Resource-efficient RISC-V Vector Extension Architecture for FPGA-based Accelerators

Md Ashraful Islam
ashraful@arch.cs.titech.ac.jp
School of Computing, Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan

Kenji KISE
kise@c.titech.ac.jp
School of Computing, Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan

ABSTRACT

For the increasing demands of embedded computation, hardware accelerators are widely used with processors. FPGA provides flexibility to design such accelerators because it is a programmable device. But developing a custom accelerator for each application is time-consuming and not reusable. On the other hand, vector processing brings the opportunity to accelerate computation by taking advantage of data-level parallelism.

This paper presents the architecture of a scalable soft Vector Processing Unit for FPGA based on a subset of the RISC-V vector extension instruction set. Maximum vector length and the number of lanes are configurable in the proposed architecture. We have integrated our proposed vector processing unit into a 32-bit scalar RISC-V core and implemented it in FPGA. The implementation result shows that our proposed architecture consumes significantly less FPGA resources and has more than four times frequency improvement than other vector processing units. It achieves 11.9 giga operation per second for 8-bit integer convolution operation. We demonstrate that the performance of the proposed vector processing unit is scalable with maximum vector length and the number of lanes.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; Architecture.

KEYWORDS

RISC-V, Soft Processor, Vector Extension, Variable Precision, IoT, Edge computing

ACM Reference Format:
Md Ashraful Islam and Kenji KISE. 2023. Resource-efficient RISC-V Vector Extension Architecture for FPGA-based Accelerators. In *The International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies 2023 (HEART 2023), June 14–16, 2023, Kusatsu, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3597031.3597047

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HEART 2023, June 14–16, 2023, Kusatsu, Japan
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0043-9/23/06…$15.00
https://doi.org/10.1145/3597031.3597047

1 INTRODUCTION

Cloud-based data processing approach is limited by bandwidth, data privacy, and latency. The demand for real-time workloads on the embedded systems at the edge is growing. Many of these embedded applications are data-intensive, but the computation performance is limited by the scalar processor due to a lack of data-level parallelism. In FPGA-based design, such data-parallel applications can be addressed by custom-designed hardware accelerators. However, the development of embedded applications with such a custom accelerator requires hardware design experience. On the other hand, a vector processing unit allows the developer to exploit such data-level parallelism without redesigning the accelerator every time.

The advent of quantized integer-based AI inference enables the development of different applications such as image classification, voice recognition, and many others at the edge. These applications require integer computation such as multiplication, addition, dot operation (for convolution), min/max, etc., with varied numbers of parallel operations. Single instruction multiple data (SIMD) allows executing operations over the wide registers of a fixed length such as ARM NEON[10], AVX-512[6] etc. On the other hand, in a Vector Architecture, vector length can be changed by application up to a predefined maximum vector length (MVL) such as ARM SVE[16], RISC-V vector extension[7].

RISC-V vector extension (RVV) is an open-source instruction set architecture(ISA) that allows researchers to develop their own architecture to implement the vector processing unit. This ISA defines arithmetic instructions for integer, fixed point, and floating point data types. Since FPGA is a resource-constrained device implementing the vector processor with all data types will consume a lot of resources. Support for all of those data type may not be required for a specific application. In this research, we have focused on the integer data type implementation, though the same architecture can be extended for other data types as well. The ISA also defines the vector memory load and store instructions for memory operations, reduction instructions to process the contents of a single vector register, permutation instructions to shuffle the elements in the vector register, and vector mask instructions for conditional executions.

In the past, several soft vector processing units and co-processors have been presented, such as VESPA[17], which is a vector co-processor coupled to a MIPS processor, VIPERS[18], a vector processing unit. Both VESPA and VIPERS have poor support for the variable width integer data type. While VEGAS[5] supports the variable width data type, but it used scratchpad memory instead of the vector register file and did not have load-store instruction. VENICE[14] or MXP[15] has reduced the area of VEGAS, but still,

it relies on scratchpad memory, and custom instruction need to be issued by the scalar core.

Initial RISC-V vector extension-based architecture was proposed in Ara[3] based on RVV version 0.5. This architecture is proposed for ASIC implementation and has a vector sliding unit (SLDU) that performs inter-lane data transfer involving data from all the vector register file (VRF) banks at once. Such banked VRF and inter-lane connection to SLDU will be a major overhead for the FPGA soft vector processing unit to implement multiple lanes. Because the number of elements to be transferred among the lanes will increases with lanes. Thus interconnecting these elements over the multiple lanes VRF banks will require larger crossbar or other interconnect with multiple ports. Another key point is that Ara architecture does not have support for vector reduction sum operation, which is important for the dot/convolution operation.

Vicuna[13] is a timing-predictable vector processor implemented on FPGA. It has implemented the vector register file as monolithic multi-ported RAM, and vector functional units read the whole vector register into a large shift register, which will overwhelm the FPGA resource with increasing vector length. SIMD-based vector processing unit was proposed in [2][1], which has a 2-stage pipeline that might limit FPGA operating frequency, and it does not support reduction instruction.

A pluggable vector processing unit is presented in [11] with banked SRAM as VRF. To reduce the cross-lane communication for vector permutation instruction, VRF banks are consolidated as a single unit and maintain the locking protocol to avoid hazards over different functional units. However, such centralized VRF with multiple bank arbitrators requires much FPGA logic, and complexity grows with the number of functional units and vector length. Spatz[4] is a vector processing unit with a shared L1 cache. Again it is based on centralized VRF and implemented using the latch for ASIC.

In this work, the architecture of a scalable soft vector processing unit (VPU) is presented supporting a subset of RISC-V Vector extension ISA similar to [4] except for permutation instructions. It is pluggable into any scalar RISC-V processor core. Development of an embedded application for such VPU is easy due to the support of the compiler. Exploiting data-level parallelism will accelerate the performance of such applications. The main contributions of this paper are summarized below:

- We introduce a 5-stage pipeline-based VPU, which is parameterized and designed for FPGA. Our design can accommodate other functional units to support different vector instructions.
- We show that our distributed VRF-based architecture provides flexibility to configure the number of lanes and the MVL. Our VPU performance is scalable with a number of lanes and MVL.
- We present an efficient reduction sum operation in conjunction with a vector multiplier to reduce the latency of integer dot/convolution operation.
- We implement the VPU in FPGA for different parameters and show that FPGA resource utilization is much lower than the other proposed designs, and the operating frequency of our VPU has a negligible drop with the number of lanes.

This paper is structured as follows: Section 2 describes the architecture of the proposed VPU. Section 3 presents the FPGA implementation results and compares our work to published works, and section 4 concludes the paper.

2 ARCHITECTURE

We implement a subset of the RVV[7] v1.0 ISA extension (Zve32) for integer computation which is commonly used for acceleration such as convolution, matrix multiplication, etc. It does not support widening and permutation instructions. However, it supports integer scalar move instructions. RVV defines the vector element length as *ELEN*, which is maximum size of a single vector element in bits. Users can select element width *SEW* from 8-bit to ELEN bits. The number of bits in a single vector register is defined as vector length (*VLEN*). Depending on the implementation VLEN can be 32-bit to 65,536-bit. The number of elements in a single vector register can be found by dividing the VLEN by SEW. In this architecture, ELEN is 32-bit and SEW can be 8-bit, 16-bit, or 32-bit. The number of lanes can be configured from 1 to 16, and the maximum vector length (*MVL*) can be configured from 128-bit to 4096-bit.

RVV also defines the vector control status register (CSR) which is used to by the software to control the execution and check the status. For example, if application software wants to process the vector register contents as 8-bit element, the user should write the vector type (*vtype*) CSR's SEW field to 0 by executing vector configuration instruction.

Figure 1 shows the block diagram of the proposed vector microarchitecture. Our proposed vector processing unit (*VPU*) executes the vector instructions irrespective of the scalar RISC-V core architecture. VPU can have single or multiple lanes of up to 16 lanes. Each lane has a 32-bit data path consisting of its own part of the vector register file, an execute and write back unit. Each lane can perform 4/2/1 operations in parallel if the selected element width (*SEW*) is 8/16/32 bit, respectively.

Vector instructions are executed in the 5-stage pipeline - decode (DEC), sequencer (SEQ), vector register file (VRF), execute (EXE), and writeback (WB). Once the scalar processor fetches and decodes the vector instruction, it pushes that vector instruction to the vector extension queue, given that branch prediction and other hazards and exceptions are resolved. For some vector instructions, a scalar operand from the scalar core register file is required, which is also pushed into the same queue along with the vector instruction. Few instructions requires the single vector register element value from the the VPU to the scalar core. For such instructions the write back unit of lane 0 will return the value directly to the scalar core. While executing such instructions, the scalar core should be stalled and wait for the return value from the VPU. For other vector instructions the scalar core can execute in parallel to the VPU, given that there are no exceptions or race conditions between the scalar core and the VPU. The functional description of major blocks is given in the following sections.

2.1 Decoder

Vector Decoder reads 32-bit instructions from the instruction queue, which are pushed by the scalar RISC-V core. Then it decodes the instruction into the appropriate control signals and required source

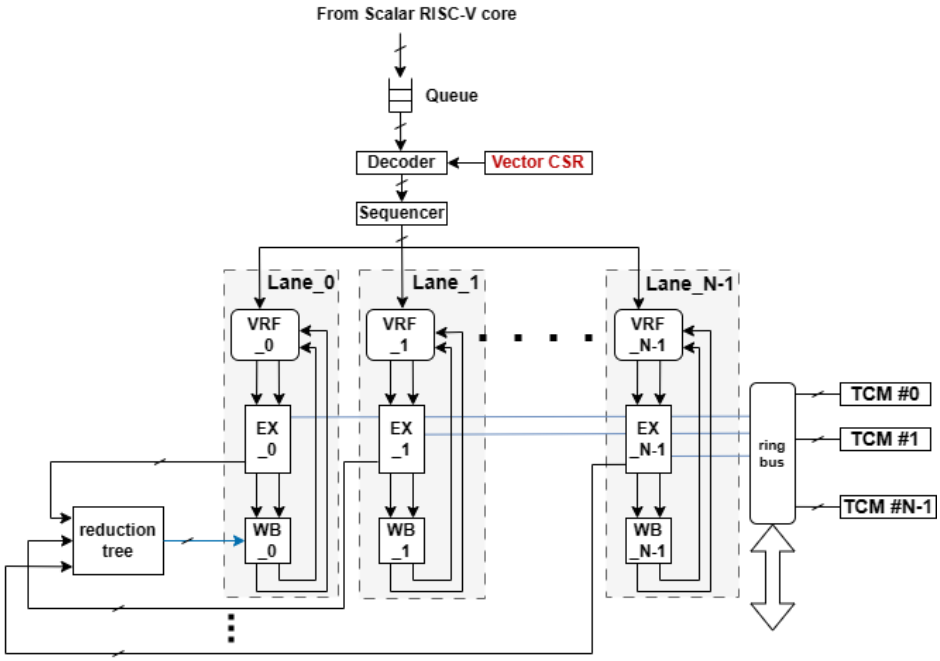

Figure 1: Proposed Vector Processing Unit's microarchitecture

and destination addresses of the vector registers for the subsequent pipeline stages. If the instruction requires a scalar value from the scalar core register file, then that value is also popped from the queue.

Decoder output is passed to the sequencer stage and waits for the sequencer to be ready if the previous instruction in the sequencer is not finished. It then reads the next instruction from the queue if the queue is not empty. For vector configuration instruction (e.g. vsetvli), the decoder reads and sets the vector CSR values and return the new value of the vector length to the scalar core. The updated vector configuration values are provided by the decoder to the sequencer for the forthcoming instructions executions.

2.2 Vector Register File and Sequencer

In our architecture, each lane has a partial set of the vector register file. The maximum vector length (MVL) of the vector register is parameterized and is equally divided among the lanes. Since the VRF is distributed among the lanes and there is no interconnections among VRF in different lanes, this architecture does not not support instructions which requires inter-lane VRF data exchange. Owing to that fact, this architecture does not support widening and permutation instructions. Because those instructions requires to access the VRF elements over the multiple lanes. Figure 2 shows the vector register file organization for MVL=512 and the number of lanes NLANE=4 and SEW=32 (i.e., each element e in this figure is 32-bit). Elements of each vector register are indexed as e0, e1, e2, and so on. Our motivation for this VRF organization is the following.

- VRF is implemented using block RAM (BRAM), which is available in the FPGA as a memory resource. BRAM is usually having a few kilobytes of RAM (in modern Xilinx FPGA devices, it is a minimum of 4KB). According to the RISC-V Vector ISA, the VRF has 32 registers. For 32-bit element width, it requires only 128 Bytes, which underutilizes the available BRAM memory. To increase memory utilization, we have allocated more bits to a single vector register by increasing the vector length (VLEN). Thus for any given SEW, the the number of elements in a single vector register will increase.
- Increasing the number of elements per vector register increases the MVL, which is beneficial for application acceleration due to reduced software loop iteration and may hide memory latency for long memory bursts.
- Implementation of a monolithic register file with banked memory for multiple lanes is inefficient for FPGA as it requires the interconnection of memory banks, and area increases exponentially with the number of lanes. Each lane has its own VRF allocated with a partial set of elements from 32 vector registers and does not require any interconnection for inter-lane data transfer from VRF. Thus implementing multiple lanes are straight forward and can be configured according to the application requirements.

The VRF is implemented as a 2-read and 2-write port (2R2W) register file using the live value table (LVT) technique[9]. Two read ports to perform two register operands read (vs1/vs3 and vs2) operation. The scalar value from the scalar core is multiplexed with the vs1/vs3 vector register read data. One register file write port is used for a vector load operation, and another write port is used for ALU and multiplication operations. There is no data bypassing path from the write port to the read port.

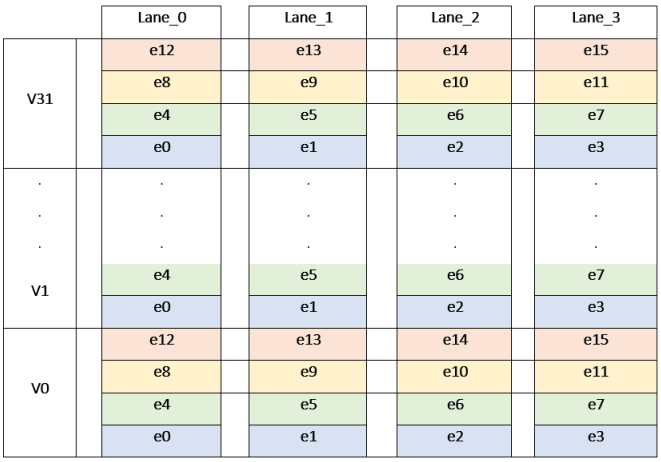

Figure 2: Vector register file organization

As the vector registers have multiple elements, the vector execution has to iterate over this number of 32-bit elements, we named it SEQLEN (MVL/(NLANE X 32)). For example, according to figure 2, lane 0 has to process elements e0, e4, e8, and e12 for the given instruction, thus requiring four iterations. The sequencer performs this iterative execution over multiple lanes in parallel.

Sequencer also checks for the data hazards of read after write (RAW), write after read (WAR), and write after write (WAW) before dispatching it to VRF for subsequent execution. Sequencer tracks the state of 32 vector registers of VRF using a state table to detect data hazards. The vector register file state of the destination register (*vd*) is updated by the sequencer, and during the writeback corresponding *vd* register state is restored by the writeback module. If the hazard is detected, the sequencer inserts a bubble for the subsequent pipeline stages.

The pipeline diagram of our VPU is illustrated in figure 3. In our design, the execution stage takes two clock cycles, and the other stages are one clock cycle. While the sequencer is doing iterative execution, the decoder is stalled by the sequencer. In figure 3, we have shown the example for SEQLEN=2. Therefore the sequencer has to iterate two times (SEQ 0 and SEQ 1) to execute the instruction. In this example, instruction K+2 has a read-after-write (RAW) data hazard with instruction K+1. As a result, the sequencer stalls two clock cycles. From this figure, it can be easily verified that if the SEQLEN is greater than or equal to 4, then there will be no stall due to data hazard except for the reduction operation, which will be explained in section 2.3.

2.3 Execution and Write back

Figure 4(a) shows the block diagram of the execution (EX) and writeback (WB) stages. The execution unit has a vector integer unit (VINT) and a vector load store unit (VLSU). VINT executes the vector integer arithmetic (e.g. vadd, vsub, vmin, vmax, vsll, vsrl etc.) and multiplication instruction (vmul, vmulh etc.) and VLSU performs vector load-store instructions(vle, vse). Vector multiplication instructions are executed in the VMUL module, and the VALU module performs vector integer arithmetic and logical instructions. The vector division instructions are not implemented, and shift instructions are optional, as those are less frequently used in many applications.

VINT takes two operands from the VRF and takes two clock cycles to compute the result in the pipeline. It performs four operations for SEW=8-bit, two operations for SEW=16-bit, and one operation for SEW=32-bit. For variable precision (8/16/32-bit) vector multiplication, we have used a similar approach presented in [12]. The block diagram of VMUL is shown in figure 4(b). In the first clock cycle, it calculates the partial product from the 16-bit input multiplicand and multiplier (for 8-bit multiplication, it is signed/unsigned extended to 16-bit) and stored in registers. In the second clock cycle, based on the multiplier precision, it performs a summation of the partial products and calculates the multiplication result put into a register (R).

Since the VRF can produce only two operands, some instructions that require three inputs, such as multiply-accumulate instruction, will require two operations, for example, multiply operation and then add operation. Implementing three read ports (3R2W) for VRF operand read will use up two more BRAMs compared to 2R2W. However, if the target application is multiply-accumulate intensive, one can configure the VRF as 3R2W without any architectural changes.

Since many accelerators manipulate multiplication and convolution functions, we have optimized the dot operation for these functions. We do not support widening operation, but we fuse widening multiplication (VWMUL) and widening reduction (VWREDSUM) instruction internally to support dot operation. The VWMUL instruction calculates the multiplication results and stores them in the internal accumulator (ACC) as shown in figure 4 (b). This accumulates the summation over the elements of the destination vector register of its own lane. For example, if there are eight elements per vector register per lane (SEQLEN=8), then the accumulator will store the summation of 8 vector element multiplication results. During VWREDSUM instruction, the reduction tree module does a summation of these accumulators from all the lanes.

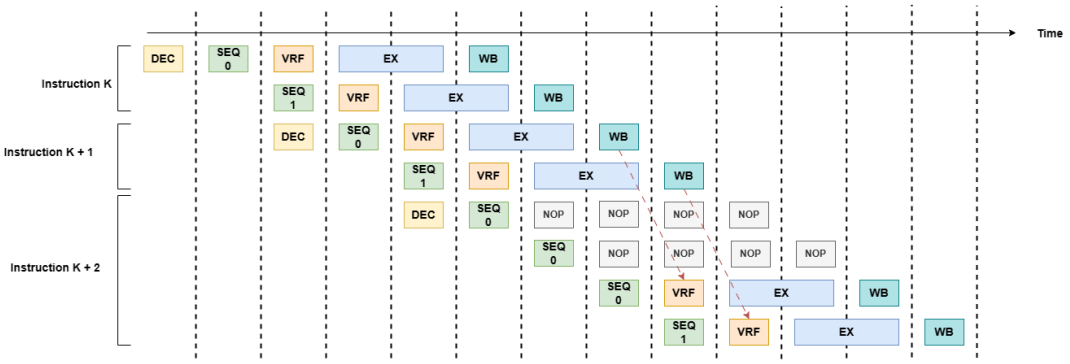

Figure 3: Pipeline execution for two elements per vector register per lane

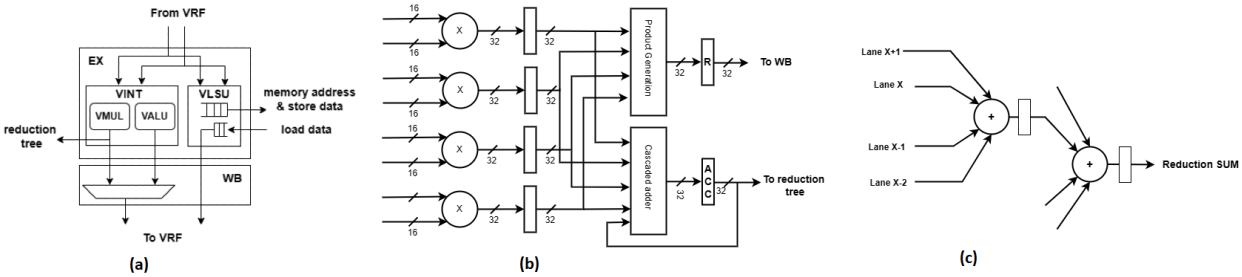

Figure 4: (a) Execution and Write back block diagram (b) VMUL block diagram (c) reduction tree

When the number of lanes increases, the addition of these accumulators over multiple lanes becomes a timing-critical path because of long wire delay and LUT chaining delay. To overcome this issue, we have used a quad-tree graph-based reduction tree presented in figure 4 (c). Accumulators from multiple lanes are the input nodes of this graph, and intermediate nodes calculate the summation of 4 inputs from the previous layer nodes and register the results. These node registers act as pipeline registers to break down the critical path of reduction operation over multiple lanes.

The aim of this reduction tree is to reduce the required number of clock cycles for reduction sum instruction while achieving higher operating frequency. For example, in 16-lane VPU, if we do the iterative execution, then it will take eight clock cycles for SEQLEN=8, while the reduction tree will take only two clock cycles. Again for iterative execution, there will be 16 addition per clock cycle, which will slow down the maximum operating frequency. On the other hand, the proposed quad-tree graph-based reduction tree will take only two clock cycles for 16 lanes. Since the reduction instruction will execute on the vector elements at once, there will be RAW data hazard, and the sequencer will insert two bubbles after the multiply instruction and then will execute the reduction sum instruction.

For instance, if MVL is 4,096 bit and SEW is 8-bit then there are 512 8-bit integer (INT8) elements in a vector register. And if NLANE is 16, then there are 32 elements per vector register per lane. It will take 8 clocks to multiply 2 vector registers and produce 16-bit results. During multiplication operation, it will also accumulate the multiplication results along the lane. Reduction sum instruction immediately after the multiplication instruction will stalled for 2 clock cycles due to RAW data hazard with the multiplication result's vector register. After stalling for 2 clock cycles the sequencer will issue the reduction instruction. Then the reduction sum operation will take 2 clock cycles to do the summation in the quad tree over the 16 lanes. Total 12 clock cycles will be required to perform dot/convolution operation on 512 INT8 elements to produce the result.

Vector load store unit (VLSU) has a memory address queue for load and store operation. Write-data queue for the store operation and the read-data queue for the load operation. Address and write-data queue have 16-depth, so if the SEQLEN=8 it can queue two vector load or store instruction. Each location in the queue holds the address of the given iteration of load or store instruction from the sequencer. Read data queue have only single depth because the load data from the TCM will be written back to the VRF immediately. RVV have define unit stride, constant and index strided address for vector load and store instruction. Our architecture implements unit stride stride as main feature and other strided load and store instructions are optional. However the VLSU do not support the segmented vector load and store instructions.

There are tightly coupled data memories (TCM) connected to each lane. TCM consists of dual port BRAM and have local and global port as presented in [8]. TCMs are organized as word (32-bit) interleaving memory among the lanes (lane 0 TCM have byte 0 to 3, lane 1 TCM have byte 4 to 7 and so on). Each lane have dedicated

read and write port to it's TCM (local port) and another port (global port) of TCM is connected to the ring interconnect. For instance, if each lane is accessing its own TCM, then there is a fixed 2-clock cycle latency. If it accesses to other lane's TCM, then it will go through the ring bus.

Since memory operation latency can vary with the memory location, the VINT can execute in parallel while the VLSU queue is not empty, given that there is no data dependency between them. Write back unit multiplexes data from VALU and VMUL and write it back to one of the write ports of VRF in that lane. The load data from the VLSU unit is written to another port of that VRF.

3 EVALUATION

We implemented our vector extension core in SystemVerilog and then integrated it with RVCoreP-32IM [8], which is a 5-stage pipeline 32-bit RISC-V soft processor core with RV32IM extension. We have configured our vector processing unit's number of lanes (NLANE) as 1, 4, 8, and 16 lanes for evaluation.

We evaluate the maximum frequency and hardware resource utilization for both Xilinx Artix-7 FPGA and Zynq UltrScale+ FPGA device. Artix-7 is relatively cost-optimized smaller FPGA than Zynq UltrScale+ FPGA. For edge computing sometimes smaller and cost-effective FPGA devices like Artix-7 is preferred. We use the Nexys A7 FPGA board which contains xc7a-100tcsg324-1 from Artix-7 FPGA device family, speed grade -1. Vivado 2021.1 is used for Synthesis, placement, and routing with a default strategy to evaluate the operating frequency and hardware resource utilization. For this evaluation, we have used 4KB of tightly coupled data RAM for each vector lane. The RISC-V core RVCoreP-32IM is connected to these lanes using a ring interconnect. RVCoreP-32IM has its own instruction and data tightly coupled memory of 32KB for program execution and a UART as a peripheral.

FPGA implementation result is shown in table 1. From the table 1, we can see that the look-up table (LUT) and register utilization do not increase linearly with the increment of NLANE, as the vector instruction decoder and sequencer are fixed overhead for all lane configurations. Since the vector register file uses the BRAM and and VMUL uses the DSP, BRAM and DSP utilizations go up straightly with the NLANE. In the FPGA implementation, two BRAMs are used as four 2KB RAMs for the 2R2W port for the vector register file. In all configurations, RVCoreP-32IM uses around 1,500 LUTs.

The maximum operating frequency (FMAX) drops only 5 MHz for the 16-lane configuration, while other configurations have 125 MHz. In our design, the critical path is partial product sum generation through vector multiplier units. For the same NLANE, increasing the MVL (that is, the number of elements per vector register per lane) does not increase the FPGA logic utilization. It does not impact the maximum operating frequency.

We have presented the comparison of resource usage with other soft vector processors in table 2. Since those publications mostly used different Xilinx Zynq UltrScale+, we have also done the synthesis for Zynq UltrScale+ FPGA device with the configuration of 8 Lane and MVL=1024. VPU1[11] has a dedicated unit for permutation, which could be a reason for higher resource usage. From the table 2 we can see that our proposed architecture has achieved more than four times higher operating frequency than VPU3 and more than three times higher operating frequency than other VPUs. And considering the NLANE, our proposed VPU uses significantly fewer FPGA resources than other VPUs.

VPU2[2] and VPU3[1] have the same architecture but the configurations are different. VPU3[1] has similar logic utilization to the proposed architecture though their operating frequency is much lower than the proposed architecture. Compared to our proposed 5-stage pipeline architecture, VPU3 has 2-stage pipeline, which could be the reason of this frequency difference. Vicuna[13] has 2 stage pipeline though the execution can take multiple clock cycles. But architecture takes much more logic resources compared to the proposed VPU which might negatively impact the Vicuna operating frequency.

For performance evaluation, we have used convolution because it is widely used in signal processing, image processing and convolutional neural network. We perform cycle-accurate RTL simulation for the convolution operation with kernel size 3x3x256 and input matrix size 16x16x256 and produce an output matrix of size 14x14. We have used the convolution filter and input data type of 8-bit(INT8), 16-bit(INT16), and 32-bit(INT32) integers, and the output is a 32-bit(INT32) integer. We use the vector load, widening multiplication, and reduction sum and move instruction to perform the convolution operation.

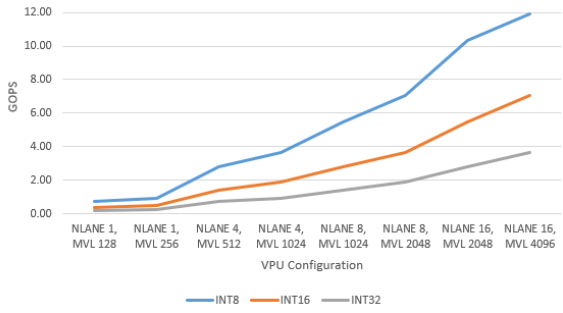

Figure 5: Giga operations per second (GOPS) for different VPU configuration

The cycle count required to perform the convolution operation is shown in table 3. As shown in table 3, for a given number of lanes, the performance improved with a higher MVL. This is because of less instruction overhead. Since our architecture supports 4 INT8, 2 INT16, and 1 INT32 operation per lane, the execution time for INT8, INT16, and INT32 also increases linearly. Table 3 also illustrates that adding more lanes significantly improves the performance, owing to higher parallel operations.

We have calculated the Giga operations per second (GOPS) from the simulation result (using FMAX for Zynq Ultrasacle+ FPGA) and presented it in figure 5. To calculate GOPS, we took the total number of operations by multiplying the number of vector instructions executed by the VPU with the number of elements processed by a single vector instruction. We then measure the execution time for the convolution operation by multiplying the clock period of FMAX for Zynq Ultrasacle+ FPGA device with the cycle count as given table 3. After that, we get the GOPS from the total number of operations divided by the execution time.

Table 1: FPGA (Artix-7) resource utilization for vector processing unit

NLANE	1		4		8		16	
MVL	128	256	512	1024	1024	2048	2048	4096
LUT	1,236	1,209	3,750	3,741	7,119	7,187	13,978	13,852
Register	737	743	2,059	2,068	3,871	3,853	7,375	7,382
BRAM	2	2	8	8	16	16	32	32
DSP	5	5	20	20	40	40	80	80
FMAX (MHz)	125	125	125	125	125	125	120	120

Table 2: Comparison of FPGA resource utilization (for Zynq UltrScale+ device)

VPU	MVL	LUT	Register	FMAX (MHz)
Vicuna[13]	2048, NLANE=32	80k	40k	80
VPU1[11]	512	136.5k	37.9k	75
VPU2[2]	256, NLANE=8	20.5k	9.5k	N/A
VPU3[1]	128, NLANE=2	7.3k	4.85k	62.5
Proposal	2048, NLANE=8	6.5k	3.8k	270

Table 3: Required number of clock cycles to perform 3x3x256 convolution operation

VPU Configuration	INT8	INT16	INT32
NLANE 1, MVL 128	682,276	1,359,652	2,714,404
NLANE 1, MVL 256	513,716	1,021,748	2,037,812
NLANE 4, MVL 512	174,244	343,588	682,276
NLANE 4, MVL 1024	132,692	259,700	513,716
NLANE 8, MVL 1024	89,572	174,244	343,588
NLANE 8, MVL 2048	69,188	132,692	259,700
NLANE 16, MVL 2048	47,236	89,572	174,244
NLANE 16, MVL 4096	40,964	69,188	132,692

Table 4: Comparison of GOPS with other VPUs

VPU	NLANE	GOPS
Vicuna[13]	32	10
VEGAS[5]	32	15
Proposal	16	11.9

For INT8 with NLANE=16 and MVL=4096, we achieved the maximum GOPS of 11.9. We have compared the GOPS values with a few other VPUs in table 4. It shows that our 16-lane VPU achieved remarkable GOPS compared to the other VPUs. The key factors of our VPU performance are substantially higher operating frequency due to pipeline execution and reduced inter-lane communication because of distributed VRF, and optimized reduction tree for lower latency.

4 CONCLUSION

In this paper, we presented a vector processing unit with the ability to configure the MVL, thus allowing more elements to be processed with lower instruction overhead. We support a parameterized NLANE, which benefits a longer vector to execute in a shorter time using parallel lanes. Performance increases with MVL for a given NLANE, whereas the FPGA resource utilization does not increases. And the resource utilization of FPGA also does not grow linearly with increasing NLANE. Our FPGA implementation has demonstrated 4x frequency improvement and the lowest FPGA resource utilization. Considering the resource utilization, our proposed VPU have achieved higher performance than other VPUs. Given those factors, our architecture provides a resource-efficient implementation for FPGA while maintaining higher frequency.

Even though the implemented subset of vector extension is suitable for many embedded applications, some other applications, like encryption, cryptography, etc., require vector permutation instruction which is not supported in this architecture. We also have not implemented the widening instructions for the simplification of VRF. In future work, we will implement the support for vector permutation instruction and study the widening instructions requirement. We will also investigate the embedded benchmark program by using RISC-V vector instruction.

REFERENCES

[1] Muhammad Ali and Diana Göhringer. 2022. Application Specific Instruction-Set Processors for Machine Learning Applications. In *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 1–4.

[2] Muhammad Ali, Matthias von Ameln, and Diana Goehringer. 2021. Vector processing unit: a RISC-V based SIMD co-processor for embedded processing. In *2021 24th Euromicro Conference on Digital System Design (DSD)*. IEEE, 30–34.

[3] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. 2019. Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 2 (2019), 530–543.

[4] Matheus Cavalcante, Domenic Wüthrich, Matteo Perotti, Samuel Riedel, and Luca Benini. 2022. Spatz: A Compact Vector Processing Unit for High-Performance and Energy-Efficient Shared-L1 Clusters. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[5] Christopher H Chou, Aaron Severance, Alex D Brant, Zhiduo Liu, Saurabh Sant, and Guy GF Lemieux. 2011. VEGAS: Soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. 15–24.

[6] Intel Corporation. 2022. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Retrieved March 28, 2023 from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[7] RISC-V International. 2022. *Working draft of the proposed RISC-V V vector extension*. Retrieved March 28, 2023 from https://github.com/riscv/riscv-v-spec

[8] Md Ashraful Islam and Kenji Kise. 2022. An Efficient Resource Shared RISC-V Multicore Architecture. *IEICE TRANSACTIONS on Information and Systems* 105, 9 (2022), 1506–1515.

[9] Charles Eric LaForest and J Gregory Steffan. 2010. Efficient multi-ported memories for FPGAs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. 41–50.

[10] ARM Limited. 2009. *Introducing NEON Development Article*. Retrieved March 28, 2023 from https://developer.arm.com/documentation/dht0002/a/?lang=en

[11] Vincenzo Maisto and Alessandro Cilardo. 2022. A pluggable vector unit for RISC-V vector extension. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1143–1148.

[12] Stefania Perri, Pasquale Corsonello, Maria Antonia Iachino, Marco Lanuzza, and Giuseppe Cocorullo. 2004. Variable precision arithmetic circuits for FPGA-based multimedia processors. *IEEE Transactions on very large scale integration (VLSI) systems* 12, 9 (2004), 995–999.

[13] Michael Platzer and Peter Puschner. 2021. Vicuna: a timing-predictable RISC-V vector coprocessor for scalable parallel computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[14] Aaron Severance and Guy Lemieux. 2012. VENICE: A compact vector processor for FPGA applications. In *2012 International Conference on Field-Programmable Technology*. IEEE, 261–268.

[15] Aaron Severance and Guy GF Lemieux. 2013. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 1–10.

[16] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. 2017. The ARM scalable vector extension. *IEEE micro* 37, 2 (2017), 26–39.

[17] Peter Yiannacouras, J Gregory Steffan, and Jonathan Rose. 2008. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. 61–70.

[18] Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. 2009. Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 2, 2 (2009), 1–34.