Vector Processor Simulator

Akshath Mahajan, avm6288   Rugved Mhatre, rrm9598

Abstract—**Vector architectures are increasingly prominent for their efficiency in handling data-parallel workloads, with even Meta AI incorporating a vector core in their latest ML training and inference accelerator [1]. To comprehensively understand these architectures, this study develops functional and performance simulators for a simple Vector Processor based on VMIPS architecture [2]. Through a test bench comprising of VMIPS assembly code for operations like dot product, fully connected layer, and convolution layer, we explore the design space to identify optimal configurations. Additionally, we discuss and implement potential future optimizations to enhance performance on the test bench, thereby providing insights into the capabilities and potential advancements of Vector Processors.**

I. Introduction

Modern vector architectures have emerged as formidable alternatives to superscalar processors, offering significant advantages across diverse compute-intensive applications. From multimedia processing exemplified by VIRAM [3] architecture, to broadband and wireless communication tasks, and even specialized domains like bioinformatics with systems such as Cray X1 [4] and Tarantula [5]. Notably, the recent integration of vector processors into machine learning accelerators like MTIA [1] underscores their adaptability to evolving computational demands. The inherent abundance of data-level parallelism in these applications enables vector processors to concurrently execute numerous arithmetic and memory operations while issuing a single instruction per cycle. This characteristic not only augments performance but also enhances power efficiency. Moreover, vector processors offer a compelling balance between performance and programmability, rivaling custom designs in data-parallel tasks while retaining the flexibility of general-purpose processors. Additionally, their simplicity in design and scalability with CMOS technology further solidify their appeal compared to traditional superscalar processors.

To gain a comprehensive understanding of the intricate workings of vector processor architecture, this study endeavors to develop both a functional simulator and a performance simulator for a simple vector processor based on the VMIPS architecture [2]. Functional simulators serve as indispensable software tools, facilitating the emulation of vector processor behavior. These simulators offer developers a means to thoroughly test and debug software tailored for vector processors without necessitating access to physical hardware, which can be both costly and less conducive to experimental flexibility. Specifically, they empower software developers to validate algorithms and applications, as well as to debug software efficiently. On the other hand, performance simulators play a crucial role in forecasting the performance attributes of computer systems or applications across varied conditions. Such simulators involve modeling the behavior of hardware components, including the functional units, registers, memory, alongside software workloads to estimate key metrics such as execution time and instruction throughput. These performance simulators prove invaluable in architecture design, allowing architects to explore diverse design choices and trade-offs prior to finalizing a hardware architecture. This proactive approach aids in optimizing designs for enhanced performance, power efficiency, and cost-effectiveness. Additionally, developers leverage performance simulators to assess the performance of their software applications across different hardware configurations, facilitating the identification of performance bottlenecks, code optimization, and the fine-tuning of software parameters to achieve superior execution speeds.

To validate the efficacy of our developed functional and performance simulators, we construct a comprehensive test bench comprising of VMIPS assembly code tailored for three fundamental operations crucial in machine learning applications. These operations include a 450×450 elements dot product, a 256×256 elements matrix multiplication (representing a fully connected layer), and a 2D convolution layer characterized by a frame size of 256×256 elements, a kernel size of 3×3 with zero padding, and a stride of 2 (representing a convolutional layer). With the test bench in place, we proceed to implement and assess the functionality of our functional simulator, thereby validating the core operations of the vector processor. Subsequently, leveraging the capabilities of our performance simulator, we explore the design space, aiming to strike an optimal balance between performance metrics and vector chip complexity and area. Through this iterative process, we identify configurations that not only maximize performance but also ensure practicality and efficiency in real-world deployment scenarios.

With insights from both the functional and performance simulators, we discuss future optimization aimed at enhancing the capabilities of our vector processor architecture. We proceed to implement select optimizations with the objective of improving the efficiency and performance of our vector processor.

This paper makes the following contributions:

1) Publicly accessible GitHub Repository that contains the code for our functional and performance simulator
2) Evaluation of the test bench, and design space exploration to find optimal configuration for our processor
3) Discussion of future optimizations that will improve the performance of our processor

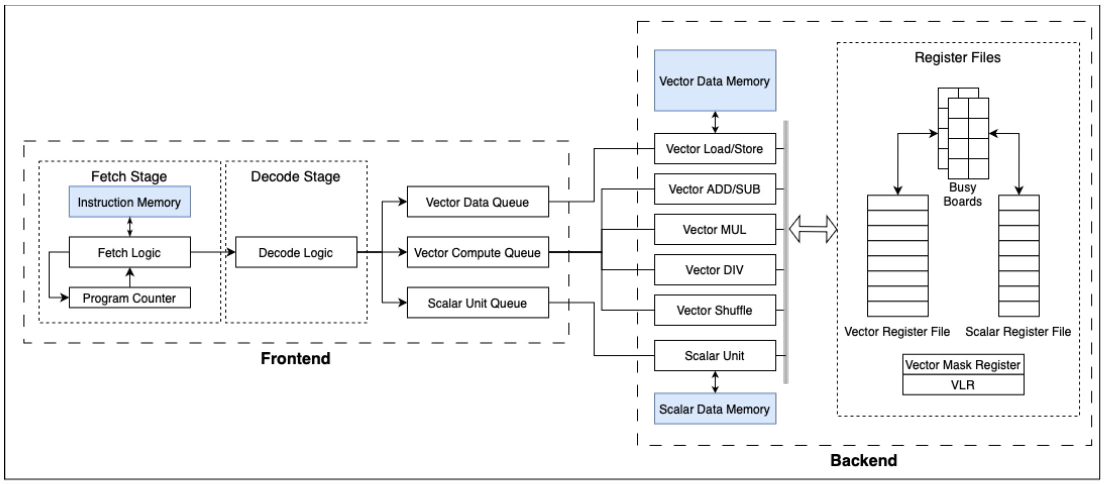

Fig. 1. Vector Processor Architecture

II. Architecture

The architecture of the vector processor, depicted in Figure 1, comprises six pipelined functional units: scalar unit, vector load/store unit, vector add/subtract/logic unit, vector multiply unit, vector divide unit, and vector shuffle unit. Each functional unit interfaces with Vector Register Files (VRFs) and Scalar Register Files (SRFs). There are eight VRFs, each capable of holding up to 2048 bits, equivalent to 256 bytes or 64 32-bit elements. The maximum vector length supported by our Instruction Set Architecture (ISA) is 64. Additionally, the architecture includes eight SRFs, each accommodating 32 bits. Two specialized registers are present: the Vector Length Register (VLR) with a size of 32 bits and a Vector Mask Register (VMR) containing 64 1-bit values.

The word-addressable memory is bifurcated into vector data memory (VDMEM) and scalar data memory (SDMEM). VDMEM has a capacity of 512KB, while SDMEM's capacity is 32KB. VDMEM is banked and exclusively connected to the vector load/store functional unit, whereas SDMEM solely interfaces with the scalar functional unit. Presently, for simplicity, these functional units support only integer data types.

All the instructions are present in instruction memory (IMEM), separately from the data memory. Only the fetch logic of our processor is connected with IMEM.

III. Functional Simulator

In the functional simulator, we translate the previously described architecture into Python classes to assess the functionality of our processor. Our simulator consists of separate files for Code, VDMEM, and SDMEM. The Code file contains the assembly program, while the VDMEM and SDMEM files store the initial states of vector data memory and scalar data memory, respectively. Utilizing the line number of instructions in the Code file as our program counter, we read the IMEM (i.e. Code file) and fetch a new instruction every cycle until encountering a `HALT` instruction. Instructions are filtered, excluding comments indicated by '#' in the Code file. Upon fetching an instruction, it is decoded and executed. The decode logic employs an `if-elif-else` block to decode instructions based on the instruction word. Operands are fetched, and values in the register files are read or updated accordingly. Additionally, VDMEM or SDMEM files are read/written based on vector load/store or scalar load/store instructions. The execute logic computes only for vector lengths specified in the VLR register and saves results for elements where the Vector Mask bits are set. Branch instructions are handled, updating the program counter based on branch outcomes. The simulator's output is a resolved execute flow file, which is input for our performance simulator. This file unrolls all loops and explicitly lists all accessed addresses for streamlined processing in the performance simulator. We also include functionality to mention the VLR value whenever it updates, aiding the development of the performance simulator. The functional simulator also outputs the final states of VRF, SRF, VDMEM, and SDMEM for result verification. Finally, we rigorously test all ISA instructions to validate the functional Simulator before establishing our test bench.

IV. Performance Simulator

In the Performance Simulator section, we extend our previously described architecture from Figure 1 to delve deeper into the hardware components constituting the processor, implementing them as Python classes. Additionally, we incorporate three queues - vector data queues, vector compute queue, and scalar compute queue. Moreover, we introduce a busy board to monitor all active VRFs, SRFs, and functional units. For clarity, we logically divide the architecture into a frontend and a backend.

The frontend is tasked with fetching new instructions from the resolved code file, decoding these instructions, and populating the data/compute queues. The decode logic also examines for any data or structural hazards by consulting the busy boards before queue population. Each cycle, an instruction from the front of all queues is retrieved and sent to the backend for execution.

The backend is responsible for executing all in-flight instructions. Each functional unit maintains its status and records active in-flight instructions. Additionally, busy boards for VRF and SRF track all registers used by in-flight instructions. Upon completion of instruction execution, they are removed from the busy boards and functional units, making room for subsequent instructions.

We delve deeper into the code flow, executing the following steps in every cycle: first, all active instructions are executed; then, new instructions are decoded and dispatched to the respective queues if no data or structural hazards are encountered. Next, the instruction at the head of the queue is popped, and a new instruction is fetched. In the execute function, we examine all busy functional units, decrementing cycles for in-flight instructions. Upon completion of execution, if there are no remaining cycles, the functional unit is marked as available, and VRFs and SRFs are cleared from the busy board. In the decode function, we create a Python dictionary for each instruction to store relevant parameters, assign a functional unit based on the instruction word, and determine source operands' values and types (scalar/vector) and the instruction's destination. We calculate the total cycles required for the instruction, accounting for bank conflicts if necessary. In the dispatch to queue function, we check for data and structural hazards and decide whether to stall the instruction or dispatch it to the appropriate queue. The pop instruction from queue function assigns instructions to functional units for execution in the next cycle. If the HALT instruction has not been encountered, we fetch a new instruction from the resolved code flow file, update the clock cycle, and continue looping until a no-operation command is received from the execution of the HALT instruction.

The output of our performance simulator is a result file detailing the total number of cycles required by the program. Additionally, we have integrated a Timing Diagram generation code, which produces a CSV file tracking the status of instructions at each cycle, aiding in debugging the performance simulator.

Finally, we conducted tests on simple cases to manually compare results and verify our code's functionality. Subsequently, we proceeded to test our performance simulator on the test bench validated in our functional simulator.

V. Experimentation and Results

A. Test Bench

We develop a comprehensive suite of assembly code tailored for three pivotal operations crucial in machine learning applications. These operations encompass a dot product involving 450×450 elements, a matrix multiplication with dimensions of 256×256 elements (representing a fully connected layer), and a 2D convolution layer characterized by a frame size of 256×256 elements, a kernel size of 3×3 with zero padding, and a stride of 2 (representing a convolutional layer).

To optimize performance, we employ stripmining techniques for efficient computation over large vector sizes. Additionally, we implement strided loads for matrix multiplication to enhance code efficiency. In the case of convolution, we store kernel values in scalar memory to streamline calculations.

We rigorously evaluate the functionality of the test bench using our functional simulator, ensuring accuracy and reliability of results through verification against actual calculations.

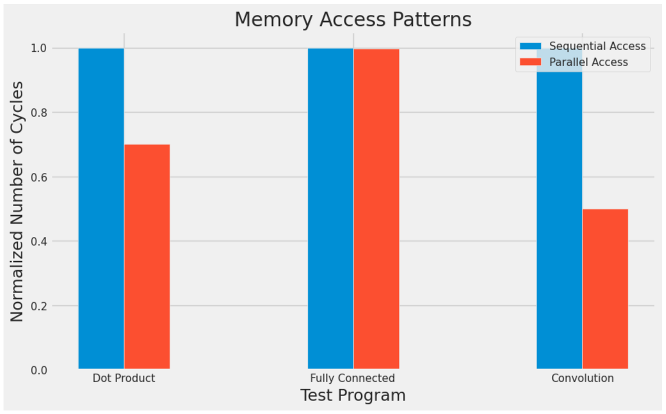

Fig. 2. Memory Access Patterns

B. Memory Access Patterns

In our investigation of memory access patterns within our vector processor, we juxtapose two distinct approaches. In legacy, low-bandwidth memory systems, sequential access of memory banks was prevalent. Conversely, modern, high-bandwidth memory systems facilitate parallel access of memory banks. This parallel access affords the ability to load and store vectors in fewer cycles, thus optimizing memory utilization. Figure 2 illustrates a comparison of both methodologies. Notably, significant performance enhancements are observed in Dot Product and Convolution tests, indicative of the efficacy of parallel access. However, in the Fully Connected Layer test, our code yields equivalent performance, suggesting that the test is constrained by bank conflicts arising from strided access patterns inherent in matrix multiplications.

TABLE I
Base Configuration

Configuration	Value
Dispatch Queue Parameters	
Vector Data Queue Depth	4
Vector Compute Queue Depth	4
Scalar Compute Queue Depth	4
Vector Data Memory Parameters	
Vector Data Memory Banks	16
Vector Data Memory Bank Busy Cycles	2
Vector Load/Store Pipeline Depth	11
Vector Functional Unit Parameters	
Number of Lanes	4
Vector ADD Pipeline Depth	2
Vector MUL Pipeline Depth	12
Vector DIV Pipeline Depth	8
Vector Shuffle Pipeline Depth	5

C. Design Space Exploration

Our processor's fundamental configuration is detailed in Table I. To assess its performance across various scenarios, we employ our performance simulator, maintaining the base configuration while systematically altering one parameter at a time. This approach allows us to ascertain the number of cycles required for each program within our test bench. Notably, all experiments are conducted utilizing the parallel memory access pattern to ensure consistency and relevance across evaluations.

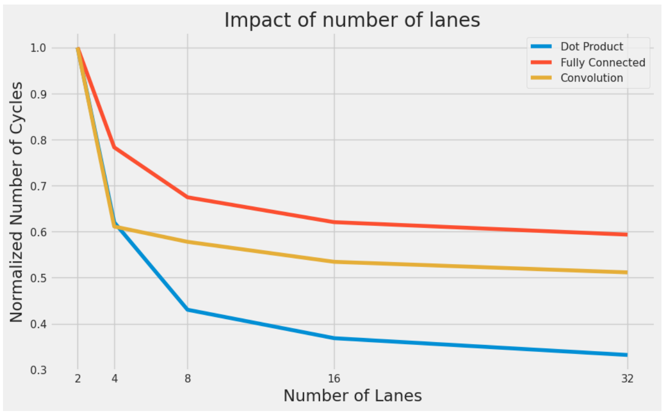

Fig. 3. Varying Number of Lanes

1) Varying Number of Lanes: To establish a baseline, we configure our system with 2 lanes and normalize the cycle count for other configurations accordingly. The results, depicted in Figure 3, illustrate a consistent performance improvement across the test bench as the number of lanes increases. This improvement can be attributed to the accelerated execution of instructions through the functional units. However, it is important to note that adding more lanes poses practical challenges in hardware implementation and yields only marginal improvements. Based on our findings, we determine that 8 lanes represent the optimal value for our test bench, striking a favorable balance between performance enhancement and implementation complexity.

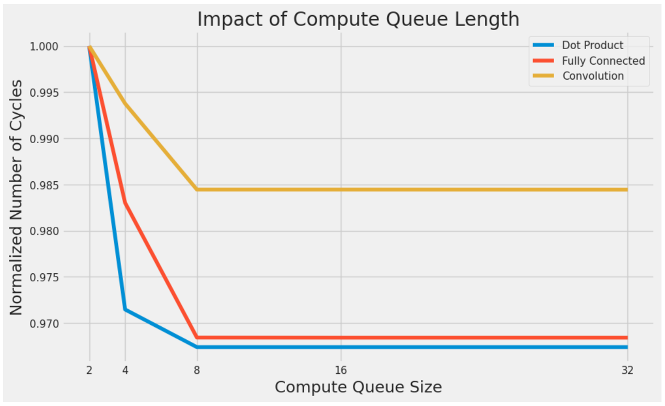

Fig. 4. Varying Compute Queue Depth

2) Varying Compute Queue Depth: To establish a baseline, we configure our system with a compute queue depth of 2 and normalize the cycle count for other configurations accordingly. The results, depicted in Figure 4, demonstrate a consistent performance improvement across the test bench as the queue depth increases. However, the magnitude of performance enhancement is not substantial, indicating that we are constrained by the number of functional units. Despite deeper queues, the presence of data and structural hazards limits the number of instructions available for queuing. Following our evaluation, we opt for a compute queue depth of 8, as beyond this point, there is no discernible performance improvement. This decision strikes a balance between performance enhancement and implementation complexity.

3) Varying Data Queue Depth: We further explore variations in the data queue depth; however, as previously mentioned, our system is constrained by having only one Vector Load/Store unit. Consequently, increasing the queue depth does not result in improved performance.

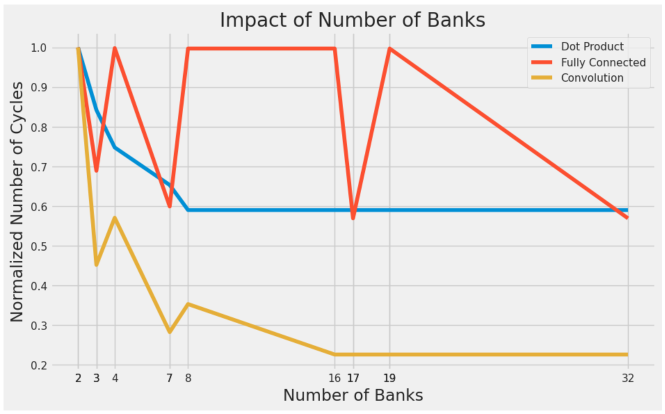

Fig. 5. Varying Number of Memory Banks

4) Varying Number of Memory Banks: Furthermore, we investigate variations in the number of memory banks. Increasing the number of banks enhances load/store speed, and utilizing a prime number of banks aids in reducing bank conflicts. The resulting improvement is illustrated in Figure 5.

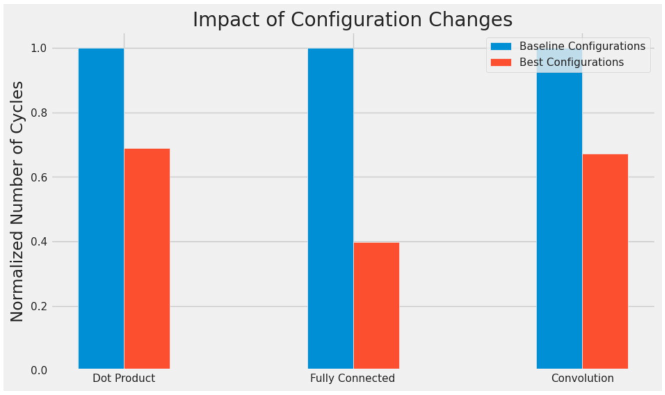

Fig. 6. Optimal Configuration

5) Optimal Configuration: Based on the evaluations, we determine an optimal configuration for our vector processor with a Compute Queue Depth of 8, Data Queue Depth of 1, 8 lanes, and 17 Memory Banks. The enhanced performance resulting from this configuration is illustrated in Figure 6. This configuration strikes a good balance between performance and implementation complexity.

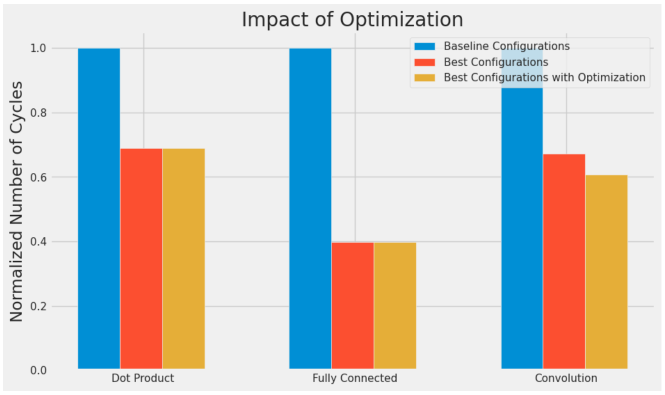

Fig. 7. Increasing VRF Read Ports

VI. Optimizations

A. Increasing VRF Read Ports

In our optimization efforts, we propose and execute a strategy to increase the number of read ports on our VRFs. By expanding the read ports, we enable two functional units to simultaneously access the same register, thereby enhancing parallelism in our tests. The enhanced performance resulting from this optimization on our best configuration is illustrated in Figure 7, where it is compared to the baseline. However, we notice that only convolution has an improvement, showing that the other instructions cannot extract the functional unit parallelism in the best configuration i.e. we seem to have reached the roofline on these two tests. However, we had noticed that in the baseline configuration, Dot Product and Fully Connected do improve a bit.

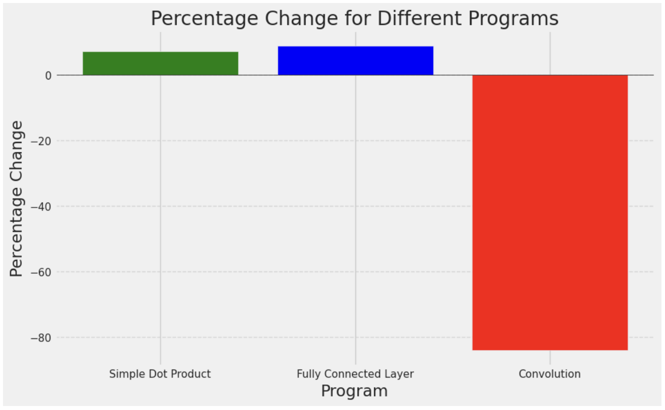

Fig. 8. Increasing VLR to 128

B. Increasing Vector Length

Next, as an additional optimization, we experiment with increasing the vector length. While we observe improvements in the Dot Product and Fully Connected tests with VLR = 128 in Figure 8, we encounter a performance decline in the Convolution test. This discrepancy primarily arises from the design of the convolution test, which was originally tailored for VLR = 64, with calculations optimized accordingly.

VII. Conclusion

Throughout this study, we delve into vector architecture design and develop functional and performance simulators for a VMIPS-based vector processor. We perform rigorous evaluation of different design decisions and identify an optimal configuration that balances the performance and complexity of the architecture. Additionally, we propose two optimizations that improve the processor performance – (1) Adding more Vector Read Ports, and (2) Increasing the vector register lengths. We demonstrate how the design-space exploration helps in determining an optimal configuration for the microarchitecture, and the effect of the proposed optimizations on performance. Overall, this study has deepened our understanding of vector processors and laid a foundation in building functional and performance simulators.

Acknowledgment

We extend our sincere gratitude to Prof. Brandon Reagen and the course CAs Tim and Vidyut for their invaluable support and guidance throughout this remarkable course. Their assistance has been instrumental in navigating the intricacies of this fascinating yet challenging project.

References

[1] “Our next generation Meta Training and Inference Accelerator — ai.meta.com,” https://ai.meta.com/blog/next-generation-meta-training-inference-accelerator-AI-MTIA/, 2024.
[2] D. A. Patterson and J. L. Hennessy, *Computer architecture: a quantitative approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.
[3] C. Kozyrakis and D. Patterson, “Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks,” in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, 2002, pp. 283–293.
[4] D. Alpert, “Scalable microsupercomputers,” *Microprocessor Report*, vol. 17, no. 5, pp. 1–6, 2003.
[5] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec, “Tarantula: a vector extension to the alpha architecture,” in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002, pp. 281–292.