Accelerating CNNs on FPGA with Fast Convolution Algorithms: PYNQ-Z2 Implementation and Optimizations

Christian Alexander Pesch , Utku Saglam , Nisse Degelin

Abstract

In this project paper, we propose an accelerated Convolutional Neural Network (CNN) architecture designed specifically for the FPGA-based PYNQ-Z2 development board. We explore the efficacy of fast convolutional operators, such as the Winograd and FFT convolutions. Since the complexities of these algorithms increase the difficulty of practical FPGA implementations, we focus on the traditional convolutional operator. The results of our work are highly promising, as we achieve a remarkable 2.7-fold overall speed-up by applying our optimizations to the CNN. Additionally, we suggest optimizations to further enhance efficiency. Our research demonstrates the potential of FPGA-accelerated deep learning, since the traditional convolutional operator on the FPGA is only slightly slower than fast convolutions such as Winograd executed on a CPU.

1 Introduction

Convolutional neural networks (CNNs) have become widely adopted for various applications, being the primary layer in successful networks such as GoogleNet [Szegedy *et al.*, 2015]. In a CNN, convolution layers play a central role, accounting for a significant portion of computational demands [Wang *et al.*, 2017]. However, the conventional convolutional algorithm's computational intensity hinders overall efficiency, necessitating the exploration of alternative approaches. This study aims to accelerate CNN performance by employing fast convolutional operators in conjunction with an FPGA-based implementation. Source code is freely available.[1]

The paper is structured as follows. We begin by comparing the algorithmic complexity of two alternative convolutional operators against the traditional convolution. Subsequently, the performance of these algorithms is benchmarked on the CPU of the PYNQ-Z2 board. Then, we extend our investigation to leverage the board's FPGA capabilities. After detailing our proposed architecture, we conduct tests on each layer of a compact CNN to determine the most efficient configuration. This is followed by insights into future optimizations and a conclusion.

[1] https://github.com/Accelerating-CNN/ConvAcc

2 The Fast Fourier Transform Convolution

2.1 Background

The Fast Fourier Transformation (FFT) revolutionized the field of signal processing by enabling the efficient computation of the Discrete Fourier Transformation. Developed by Cooley and Tukey in 1965, the FFT algorithm significantly reduces the computational complexity from

$$O(n^2) \rightarrow O(n \log n) \tag{1}$$

making it indispensable for various applications like image processing, audio compression, and scientific simulations. Its remarkable speed and versatility continue to impact diverse domains to this day. In this work, we utilized the FFT algorithm to efficiently perform convolution. This enabled us to efficiently process and analyze data by leveraging the computational benefits of the FFT.

2.2 Implementation

We have employed two implementations of the FFT algorithm: a 1-D FFT for processing one-dimensional data, and a 2-D FFT for analyzing two-dimensional data.

2.3 1-D FFT

The implemented FFT algorithm follows a recursive approach. It begins by checking if the input size, N, is equal to 1. If this condition is met, the algorithm assigns the input value to the corresponding output and halts. Otherwise, it proceeds to split the input into even and odd components. To store these components, the code dynamically allocates memory for temporary arrays. Subsequently, the algorithm recursively calls the FFT function on these subarrays, computing the corresponding Fourier transforms, Xeven and Xodd. This recursive process continues until the base case (input size equals 1) is reached. Next, a loop applies the butterfly operation, combining the even and odd components using complex twiddle factors. The resulting values are stored in the output array, Xout, with appropriate indexing for the second half of the spectrum.

Finally, the dynamically allocated memory is deallocated using the delete[] operator to avoid memory leaks. The code efficiently computes the FFT by recursively dividing the input into smaller subproblems. However, it is important to ensure proper memory management to avoid memory leaks and potential performance issues. Also, Inverse FFT implementation is performed using FFT.

2.4 2-D FFT

To implement the 2D-FFT, we followed a specific procedure. First, we perform the FFT on each row of the input matrix, storing the results in the same matrix. Then, we transpose the matrix. Next, we apply the FFT on each row of the transposed matrix. Finally, we transpose the resulting matrix again and store it as the output matrix, representing the 2D-FFT of the input data. This process effectively computes the 2D-FFT by transforming both the rows and columns of the input matrix. To implement the Inverse 2-D FFT, we reverse the steps of the forward transformation.

2.5 FFT for Convolutions

To perform Convolution using the FFT algorithm, we leverage the properties of the Fourier transform. The process involves three main steps. First, we transform the input signals into the frequency domain using the FFT. Then, we multiply the corresponding frequency components of the transformed signals. Lastly, we apply the inverse FFT to obtain the convolution result in the time domain. This approach significantly reduces the computational complexity compared to traditional convolution methods.

When performing convolution using the FFT algorithm, one common issue is **aliasing**. Since the FFT assumes periodicity, the convolution result may exhibit circular artifacts. To address this, padding can be applied to the input signals before the FFT to avoid wrapping around. Using **overlap-add** or **overlap-save** methods, where the signals are segmented into smaller chunks, convolved, and then combined with appropriate overlap to solve the circular artifacts.

In our implementation, we used overlap-add method to prevent mentioned problems.

2.6 Weight Transformation

Before applying FFT to the input data, we performed FFT on the weights to utilize them as precomputed values. Given the tile size, we used it to address the aforementioned issues of overlapping and circular artifacts. However, before applying the FFT to the weights, we flipped them because of incorrect storage. By applying the FFT to the weights and storing the transformed values, we were able to utilize them later during the convolution process.

This additional step of performing FFT on the weights allowed us to leverage the efficiency of the FFT algorithm and optimize the convolution process, contributing to improved performance and accuracy in our implementation.

2.7 FFT for Convolution

We implemented a **convFTT** function to perform teh FFT convolution. The convFFT function performs convolution using the FFT algorithm. It takes as input the tensors **X** (input data), **Ufft** (FFT-transformed weights), **B** (bias), and **Z** (output tensor). The variable ksize represents the kernel size.
First, the function retrieves the dimensions of the input and weight tensors. It also obtains parameters such as tile size, overlap, and stride from the **FFTSTRUCT**. The size of the output tensor **Z** is determined.
To address overlapping, the input tensor **X** undergoes padding if its width is less than the tile size. Otherwise, **X** is padded to accommodate the weights' dimensions. The padded width and height are then calculated. A temporary **CTensor** array, **temp**, is created to store intermediate FFT results. Its size is determined based on the number of weight channels and the tile size.
For each weight channel in Z, the function iterates over the weight channels in **Ufft**. It then slides through the input tensor X in tile-sized increments, creating temporary tensors for each tile. These tensors are transformed using the **fft2d** function and multiplied element-wise with the corresponding **Ufft** values.
The resulting tensors are then added to the **temp** array according to the number of times the tiles have been processed. After all the tiles have been processed, an inverse FFT **ifft2d** is applied to each tensor in **temp**. The discard region is excluded, and the values are stored back into **temp**. The number of tiles in a row is calculated, and for each tile in **temp**, the values are extracted and stored in the output tensor **Z**. The bias **B** is added to the appropriate locations in **Z**. Finally, memory allocated for **temp** and **Xpadded** is released.
This algorithm utilizes the FFT algorithm to efficiently perform convolution and incorporates padding, sliding, and inverse transformations to handle overlapping and obtain the final convolution result in the output tensor **Z**.

3 Winograd Convolution

3.1 Weight transformation for Winograd

The provided weight matrix needs to be transformed to be used for the Winograd convolution. In this case, g is our weight matrix.

$$(GgG^T)$$

The (transposed) matrix G is created with genWino.py beforehand and stored afterwards. To calculate this multiplication we use a temporary variable in which we store the value of the first multiplication of G and g. In the next step, we multiply the product of it with the transposed G.

3.2 Winograd Convolution

The entire equation for the two-dimensional Winograd transformation is:

$$Y = A^T \left[(GgG^T) \otimes (B^T dB) \right] A$$

The (transposed) A matrix and the (transposed) B matrix are generated and stored using the same file beforehand. We use the winoWeights function to pre-calculate the converted weight matrix. In the next step we loop through the input feature map with tile stride. The input tile size will be separately computed using

$$(B^T dB)$$

and then element-wise multiplied with the weight matrix. The product m of this will be converted back with the equation

$$A^T mA$$

which can be done by a function that basically looks like the weight transformation, but gets m as an input. After that, we use two for loops to generate the output tensor. When we look at the memory requirements we can see, that Winograd needs less memory than for example FFT, but more than a standard convolution. The tile sizes are defined in the winograd.h header file and are connected to the kernel sizes.

4 Fast Convolutions on a PYNQ-Z2 CPU

Next to a conventional PC, accuracy and speed experiments were run on a PYNQ-Z2 Zynq 7000 development board equipped with a Dual-Core ARM Cortex-A9 processor and 512 MB DDR memory. All the runtime code is written in c++.

4.1 Performance for four CNNs

For this experiment, the three convolutional operators (basic, FFT, Winograd) are used to compute the convolutional layer for four CNN's. The four networks differ not only in the used input, kernel and output dimensions, but also in the number of layers (giant net having the most layers). To see the exact dimensions per layer, the reader is kindly referred to figure 18 in appendix 11.1. Each network takes as input a 3-channel 128 x 128 px RGB image and classifies this into one of 100 different classes. At this point, they are not pre-trained and can't predict anything yet. The results are plotted in figure 1 for a PC. It demonstrates that the Winograd convolution requires the least amount of time for each network, but as table 3 illustrates, it is not perfect in terms of accuracy for certain kernel sizes, what we are going to discuss in this chapter.

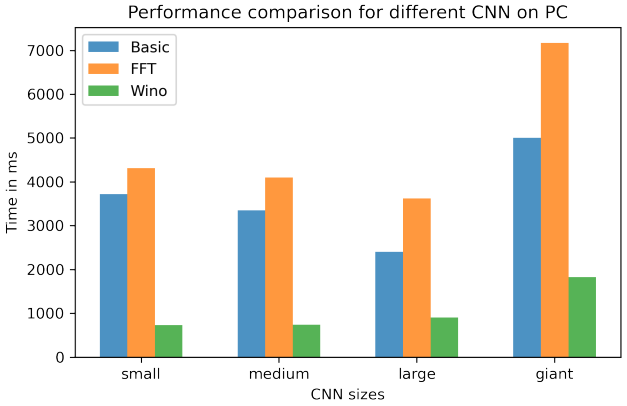

Figure 1: Comparison of the runtime for the three convolutional operators: the basic convolution abd the Winograd, and FFT convolution on a PC for four CNNs.

Figure 2 is almost identical, the only difference is a multiplication of the time with a factor due to the limited computational power of the PYNQ-Z2 CPU compared to the resources of a PC.

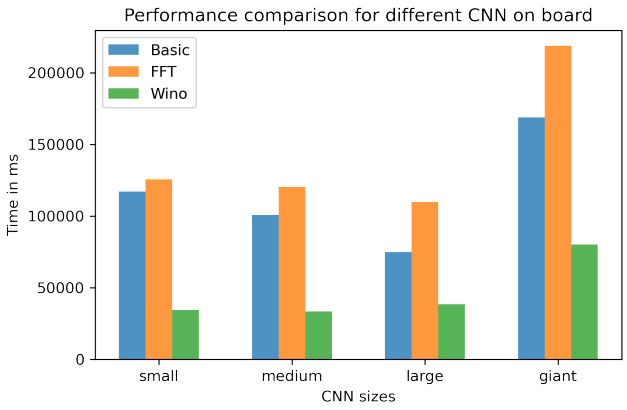

Figure 2: Comparison of the runtime for the three convolutional operators: the basic convolution abd the Winograd, and FFT convolution on the PYNQ-Z2 CPU for four CNNs.

4.2 Performance for each layer

The CNN's mentioned above apply various convolutional layers. An obvious, more fine grained experiment consists of comparing the time of the different algorithms for each individual layer. Figure 17 and 18 in appendix 11.1 display the time of each algorithm relative to the time needed by the basic convolution. The differences between the two figures in appendix make clear that the used hardware architecture is an important factor when comparing the performance. We will focus on the benchmarks run on the PYNQ-Z2 CPU.

Overall, there is only one layer where the Winograd convolution does not have the best performance, namely the first layer of the large CNN. It is also the layer with the smallest number of input channels (64). For this situation, the basic convolution outperforms the Winograd convolution. Also note the small difference in performance between the basic and Winograd convolution for the first and last layer of smallNet. The FFT convolution is faster then the basic convolution for one third of the layers, but never faster than the Winograd convolution. Since there is only one case where Winograd is not the fastest convolution, it is clear that, for our implementation, the best scheme for processing the given CNNs is Winograd. As such, after having explored fast convolutions, our first optimization is to apply Winograd in every layer.

4.3 Performance with varying in –and output factors

A similar experiment is carried out in figure 3 and 4, exploring the relationship between various factors that affect the processing time for a convolution. The average difference between the elements in the output tensor and the correct output tensor is next to the number of milliseconds.

X_channels	X_height	W_size	output channels	$Time_{Wino}$	$av,diff_{Wino}$	$Time_{Basic}$	$av,diff_{Basic}$	$Time_{FFT}$	$av,diff_{FFT}$
1	6	3	1	0.029	0	0.001	0	0.130	0
2	6	3	1	0.029	0	0.001	0	0.162	0
2	6	3	2	0.033	0	0.002	0	0.249	0
1	24	3	1	0.250	0	0.015	0	1.049	0
8	24	5	8	1.065	0	1.664	0	11.194	0
64	112	3	128	495.927	0	1491.963	0	2862.054	0
128	64	3	256	583.069	0	2373.930	0	2890.142	0
256	32	3	512	652.744	0	2360.208	0	4202.449	0
16	512	5	33	795.434	0	4242.394	0	5660.074	0
13	248	7	52	333.560	0	1432.854	0	2464.172	0
24	124	11	24	134.400	0.018	675.716	0	537.801	0
12	114	11	1	30.135	0.013	11.595	0	101.212	0

Figure 3: Comparison of the runtime and the average difference for the basic, Winograd, and FFT convolution on the PC.

For both the PC and the board, the Winograd convolution is faster for nearly all test cases, except for small in -and output sizes. For these cases, the basic convolution outperforms both FFT and Winograd. For larger kernels, the Winograd algorithm gets slightly worse in precision but is significantly faster then the two other convolutions. It is no coincidence that the Winograd precision decreases when the kernel sizes gets larger (7 and 11), since the Winograd transformation involves multiplication with values greatly varying in magnitude it results in numerical problems, e.g. numerical instability and over-/underflow. When the kernel size increases, the accuracy of this approximation decrease, leading to a loss of information and potential degradation in accuracy. In theory, in these cases, FFT should outperform Winograd, but it did not happen for our implementation for the four CNNs. The accuracy of FFT remains perfect, but the computional time is significantly worse compared to Winograd.

X_channels	X_height	W_size	output channels	$Time_{Wino}$	$av,diff_{Wino}$	$Time_{Basic}$	$av,diff_{Basic}$	$Time_{FFT}$	$av,diff_{FFT}$
1	6	3	1	0.17	0.00	0.01	0.00	0.63	0.00
2	6	3	1	0.16	0.00	0.01	0.00	0.65	0.00
2	6	3	2	0.22	0.00	0.01	0.00	1.00	0.00
1	24	3	1	1.89	0.00	0.07	0.00	4.28	0.00
8	24	5	8	11.44	0.00	7.87	0.00	86.02	0.00
64	112	3	128	12975.29	0.00	25499.50	0.00	37305.17	0.00
128	64	3	256	15439.76	0.00	48659.23	0.00	45546.41	0.00
256	32	3	512	15005.46	0.00	66713.12	0.00	60181.23	0.00
16	512	5	33	23168.10	0.00	93056.62	0.00	89987.59	0.00
13	248	7	52	10180.95	0.00	21915.73	0.00	40252.33	0.00
24	124	11	24	4879.80	0.02	13367.42	0.00	9788.97	0.00
12	114	11	1	854.30	0.01	173.83	0.00	1510.45	0.00

Figure 4: Comparison of the runtime and the average difference for the basic, Winograd and FFT convolution on the board.

On the board, FFT performs relatively better. It outperforms the basic convolution in most cases for sizeable inputs, while there are only two cases on the PC where FFT is faster then the basic convolution. Considering the less powerful hardware of the board, it is remarkable that the FFT and basic convolution sometimes actually perform better on the board in absolute figures.

4.4 Performance With Varying Kernel Sizes

The problem with the previous experiments is that multiple input/kernel/output dimensions are changing at the same time, making it hard to establish which dimensions are most significant for determining the computation speed. For that reason, figure 5 and 6 compare the time for calculating a convolution with identical in -and output dimensions but various kernel sizes on a PC and the board.

On both architectures, the Winograd convolutional algorithm is the fastest for each kernel size, although the graphs

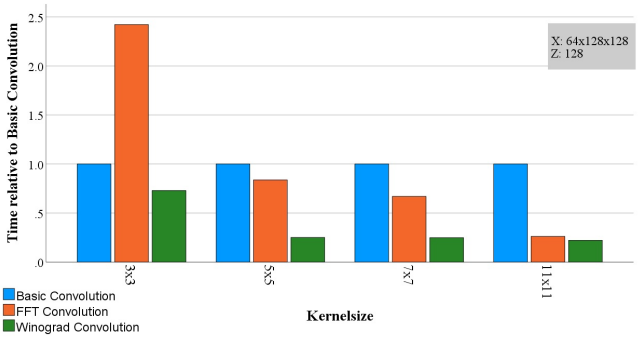

Figure 5: Comparison of the PC runtime for each convolutional operator for equal in -and output dimensions and four different kernel sizes. X stands for the input tensor, Z for the number of output channels.

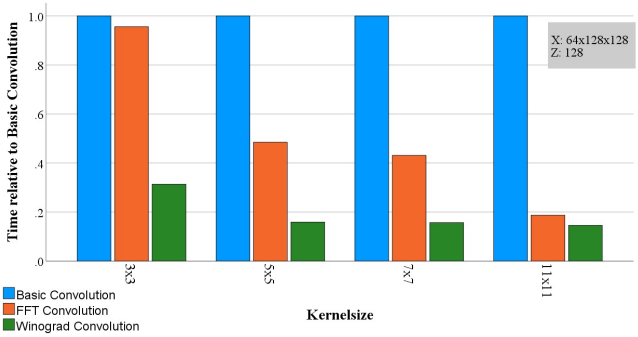

Figure 6: Comparison of the board runtime for each convolutional operator for equal in -and output dimensions and four different kernel sizes. X stands for the input tensor, Z for the number of output channels.

suggest that this isn't necessarily the case for kernels larger than eleven. Moreover, the gap between the basic (or traditional) convolution and the other two algorithms is widening if the kernel size grows.

The differences between a PC and the board are fairly small, with only one notable difference: the basic convolution outperforms the FFT convolution for a three-by-three kernel.

5 A Channel-wise Hardware Accelerator

Now we want to adept our software solution by adding a hardware part. In this section, our fixed-point accelerator based on a channel-by-channel algorithm is presented. This is followed by the design of the actual accelerator and overall performance tests.

Because the tensor class is not supported on the FPGA with our implementation, we chose to flatten tensors in a one-dimensional array for the accelerated convolutional layer. Given the scope of this project and the complexities of the FFT and Winograd algorithms, we focused on accelerating the traditional convolutional operator on the PYNQ-Z FPGA. To further simplify the hardware design and to increase parallelism, the FPGA applies the convolution on a channel-by-channel basis (see section 9.2). As such, the accelerator ap-

plies the basic convolution on a one-dimensional array representing a 2D matrix. After the FPGA returns its output, the one-dimensional output is again converted to a Tensor on the CPU. Consequently, the associated loops for generating the FPGA input are executed on the PYNQ-Z2 CPU, as is adding the bias and all executing the other layers of the CNN.

Algorithm 1 Passing single image layers to the FPGA

procedure CONVOLUTION($N_{\text{inputChannels}}$, $N_{\text{outputchannels}}$)
  for $i \leftarrow 1$ to $N_{\text{inputchannels}}$ **do**
    for $j \leftarrow 1$ to $N_{\text{outputchannels}}$ **do**
      Processing of x, w and b(i, j)
      Store x and w in shared memory(i, j)
      Start FPGA(i, j)
      Get z from sm and concetenate(i, j)
    end for
    Sum up over inputs and add bias(i, j)
  end for
end procedure

The first step of our function is to prepare the arrays of x, w to pass them to the shared memory. For instance, shared x contains just the input values for the first input and output channel combination. After the values are stored we start the FPGA by setting the first bit to one (0b1) and waiting until the FPGA is done. This is signalized by the third bit set to one. After that, we can read the output value from the shared memory and concatenate all output channels. Indeed, the inner for loop of the convolution process computes only a fraction of the final result. To obtain the complete convolution output, the algorithm needs to sum up all the input channels and add the corresponding bias. The entire convolution output can now be stored in a tensor again and passed to the next layer (ReLU). Just to mention: we can also loop over the output channels in the outer loop and use the input channels in the inner loop. That allows you to start the next layer after your first loop. This approach was not implemented and tested yet but is promising for additional parallelism. We will explain this algorithm in more detail in our explanation for future works in section.

- We used different synthesizing times like 10ns or 5ns, but for 5ns we got a negative slack of 0.2. The highest frequency we can get with no negative slack with an uncertainty of 25% is 171 MHz. We decided to go with just 8ns to create the second-bitstream to achieve a frequency of 125 MHz, but it did not really effect the computing time.
- The FPGA convolution function is called multiple times during the inference. The values for all output channels are concatenated together with fractions for each input channel plus the fraction of the output. After this step, we sum up all the fractions for each input channel to get the complete output. For practical purposes, we chose to hardcode the different convolutions in a separate bitstream file. Since it turned out that a division between CPU and FPGA was best reached by executing the first and last convolutional layers on the FPGA and the other convolutional layers on the board CPU, there are only two bitstreams that have to be loaded during the inference. Later we will discuss how this loading times effect the performance.

6 Proposed Accelerator Design

The proposed accelerator design is synthesized using the Xilinx Vitis HLS (v2022) tool with the Xilinx Zynq-7000 (xc7z020clg400-1) FPGA chip. The models are then exported to the Xilinx Vivado Design Suite tool (v2022.2) to generate a bitstream file. This section explains the 2D convolution strategy on the FPGA.

6.1 Normal Convolution

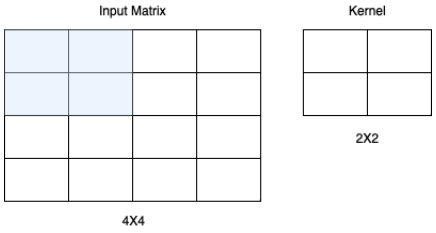

Figure 7: First Step

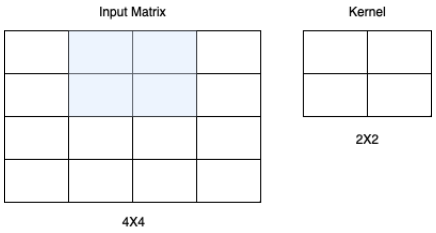

Figure 8: Second Step

Before we move to the line buffer, we want to demonstrate a normal convolution and discuss some possible drawbacks of the buffer structure. Figure 7 and 8 show the normal convolution. To calculate the first output value, we perform element-wise multiplication of the kernel with a window extracted from the input matrix. If the stride is set to one, to calculate the second output value, we shift the window one index to the right and again perform element-wise multiplication with the kernel. This process continues iteratively, generating subsequent output values as the window moves with the specified stride over the input matrix.

To understand the advantage of using a line buffer, let's consider the following situation. In our convolution operation with a stride of 1, we can store the values of the previous window (from the input matrix) inside the buffer. When we move to the next window, we can use the last window's column from the buffer. However, we still need to access the new values as they are not yet stored in the buffer. In this example, this additional access is equal to the kernel's height.

Storing values in this manner is not preferable because our goal is to maximize pipelining and parallelization. We want to access just one new value for every step of the convolution.

The solution to this problem is to use a line buffer. The line buffer allows us to store the necessary values in a way that facilitates accessing only the required new value for each step of the convolution. This approach helps improve the efficiency of the operation and enables better pipelining and parallelization, making it more suitable for FPGA implementations, especially in Xilinx devices.

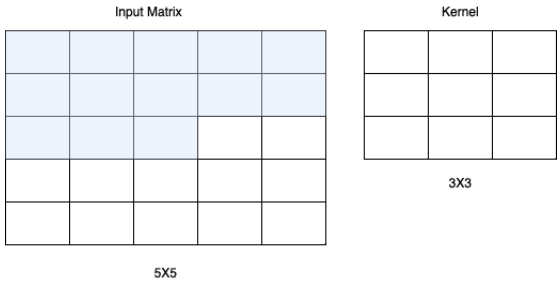

Figure 9: Line buffer First Step

Figure 10 demonstrates the first step of the line buffer using a relatively small input size and kernel size to provide a clearer understanding. In this example, the objective is to access just one value from a RAM (memory) in every step of the convolution. To achieve this, we store 2 rows in the buffer, which is determined by the kernel height. The exact formula for buffer size is (kernel height - 1). Additionally, we store the first 3 elements from the 3rd row, which is based on both the kernel width and kernel height.

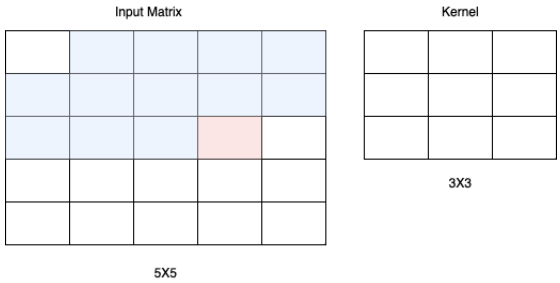

Figure 10: Line buffer Second Step

Figure 11 illustrates the second step of the convolution. As we are storing the light blue pixels in the line buffer, we only need to access the light pink value from the input matrix. This process continues iteratively. Each time, we discard the first value from the line buffer and add the new value to the end of it. By following this approach, we can efficiently apply the convolution operation, accessing just one value at a time during each step.

This technique allows us to decrease the initiation interval to 1 for the convolution loop. It is particularly valuable as it significantly reduces the total time required for the convolution process. By achieving an initiation interval of 1, we can maximize parallelization during convolution.

Algorithm 2 shows the implementation of the main code on the board as pseudocode. Since we are performing convolution channel by channel, our conv2D function receives the channel values of the output, input, and weight as sequential arrays due to limitations in using pointer-to-pointer.

To minimize storage usage, we store these values using `ap_fixed`$< 24, 2 >$ fixed-point data type. We will discuss this later.

Then we use pragma HLS ARRAY PARTITION to partition the arrays line buffer, weight, and window. This allows us to optimize the data storage and resource usage.

After this step, the convolution operation is initiated, and we follow the same steps as mentioned before. The HLS PIPELINE directive provide us to achieve an initiation interval of 1 for every type of kernel and input type.

Once the convolution is applied, we slide the line buffer one index to the right and update the last index of the line buffer with the new value. Next, we slide the window to the right, utilizing the already stored elements. Finally, we place the last value of the window in the last index of the line buffer.

With these steps, we can efficiently perform the convolution operation with initiation interval 1, accessing just one element at a time. This approach optimizes the memory access pattern, enabling improved performance and efficient resource utilization.

Algorithm 2 conv2D(float out[output size], float in [input size], float weight[filter size])

```
1: fixed point line buffer[.....]
2: fixed point weight[.....][.....]
3: fixed point window[.....][.....]
4: pragma HLS ARRAY PARTITION line buffer, weight, window
5: for height = 1, 2, ..., y do
6:     for width = 1, 2, ..., x do
7:         pragma HLS PIPELINE
8:         if if x is smaller than then...
9:             out[current index] = convolution(window, weight)
10:        end if
11:        for linebuffersize = 1, 2, ..., i do
12:            line buffer[i] = line buffer[i+1]
13:        end for
14:        line buffer[last index] = new pixel value
15:        for fheight = 1, 2, ..., i do
16:            for fwidth = 1, 2, ..., j do
17:                window[i][j] = window[i][j+1]
18:            end for
19:        end for
20:        for fheight = 1, 2, ..., ji do
21:            window[i][filter width -1 ] = linear buffer[correct index]
22:        end for
23:    end for
24: end for
```

Type	Sensitivity	Avg. Diff.
Float	1×10^{-4}	Equal
Float	1×10^{-7}	$1.788\,14 \times 10^{-7}$
ap_fixed$< 24, 2 >$	1×10^{-4}	Equal
ap_fixed$< 24, 2 >$	1×10^{-7}	$8.761\,88 \times 10^{-6}$
ap_fixed$< 16, 2 >$	1×10^{-7}	0.00239
ap_fixed$< 8, 2 >$	1×10^{-7}	0.520215

The above table shows the accuracy based on different data types. To decrease the resource usage, we decided to use fixed-point data types rather than float data type. However, when we reduce the number of bits we use, we also lose accuracy in the calculations. To make this decision, we examined the following values in the table. From the table, we can understand that for ap_fixed$< 8, 2 >$ and ap_fixed$< 16, 2 >$, we are not able to obtain correct results. On the other hand, ap_fixed$< 24, 2 >$ provides correct values for the sensitivity of 1×10^{-4}, which is sufficient for us to proceed with it.

Type	BRAM,URAM%	DSP%	FF%	LUT%
Float	0	880	357	793
Fixed P	0	22	58	40

After deciding to go with ap_fixed$< 24, 2 >$, we examined the resource utilization for both float and fixed-point data types. As evident from the values, float is not a viable option due to its high resource usage. In contrast, by using fixed-point representation, we significantly save space in terms of DSP, FF, and LUT resources.

BRAM and URAM are 0 percent because we are not utilizing explicitly so it's natural to have 0 percent usage. DSP resources are commonly used for performing multiplication and accumulation operations, which are often needed in convolution operations. In the code, the multiplication and accumulation may occur inside the convolution() function, which is likely contributing to the DSP usage.

Flip-Flops (FFs) are used for storing data temporarily during computations. The code utilizes a fixed-point line buffer (line buffer[]), weight (weight[][]), and window (window[][]) arrays, which would require FFs to store intermediate values and data.

Look-Up Tables (LUTs) are used to implement combinatorial logic in FPGA designs. The control logic, loop handling (for loops), and pipeline (pragma HLS PIPELINE) in the code might be using LUT resources.

Our throughput is due to our architecture comparably low. From a total latency perspective, you can observe the difference between the unoptimized and optimized implementations for the input tensor 1X134X134 and the kernel 7X7.

Unoptimized	Latency	IterationL	Trip C
Loop 1	9208064	71938	128
Loop 2	71936	562	128
Loop 3	553	79	7
Loop 4	77	11	7

The general trade-off between latency and throughput is handled here by using a very low latency as you can see in the next table.

Optimized Version:

Optimized	Latency	IterationL	II	Trip C	Pipe
Loop 1	18931	-	-	-	N
Loop 2	817	8	1	811	Y
Loop 3	55	8	1	49	Y
Loop 4	52	5	1	49	Y
Loop 5	18034	80	1	17956	Y

Bullet points:

- We prefer to use Fixed point structure to save space from BRAM, DSP, FF, LUT, URAM. We save almost 300 percent space, which helps us decrease the initiation interval to 1. We used ap_fixed<24, 2> instead of the float type.
- We used a line buffer to utilize the buffer efficiently, ensuring that we only need to load one index per loop of the convolution.
- We partitioned the buffer, window, and weight variables so that we don't have dependencies and used registers to operate.
- After we applied the above approach, we achieved an initiation interval of 1 for every convolution layer in SmallNet. Additionally, we were able to reach up to **171 MHz with this approach**. We could reach more but we didn't want to increase the initiation interval.

7 Benchmarking the Accelerated onvolution on a PYNQ-Z2 FPGA

The benchmarking network consists of smallNet, this time with pre-trained weights. It is able to classify images despite the limited amount of layers. For the different layer sizes, the reader is kindly referred to figure 18 in appendix 11.1. Compared to largeNet and giantNet, smallNet offers a broader benchmark because it uses three different kernel sizes (3,5,7). Compared to mediumNet, smallNet gets the upper hand because the larger variance with the number of input - and output channels, offering material for a broader analysis.

7.1 Layer Partitioning and Hardware/Software Co-Design Rationale

The first question is which layers of smallNet should be offloaded onto the FPGA hardware while retaining others on the CPU. As the barchart of smallNet in figure 18 in appendix 11.1 illustrates, executing the FFT convolution takes a longer time then both the basic convolution and the Winograd convolution. Thus, the FFT operator is not taking into account anymore. Since we implemented the basic convolution on the FPGA, we can expect the FPGA convolution to be the most suitable for the layers where the basic convolution on the CPU already outperforms Winograd. This is the case in the first and last layer. Figure 11 demonstrates that this is indeed the case: only for the first and last layer, there is an added value of executing the convolutional layer with the FPGA. It is remarkable that the basic convolution on the FPGA is not much slower then the Winograd convolution even in the middle layers. However, sine Winograd is still faster by a fraction, these should be executed using Winograd

and the first and last layer with the basic convolution on the FPGA.

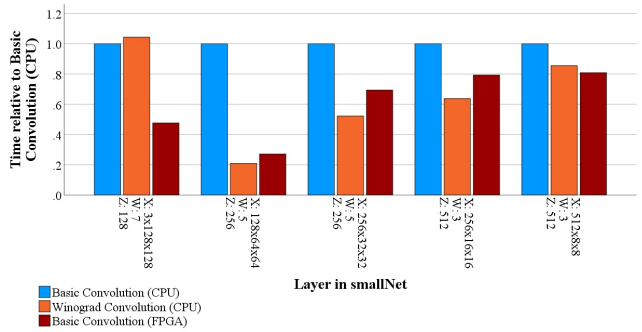

Figure 11: Comparison of the time per layer in smallNet for the basic convolution executed on the FPGA and the Winograd convolution on the CPU relative to the basic convolution on the CPU. X stands for the input tensor, W for the weight tensor, and Z for the output tensor.

7.2 Time Breakdown: Loop Overhead vs. FPGA Computation

To further understand the different performance of the basic convolution on the FPGA per layer as shown in figure 11, we need to delve into the time distribution within our architecture's convolutional layer. As explained in section 5, the execution of the convolutional layer using the FPGA, referred to as 'basic convolution (FPGA)', can be divided into two parts. One part is calculating the 2D convolution on the FPGA. The other part is going through the loops for a number equal to the number of input channels times the number of output channels. Here, we need to extract the part of the input, weight and bias array related to the current 2d convolution. The part of the input and weight array also has to be stored in shared memory. After the FPGA returns the output, the bias needs to be added on the CPU side. Outside of the loops, there is also the time needed for loading the bitstream and for loading the output array back into the Tensor class data type. Since figure 19 in appendix 11.2 shows that the loading of the bitstreams to the FPGA only takes a fraction of a second (about 1 percent of the time for the basic convolution (FPGA)), we can refer to the last part as the 'loop overhead'.

Figure 12 plots the average time the FPGA requires for calculating a single 2D convolution in each layer It is a logical consequence of our implementation that it only depends on the input width and height. As such, the first layer requires the most time and the last layer the least time. However, as we know, the basic convolution (FPGA) is most suitable on the first layer. This suggests that this part of the basic convolution (FPGA) is not the decisive part. The question is than, what is the exact division of time between the FPGA part and the loop overhead part?

In figure 13, we can see that the biggest part of the basic convolution (FPGA) is actually spent on the PYNQ-Z2 CPU. Over the five layers of smallNet, around 70 percent is on average spent on the PYNQ-Z2 CPU, while roughly 30 percent

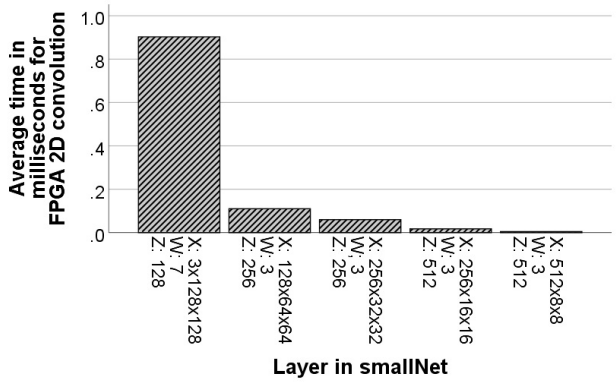

Figure 12: Average time needed by the FPGA to calculate a single 2D convolution per layer.

of the total time is consumed by the PYNQ-Z2 FPGA. It is remarkable how a higher number of iterations in the last layers is not associated with a higher loop overhead. The first layer, with only 384 iterations compared to 262.144 iterations in the last layer has the second highest loop overhead. One explanation is that in each iteration, the filter matrix for a particular 2D convolution has to be extracted from the filter array. Since the first layer has the biggest filter, this loop has a significant effect. The same could be true for the second layer, combined with a higher number of output channels. This suggests that a higher filter size, such as 11, is not as suitable for our approach as a small kernel size.

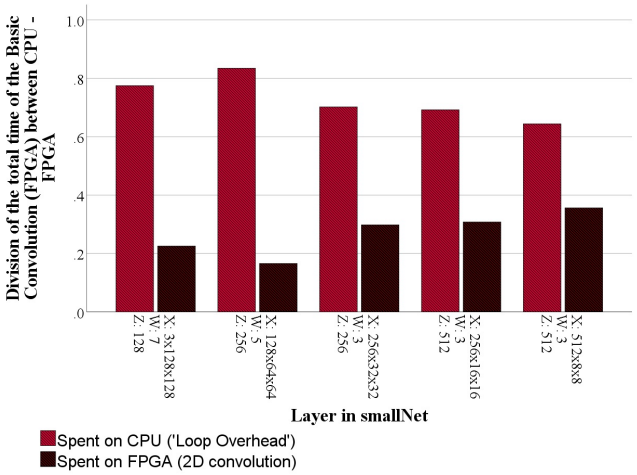

Figure 13: Division of the total time required by basic convolution (FPGA) between the actual 2D convolution on the FPGA and the loop overhead on the CPU.

7.3 Performance Evaluation: Unoptimized vs. Winograd vs. Fully Optimized smallNet

Now that we have an optimization for each layer, it is time to compare the computation speed of the three distinct approaches to perform the inference in smallNet: using the ba-

sic convolution (CPU) in every layer, the Winograd convolution in every layer, and a combination of Winograd and the basic convolution (FPGA). The last approach is referred to as 'the fully optimized version'. This is plotted in figure 14. Absolute times can be found in figure 20 in appendix 11.3.

Figure 14 shows the total time needed by the the three systems relative to an inference where only the basic convolution (CPU) was used. As could be expected from section 4.1, an inference on smallNet where the Winograd convolution is used for every layer is about 2,6 times faster then its equivalent with the basic convolution (CPU). It is stunning that an inference using only the basic convolution (FPGA) is not far off, being almost 2,5 times faster then the unoptimized smallNet. This suggests an even more sizeable speed-up if the Winograd convolution would be implemented on the FPGA. Lastly, the fully optimized version where the most appropriate convolutional operator is selected for each layer is 2.72 times faster. More then 70 seconds are saved by using this configuration. Figure 14 shows the total time needed to classify an image with smallNet was calculated for four systems: the basic convolution (CPU) for every convolutional layer, the Winograd convolution for every convolutional layer, the basic convolution (FPGA) for every layer, and the fully optimized version that combines Winograd and the basic convolution (FPGA). Then, the time relative to an inference only using the basic convolution (CPU) was plotted. Timings are the average over 15 image classifications.

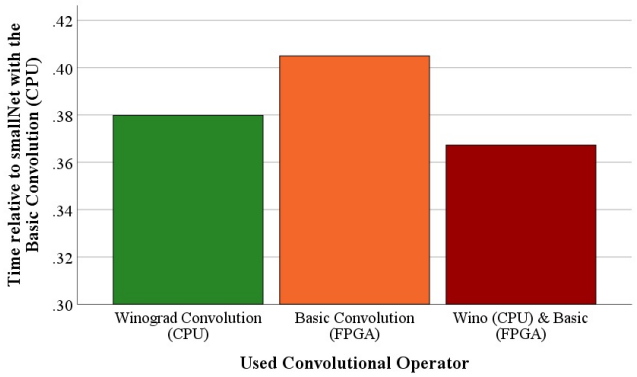

Figure 14: Total inference time for different approaches relative to the basic convolution on CPU

8 Implementation for SmallNet

The benchmarks results in section 7 suggest to implement the first and last convolution on the FPGA, while running the remaining operations on the CPU. The time needed for ReLU and MaxPooling is negligible, therefore it can be done by the CPU. To understand our implementation it is important to take a look at the architecture which we want to accelerate with our approach. In figure 15 you can see the general SmallNet architecture that consists of five convolution layers with ReLU and MaxPooling and a classifier. In our architecture we are using the already implemented functions from lab1. So we created a function, that converts our one-dimensional

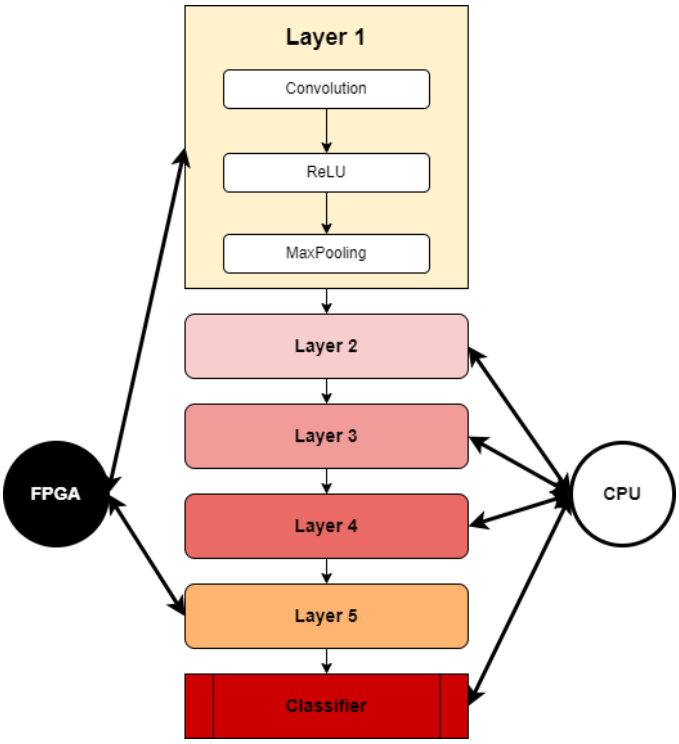

Figure 15: SmallNet architecture and demonstration of task-split

output array from the first convolution to a tensor again. Layers 2-4 are calculated by the CPU and the outputs then passed to a flattening function, to generate an one-dimensional array again. The next step is to load a new bitstream that is specialized for convolution 5 and then the steps are the same like for convolution 1. In the last step we generate a tensor again to use our implemented ReLU and MaxPooling functions. Finally the classifier is operated on the CPU. It contains multiple linear layer and activation functions that are not computational extensive so there is no need in running them on the FPGA.

9 Beyond our implementation

There are aspects that need to be considered beyond our implementation. For example, what changes need to be made to be able to use our solution for other networks. Additionally there are aspects that need to be further improved for a maximum acceleration of the inference.

9.1 Modularity of our Approach

This report primarily centers on detailing the implementation of our acceleration approach for the SmallNet architecture. How can these optimizations be used for other architectures? There are only a minor number of changes that need to be made to adept our approach to other networks. Since the layer dimensions are stored in the bitstream itself, a new bitstream encoding this data should be created for each layer. It is also possible to generate just one bitstream that is capable of handling different input and kernel sizes. It needs to be

evaluated which logic gains more performance. On the CPU side most parts are already modular. Just the logic which bitstream should be loaded must be dependend on the choosen architecture (when you stick to multiple bitstreams). Important to mention: It is necessary to run new benchmarks and see, which layer should be handled by the FPGA and which layers are promising more performance on the CPU. As you can see for SmallNet it is just the case for the first and last layers.

9.2 Future Perspectives

Parallelization plays a vital role in achieving efficient acceleration. Our primary objective is to maximize the utilization of both the PYNQ-Z2 FPGA and the PYNQ-Z2 CPU resources continually. To achieve this, we propose a strategy where each convolution operation for an output layer is divided between the FPGA and the CPU.

To achieve our goal of parallelization, we will modify the for loop responsible for passing values to the shared memory. For every three iterations, we could pass only two iterations to the FPGA and run one iteration on the PYNQ-Z2 CPU. The results will be combined at the end.

Furthermore, we plan to use a single bitstream since our Vitits code requires the correct dimension values. To achieve this, we will pass a single value to the shared memory, which will be used to select a specific case containing the required sizes for the computation. That saves the time needed to load a new bitstream.

To increase the parallelism it is advisable to switch the order of the for loops. That allows you to pass the first output channel of the convolution already to the next layer, without waiting for the remaining output channels. Once we complete the first output channel iteration, we have two options for further processing. We can either pass the obtained value to the ReLU and MaxPooling functions separately, or we can choose to concatenate all the output arrays. To determine the most efficient approach, we need to conduct thorough testing and evaluation.

Algorithm 3 Passing single image layers to the FPGA and CPU parallel

procedure CONVOLUTION($N_{\text{inputChannels}}$, $N_{\text{outputchannels}}$)
  for $i \leftarrow 1$ to $N_{\text{outputchannels}}$ **do**
    for $j \leftarrow 0$ to $N_{\text{inputchannels-1}}$ **do**
      Processing of x, w and b(i, j)
      Store x and w in shared memory(i, j)
      Start FPGA(i, j)
      Get z from shared memory(i, j)
    end for
    for $j \leftarrow 1$ to $N_{\text{inputchannels}}$ **do**
      Processing of x, w and b(i, j)
      Start Winograd on CPU(i, j)
    end for
    Sum up over inputs and add bias(i, j)
    Concatenate arrays(i, j)
  end for
end procedure

Figure 16 illustrates the concept of algorithm 3, where both the CPU and FPGA can operate simultaneously for one convolution. While this example assumes an equal number of convolutions on both the CPU and FPGA, our benchmarks indicate that the optimized code enables the FPGA to perform convolutions significantly faster. As a result, we would prefer to offload more convolutions to the FPGA rather than the CPU to leverage its accelerated performance.

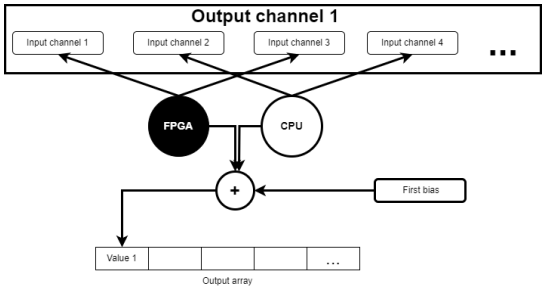

Figure 16: Principle of overlapping convolution for increased parallelism

Most importantly, Winograd needs to be implemented on the FPGA, for further enhancing the time boost and overall efficiency of the convolution operations. This implementation represents an exciting direction for future optimizations, leading to even more significant acceleration gains.

10 Conclusion

This project paper proposes an accelerated Convolutional Neural Network (CNN) architecture designed specifically for the FPGA-based PYNQ-Z2 development board. The objective is to explore the efficacy of fast convolutional operators, such as the Winograd and FFT convolutions, in conjunction with FPGA-based implementations. The first idea was to optimize convolution algorithms on the software side using different approaches. The result was, that the implementation of a convolution using the Winograd algorithm promised us the best results. In the next step we wanted to enhance our inference speed even further with using additional hardware. Due to the fact, that the optimized convolution algorithms are more complex and need more time to implement, we decided to begin with implementing the basic convolution on the FPGA. Caused by the good results and practical contstraints we remained with this approach.

The experimental results are highly promising, showing a remarkable 2.7-fold overall speed-up by applying optimizations to a small CNN. Also the decrease in memory needs is significant. These optimizations include running the basic convolution on the FPGA for the first and last layer and using Winograd convolutions for the middle layers. The basic convolution (partly) executed on the PYNQ-Z2 FPGA performs above expectation, competing closely with the fast convolutions executed on the PYNQ-Z2 CPU.

We highlight the modularity of the approach, making it possible to apply the proposed optimizations to other CNN architectures.

However, since the Winograd convolution outperforms the basic convolution significantly for the middle layers, it is important to mention that implementing the Winograd convolution on the FPGA is the next step. In addition, in theory the FFT convolution could be significantly faster then it currently is. Especially for larger kernels such as 7x7 or 11x11, it should outperform the Winograd convolution. Our current implementation is slower then expected and needs to be optimized for a better comparison. For instance we could reduce the amount of loops to speed up the calculation since this is taking the biggest chunk of computation. Future research should explore further parallelism strategies and adapt the implementation of the Winograd and FFT convolution on the FPGA to achieve significant acceleration gains. Especially the lack of parallelism reduces the technically possible acceleration of the process.

In general our implementation demonstrates that the combination of hardware accelerators and optimized convolutional operators are holding great promises for the future of FPGA-accelerated deep learning systems, enabling faster and more efficient computations for a wide range of applications. It also demonstrates the challenges like handling larger amounts of data.

References

[Szegedy *et al.*, 2015] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[Wang *et al.*, 2017] Zelong Wang, Qiang Lan, Hongjun He, and Chunyuan Zhang. Winograd algorithm for 3d convolution neural networks. In *Artificial Neural Networks and Machine Learning–ICANN 2017: 26th International Conference on Artificial Neural Networks, Alghero, Italy, September 11-14, 2017, Proceedings, Part II 26*, pages 609–616. Springer, 2017.

11 Appendices

11.1 Relative time per layer for basic, FFT and Winograd

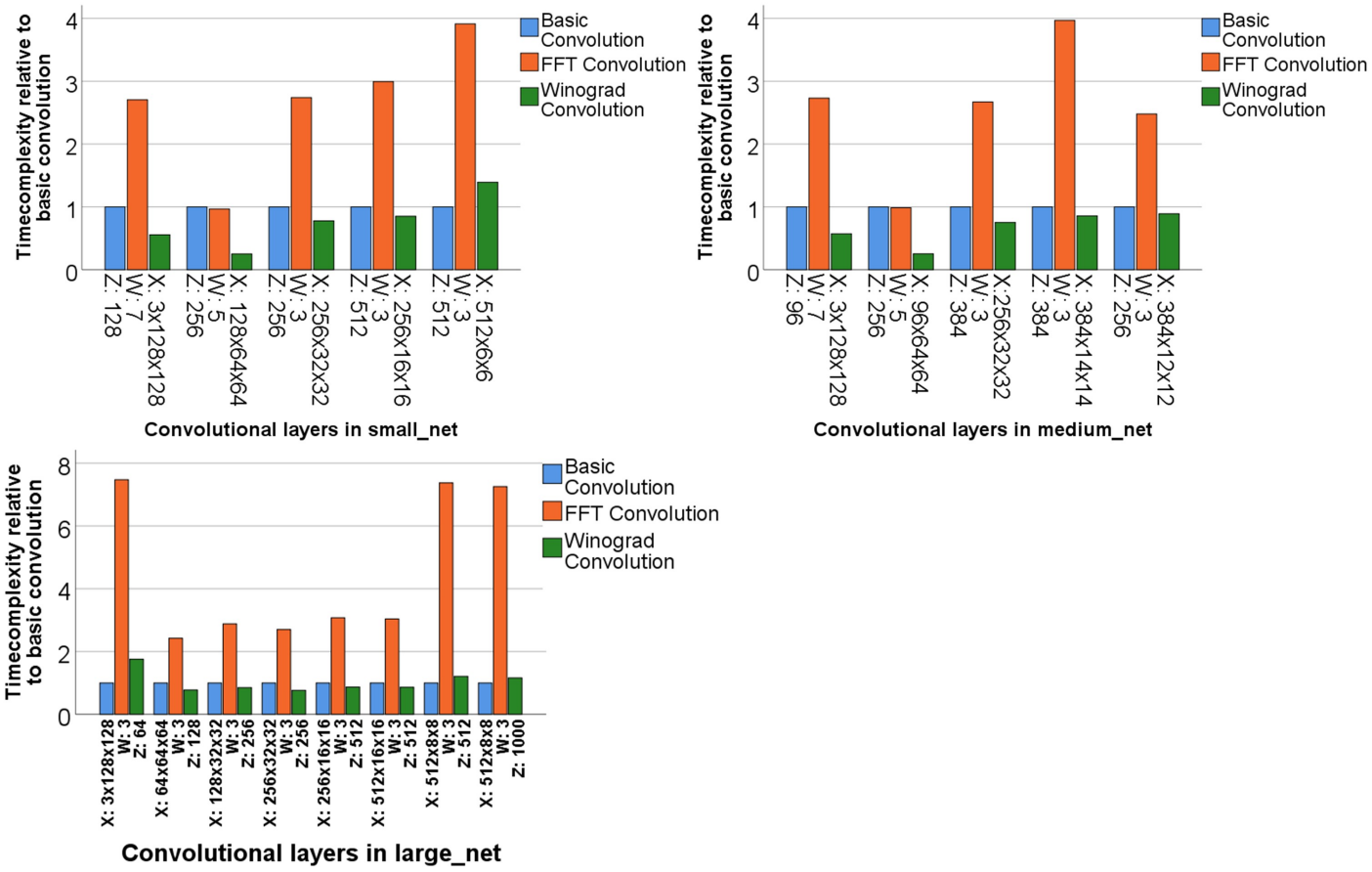

Figure 17: Comparison of the runtime relative to the basic convolution for each layer per CNN tested on a PC. X stands for the input tensor, W for the kernel height and width, Z for the number of output channels.

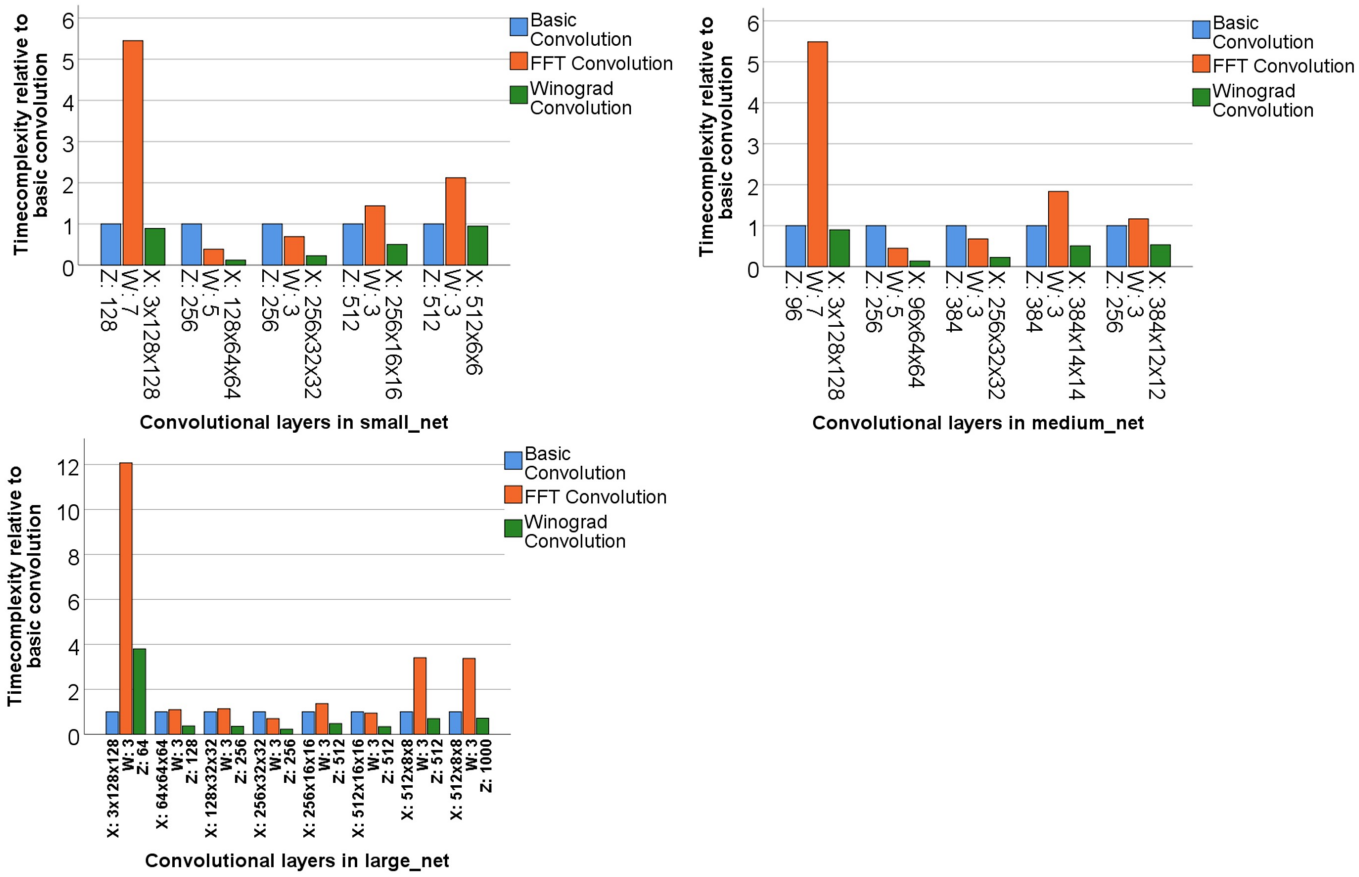

Figure 18: Comparison of the runtime relative to the basic convolution for each layer per CNN tested on the board. X stands for the input tensor, W for the kernel height and width, Z for the number of output channels.

11.2 Time for loading the bitstreams

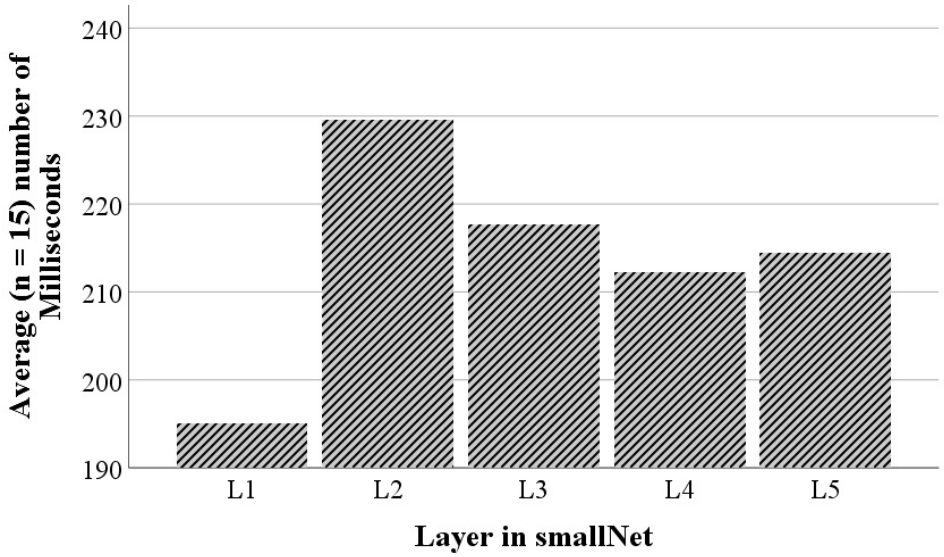

Figure 19: Average time in milliseconds for loading the bitstream to the FPGA for each layer in smallNet.

11.3 Absolute timing for the final four adaptions of smallNet

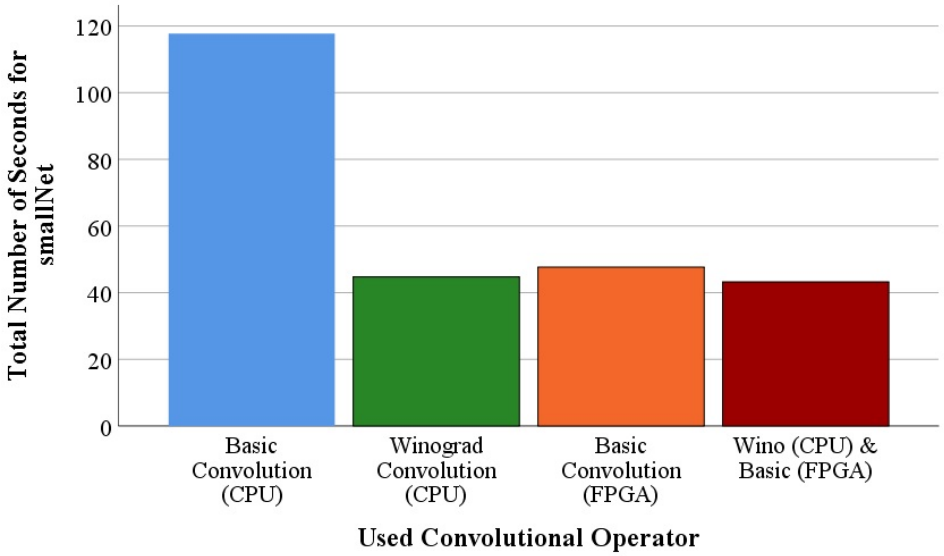

Figure 20: Time in milliseconds for three convolutional operators and the fully optimized inference.