Report on Vector Processor Project

Gaurav Kuwar, gk2657; Ritvik Nair, rn2520

***Abstract*—In this paper, we discuss our vector processor project, with a focus on the implementation of the timing simulator, and the results of our assembly codes. Additionally, we discuss a number of potential optimizations. GitHub repo - https://github.com/Rikaero/Vector-Computer-Project**

I. Introduction

This project focused on the implementation of a vector processor using Python. The project was divided into three parts. The first part was about creating a vector simulator, the second part was about implementing different functions in the simulator, and the third and final part was about implementing a timing simulator for the different functions that were tested on the simulator.

In the first part of the project, we created a vector simulator using Python. This simulator inputs assembly code and executes the different processes line by line outputting the vector and scalar register files and memories. The simulator can perform many different scalar and vector operations such as load, store, arithmetic, and shuffle. The simulator was tested on a dot product code.

The second part of the project was focused on the implementation of multiple different functions on the simulator. We tested the simulator on a fully connected layer, a convolution, and a fast Fourier transform. We wrote the code for these functions in assembly that is compatible with our simulator from part 1.

The final part of the project was a timing simulator. The timing simulator calculated the time it would take to run the functions from parts 1 and 2 based on certain configurations. The simulator outputs the number of cycles taken by a function based on the configuration.

II. Functional Simulator

The functional simulator takes a VMIPS assembly code as input and simulates output changes in register and memory values over the iteration of each instruction. Key We ran assembly code for dot product, convolution, and fully connected layer with the functional simulator.

III. Timing Simulator

The timing simulator takes the trace of the VMIPS assembly code and outputs the number of cycles it would take the vector processor to execute all the instructions given some configuration parameters. In our implementation, we iterate cycle-by-cycle, simulating events occurring during each cycle, which affects the total number of cycles, ignoring data changes

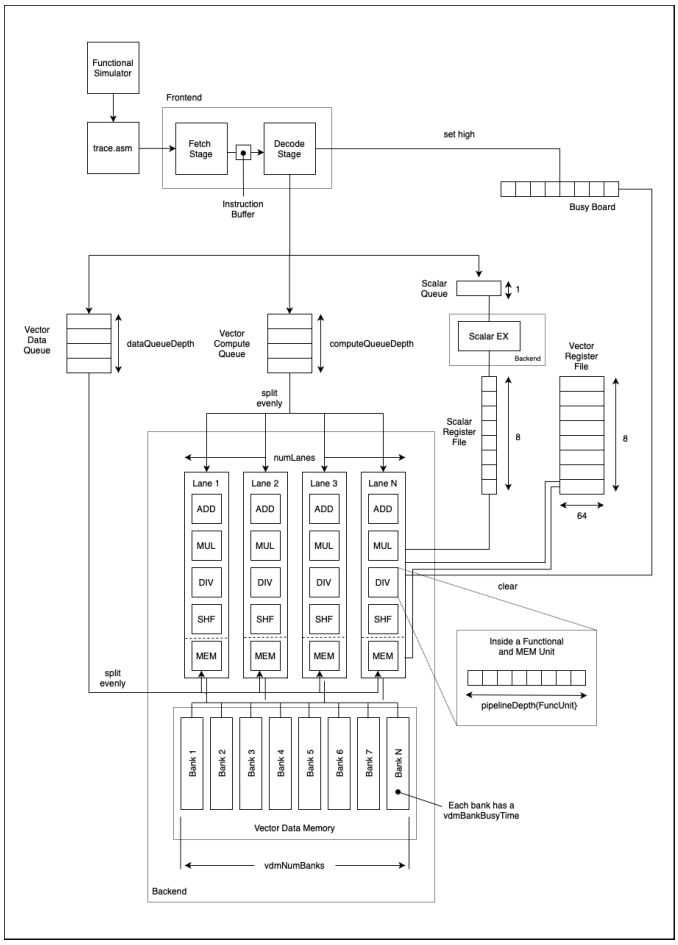

Fig. 1. Block Diagram of Vector Processor

The simulator is divided into the frontend and backend, which decouples instruction access and execution. The frontend includes the fetch, and decode stages, which push instructions to queues after resolving dependencies, and the backend processes the instructions accordingly.

A. Fetch Stage

First, we modified our functional simulator to optionally output a *trace.asm* file, which resolves branches and memory addresses of load/store instructions. This file is inputted into our main **TimingSim** class. At the fetch stage, each line of the trace is read per cycle and stored into the instrBuffer

accessed by the decode stage.

B. Decode Stage

We check the busy board, and only add to the queues, if the registers in the instruction are not in use or if the queue is not full. If these conditions are not met we stall the frontend and repeat the decode of the instruction in the instrBuffer until they are met. Once the conditions are met we mark the registers the instruction will use as high in the busy board and then queue the instruction into the appropriate queue. The use of the busy board allows us to avoid data hazards, although the logic could be relaxed as discussed in the optimization discussion.

The scalar instructions (and some other exceptions) simply take one cycle, and to keep things consistent we add these to a scalar queue of size 1, which are decoded in the next stage.

C. Vector Functional Unit

To implement the vector functional units, we created a class to which simulates the process of these units. For each lane, we have a list which represents the pipeline of the appropriate pipeline depth. Once an instruction is stated to enter the unit, at each cycle we update these pipelines to take *numLanes* elements from the vector (split evenly across lanes) and append to the pipeline and shift the pipeline right. Vector elements at the end of these pipelines are said to be processed.

We set the unit to be busy when an instruction is being processed, which allows us to avoid structural hazards.

D. Vector Data Unit

This unit builds on this functional unit. We still have pipelines per lane as lists, but instead of popped elements being discarded, we input them into the banks. Banks are represented as list of timings, where the timings are time till the bank is not busy, and 0 means the bank is free. If the bank is not free, we stall the pipeline to avoid bank conflicts.

Each address is mapped to a bankIdx using the formula: **bankIdx = address % vdmNumBanks**.

IV. Results

We graphed change in cycle time over the range of 1-25, for the following list of parameters and their baseline values:

- *dataQueueDepth = 4*
- *dataQueueDepth = 4*
- *vdmNumBanks = 16*
- *vlsPipelineDepth = 11*
- *vdmBusyBankTime = 2*
- *pipelineDepthMul = 12*
- *pipelineDepthAdd = 2*
- *pipelineDepthDiv = 8*
- *pipelineDepthShuffle = 5*

We change 1 parameter for each graph while keeping the rest as a baseline. We test on the assembly code for dot product, fully connected layer, and convolution. Here, we will discuss key patterns and odd results, and an explanation for them.

A. Dot Product

The dot product results provide a good baseline for us to compare with the other functions. Most other function graphs have similar curves to that of the dot product. With this knowledge, we can focus on the differences we see between the dot product and the other functions when looking at the graphs.

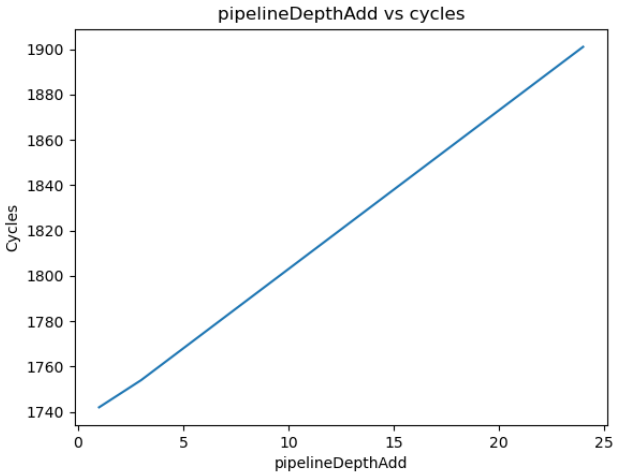

Fig. 2. Changing pipelineDepthAdd for Dot Product code

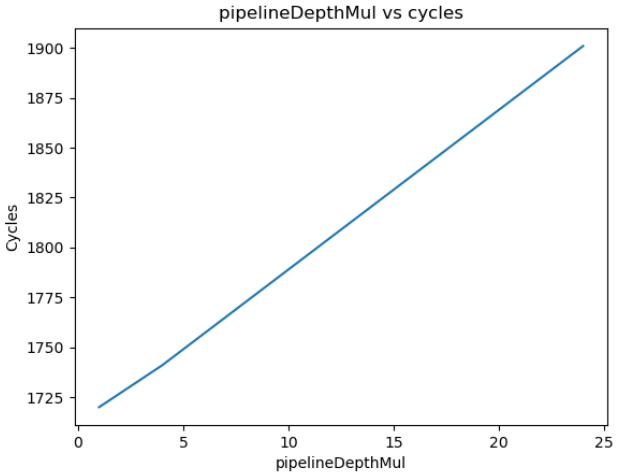

Fig. 3. Changing pipelineDepthMul for Dot Product code

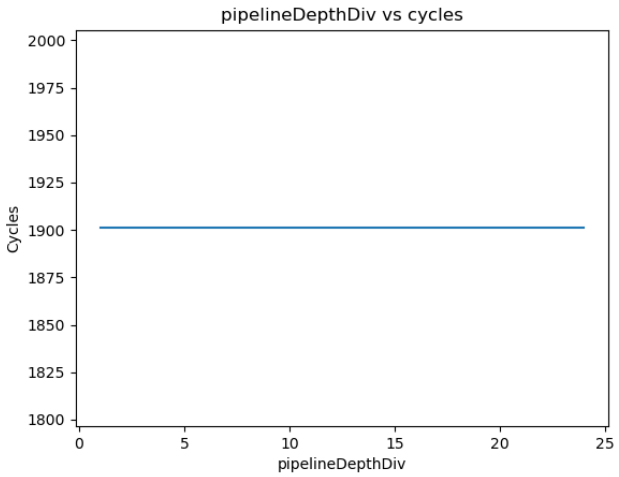

Fig. 4. Changing pipelineDepthDiv for Dot Product code

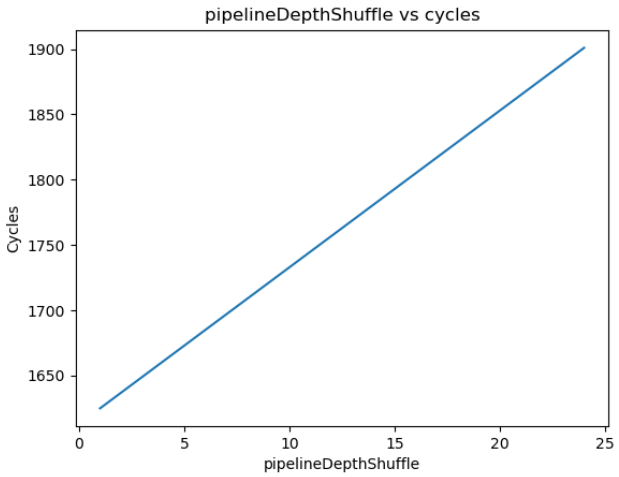

Fig. 5. Changing pipelineDepthShuffle for Dot Product code

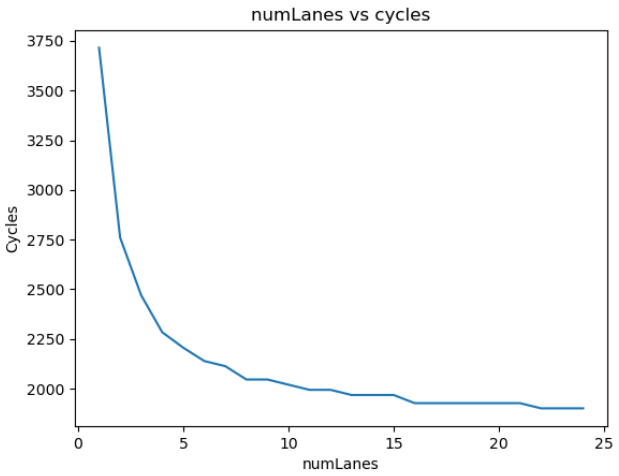

Fig. 6. Changing numLanes for Dot Product code

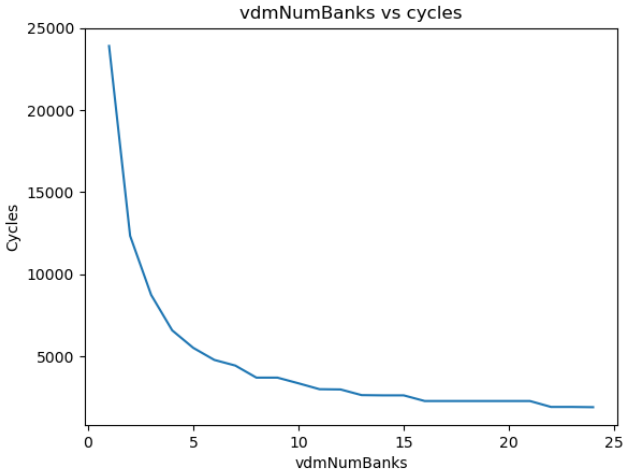

Fig. 7. Changing vdmNumBanks for Dot Product code

Figures 2 to 5 show the curves for the modified pipeline depth of the functional units for the dot product. We can see in figure 4 that modifying the pipeline depth of division has no effect on the execution time. This is due to there being no use of the division in the dot product assembly code. The other three figures show a similar linear increase in cycles as these operations are used repeatedly and in a similar amount in the code.

Figures 6 and 7 show the differing cycles as the number of lanes and the number of memory banks are modified. Both of these follow a similar curve. We can see that as these values increase, the number of cycles decreases exponentially.

Figures 8 and 9 were also tested. Both of these showed an increase in execution time as the value increased. This is expected as both of these increase the cycles it takes to process something. What is odd is the inflexion seen in the vlsPipelineDepth. This can be due to the number of banks negating the effects of vlsPipelineDepth at larger values.

B. *Convolution*

The results we discovered for convolution were the most abnormal. Most of the strange results we obtained were from memory-related modifications in the configuration file.

Figure 10 and figure 11 show some oddities we discovered in the convolution code. We can attribute the zigzags in the vdmNumBanks to how we have done stride. When the banks are even, the code performs worse than when the bank is odd. On the other hand, the irregular graph of numLanes makes very little sense. We speculate that this could be due to the value of vdmBankBusyTime.

V. Optimization Discussion

Although we didn't implement optimizations, we did attempt to start with the chaining optimization and went through the thought process of other optimizations such

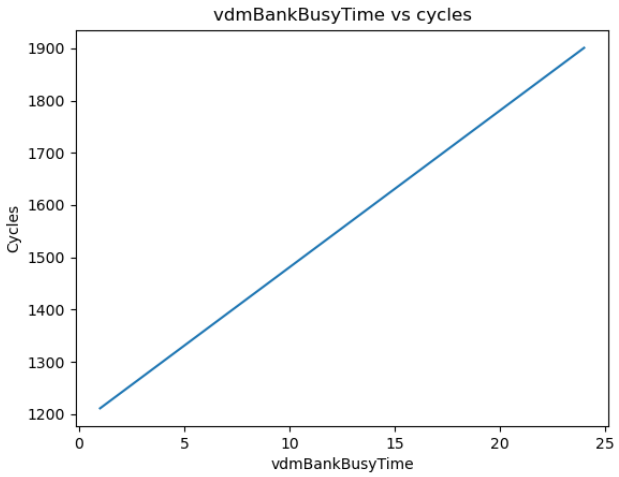

Fig. 8. Changing vdmBankBusyTime for Dot Product code

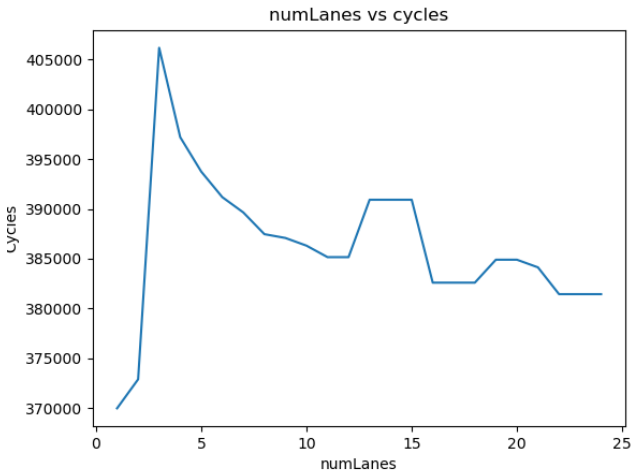

Fig. 10. Changing numLanes for Convolution code

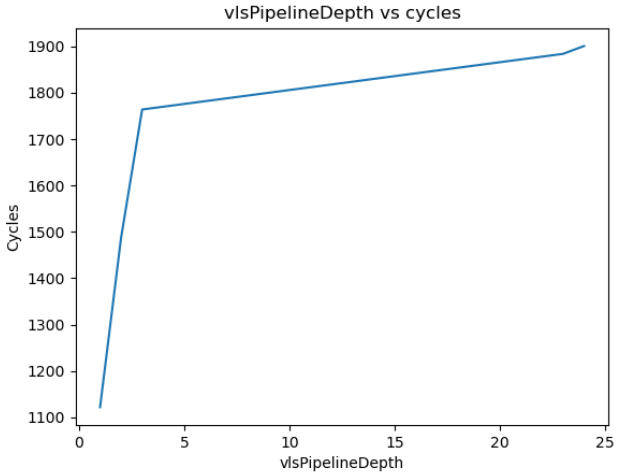

Fig. 9. Changing vlsPipelineDepth for Dot Product code

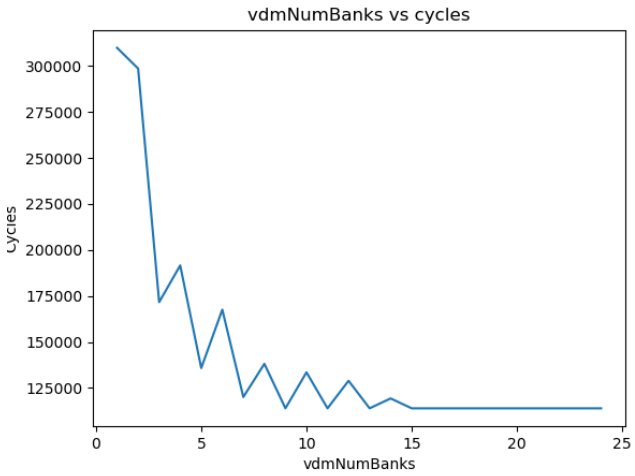

Fig. 11. Changing vdmNumBanks for Convolution code

as busy board stalls reduction, and chaining consecutive instruction which uses the same functional/vector load/store units. But, we think the the ideas are worth discussing here.

A. *Chaining with different units*

Let's say we have a scenario where we output into some vector register *a* in one instruction *I1* and use *a* as input to the next instruction *I2*. In the current implementation, we wait for *I1* to run each element through the pipeline. However, this is unnecessary since after the pipeline runs once, we have computed the first element *a[0]*, and consecutive elements can be accessed every cycle thereafter. Therefore, if we forward per element *a[i]* to *I2*, we run *I1* and *I2* closely parallel (taking into start-up times and an offset of 1).

This does come up as an overhead, since now we have to keep track of each element in each vector register in our busy board, hence we will need 64*8 = 512 bits versus the current 8. Another idea would be to keep track of how many bits from the start of the vector after not busy which would need 6*8 = 48 bits, but this would need an adder to update these values.

Theoretically, this should lead to a massive speedup. Here, it is important to note, that this only works if *I1* and *I2* are *not* using the same unit.

B. *Chaining with same units*

For consecutive instructions where we use the same units, we can add a minor optimization. In the current implementation, when we have two consecutive instructions which use the same unit, we wait for the pipeline to finish,

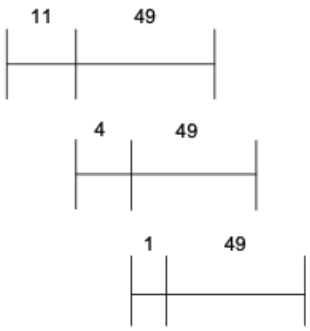

Fig. 12. Chaining

however, we can continue the pipeline, by starting the processing of the next instruction once, the first stage in the unit is free. This goes hand-in-hand with chaining with different units.

C. Busy board stalls reduction

Due to the way we implemented the busy board logic, I noticed that a lot of instructions were stalling even when there would have been no data hazards in running them in parallel because they were simply reading a register, by grouping instructions that *only* read, we can eliminate some stalls.

D. Other thoughts

Another, optimization we thought of was implementing out-of-order execution to reduce stalling. As we discussed in class, this would add quite a lot of overhead and complexity to our hardware, therefore, we are better off implementing this as a compiler/assembly code optimization, and effectively get the same results.

VI. Conclusion and Future Plans

Overall, this project was very interesting and fun, although I wish I could have given it more time. Learned a lot about not just vector processors, and assembly, but also computer architecture in general, specifically how memory works and memory banks work. I hope to work on this project more in the future, there is room to optimize the functional and timing simulator Python code, and the assembly code, and implement the optimizations.