Functional and Timing Simulator base on VMIPS Project Report

Tianheng Xiang tx701, Xinrang Tang xt2191

***Abstract*—In this project we construct a functional simulator and a timing simulator to enumulate an architecture with baseline architecture VMIPS. Three test cases including dot product, fully connected layer and convolution assembly is generated and tested and benchmarked using the simulators. An optimization regarding decreasing the vector memory bank conflict in order to decrease the vector memory access stall along with the instructions that has dependency with it is purposed.**

I. Project Summary

The project construct a functional simulator and a timing simulator to enumulate the data flow and time cycle for an architecture base on VMIPS. The functional simulator reads in an assembly file contains the VMIPS architecture intructions and simulate the actual data flow of the instructions to output the scalar memory, vector memory, scalar registers and vector registers. After the branching and memory addresses of vector/scalar load/store are resolved, the functional simulator will also output the resolved instructions and the vector length registers into two files and these files will serve a part of input of timing simulator. The timing simulator, on the other hand, only count the execution time in cycle of the instructions by taking various of configurations of the architecture into account, without simulating the actual data flow for simplicity. The output of the timing simulator will be a file contains the cycle spent of the assembly code in instruction level and the total cycle spent. We generate three assembly codes including dot product, fully connected layer and convolution and use them as the test case for this project. More details, including the usage, test case and simulating output, could be found in this github repo: project repo link.

II. Timing Simulator Design

The timing simulator consists of two parts: a frontend and a backend.

A. Frontend

The first part is a frontend to read all the inputs requires for this simulator, parse the instructions, resolve RAW, WAW conflicts. The front end maintains three instruction queues for scalar computation and scalar memory access, vector computation and vector memory access respectively. Two busyboards: one for vector register and another for scalar registers are used in order to keep track the data dependency of the in-flight instructions. Both RAW and WAW are considered as harzard and will be resolved in the front end by stalling the instructions. After the hazards are cleared, the instructions will be put into three intruction queue to be dispatched to the backend for computation/memory access cycle counting. All scalar related instructions are put into the scalar queue including but not limited to CVM, OIO, MTCL, MFCL and branching instructions.

B. Backend

The backend consists of four parts: a scalar related timer and vector compute timer, a vector memory handler and a vector memory bank. For each cycle, the frontend will try to dispatch each of the top of its three queues to check whether the backend is available. If not it will try the next cycle, if it successes the instructions will be handed to the backend to count the cycle of the instructions execution. Note that the three queues are independent to each other and could be dispatch in the same time, but only one instruction from each queue could be dispatched, if it succeed. The scalar related timer handles all the request from the scalar queue, the vector compute timer only handles instructions that involves vector operations and S__VV/S__VS instructions, the vector memory handler handles vector memory access. For the vector memory handler, once it receives the instruction and the memory access addresses, it will try to perform the memory access to the vector memory bank, and resolves the bank conflicts. The vector memory bank partitions the whole memory address in the interleaved way with the number of the vector memory bank. The vector memory handler contains two queues: one request queue and one return queue that both store all the memory addresses from the memory access instructions. For each cycle, the vector memory handler will only try to dispatch the first memory access request from the request queue to the memory bank, if this try fails the handler will wait until the next cycle. If the memory bank accepts the memory access request, the vector memory handler will pop this memory access request from the request queue. Once a memory bank is in use, a memory bank busyboard inside the memory bank records this status and prevent any other memory access to this bank until the current memory access in this bank is complete. After the completion of a memory access, this memory address is sent back to the vector memory handler, and this address will be deleted from the return queue in the memory handler to indicate this completion. When the return queue is empty, the vector memory handler will delete this vector memory access instruction and accepting new vector memory access instructions.

C. Configuration of the Architecture

The architecture could be configurated in 9 dimensions in three areas: dispatch queue parameters, vector memory parameters and compute pipeline parameters.

- Dispatch queue parameters
  - dataQueueDepth: the max number of instructions that a vector memory access queue can hold
  - computeQueueDepth: the max number of instructions that a vector computation queue can hold
  - scalarQueueDepth: the max number of instructions that a scalar memory access or computation queue can hold
- Vector memory parameters
  - vdmNumBanks: the number of banks in the memory bank
  - vlsPipelineDepth: the memory access latency in cycle
- Compute Pipeline parameters
  - numLanes: the max number of parallel instructions could execute for each vector add, mul and div instructions in the backend
  - pipelineDepthMul: the vector multiplication instruction latency in cycle
  - pipelineDepthAdd: the vector addition instruction latency in cycle
  - pipelineDepthDiv: the vector division instruction latency in cycle

III. Results

In this project, we generate three assembly codes including dot product, fully connected layer and convolution and use them as the test case. The dot product assembly performs dot product operation on two arrays with both 450 length. The fully connected layer assembly perform a matrix multiplication to a vector, the matrix is 256x256 and the vector has 256 elements. The convolution assembly will perform a convolution operation of a kernel to a matrix with 0 padding and stride value 1. The size of the kernel should not exceeds 8x8. In the convolution test case, we use kernel size of 8x8 and matrix size of 128x128.

We measured all three assembly codes with varying 6 dimensions of the configurations: number of vector data memory bank, number of vector compute lane, dispatch queue depth(we keep this value the same of all three queues), vector memory access pipeline depth(vector memory latency), vector multiplication pipeline depth(vector multiplication latency) and vector addition pipeline depth(vector addition latency). We omit the vector division latency since we do not use any vector division instruction throughout all the three test cases, other than that we cover all the dimensions of the parameters.

A. Dot Product Result

The performance result of the dot product is shown in Fig1. The number of cycles is propotional related to the vector memory access and computation pipeline, which is obivious since larger the pipeline depth, more cycles a related instructions will take. Since there are lots of vector memory access in our dot product assembly, we could see that the slope of the trend in vector L/S pipeline depth also increases with higher vector memory access latency. The cycle of instructions, however, is independent with the dispatch queue depth, which indicates that most of the time the queue is not full because of the RAW and WAW harzards, and the instructions is usually stalled. From the top right figure in Fig 1, we could see that the cycles executed decreases sharply initially with more parallel vector lanes as this provides more hardware capability exploiting parallelism. But the effect also diminishes with higher lanes as the parallelism of the dot product assembly is nearly fully exploited. From the top right figure in Fig 1, the number of cycles also rapidly decreases with initial increase in memory bank as it enables the memory bank to access memories more elements simultaneously. But once the number of memory bank exceeds the number of vector memory pipeline depth, in our test case the baseline memory pipeline depth is 11, there is no extra benefits for the extra banks that is more than the pipeline depth as there will not be any memory access stalls due to bank conflicts.

B. Fully Connected Layer and Convolution Result

The performance result of the dot product is shown in Fig 2. We could see the same trend in dot product in all the 6 dimensions we measured. However, we could also see an anomoly circled in red in the top left figure in Fig2: the number of cycle executed in convolution assembly is higher when there is 13 memory banks than 10 memory banks. By further analyzing the per-instruction latency of this case, we find out that this is due to the use of LVI in the convolution assembly. LVI no longer access a complete set of continuous memory address but it may jump to different continuous memory subsets, which causing more bank conflicts and incuring more stalls not only in the memory access, but also for the rest of the instructions that has dependency on this instructions.

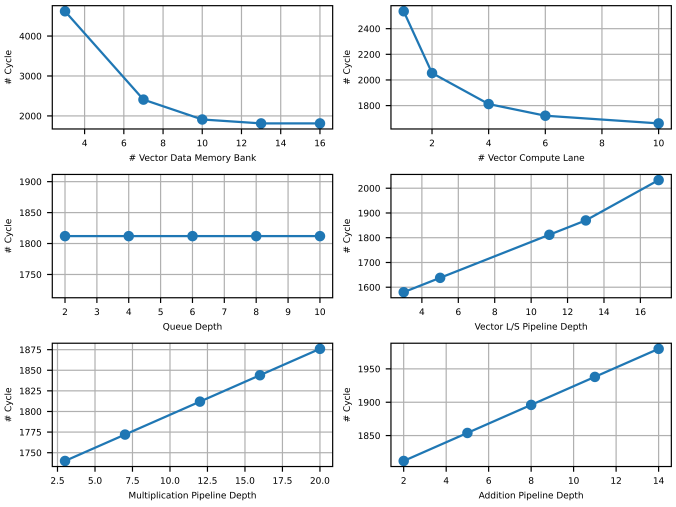

Fig. 1. Dot Product Performance

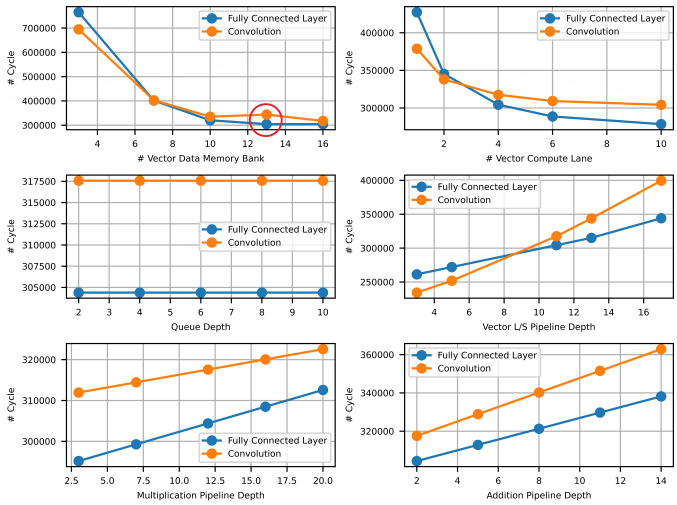

Fig. 2. Fully Connected Layer and Convolution Performance

IV. Optimization

Base on the results of the test cases, we could see that the most influencial factor to the performance is vector memory access, and we can also validate this through examining the per-instruction latency: most latency occurs in the instructions that perform vector memory access and any instructions that has dependency on these memory access. Thus, one optimization I purpose is that, instead of sequntially request the memory access base on the its order in the request queue, we can try to request all the memory inside the request queue if there exist any bank conflicts in the previous tries.

After the optimization of the memory access, shown in Fig 3 the convolution performance(green line) is now achieving better performance comparing with the previous edition of the timing simulator. However, the cycle number bounce back when the number of vector data memory bank is 16, and reaches the exactly same number of cycle with the baseline configuration. This result is confusing and needs further inspection. Also the time that searching for the available memory address is not included in the cycle counting, thus the actual effect of this optimization is doubtful.

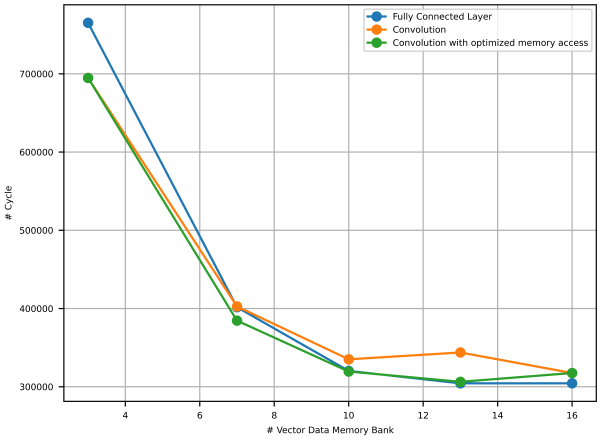

Fig. 3. Performance with optimized memory access

References

[1] J. L. Hennessy, K. Asanovic, D. Goldberg, and D. A. Patterson, *Computer Architecture: A quantitative approach*. Morgan Kaufmann, 2004.