Ho Chi Minh City University of Technology

Faculty of Computer Science and Engineering

Department of Computer Engineering - Semester 231

Logic Design Project

2D Convolution on FPGA

Name:	*Student's ID:*
Pham Gia Hung	2110220
Phan Le Khanh Trinh	2151268

Instructor:

Tran Ngoc Thinh

Huynh Phuc Nghi

Contents

1 Introduction **2**

2 Theoretical foundation **2**

2.1 Two Dimension convolution 2

2.2 Kernel & Filter 3

2.3 Padding 3

2.4 Stride 4

2.5 Pooling 5

2.6 Input data type 5

3 Architecture design **6**

3.1 Workflow 6

3.2 Convolution block diagram in Verilog 7

3.2.1 Convolution module 7

3.2.2 Max Pooling module 9

4 Simulation results **11**

4.1 Edge detection 11

4.2 Time and Power consumption 12

5 Conclusion & Future work **15**

6 Source code **15**

7 References **16**

1 Introduction

Digital image processing and computer vision are rapidly evolving research fields with a growing number of applications for commercial, medical and military purposes. In these areas, improving perceptual information for human vision and processing image data for efficient storage, transmission and representation are two of the main goals.

Modern image processing and computer vision algorithms require high computational capability, especially when high-resolution images have to be elaborated under real-time requirements. In such applications (e.g. image filtering, image restoration, edge detection, etc.), the spatial domain two-dimensional (2D) convolution plays a fundamental roles. For theses reasons, the design of efficient Convolution Neural Network (CNN) accelerator receives great interest in other to minimize the required memory space and time computation while still maintains the correctness of the output results.

In this work, a new fully re-configurable 2D convolution layer for FPGA-based image processor is presented. The proposed architecture exhibits extreme flexibility and very high computational capability.

The paper is organized as follows: the theoretical foundation, related works is presented in section 2 & 3, then , the overall architecture flowchart & Methodology is described in section 4. Finally, simulation results and conclusions are provided.

2 Theoretical foundation

2.1 Two Dimension convolution

Convolution is a simple mathematical operation which is fundamental to many common image processing operators. Convolution provides a way of 'multiplying together' two arrays of numbers, generally of different sizes, but of the same dimensionality, to produce a third array of numbers of the same dimensionality. This can be used in image processing to implement operators whose output pixel values are simple linear combinations of certain input pixel values.

In an image processing context, one of the input arrays is normally just a graylevel image. The second array is usually much smaller, and is also two-dimensional (although it may be just a single pixel thick), and is known as the kernel.

Figure 1 shows an example image and kernel that we will use to illustrate convolution.

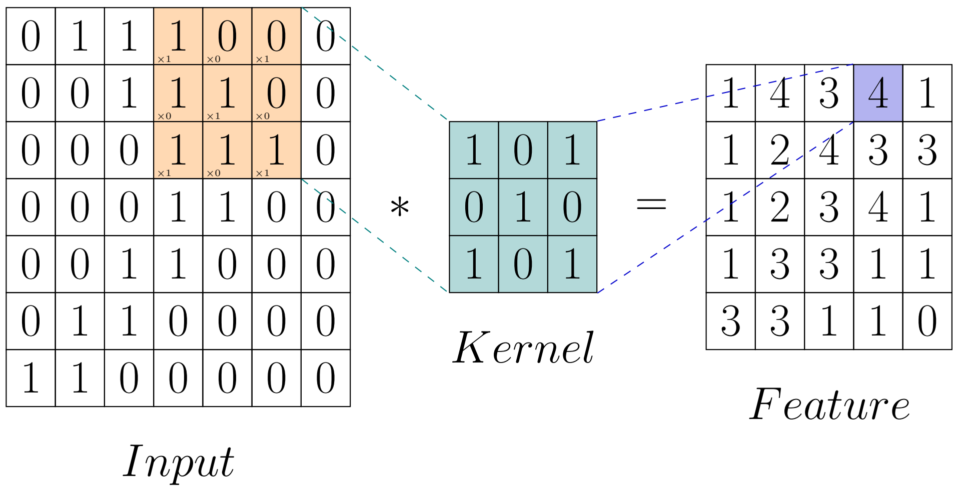


Figure 1: A simple 2D convolution illustration, by Janosh Riebesell (2021)

Mathematically we can write the convolution as:

$$O(i,j) = \sum_{k=1}^{m}\sum_{l=1}^{n} I(i+k-1, j+l-1)K(k,l)$$

- where $\boldsymbol{i}$ runs from $\boldsymbol{1}$ to $\boldsymbol{M - m + 1}$ and $\boldsymbol{j}$ runs from $\boldsymbol{1}$ to $\boldsymbol{N - n + 1}$

After the convolution stage, the feature matrix size can be calculate as:

$$O = [i - k] + 1$$

- where $\boldsymbol{i}$ is the size of input image, $\boldsymbol{k}$ is the size of kernel

2.2 Kernel & Filter

While research about this topic, lots of people misunderstanding about the term of kernel and filer. To clarify, Kernel is a matrix of weights which are multiplied with the input to exact the relevant features. The dimensions of the kernel matrix is *how the convolution gets it's name.* For example, in 2D convolutions, the kernels matrix is a 2D matrix.

A filter however is a concentration of multiple kernels, each of kernel assigned to a particular channel of the input image. Filters are always one dimension more than the kernels. For example, in 2D convolutions, filters are 3D matrices (which are essentially a concentration of 2D matrices.

2.3 Padding

Padding in Convolution refers to the addition of extra pixels around the edges of the input images or feature map. This process removes aggregation bias from the convolution operation. In other words, it makes sure every pixel gets considers.

In convolution stage, it is usually zero padding and this methodology also do it. Zero padding enables consistent receptive fields across different layers of the CNN. By padding zeros, the filters in each layer have access to the same number of surrounding pixels. Moreover, zero padding is straightforward to implement and computationally efficient. Adding zeros does not involve any complex calculations or additional memory requirements.

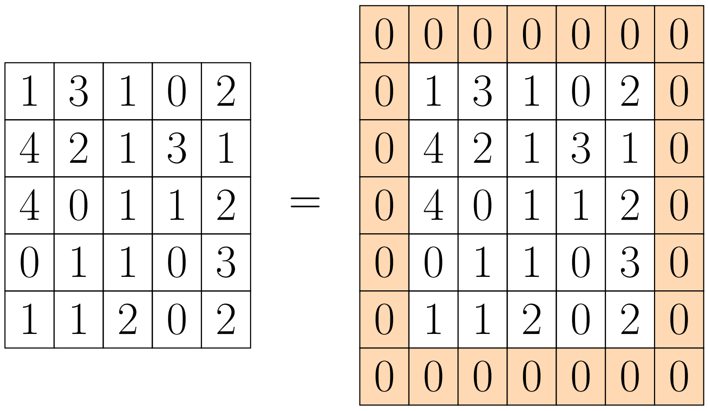


Figure 2: An illustration of zero padding in 2D convolution

2.4 Stride

Stride is a parameter that dictates the movement of the kernel, or filter, across the input data, such as an image. When performing a convolution operation, the stride determines how many units the filter shifts at each step. This shift can be horizontal, vertical, or both, depending on the stride's configuration.

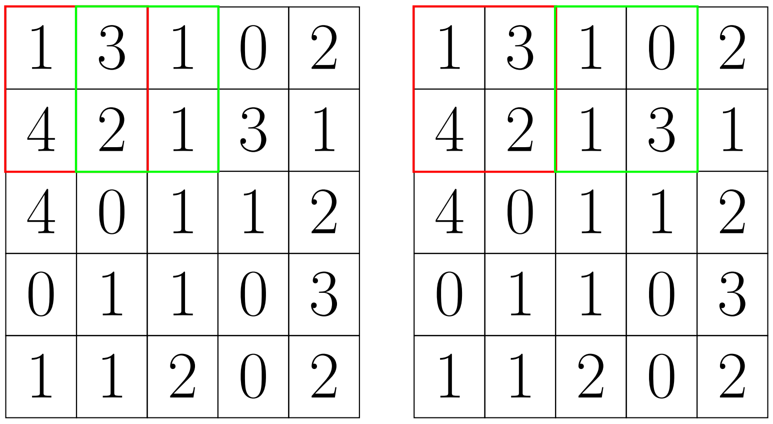


Figure 3: On the left, Stride = 1. On the right, Stride = 2

The stride can reduce the resolution of the output, for example reducing the height and width of the output to only ***1/n*** of the height and width of the input for n > 1 . In our implementation, the stride is set as default, ***stride = 1***.

2.5 Pooling

The Pooling operators consist of a fixed-shape window that is slid over all regions in the input according to its stride. This layer combines each group of the outputs of the previous layer into a matrix. There are two common variations of pooling operations: average pooling and max pooling. An average pooling layer averages its input values by taking the mean of them. On the other hand, max pooling takes the biggest value.

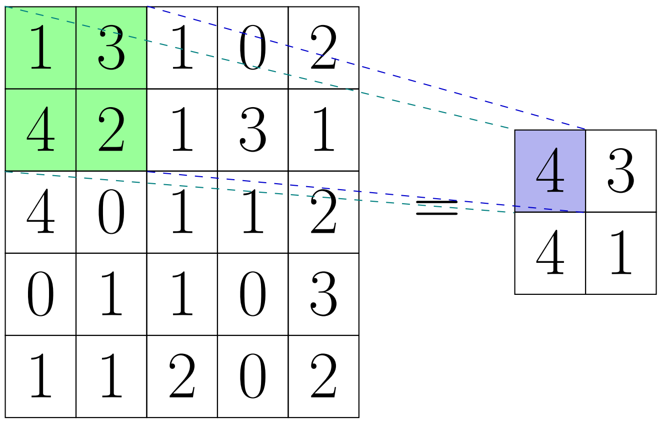


Figure 4: Max pooling with stride = 2

2.6 Input data type

In the scope of this project, to understand logic and verify correctness of the convolution algorithm, we use an input matrix with maximum size is 100x100 and minimum is 28x28, each element has positive integer value from 0-15, which is 4 bits in binary (we just take 4 bits in order to handle overflow bits in the final result - up to 8 bits, to easily observe and verify the correctness of convolution logic in Verilog); kernel matrix has the fixed size 3x3 and values [(0,-1,0); (-1,4,-1); (0;-1;0)] - this is Laplacian kernel of Laplacian of Gaussian (LoG) algorithm, one of the common kernels for edge detection in image processing. Besides this, there are other algorithms which are widely used, such as: Sobel Operator, Canny Edge Detector, Prewitt Operator, etc. However, the scope and topic of this project does not delve into the implementation of image processing algorithms, it mainly focuses on the logic of convolution operations, so we just take the Laplacian kernel for convolving with input matrix, then continue to use max pool layer to reduce size of matrix, get the final output and prove the correctness of convolution logic by converting output to a grayscale image. If edge of object in the original image can be observed, that means those convolution steps have been correct.

3 Architecture design

3.1 Workflow

In this section, the design of the architecture is described in detail. The entire process is divided into three stages, pre-processing, processing and post-processing.

The pre-processing consist of generating input image and kernel using Python script. The input and kernel is generated as an INT8 data type, convert to 4-binary-bit, then write to *image.txt* and *kernel.txt* for file reading in the next stages. During this stage, the Python script also generate the result of convolution and max pooling in other to archive the result as the golden model for later comparison with the output in the simulation. In addition, we use Keras for computing the convolution stage.

In the second stage, we FPGA simulation for the computation of 2D convolution. Specialized sub-modules are designed using Hardware Description Language (HDL) to optimize the process. FPGAs offer parallelism and reconfigurability, making them ideal for acceleration. These sub-modules, written in Verilog, handle tasks such as input data loading, filter application, and result accumulation. The FPGA simulation phase allows for thorough testing and optimization before deploying the hardware, ensuring an efficient and reliable implementation of 2D convolution.

Finally, the during the post-processing, simulation result is compared with the golden model. Clock cycles, clock speed and memory accounted are weight to conclude how the efficient of the CNN accelerator is.

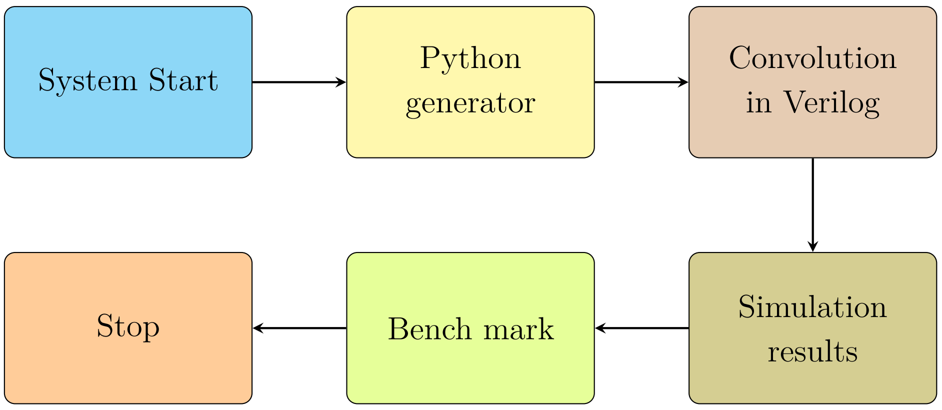


Figure 5: State diagram that describes the transitions of events for the convolution.

3.2 Convolution block diagram in Verilog

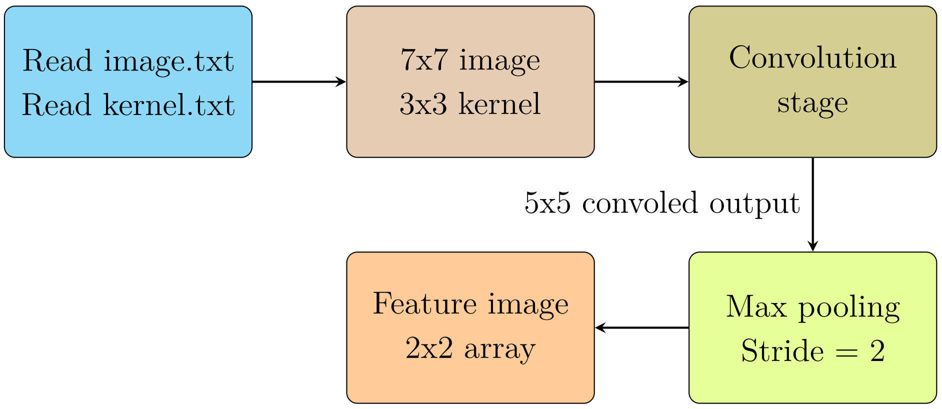


Figure 6: Block that describes stages in HDL convolution.

The figure 6 illustrates that this model test the correctness of our system. The input image and kernel we generate randomly by python, write to text file: **image.txt** and **kernel.txt** for convolution on FPGA. After getting the input data, the data will get through convolution and max pooling layer and write out an **output.txt** text file for comparing with python convolution model. Through this testing model, we can conclude that out model is precise and ready for larger input image and kernel.

3.2.1 Convolution module

The Figure 7 shows an example of workflow in HDL algorithm. After get the result from multiplication state (image * kernel), 3 data output will go through 3 adder stage, then add those result 1 more time to achieve 1 pixel of the feature map. Repeat this step to achieve the whole feature map after convolution.
Here is the code of module convolution:

```verilog
module ConvolutionStage1 (
    input [3:0] a,
    input [3:0] b,
    input clk,
    input enable,
    output wire done,
    output reg [7:0] prod
);

    reg [3:0] b_reg;
    reg [3:0] count;

    always @(posedge clk) begin
        prod = 0;
        b_reg = b;
        count = 4'b0100;
        if (((a!=0) && (b!=0)) && enable)
        begin
```

```
            if(b_reg == 4'b1111) begin
                prod = {4'b1111, ~(a)} + 8'b00000001;
            end
            else begin
                while (count) begin
                    prod = {(({4{b_reg[0]}} & a) + prod[7:4]), prod
    [3:1]};
                    b_reg = b_reg >> 1;
                    count = count - 1;
                end
            end
        end
    end

    assign done = 1;
endmodule
```

Listing 1: Convolution module

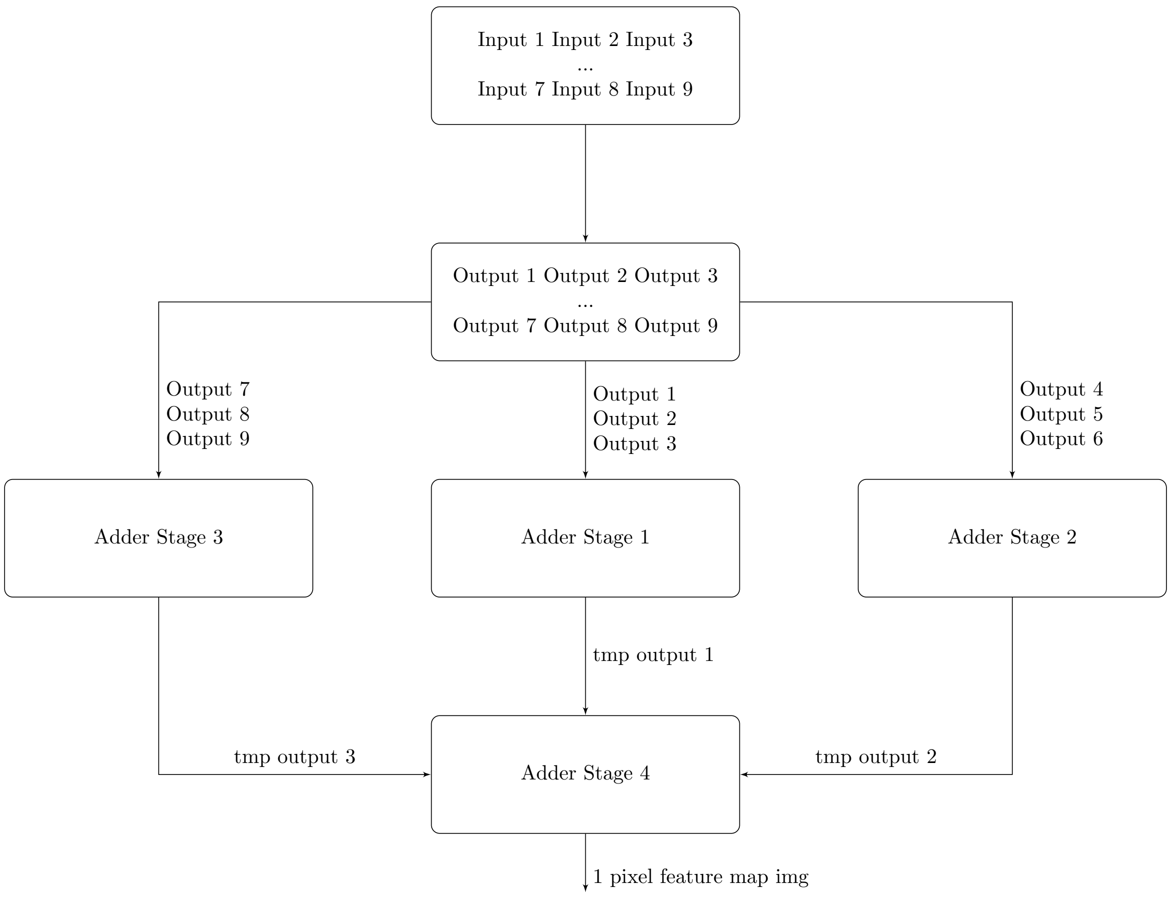


Figure 7: Workflow in Convolution Block

Input register ”a” is the element of image matrix, ”b” is the element of kernel matrix. They will be multiplied and the result is stored in output register ”prod”. Because our edge detection kernel has negative value, we have to check if ”b” reg equal to 4’b1111 (-1), to obtain the result when multiply positive value of ”a” with negative value of ”b” (-1), invert all bits of ”a”, then add with 1 to obtain the negative value of ”a”.

On the other hand, if ”b” reg is not negative value, we set ”count” variable to 4, then perform multiplication step by step while ”count” is greater than 0. For each iteration: takes the value of ”b_reg[0]”, replicates it four times using ”{4{b_reg[0]}}”, and then bitwise ANDs the result with ”a”. Then this part ”(({4{b_reg[0]}} & a) + prod[7:4])” adds the result from the previous step to the four bits (7 to 4) of the ”prod” variable. Continue to concatenate the previous result obtained with 3 bits (3 to 1) of ”prod” variable. Finally, shift ”b_reg” to the right by 1 bit and decrease the ”count” variable by 1.

The result ”prod” variable has 8 bits after multiplication complete, because of when multiplying 2 value of 4 bits, the maximum total of bits in the result is equal to the sum of the bit lengths of the multiplicands (4+4 = 8 bits).

3.2.2 Max Pooling module

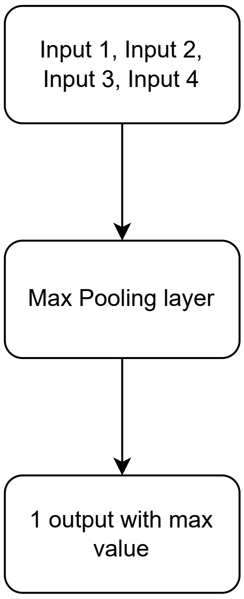


Figure 8: Workflow in Max Pool Block

In this stage, after obtaining all the output from convolution stage, continue to reduce the size of matrix by applying max pool layer.
Here is the code of module max pool:

```
module maxPooling(
    input clk,
    input enable,
    input [7:0] input1,
    input [7:0] input2,
    input [7:0] input3,
    input [7:0] input4,
    output reg [7:0] output1,
    output reg done
    );
    reg [7:0] initialMax = 8'b10000000;
    always @ (posedge clk) begin
        if(enable) begin
            if($signed(initialMax) < $signed(input1)) begin
                if($signed(input2) < $signed(input1)) begin
                    if($signed(input3) < $signed(input1)) begin
                        if($signed(input4) < $signed(input1)) begin
                            output1 <= input1;
                            done <= 1;
                        end
                        else begin
                            output1 <= input4;
                            done <= 1;
                        end
                    end
                    else begin
                        if($signed(input3) < $signed(input4)) begin
                            output1 <= input4;
                            done <= 1;
                        end
                        else begin
                            output1 <= input3;
                            done <= 1;
                        end
                    end
                end
                else begin
                    if($signed(input3) < $signed(input2)) begin
                        if($signed(input4) < $signed(input2)) begin
                            output1 <= input2;
                            done <= 1;
                        end
                        else begin
                            output1 <= input4;
                            done <= 1;
                        end
                    end
                    else begin
                        if($signed(input3) < $signed(input4)) begin
                            output1 <= input4;
                            done <= 1;
                        end
                        else begin
                            output1 <= input3;
                            done <= 1;
                        end
                    end
                end
            end
            else begin
                output1 <= initialMax;
                done <= 1;
            end
        end
        else begin
            output1 <= 0;
            done <= 0;
        end
    end
endmodule
```

Listing 2: Max Pooling module

Take 4 corresponding inputs from result matrix after convolutional layer, initiate a max variable ”initialMax”, then compare each input with this variable, if the input has value greater than ”initialMax” variable, assign value of that input to ”output1”. Continue to compare the next input's value with previous input until there are no more inputs left. The final value of ”output1” will be the largest among 4.

4 Simulation results

4.1 Edge detection

For testing the efficiency and correctness of this 2D convolution module, we come up with a testing ***of a 100*100 pixel*** image through ***an edge detection kernel***.

Firstly, find a 1:1 image and scale it to size less than or equal to 100x100, but we recommend the size should between 80x80 to 100x100 in order to see the result more clearly, because if we take the input matrix to small, the convolution and max pool layer will continue to reduce size of image to a much smaller result, which is very difficult to observe the edge detection. We will take the image of maximum size 100x100 to illustrate.
Convert original RGB image to Grayscale image

After get the grayscale, becasue the original grayscale image has values from 0 to 255, we need to convert this image to array and scale the array values to range from 0 to 15, due to 4 bits input data type of convolution model.

0	-1	0
-1	4	-1
0	-1	0

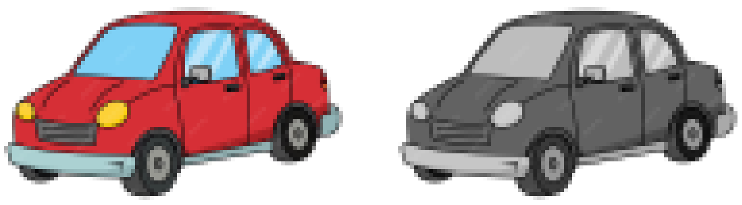


Figure 9: RGB 100x100 image convert to Grayscale imgae

Figure 10: Edge detection kernel

Run the convolution algorithm in verilog and we get the resutl after convolution through the above kernel. After that, we continue to use max pool layer to reduced the size of output from convolutional layer. Then, we get the final output, however the grayscale image does not have negative value, we must convert those output elements that have negative value to value 0. Finally, with the help of Python, we draw that image again from the final output obtained:

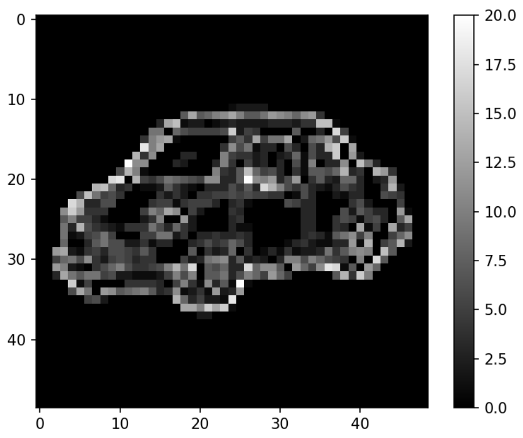
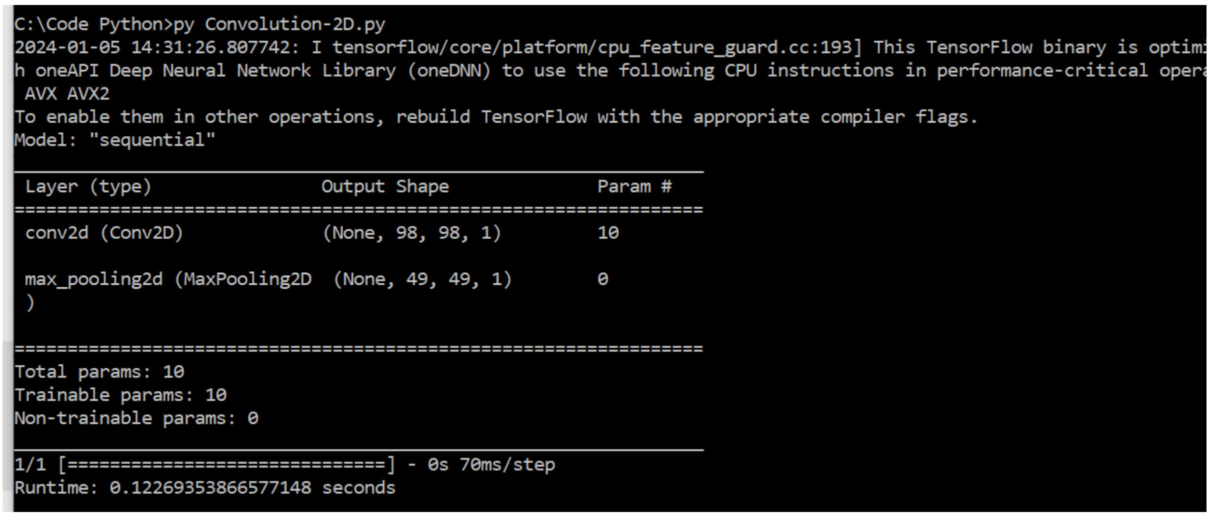


Figure 11: Result image

It can be seen that the car's edge is detect. Moreover, in order to prove for correctness of the output generated by verilog, we also compare the output file with the one that generated by python, and they are the same (the code to compare 2 files is in github link)

4.2 Time and Power consumption

Comparing the Python convolution model with the Keras model, it can be seen that the time consumption for running the convolution and max pool layer is around 0.12 seconds, while processing on an FPGA with our algorithm takes about 0.0026 seconds for simulation. Thus, the efficiency in time savings is around 50 times.

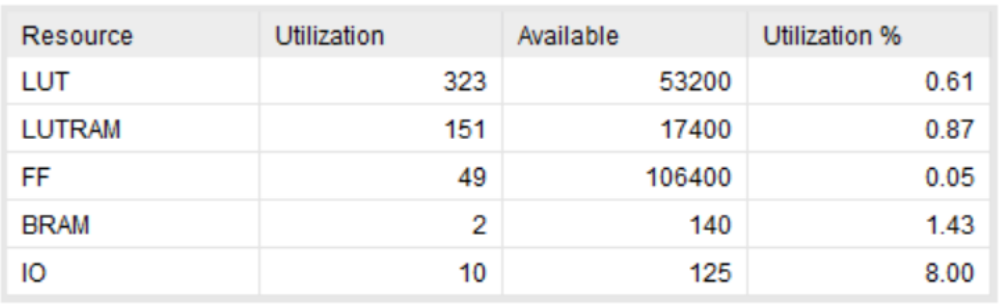


```
C:\Code Python>py Convolution-2D.py
2024-01-05 14:31:26.807742: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optim
h oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical opera
 AVX AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 98, 98, 1)         10

 max_pooling2d (MaxPooling2D  (None, 49, 49, 1)        0
 )

=================================================================
Total params: 10
Trainable params: 10
Non-trainable params: 0
_________________________________________________________________
1/1 [==============================] - 0s 70ms/step
Runtime: 0.12269353866577148 seconds
```

Figure 12: Python runtime

```verilog
`timescale 1ns / 1ps
module read_txt2_tb;

    reg clk;
    reg rst;
    wire [7:0] result;
    always #5 clk = ~clk;
    initial
    begin
        clk = 0;
        rst = 0;

    #10
        rst = 1;
    #10
        rst = 0;
    #2617100
    $stop;
    end

    read_txt2 inst1(.clk(clk), .rst(rst), .result(result));


endmodule
```

Listing 3: Test bench for simulation



Resource	Utilization	Available	Utilization %
LUT	323	53200	0.61
LUTRAM	151	17400	0.87
FF	49	106400	0.05
BRAM	2	140	1.43
IO	10	125	8.00

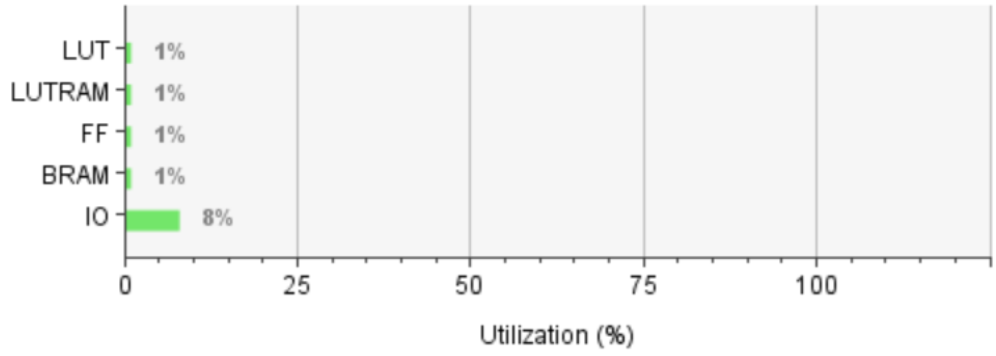
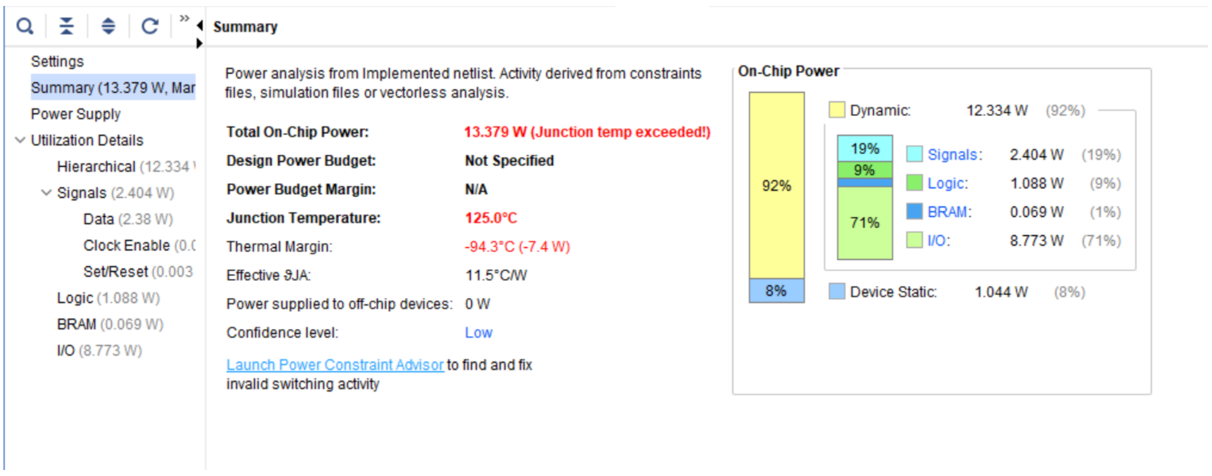


Figure 13: Resource utilization

However, our algorithm also has some limitations, it causes high power and temperature. As can be seen from the power report, the total power on-chip is around 13000W, which has exceeded average power, and this can lead to reliability problems. In order to improve this problem in the future, code should be organized in pipelining to reduce the number of flip-flops and reduce clock frequency for specific usage in practice.



Summary

Settings
Summary (13.379 W, Mar
Power Supply
Utilization Details
Hierarchical (12.334 W
Signals (2.404 W)
Data (2.38 W)
Clock Enable (0.0
Set/Reset (0.003
Logic (1.088 W)
BRAM (0.069 W)
I/O (8.773 W)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	13.379 W (Junction temp exceeded!)
Design Power Budget:	Not Specified
Power Budget Margin:	N/A
Junction Temperature:	125.0°C
Thermal Margin:	-94.3°C (-7.4 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	12.334 W	(92%)	92%
Signals:	2.404 W	(19%)	19%
Logic:	1.088 W	(9%)	9%
BRAM:	0.069 W	(1%)	
I/O:	8.773 W	(71%)	71%
Device Static:	1.044 W	(8%)	8%

Figure 14: Power report

5 Conclusion & Future work

In our current project, we have successfully developed and implemented a 2D CNN accelerator on an FPGA, demonstrating improved performance in terms of speed and efficiency for image convolution tasks. Building upon this foundation, an intriguing avenue for future investigation is the application of our model to lane detection in large video datasets.

Lane detection is a critical aspect of various computer vision applications, especially in the field of autonomous driving. By integrating our image convolution algorithm as a subsystem, we aim to enhance the performance of lane detection tasks within the context of extensive video streams.

In addition to the successful development of our 2D CNN accelerator on an FPGA and its application to lane detection in large videos, we recognize the critical importance of further optimizing the code pipeline. This optimization is aimed at achieving two key objectives: minimizing power consumption and reducing processing time.

6 Source code

Here is the github link for our source code (verilog and python): `https://github.com/hungpham1406/2D_CNN`

7 References

1. Akinola-Alli, G. (2023, September 14). Convolution in image processing. Medium. https://medium.com/@gakinolaalli/convolution-in-image-processing-6d382da5048e

2. Lai, Liangzhen & Suda, Naveen & Chandra, Vikas. (2018). CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs.

3. Henrik Ström. (2016). A Parallel FPGA Implementation of Image Convolution. Master of Science Thesis in Electrical Engineering. https://www.diva-portal.org/smash/get/diva2:930724/FULLTEXT01.pdf

4. Ma, X., Zhao, R., & Zhou, J. (2019). Convolutional neural network (CNN) accelerator chip design. 2019 IEEE 13th International Conference on Anti-counterfeiting, Security, and Identification (ASID). https://doi.org/10.1109/icasid.2019.8925182

5. Sharma, N., Jain, V., & Mishra, A. (2018). An analysis of Convolutional neural networks for image classification. Procedia Computer Science, 132, 377-384. https://doi.org/10.1016/j.procs.2018.05.198