from the pool size. As an example, 48x48 image with max_pooling2x2 will result in 24x24 image with the biggest pixels in each 2x2 pool. The Max Pooling methodology is shown in the figure below.

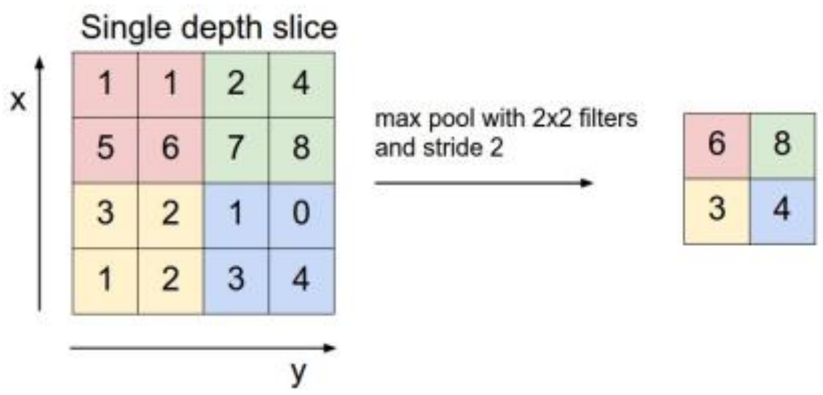


Figure 3. Max Pooling 2x2 explanation.

In the used CNN there are 6 convolutional layers using 8x8 or 4x4 feature maps, each followed by a ReLU and sometimes by max-pooling. Each layer recognises 22 feature maps. The last 7th layer is a fully connected layer which transforms 22 6x6 images into the classification constants. The detailed overview of the algorithm is shown below.

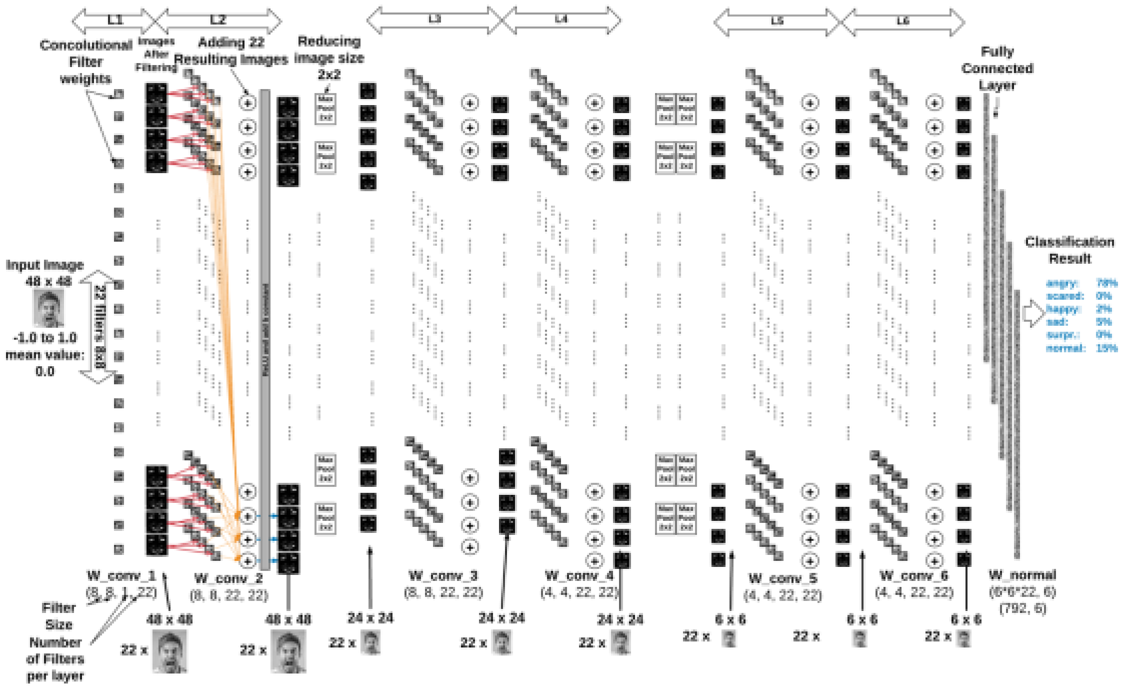


Figure 4. Simplified Computational Structure of the CNN.

Comments:

- Note the names and sizes of the Feature Maps (**W_Conv_1 (8,8,1,22)**). The notation is consistent with the python/tensorflow implementation.

The summary of the 6 convolutional Layers parameters is shown on a table below:

Table 1. Summary of each layers' properties.

	L1	L2	L3	L4	L5	L6	L7
n 2D conv to compute	22	484	484	484	484	484	6x6x22 →6 matrix multiplication
kernel length	8	8	8	4	4	4	
n img before layer (in)	1	22	22	22	22	22	
n img after layer(out)	22	22	22	22	22	22	
img length	48	48	24	24	6	6	

About the features and training:

To train all the feature maps automatically, a labelled dataset of more than 35 000 images set was used. One can observe, that some of the features have a real physical meaning and could represent part of a human face. An example of trained features and the images after applying them is shown below.

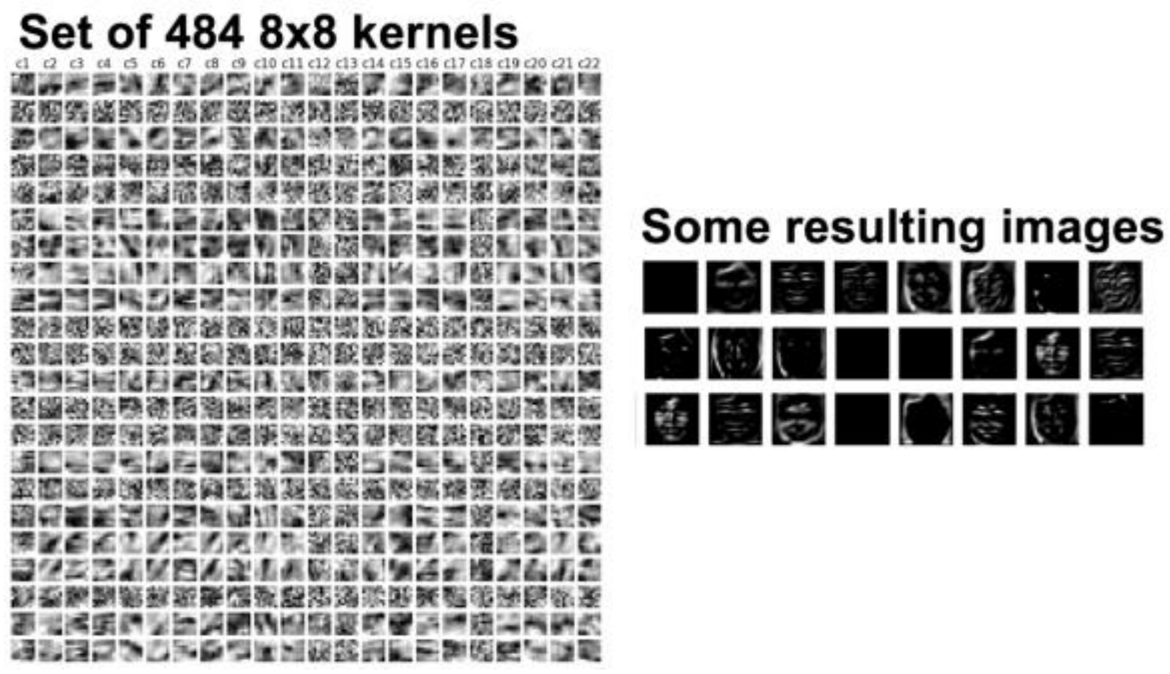

Set of 484 8x8 kernels
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18 c19 c20 c21 c22
Some resulting images

Figure 5. Example of feature maps and resulting images.

Both feature maps and the resulting images can be positive or negative, floating point. After performing a set of convolutions (22*22 in that case) on the corresponding input images (22 in that case, by row) with different kernels, columns of the resulting images(22 in that case) are added (pixel by pixel) to obtain again the number of resulting images after the layer (22 in that case). After the convolution, some the constant values are added to the (22 in that case) images and ReLU operation is performed on each pixel.

The code below is showing a single layer operation based on second layer of the CNN.

```
## convolved2   = np.zeros((22,48,48)) # 22- number of inter-layer imgs, image 48x48
## convolved2_p = np.zeros((22,24,24)) # 22- number of inter-layer imgs, image 24x24
(maxpool2x2)

## LAYER 2 ##
for i in range(22):
    for j in range(22):  #accumulating convolutions for each row
        #single convolution operation (2 for loops inside)
        convolved2[i,:,:] = convolved2[i,:,:] + conv2d_single(convolved1[j,:,:], Wc2[:,:,j,i])
    convolved2_p[i,:,:] = maxpool2x2(convolved2[i,:,:])    #maxpooling 2x2
    convolved2_p[i,:,:] = ReLU( convolved2_p[i,:,:] + b_c2[0][i] )    #ReLU

## LAYER 3 operates on convolved2_p
```

Code 1. Pseudo-code showing operations in layer 2.

To fully understand the algorithm performed by a tensorflow library, a python and then C/C++ implementation has been written. To allow the reader to understand the algorithm structure and operations performed, both codes are attached to the document.

Important notices: Each layer contains a series of for-loops which then will be referred to in the accelerator section. The codes written in C++ and python are single threaded, therefore cannot use any other processor core, even if present.

2.2 Profiling on a PC and Embedded Platform

The python and C/C++ implementations of the algorithms were used to identify the classification time for each of them and understand the acceleration needs, if any. The algorithm execution times, averaged by 10x execution, are shown below.

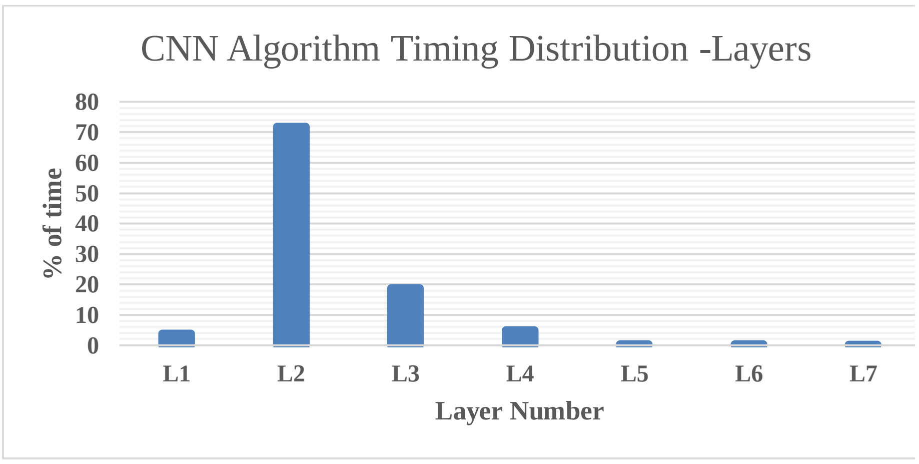
Table 2. Timing results per single 48x48 face image

1. Macbook PC (3.4GHz Intel) in Python	–python / tensorflow	3.55 s
2. Macbook PC (3.4GHz Intel) in Python	–for loops (single thr)–	126 s
3. Macbook PC (3.4GHz Intel) in C/C++	–for loops (single thr)–	0.6 s
4. Zynq PetaLinux(900MHz) in C/C++	–for loops (single thr)–	11.5 s

Observations:

- The execution in python/tensorflow on a PC is very fast as for the high-level language, since the library is well optimized. Hence, even the python code (1.) is high-level, it's very close to the C++ (3.).
- Execution 3. Is the fastest, because it's executed on the fastest machine and the code is most low-level (C++).
- We could observe how much longer is the execution on the Embedded platform. For most of the processing tasks, the classification time of more than few seconds is unacceptable. Especially for real-time emotion recognition from the camera, which the CNN was designed to, is unacceptable.

To identify the bottlenecks of the algorithm, the profiling was executed on the C-code. The results for each layer are shown below.



L1: 3.8 %

L2: 71.8 %

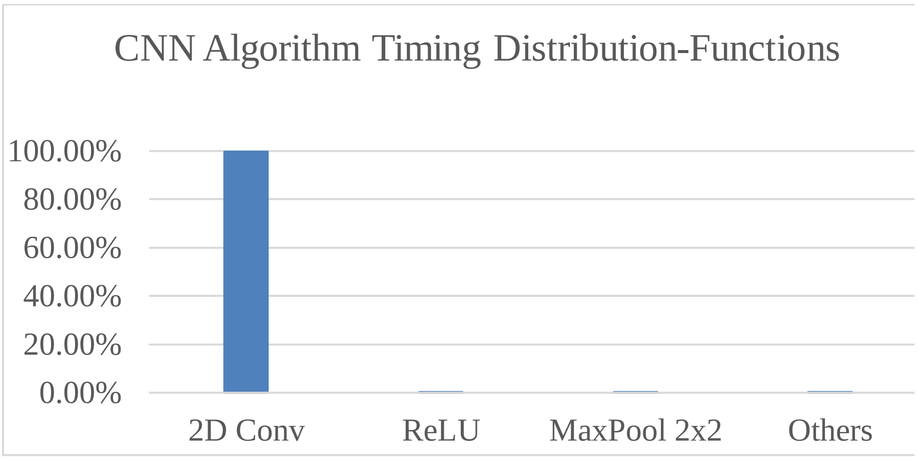
L3: 18.8 %

L4: 4.8 %

L5: 0.3 %

L6: 0.2 %

L7: 0.02 %



2D Conv:	99.6%
ReLU:	0.2 %
MaxPool:	0.15%
Others:	0.05%

Figure 6. CNN Algorithm Timing Results.

Observations:

- The most computationally expensive is Layer 2, since it has 22 48x48 input images (it has the highest parameters from each of the layers).
- By analyzing the detailed profiling info, the main computational task could be specified – the multiple 2D convolutions.
- The ReLU or max-pooling did not show to take more than 0.5% of the whole processing time.

Conclusions:

- Acceleration of the following functions should visibly shorten the classification time:
  - 2D image convolution, because it's the most computationally heavy task executed with the most of time.
  - ReLU, because it's very easy to implement it in the FPGA as a multiplexer (if x<0 assign 0, else assign x).
- Due to the acceleration, there will be some delays to exchange information between the processor and the accelerator. Since we need to share a lot of data, it'd be valuable to design a solution that would also minimise this communication and therefore minimise the accelerator←→processor time.
- Another effort of the data transfer between the float ←→ fixed point, which would need to be executed every transfer to/from the accelerator.

2.3 Quantization

The usual requirement for signal processing using FPGA is quantisation, since they cannot easily handle the floating-point operations. For the purpose of the CNN acceleration, all of the operations to be accelerated have to be performed in the fixed-point domain. To clarify the needs for the CNN, the summary of the filter coefficients, as well as some example middle-results (the images in between the layers) content are shown below.

```
The filter coefficients (constant):
Max Wc1:  0.581492   Avg Wc1:   0.00039835 Min Wc1:  -0.942267
Max Wc2:  0.919815   Avg Wc1:  -0.0357445  Min Wc2:  -1.39372
Max Wc3:  0.817923   Avg Wc2:  -0.014101   Min Wc3:  -1.02589
Max Wc4:  0.604326   Avg Wc3:  -0.0391957  Min Wc4:  -0.990101
Max Wc5:  0.626381   Avg Wc4:  -0.0127036  Min Wc5:  -1.15402
Max Wc6:  0.624091   Avg Wc5:  -0.0218916  Min Wc6:  -0.839514
Max Wn6:  0.68207    Avg Wn6:  -0.0214262  Min Wn6:  -1.14417

The adder coefficients b (constant):
Max bc1:  0.0133187  Avg bc1:  -0.0378035  Min bc1:  -0.13341
Max bc2:  0.223921   Avg bc2:  0.0253297   Min bc2:  -0.243458
Max bc3:  0.307745   Avg bc3:  0.0571535   Min bc3:  -0.153931
Max bc4:  0.345763   Avg bc4:  -0.000638   Min bc4:  -0.35225
Max bc5:  0.297876   Avg bc5:  0.0265728   Min bc5:  -0.291342
Max bc6:  0.0384562  Avg bc6:  0.00243238  Min bc6:  -0.0225041

The middle results in between the layers h
(calc for 64 test images) :
Max h1:  0.424823    Avg h1:  0.0094548    Min h1:  0.0
Max h2:  2.96478     Avg h2:  0.0251095    Min h2:  0.0
Max h3:  7.43573     Avg h3:  0.267952     Min h3:  0.0
Max h4:  18.9776     Avg h4:  0.128376     Min h4:  0.0
Max h5:  23.1608     Avg h5:  0.653597     Min h5:  0.0
Max h6:  15.8007     Avg h6:  0.350184     Min h6:  0.0
```

Table 3. Min, avg and max values of the coefficients and example inter-layer results.

The aforementioned results were generated using python notebook and are based on the tensorflow CNN model. For each execution, the filter and b coefficients are constant and have positive or negative values. The inter-layer results are different for each image. To take a reasonable example to see what is the maximum and average value of the images generated after each layer, the values were calculated from 64 example images. The minimal value of the inter-layer images are 0, because of the ReLU function which zeros the negative part of the image after the resulting image is calculated.
The maximum value of the coefficients are usually very close to 1 and none of the values are bigger than 2.0. However, we can see that, especially for the layers 3-15, the maximum values are quite high and exceed 2.0 value. That is why, the fixed point value used in the computation should include some more bits for the high integer values. Another consideration is using different fixed point scheme for filter coefficients and the result calculation (which is considered to be implemented in the future). For simplicity, the same fixed-point scheme for all operations will be applied.
To understand how the quantisation will affect the classification results, the following experiment in C++ on float coefficients was performed:

1. Prepare the set of 64 test images.
2. Classify each of the image and obtain the classification probabilities for each class.
3. Quantise the coefficients with the certain, signed fixed-point scheme.
4. Perform the CNN algorithm on the same 64 test images, with the quantised coefficients from 3. and “flatten” each of the resulting imaged after each layer, according to maximum value that the signed point can store.
5. Compare the 64*6 class probabilities with results from 2. and calculate the mean and max error.*

* Note, that all the probability values was taken into consideration when calculating the error. As an example, if the original classification probabilities were [0.1 0.1 0.5 0.2 0.0 0.1] and the result after quantisation was [0.0 0.0 0.55 0.45 0.0 0.0], the mean error is 0.108 (0.65/6) and the maximum error os 0.25 (0.45-0.2).

The quantisation sweep was performed for the following fixed-point schemes:

- 2.5 to 2.10 - max values: (-2,2)
- 3.5 to 3.10 - max values: (-4,4)
- 4.5 to 4.10 - max values: (-8,8)
- 5.5 to 5.10 - max values: (-16,16)

The fixed point fractional values from .5 to .10 represent steps from 0.03125 to 0.00097. The result of the experiment is shown on the figure and table below.

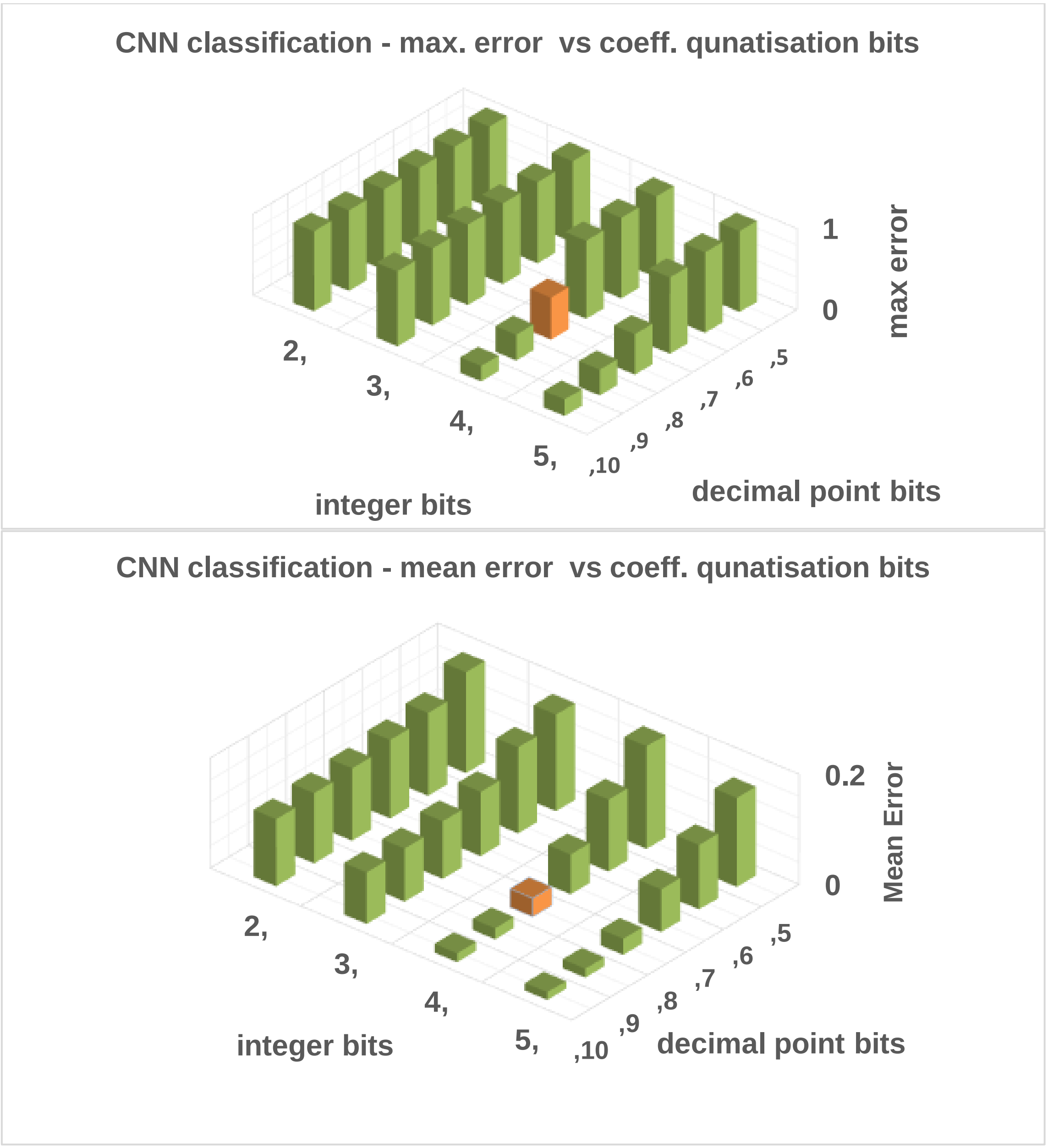


Figure 7. Plots showing relative error vs. different fixed-point quantisation.

CNN Quantisation - Mean Error / Max Error

INTEGER BITS

FRACTIONAL BITS	2.	3.	4.	5.
.5	0.18 / 0.99	0.17 / 0.99	0.18 / 0.99	0.16 / 0.99
.6	0.15 / 0.99	0.15 / 0.99	0.13 / 0.99	0.11 / 0.99
.7	0.14 / 0.99	0.11 / 0.99	0.07 / 0.96	0.07 / 0.95
.8	0.13 / 0.99	0.1 / 0.99	0.03 / 0.52	0.02 / 0.49
.9	0.12 / 0.98	0.09 / 0.98	0.02 / 0.31	0.01 / 0.31
.10	0.12 / 0.98	0.09 / 0.93	0.01 / 0.19	0.01 / 0.20

Table 4. Detailed results from the CNN quantisation.

We can observe, that the quantisation error drops dramatically when increasing the number of bits. The best trade-off between the number of bits and reasonable error is fixed point 4.8.

That is why all the future accelerators will be done for 12 bit fixed-point (4.8) per pixel.

3. Resource Available on the Xilinx SoC platform

To accelerate the algorithm, Xilinx ZC7006 platform has been chosen. It has multiple available resources, including a ZynQ processor – to run the algorithm in software - and FPGA which could accelerate it. The summary of its properties is shown below:

Name	Number	Comments
DSPs	900	
BRAMs	545	Each BRAM at least 8kB
FFs	437200	
LUTs	218600	

Table 5. Properties of the ZC706 platform.

The most important property of the platform is the number of DSPs, since for convolution a lot of multiplications-addition operations are done. Therefore, the accelerator design, especially for Solutiuon 2, will be fitted to the available DSPs. The availability of internal BRAMs is also important, since they can be used to store parts of image data for faster access.

4. Image Convolution Acceleration

When accelerating a signal processing algorithm, first there’s a need to understand the algorithm behind. Algorithm for single layer convolutions, with multiple for loops, is shown below.

```
## convolved2   = np.zeros((22,48,48)) # 22- number of inter-layer imgs, image 48x48
## convolved2_p = np.zeros((22,24,24)) # 22- number of inter-layer imgs, image 24x24
(maxpool2x2)

## Wc2 = np.zeros((22,22,8,8)) # set of 484 kernels

## LAYER 2 ##
for i in range(22): -- 1st for loop
   for j in range(22):  -- 2nd for loop
      # img-kernel convolution operation
      for k in range(48):  -- 3rd for loop
         for l in range(48): --4th for loop
            # convolution result for single pixel with kernel
            for ker_w in range(8): --5th for loop
                  for ker_h in range(8): --6th for loop
                     convolved2[i,k,l] += convolved1[j,k+ker_w,l+ker_h] *Wc2[i,j,ker_w,ker_h])
```

Code 2. Pseudo-code showing multiple loops of the CNN convolution.

Note: Some of the details, like 0-padding have been omitted to simplify the code above.

While analysing this code, there are multiple ways of acceleration. The main purpose in the FPGA acceleration is operation parallelization. As we can see, most of the data processed above is not dependent on each other, therefore can be easily parallelized. The operation used here is multiplication – accumulation. That is why probably the best solution will be to use DSPs present on FPGA platform. Since the same image pixel and same kernel pixels are used a lot, there's a huge

The main challange is to minimise pixel and kernel data transfer → re-use them inside the FPGA. For algorithm parallelization, 2 main solutions have been analysed and implemented.

4.1 Solution 1

The first idea to accelerate the kernel was to accelerate loop 5th and 6th from Code 2. Therefore, a pixel shifting logic has been designed and implemented. The simplified idea is shown in figure below.

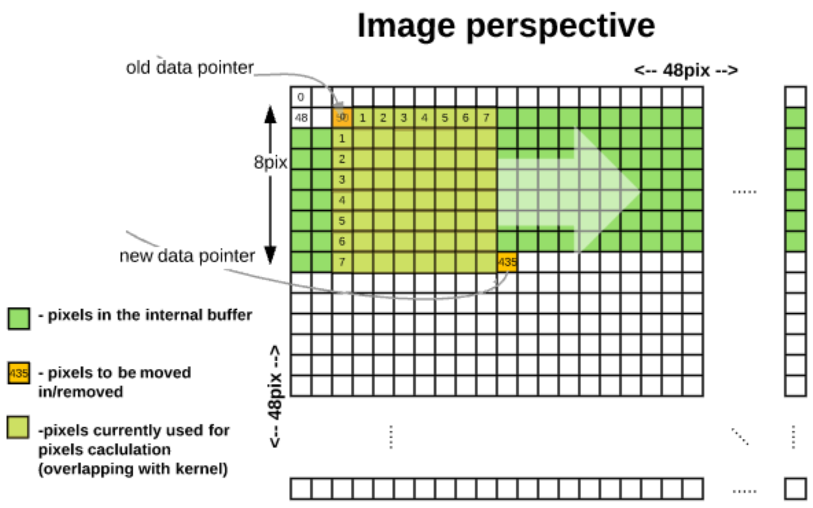


Image perspective
old data pointer
<-- 48pix -->
8pix
new data pointer
<-- 48pix -->
0
48
0 1 2 3 4 5 6 7
435
- pixels in the internal buffer
- pixels to be moved in/removed
-pixels currently used for pixels caclulation (overlapping with kernel)

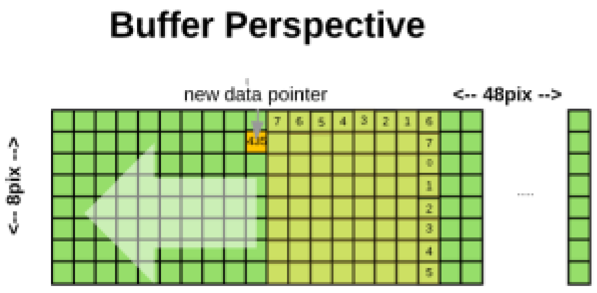


Figure 8. Block diagram showing the main idea of solution 1.

Characteristics:

- 1 output pixel per clock cycle (after filling the buffer at the beggining).
- 64 Multiplications and additions done at a single clock cycle.

4.2 Evaluation of solution 1

Drawbacks:

- the pixels shifting logic is very complicated and takes a lot of resources
- the design takes 70% of the FPGA LUT/FF resources and most of it is for pixel shifting logic.
- The maximum clock frequency has been set as ~70MHz
- With the aforementioned characteristics, the estimated classification rate is every 0.5s.

The solution of the pixel shifting logic has been implemented and synthesized. After seeing the results of the synthesis and estimating the classification rate, a search for better solutions has been started, since the acceleration was not satisfactory.

4.3 Solution 2

The idea of this solution is inspired from the paper “Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks [2]. Unlike the previous solution, the registers here do not take a lot of FPGA resources and it’s using a different approach to the parallelization. The solution is using an additional level of image storage: internal FPGA BRAMs, which have 2 ports and are accessible both from the processor (by some automated interface) and from the accelerator. The paper presents very efficient pixel management technique and different level of parallelization.

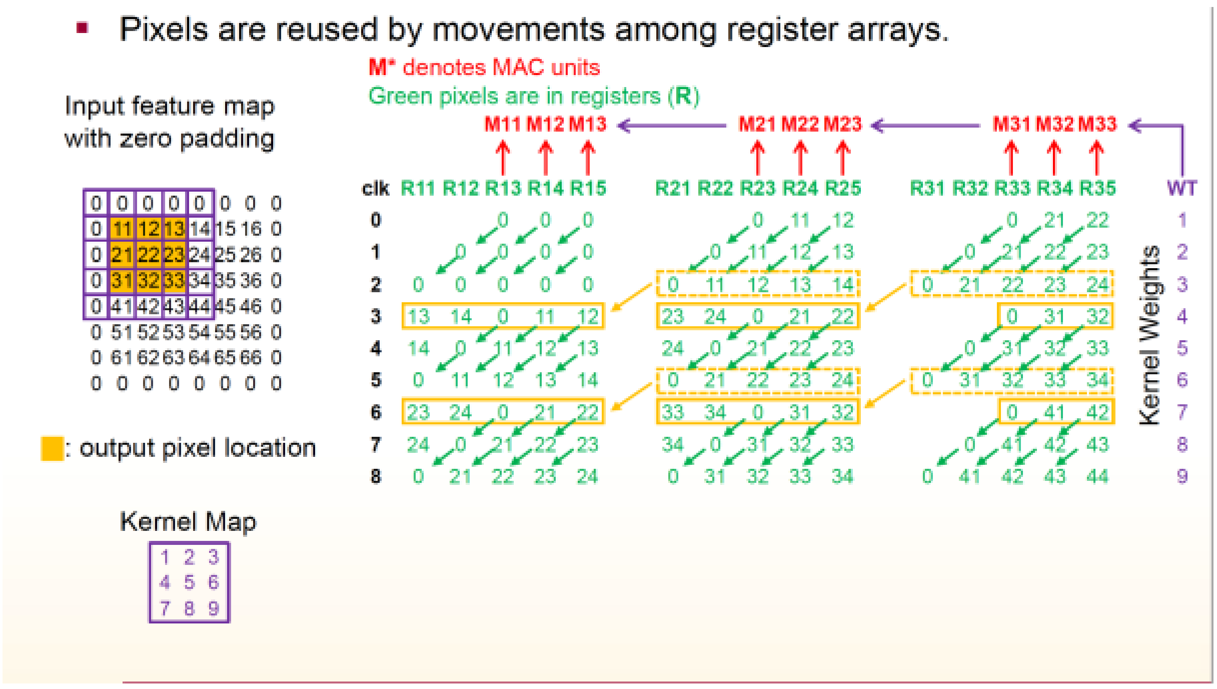


Figure 9. Slide taken from [2] showing the way of pixel re-utilization and parallelization. This example is based on 9-pixel kernel (3x3) and 9-pixel block (3x3).*

*Note: After 9 clock cycles, the convolution results of 3x3 block of pixels is available in the DSPs. Moreover, the same pixel shifter can be used to use it with another sets of DSPs and another sets of Kernels to compute another block results in parallel. To fully understand the solution, read the referenced paper.

When analyzing the paper, we can observe that the pixel shifting logic is complicated, but the operations on the shifter pixels are very generic and once understood and properly implemented for one block, can be re-used for all the pixel blocks.

Based on the method presented in the paper, similar simulations of pixel re-utilization for kernel size of 8x8 (64 length) has been created in MS Excel (poleniuk_CNN_method2.xlsx file, tab: registers). The results have shown a similar way to shift the pixels and allowed to think about a design of a pixel shifting logic.

Choice of the block size and levels of parallelizations
The decision about the exact level of parallelization was determined on basis of the number of DSPs. In particular, 900 DSPs are available. The CNN algorithm has image sizes of 48x48, 24x24, 6x6 → so it makes sense to make the single block size (that is computed as single convolution result after N clock cycles) as 6x6 and since the CNN has 22 images as input/output, so it makes sense to use 6 * 6 * 22 = 792 DSPs. This means, 88% of the available DSPs are utilised.

With the aforementioned level of parallelization, the algorithm together with parallelized operations is presented below.

```
## Wc2 = np.zeros((22,22,8,8)) # set of 484 kernels

## LAYER 2 ##
for i in range(22): --- 1st for loop
   for j in range(22): -- 2nd for loop
      for k in range(48): -- 3rd for loop    # partly parallelised (blocks by 6)
         for L in range(48): --4th for loop  # partly parallelised (blocks by 6)
            for ker_w in range(8): --5th for loop
               for ker_h in range(8): --6th for loop
                  convolved2[i,k,L] += convolved1[j,k+ker_w,L+ker_h] *Wc2[i,j,ker_w,ker_h])
convolved2[:,:,:] +=  b; # adding constant → also done in FPGA
ReLU(convolved2[:,:,:] ) # ReLU rectivier → also done in FPGA
```

Code 3. Example code showing L2 computations. Parallelization are highlighted in red.

4.3.1 Implementation Explanation

The aforementioned method has been implemented in VHDL for kernel size 8x8 and image sizes 48x48, 24x24 and 6x6.

4.3.1.1 Pixel Shifting Logic

Based on the Excel file simulations of the 8x8 kernel and image pixels, the design of single pixel shifting block has been designed.

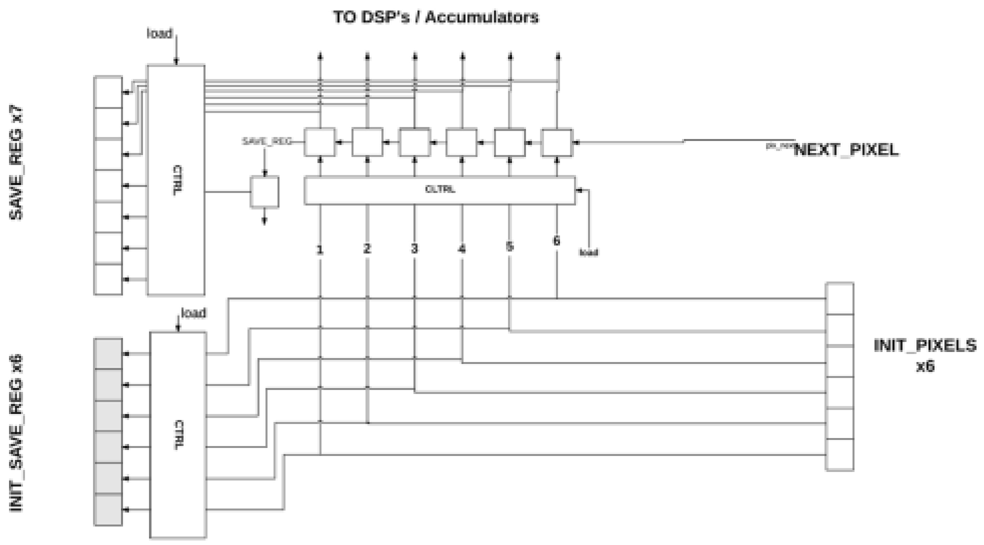


Figure 10. Block Diagram of the single block pixel shifter.

At first clock cycle, the initial pixels are loaded (either from the previous blocks or from the img BRAM) and the pixels start to shift, shifting in the NEXT_PIX which also comes from BRAM or previous block. At the end, all the loaded pixels appear in the output registers INIT_SAVE_REG(6 pixels to initialize the next block) an SAVE_REG(7 pixels saved and then multiplexed as NEXT_PIX for the next block) to be utilized.

The test bench showing pixel shifter is shown below.

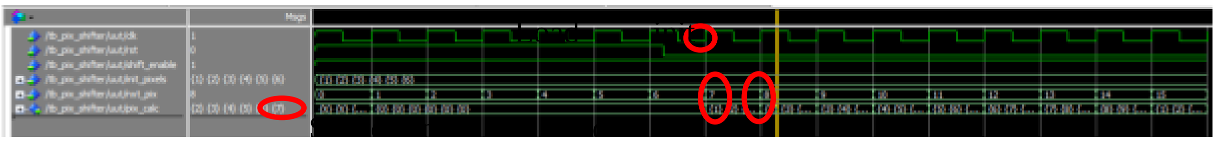


Figure 11. Test Bench showing basic operation of the pixel shifter.

Each pixel shifter is responsible for 6 pixels and therefore we need 6 of them to shift 6x6 block image. The pixels are loaded from BRAM at the first cycle, and then only last pixel shifter takes the image from different image BRAMs and all others are taking the values from the previous block.

The block Diagram of all the pixel shifting logic is shown below.

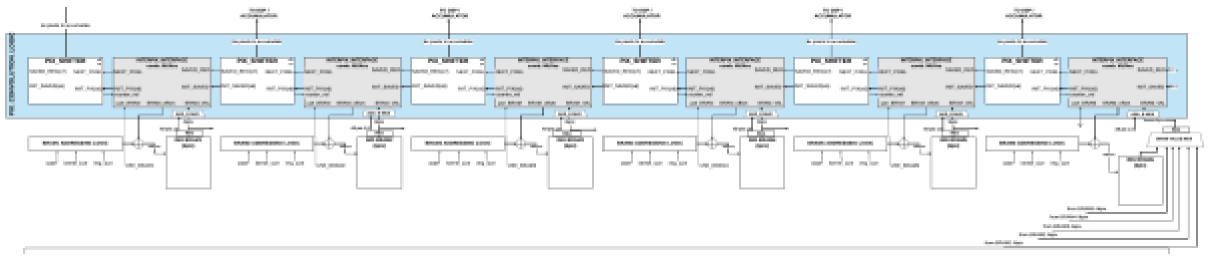


Figure 12. Diagram Showing details of pixels shifting logic.*

Each pixel shifter block needs to store 7 + 6 = 13 pixels. In the pixel logic there are 6 pixel shifter, so all the logic need only 13 * 6 = 78 registers storing 12-bit pixels.

***Notes:**
-Between single pixel shifter, there are combinatorial interfaces (grey) that are set of multiplexers that are interfacing the data between pixel shifters.
- each image shifter is taking it's data from it's own BRAM(first 8 cycles) or the previous pixel shifter(other cycles).
- The last (right) pixel shifter takes its data from different image BRAMs every clock cycle (it's a data source for the system).
- the image is split in between 6 image BRAMs without any repetitions. See Excel file for image BRAM structure.
- the control inputs of the pixel shifters are managed by the state machine logic.

The test bench showing the operation of the pixel shifting logic is shown below.

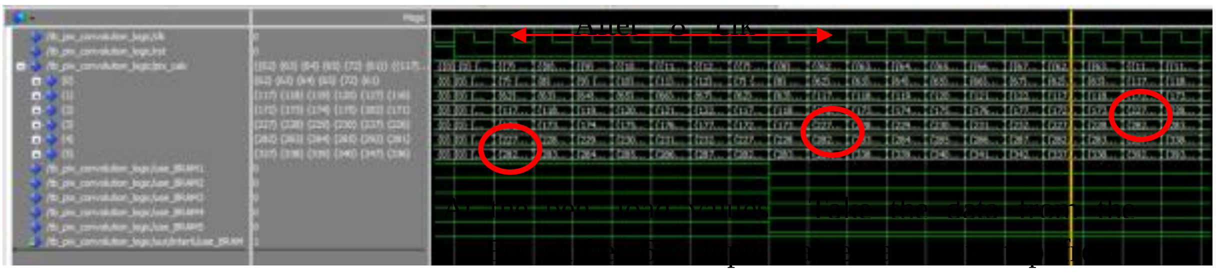


Figure 13. Test Bench showing pixel shifting logic.

4.3.1.2 Pixel Multiplication / addition

For the purpose of parallel addition and multiplication, a block of 792 DSP has been designed. Each cycle it's supposed to multiply the 36 input pixels by 22 different kernel pixels and accumulate it into separate DSP registers. It has a synchronous zero (zero the accumulation result) and accumulate enable signals. The block diagram of the design is shown below.

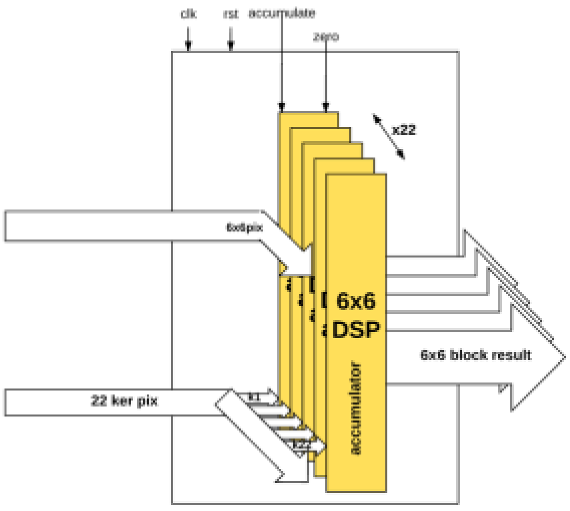


Figure 14. Block diagram of the pixel accumulator block (792 DSPs).

The accumulator design handles maximum and minimum value saturation and after each cycle updates the result at the output. When accumulate enable signal is turned off, the output result is still available.

4.3.1.3 ReLU Rectifier in Hardware

Before saving the pixel values to BRAM, the rectifier function has to be applied. The function is simply a hardware realization of y = max(x,0) in signed 12 bits. If the MSb of the input pixel is 1(the value of pixel is negative then), all the results is zeroed. If not, the results is the same as input.

4.3.1.4 Zero Padding

To solve zero padding problem when performing the kernel operation, 3 pixel boundary is added in the image. That is why, the image size in the BRAM is 54 x 54 pixels. The values in the boundary should be kept at 0. The example image (with numbered pixels) is shown below, on basis on "IMG BRAM" tab in the excel file "poleniuk_cnn_method2.xslx".

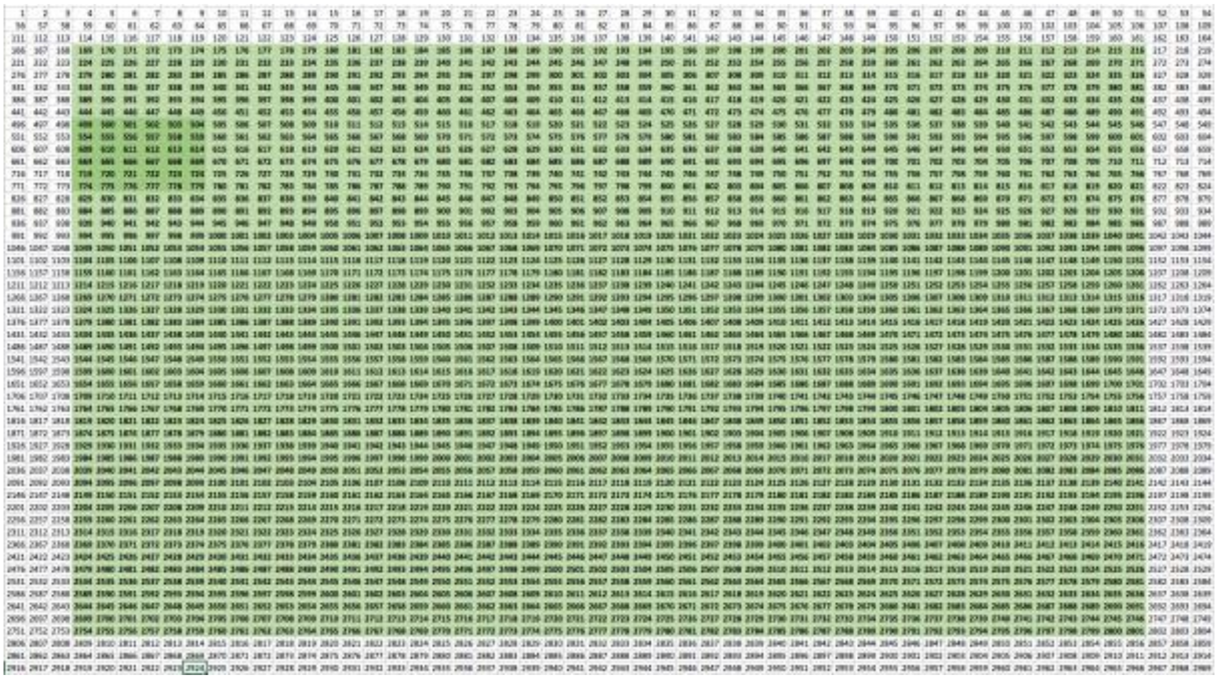
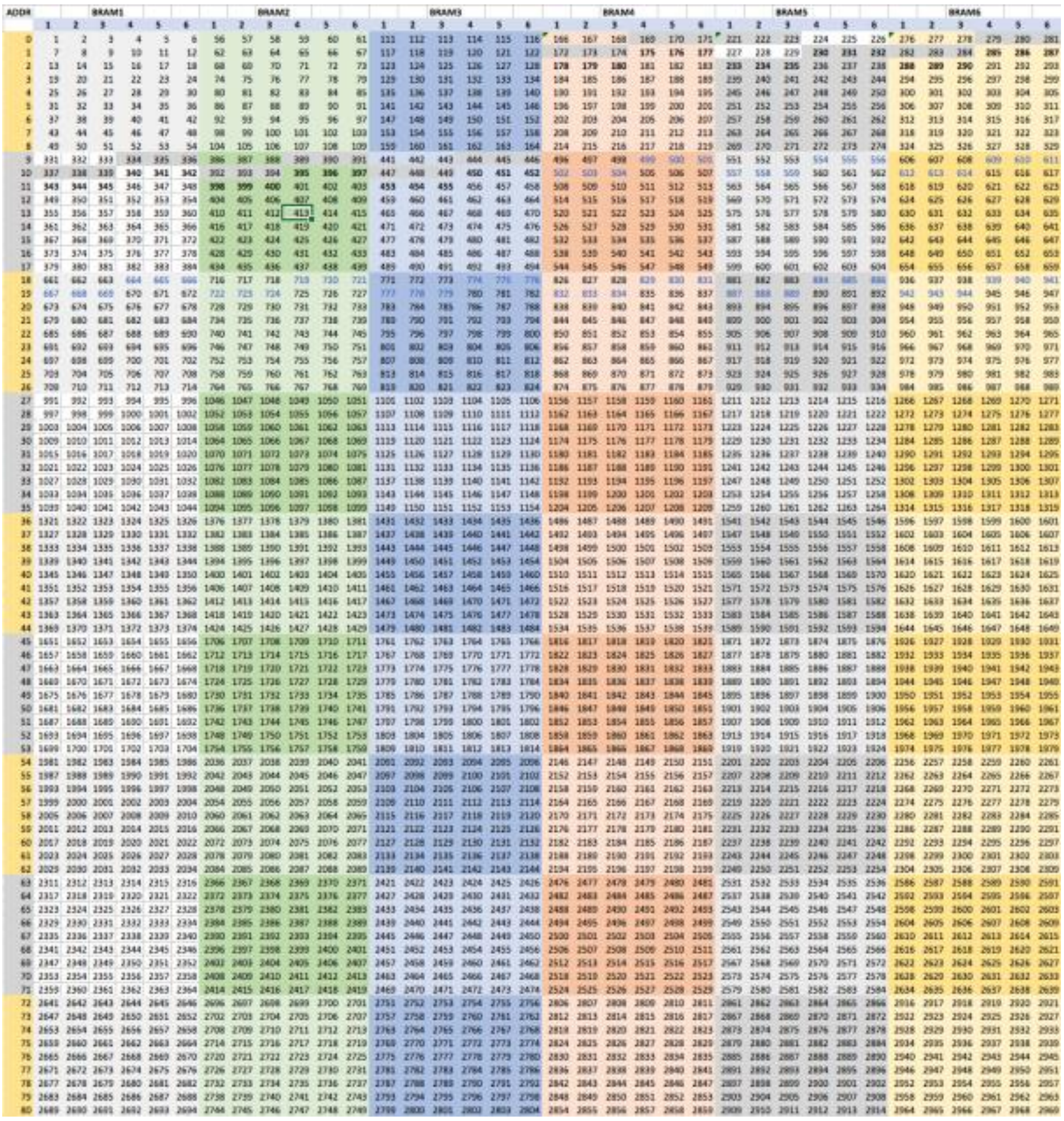


Figure 15. An 48x48image with zero-padding (54x54), nr of pixels used for further considerations.

4.3.1.5 BRAM Image Structure

The BRAM image structure is designed in a way, that during the first 8 cycles of the single block computation each pixel shifter can take the data only from single BRAM image (apart from the last one) and so that there is no repetitive data in BRAMs. This means, that there are 6 BRAMS and each BRAM cell stores 6 pixels (72 bits). Pixel bits are concatenated with each other.



ADDR	BRAM1						BRAM2						BRAM3						BRAM4						BRAM5						BRAM6					
	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6
0	1	2	3	4	5	6	56	57	58	59	60	61	111	112	113	114	115	116	166	167	168	169	170	171	221	222	223	224	225	226	276	277	278	279	280	281
1	7	8	9	10	11	12	62	63	64	65	66	67	117	118	119	120	121	122	172	173	174	175	176	177	227	228	229	230	231	232	282	283	284	285	286	287
2	13	14	15	16	17	18	68	69	70	71	72	73	123	124	125	126	127	128	178	179	180	181	182	183	233	234	235	236	237	238	288	289	290	291	292	293
3	19	20	21	22	23	24	74	75	76	77	78	79	129	130	131	132	133	134	184	185	186	187	188	189	239	240	241	242	243	244	294	295	296	297	298	299
4	25	26	27	28	29	30	80	81	82	83	84	85	135	136	137	138	139	140	190	191	192	193	194	195	245	246	247	248	249	250	300	301	302	303	304	305
5	31	32	33	34	35	36	86	87	88	89	90	91	141	142	143	144	145	146	196	197	198	199	200	201	251	252	253	254	255	256	306	307	308	309	310	311
6	37	38	39	40	41	42	92	93	94	95	96	97	147	148	149	150	151	152	202	203	204	205	206	207	257	258	259	260	261	262	312	313	314	315	316	317
7	43	44	45	46	47	48	98	99	100	101	102	103	153	154	155	156	157	158	208	209	210	211	212	213	263	264	265	266	267	268	318	319	320	321	322	323
8	49	50	51	52	53	54	104	105	106	107	108	109	159	160	161	162	163	164	214	215	216	217	218	219	269	270	271	272	273	274	324	325	326	327	328	329
9	331	332	333	334	335	336	386	387	388	389	390	391	441	442	443	444	445	446	496	497	498	499	500	501	551	552	553	554	555	556	606	607	608	609	610	611
10	337	338	339	340	341	342	392	393	394	395	396	397	447	448	449	450	451	452	502	503	504	505	506	507	557	558	559	560	561	562	612	613	614	615	616	617
11	343	344	345	346	347	348	398	399	400	401	402	403	453	454	455	456	457	458	508	509	510	511	512	513	563	564	565	566	567	568	618	619	620	621	622	623
12	349	350	351	352	353	354	404	405	406	407	408	409	459	460	461	462	463	464	514	515	516	517	518	519	569	570	571	572	573	574	624	625	626	627	628	629
13	355	356	357	358	359	360	410	411	412	413	414	415	465	466	467	468	469	470	520	521	522	523	524	525	575	576	577	578	579	580	630	631	632	633	634	635
14	361	362	363	364	365	366	416	417	418	419	420	421	471	472	473	474	475	476	526	527	528	529	530	531	581	582	583	584	585	586	636	637	638	639	640	641
15	367	368	369	370	371	372	422	423	424	425	426	427	477	478	479	480	481	482	532	533	534	535	536	537	587	588	589	590	591	592	642	643	644	645	646	647
16	373	374	375	376	377	378	428	429	430	431	432	433	483	484	485	486	487	488	538	539	540	541	542	543	593	594	595	596	597	598	648	649	650	651	652	653
17	379	380	381	382	383	384	434	435	436	437	438	439	489	490	491	492	493	494	544	545	546	547	548	549	599	600	601	602	603	604	654	655	656	657	658	659
18	661	662	663	664	665	666	716	717	718	719	720	721	771	772	773	774	775	776	826	827	828	829	830	831	881	882	883	884	885	886	936	937	938	939	940	941
19	667	668	669	670	671	672	722	723	724	725	726	727	777	778	779	780	781	782	832	833	834	835	836	837	887	888	889	890	891	892	942	943	944	945	946	947
20	673	674	675	676	677	678	728	729	730	731	732	733	783	784	785	786	787	788	838	839	840	841	842	843	893	894	895	896	897	898	948	949	950	951	952	953
21	679	680	681	682	683	684	734	735	736	737	738	739	789	790	791	792	793	794	844	845	846	847	848	849	899	900	901	902	903	904	954	955	956	957	958	959
22	685	686	687	688	689	690	740	741	742	743	744	745	795	796	797	798	799	800	850	851	852	853	854	855	905	906	907	908	909	910	960	961	962	963	964	965
23	691	692	693	694	695	696	746	747	748	749	750	751	801	802	803	804	805	806	856	857	858	859	860	861	911	912	913	914	915	916	966	967	968	969	970	971
24	697	698	699	700	701	702	752	753	754	755	756	757	807	808	809	810	811	812	862	863	864	865	866	867	917	918	919	920	921	922	972	973	974	975	976	977
25	703	704	705	706	707	708	758	759	760	761	762	763	813	814	815	816	817	818	868	869	870	871	872	873	923	924	925	926	927	928	978	979	980	981	982	983
26	709	710	711	712	713	714	764	765	766	767	768	769	819	820	821	822	823	824	874	875	876	877	878	879	929	930	931	932	933	934	984	985	986	987	988	989
27	991	992	993	994	995	996	1046	1047	1048	1049	1050	1051	1101	1102	1103	1104	1105	1106	1156	1157	1158	1159	1160	1161	1211	1212	1213	1214	1215	1216	1266	1267	1268	1269	1270	1271
28	997	998	999	1000	1001	1002	1052	1053	1054	1055	1056	1057	1107	1108	1109	1110	1111	1112	1162	1163	1164	1165	1166	1167	1217	1218	1219	1220	1221	1222	1272	1273	1274	1275	1276	1277
29	1003	1004	1005	1006	1007	1008	1058	1059	1060	1061	1062	1063	1113	1114	1115	1116	1117	1118	1168	1169	1170	1171	1172	1173	1223	1224	1225	1226	1227	1228	1278	1279	1280	1281	1282	1283
30	1009	1010	1011	1012	1013	1014	1064	1065	1066	1067	1068	1069	1119	1120	1121	1122	1123	1124	1174	1175	1176	1177	1178	1179	1229	1230	1231	1232	1233	1234	1284	1285	1286	1287	1288	1289
31	1015	1016	1017	1018	1019	1020	1070	1071	1072	1073	1074	1075	1125	1126	1127	1128	1129	1130	1180	1181	1182	1183	1184	1185	1235	1236	1237	1238	1239	1240	1290	1291	1292	1293	1294	1295
32	1021	1022	1023	1024	1025	1026	1076	1077	1078	1079	1080	1081	1131	1132	1133	1134	1135	1136	1186	1187	1188	1189	1190	1191	1241	1242	1243	1244	1245	1246	1296	1297	1298	1299	1300	1301
33	1027	1028	1029	1030	1031	1032	1082	1083	1084	1085	1086	1087	1137	1138	1139	1140	1141	1142	1192	1193	1194	1195	1196	1197	1247	1248	1249	1250	1251	1252	1302	1303	1304	1305	1306	1307
34	1033	1034	1035	1036	1037	1038	1088	1089	1090	1091	1092	1093	1143	1144	1145	1146	1147	1148	1198	1199	1200	1201	1202	1203	1253	1254	1255	1256	1257	1258	1308	1309	1310	1311	1312	1313
35	1039	1040	1041	1042	1043	1044	1094	1095	1096	1097	1098	1099	1149	1150	1151	1152	1153	1154	1204	1205	1206	1207	1208	1209	1259	1260	1261	1262	1263	1264	1314	1315	1316	1317	1318	1319
36	1321	1322	1323	1324	1325	1326	1376	1377	1378	1379	1380	1381	1431	1432	1433	1434	1435	1436	1486	1487	1488	1489	1490	1491	1541	1542	1543	1544	1545	1546	1596	1597	1598	1599	1600	1601
37	1327	1328	1329	1330	1331	1332	1382	1383	1384	1385	1386	1387	1437	1438	1439	1440	1441	1442	1492	1493	1494	1495	1496	1497	1547	1548	1549	1550	1551	1552	1602	1603	1604	1605	1606	1607
38	1333	1334	1335	1336	1337	1338	1388	1389	1390	1391	1392	1393	1443	1444	1445	1446	1447	1448	1498	1499	1500	1501	1502	1503	1553	1554	1555	1556	1557	1558	1608	1609	1610	1611	1612	1613
39	1339	1340	1341	1342	1343	1344	1394	1395	1396	1397	1398	1399	1449	1450	1451	1452	1453	1454	1504	1505	1506	1507	1508	1509	1559	1560	1561	1562	1563	1564	1614	1615	1616	1617	1618	1619
40	1345	1346	1347	1348	1349	1350	1400	1401	1402	1403	1404	1405	1455	1456	1457	1458	1459	1460	1510	1511	1512	1513	1514	1515	1565	1566	1567	1568	1569	1570	1620	1621	1622	1623	1624	1625
41	1351	1352	1353	1354	1355	1356	1406	1407	1408	1409	1410	1411	1461	1462	1463	1464	1465	1466	1516	1517	1518	1519	1520	1521	1571	1572	1573	1574	1575	1576	1626	1627	1628	1629	1630	1631
42	1357	1358	1359	1360	1361	1362	1412	1413	1414	1415	1416	1417	1467	1468	1469	1470	1471	1472	1522	1523	1524	1525	1526	1527	1577	1578	1579	1580	1581	1582	1632	1633	1634	1635	1636	1637
43	1363	1364	1365	1366	1367	1368	1418	1419	1420	1421	1422	1423	1473	1474	1475	1476	1477	1478	1528	1529	1530	1531	1532	1533	1583	1584	1585	1586	1587	1588	1638	1639	1640	1641	1642	1643
44	1369	1370	1371	1372	1373	1374	1424	1425	1426	1427	1428	1429	1479	1480	1481	1482	1483	1484	1534	1535	1536	1537	1538	1539	1589	1590	1591	1592	1593	1594	1644	1645	1646	1647	1648	1649
45	1651	1652	1653	1654	1655	1656	1706	1707	1708	1709	1710	1711	1761	1762	1763	1764	1765	1766	1816	1817	1818	1819	1820	1821	1871	1872	1873	1874	1875	1876	1926	1927	1928	1929	1930	1931
46	1657	1658	1659	1660	1661	1662	1712	1713	1714	1715	1716	1717	1767	1768	1769	1770	1771	1772	1822	1823	1824	1825	1826	1827	1877	1878	1879	1880	1881	1882	1932	1933	1934	1935	1936	1937
47	1663	1664	1665	1666	1667	1668	1718	1719	1720	1721	1722	1723	1773	1774	1775	1776	1777	1778	1828	1829	1830	1831	1832	1833	1883	1884	1885	1886	1887	1888	1938	1939	1940	1941	1942	1943
48	1669	1670	1671	1672	1673	1674	1724	1725	1726	1727	1728	1729	1779	1780	1781	1782	1783	1784	1834	1835	1836	1837	1838	1839	1889	1890	1891	1892	1893	1894	1944	1945	1946	1947	1948	1949
49	1675	1676	1677	1678	1679	1680	1730	1731	1732	1733	1734	1735	1785	1786	1787	1788	1789	1790	1840	1841	1842	1843	1844	1845	1895	1896	1897	1898	1899	1900	1950	1951	1952	1953	1954	1955
50	1681	1682	1683	1684	1685	1686	1736	1737	1738	1739	1740	1741	1791	1792	1793	1794	1795	1796	1846	1847	1848	1849	1850	1851	1901	1902	1903	1904	1905	1906	1956	1957	1958	1959	1960	1961
51	1687	1688	1689	1690	1691	1692	1742	1743	1744	1745	1746	1747	1797	1798	1799	1800	1801	1802	1852	1853	1854	1855	1856	1857	1907	1908	1909	1910	1911	1912	1962	1963	1964	1965	1966	1967
52	1693	1694	1695	1696	1697	1698	1748	1749	1750	1751	1752	1753	1803	1804	1805	1806	1807	1808	1858	1859	1860	1861	1862	1863	1913	1914	1915	1916	1917	1918	1968	1969	1970	1971	1972	1973
53	1699	1700	1701	1702	1703	1704	1754	1755	1756	1757	1758	1759	1809	1810	1811	1812	1813	1814	1864	1865	1866	1867	1868	1869	1919	1920	1921	1922	1923	1924	1974	1975	1976	1977	1978	1979
54	1981	1982	1983	1984	1985	1986	2036	2037	2038	2039	2040	2041	2091	2092	2093	2094	2095	2096	2146	2147	2148	2149	2150	2151	2201	2202	2203	2204	2205	2206	2256	2257	2258	2259	2260	2261
55	1987	1988	1989	1990	1991	1992	2042	2043	2044	2045	2046	2047	2097	2098	2099	2100	2101	2102	2152	2153	2154	2155	2156	2157	2207	2208	2209	2210	2211	2212	2262	2263	2264	2265	2266	2267
56	1993	1994	1995	1996	1997	1998	2048	2049	2050	2051	2052	2053	2103	2104	2105	2106	2107	2108	2158	2159	2160	2161	2162	2163	2213	2214	2215	2216	2217	2218	2268	2269	2270	2271	2272	2273
57	1999	2000	2001	2002	2003	2004	2054	2055	2056	2057	2058	2059	2109	2110	2111	2112	2113	2114	2164	2165	2166	2167	2168	2169	2219	2220	2221	2222	2223	2224	2274	2275	2276	2277	2278	2279
58	2005	2006	2007	2008	2009	2010	2060	2061	2062	2063	2064	2065	2115	2116	2117	2118	2119	2120	2170	2171	2172	2173	2174	2175	2225	2226	2227	2228	2229	2230	2280	2281	2282	2283	2284	2285
59	2011	2012	2013	2014	2015	2016	2066	2067	2068	2069	2070	2071	2121	2122	2123	2124	2125	2126	2176	2177	2178	2179	2180	2181	2231	2232	2233	2234	2235	2236	2286	2287	2288	2289	2290	2291
60	2017	2018	2019	2020	2021	2022	2072	2073	2074	2075	2076	2077	2127	2128	2129	2130	2131	2132	2182	2183	2184	2185	2186	2187	2237	2238	2239	2240	2241	2242	2292	2293	2294	2295	2296	2297
61	2023	2024	2025	2026	2027	2028	2078	2079	2080	2081	2082	2083	2133	2134	2135	2136	2137	2138	2188	2189	2190	2191	2192	2193	2243	2244	2245	2246	2247	2248	2298	2299	2300	2301	2302	2303
62	2029	2030	2031	2032	2033	2034	2084	2085	2086	2087	2088	2089	2139	2140	2141	2142	2143	2144	2194	2195	2196	2197	2198	2199	2249	2250	2251	2252	2253	2254	2304	2305	2306	2307	2308	2309
63	2311	2312	2313	2314	2315	2316	2366	2367	2368	2369	2370	2371	2421	2422	2423	2424	2425	2426	2476	2477	2478	2479	2480	2481	2531	2532	2533	2534	2535	2536	2586	2587	2588	2589	2590	2591
64	2317	2318	2319	2320	2321	2322	2372	2373	2374	2375	2376	2377	2427	2428	2429	2430	2431	2432	2482	2483	2484	2485	2486	2487	2537	2538	2539	2540	2541	2542	2592	2593	2594	2595	2596	2597
65	2323	2324	2325	2326	2327	2328	2378	2379	2380	2381	2382	2383	2433	2434	2435	2436	2437	2438	2488	2489	2490	2491	2492	2493	2543	2544	2545	2546	2547	2548	2598	2599	2600	2601	2602	2603
66	2329	2330	2331	2332	2333	2334	2384	2385	2386	2387	2388	2389	2439	2440	2441	2442	2443	2444	2494	2495	2496	2497	2498	2499	2549	2550	2551	2552	2553	2554	2604	2605	2606	2607	2608	2609
67	2335	2336	2337	2338	2339	2340	2390	2391	2392	2393	2394	2395	2445	2446	2447	2448	2449	2450	2500	2501	2502	2503	2504	2505	2555	2556	2557	2558	2559	2560	2610	2611	2612	2613	2614	2615
68	2341	2342	2343	2344	2345	2346	2396	2397	2398	2399	2400	2401	2451	2452	2453	2454	2455	2456	2506	2507	2508	2509	2510	2511	2561	2562	2563	2564	2565	2566	2616	2617	2618	2619	2620	2621
69	2347	2348	2349	2350	2351	2352	2402	2403	2404	2405	2406	2407	2457	2458	2459	2460	2461	2462	2512	2513	2514	2515	2516	2517	2567	2568	2569	2570	2571	2572	2622	2623	2624	2625	2626	2627
70	2353	2354	2355	2356	2357	2358	2408	2409	2410	2411	2412	2413	2463	2464	2465	2466	2467	2468	2518	2519	2520	2521	2522	2523	2573	2574	2575	2576	2577	2578	2628	2629	2630	2631	2632	2633
71	2359	2360	2361	2362	2363	2364	2414	2415	2416	2417	2418	2419	2469	2470	2471	2472	2473	2474	2524	2525	2526	2527	2528	2529	2579	2580	2581	2582	2583	2584	2634	2635	2636	2637	2638	2639
72	2641	2642	2643	2644	2645	2646	2696	2697	2698	2699	2700	2701	2751	2752	2753	2754	2755	2756	2806	2807	2808	2809	2810	2811	2861	2862	2863	2864	2865	2866	2916	2917	2918	2919	2920	2921
73	2647	2648	2649	2650	2651	2652	2702	2703	2704	2705	2706	2707	2757	2758	2759	2760	2761	2762	2812	2813	2814	2815	2816	2817	2867	2868	2869	2870	2871	2872	2922	2923	2924	2925	2926	2927
74	2653	2654	2655	2656	2657	2658	2708	2709	2710	2711	2712	2713	2763	2764	2765	2766	2767	2768	2818	2819	2820	2821	2822	2823	2873	2874	2875	2876	2877	2878	2928	2929	2930	2931	2932	2933
75	2659	2660	2661	2662	2663	2664	2714	2715	2716	2717	2718	2719	2769	2770	2771	2772	2773	2774	2824	2825	2826	2827	2828	2829	2879	2880	2881	2882	2883	2884	2934	2935	2936	2937	2938	2939
76	2665	2666	2667	2668	2669	2670	2720	2721	2722	2723	2724	2725	2775	2776	2777	2778	2779	2780	2830	2831	2832	2833	2834	2835	2885	2886	2887	2888	2889	2890	2940	2941	2942	2943	2944	2945
77	2671	2672	2673	2674	2675	2676	2726	2727	2728	2729	2730	2731	2781	2782	2783	2784	2785	2786	2836	2837	2838	2839	2840	2841	2891	2892	2893	2894	2895	2896	2946	2947	2948	2949	2950	2951
78	2677	2678	2679	2680	2681	2682	2732	2733	2734	2735	2736	2737	2787	2788	2789	2790	2791	2792	2842	2843	2844	2845	2846	2847	2897	2898	2899	2900	2901	2902	2952	2953	2954	2955	2956	2957
79	2683	2684	2685	2686	2687	2688	2738	2739	2740	2741	2742	2743	2793	2794	2795	2796	2797	2798	2848	2849	2850	2851	2852	2853	2903	2904	2905	2906	2907	2908	2958	2959	2960	2961	2962	2963
80	2689	2690	2691	2692	2693	2694	2744	2745	2746	2747	2748	2749	2799	2800	2801	2802	2803	2804	2854	2855	2856	2857	2858	2859	2909	2910	2911	2912	2913	2914	2964	2965	2966	2967	2968	2969

Figure 16. Single image BRAM structure (in 6 image BRAMS).

Each consecutive 9 BRAM cells contain a row of image. Therefore, rows 0-8 of BRAM 1 contain 1st line(1-54) of the image, and then rows 9-18 contain 6th line of the image etc. For BRAM2, rows 0-8 contain 2nd line of the image and 9-18 contain 7th line of the image. Each image takes 81 rows of image BRAMs 1-6.

It's needed to store 22 full images; therefore, it takes 81* 22 = 1782 addresses. Each image is stored just after the previous one. Since both read and resulting images are stored in the same BRAM, the BRAM is divided into 2 sectors of the 22-image size and for different layers images are read and write in the swapped sectors. If the image has different size(24x24, 6x6), it takes smaller space (but the sector and size per image remains the same). The table showing R/W sectors for each layers are shown below.

Layer	Read Addresses	Write addresses
L1	0-1781	1782-3563
L2	1782-3563	0-1781
L3	0-1781	1782-3563
L4	1782-3563	0-1781
L5	0-1781	1782-3563
L6	1782-3563	0-1781

Table 6. Addressing of different image sectors.

4.3.1.6 BRAM addressing

Each BRAM has 2 ports. Each port works the same way. One port is connected to the processor, where the software can read and write images. The other port is connected to the accelerator, which reads the input image and writes the result.

Even though the BRAM needs to store 72 bits per cell, its size been configured to 128 bits to be able to handle it from the processor side easily (the available cell sizes are only powers of 2). The 72 image pixels are aligned to LSb and MSb's are padded with 0's. Since the address of the BRAM is byte aligned, keeping the previous notation, the real address that the image should be should be written or read is shifted left (padded 0's) 4 bits. As an example, cell address 81 (0x51) will become 1296 (0x510).

4.3.1.7 BRAM Image Writing

After the result of single block (6x6 pixels) is ready, the image needs to be written back to the BRAM memory into the proper space, i.e. not from the beginning address, where the zero-padding zero begins, but where the real image begins. The image below shows the sample image result block (on the right) and the image BRAMs (on the left).

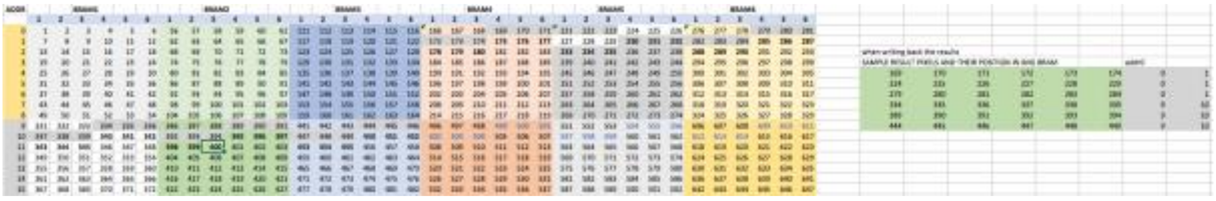

[illegible]

Figure 17. Bram structure when writing: Unaligned pixels.

Please note, that the results need to be written partly, into 2 cells in the BRAM. That is why, saving state machine first needs to read what is in the BRAM, save it, write the new values, and then do the same for the second address.

4.3.1.8 BRAM output registers for Read Access

To allow to obtain the timing between BRAM and the pixel shifting design, a register has been added in the output of the BRAM. Hence, after putting the correct address, the result is present in BRAM output after 2 clock cycles. The writing works as in the documentation, after 1 clock cycle.

4.3.1.9 Kernel BRAMs

The kernel BRAM structure is shown in the excel file “poleniuk_cnn_method2.xslx” in the tab “KERNEL BRAM”. Since the kernel shifting logic had only been designed for 8x8 kernels, the 4x4 kernels for layers L3-L6 has been padded by 0’s and used as 8x8.

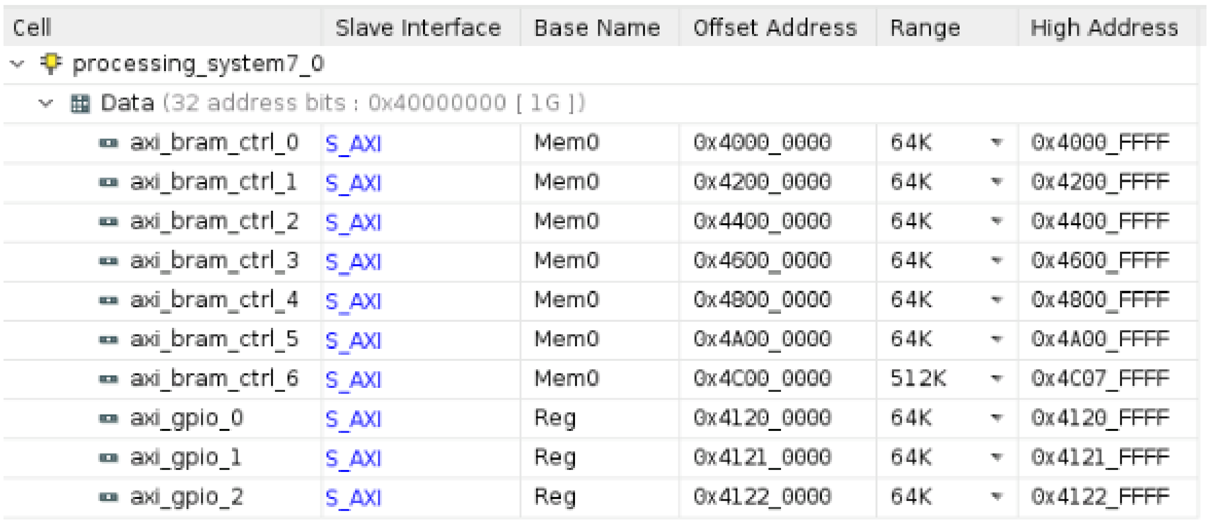
Each kernel is 8x8 = 64 pixels and 22 kernel results are parallelized. Hence, for each kernel single pixel needs to be accessible in parallel. For instance, 1st pixel of all 22 kernels for L1 needs to be available in parallel.

The kernel pixels are also 12 bits, and it’s need 22 of them in one cell to be read in parallel. Therefore, 22 * 22 = 264. The BRAM cell size needs to be the power of 2, so the BRAM cell size has been chosen as 512 bits. The kernel values are aligned to LSb and all the MSb are padded with 0’s. The kernel BRAM structure is shown in the excel file “poleniuk_cnn_method2.xslx” in tab “KERNEL BRAM”.

1st layer uses only 22 kernels but L2-L6 uses 484 kernels each. That is why, the number of BRAM kernel cells is 64 * 1 + 5 * 64 * 22 = 7104. The 6 b constants for each layer, are stored at the end of the BRAM size. That is why, the final number of cells is 7104 + 6 = 7110.

4.3.1.10 Accessing BRAMs from ZynQ processor

It is possible to access each 32-bit part of the BRAM cell with the byte-aligned addressing. The address space is shown below:



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	64K	0x4000_FFFF
axi_bram_ctrl_1	S_AXI	Mem0	0x4200_0000	64K	0x4200_FFFF
axi_bram_ctrl_2	S_AXI	Mem0	0x4400_0000	64K	0x4400_FFFF
axi_bram_ctrl_3	S_AXI	Mem0	0x4600_0000	64K	0x4600_FFFF
axi_bram_ctrl_4	S_AXI	Mem0	0x4800_0000	64K	0x4800_FFFF
axi_bram_ctrl_5	S_AXI	Mem0	0x4A00_0000	64K	0x4A00_FFFF
axi_bram_ctrl_6	S_AXI	Mem0	0x4C00_0000	512K	0x4C07_FFFF
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_1	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_2	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF

Table 7. Kernel and BRAM address space in AXI bus.

4.3.1.11 State Machine

The block diagram of the full accelerator system and state machine is shown below.

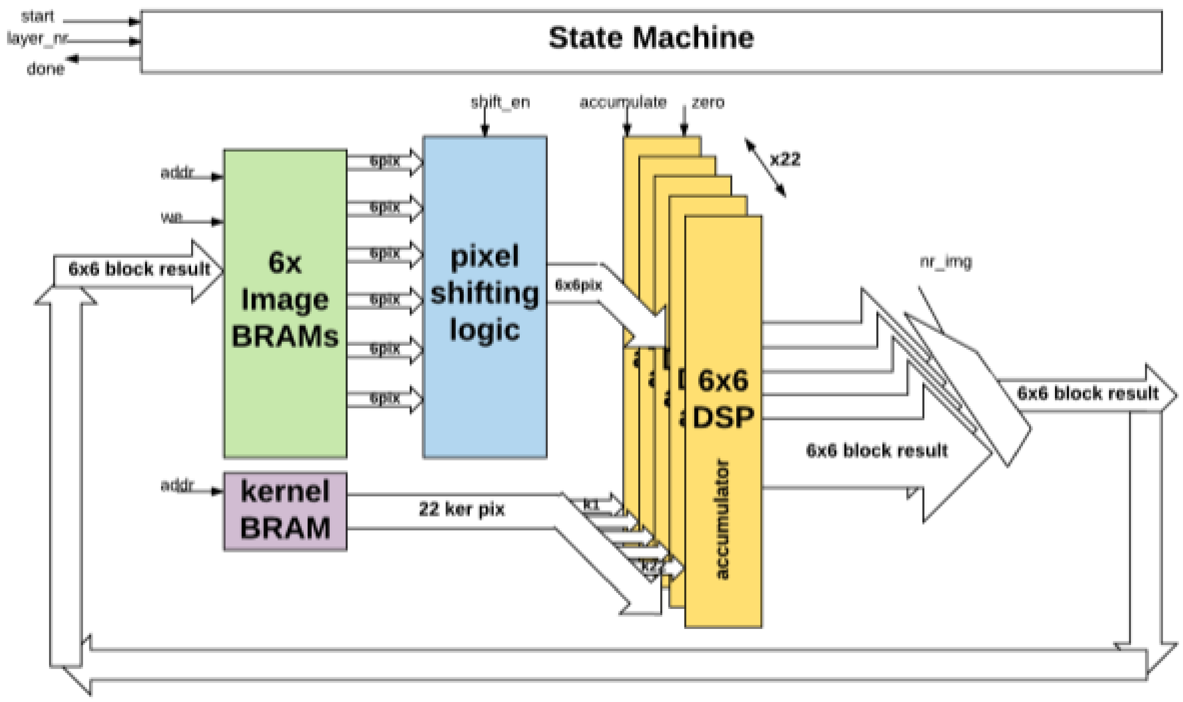


Figure 18. Simplified Block Diagram of Solution 2.

The main purpose of the state machine is to manage the addresses of the BRAMs, based on layer number (which determines image size, sector to R/W, number of input images etc.) and control the pixel shifting logic, DSPs and writing the result. The simplified state machine diagram is shown below.

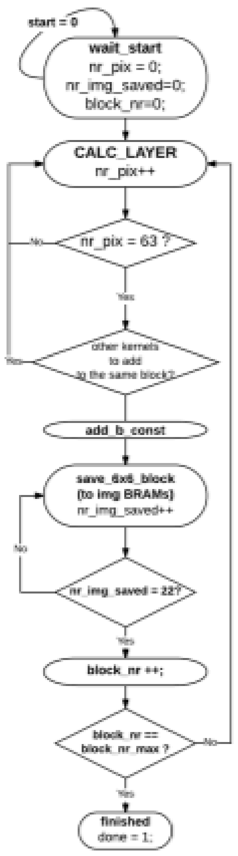


Figure 19. Simplified State machine Diagram of Solution 2.

The explanation for each of the states and different counters are shown below:
counters:
Nr_pix – counts from 0 to 63, when the DSPs are accumulating the kernel result.
Nr_img_saved – from 0 to 21, when saving the images needed to iterate through the DSP results.
Block_nr (block_row and block_col) – the results are computed in 6x6 blocks and each image has 8x8 blocks (for 48x48 image), 4x4 blocks (for 24x24 image) or 1x1 block(for 6x6 image).

States:
Wait_start – waiting for start signal. Connected to GPIO.

CALC_LAYER – wait for the result, change the kernel pixels and shift the pixel values.

Other kernels to add to the same block ? – depending on the layer, after computing single results for 22 kernels we need to iterate to another set / column of kernels(L2-L5) or not (L1).

Add_b_const – puts the kernel address to point to the b constant for the current layer and make them accumulate it in the DSP (the pixels at this point are multiplexed to be 1.0, and therefore the DSP_value += pixel_value * b_const(from kernel BRAM), pixel value = 1.0 so DSP_value += b_const.

Save_6x6_block – reads what in the current cell , writes the partial value from the DSPs, reads another cell and writes another partial result from the DSPs.

Finished – when all the image blocks(8x8 for L1-L2, 4x4 for L3-L4, 1x1 for L5-L6) are saved, the done flag is put to 1.

The test bench for the whole accelerator design, as an example for L1, is shown below:

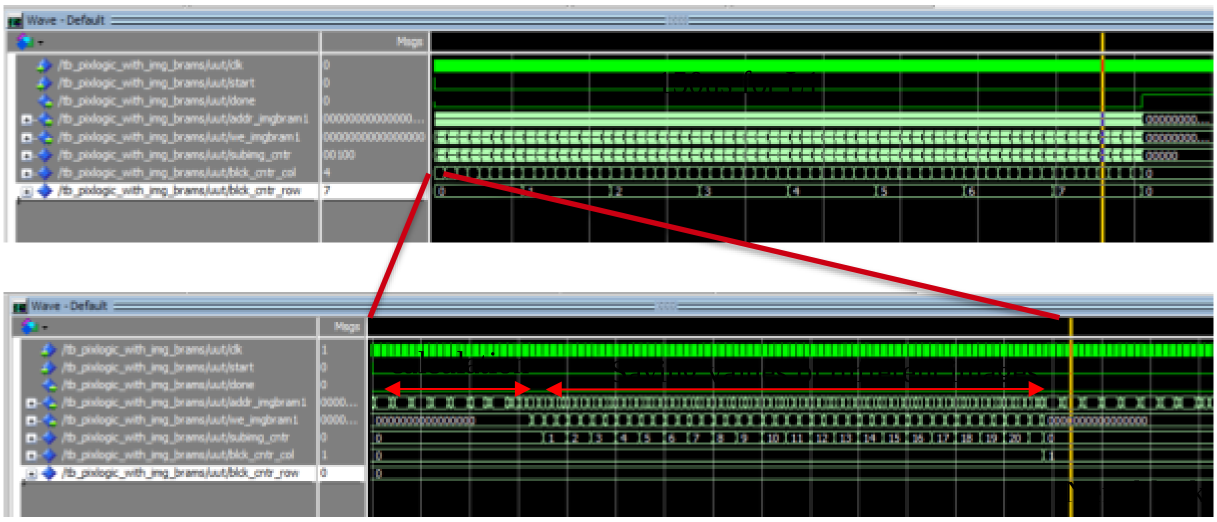

Wave - Default
Msgs
/tb_pixlogic_with_img_brams/uut/clk
/tb_pixlogic_with_img_brams/uut/start
/tb_pixlogic_with_img_brams/uut/done
/tb_pixlogic_with_img_brams/uut/addr_imgbram1
/tb_pixlogic_with_img_brams/uut/we_imgbram1
/tb_pixlogic_with_img_brams/uut/subimg_cntr
/tb_pixlogic_with_img_brams/uut/blck_cntr_col
/tb_pixlogic_with_img_brams/uut/blck_cntr_row
150us for L1
Calculation
Saving values of different images
Next block

Figure 20. Test bench of the accelerator logic.

4.3.1.12 Block Design with ZynQ processor

To be able to load the kernel coefficients and images, each BRAM ports 1 has been connected to the BRAM controller (connected to AXI bus and ZynQ) and BRAM ports 2 to the accelerator. The accelerator start, reset and layer_nr signals are managed by connecting it into a GPO, as well as the done signal to GPI.

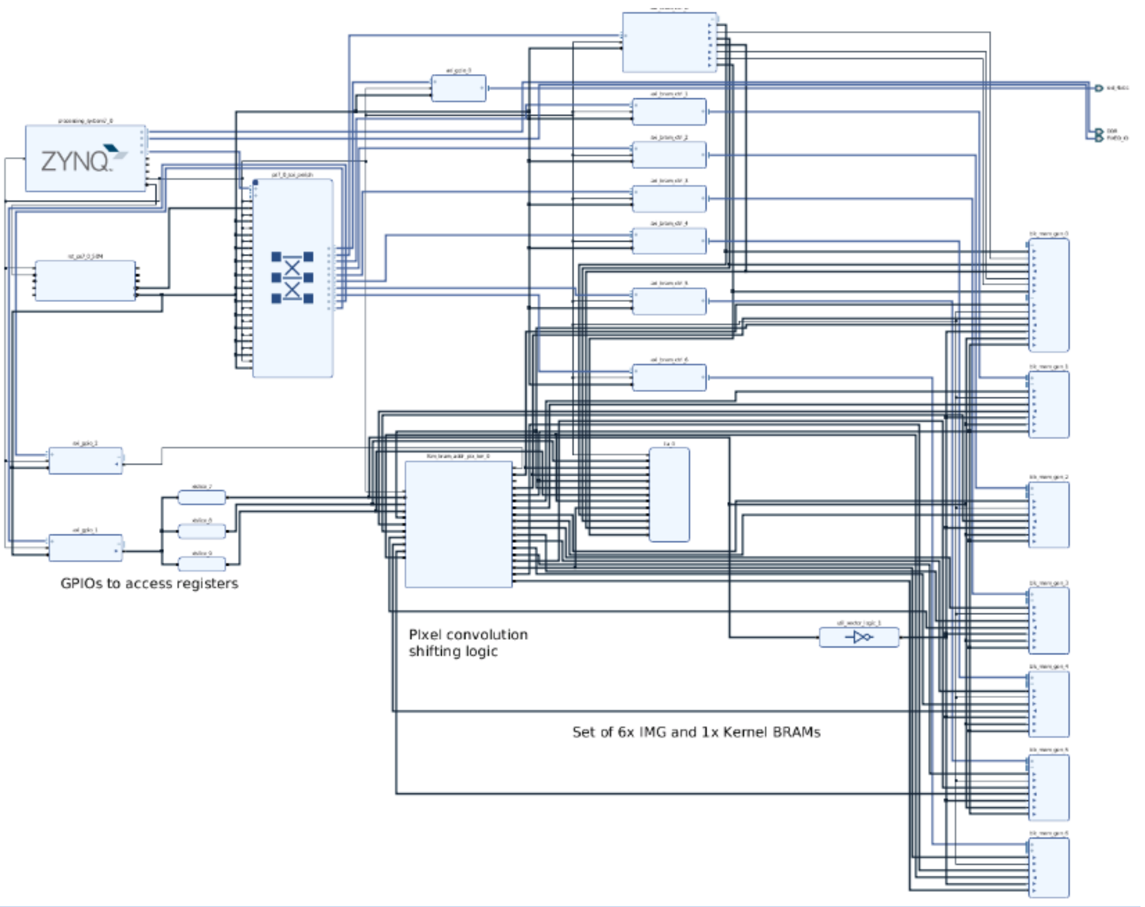
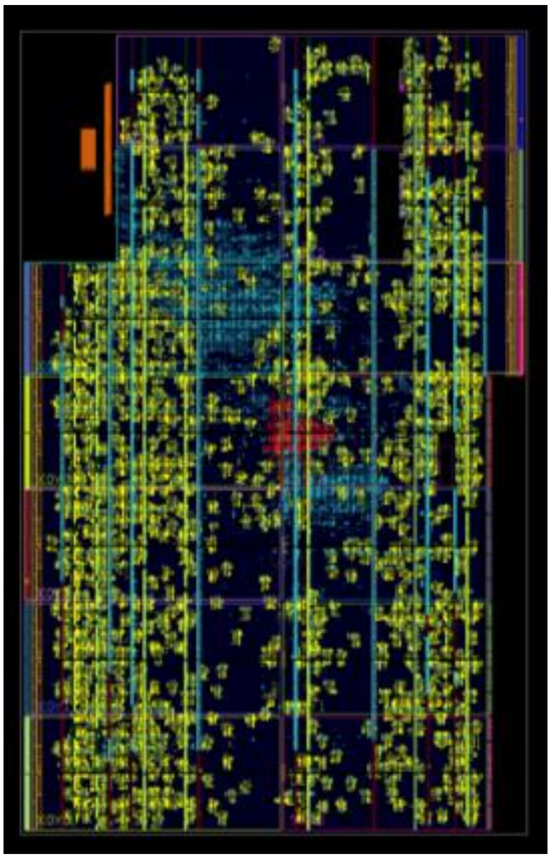
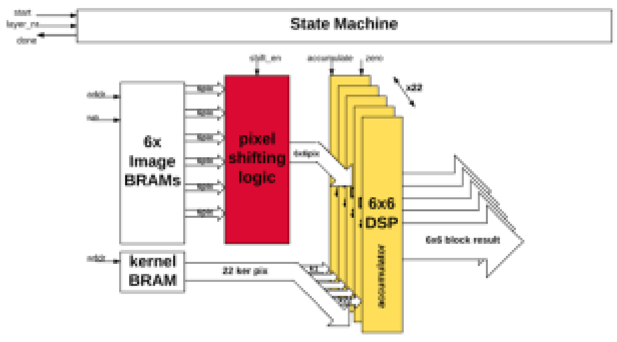


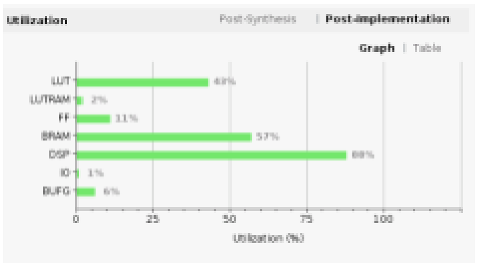
Figure 21. System block diagram with BRAMs and ZynQ processor.

4.4 Evaluation of Solution 2

The aforementioned solution had been synthesized in Vivado suite with standard implementation parameters. The details of the implementation are shown below:







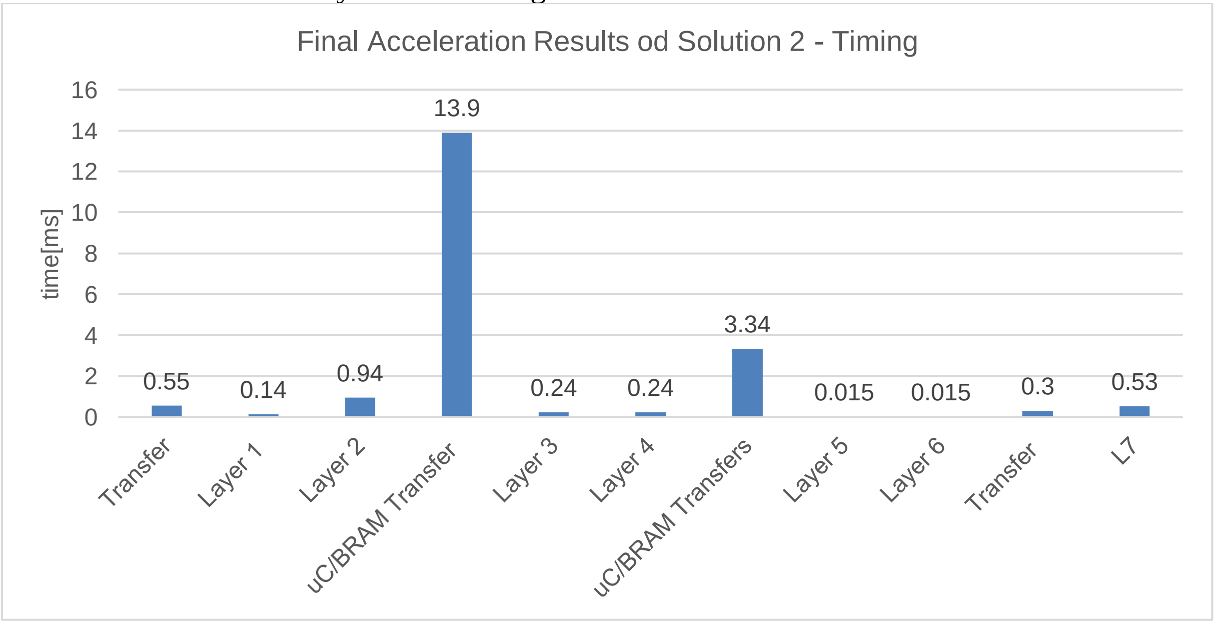
FPGA Frequency : 110MHz

Figure 22. Place & Route details of Solution 2.

Observations:

- Most of the design is pixel accumulators-DSPs (yellow) and the pixel shifting logic takes only small part of it.
- 792 DSPs are utilised.
- 896kB of BRAM space is used.

To check the timing of the solution, a ZYNQ software has been used to trigger the start of the accelerator for different layers. The timing results are shown below.



	Software timings[s]	**Accelerated timings [s]**	**FPGA accel. (xNR)**	**posiible when maxpool on FPGA (xNR)**
python tensorflow	3		149	48051
Mac C/C++	0.6		30	1508
ZynQ Petalinux	11.5	0.02	571	28212

	before	**now**	**future**
Classifications per second	0.1 to 2	50	~500

Figure 23. Timing Results After Acceleration of Solution 2. *

*Only pixel results after L1 has been validated, the mean error (comparison to the non-quantized C software CNN execution) after converting to float is smaller than 3%.

5. Results and Conclusions

5.1 Acceleration ratios

One can clearly observe on Figure 23 that for the current acceleration results the 2D convolution is not a bottleneck any more. Thanks to multiple DSP utilization and smart pixel shifting logic the amount of time to compute convolutions and image accumulations results was reduced massively. Unfortunately, there is a max-pooling algorithm to be performed between L2 and L3 and between L4 and L5. This algorithm is not implemented in hardware, therefore all the images need to be transferred from BRAM to the operational memory, the algorithm is executed and the results is transferred back to BRAMs. The transfer between L2 and L3 is much bigger (and therefore much longer) because the image size there is 48x48 – after maxpoooling 24x24, transfer of 22 images. The transfer between L4 and L5 is much smaller – BRAM -> mem : 22 images, 24x24 , mem-> BRAM 22 img 6x6 each.

If we reduce the bottleneck of MaxPooling (implement it in hardware), the acceleration results can be obtained much faster, i.e. at least every ~3ms.

6. Challenges and Problems

6.1.1 Not Solved Problems

6.1.1.1 Pixel results of L2 .. L6 are not fully validated

Pixel values for L1 has been fully validated – the values in the memory after accelerating L1 with a test images are consistent with the siftare-only version of the CNN algorithm within 3% of error which is cause by computational limitatiopns of the DSPs. However, the pixel values for L2-L6 are not matching with the software execution of CNN algorithm, therefore the result of the classification using the accelerated design is wrong. Because of the limited timeframe of the semester project, the solution to this problem could not be found.

Pixel saving in L2-L6
Pixels in the memory are saved in the correct place, with respecting the zero-padding etc.

Wrong Pixel Values Debugging in L2-L6
Probably there is a mistake in the addressing state machine when iterating through the input images

- Very complicated design, debugging FPGAs
- Didn't manage to fix it within the given time frame,
- Would need about 2-4 more weeks of hardware debugging with test patterns.

6.1.2 Solved Challenges:

6.1.2.1 Analyzed the computational graph of the CNN.

The computations of the CNN has been fully understood in the means of single * and +. The difficulty was mainly to understand the high-level functions used in python / tensorflow and to map it into the C code executable efficiently on the embedded platform.

6.1.2.2 Profiled the CNN algorithm

After implementing fully functional classification algorithm in C, it was possible to precisely profile which operation was the bottlenech of the classification. That is why it was discovered that the 2D image convolution take more than 99% of the classification time.

6.1.2.3 Made the quantization analysis

Generated coefficients with different quantization levels and executed the algorithm with "flattening functions" to simulate the DSP behaviors. Then compared the generated classification errors in comparison to the "not quantized/flattened" coefficient solutions.

6.1.2.4 Reviewed different accelerator architectures.

During the project, multiple designs of convolution acceleration has been reviewed and the solutions appropriate to the platform, the CNN and the level of complexity for the semester project has been chosen.

6.1.2.5 Implemented and reviewed simple Solution 1

When implementing the simplified solution 1 in VHDL it was observed, that the design cannot efficiently accelerate the CNN very much because of the size of internal registers logic and prone utilization of DSPs. That initiated search for better acceleration architectures.

6.1.2.6 Implemented and reviewed Solution 2

The state-of-art solution 2 has been implemented for kernel size 8x8 and all the layers, which allowed to accelerate the CNN classification algorithm very efficiently by utilizing BRAMs and shifting the pixels in a very generic manner, therefore it could be implemented using much less logic cells of the FPGA than Solution 1 and so also more accumulator logic could fit into it.

6.1.2.7 Python Software to Generate the Constants in *.h file

After the CNN is trained in python, there is a need to transfer the trained CNN kernels and other coefficients (like B constants) and test faces into C code arrays. The python code has been prepared that could automatically convert tensor flow coefficients into C multi-dimensional arrays.

6.1.2.8 C libraries to transfer to / from 12-bit fixed point

To transfer the data to/from the BRAMs, functions has been written to transfer 12 bits in the proposed fixed point format from/ to double floating point.

6.1.2.9 C Libraries to control the accelerator

The functions has been written to reset, stop and check done flag of the accelerator (pins connected to different GPIOs).

6.1.2.10 C Libraries to transfer images to/from the BRAMs

The functions has been written to take the double image (array) and put it in the BRAM (align the 12 bit pixels etc.). The same the other way, the function to read a single image (return array of double) from the BRAM address has been written.

6.1.2.11 Summary of Code Written:

- ~800 lines of python for:
  - analysis
  - quantisation
- ~3000 lines of C/C++
  - BRAM control
  - accelerator control
  - image manipulation
  - profiling
- ~ 2500 lines of generic VHDL:
  - implementation of Solution 1 and test benches
  - implementation of Solution 2 and test benches

6.1.2.12 Pixel values for L1 fully validated

The values in the memory after accelerating L1 with a test images are consistent with the software-only version of the CNN algorithm within 3% of error (in comparison which is probably caused by computational limitations of the DSPs.

7. Next Steps

7.1 Make it fully work with the addressing.

The L2-L6 pixel results still have to be debugged and the result fully validated.

7.1 Implement Max Pooling in hardware

Currently, the main classification speed bottleneck is tr image transfers to and from the software to do the maxpooling. Since the maxpooling algorithm is not very complicated, it should be implemented in hardware and then the classification result can be reduced into ~2ms (the previous timings subtracted by internal transfer times for maxpooling).

7.2 Make 4x4 kernels pixel shifting logic

Currently the pixel shjifting logic is working only for 8x8 kernels and 4x4 kernels are padded by 0's and treated as 8x8 kernels. The number of computations required for computing 4x4 kernels is much smaller (single block result in 16 not 64 cycles) and therefore the obtained acceleration could be up to 8 times faster for L4-L6.

7.3 Adding non-equal fixed-point scheme for filter coeff. and the inter-layer images.

Since the kernel coefficients values are from -2.0 to 2.0, their fixed point scheme could be adjusted and more bits for fractional points could be used. For the images, to obtain better classification accuracy and not “flatten” the result in some cases, it’d be better to have more bits in the integer part of the image. Further simulations needs to be dine for that purpose.

7.4 Saving state machine can take less cycles

Currently the saving state machine used 6 clock cycles to save single 6x6 block result (Read -2clk and Write -1 1clk, for 2 addresses). This can be reduced to only 2 cycles by using byte-aligned write enable signal in BRAMs.

8. References

1. Maurice Peemen, et al *Memory-Centric Accelerator Design for Convolutional Neural Networks,* Department of Electrical Engineering, Eindhoven University of Technology
2. Yufei Ma et al., Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks, Arizona University.
3. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning