Logic Design Laboratory: Final Project

〇一二三四五六七八九：1A2B game

黃恩明 109062221

楊東翰 109062273

January 16, 2022

Abstract

We develop a DNN model which can recognize Chinese numerals and ported it on FPGA. To reduce the model size, the parameters of the model are quantized from FP32 into INT8. The performance drops only 1% in our validation set. The deep neural network processing element is parameterized, also designed to support pipelining and parallel computing for enhanced hardware utilization. In result, the inference time of our implementation on FPGA is 25x faster than the previous work done by 刁彥斌 and 孫偉芳 [1].

1 Introduction

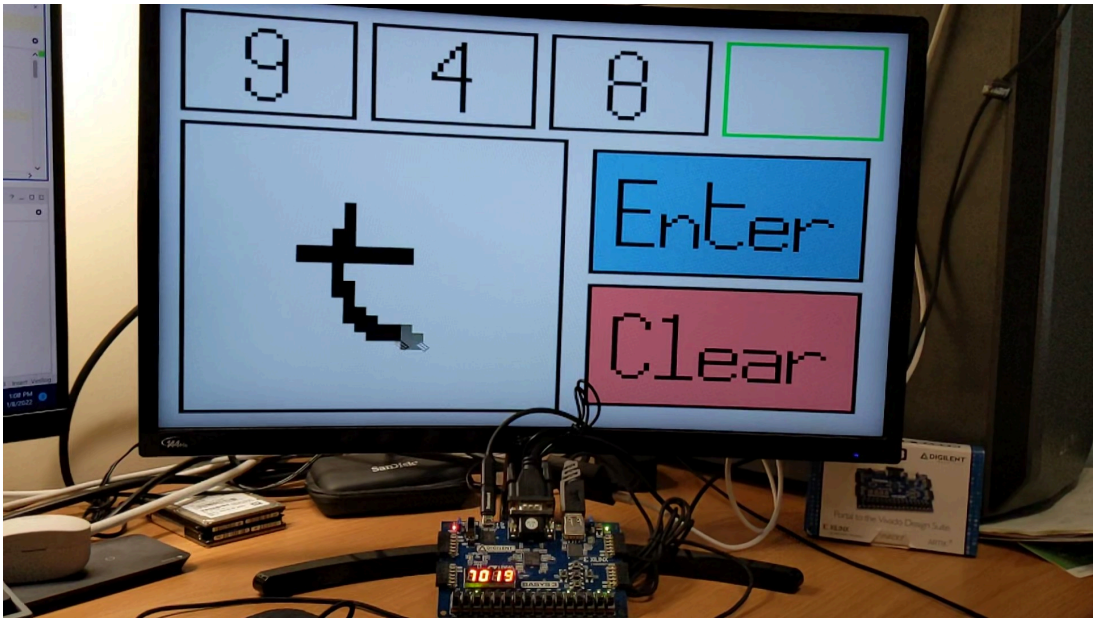


Figure 1: Gameplay

” 〇一二三四五六七八九：1A2B game” is a game based on Basys3 Artix-7 FPGA. The main feature is that it has a **canvas painter** with the resolution of 32x32 and a **deep neural network** which recognizes the Chinese numerals drawn on canvas. The game itself is same as lab4′ s 1a2b game.

2 Specification

2.1 Control and I/Os

- Down button: Reset
- 7-Segment display: Deep neural network output
- Mouse Left: Draw or click button
- VGA: Gameplay display
- LEDs: 1A2B answer

2.2 Block Diagram

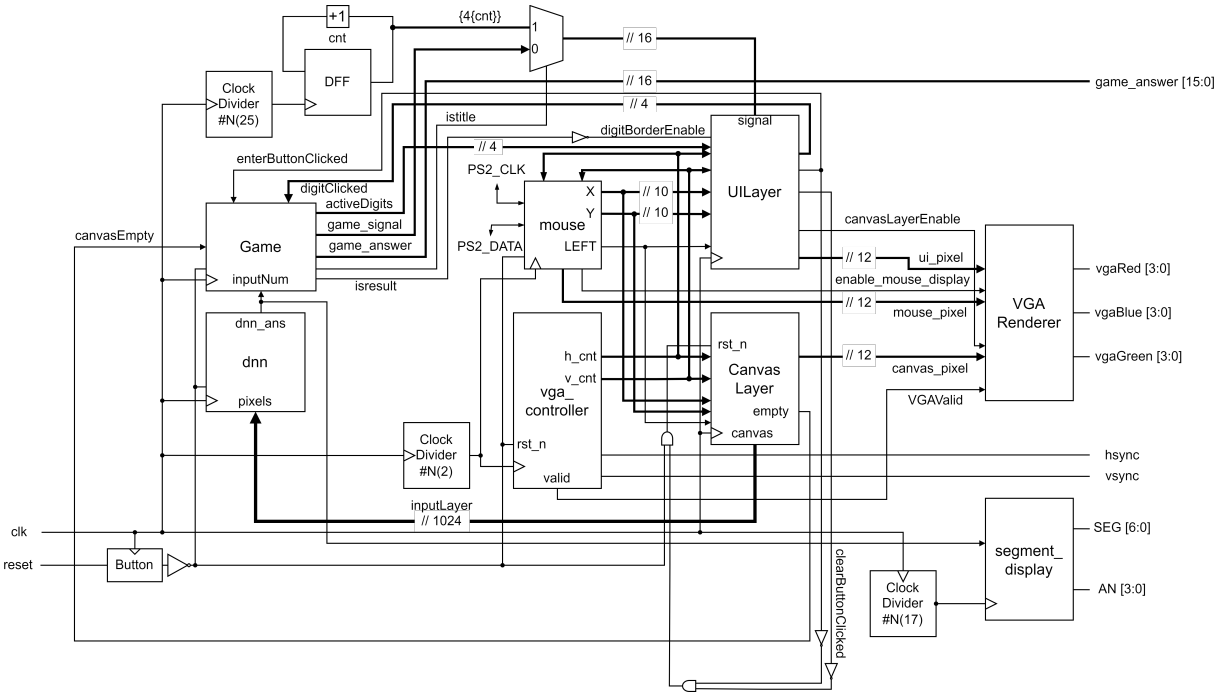


Figure 2: Block Diagram - top

2.3 Game Play

1. **INIT**

   Menu page, digits is counting between 0, 1, 2..., 9, a, b infinitely. Click ”Enter” button to start game.

2. **GUESS**

   Click the digit with the mouse to specify which digit you would like to input.

   Draw in the canvas, click ”Enter”, and the number would be stored in your selected digit.

   If ”Enter” is clicked without any active (selected) digits, the game would enter **SHOW** state.

3. **SHOW**

   Show the result of the guess. In this state, the border of digits would disappear.

2.4 User Interface

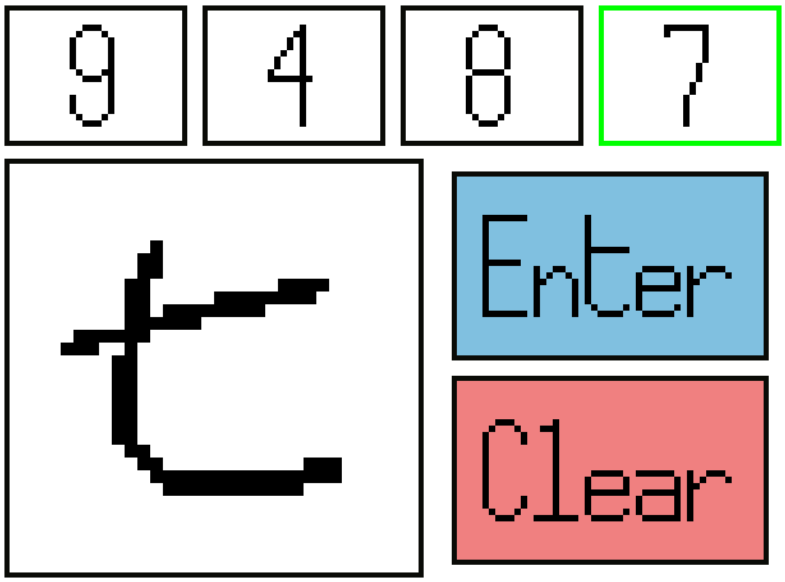


Figure 3: User Interface Layout

- Screen: 640 * 480
- Padding: 16px
- Components
  - Character
    - Size: 40 * 80 px
    - Resolution: 8 * 16
  - Digit
    - Size: 143 * 110 px (including 4px border)
    - Input Signals
      - `digit [3:0]`
      - `active` (currently selected digit)
      - `digitBorderEnable`
  - UIButton
    - Size: 248 * 148 px (including 4px border)
  - Canvas
    - Size: 328 * 328 px (including 4px border)
    - Drawing Area Size: 320 * 320 px
    - Resolution: 32 * 32

3 Motivation

We discovered that previous work has been done by 刁彥斌 and 孫偉芳 [1]. Their main contribution is

1. recognize Arabic numerals ($0 \sim 9$)
2. dynamic fixed point which reduces the parameters from 32-bit float to 16-bit integer
3. real time inferencing in just 1ms

. The implementation of their neural network is simple, which is similar to the sequential code of writing it as a software program. No hardware performance enhancement is applied. Therefore, we believe this isn't the best. Our target in this project is to recognize Chinese numerals, faster inference time, and modularize and parameterize layers for different varieties of model.

4 User Interface

4.1 Layers

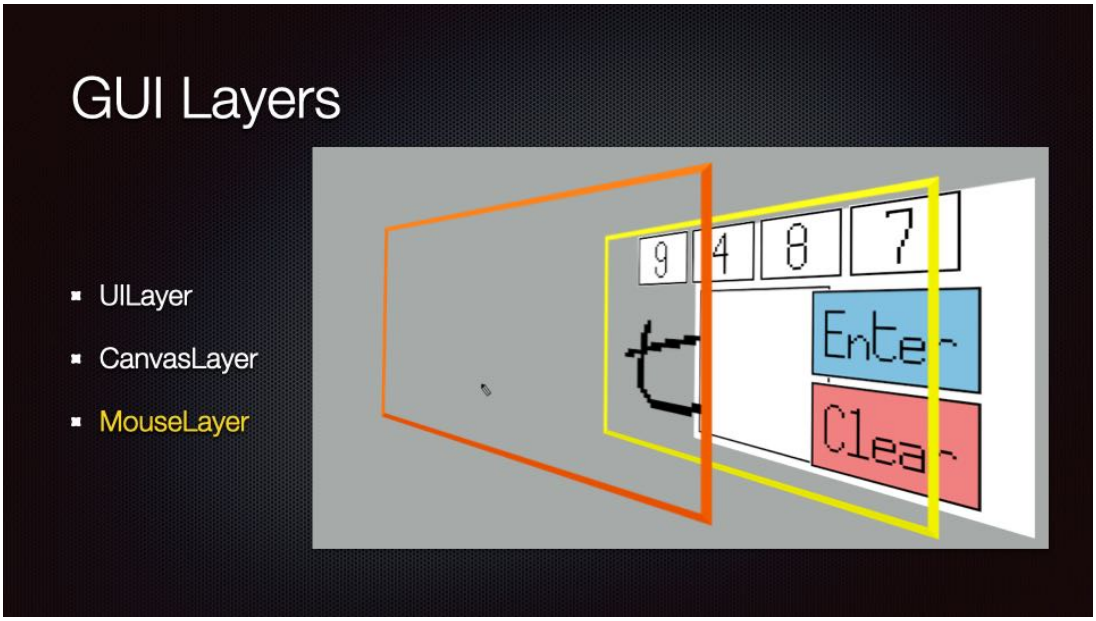


Figure 4: UI Layers

We divided UI into three layers to seperate the logic with different functionalities instead of creating a huge module to handle everything.

The layer at the top is MouseLayer, followed by CanvasLayer, and the bottom one is UILayer.

4.1.1 UILayer

This layer consists of all static components whose position won't change on the screen. The only thing it cares about is what to show on screen. In addition, it propagates the click event to the other layers. In other words, this layer is repsonsible for presentation and user interaction with UI, but not the game logic.

4.1.2 CanvasLayer

What the user have drawn would be presented in this layer instead of UILayer to keep it simple. This layer not just render the canvas, but also deal with the drawing event.

4.1.3 MouseLayer

Since the pointer would change its position frequently, it needs an independent layer to handle it. It outputs `enable_mouse_display` signal to VGARenderer, which means that it should show the pointer pixel at this position. The source code of this layer are derived from Lab 6 demo code.

4.2 Components

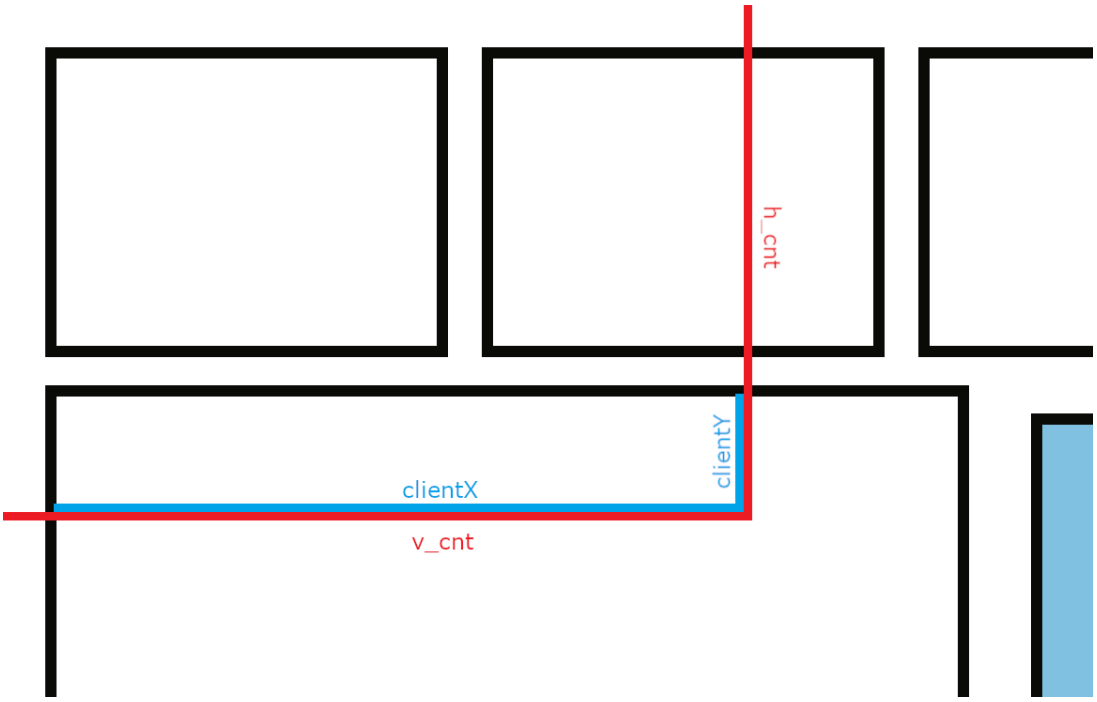


Figure 5: clientX/Y Visualization

There are signals called `clientX/Y` and `mouseClientX/Y` in each component, in which "client" stands for the position relative to the component's origin. By making use of these signals, we can reuse these components with ease. We can change the `originX/Y` parameter without other modification to instantiate same component at different position, instead of creating serveral components with minor difference.

What's more, `canvasClientX/Y` (or `characterClientX/Y`) is introduced as the position relative to the origin which has been converted to the component's unit. The calculation is simple, just dividing the `clientX/Y` by `unitSize` (= `width`/`canvasWidth`). For example, the drawing area is 320 * 320 px, and the canvas size is 32 * 32, so `unitSize` $= 320/32 = 10$. Therefore, if `clientX` is 185, then `canvasClientX` is 18.

5 Deep Neural Network

The model is developed and trained by TensorFlow. We designed a prototype website (`https://t510599.github.io/handdrawn-1a2b/`) to test whether the model is usable or not. By using TensorFlow.js, we are able to convert the model from python to web.

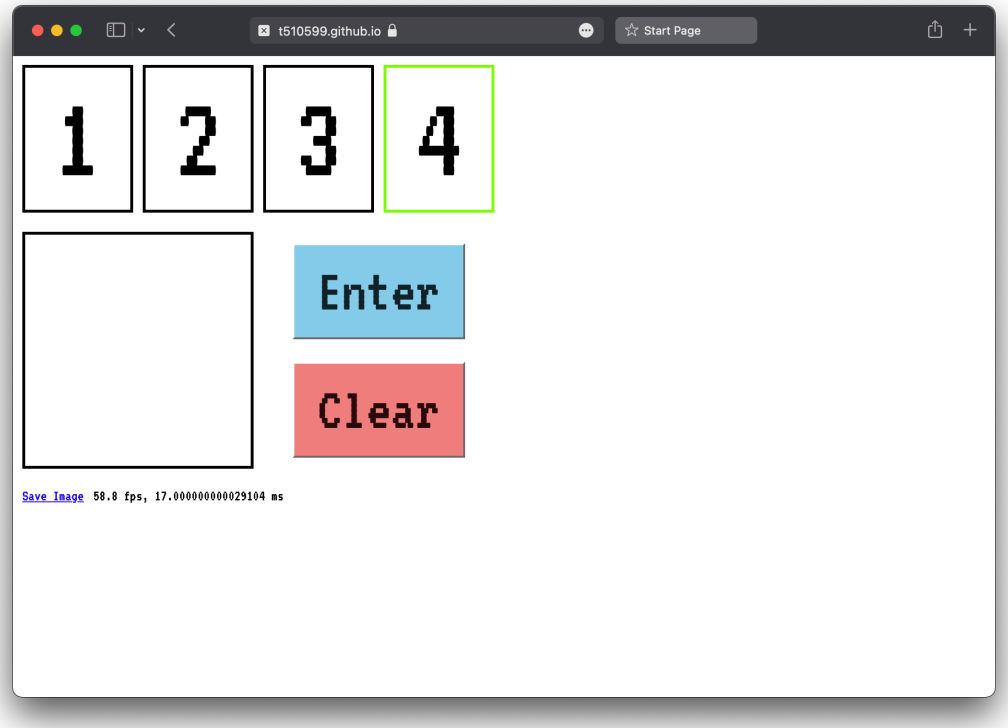


Figure 6: Screenshot of the prototype website

5.1 Training

5.1.1 Training Data

Since there's no existing Chinese numerals' data set, we developed a website (`https://t510599.github.io/handdrawn-1a2b/canvas.html`) for us to draw the numerals in an easy and convenient way. We can draw a bunch of images and pack it into a zip file.

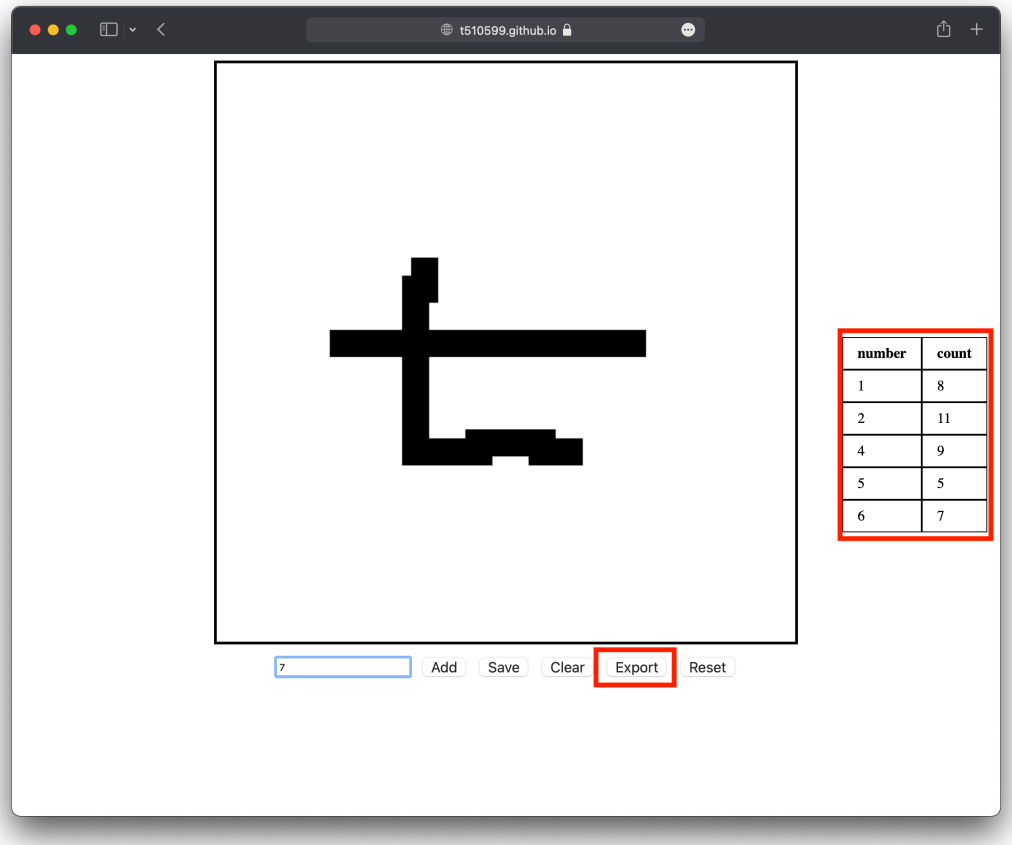


Figure 7: Drawing tool

We drew 1200 images in total. The data is splited into two groups. Nine out of ten belongs to the training set, while others are the validation set.

5.1.2 Training

Figure 8 is our model, which has three fully-connected layers. We tried two-layer model first, but ended up in poor accuracy since it has difficulties on distinguishing ” 三” and ” 五”. With using three layers, our validation accuracy increased to 85%. However, it is still not a reliable model when we tested it on our website, because the amount of our data is less. In the end, with the help of NVIDIA deep learning online course, we utilize **Keras.ImageDataGenerator** to produce rotated , resized and shifted images to generate more training data. This increased our validation accuracy from 85% to 99%.

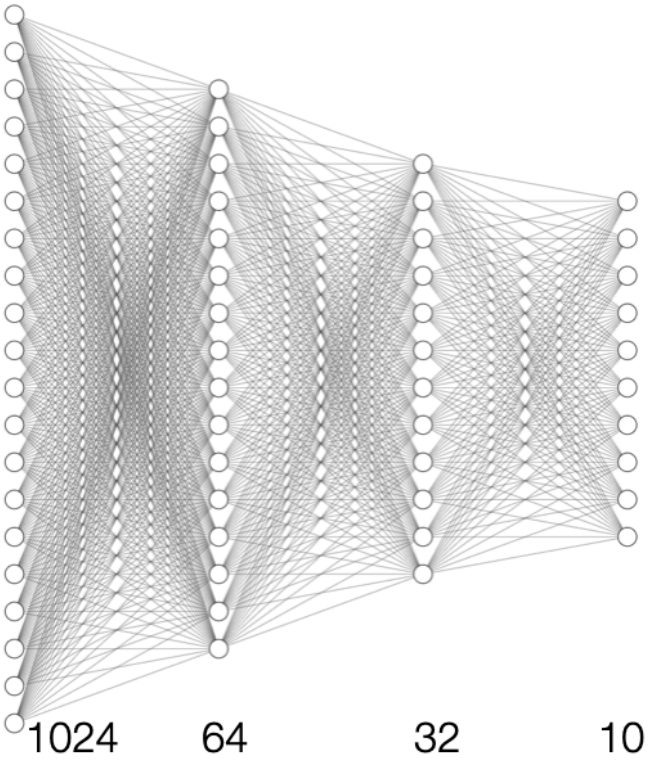


Figure 8: Model

5.2 Inferencing on FPGA

To excute the model on FPGA, we reduce the parameter size to fit it in the onboard fast block RAM, and we design a parameterized process element which only calculates a dense layer. This enables us to explore pipelining and parallel computing in a simple way. In the end, the inferencing time of our design is only 41µs, which is 25x faster than the previous work.

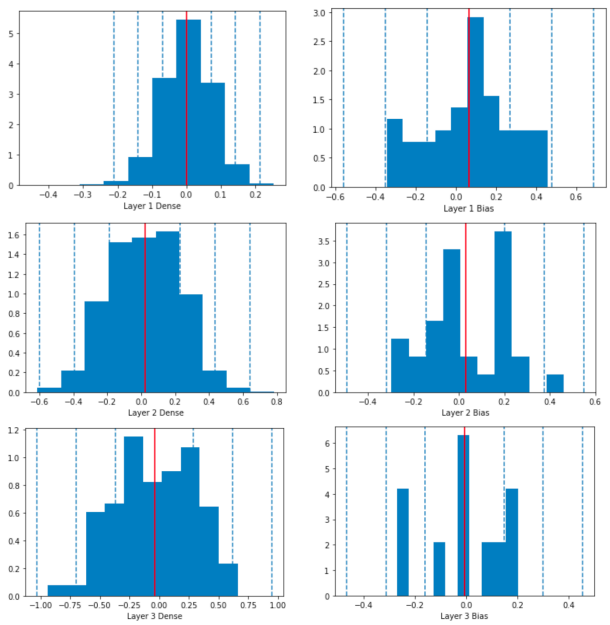
5.2.1 Parameter Quantization

Figure 9 is the model summary produced by TensorFlow. It contains 68,010 32-bit floating points so that the total size is 2.2 Mb. However, the memory of Basys 3 is only 1.8 Mb.

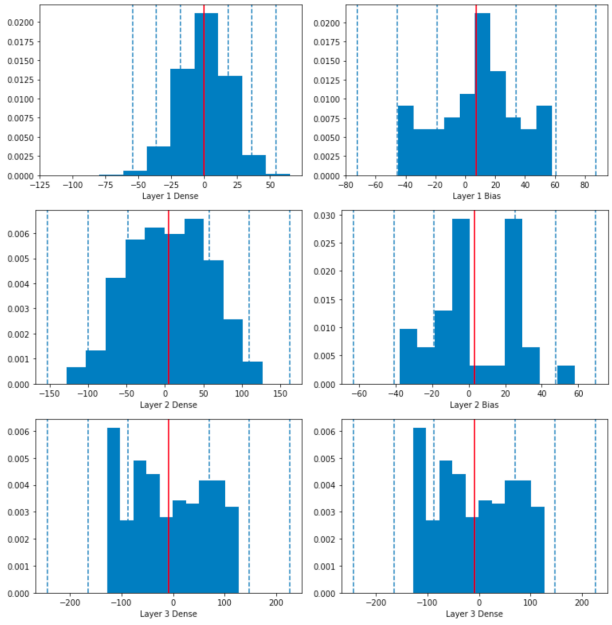
```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 reshape (Reshape)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 re_lu (ReLU)                (None, 64)                0

 dense_1 (Dense)             (None, 32)                2080

 re_lu_1 (ReLU)              (None, 32)                0

 dense_2 (Dense)             (None, 10)                330

=================================================================
Total params: 68,010
Trainable params: 68,010
Non-trainable params: 0
_________________________________________________________________
```

Figure 9: Model summary

To overcome this problem, the parameters are quantized from FP32 to INT8, where the size is only 0.5Mb. We scale the parameters linearly from $[-1, 1]$ to $[-128, 127]$. Figure 10a and Figure 10b are the distributions of the parameters. The accuracy drops from 99% to 98% in our validation set (118/120 images are correct).



(a) Parameter distribution



(b) Quantized parameter distribution

5.2.2 Hardware Module

First, we design a processing element, or PE, to calculate a dense layer with parameterized setting. It can compute any size of $n * m + m$ dense layer ($\vec{y} = \max(\vec{x}A + \vec{b}, \vec{0})$, where $\vec{x}$ is a $1 \times n$-row vector and A is a $n \times m$ matrix). The answer is stored in a series of flip-flop, while the parameters are read from block RAM. Figure 11 is the state diagram of the PE. The finish signal can be used to trigger the calculation of next PE. To make the design simple, n is restricted to be a power of 2.

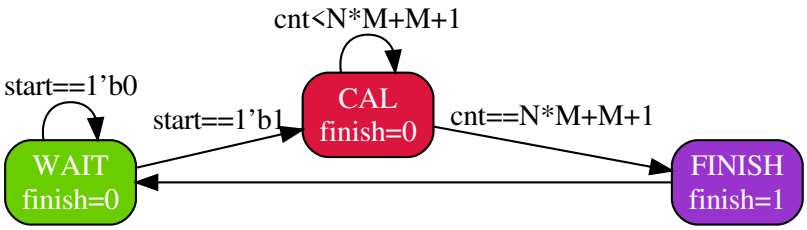
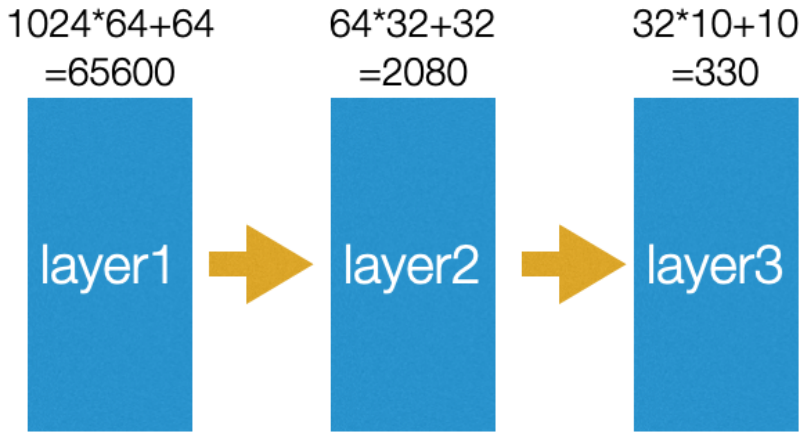


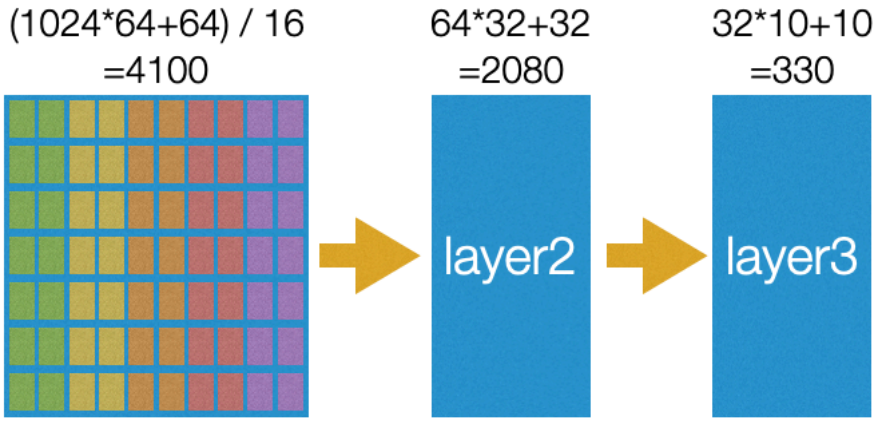
Figure 11: State diagram of the process element

The parameterized design enables two performance optimizations, we discuss them as follows:

1. Pipelining: Each PE is responsible for a dense layer, so that computational throughput can be increased by pipelining. However, first layer becomes the computing bottleneck since its computing time is much greater than others.



2. Parallel computing: We break the matrix of the first layer from a 1024×64 matrix into sixteen 1024×4 matrices. We can calculate the submatrix in parallel to reduce computation time because there is no data dependency between columns in a matrix multiplication. Note that the memory block width is increased from 8-bit to $8 \cdot 16 = 128$-bit, while the maximum width is 256-bit. In result, the average inferencing time is 41 µs (1 clock = 10ns).



6 Experiments

We compare the runtime of the model excuted on CPU with our FPGA implementation. The testing platform are Intel i9-10900 desktop computer and Apple M1 MacBook Air. The single-threaded program is

written in C++ and compiled with two different optimization flag: `-O0` and `-O2`. We have two implementations of the program, one is to store the right matrix in row order, which follows the logic of accessing M_{ij} as `M[i][j]`. Another is in column order, which is more friendly to cache because the right matrix is read in column order in the matrix multiplication.

Platform	Specification	Compiler	Optimization
Basys 3	100 MHz, CLB (LUT, FF and MUX), AXI4-Lite	-	-
Intel i9-10900	4.6 GHz, x86_64, DDR4 3600MHz memory	g++ -O0 g++ -O2	None Continuous memory access
Apple M1	Firestorm: 3.6 GHz, ARMv8, LPDDR4 4266MHz unified memory	clang++ -O0 clang++ -O2	None Continuous memory access

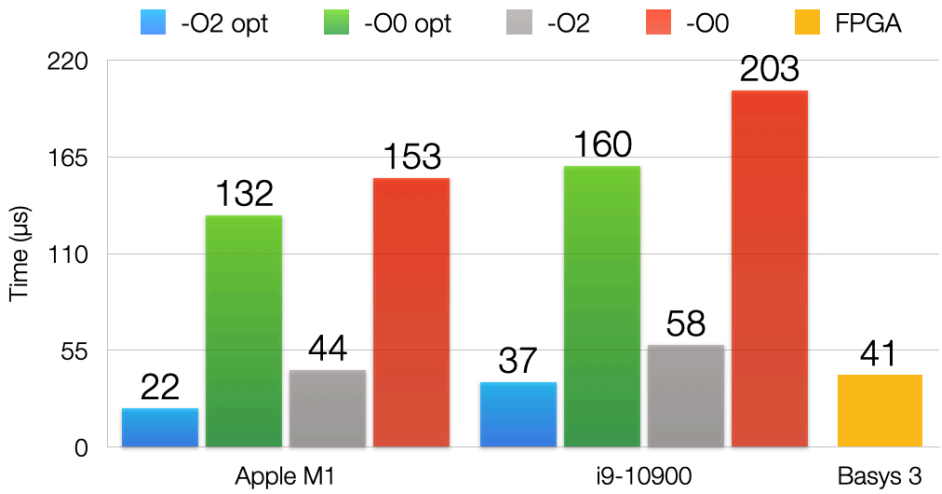


Figure 12: Experiment results

Figure 12 is the experiment results, it is shown that our FPGA implementation offer similar performance with significantly higher power efficency. The energy consumption reported in VIVADO design tool is only 0.5 Watt.

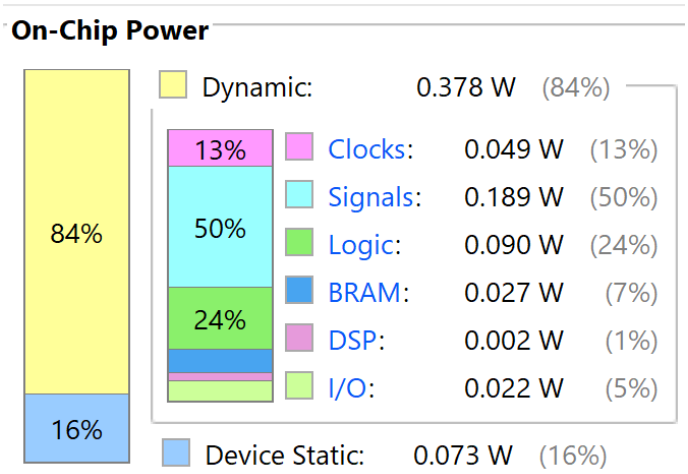


Figure 13: Power reported in Vivado

7 Future Work

We can try the CNN model, which has fewer parameters. Besides, binary quantizing the parameters is another effective way to reduce the model size. Duy Thanh Nguyen et al presented an effective way of FPGA Implementation of CNN [2].

8 Conclusion

We put deep learning, pipelining and parallel computing into practice. Furthermore, we utilize peripheral components such as mouse and VGA. This implies that we integrate all the knowledge teached in this course together and present it in our final project. Our deep neural network is larger than previous work, but it is 25x faster and it can recognize Chinese numerals. In the experiment, we shown that FPGAs are not only suitable for verifying circuit but also offer high efficiency and low energy consumption in specific tasks.

9 Contribution

- 黃恩明
  - Deep Neural Network
  - 1A2B Game
  - Drawing Training Data
- 楊東翰
  - Web prototype
  - User Interface
  - Drawing Training Data

References

[1] 刁彥斌 and 孫偉芳. *全知全能智多星可愛畫布? The Omniscient and Omnipotent Smartie, Cutie Canvas?* Last accessed 15 January 2022. 2018. URL: `https://github.com/j3soon/Handwritten-Digit-Recognition-Painter`.

[2] Duy Thanh Nguyen et al. "A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.8 (2019), pp. 1861–1873. DOI: `10.1109/TVLSI.2019.2905242`.