Final Report

Gong Yutao
2022231081

An Lihua
2022131001

Xin Yue
2020231073

***Abstract*—Through this project, we can connect the content of the previous several labs to truly design a valuable intelligent computing system, including model training, model quantization, hardware design, operator and algorithm development. Due to time constraints and teammates infected with COVID-19, we were unable to further optimize the model to reduce its running time.**

***Index Terms*—MAC, RISC-V, verilog**

I. Introduction

In the previous labs, we have learned the training of neural networks, the quantization of neural networks, the design of systolic arrays, and the design of general-purpose processors. However, the content of the previous experiment is relatively independent, and the knowledge is modular. Through this experiment, we connected all the previous contents in series to truly realize a unified system.

Section II is about neural networks. In the previous lab0 and lab1, we implemented the training and quantization of the neural network based on python. Generally speaking, deploying a neural network on hardware requires customizing the neural network's quantization scheme according to the hardware's characteristics. In lab1, we implemented an 8-bit quantized neural network, which is a feasible solution to deploy on an edge hardware platform. However, we made minor modifications to it, and Section II gives a detailed description.

Section III introduces the hardware platform of this project. As we all know, the implementation of neural networks requires huge computing requirements, and using only an embedded system to run neural networks will produce unacceptable delays. Therefore, we usually consider the deployment of processor cores and accelerators when computing at the edge. The specific hardware platform design will be described in detail in Section III.

Through the above steps, we have obtained the neural network model and hardware platform, and then we will focus on how to deploy the trained parameters to the hardware platform. In lab0, we obtained the operator of the neural network based on NumPy in a high-level language, and in lab3, we implemented the operator of matrix multiplication in assembly language. However, the flexibility of assembly language is low, and it is not suitable for realizing a larger network. Here we use C + assembly inline to build a real neural network that conforms to the hardware platform.

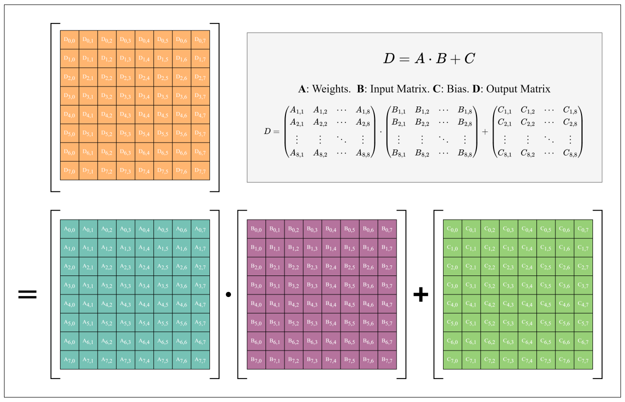


Fig. 1. MAC operation.

II. Neural Networks

A neural network is mainly composed of convolutional layers, activation functions and pooling layers. The training and quantization of the neural network have been completed in lab0, but the previous quantization coefficients are floating-point numbers, which are not suitable for application on hardware platforms. For deployment on specific hardware platforms, we restrict quantization coefficients to integers.

An efficient method is to use shifting to implement the division of integer numbers, for example, shifting two bits is analogous to multiplying by a quantization factor of 0.25. By analogy, the scaling factor of each layer can be adjusted from any floating-point type to an integer parameter. Then, use the model provided by the teaching assistant to export the trained neural network. Use model.py to generate a test stimulus source for subsequent hardware platforms.

III. Hardware

The hardware platform is improved on the basis of lab3, including a scalar processor and a vector processor. The registers in lab 3 are 32-bit, and the registers here are 64-bit. The scalar processor has been completed, and you can directly use GCC to compile C++. We added another vector processor consisting of eight 64-bit registers.

In addition, on these bases, we also customized an accelerator to assist in the implementation of convolution, activation and pooling operations. Our accelerator uses a systolic array architecture and embeds it in an array of vectors. That is to say, we can control the vector processor through custom instructions, and at the same time, the memory access of the accelerator is realized through the load and store instructions of the vector processor, that is, we limit the bandwidth of the accelerator memory access. According to the memory access bandwidth, we designed a reasonable accelerator and compatible with our vector processor. Referring to last year's lab 3, we designed a custom instruction control accelerator.

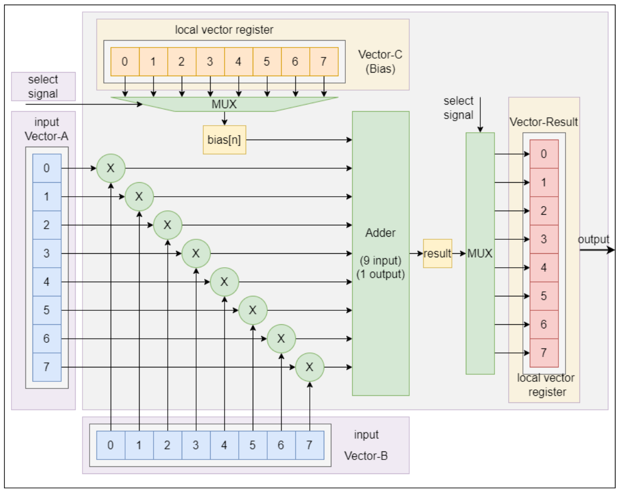
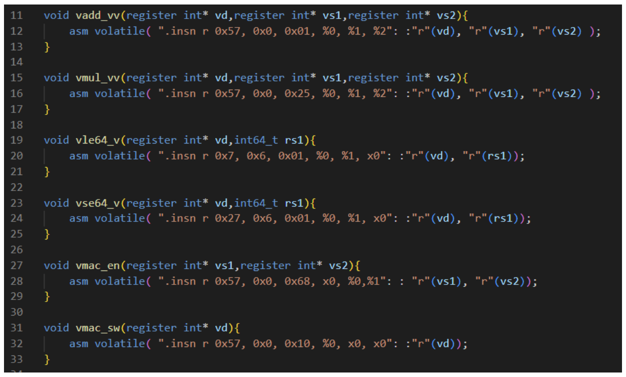


Fig. 2. The diagram of multiply-accumulate module.

IV. Software

Our processors are general-purpose and can execute multiple instructions. Like a circuit implemented with Verilog code on a hardware platform, application software is developed to assemble the instructions. Assembly code in lab 3 is a form of implementation, but assembly code is less flexible. Here we implement all common operations based on C language and compile the C language program into a binary file.

In the C code, we have create following custom vector instructions:



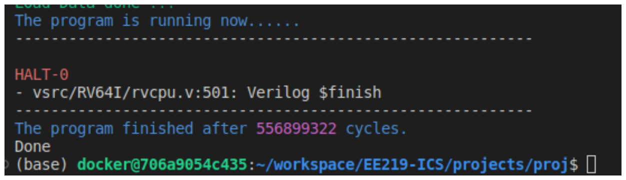
```
void vadd_vv(register int* vd,register int* vs1,register int* vs2){
    asm volatile( ".insn r 0x57, 0x0, 0x01, %0, %1, %2": :"r"(vd), "r"(vs1), "r"(vs2) );
}

void vmul_vv(register int* vd,register int* vs1,register int* vs2){
    asm volatile( ".insn r 0x57, 0x0, 0x25, %0, %1, %2": :"r"(vd), "r"(vs1), "r"(vs2) );
}

void vle64_v(register int* vd,int64_t rs1){
    asm volatile( ".insn r 0x7, 0x6, 0x01, %0, %1, x0": :"r"(vd), "r"(rs1));
}

void vse64_v(register int* vd,int64_t rs1){
    asm volatile( ".insn r 0x27, 0x6, 0x01, %0, %1, x0": :"r"(vd), "r"(rs1));
}

void vmac_en(register int* vs1,register int* vs2){
    asm volatile( ".insn r 0x57, 0x0, 0x68, x0, %0,%1": : "r"(vs1), "r"(vs2));
}

void vmac_sw(register int* vd){
    asm volatile( ".insn r 0x57, 0x0, 0x10, %0, x0, x0": :"r"(vd));
}
```

Fig. 3. Custom Instructions.

Using the above instructions and combined with scalar instructions, we complete the MAC operation.

V. Results

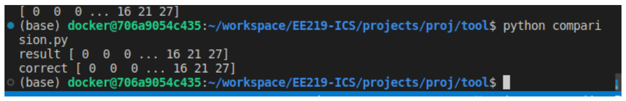
Firstly, we get the following results based on the scalar processor:



```
The program is running now......
----------------------------------------------------------

HALT-0
- vsrc/RV64I/rvcpu.v:501: Verilog $finish
----------------------------------------------------------
The program finished after 556899322 cycles.
Done
(base) docker@706a9054c435:~/workspace/EE219-ICS/projects/proj$
```

Fig. 4. Result cycle.

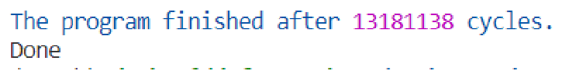
Then we write a comparision code and check the results with correct answer:



```
 [ 0  0  0 ... 16 21 27]
(base) docker@706a9054c435:~/workspace/EE219-ICS/projects/proj/tool$ python compari
sion.py
result [ 0  0  0 ... 16 21 27]
correct [ 0  0  0 ... 16 21 27]
(base) docker@706a9054c435:~/workspace/EE219-ICS/projects/proj/tool$
```

Fig. 5. Compareing Result.

After that, we have optimized the accelerator, having shorter cycle:



```
The program finished after 13181138 cycles.
Done
```

Fig. 6. Result cycle.

VI. Conclusion

In this project, we actually implemented the neural network on the hardware platform, which is a summary of the knowledge learned throughout the course. We mainly achieved the following key tasks: First, we skillfully converted the scaling factor from a floating-point number to an integer through a shift operation. Through the python module, the trained network is used to generate an incentive source for testing subsequent hardware platforms. Second, we complete the involved vector processors. The systems based on scalar and vector processors were tested separately, and the result cycle was obtained as shown in the results section. In addition, we have also achieved a good acceleration effect based on the design of a suitable accelerator. Finally, our system exhibits competitive performance. However, due to time reasons, the server often reports errors, and teammates are infected with COVID-19, etc., which limit the further improvement of our system. Although there are small regrets, we have benefited a lot from the course of intelligent computing, and it is also satisfying to be able to realize this project.

References

[1] Siting, Liu. EE219: ICS, 2022-Fall