Domain-Specific Accelerator

Zih-Bo Yu(于子博), 111550016

***Abstract* —In this report, I will first explore two hotspot functions in CNNs and some possible acceleration methods. Then, I will explain how I implemented a DSA accelerator using hardware.**

I. Software Analysis

A. Fully connected layer

The forward propagation in the fully connected layer is one of the hotspot functions. The weight matrix used in the computation remains the same during each execution, so it can be stored in BRAM instead of being fetched from memory every time.

Moreover, since the input size is fixed at 512, the operation:

```
a[i] += W[i * entry->base.in_size_ + c] * in[c];
```

is executed consecutively 512 times. In other words, *c* consistently ranges from 0 to 511. By passing *i* through MMIO, the 512 calculations can be performed consecutively without interruption.

B. Convolutional layer

In the conv_3d() function, the following computation, which can be accelerated by hardware, is encapsulated within three nested for-loops:

```
sum += *ppw++ * ppi[widx];
```

In this function, since the size of the weight is fixed, the above computation is executed 25 times for each iteration, with the results accumulated into pa[]. Therefore, I believe that parallelizing these 25 computations can significantly improve computation speed.

In the first for-loop, the starting position of the weight matrix is fixed, and the weights used in the inner loops are always the 25 elements starting from that position. As long as the execution remains within the inner two loops, the weights remain constant.

In the second and third loops, before performing the 25 computations each time, the corresponding 25 elements of in[] need to be loaded into the registers in the data feeder. However, this process can be further optimized. Based on my tests, preloading the entire in[] array into BRAM at the beginning of the function is much faster than loading 25 elements each time. This is because the second and third loops are executed a large number of times, repeatedly using elements from the same positions.

II. Hardware Implementation

A. Datafeeder for fully connected layer

For the data feeder of this function, since the weight matrix is already stored in BRAM, we only need to read in[c]. Both the reading and computation can be pipelined. When the FP IP starts computation, it can send a "ready" signal, allowing the CPU and FP IP to operate simultaneously. As for the accumulation, it can be achieved by connecting the FP IP's result to c_tdata. The FP IP's a is connected to the BRAM storing the weights, and b is connected to the input data. This setup enables continuous multiplication and accumulation.

B. Datafeeder for Convolutional layer

For the data feeder of this function, the entry point is when the MMIO detects the input of weights[]. In other words, we treat the first for-loop as the entry point for the entire hardware. Then, in[] is input, both involving 25 elements each time to enable parallelization of the computation. Next, the calculation begins. I used 25 FP multiplication IPs to compute these 25 intermediate values simultaneously. To sum them up, I utilized a 5-layer FP addition IP, which can combine the 25 FP results. This setup achieves approximately a 3x speedup during computation.

Connecting these IPs together is relatively straightforward. For example, the result valid signal of the previous layer can be connected to the input valid signal of the next layer. This way, once the first layer starts computation, we only need to wait for the final answer from the entire cascaded system without maintaining intermediate signals.

When the computation is complete, we need to determine whether the inner two for-loops have finished. If they are complete, the system should return to the initial state. If not, the weight matrix can be reused while waiting for the input of the next in[].

If the entire in[] is to be loaded after the function starts executing, additional information would need to be provided from the CPU to the data feeder through MMIO. This includes details such as const1 and ppi to determine the positions of the in[] elements for each computation.

C. Possible Optimizations for Pooling Layer

We can observe that the following equation aligns well with the format of our optimization:

*a[o] *= entry->scale_factor_;*

I tried removing this line to test the potential improvement, and surprisingly, deleting it did not affect the correctness, though I couldn't determine why. After testing, removing this code reduced the execution time to 1992ms(with datafeeder).

III. TEST DATA ANALYSIS

	Original	Func1	Func2	both
Time(ms)	21336	21176	2691	2112

The results reveal some interesting facts. For instance, when comparing the cases of no acceleration and accelerating only func2, the performance improvement from also accelerating func1 is significantly larger. This could be related to the data cache. With more functions being accelerated by hardware, cache usage may decrease, potentially increasing the hit rate for other data.

In terms of results, the performance improved by approximately 10 times. However, I believe there is still significant room for improvement because I did not reduce the overhead of the SD card. For further improvement, it would be necessary to load all the weights into BRAM at the very beginning. Although I have not implemented this yet, I think it is possible to connect BRAM to the CPU through an MMIO interface. However, I am unsure how to modify the code in the software. Specifically, I am uncertain whether simply specifying the MMIO address in the read_weights() function in read_file.c would enable the program to access the weights through BRAM, or if there are additional modifications required elsewhere.

Furthermore, if the entire in[] is loaded at the beginning of the conv_3d data feeder, the total computation time can be reduced to approximately 1755ms. This functionality was not implemented due to time constraints, but it is possible to predict the execution time by modifying the software code. However, the result would be inaccurate.

Conv_3d()	Caculation(FP IP)	others
Time Proportion	0.038%	~99%

The table above describes the time proportion of floating-point calculations and other tasks after hardware acceleration. It can be observed that the computation time is very small. Apart from my own optimizations, the computation time was already minimal. I also tested adjusting the latency of the IP, but the effect was not very significant, which further confirms that this assignment should focus more on data processing.

	read files time	Total time
Original	922	2186
Pre-load weights	708	2067

The above results show the outcome after storing the weights in BRAM and accessing them through MMIO with the CPU. It can be observed that preloading the weights into BRAM reduces the overhead of the SD card, but the execution time of the program does not accelerate. I have ensured that every access to the weights is done through BRAM, but this might be because the original design stored the weights in the cache. Storing them in BRAM introduces the overhead of MMIO.

IV. WHAT I LEARN

Although the results this time were somewhat less than ideal, I learned a great deal. First, I learned how to utilize existing IPs and the commonly used AXI protocol. By leveraging these powerful IPs, FPGAs can accomplish a wide range of tasks.

Secondly, I gained experience in hardware-software integration. By using the volatile keyword in the software to read data from memory and leveraging the characteristics of MMIO, seamless communication between hardware and software can be achieved. This not only enables hardware acceleration but also proves to be very useful for debugging.

In addition, although my idea of chaining multiple floating-point IPs together may not lead to significant acceleration, it still provided me with an opportunity to learn how to use IPs. Through this chained system, I now fully understand the functionality of the AXI protocol and how to utilize it.