Final Project: Hardware Dot Product Acceleration

Clare Barnes

14 December 2022

For this project, I was able to accelerate an 80x40 hardware dot product implementation from 35,241 cycles to 880 cycles using a combination of pipelining and parallelism. Pipelining alone saved 31,970 cycles and adding four-way parallelism saved 26,381 cycles. My hardware implementation utilizes a floating-point MAC unit with an eight cycle latency. To compensate for this, my dot.sv implementation utilizes the unit's ability to intake inputs each cycle and the free addition operation at the end of the multiplication operation. Accel_dot.sv splits the weight matrix horizontally and routes incoming inputs across four parallel dot.sv implementations. Because the weights were split up horizontally, I had to use floating point addition modules to add the results from dot.sv. I implemented 3 units, parallelizing and pipelining them to take as little time as possible at the end.

My accelerated dot implementation can execute 80 floating point multiply-accumulate (MAC) operations in 0.766402 seconds in a Python environment. Based on performance scaling, we estimate that 0.76 seconds of that time are overheads associated with loading and starting communications with the program. A NumPy implementation required .094458 seconds for the same computation. Based on scaling, I estimate our hardware implementation will be faster than NumPy when the dot product requires about 171,318 MAC operations.

The operation I am trying to implement and improve on hardware is the dot product of large matrices. Dot product is a form of matrix multiplication where each element of the row of the first matrix is multiplied by the corresponding column element of the second matrix and all the individual values of the column are added at the end to output a differently shaped matrix. Dot product is often used in machine learning, especially neural networks by using an input matrix and a weight matrix. These operations can be huge and need to happen for each neuron of a network. And most likely, the neural network would need to perform more that 172,000 MAC operations, so using hardware would be worth it. The baseline solution I was given to improve did 3,200 operations in about 35,241 cycles. For context the solution that was run in the Python environment did the same calculations in 3,321 cycles. So the solution above would need an even larger neural network to become more beneficial than the numPy dot product. But, it is possible to speed that number of cycles up, and would be worth doing.

There are two main modules to speed up, the module that performs the operations, dot.sv, and the module that acts as an in between to dot.sv and the AXI-Stream module sending and receiving data. To optimize dot.sv I used a pipelining strategy. The original code waited for the FMAC engine to complete a multiply and accumulate operation before sending in the next value. This meant that every FMAC operation took 8 cycles each. But the engine can receive a new value every clock cycle and start the multiplication. Meaning two operations that take 16 clock cycles now takes about 9 cycles. So, every row was pipelined into the FMAC and the program only had to wait for the end of the row instead of the end of every value. For a 3x4 matrix, the wait time was about 10 cycles. This is because I coordinated the wait time with switching

between states. Meaning that the wait time was decreased for the FMAC by two clock cycles because of the clock cycles taken between changing states from ST_RUN_FMAC to ST_TERM_ROW. By doing this I was able to get down to 32 cycles on the 3x4 testbench, which is very close to the theoretical minimum- 30 cycles. This minimum is the cycles needed to compute each row (10) multiplied by the rows of the input (3). The cycles for each simulation with just pipelining are below in Figure 1.

I think that if I had time in the end, I would have worked on parallelizing the FMAC operations- so I split the weight matrix up into two FMAC engines, as some of the inputs do not rely on each other. That is the only optimization that I can see that I did not take advantage of. I think it could shave a few cycles off on larger matrix operations. When I implemented it without an accelerated accel_dot.sv, it returned about 32 to 30 cycles on the 3x4 matrix test bench simulation.

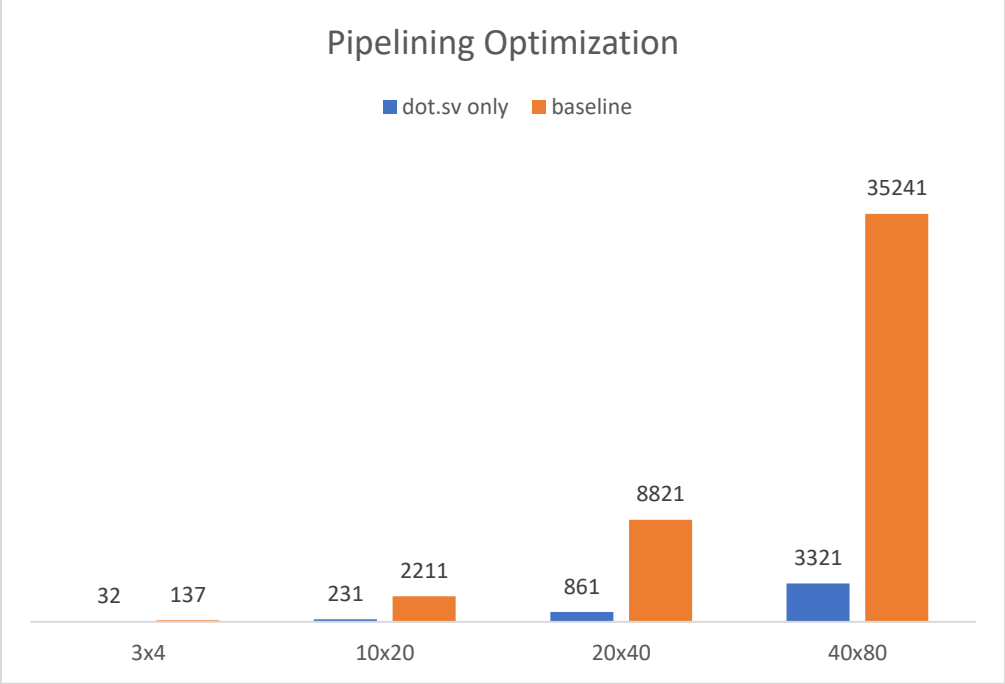

Figure 1 – Pipelining Clock Cycles

In order to accelerate the process more, I used accel_dot as an intermediary between the top modules and dot so that I could break up and spread the calculation over two different dot modules. At first, I wanted to divide the weight matrix by columns, so that I could take advantage of the free add in the FMAC engine and because it would be simpler. I also used arrays for each signal across the dot modules so that the code was easier to expand if I needed to add more dot modules. I created a state machine to coordinate the outputs of the two dot modules to output in the correct order over the AXI-Stream. While this was a much simpler implementation of parallelization, this turned out to be not much faster than a non-parallelized dot. This is because it did not take advantage of the pipelining in dot itself.

So, I decided to split the weight matrix horizontally instead. At first I designed the module to take a 3x4 matrix and split it into two matrices, one with 1 row and the other with 2. Splitting the

matrix into 3 modules would have been too hard to do without use outside of the test module (all of the other input matrices were even). This was much more complex, because that meant I also had to split the input array across the two modules and use a floating-point addition unit (FADD) to add the output from each together in the end. Luckily the FADD, like the FMAC, can receive a new value every clock cycle, so I was able to pipeline this process. Now rather than taking 32 cycles to do each add individually for the 3x4 matrix, it takes about 11 cycles. I modified the existing state machine to coordinate the output from the dot modules into the FADD so that the right values were added together. Then the FADD just directly outputs for accel_dot.sv. After getting this to work, I tried to replace the code splitting up an odd number of rows into two to split an even matrix. I tested this on the larger testbenches, but it did not work immediately. So, I had to rewrite the test bench to give a 4x4 matrix rather than a 3x4. This is why some of the charts use 4x4 rather than 3x4 for the smallest matrix.

After trouble shooting the two dot module implementation, I found out that it was a problem with coordinating the inputs to the FADD. This was a recurring problem. After getting the 4x4 matrix to work, I broke up the weight matrix into four dot modules instead of two modules. I had to add two more FADD units to parallelize the addition process. The structure of the program is illustrated in the figure below. I had trouble here with coordinating the three FADD units. I also had trouble with passing the inputs to each module because I was not properly latching my counters, so my state machine was not working as intended and not passing the right values. I also found that the INPUT_AXIS_TLAST signal was never used, so I had to use counters to keep track of how many values should be input. I suppose, though, that it helps to make sure the input array is not too long. I also tried to just directly input both FADD units to the last one. This doesn't work well, though, because the FADD will say it's ready before it should be. So I control both the valid and ready signal for each FADD unit's input and the output for the first two units. The resulting cycles for the dot product with just parallelization is below as well. It is less effective than the pipelining, both increase at about the same rate.

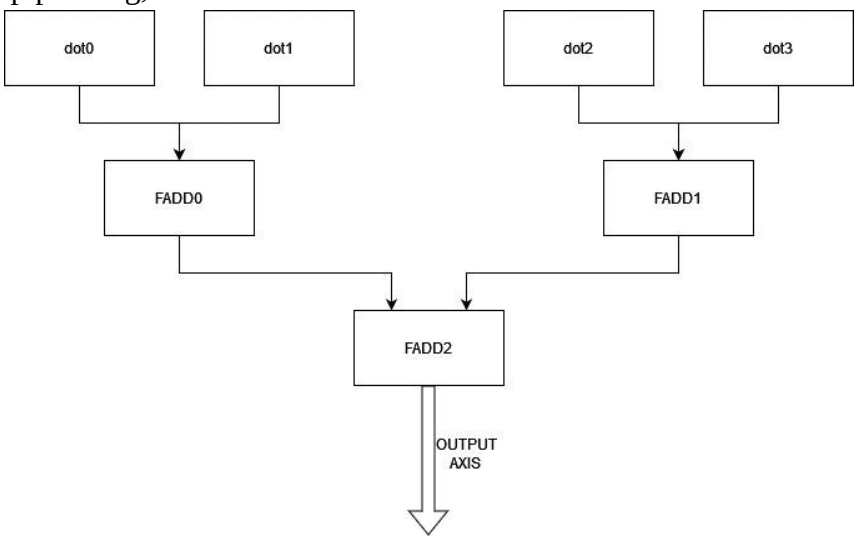

Figure 3 – Accel_dot Structure

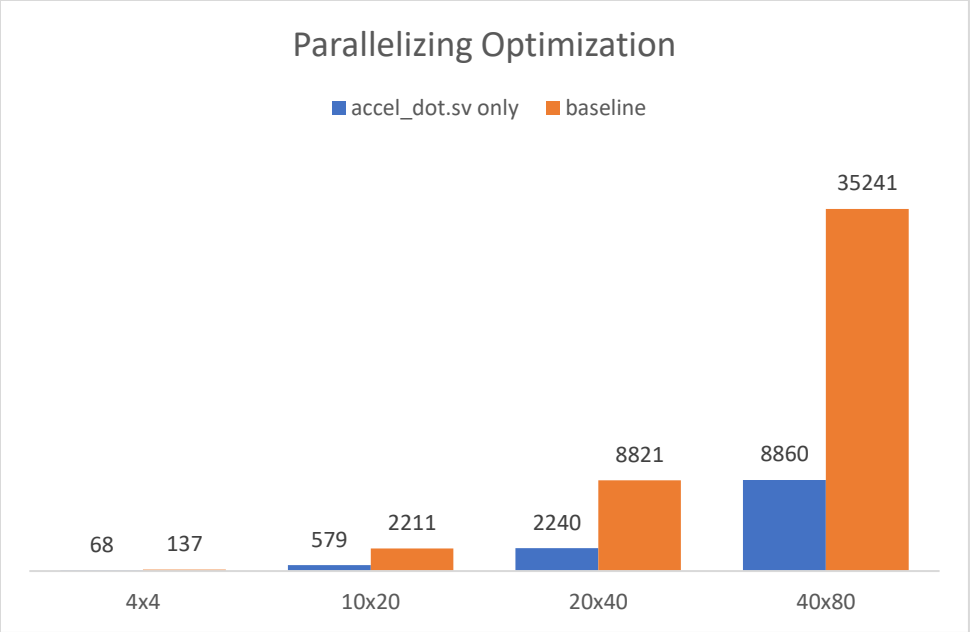

Figure 4 – Parallelization Clock Cycles

To optimize accel_dot more, I think I would make it more scalable. Meaning that it would have a certain amount of dot modules and use more or less based on the input matrix. I would be able to do this by putting the dot modules into an array and by using more checks at the beginning. I think that would be much more complex than necessary for the scope of this project, though. The combination of parallelization and pipelining works very well. The clock cycles that it takes are shown below.

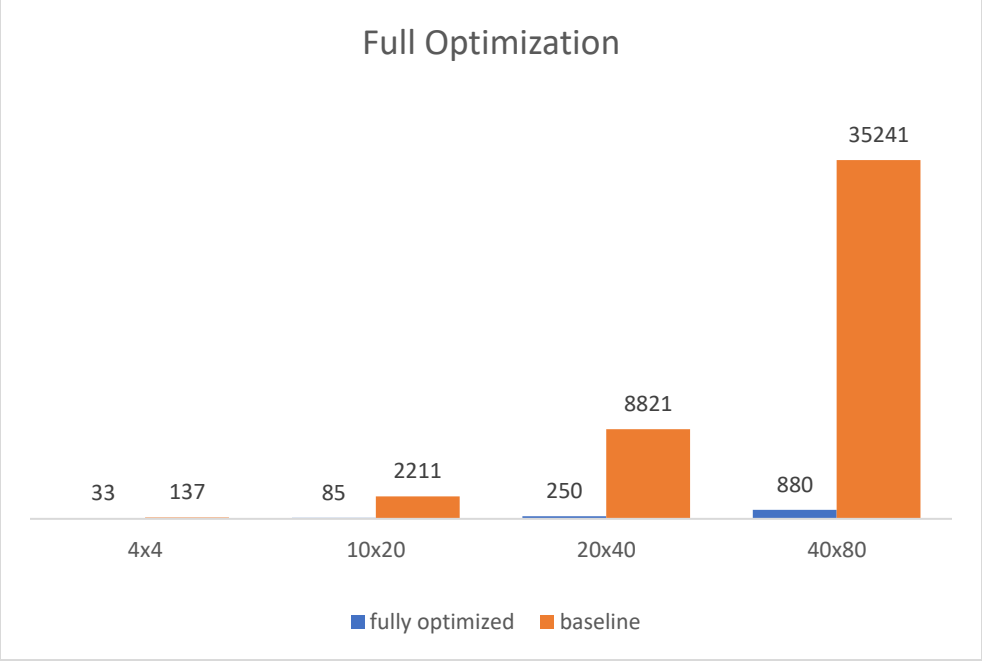

Figure 5 – Pipelined and Parallelized Clock Cycles

My accelerated dot implementation can execute the 20x40 matrix dot product in 0.766402 seconds in a Python environment. The NumPy implementation does the same operation in .094458 seconds. While Python is a runtime language, NumPy is a library that uses both Python and C, as well as coming pre-optimized. Also, there is an overhead of about 0.76 seconds associated with starting up the hardware and giving it the weight matrix, as well as sending the input array over the AXI-Stream. Sending the data to NumPy uses addresses, because it is executed within the environment. A plot with the results of each matrix input in NumPy and hardware is in the figure below. The slope for the hardware is about 7.34E-6 and the slope for NumPy is 1.12E-5. So NumPy is increasing at a faster rate and will eventually be slower than the hardware. Based on scaling, we estimate our hardware implementation will be faster than NumPy when the dot product requires about 171,318 MAC operations. For reference, there is a table below with the FMAC operations needed for each matrix.

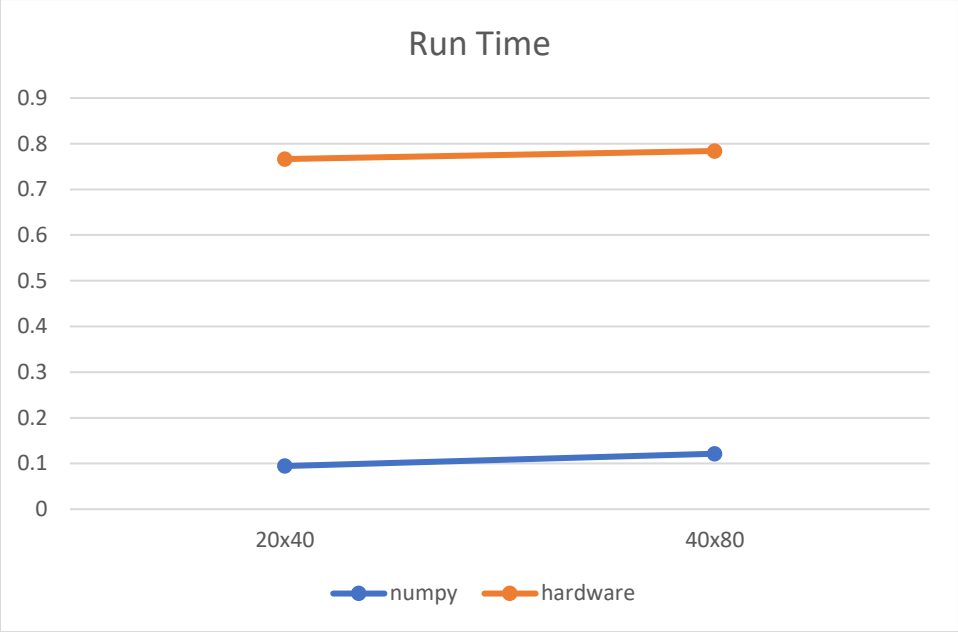

Figure 6 – Run Times of NumPy and Hardware Implementation in Jupyter Notebook Environment

Matrix Size	FMAC Operations Needed
3x4	12
10x20	200
20x40	800
40x80	3,200

Table 1 – Number of Operations per Matrix

At first I made two FMAC engines and parallelized the dot module itself. This gave me a result of 30 cycles instead of 32. But then I realized I should have actually been parallelizing dot by having multiple dot modules, especially because this severely limited the amount of parallelizing I could do. I tried to implement the double fmac dot after parallelizing accel_dot,

but I did not have enough time to figure out the bugs. When splitting the matrix vertically, the output update flip flop in dot would overwrite the initial values of the output array. This usually does not matter, but the second dot module usually needs to wait for a few cycles before it can start sending its own outputs. I fixed it, but like I had said above it did not really go fast enough. The large problem I ended up having with the horizontal division was coordinating the inputs from the dot modules into the FADD. I had to control the ready and valid signals for the inputs and outputs of each FADD but the last. I did not get any bad advice, the only advice I would give is that the horizontal implementation is less complex than one would initially think.

This was an individual effort, although I had a lot of help from the undergraduate assistants. My approach was to parallelize accel_dot.sv and pipeline dot.sv. I made accel_dot easy to expand using arrays for the different variables needed for each dot module. I improved at reading wave diagrams and designing state machines. I learned how to better make distinctions on the variables that need to be in the flip-flop and those that do not. I also learned that when parallelizing it is important to do so in a way that takes advantage of the fastest parts of the code.