HW5 Report
Domain-Specific Accelerator

Tzu-Han Hsu
dept. Computer Science
National Yang Ming Chiao Tung University
Student ID: 081358

***Abstract*—We integrate a floating-point MAC unit as DSA to accelerate inner-product calculation. Experiments show at most 14x faster could be reached over software emulation when vector size is larger than a threshold.**

***Index Terms*—FPU, accelerator, MAC, RISC-V, Aquila**

I. Floating-Point accelerator

Most devices nowadays use IEEE-754 standard floating point to record, compute and exchange real data. Floating-point computation offers higher precision and dynamic range over fixed-point. However, such advantage comes with a cost, floating point units are more expensive and power-consuming over fixed-point units. Nevertheless, system that requires the range of floating-point arithmetic could implement their design using software emulation or dedicated floating-point units.

Floating point arithmetic is applicable in IOT devices to record and to process the environment statistics in order to increase precision. Most microcontrollers would handle floating point numbers by software emulation, yet a lightweight DSP could potentially accelerate the computation with lower power consumption. Neural networks are millions of neurons with sophisticated interconnections, also operate and train on large quantities of floating-point numbers. Since inner-product is the core operation of Convolutional Neural Network (CNN), the capability of the hardware to multiply and sum a large number of floating-points is the key to success. In this homework, we aim to integrate a floating-point accelerator IP into the Aquila SoC.

To validate our accelerator, a benchmark program is designed. The program would initialize to vectors with defined length with random numbers, then calculate the inner-product of the two vectors using both software simulation and with our integrated floating-point accelerator. We would compare and discuss performances between the two implementation methods. The work would further instantiate our inner-product testbench to an array multiplication testbench, we would also discuss the results obtained.

II. Integrating Floating-point accelerator

The domain-specific accelerator (DSA), in our case the FPU, should communicate with the microprocessor through a bus protocol. Since Memory Mapped I/O (MMIO) architecture is implemented in Aquila, through an address decoder, we could directly access the accelerator through unified load/store interface instructions. To bridge the MMIO signals sending from the microprocessor and the AXI interface communicating with the DSA, we would construct a middlemen IP to move data around. The overall architecture is shown in Fig.1.

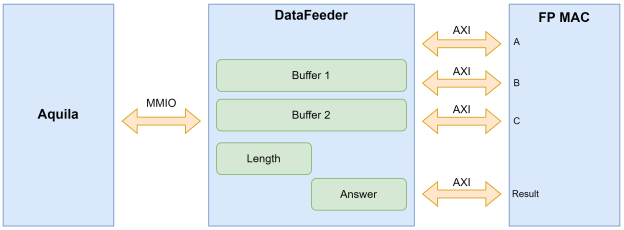

Fig. 1. Floating-point accelerator architecture block diagram

A. Xilinx Floating-point IP

Inner product calculation is a procedure of interleaving multiplications and additions, such operation could be further broken down into consecutive fused multiply-adds (a.k.a. Multiply-ACcumulate, MAC). A MAC accelerator outshines dedicated multiplier and adder in several reasons. Firstly, dedicated multiplication and addition introduces two times rounding, which MAC accelerator only rounds once. Such design would significantly increase accuracy, especially under the circumstance where operators are short is bit length like IEEE-754 single precision or IEEE754 half precision. Secondly, MAC units could merge addition and multiplication operations to achieve parallelism for better speed. Thirdly, introducing dedicated multiplier and adder would require data to flow between two accelerators. We have to design control signals to regulate the behavior which would require extra controllers. To conclude, it's clear that we could benefit most from using a MAC as our floating-point accelerator.

Xilinx provides a floating-point IP supporting fused multiply-add operations. We instantiate the IP to target IEEE-754 single precision (32 bits) operations. The IP operates with blocking mode and the configuration is optimized for resource usage, it means with a slightly longer calculation time but only 2 DSP slices on our FPGA would be used. The IP runs with

TABLE I
STATISTICS OF BENCHMARK PROGRAM EXECUTION WITH AND WITHOUT DSA ON DIFFERENT VECTOR SIZE

Vector length	1	2	4	8	16	64	256	1024	2048	4096	8192
Load time	34	54	83	143	263	983	3863	15383	30743	61463	122903
Calculation time	57	78	117	197	357	1317	5157	20517	40997	81957	163877
Total time	192	213	252	336	492	1453	5292	20652	41133	82092	164016
S.W. Emulation (μs)	78	94	118	157	214	568	1954	7136	14065	27537	54999
H.W Accelerator (μs)	18	21	20	21	25	49	140	510	1002	1984	3950
Acceleration	433.3%	447.6%	590.0%	747.6%	856.0%	1159.2%	1395.7%	1399.2%	1403.7%	1388.0%	1392.4%

AXI4-stream interface with our custom designed middlemen IP.

B. Middlemen IP

The middlemen IP, called the datafeeder, communicates with the MAC unit with AXI4-stream bus. There are 3 input lanes to send in both multiplicands and the addend. The answer flows out the MAC by the result lane. Each lane has dedicated interconnects with a data wire, a valid wire a ready wire. The master circuit sends the data with valid turned on, the slave responses with the ready wire.

The middlemen IP communicates with Aquila via MMIO interface. An address decoder would enable the middlemen IP using predefined address regions, then Aquila could communicate with the accelerator with unified read & write interface instructions. In our design, the middlemen IP would buffer the data sent from Aquila, and keep sending them to the MAC. After the calculation is done, we may response to the request of reading instruction coming from Aquila and deliver the result.

We store the vectors in side a C array, results in memory access in a patterned behavior. Little cache miss is observed in our experiment. Aquila is able to send out one floating point every 7 clock cycles. On the other side of the circuit, we could process one MAC operation on average 19 cycles. The most naive method is to read in all three operands and store them in registers. Once all data is buffered, we start the calculation. Such design is cheap in hardware with exchange of a fairly low speed, taking about $19 + 7 \times 3$ cycles.

If we would sacrifice some extra LUT to build distributed RAM for both vectors, we may receive operands from Aquila while calculating at the same time. Distributed RAM is slightly faster than Block RAMs due to their reading is combinational. However, BRAM are larger in size and could support much larger amount of storage. We employ two 8192-unit buffers for both vectors.

More time could be saved if we eliminate unnecessary communications between the data-feeder circuit and Aquila. Since we aim to accelerate inner product calculation and the length of the vector is fixed. We could design the calculation behavior as follow: Aquila sends in the length of the vector first by MMIO, it is stored in 0xC202_0000; After the vector length is certified, Aquila would continuously send data from both vectors in an interleaving manner, where MAC operates as soon as there is something to work on. Vectors are stored from 0xC200 and 0xC201. Where the answer is continuously accumulated during the process (stored in 0xC203_0000). After the calculation is completed, Aquila could again read the result through the MMIO interface. Such design could effectively accelerate the calculation process, The calculation time now is about the time of MAC calculation, which is the bottleneck of the system.

III. ANALYSIS OF BENCHMARK STATISTICS

Table I shows the statistics collected running the testbnech program with both the FP accelerator turned on and off. Note that 8192 is the designed buffer length, any vector calculation greater than that would lead to incorrect results (slower but correct result could be obtained by using software methods utilizing the accelerator twice). The load time in Table I is calculated when the first data of the A vector is sent from Aquila to the last data of vector B is received. Calculation time is the time MAC starts it's first operation till it finishes the last operation. Total time is the total time to send in the vector length (approximately 135 cycles), send in the data and finish the calculation.

From Table I, our design seems to "hide" loading the data from the Aquila side pretty well. As when the vector size grow large, total time spent is almost the same with the calculation time spent on the MAC, it is also a sign the DSA fully utilizes it's computational resources. Table I shows if the full potential of the accelerator is realized, 14 times acceleration could be reached. Under the circumstance where vector size is 256 or above.

We also instantiate the benchmark to run full array multiplication, which is crucial in CNN networks and more commonly used in practice. Running with software emulation would spent 15 seconds to accomplish the task, which only takes the SoC with the accelerator seconds. It's solid evidence that integrating DSA in SoC could boost user experience drastically.

IV. CONCLUSION

Hardware accelerators could be easily integrated with unified bus protocols like AXI bus. The architecture of MMIO is also beneficial for clean interfacing toward both the S.W. and the H.W. end. This H.W sends a clear message how DSA could change the game rule of SoC design. After all, Apple's M-series chip's success shows that DSA integration in SoC could potentially be the trend of next-step application processor design.