Implementation of a Convolutional Neural Network on an FPGA using Verilog

A Research Paper highlighting recent technologies and drawing comparisons to our own implementation

Ahmed Abdulkader	Daniel Eskandar	Omar Essam	Omar Tarek	Ahmed Ezzat
1170472	1170524	1162285	1170331	1162033

ABSTRACT

Implementing a multi-layer convolutional neural network (CNN) using hardware is a very difficult task. This is mainly due to the numerous tedious, abstract and highly conceptual procedures that go in the process of making the network function. Such difficulty calls for creative use of hardware resources to implement the network and dependence on algorithms that make the code more intuitive to be coded in a hardware description language. Several Architectures are implemented and tested, each with different perks over the others. We'll take a look at a few of those and compare them to our own implementation of this project.

Introduction

The project we implemented is a convolutional neural network of the leNet-5 architecture on an FPGA module. The hardware description language we used is Verilog. The data nature is of floating point numbers using the IEEE 754 standard, and with implementations of both half precision and full precision variants. The project consists of several layers to make the overall layered structure of the network. Each layer of hardware implements a different conceptual aspect of the network. The network could be thought of as two distinct layers. The first part of the network is the part dealing with convolution. This contains layer is responsible for convolution and average pooling. The other part is the fully connected section of the network. This part contains fully connected layers with activation layers between them. The goal of the network is to recognize a series of hand written images of numbers and to be able to classify those numbers into one of the 10 decimal digits.

Recent Attempts at FPGA models

Each member of our team chose a recent attempt from research papers most relevant to his scope of work:

Daniel Eskandar:

X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou and X. Ji, "High-Performance FPGA-Based CNN Accelerator with Block-Floating-Point Arithmetic," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874-1885, Aug. 2019, doi: 10.1109/TVLSI.2019.2913958.

M. Sreenivasulu and T. Meenpal, "Efficient Hardware Implementation of 2D Convolution on FPGA for Image Processing Application," *2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, Coimbatore, India, 2019, pp. 1-5, doi: 10.1109/ICECCT.2019.8869347.

Ahmed Ezzat:

B. Pasca and M. Langhammer, "Activation Function Architectures for FPGAs," *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Dublin, 2018, pp. 43-437, doi: 10.1109/FPL.2018.00015.

Omar Essam:

B. Yuan, "Efficient hardware architecture of SoftMax layer in deep neural network," *2016 29th IEEE International System-on-Chip Conference (SOCC)*, Seattle, WA, 2016, pp. 323-326, doi: 10.1109/SOCC.2016.7905501.

Ahmed Abdulkader:

S. Saves, E. Hertz, T. Nordström and Z. Ul-Abdin, "Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis," *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Bochum, 2017, pp. 110-115, doi: 10.1109/ISVLSI.2017.28.

Omar Tarek:

A. Shawahna, S. M. Sait and A. El-Maleh, "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review," in IEEE Access, vol. 7, pp. 7823-7859, 2019.
doi: 10.1109/ACCESS.2018.2890150

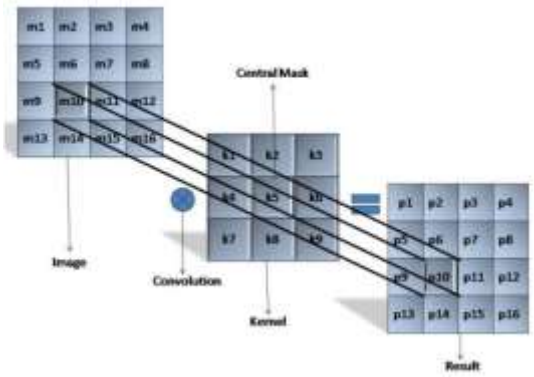
Literature Review of attempts

Daniel Eskandar:

Paper 1

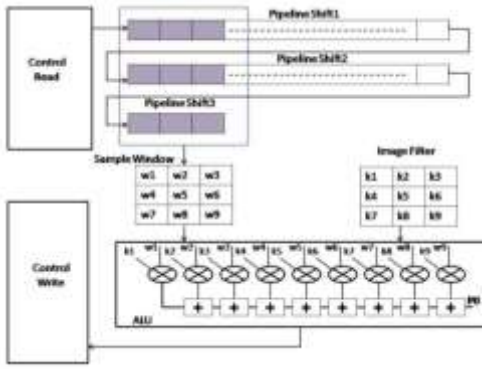
The purpose of this paper is to implement a 2D convolution technique to process an array of images with a reduced number of shift registers, multipliers, adders, and control blocks, thus leading to considerable hardware saving and fewer number LUTs. The design has been implemented and tested on Verilog Xilinx Spartan 3E FPGA. The convolution is used for 3 main applications: improvement of pictorial information, autonomous machine applications and efficient storage and transmission of pictures. The main objective of the paper is to achieve the best tradeoff between area, LUTs and speed.

This figure below explains the process of the convolution on pixel number 10 and the equation used to calculate the output pixels (The same procedure applies for the other pixels)



P10 = (m5 × k1) + (m6 × k2) + (m7 ×k3) + (m9 × k4) + (m10 × k5) + (m11 × k6) + (m13 × k7) + (m14 × k8) + (m15 × k9).

The following picture shows the design of the convolution unit used. It contains pipelined shift registers to take hold the rows of the matrix of the part of the image. They are shift register because they shift each row of matrix to pad the matrix with zeros if need. A control unit is responsible for handling the filling of the registers, the padding and sending the image part to the ALU that performs the convolution operation. The ALU takes the image part and the filter and calculates the result with a combinational logic to write the new pixels. The data width of the pixels is 8 bits in a fixed-point representation.



The results of this convolution are fast, the design occupies a very small area and use a very small number of LUTs. The utilization of flip flops is 4% and the utilization of 4 input LUTs is 3%.

Paper 2

This paper implements a complete CNN (Convolution Neural Network) using an optimized floating-point notation called the Block Floating Point (BFP). The feature maps and model parameters are represented in 16-bit and 8-bit formats which reduce chip utilization compared 32-bit FP counterparts. The design is implement using Verilog on Xilinx VC709 FPGA. The choice of using FPGAs to implement convolution operations has the advantage of connecting directly to image sampling devices. The main obstacle is the floating-point arithmetic overhead.

The paper explains the details of the implementation of different architectures of CNNs but in this literature review I will concentrate on the convolution part and the optimizations made in the floating-point notations to accelerate the convolution process while reducing the utilization and the area of the design.

The paper offers 3 different ways to implement the convolution entry wise, row wise and column wise. The implement design is the first one (the element wise multiplication and accumulation) because it is the method with less storage and the most common used method with floating-points.

The idea of BFP is the following: An N-data block represented with the BFP format consists of two parts: N mantissas and one exponent shared by the N numbers in a block. The process of the BFP conversion is defined as follows. Assuming that X is a data set containing N FP numbers, we can express the set as:

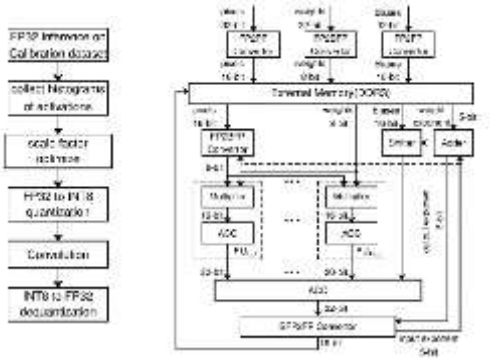
$\mathbf{X} = (x1, \ldots, xi, \ldots, xN) = (m1 \times 2\text{^}e1, \ldots, mi \times 2\text{^}ei, \ldots, mN \times 2\text{^}eN)$.

After deriving the common block exponent ϵX (maximum exponent), the mantissa number mi is right shifted by di bits,

where di = $\epsilon X - \epsilon I$. Thus, the BFP format of X, i.e., Xb is expressed as:

$\mathbf{X}b = (xb1, \ldots, xbi, \ldots, xbN) = (mb1, \ldots, mbi, \ldots, mbN) \times 2^{\epsilon X}$

The following block diagrams show that the 32-bit floating point data are gathered in block using BFP and transformed into smaller notations (8-bit and 16-bit notations) using converters. The multiplication and addition are performed on smaller numbers and the result of the convolution is the converted to 32-bit FP and stored in the memory. This not only accelerates the convolution process it only lowers the utilization of the chip.



The convolution results after this conversion has to be approximated using different methods. The best one with the least error is RN (round to nearest) and the error is not more than 0.12%.

Ahmed Ezzat:

SP-> Single precision
FP-> full precision
DSP -> digital signal processing
In this Paper, Using Hard FP DSP Blocks:
It is implemented on the interval [0, ≈ 3.81].For values larger than ≈ 3.81, return 1. For Values smaller than 2^-5, return the input. The results for the negative input range are constructed by appending the input sign to the result because the tanh is a symmetric function tanh (-x) =-tanh(x).
For Half-Precision:
The input interval, extended to [0, 4) is split into 64 subintervals. On each subinterval a degree 1 polynomial having single-precision coefficients is used to approximate tanh. It can be observed that the approximation error is significantly better as we approach the right of the approximation interval. The input interval, extended to [0, 4) is split into 64 subintervals. On each subinterval a degree 1 polynomial having single-precision coefficients is used to approximate tanh. The approximation error is plotted using solid lines in Figure 1. It can be observed that the approximation error is significantly better as we approach the right of the approximation interval.

We can reduce the number of stored coefficients by using a non-uniform segmentation scheme; as long as the complexity associated with the interval decoding is

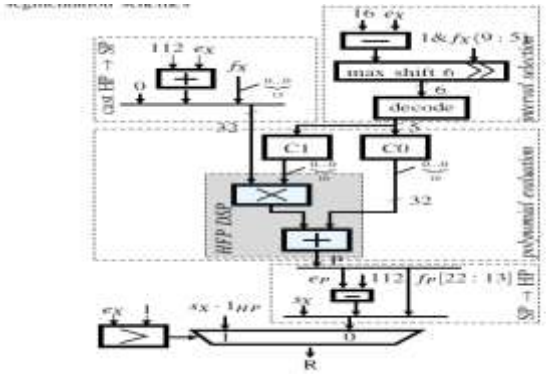


Figure 1 HP tanh(x) architecture using Hard FP DSP Blocks

low, this scheme can reduce ALM count. In Figure 1 the approximation error and the interval sizes for a non-uniform segmentation scheme are plotted. It can be observed that the number of subintervals used on the right side of the input interval is reduced. Overall, the number of subintervals is decreased to 30; the cost of decoding the subinterval index is a 5-bit LUT6 table lookup.
The architecture is depicted in Figure 5. A small barrel shifter is used to create a 6-bit fixed-point value that identifies 64-subintervals. The decode table maps the 6-bit fixedpoint to a 5-bit fixed-point value that identifies the table address storing the coefficients for the non-uniform intervals.
The HP input is cast to SP by adjusting its exponent

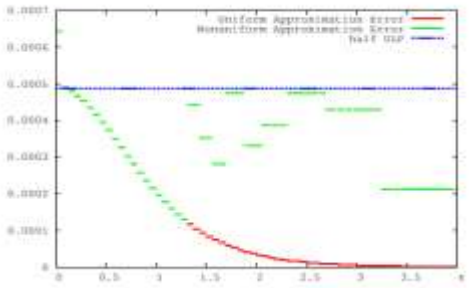


Figure 2 HP tanh(x) approximation error for uniform and non-uniform

bias offset (adding 127-15) and padding the fraction with 13 zeros. The coefficient tables only output a 13-bit fraction. Consequently, at the output of these tables the coefficients have their fractions zero extended to the right before feeding these into the polynomial evaluator. The evaluation is then mapped directly on the Hard FP DSP Block Mult-Add pair.
The output of the evaluation is in SP and needs to be cast back to HP. This is done by adjusting the exponent

(subtracting the bias offset 112) and truncating the fraction to 10 bits. The accuracy can be improved at this stage by rounding the SP mantissa to the nearest HP mantissa, which may involve an exponent update. The input sign is then added to this half-precision output value. Finally, a multiplexer selects between the implementation branches. For exponent values larger or equal to 2 we return sx ×1.0. The branch that returns x if input < 2−5 is fused with the polynomial approximation; one extra set of polynomials (C1=1, C0=0) is stored for the case the 6-bit input is all zeros.

Single-Precision:
The SP architecture is centered on a degree-5 odd FP PPA. For inputs larger than 8 we approximate the function with the value 1. For inputs smaller than 2−12 we return the input. The degree-5 polynomial is evaluated as follows:

$$P(x) = x(c_1 + x^2(c_3 + x^2 c_5))$$

For a sufficiently accurate approximation, the interval [0, 8)
is split in 512 subintervals. The approximation error is depicted
in Figure 4 for segmentations with both 512 and 256 subintervals. The maximum approximation error is between 1/4ulp and 1/8ulp for 512 subintervals. For 256 subintervals the maximum approximation error is approximatively 1.5ulp.

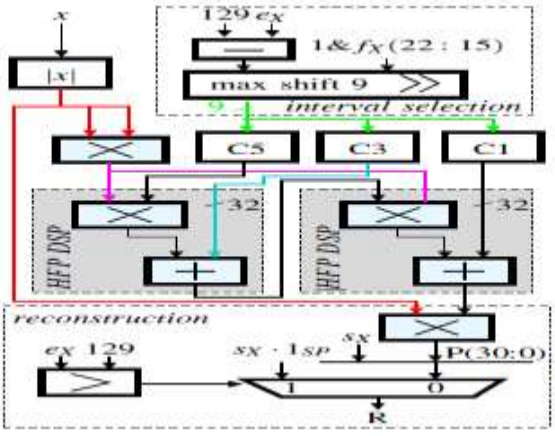
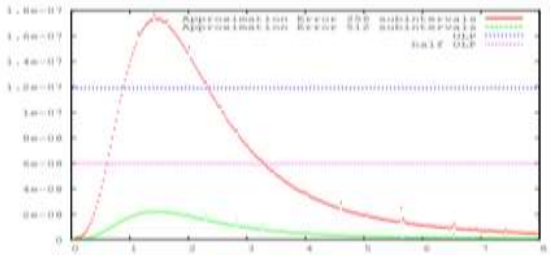
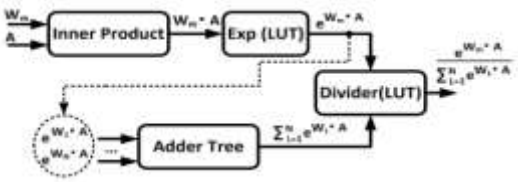


Figure 3 Single-precision tanh(x) architecture using Hard FP DSP Blocks



Omar Essam:
The Softmax layer is an computationally expensive layer as it uses exponent and division modules which make the design complex , and make it suffer from critical path delay and overflow problem , this paper discusses an efficient approach to fix the problems mentioned, and decrease the complexity and critical path delay.
The straightforward design which is the design used in our implementation, consists of the exponent blocks , addition block to get the sum of the exponents, Division block to get the reciprocal of the sum using Newton-Raphson, and multiplication block to get the conditional probability of each class.

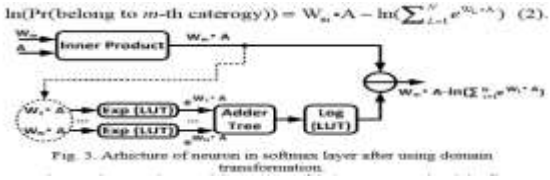


The straightforward design has these disadvantages

1. Division Problem:
    a. Division used a big LUT and uses a lot of hardware.
    b. Sensitive to accuracy of denominator.
    c. It has long critical path.
2. Overflow Problem:
    a. the values of WL*A which we get the exponent to can be very large which can ` cause an overflow problem.

Proposed Design:
In the proposed design there is no division to avoid the problems discussed in the straightforward. So Domain Transformation is used to avoid the complex division part

Which can be achieved the log transformation as in the figure below.



So as shown in the figure the division block will be replaced with a subtraction which will save hardware and avoid long critical path.

So the other problem to fix is the overflow problem , so to avoid this problem we use Down-scaling Exponentiation so will define (W•A)max=max(W1•A, W2•A, …, WN•A), and the equations will be transformed to

$$\Pr(\text{belong to } m\text{-th caterogy}) = \frac{e^{W_m \cdot A}}{\sum_{L=1}^{N} e^{W_L \cdot A}}$$

$$= \frac{e^{W_m \cdot A} / e^{(W \cdot A)_{max}}}{(\sum_{L=1}^{N} e^{W_L \cdot A}) / e^{(W \cdot A)_{max}}}$$

$$= \frac{e^{W_m \cdot A - (W \cdot A)_{max}}}{(\sum_{L=1}^{N} e^{W_L \cdot A - (W \cdot A)_{max}})}$$

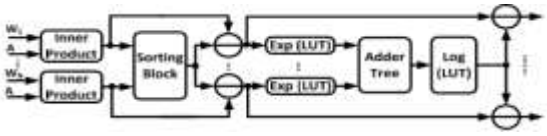
And the logarithm equation will be transformed to

$$= W_m \cdot A - \ln(\sum_{L=1}^{N} e^{W_L \cdot A})$$

$$= (W_m \cdot A - (W \cdot A)_{max}) - \ln(\sum_{L=1}^{N} e^{W_L \cdot A - (W \cdot A)_{max}})$$

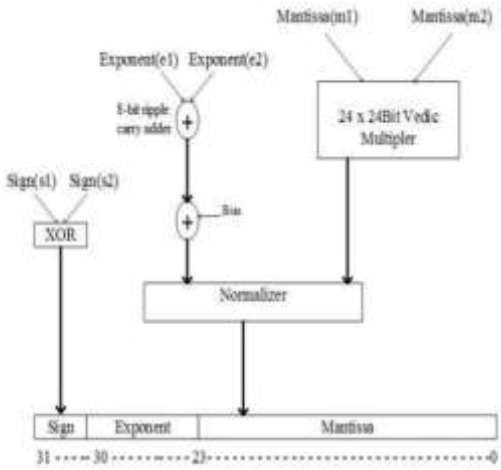
So to get the maximum value (W.A)max we need to have a sorting block to be able to compare all values and get the max value.

So the overall architecture will be as shown.



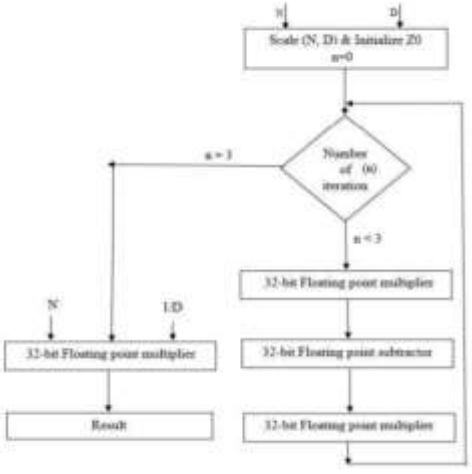
Ahmed Abdulkader:

There are a lot of ways to handle floating point arithmetic. The easiest ways are the float adders and multipliers. We implemented a very simple float adder and multiplier that performs hand like calculations. The paper I chose does something really similar.



It is a very simple procedure of adding exponents and multiplying and shifting mantissas. The sign bit is a simple XOR function on the two input signs.

These simple arithmetic processes pose no problem and can easily be ported from 32 to 16 and back. The difficulty rises from division. While in my implementation of average pooling I opted for the simple solution of simply multiplying my number by the reciprocal of the divisor, the paper uses Newton-Raphson's algorithm to handle the division part. The algorithm calculates the multiplicative inverse, the reciprocal and carries out a multiplication procedure. It is an iterative method where carrying out multiple iteration grants you a more accurate result. The flowchart used us shown below, it utilizes two multipliers and one adder configured to subtraction:



The algorithm above proceeds to calculate the reciprocal by the following formula,

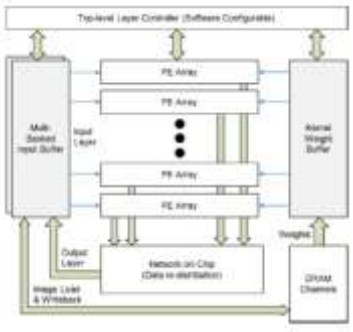
$$Z_{i+1} = Z_i + Z_i (1 - DZ_i) = Z_i (2 - DZ_i)$$

Where Zi is the inverse at a given iteration Z. In cases where division is done by a constant, as in my module of average pooling, a much cleaner, faster and efficient approach is multiplying by a constant.

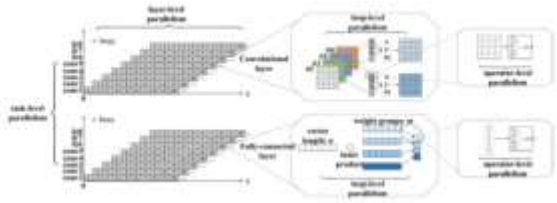
Omar Tarek:

The paper reviews the very basics of neural networks. The main highlight is of course the convolutional neural network which holds great advantages against other types. It covers the fundamental blocks of the structure of said networks. The main components being convolution, activation functions, normalization, pooling and the fully connected layer. It also goes on to discuss various different architectures like AlexNet and VGG, all of which where developed in this decade. Next it summarizes the basic functionalities of the FPGA system, talking about advantages and drawbacks of using it to implement accelerated neural network architectures.

The main focus of the last part of the paper is CNN acceleration, using several techniques like reduction etc. CNN compression is a very obvious candidate to achieve such acceleration, however it is not the most effective. Then a few more effective solution based on ASIC and FPGA technologies are proposed. Several accelerators are then highlighted, and each is discussed on its own merits like for example the Microsoft CNN accelerator:



Other more impressive and advanced structures are also discussed, achieving high levels of speed. This is done using extreme levels of parallelism, which as we found out was very hardware extensive. One example of such architecture is the following:



Showing parallelism at not 1 or 2, but rather 4 levels. Where each module performs parallel calculations on massive arrays of data.

It is evidently clear that these advancements in machine learning are only beginning and more creative implementations will make way to faster more efficient models in the future.

Overview of our chosen CNN

The CNN we implemented models a light version of the leNet 5 architecture developed by Yann LeCun in the 1990's. The architecture consists of several layers each having specific function. The first part is the convolution part mainly utilizing the convolution layers and the average pool.

The network has 3 convolution layers, each with different filters, all of Kernel Size 5x5. Details of implementation would be found at the individual report by Daniel. Several architectures were attempted and implemented by Daniel, in an attempt to reach a compromise between Parrallelism, Speed and Utilization. The trade off here is more parallelism requires higher power and is more hardware extensive. Less parrallelism causes a decrease in speed. This balance is very tricky in convolution due to the huge sizes of Data required for this complicated function. A compromise was reached and a comparison between architectures is outlined in Daniel's individual report.

The network has 2 average pooling layers. These decrease the size of the output data from the first 2 convolution layers. While fairly easy to implement, they suffer from the same Data Size issue found in convolution layers (part 1). These issues cause synthesis and utilization problems, leading us to use compromises that may slow down the process but improve hardware utilization significantly. The other drawback of the large data size, is the inability of producing proper post synthesis timing simulations. This is an issue that was agreed on with faculty staff and we reached a compromise of simulating the smallest processing element of each matrix.

The activation function selected between the layers and each another is the Hyperbolic Tangent Function. More details on structure and testing are attached in the individual report prepared by Ahmed Ezzat. The function deals with a large number of inputs, same issue as convolution. During the convolution part of the integration, the TanH function is of size 16, and at the fully connected parts it's at size 32. The convergance condition and the limiting range of 16 bit half precision floating point numbers, and also the fact that this is implemented using the taylor approximation of the function, make this a bottleneck of accuracy. The accuracy is within the accepted range though.

The last layer in the leNet-5 architecture is a SoftMax activation layer. This special activation layer takes the 10 produced values and through a series of calculations produces the final classification of our neural network. The series of calculations mentioned, is an approximation of a function involving division of exponents. Implementations of hardware division of several floating point numbers in Verliog is exceptionally difficult, which led Omar Essam to develop multiple architectures, and choosing the best compromise. Difference between the architectures and the merits of each are discussed in detail in Omar Essam's individual report.

Arguably the hardest, least synthesizable and largest part of this project is making sure that the network integrates properly. To make this happen collaboration between the member responsible for integration, Omar Tarek, and the rest of the team was essential. To also make the testing easier, testing on the convolution units and the average pool units was done consecutively with one input number image and plugging the output data of the previous layer into the next layer. More on this could be found on the testing section of the group report. Due to the sheer size of this network synthesis was not attempted due to hardware limits. More details are in the individual report.

Verilog Modules Used

Daniel Eskandar:

Section 7: Developed Modules and Architectures for Convolution

Module Name	Function
floatMult.v	Takes 2 FP numbers and performs multiplication
floatAdd.v	Takes 2 FP numbers and performs addition
processingElement.v	Takes 2 FP numbers multiplies them and accumulates on the result (MAC unit)
convUnit.v	Takes the image part and the filter and performs the convolution operation (elementwise multiplication and accumulation)
RFselector.v	Takes the image, a row number and a column and returns the matrices of the parts of the images to be sent to the conv units in the single layer module
convLayerSingle.v	Takes the image and a filter and calculates the output image (makes instances of convUnit.v and and instance of RFselector.v)
convLayerMulti.v	Takes the image and different filters and calculates the output of the convolution layer (makes two instances of convLayerSingle.v and calculates the output images of the convolution process sequentially: 2 by 2 filters)

Architectures:

Architecture 1: convolution (32-bit FP precision) implemented with full parallelism (all pixels of the output image)

Architecture 2: convolution (32-bit FP precision) with parallelism of number of conv units equal to number of the pixels of a single row

in the output image (28 for layer 1). The output is filled sequentially.

Architecture 3: architecture 2 (32-bit FP precision) with parts of the images calculated sequentially only for the used conv units and not for the whole picture

Architecture 4: convolution (32-bit FP precision) with parallelism of number of conv units equal to half of the pixels of a single row in the output image (14 for layer 1). The output is filled sequentially. And The receptive field array is optimized.

Architecture 5: architecture 4 with half floating precision (16 bits).

Comparison:

	Architecture 1	Architecture 2	Architecture 3	Architecture 4	Architecture 5
Floating Point Precision	32 bits	32 bits	32 bits	32 bits	16 bits
Number of clock cycles needed to output the result	26	728	728	1456	1456
Number of LUTS	1712256	177300	135176	117521	48422
Utilization on the KCU	707.54%	73.26%	55.86%	48.48%	19.98%
Utilization on the ZCU	738.04%	77.09%	58.77%	51.10%	21.05%

Architecture 5 is implemented in the integration of the whole CNN.

Ahmed Ezzat:

The Developed modules:

1. HyperBolicTangent which is the processing unit of the layer
2. UsingTheTanh which is the layer of the activation function
3. floatAdd
4. floatMult

In the hyperbolic tangent module, the taylor expansion was used to approximate the tanh of any number in the convergence region which |x|>pi/2.

Exactly 4 terms were used to implement this approximation because this produces the optimum curve for the average error and as the number of terms increases the average error increases and this was by the help of a python script to test this graph and show it.

Now regarding the Using the tanh module was mainly based on the hyperbolic tangent processing unit which acts like a sliding window on each array entering it. It processes the input as 32 bits respectively and produces the output based on the rules given to the hyperbolic tangent module.

The main rules where:

1. if the input is more or less than pi/2 : it is given an immediate value of 1
2. if the input is inside this region : it is processed and given a its calculated input
3. I added a special case that if the input is exactly equal to 1.57 which is the edge of the conversion: I assigned also an immediate value to it.

Omar Essam:

The Verilog modules used to develop the softmax module was

a. Softmax
b. Softmax_1
c. Softmax_2
d. Exponent
e. floatAdd
f. floatMult
g. floatReciprocal

In the exponent module the Taylor approximation was used to get an approximation the exponent, it was done sequentially, it takes 7 clock cycles to output the exponent.

floatReciprocal used the Newton Raphson algorithm to get the mantissa of the reciprocal of the number , then the exponent is 253-exponent of the number , and the sign is the same sign of the number.

Softmax is the first design

Design Description: 10 exponent units, 1 floating adder , 1 multiplier, 1 division unit.

Clock Cycles: 31 clock cycles as shown below. ackSoft is triggered

Description : 10 parallel exponent units to calculate exponent of 10 inputs, then 1 floating adder will used with 10 clock cycles to calculate their sum and send an enable signal to Newton Raphson module which will calculate the Reciprocal after the Newton Raphson module sends the ackDiv signal , the 1 multiplier will be used for 10 clock cycles to multiply the 10 exponents with the 10 reciprocals.

Softmax_1 is the second design

Design Description: 10 exponent units, 10 floating adder , 1 multiplier, 1 division unit.

Clock Cycles: 22 clock cycles as shown below, ackSoft is triggered

Description: same as Design 1 the only difference is 10 adders are generated to calculate the sum in one cycle.

Softmax_2 is the third design

Design Description: 10 exponent units, 10 floating adder , 1 multiplier, 1 division unit.

Clock Cycles: 22 clock cycles as shown below, ackSoft is triggered

Description: same as Design 1 the only difference is 10 adders are generated to calculate the sum in one cycle.

Ahmed Abdulkader:

The Verilog modules I uniquely developed are:

AvgUnit- The simple averaging unit for 2x2

AvgLayerSingle- A layer handling a 2d input

AvgLayerMulti- A layer handling 3d input

The averaging unit takes in 4 floating point numbers and calculates the average by adding them and multiplying by a quarter. This is done combinationally which saves a lot of time.

The single Layer handles input as a 2d layer that passes 2x2 matrices with a stride of 2 to the averaging units in the layer.All calculation is done in parallel, and so the output comes out in 0 clock cycles, the circuit is combinational.

The Multiple layer average pool passes all the distinct layers in the depth of a 3d input sequentially to the single layer. It has only one single processing module. It executes in 6 clock cylcles or 16, depending on the depth of the input.

Omar Tarek:

The two modules developed were Integration.v

Which contains IntegrationFCpart,v and IntegrationConvPart.v and a 16 to 32 bit converter to convert the output. Pipeline was not attempted. All modules from before are included to make the network functional.

Comparison with papers

Daniel Eskandar:

The first paper implements the convolution operation using a combination circuit on fixed point numbers. That is why the design of the ALU is simple. In our CNN the number are processed in 16-bit floating-point representation which make the design of the multiplier and the adder more complex. And the result of the convolution is calculated sequentially which lowers the utilization. The utilization of convolution unit in my design is 0.27% which is a very small number taking into consideration that the floating-point arithmetic is more complex. However, in the paper, the padding of the image is done by the shift register in the design of paper 1 which is a good and simple idea if we want to implement padding in Verilog. In our CNN the input image is already padded.

The second paper proposes a solution to the complication of the arithmetic floating-point logic by implementing converters. We implemented a similar but simplified reasoning: I changed the adder and multiplier to operate on half precision floating point (16 bits). The problem of accuracy was also resolved by the

same method: since the floating-point method only expresses a finite amount of numbers, we added conditions on the adder and multiplier to approximated the numbers close to zero which one of the rounding solutions implemented in the paper.

Ahmed Ezzat:

Comparison between me and the research paper:
My design works according to Taylor approximation and I work on floating number either they are positive or negative and they converge after a certain range not like the paper which mainly works on specifying the ranges of inputs according to groups and each group is approximated to a certain number and if the input is big so 1 is returned and it is less than 2^-5, the input is returned back.
The paper's design mainly depends on low latency and high efficiency activation functions but my design has high latency depending on the number of terms I chose and also good precision.
The paper's tangent function takes an input either SP or FP. My design can take any number of inputs given to it and only the latency is the drawback in this design.

Omar Essam:

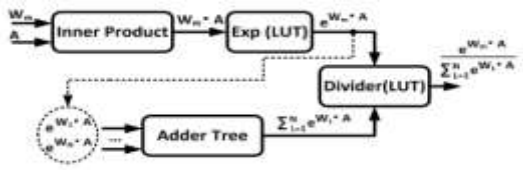
As Mentioned in the Section 4,5 of Softmax that the straightforward approach that we used of softmax had two issues, the division problem which uses a lot of hardware and takes a lot of time, and the overflow problem which could occur from multiplication and in the exponent process.

These issues are solved in the paper reviewed in section 4,5 where Domain transformation is used to avoid using division but use logarithm and subtraction instead which solved the division problem.

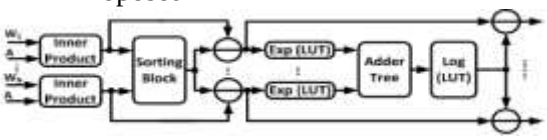
The other problem was the overflow problem which was solved by applying Down-scaling Exponentiation which will make sure overflow does not occur, we need the max power of exponent to do this so we use a sorting module to be able to get the max power of exponent (W•A)max and the equations will change accordingly as discussed before.

Design comparison

Straightforward



Proposed



Ahmed Abdulkader:

Most implementations use MAX pooling as a more efficient and easier alternative to AVG pooling, with some evidence pointing towards it yielding even better results. Division is also non parameterized meaning that the implementation works only for 2x2 areas. This is effective for the purposes of lenet-5 but for a more modular network, division modules like the one mentioned in the paper will have to be used.

Omar Tarek:

The paper illustrates many different advanced networks, which have better performance than our implementation. The Network we used is effective for a single purpose, is fairly slow and isn't very modular. Accelerators like the ones discussed in the paper could be effective in speeding up the data flow, but larger amounts of multi-channel data must be fed into the network for it to have any real life purpose. FPGA's might seem like a very inefficient option to create a machine learning model, however with proper acceleration and application, an embedded system employing an already taught network could have very powerful applications. That being said, leNet still remains one of the most fundamental models of convolutional neural network in our modern day.

Unique aspects

Daniel Eskandar:
My convolution design is the result of evolution of many architectures. The final architecture presents a **balance between the speed** of the convolution network **and the utilization** of the FPGA. A high amount of parallelism is implemented with sequential filling of the output. The convolution design **can be pipeline**d since leaving extra clock cycles without resetting after finishing the operation maintains the right results. **The feature maps of the second convolution layers can be implemented** like LeNet table by replacing some depths of the input filters with zeros.
The design performs the convolution operation with 16-bits floating point **with a high accuracy** due to the approximations and the added conditions implemented in the adder and the multiplier.

Ahmed Ezzat:
The unique about my work I think is that low utilization is achieved and not so many clock cycles to get an output and even on very high number of pixels I can achieve only 41% utilization on 32 bits –precision and this utilization is decreased to more than half and also number of cycles decrease if 16 bits –precision is used, so I think this is good approach for the tanh function

Omar Essam:
The process taken to reach the final design showed incremental progress at every step. The SoftMax has a very efficient division system and calculates values to a very high degree of accuracy, relative to floating point arithmetic. This makes my implementation unique to other modules in the project.

Ahmed Abdulkader:
What makes my implementation unique is the speed at which the processing happens. The data is ready combinationally, with propagation delay being the only thing preventing it from being instantaneous. This speed improvement contrasts the other layers, making my module the fastest processing and most parallelized.

Omar Tarek:
A clear unique factor of our network is the implementation of heterogenous precision. This enable utilization to be effective for the huge data layers while making the smaller processing elements effective at a smaller level. This adds a whole new level of optimization to our CNN.