Mini Project: CFU Playground

P N Neelesh Sumedh EE19B047

July 6, 2022

1 Introduction to CFU Playground

CFU Playground is a full-stack open-source framework which enables fast and iterative design of Machine Learning (ML) accelerators for embedded ML systems. CFU stands for Custom Function Unit. The CFU is the accelerator hardware that is tightly coupled into the pipeline core of the CPU. The instruction of the Custom Function Unit are similar to the standard RISC-V CPU functions such as logic operations and arithmetic operations. The CPU has an opcode allocated for CFU. When an instruction has an opcode which points to CFU, the contents of the registers are passed into the CFU and some specific operation is done using a separate 10bit opcode which is different from the opcode of the RISC-V.

2 Problem Statement

The initial plan of work was to improve the CFU given in the CFU Playground documents website to store 4 filter values in the CFU and to reuse them until all the multiplications with those filter values are completed. The CNN model was chosen to be a custom SVHN model which predicts digits from the SVHN dataset.

2.1 Final Problem Statement worked upon

Accelerate one MNIST CNN model and one SVHN CNN model using the CFU. Model the CFU such that it stores the filter values taking 4 layers at a time or 8 layers at a time based on the CNN model. Also pipeline the Input Channel Values to reduce the number of memory accesses.

Note: Since the optimizations are done iteratively, each optimized version is given a version number to track them easily. We will use those numbers here aswell. The version details are given in the Readme file in the Code Files folder. The convolution code will be named conv_ver_A_B_C_D.h and verilog code will be named cfu_ver_A_B_C_D.v if the version is A.B.C.D.

GitHub Repository Link: LINK
Video Link: LINK

3 CNN Models

The MNIST model is shown in the figure 1. The number of parameters for this model are 9558. The accuracy for this model on the test data is 99.09%. The code for writing this model and training it is given in the following colab link: MNIST MODEL

The SVHN model is shown in the figure 2. The number of parameters for this model are 118762. The accuracy for this model on the test data is 94.7%. The code for writing this model, training it and converting it into tflite model is given in the following colab link: SVHN MODEL

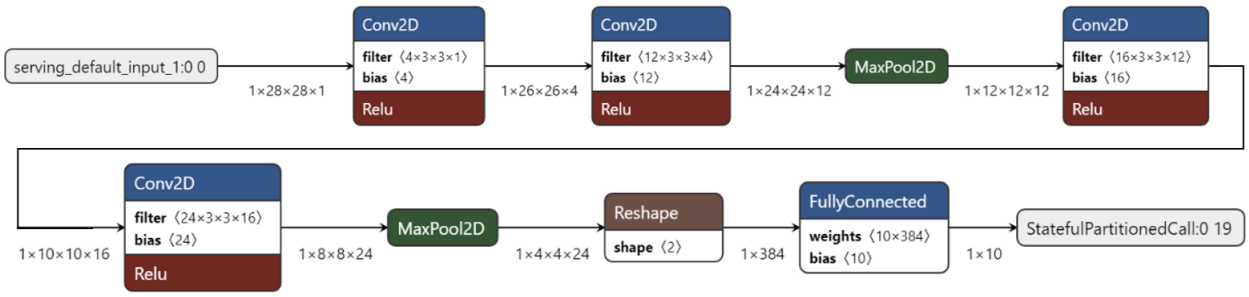


Figure 1: MNIST Model

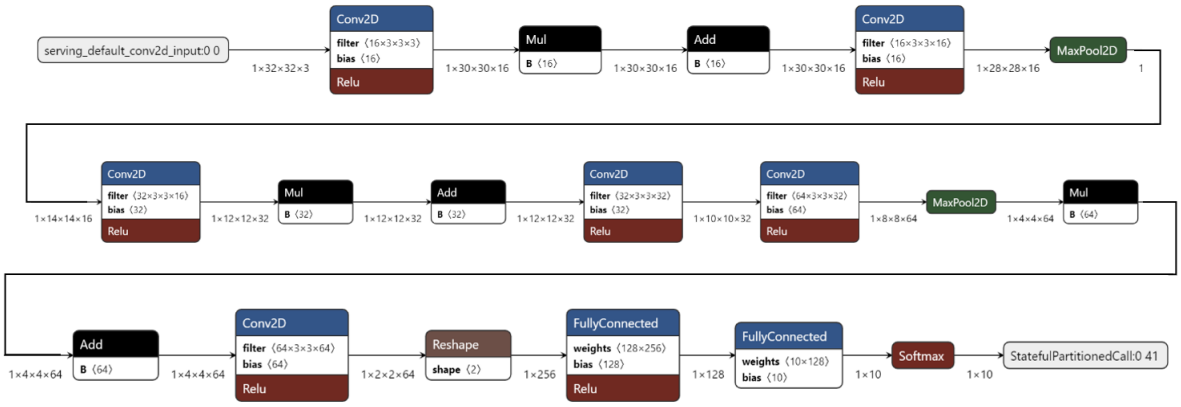


Figure 2: SVHN Model

4 Baseline Code

The baseline code chosen for comparison is the conv.h code with constant optimizations. The CFU playground framework has reference codes for basic operations like convolution, addition, fully connected layer etc. The conv.h file has the general code for convolution. For the CNN models, there are some parameters which do not change for the convolution layers in the model. Some examples are filter sizes, stride width, stride height, pad width, pad height etc. Instead of storing these values in a variable and using them in the code, we directly write the value in code instead of the variable. This constant optimized code is taken as the baseline for both the CNN models.

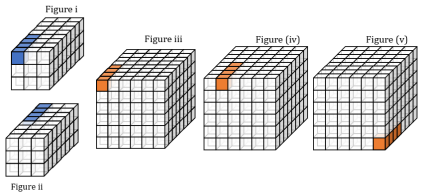
5 Accelerating the CNN Models

The optimizations and the accelerator part for both the MNIST and SVHN models are same with some minor changes due to the nature of the CNN models. First we explain the optimizations and accelerators done on the MNIST. The changes in the accelerators of SVHN will be explained in the SVHN section.

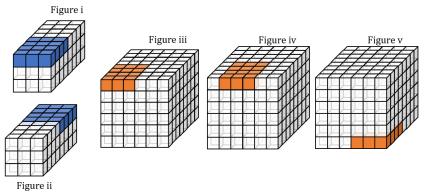
5.1 MNIST Model

5.1.1 Loop Unrolling (Version 1.1.0)

If we look into the baseline code of the MNIST model , you can see that in the innermost for loop we iterate through the input layers. But from our MNIST model, we can see that the number of layers for input of all convolution layer except the first convolution is a multiple of 4. So we wont face any



(a) Version 1.2.1.0



(b) Version 1.3.1.1

The values in Blue box of figure i are the filter values stored in the CFU. The values in the orange boxes are the input values. First the blue box values are multiplied with the orange values in figure iii. Next the blue values are multiplied with the ones in figure iv and so on till figure v.

After that the filter values highlighted in figure ii will be stored and process is continued with input values from the next four layers.

(c) Note

Figure 3: Versions 1.2.1.0 and 1.3.1.1

problem if we unroll the loop 4x times for all those convolution layers whose number of input layers is a multiple of 4. After Unrolling, we write the code in the innermost loop to give the correct result.

5.1.2 Changing the Order of For Loops (Version 1.1.1.0)

Our problem statement is to store the filter values and to use them completely before moving on to the next set of filter values. It is not possible to use the above code for our problem statement as we iterate through the input channels in the innermost layer. So we reorder the for loops to match our requirement.

5.1.3 Integrating CFU (Version 1.2.1.0)

In the above version, we are reading 8 bits of values four times from memory. Since we use 32 bit busses, we can read those 4 values with just one memory read. Now we model the CFU such that it stores the four filter values and multiplies those with the input values. With just 1 CFU instruction, we perform 4 multiply-accumulates. This here is known as **SIMD** operation.

5.1.4 Adding an out_values Buffer to CFU (Version 1.2.1.1)

In the above version, we use an array of memory to store the accumulated values for post processing. To reduce this memory accessing, we add an output values buffer to the CFU. Instead of storing the values in memory, we directly store it in buffer to do post processing. This way we reduce the number of memory reads and stores.

5.1.5 Storing 12 Filter Values in CFU (Version 1.3.1.1)

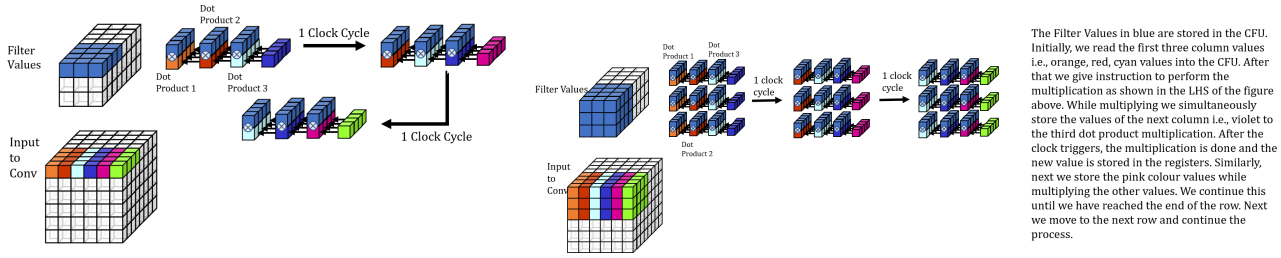
Extending the CFU from Version 1.2.1.1, we will unroll the for loop which iterates along the *filter_x*. So we will store the 12 values using the functions, **cfu_op1(1,filter_val,0)** and **cfu_op2(1,filter_val,filter_val_1)**. We also update the CFU to store the 12 filter values and to do these 12 multiplications simultaneously using one cfu function.

5.1.6 Pipelining the Input Values (Version 1.4.1.1)

In the above case, we are reading same values multiple times as seen in Figure 3(b). The highlighted values of 2nd column are read twice, the third column are read 3 times etc. To reduce this, we introduce pipelining of input values like in **Systolic Arrays**. This is shown in Figure 4(a). As we can see, we read a value only once and it is passed through a pipeline sequentially.

5.1.7 Storing 36 Filter values and Pipelining the Inputs (Version 1.5.1.1)

We extend the version 1.4.1.1 by unrolling along the filter_y for loop. Thus instead of storing 12 filter values, we store 36 filter values in CFU. Similar to above case, we pipeline the inputs as shown in Figure 4(b). Thus we perform 36 multiplications using a single cfu instruction instead of 12 multiplications. We also reduce the memory reads by adding the pipeline.



(a) Version 1.4.1.0 (b) Version 1.5.1.1 (c) Note

Figure 4: Weight Stationary Systolic Array Implementation

5.2 SVHN Model

The optimizations done for the SVHN model are the same as that done for MNIST model. One change done here is that for the first convolution, the input image has 3 layers. So when we first do the unroll optimization, we also unroll for the first convolution layer. The rest of the optimizations are combined and we directly write the version 2.5.1.1 which is similar to 1.5.1.1 . To further accelerate we do the following processing.

5.2.1 Adding an extra zero layer to the input image (Version 2.5.1.2)

The input image for the SVHN data set is of size 32x32x3. Since the depth is 3, we cant read 4 input values at a time from memory. But when we append an extra layer of zeros to the image, the depth becomes 4 and we can read values 4 at a time. For the 4th dummy value, we use a filter value of 0. Hence the final calculated answer doesnt change. So effectively instead of 3 memory reads, we are doing only 1 memory read. Incase of the filter values, we store all the filter values and reuse them completely thus reducing memory access.

5.2.2 Further unrolling across the input channel (Version 2.5.4.2)

For the SVHN model, the number of input channels for all the convolution layers is a multiple of 8. So instead of unrolling it by 4 times, we can unroll it by 8 times. We update this in the conv.h file and also in the cfu.v file to do 72 multiplications using a single CFU instructions.

The final CFU design is given in the Figure 5.

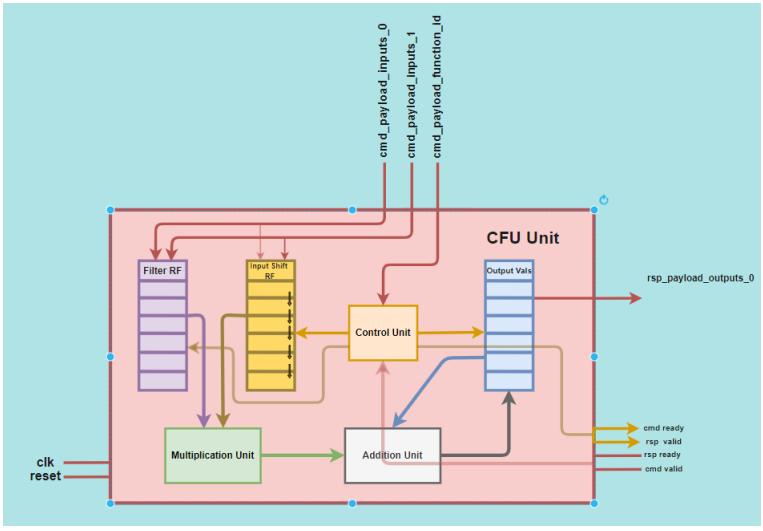
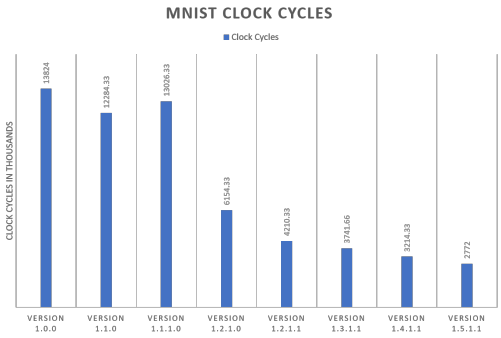
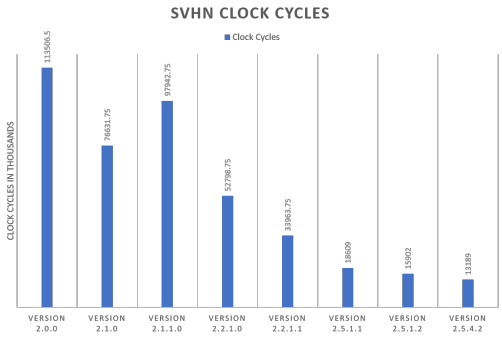


Figure 5: CFU Module

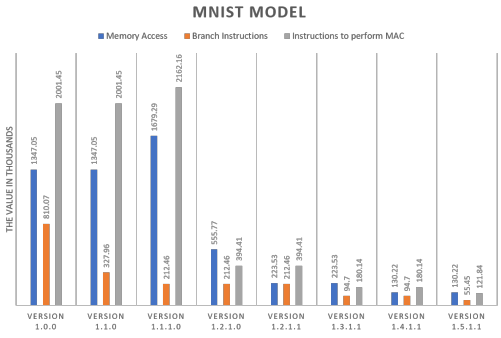


(a) Mnist Clock Cycles

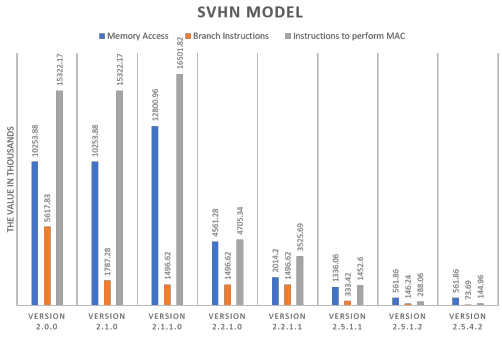


(b) Svhn Clock Cycles

Figure 6: MNIST & SVHN Clock Cycles



(a) MNIST Model



(b) SVHN Model

Figure 7: Factors effecting the clock cycles

6 Graph Data and Analysis

We collect the clock cycle data by simulating the code using verilator to get cycle-accurate RTL level simulation. We get the final number of clock cycles for each input. The number of clock cycles taken for each version is shown in the Figures 6(a) and 6(b). There are multiple factors due to which the clock cycles have reduced for each successive version. The main factors are the Number of Memory Accesses, No of Branch Instructions and No of Instructions to perform all the MAC operations. The graphs in Figures 7(a) and 7(b) show the count for each of these factors for each version. Before going to the analysis, the following assumptions were taken to calculate the numbers for the factors. All these values are hand calculated from the conv.h files for each version.

- Assumption 1: Every time we access elements from input or output array, it is taken as 1 memory access.
- Assumption 2: For every for loop there is 1 branch instruction and for every iteration, we read the branch instruction.
- Assumption 3: All the instruction which lead to either multiply or add or both of the channel values will be included in the instructions to perform MAC operations.

6.1 Analysis

6.1.1 Analysing Optimiations done on MNIST

The baseline for the MNIST model took around 13.8M clock cycles. The version is 1.0.0. This is taken as the baseline code. For this model we do loop unrolling to get version 1.1.0. After loop unrolling, the clock cycles reduce to 12.2M cycles. The reduction in this case is due to the reduction in number of branch instructions. As we can see in the Figure 7(a), the no of branch statements is reduced from 810k to 327k. The computation part and other factors remain the same here because unrolling doesnt change the computation but just reduces the loop overhead.

For the next change to get version 1.1.1.0, we face a problem with the post processing after changing the loops order. So, we store the accumulated values in an array to process them afterwards. This storing and reading from this new array results in more accessing of the memory, thus the memory access increases from 1347k to 1720k. Since we reordered the for loops, there is a change in the branch instructions and instructions for MAC operation which can be seen in Figure 7(a). The clock cycles increase from 12.2M to 13.0M. The major contributor for this increase is the increase in memory reads and writes. Though branch instructions decrease, the increase in the remaining two factors dominate.

The next step is to integrate the CFU with the CPU to perform tasks efficiently. After integrating the CFU the clock cycles reduce from 13M to 6.1M. In this version 1.2.1.0, we read 4 values of the input channels using just 1 memory access by using the 32 bit busses instead of reading just 8 bits from 1 memory access. Further we store 4 filter values and reuse them completely instead of reading them multiple times which is the case in all previous versions. By this we reduce the no of memory accesses drastically from 1679k to 555k. In the previous version, we require 3 instructions to perform 1 MAC operation. But in this version, we use just 1 CFU instruction to perform 4 multiplications and additions. So with just 1 instruction we perform 4 MAC operations. This type of computing is known as **SIMD (Single Instruction Multiple Data)**. The instructions to perform MAC fall to 394k from 2162k. The branch instructions remain the same as we are not changing the for loops. Thus reduction in memory accesses and instructions to perform MAC lead to the drastic decrease in clock cycles from 13M to 6.1M.

The next optimization is to add a buffer to the CFU to store the accumulated values to process them afterwards. By storing the values in this buffer instead of storing them in memory, we reduce the number of memory accesses from 555k to 223k. The remaining two factors remain same as we are just storing the values in buffer rather than in memory. So there is no change in computation or changes in loop to change the remaining to factors. Thus due to the reduction in memory accesses from 555k to 223k, the clock cycles reduce from 6.1M to 4.2M. The changes in factors is given in Figure 7(a). This version of code is 1.2.1.1.

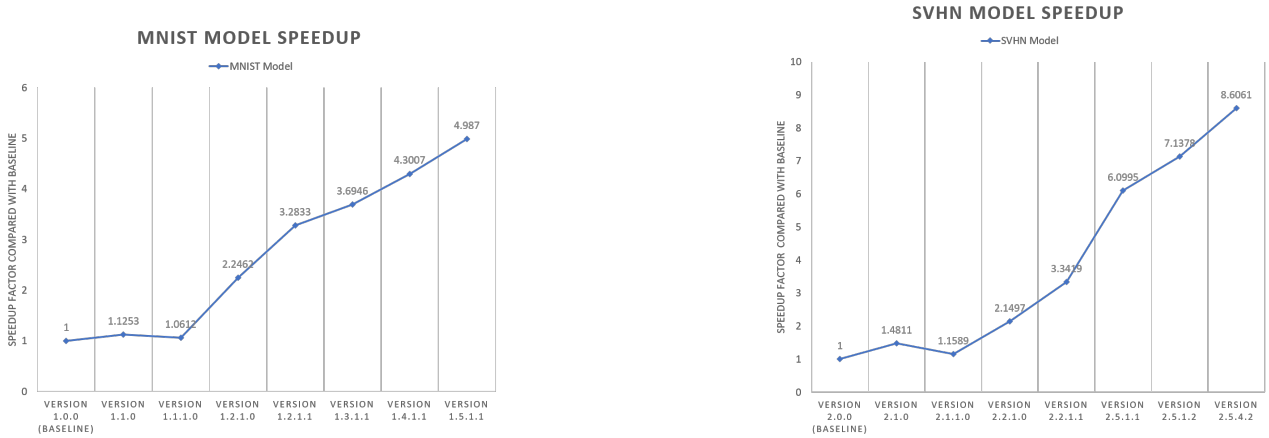
In the next step of iterative optimization, we get the version 1.3.1.1. In this we do unrolling along the filter_x direction. Since we have unrolled with a factor of 3, we can store 4x3, 12 filter values in CFU instead of 4 values. Since we have done unrolling, the number of branch instructions decrease. Apart from this, since we store 12 filter values, we do 12 multiplications using just 1 CFU instruction. Thus instead of doing 4 multiplications, we are able to do 12 using just 1 instruction. Thus the no of instructions to do MAC operation decreases from 394k to 180k. The no of branch instructions decrease from 212k to 94k. The memory accesses do not change as we are just unrolling. So, due to the reduction in branch instructions and instructions to do MAC operation, the clock cycles reduce from 4.2M to 3.7M.

Next we pipeline the input values and model the CFU like a small **Weight Stationary Systolic Array**. As explained in section 5.1.6, some values are read multiple times to do the MAC operation. So by introducing the pipeline as shown in Figure 4(a), we reuse the values and reduce the memory accessing. The memory accesses reduce from 223k to 130k. The remaining factors remain same as there is no change done which affects those factors. Due to the reduction in memory accesses from 223k to 130k, the no of clock cycles reduce to 3.2M. The data on clock cycles and memory accesses is given in Figures 6(a) and 7(a) respectively. The version of this code is 1.4.1.1.

Next we unroll along the *filter_y* direction. After unrolling, we can store 12x3, 36 filter values in the CFU for reuse. We also unroll the pipeline in the CFU three times as shown in figure 4(b). Thus we design the CFU to store 36 filter values and to have three separate input pipelines corresponding to 3 rows of the input channels. Since we are doing unrolling, the branch instructions decrease from 94k to 55k. Apart from this, we are doing 36 multiplications using one cfu instruction. Thus we are also reducing the instructions to perform MAC. They decrease from 180k to 121k. The final cycle count turns out to be 2.7M. Due to the reduction in branch instructions and instructions to perform MAC, the cycles reduce from 3.2M to 2.7M.

6.1.2 Analysing Optimizations done on SVHN

The baseline code version for SVHN is 2.0.0. The optimizations done for SVHN are same as done for MNIST. So the reason for the reduced cycles till version 2.5.1.1 is same as that of MNIST. In



(a) Mnist Speedup

(b) Svhn Speedup

Figure 8: MNIST & SVHN Speedup Factor

addition to this, a few more optimizations and processing is done to accelerate the model. These will be discussed in the below paragraphs.

We add an extra layer of zeros to the input images of SVHN dataset, so that we make the no of input layers be a multiple of 4. Once it is a multiple of 4, we will be able to use the CFU for this convolution layer aswell to accelerate it a bit more. Now that we have added an extra layer, we can read all the three input values with 1 dummy value in 1 memory read. Previously, there were 3 memory reads to get 3 input values. These input values are also sent into the pipeline as in Figure 4(b) for reuse. In addition to this, we store all the 27 filter values in the CFU to use them. So we have also reduced the memory access corresponding to the filter values. All these memory access reductions lead to decrease in the memory accesses from 1336k to 561k. The clock cycles reduce from 18M to 15M. The remaining two factors do not change as we have not affected the for loops. Thus the clock cycles in this case reduce due to the reduced memory accesses. The version of this code is 2.5.1.2.

Next we increase the unrolling factor of the convolution layers except for the first layer from 4x to 8x. Since for all the convolution layers, the count of input layers is a multiple of 8 except for the case of the first convolution layer. Since we unrolled from 4x to 8x, we can store twice of the filter values previously stored. So we can store 72 filter values in the CFU. Previously the input values are pipelined upto a depth of 4, now after unrolling we can pipeline them upto a depth of 8. So we update the CFU accordingly to get these changes. After this change, we need just 1 cfu instruction to perform 72 multiply accumulate operations. Thus there is a decrease in instructions for MAC from 288k to 144k and since we did loop unrolling the branch instructions also decrease to 73k from 146k. Due to these changes in factors, the clock cycles reduce from 15.9M to 13.1M.

7 Results and Conclusion

From the accelerations and optimizations done to each model we get some significant decrease in the clock cycles. For the MNIST model, we start with 13.8M cycles and it is reduced to 2.7M cycles with a speedup factor of 4.9. The speedup gained for each version is given in the Figure 8(a). For the SVHN model, we start with 113.5M clock cycles and reduce them to 13.1M clock cycles. We get a reduction of 100M cycles with a speedup of 8.6. The speedup gained for each version of SVHN is given in figure 8(b).

When comparing the speedup for MNIST and SVHN, we get higher speedup for SVHN for the similar versions. This is because of the larger network size of the SVHN model. The number of parameters for SVHN model is 118762 while for MNIST is 9558 which is almost 10 times lesser. For a larger network since the number of parameters are more, the effect of the optimizations done is more effective.

Thus we used the CFU interface to create an accelerator for two different CNN models one trained on MNIST dataset and other trained on SVHN dataset. Even though we have pipelined the input

values, we are still reading the input values multiple times. So we can try to add an input buffer to the CFU to store the input values and reuse them completely. We can also implement special module in CFU to do the post processing of output data before storing them. So, there is still scope of optimizations for the models and we can improve the clock cycles to even further extent.

8 References

[1]: CFU Playground Official Documentation
[2]: CNN Model Overview Generator