VVIP: Versatile Vertical Indexing Processor for Edge Computing

Hyungjoon Bae
KAIST
Daejeon, Republic of Korea
jo_on@kaist.ac.kr

Da Won Kim*
Columbia University
New York, NY, USA
dk3311@columbia.edu

Wanyeong Jung
KAIST
Daejeon, Republic of Korea
wanyeong@kaist.ac.kr

ABSTRACT

This paper presents a versatile vertical indexing processor (VVIP) based on a single-instruction multiple-data architecture for edge computing. In VVIP, the vertical source and destination indexing instructions are customized for area-efficient computations. The proposed indexing method reorders data within a processing module by using more registers and data-steering logic in the calculations. In particular, VVIP supports multibit-serial multiplication and sparse data operations by leveraging register files as lookup tables or accumulators. The VVIP, verified on a vector processor, has an area overhead of less than 2.8%. It exhibits an average computation rate that is 10.1 times faster than the 1-bit-serial multiplication in linear algebra benchmarks, and 1.2 times average performance improvement in unstructured sparse point-wise convolution tasks when compared to conventional control sequences.

KEYWORDS

Indexing instruction, single-instruction multiple-data, RISC-V, edge computing

ACM Reference Format:
Hyungjoon Bae, Da Won Kim, and Wanyeong Jung. 2024. VVIP: Versatile Vertical Indexing Processor for Edge Computing. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3655676

1 INTRODUCTION

Intelligent edge computing is widely implemented in various applications such as self-driving cars, robotics, mobile devices, surveillance systems, and speech recognition. They can be integrated with deep learning (DL) to achieve better processing accuracy, but the considerable computational and energy overhead of DL exceeds the capacity of conventional general-purpose processors. Therefore, several edge computing devices have employed domain-specific accelerators that provide high computing power at low energy costs in recent years [3, 7, 10].

However, domain-specific accelerators still face various challenges. Firstly, highly energy-efficient computation is achieved at the cost of operational diversity. Minor algorithmic or target application changes may require new, dedicated hardware involving additional design and chip manufacturing costs. Since edge computing devices face hardware constraints such as area, energy, and memory bandwidth, integrating separate accelerators for each application is challenging [3].

Secondly, most accelerators are optimized for computational kernels, such as general matrix multiply (GEMM), so supporting end-to-end operations within edge computing devices results in underutilization during pre/post-processing. [6] reports that only 40% of the total DL execution time is used for computational acceleration. These subtasks may require a separate processing unit, but it would lead to chip area increase and render the accelerator less favorable to integrate. Furthermore, the internal data movement between the two types of hardware increases the overall computation time and reduces the energy efficiency.

Lastly, the existing accelerators are not optimized for changes in the data characteristics. For example, data and control paths that focus on dense data are underutilized with sparse data. Conversely, internal hardware components that are suitable for sparse data become redundant and underutilized when performing dense data calculations. In particular, the index decoders and registers that are added to handle sparse data impose a significant area overhead and complicate the design of edge computing devices [2, 8].

Previous studies have addressed these issues to some extent by integrating general-purpose processors with multiple modules [4, 6]. Versatile processors with augmented data processing capacity can successfully perform computations while balancing the application scope and computational efficiency. However, they are constructed only as a sum of functional modules, and do not reuse the existing circuits. These approaches exhibit low circuit utilization, so there is room for improvement.

This study presents a versatile vertical indexing processor (VVIP) designed for edge computing devices that maximizes the utilization of internal circuits. The single-instruction multiple-data (SIMD)-based VVIP utilizes custom source and destination indexing to activate registers and data-steering logic in the calculations. The proposed indexing method seamlessly integrates arithmetic instructions and transforms address operands into vertical index operands. VVIP reduces the computational overhead incurred by conventional control sequence indexing by directly accessing the register files (RFs) as lookup tables (LUTs) or accumulators through a dynamic index. VVIP accelerates essential computational tasks, such as bit-serial multiplication and sparse data handling, by incorporating RF indexing into data processing paths. Since it largely reuses the circuits in the original RF, the proposed vertical indexing incurs a minimal, constant overhead that remains unaffected by the degree of parallelism, unlike legacy horizontal indexing instructions.

VVIP is verified on a RISC-V vector processor. It is evaluated using an in-house cycle-accurate simulator, and the hardware area is measured using a 28 nm CMOS process library. It has an area overhead of less than 2.8% and exhibits a 10.1 times average execution speedup on linear algebra benchmarks when compared with

*Performed this work while affiliated with KAIST

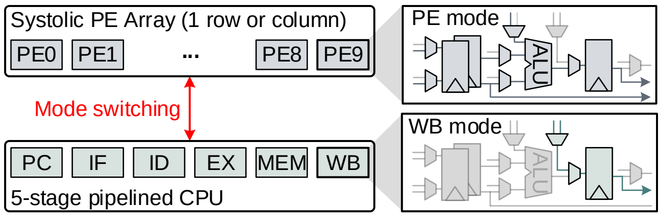


Figure 1: Overview of reconfigurable processor

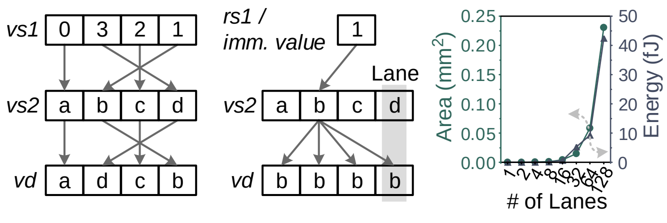
the 1-bit-serial multiplication. It also achieves a 1.2 times average performance enhancement on unstructured (10-90%) sparse point-wise convolution when compared with conventional switch-case statement sequences.

2 BACKGROUND AND RELATED WORK

Processing module. This study defines the "processing module" as a fundamental computational component that receives an instruction, processes the data accordingly, and outputs the result. It comprises an RF that supplies or stores data, an arithmetic logic unit (ALU) for data operations, and data-steering logic that selects the operands and data. The RF contains dozens of built-in registers, and the ALU contains various arithmetic and logic components based on the range of functions supported by the module. The more these three circuits are reused, the higher the area-efficient computation.
Reconfigurable processor. In [6], a versatile neural processor is developed by integrating a reconfigurable systolic processing element (PE) array with a 5-stage pipelined central processing unit (CPU), as shown in Figure 1. Extensive internal logic is included in the processing module for mode switching. Additionally, a specialized register and custom instructions are designed to enhance flexibility while ensuring RISC-V compliance. Although it seems module reusability increases to 80% and data movement is significantly reduced through mode reconfiguration, certain internal components are still redundant. For example, in the write-back (WB) stage, many data paths, including the ALU, are left unused. It is noteworthy that the ALU includes a multiplier, which imposes considerable overhead in the area and leakage. The RF, another massive circuit unit, is also unused for a long time in the DL mode, and lowers area efficiency. Determining the reuse of the primary circuit is essential, rather than simply designing the processing module as a sum of circuits for various functions.
GEMM tile acceleration on CPUs. [4] proposes a microarchitecture that integrates a matrix engine next to a vector processor to accelerate sparse data processing while maintaining generality. To support structured/unstructured sparsity, the tile and metadata registers are added outside the engine, and ISA is specialized for this purpose. However, the engine and registers occupy space, and registers are underutilized when processing dense data. Therefore, rather than adding a separate engine to a limited area, it is imperative to enhance the efficiency of the internal components of the existing module.

3 MOTIVATION

This paper focuses on optimizing the processing module for area-efficient computations at the circuit level. Since ALU contains essential components with high activity, there is limited scope for further optimization. However, the RF is less active than the ALU owing to its inherently non-computational nature and occupies most of the area within the module, resulting in a lower computational efficiency per unit area. This limitation is attributed to the data-steering logic, such as multiplexers and address decoders, which only processes non-computational data in a constant format. In other words, this prevents the RF from engaging in information processing tasks other than feeding and storing the data.



(a) vrgather.vv (b) vrgather.vx/vi (c) overhead

Figure 2: RISC-V vector register gather instructions, and their interconnection network hardware overhead

One method of solving this limitation involves using data-steering logic to receive the address of a variable containing the calculation information as an index, further activating the RF to participate in the operation. The existing vector ISAs, such as RISC-V, enable this through horizontal index-based gather instructions [1]. These instructions facilitate data exchange between the processing modules of the vector processor, known as lanes, as shown in Figure 2b. Figures 2a and 2b depict examples of **vrgather** of the RISC-V vector extension, where the index source is a register or an immediate value, and the destination is a vector register. In the case of **vrgather.vv**, each lane uses the value from its index register and copies it from its corresponding position in the source register to its destination register. This is expressed as follows:

$$vd[i] = vs2[vs1[i]], \tag{1}$$

where *vd*, *vs2*, *vs1* represent the destination, second source, and first source vector operand, respectively; and *i* denotes the index of the vector element. Both **vrgather.vx** and **vrgather.vi** use only one global index stored in a scalar register or an immediate value. Therefore, all the lanes perform a broadcasting operation that fetches the same data from the source register indicated by the index value. They are represented as follows:

$$vd[i] = vs2[rs1\,/\,imm.\ value], \tag{2}$$

where *rs1* denotes the first scalar source operand.

The gather instructions engage the RF in the computation. However, data changes are made between the processing modules, which require additional n:n connection hardware. In particular, the hardware complexity scales quadratically with the total number of lanes, as shown in Figure 2c. This limits the number of lanes for high parallelism, especially in the area- and power-constrained hardware.

The disadvantage mentioned above can be compensated for by narrowing the scope of indexing within each processing module, rather than between them. The existing data-steering logic supports 1:n addressing of RF entries; therefore, it can be repurposed for variable indexing. Since this indexing overhead is affected only by the number of registers and not the number of processing modules, it poses an overhead that is unrelated to parallelism. Therefore, this study explores vertical indexing instead of horizontal indexing.

Algorithm 1 Switch-case statement-based indexing

```
1: Inputs: $i\ (multiplicand), w_0, w_1, w_2\ (multiplier)$
2: Output: $o = iw_0 + iw_1 + iw_2$
3: Initialize: $REG[0] = w_0$, $REG[1] = w_1$, $REG[2] = w_2$, $o = 0$
4: for $index$ in $\{0, 1, 2\}$ do
5:     switch $index$ do
6:         case 0: $w = REG[0]$
7:         case 1: $w = REG[1]$
8:         case 2: $w = REG[2]$
9:     end switch
10:    $o = o + i \times w$
11: end for
```

Algorithm 2 Vertical index-based indexing

```
1: Inputs: $i\ (multiplicand), w_0, w_1, w_2\ (multiplier)$
2: Output: $o = iw_0 + iw_1 + iw_2$
3: Initialize: $REG[0] = w_0, REG[1] = w_1, REG[2] = w_2, o = 0$
4: for $index$ in $\{0, 1, 2\}$ do
5:     $w = REG[index]$
6:     $o = o + i \times w$
7: end for
```

4 INFORMATION PROCESSING WITH VERTICAL INDEXING

4.1 Programming Benefit

Traditional systems without vertical indexing use control sequences such as if-else and switch-case statements to support dynamic register-entry access. Conversely, vertical indexing assigns the index variable directly to the RF address decoder; thus, no separate control commands are required when using RFs as LUTs or accumulators. Algorithms 1 and 2 demonstrate the programming differences in register access for weighted sum examples when the index is used as a constant and when it is used as a variable. When comparing lines 5 to 9 of Algorithm 1 and line 5 of Algorithm 2, the vertical-index-based register access ensures that the program is concise and intuitive. This indexing is not limited to the source, and can also be used at the destination. Vertical indexing has two index paths: one for all the processing modules to receive the same index, and the other for each module to receive a different index, similar to the gather instructions, as shown in (1) and (2). From a programming perspective, both cases can still be accessed with a single instruction while maintaining the SIMD characteristics.

4.2 Wide Range of Applications

LUT-based multiplication. As an application of 1-bit-serial multiplication, if the multiplicand, a, is grouped into k bits, the bit-serial multiplication can be accelerated by k times by accumulating the partial products as follows:

$$a \times b = \sum_{n=0}^{\lfloor \frac{N-1}{k} \rfloor} \left[(a_{kn+k-1} \cdots a_{kn+1} a_{kn})_2 b \right] \times 2^{kn}, \tag{3}$$

where $(a_{kn+k-1} \cdots a_{kn+1} a_{kn})_2$ represents the concatenated binary bit from the $(kn + k - 1)$-th to kn-th bit of a, and N denotes the bit width of a [10]. The partial product expressed in parentheses in (3) is one of the 2^k different values $\{0, b, ..., (2^k - 1)b\}$; therefore, it can be stored in a 2^k-row LUT and $(a_{kn+k-1} \cdots a_{kn+1} a_{kn})_2$ can be used as a vertical index. A small number (approximately three or four) of k is typically sufficient for significant speedup without exponentially increasing the size of the LUT. The 2^{kn} term is computed by using a shifter, and the summation is calculated through an adder. These operations are repeated $\lfloor (N-1)/k \rfloor + 1$ times to obtain the final product, which is independent of the multiplicand value. This method actively utilizes essentially all major circuits (RF, ALU, and data-steering logic) within the processing module without a multiplier, enabling area-efficient computations.

Distributed arithmetic. The weighted sum of the input vectors can be calculated efficiently using an elaborate parallel version of bit-serial computation called distributed arithmetic (DA) [11]. The weighted sum of M input vectors, b, each having a weight of a_m, can be expressed as follows:

$$\sum_{m=0}^{M-1} a_m b_m = \sum_{n=0}^{N-1} \left[\sum_{m=0}^{M-1} a_m (b_{mn})_2 \right] 2^n, \tag{4}$$

where $(b_{mn})_2$ represents the n-th binary bit of the m-th input b; N denotes the bit width of b_m. The sigma term inside the parentheses in (4) has only 2^M cases, which can be stored in the 2^M-row LUT. If the number of input vectors, M, is grouped into a small value, such as three or four, DA can be computed by using RF. The RF entry can be vertically indexed by the concatenated $(b_{mn})_2$ value [7]. The partial sum obtained by using the vertical index is accumulated N times through the addition and shift operations. DA can also exhibit area-efficient computation involving all major circuits with a mere addition of vertical indexing.

Bit-serial factorization. If an output has stationary characteristics computed through many multiply-accumulate (MAC) operations, it can be calculated by using a group of accumulators [5]. This approach is called factored accumulation, and when performing MAC operations on one multiplicand and M multipliers, it is expressed as follows:

$$\sum_{m=0}^{M-1} a \times b_m = \sum_{n=0}^{N-1} (a_n)_2 \left[\sum_{m=0}^{M-1} b_m \times 2^n \right], \tag{5}$$

where $(a_n)_2$ represents the n-th binary bit of the multiplication a. The sigma inside the parentheses in (5) can be accumulated in the accumulator with $(a_n)_2$ as the vertical index. However, since the number of iterations can be as high as $N \times M$, it needs to be reduced by grouping $(a_n)_2$ into k, similar to the LUT-based multiplication. This is expressed as follows:

$$\sum_{m=0}^{M-1} a \times b_m = \sum_{n=0}^{\lfloor \frac{N-1}{k} \rfloor} (a_{kn+k-1} \ldots a_{kn+1} a_{kn})_2 \left[\sum_{m=0}^{M-1} b_m \times 2^{kn} \right], \tag{6}$$

where k is approximately three or four, which is an appropriate value for configuring accumulators. The rightmost parentheses value in (6) is first shifted and accumulated in the accumulators with $(a_{kn+k-1} \cdots a_{kn+1} a_{kn})_2$ as the vertical index, and the final weight $(a_{kn+k-1} \cdots a_{kn+1} a_{kn})_2$ is then multiplied. Similar to the other methods, it maintains a high circuit activity of the RF, adder, shifter, and data-steering logic when equipped with vertical indexing capability.

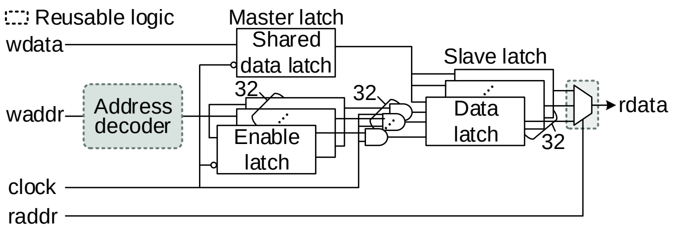
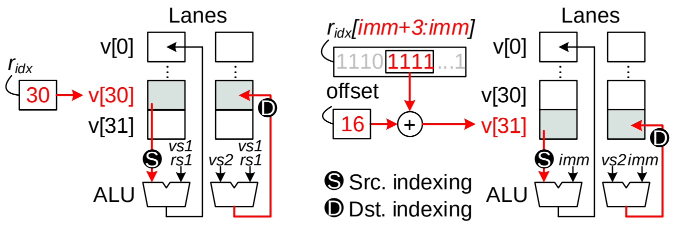
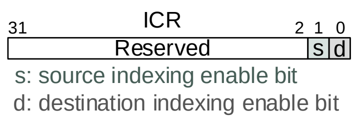


Figure 3: Architecture of RF and data-steering logic



(a) $\mathbf{vr_{idx}}$.vv/vx (b) $\mathbf{vr_{idx}}$.vi

Figure 4: Overview of $\mathbf{vr_{idx}}$ instructions


ICR
31
2 1 0
Reserved
s
d
s: source indexing enable bit
d: destination indexing enable bit

(a) ICR bit field

ICR[1:0]	Type	r_{idx}
00_2	no	-
01_2	dst.	*r[vd]*
10_2	src.	*r[vs2]*
11_2	bc.	*r[vs2]*[1]

[1] *vs2* is equal to *vd*.

(b) Configurations

Figure 5: Indexing configuration register and its configurations

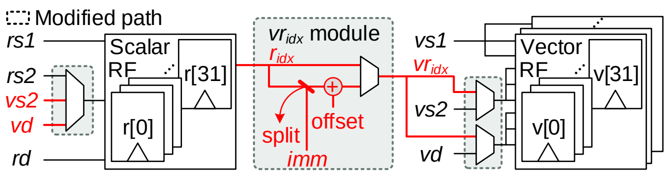


Figure 6: Datapath modifications of scalar and vector RFs for $\mathbf{vr_{idx}}$ instructions

Sparse data handling. A sparse matrix can be encoded by using non-zero data and their indices. These indices can be used as the vertical addresses of the LUTs or accumulators to perform sparse operations such as sparse matrix-vector multiplication [2, 8]. Encoded data processing enables high-RF, ALU, and data-steering logic activities within the processing module.

5 VERTICAL INDEXING ON REAL PROCESSOR

5.1 Baseline Vector Processor

Architecture. The proposed vertical indexing method can be combined with various SIMD architectures. In this study, VVIP is implemented on a RISC-V-based vector processor comprising one ALU and 32 registers of 32-bit integer per vector lane.

Area-efficient and high activity RF circuit. The data master latch is shared across all registers to increase RF area efficiency, as shown in Figure 3. This design achieves 0.75 times less area than a flip-flop-based RF with separate data master latches. When a write access occurs, write data and a write enable signal activated by the address decoder are captured in the master latch when the clock value is low, and latched data is stored in the slave latch when the clock value is high. The latched write enable and clock signals are combined using AND gates to generate a pulsed clock, enabling only one data slave latch. When a read access occurs, the read data is selected by the multiplexer using the read address. Vertical indexing uses the same read/write mechanism without modifying these RF internal circuits. In particular, it increases RF activity by reusing the address decoder and data read multiplexer for indexing.

5.2 Customized Vertical Indexing ISA

$\mathbf{vr_{idx}}$ instruction. If all vector lanes receive a global index, such as **vrgather.vx** and **vrgather.vi** shown in (2), but when indexing vertically within a vector lane, it can be expressed as $\mathbf{vr_{idx}}$. The shared vertical index value is from a scalar RF entry. This instruction involves scalar and vector RFs and their data-steering logic.

$\mathbf{vv_{idx}}$ instruction. If all vector lanes access different entries, this can be expressed as $\mathbf{vv_{idx}}$. This is equivalent to setting the indexing of **vrgather.vv** shown in (1) to be vertical. This instruction can be realized by reusing vector RFs and their data-steering logic, but recurrent RF access of each lane may burden their read ports.

$\mathbf{rv_{idx}}$ instruction. If all vector lanes use a shared LUT through the individual vertical index, this is expressed as $\mathbf{rv_{idx}}$. The index values are from all vector lanes, each pointing to a scalar RF entry. This instruction can be implemented by reusing the data-steering logic in the vector lanes. However, it also requires additional data paths that scalar RF values can be broadcast to all vector lanes.

5.3 Implementation Details of $\mathbf{vr_{idx}}$

In this study, only $\mathbf{vr_{idx}}$ indexing is implemented in our evaluation setup. Figure 4 depicts the $\mathbf{vr_{idx}}$ instructions according to the type of source operands. All arithmetic and logic instructions can be combined with three types of indexing: source, destination, and broadcast. Broadcast indexing is used when both the source and destination are vertically indexed by the same value. The indexing type is configured in the lower two bits of the indexing configuration register (ICR), as shown in Figure 5a. ICR is mapped to the control and status register (CSR) region of the RISC-V and is read and written with legacy CSR instructions. Figure 5b depicts the r_{idx} value corresponding to the ICR bit.

In the case of $\mathbf{vr_{idx}}$**.vv** and $\mathbf{vr_{idx}}$**.vx** instructions, r_{idx} points directly to the RF. Thus, the operations are expressed as follows:

$$\text{src. indexing: } vd[i] = vr_{idx}[i] \text{ op } (vs1[i]\,/\,rs1), \tag{7}$$

$$\text{dst. indexing: } vr_{idx}[i] = vs2[i] \text{ op } (vs1[i]\,/\,rs1), \tag{8}$$

$$\text{bc. indexing: } vr_{idx}[i] = vr_{idx}[i] \text{ op } (vs1[i]\,/\,rs1), \tag{9}$$

where op is the ALU opcode. In the case of $\mathbf{vr_{idx}}$**.vi**, some bits of the r_{idx} value are extracted and used as an index to support (3) and (6). VVIP uses a 4-bit width $r_{idx}[imm+3:imm]$, which is a divisor of 32 bits, and fixes the offset of the index to 16. Therefore, *v16–v31* in each lane are used as a LUT or accumulators.

Figure 6 depicts the modified data paths within the scalar and vector processor for $\mathbf{vr_{idx}}$ instructions. Since the vector arithmetic

Algorithm 3 LUT-based multiplication with vertical indexing

```
1:  Inputs: $i_0 - i_{(L-1)}, w$
2:  Output: $o = \sum_{l=0}^{L-1} i_l \times w$
3:  Initialize: $v16$ - $v31 = 0$ - $15w$ (partial products), $v2 = 0$
4:  for $l = 0$ to $L-1$ do
5:      $r31 = i_l$
6:      vmv.v.i v1, 0             // v1 = 0
7:      csrrwi x0, ICR, 0b10      // src. indexing
8:      vshiadd.vi v1, v31, 0   // v1 = vr31[3:0] « 0 + v1
9:      vshiadd.vi v1, v31, 4   // v1 = vr31[7:4] « 4 + v1
10:     vshiadd.vi v1, v31, 8   // v1 = vr31[11:8] « 8 + v1
11:     vshiadd.vi v1, v31, 12 // v1 = vr31[15:12] « 12 + v1
12:     vshiadd.vi v1, v31, 16 // v1 = vr31[19:16] « 16 + v1
13:     vshiadd.vi v1, v31, 20 // v1 = vr31[23:20] « 20 + v1
14:     vshiadd.vi v1, v31, 24 // v1 = vr31[27:24] « 24 + v1
15:     vshiadd.vi v1, v31, 28 // v1 = vr31[31:28] « 28 + v1
16:     csrrwi x0, ICR, 0b00      // no indexing
17:     vadd.vv v2, v1, v2        // v2 = v1 + v2
18: end for
```

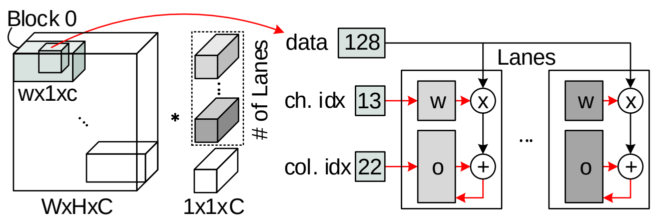


Figure 7: Block-level encoding and register mapping of point-wise convolution

Algorithm 4 Sparse data handling with vertical indexing

```
1:  Inputs: $data_0 - data_{(L-1)}$, $ch.idx_0 - ch.idx_{(L-1)}$, $col.idx_0 -$
    $col.idx_{(L-1)}$ (block size: $16 \times 1 \times 3$)
2:  Output: $o_{col.idx_l}$ += $w_{ch.idx_l} \times data_l$ ($l = 0, 1, \ldots, L-1$)
3:  Initialize: $v13$ - $v15 = w_0 - w_2$ (weight LUT), $v16$ - $v31 = o_0 -$
    $o_{15}$ (output accumulators)
4:  for $l = 0$ to $L-1$ do
5:      $r30 = ch.idx_l$
6:      $r31 = col.idx_l$
7:      csrrwi x0, ICR, 0b10       // src. indexing
8:      vmul.vx v1, v30, $data_l$ // $v1 = vr30 \times data_l$
9:      csrrwi x0, ICR, 0b11       // bc. indexing
10:     vadd.vv v31, v31, v1       // vr31 = vr31 + v1
11: end for
```

instructions of RISC-V do not use the *rs2* port, they reuse this path to read the global index. For the $\mathbf{vr_{idx}}$**.vv** and $\mathbf{vr_{idx}}$**.vx** instructions, r_{idx} is used as the vertical index, whereas for the $\mathbf{vr_{idx}}$**.vi** instruction, four bits are extracted from r_{idx} located by *imm* and the value added with the offset is used as the vertical index. The calculated vr_{idx} dynamically indexes the vector RF to access the desired location. This modification presents low overhead since every vector processing module shares only one vr_{idx} module, and it requires only the addition of tiny index selectors and an offset adder.

Table 1: Polybench/C Linear Algebra Benchmarks

Name	Equation[1]	Output size
2mm	$D = \alpha ABC + \beta D$	800×1200
3mm	$E = (AB)(CD)$	800×1100
atax	$y = A^T(Ax)$	2100×1
bicg	$q = Ap,\ s = A^T r$	2100×1, 1900×1
doitgen	$A(r,q,p) = \sum_{s=0}^{S-1} A(r,q,s)x(p,s)$	$150 \times 140 \times 160$
mvt	$q = q + Ap,\ s = s + A^T r$	2000×1, 2000×1
gemm	$C = \alpha AB + \beta C$	1000×1100
gemver	$\hat{A} = A + u1v1 + u2v2$, $w = \alpha \hat{A} x,\ x = \beta \hat{A}^T y + z$	2000×2000, 2000×1, 2000×1
gesummv	$y = \alpha Ax + \beta Bx$	1300×1
symm	$C = \alpha AB + \beta C$, where $A = A^T$	1000×1200
trmm	$B = AB$, where $a_{ij} = 0\ (i < j)$	1000×1200

[1] Uppercase represents a matrix; lowercase represents a vector or an element.

5.4 Data Mapping and Programming of $\mathbf{vr_{idx}}$

Multibit-serial multiplication. VVIP without built-in multipliers can accelerate the bit-serial multiplication using RF by performing (3) and (6) with vertical indexing. Since both equations require the shift and addition operations, the custom left-shift-add instruction (**vshiadd.vi**) is assigned to the ALU opcode, 111000_2, and $\mathbf{vr_{idx}}$**.vi** is combined. Algorithm 3 presents a code for (3) when $k = 4$. The proposed $\mathbf{vr_{idx}}$**.vi** supports partitioned 4-bit vertical indexing, so eight shift-add instructions are issued for 32-bit multiplication, as illustrated in lines 8 to 15. The code promotes high multiplication efficiency per unit area in small processing modules without built-in multipliers.

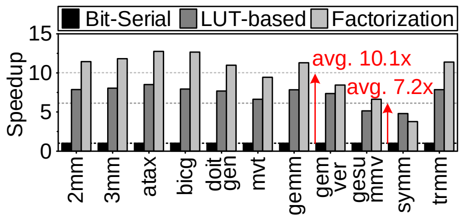
Sparse data handling. Sparse data encoded using indices can be calculated by using vr_{idx} value combined with existing arithmetic logic instructions. Figure 7 depicts an example of block-level encoding and register mapping for supporting sparse point-wise convolution on a vector processor. The input is divided into small blocks, each encoded by data, channel, and column indices. Each vector lane is responsible for one output channel. Therefore, the weights are stored as a channel-index-based LUT, and the output values are accumulated in column-index-based accumulators. Algorithm 4 presents a code when the block size is 16×1×3. The input and weight are multiplied via source indexing and accumulated in the accumulators via broadcast indexing. Therefore, it operates without any additional modifications other than the basic vr_{idx} module. Multidimensional convolutions can be computed via vertical indexing, similar to point-wise convolution.

6 EXPERIMENTAL RESULTS

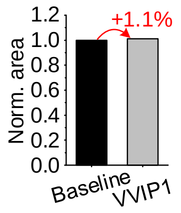
6.1 Experimental Setup

Parallelism. The number of lanes in the baseline and VVIP vector processor is set to 64.

Workloads. VVIP is analyzed with two RISC-V-based vector designs: VVIP1 on RV32IMV ISA and VVIP2 on RV32IMVM ISA. The



(a) Performance



(b) Normalized area

Figure 8: Performance and area comparison of baseline and VVIP1

performance of VVIP1 is evaluated by using 11 benchmarks selected from the linear algebra Polybench/C suite [9], which are listed in Table 1. These benchmarks are modified into integer versions and programmed in the RISC-V vector extension assembly language [1]. All the codes are written using masking; therefore, they are computed even on data that do not fit the number of lanes. The operational accuracy is verified against the baseline C benchmark results. VVIP2 is evaluated by performing a point-wise convolution of a 224x224x3 image with 64 1x1x3 kernels and by varying the sparsity of the image from 10% to 90% in steps of 10%.

Environments. Customized vertical indexing instructions are implemented by modifying the RISC-V compiler. All codes are evaluated by using a cycle-accurate in-house C-simulator. The hardware is designed through SystemVerilog and synthesized using Synopsys Design Compiler on a CMOS 28nm process library to measure the area.

6.2 Performance of VVIP1

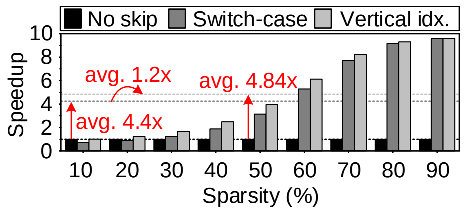
Figure 8 presents the benchmark evaluation results and the synthesized area on the 1-bit-serial vector processor and VVIP1. In the baseline processor, 32 left-shift-add operations are required to perform 32-bit multiplication, thereby reducing the operation speed in all the applications. However, VVIP1, which performs the (3) and (6) operations using 4-bit split indexing, achieves average performance improvements of 7.2 and 10.1 times, respectively. Since most applications exhibit output stationary characteristics, factorization presents the highest average performance. The vr_{idx} module and related path modifications exhibit an area overhead of 1.1%.

6.3 Performance of VVIP2

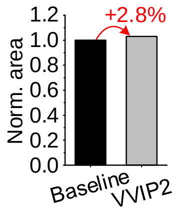
Figure 9 presents a comparison of the sparse point-wise convolution performance and synthesized area with and without block-level encoding in a vector processor with built-in multipliers and VVIP2. When zero-skipping with encoding is enabled, switch-case statements and vertical indexing are faster than dense operations, achieving a performance improvement of 4.4 times and 4.84 times on average, respectively. The performance enhancement of vertical indexing compared to switch-case statements is 1.2 times on average. Since more branches occur when the sparsity is lower than higher, the performance gap between them is more significant when the sparsity is low. The VVIP2 exhibits the highest area-efficient computations, with an area overhead of only 2.8%.

7 CONCLUSION

This study proposes a general-purpose SIMD processor with vertical indexing that achieves area-efficient computations, called VVIP.



(a) Performance



(b) Normalized area

Figure 9: Performance and area comparison of baseline and VVIP2

The proposed indexing combines with arithmetic instructions, utilizing all major circuits, such as the RF and ALU, and data-steering logic within the processing module. The vertical indexing design overhead is minimal, as the existing circuits within the processing module are reused. The experimental results demonstrate that on average, VVIP is 10.1 times faster than 1-bit-serial multiplication and 1.2 times faster than switch-case-based sparse point-wise convolution.

ACKNOWLEDGMENTS

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (Ministry of Science and ICT, MSIT) (No.2020-0-01297, Development of Ultra-Low Power Deep Learning Processor Technology using Advanced Data Reuse for Edge Applications). Additionally, the EDA tool was supported by the IC Design Education Center (IDEC), Republic of Korea.

REFERENCES

[1] Krste Asanovic. 2019. RISC-V Vector Extension.
[2] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.
[3] Jun-Woo Jang, Sehwan Lee, Dongyoung Kim, Hyunsun Park, Ali Shafiee Ardestani, Yeongjae Choi, Channoh Kim, Yoojin Kim, Hyeongseok Yu, Hamzah Abdel-Aziz, et al. 2021. Sparsity-aware and re-configurable NPU architecture for Samsung flagship mobile SoC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 15–28.
[4] Geonhwa Jeong, Sana Damani, Abhimanyu Rajeshkumar Bambhaniya, Eric Qin, Christopher J Hughes, Sreenivas Subramoney, Hyesoon Kim, and Tushar Krishna. 2023. VEGETA: Vertically-Integrated Extensions for Sparse/Dense GEMM Tile Acceleration on CPUs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 259–272.
[5] Zexi Ji, Wanyeong Jung, Jongchan Woo, Khushal Sethi, Shih-Lien Lu, and Anantha P Chandrakasan. 2020. CompAcc: Efficient Hardware Realization for Processing Compressed Neural Networks Using Accumulator Arrays. In *2020 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 1–4.
[6] Yuhao Ju and Jie Gu. 2022. A Systolic Neural CPU Processor Combining Deep Learning and General-Purpose Computing With Enhanced Data Locality and End-to-End Performance. *IEEE Journal of Solid-State Circuits* (2022).
[7] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. 2018. UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits* 54, 1 (2018), 173–185.
[8] Jinsu Lee, Juhyoung Lee, Donghyeon Han, Jinmook Lee, Gwangtae Park, and Hoi-Jun Yoo. 2019. 7.7 LNPU: A 25.3 TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 142–144.
[9] Louis-Noël Pouchet and Tomofumi Yuki. 2015. PolyBench/C 4.2.
[10] Dongjoo Shin, Jinmook Lee, Jinsu Lee, and Hoi-Jun Yoo. 2017. 14.2 DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 240–241.
[11] Stanley A White. 1989. Applications of distributed arithmetic to digital signal processing: A tutorial review. *IEEE Assp Magazine* 6, 3 (1989), 4–19.