Lab5, Domain-Specific Accelerator

JUN-KAI CHEN, 111550001

***Abstract*—This paper describes the design of a domain-specific accelerator (DSA) for the Aquila SoC. It uses a hardware IP with AXI protocol which calculates single-precision floating-point numbers. The content further records and compares the performance with this DSA.**

***Index Terms*—Aquila, DSA, Matrix Multiplication, MMIO**

I. Floating-point Calculation in Aquila

Nowadays, most machines adopt the IEEE floating-point number standard, using it as the method to store and describe real numbers. Unfortunately, the Aquila processor we used in this lab implements only the RV32IMA ISA, which means it can't handle floating-point numbers as modern PCs do.

If a processor doesn't support floating-point numbers, it usually needs a software floating-point emulator. In this case, the compiler simply links the statement to another function, calculating through bit operations. The most significant problem with software emulation is its low efficiency. To address this issue, we integrated an accelerator into the SoC to improve performance.

II. DSA Design

To design the DSA, we need to know which part of the software we wnat to convert into a circuit. In addition, we must also design the interface of the hardware device. Here we focus on the two functions provided in the slide: `conv_3d()` and `fully_connected_layer_forward_propagation()`.

These two functions consist of nested loops to perform matrix multiplication. Thus, I planned to design a one-dimensional vector multiplier. The design of the DSA is described in Figure 1. Basically, it will have a buffer to store data, and a floating point arithmetic unit and controller to perform calculations.

Different from using lock to represent the current state of the device, I used the protocol features of Aquila to use the controller. If Aquila sends a request to an external device, it expects a reply after a short delay, at which point the processor stalls. DSA currently stores the request and will respond to it when it is ready. Although this will cause problems in multi-tasking, the effect in this case is roughly the same as using busy-waiting.

The execution of the DSA consists of two tasks:

1) Reading the given data and placing it in the correct address space.
2) Calculating the inner product of the given data and sending the result to the processor when it's requested.

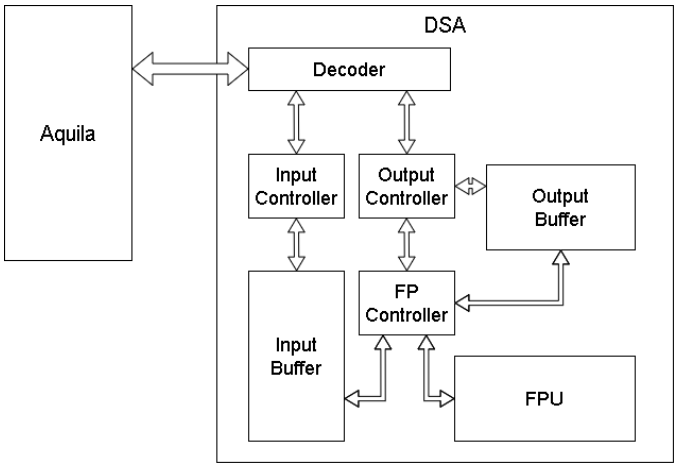

Fig. 1. The block diagram of the DSA.

Therefore, we need to take turns inputting the elements of the two vectors and let the DSA start operating after receiving both values.

Since Aquila uses MMIO to communicate with external devices, a decoder to decode the address and request type is necessary. Three vectors, A, B, and result C, use the address spaces `0xC400_XXXX`, `0xC401_XXXX`, and `0xC402_XXXX`, respectively. The FPU should calculate the result of `(A[i]*B[i])+C[i-1]` and place it in `C[i]`.

The floating point unit we use here is the IP core provided by Xilinx, performing the FMAC operation. In this configuration, the FPU takes 17 cycles to calculate the result. With additional cycles for each input and output, it takes 19 cycles in total. However, because the FPU is controlled by the FPU controller, the state machine spends an additional 2 cycles on storing data and initializing.

On the other side of the DSA, the frequency of requests sent by Aquila is influenced by two factors: branch misprediction and data cache misses. The former occurs when entering or exiting loops, while the latter happens when reading new data. Without these issues, Aquila transfers data every 8 cycles on average. However, this scenario is highly optimistic, so we can't assume that Aquila transmits data faster than the DSA receives it.

Because the processor and FPU work asynchronously, two register arrays are used in the DSA to signal the state machine to start. This design allows the FPU to start earlier, but the trade-off is that it clears all memory when transmitting data. Otherwise, more circuit would be needed to maintain the data.

III. Statistic

To collect statistics on the DSA, I add other modules to count the cycles spent on receiving data and performing calculations based on the controller.

The details of data transmission and calculation are listed in Table 1 below. The transmission time is defined as the duration between the transmission of the first element and the request for the result. Similarly, the calculation time is the duration between the first element sent to the FPU and the result sent to the processor. Since Aquila stalls when it hasn't received the ready signal, we also count the cycles between the processor sending a request and receiving the result.

In addition, the respective data and final execution time of different functions are also recorded.

To ensure the correctness of our program, we checked the inner products calculated by the DSA and compared them to the results of software emulation. Only some results had an acceptable error (less than 0.0001). This kind of error might be caused by the rounding policy. In the end, the OCR test also got the same result as the body, so I think the DSA result should be correct.

It can be seen from the results that the entire program spends more time calculating the convolution layer. However, most of these calculations are shorter vector inner products, because the calculation time is significantly less. In contrast, although the calculations of the fully connected layer are less, there is no obvious performance optimization. These calculations actually require more data and calculation time.

The statistics here do not list information related to cache, because I did not design to rewrite the data to BRAM when reading the data.

IV. Conclusion

In this lab, we integrated an IP core into the Aquila SoC and significantly improved performance. This outcome clearly demonstrates the power of the DSA and its impact on processor design. However, beyond integrating a DSA, it is crucial to consider how the processor interacts with the DSA. The challenges mentioned in the previous section are also a result of the interaction design. The program currently sends data and requests via load and store instructions, but this interaction could potentially be enhanced by using status registers, similar to those in the UART module, or even by designing custom instructions specifically for the DSA.

Another important consideration is cache behavior. Since convolution neural networks primarily perform matrix operations, optimizing these processes using techniques like block multiplication could yield further performance gains. While these aspects might not directly relate to the experiment conducted, they offer valuable insights into improving both software and hardware performance.

TABLE I
TABLE 1. THE TIME SPENT IN DIFFERENT BUFFER SIZE.

function	type	number	latency(cycles)	avg	time(seconds)
forward propagation	calculation	200	3,109,471	15,548.71	20.865
	wait	200	2,937	14.685	
	transmission	200	3,110,599	15,553	
convolution 3d	calculation	157440	83,994,614	533.50	3.782
	wait	157440	456,755	2.90	
	transmission	157440	84,958,801	539.63	
both	157640	calculation	87,104,675	552.55	3,782
	157640	wait	459,257	2.91	
	157640	transmission	88,070,136	558.68	
original					21.243