# Hardware/Software Co-Design of RISC-V Extensions for Accelerating Sparse DNNs on FPGAs

Muhammad Sabih, Abrarul Karim, Jakob Wittmann, Frank Hannig, and Jürgen Teich Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany {muhammad.sabih, abrarul.karim, jakob.wittmann, frank.hannig, juergen.teich}@fau.de

Abstract—The customizability of RISC-V makes it an attractive choice for accelerating deep neural networks (DNNs). It can be achieved through instruction set extensions and corresponding custom functional units. Yet, efficiently exploiting these opportunities requires a hardware/software co-design approach in which the DNN model, software, and hardware are designed together. In this paper, we propose novel RISC-V extensions for accelerating DNN models containing semi-structured and unstructured sparsity. While the idea of accelerating structured and unstructured pruning is not new, our novel design offers various advantages over other designs. To exploit semi-structured sparsity, we take advantage of the finegrained (bit-level) configurability of FPGAs and suggest reserving a few bits in a block of DNN weights to encode the information about sparsity in the succeeding blocks. The proposed custom functional unit utilizes this information to skip computations. To exploit unstructured sparsity, we propose a variable cycle sequential multiply-and-accumulate unit that performs only as many multiplications as the non-zero weights. Our implementation of unstructured and semi-structured pruning accelerators can provide speedups of up to a factor of 3 and 4, respectively. We then propose a combined design that can accelerate both types of sparsities, providing speedups of up to a factor of 5. Our designs consume a small amount of additional FPGA resources such that the resulting co-designs enable the acceleration of DNNs even on small FPGAs. We benchmark our designs on standard TinyML applications such as keyword spotting, image classification, and person detection.

# I. INTRODUCTION

Deep Neural Networks (DNNs) are known to be computationally demanding, requiring significant amounts of computational resources and memory for both training and inference. DNNs are widely used in resource-constrained applications such as the *Internet of Things* (IoT) [1], *EdgeAI* [2], *TinyML* [3], etc. On the hardware side, advances in the semiconductor industry have made the development of *custom AI accelerators* feasible. RISC-V [4] is an open standard instruction set architecture (ISA) that is quickly gaining traction in academia and industry. RISC-V has various customization possibilities, one of which is to design custom accelerator logic that is tightly coupled with the CPU and is readily utilized on the software side. This enables a smooth hardware/software co-design approach [5] that can unlock significant degrees of optimization potential.

Building ASICs is time-consuming and costly, and they are inflexible in terms of the *flexibility-performance* tradeoff. On the other hand, general-purpose processors are not optimized for specialized DNN applications. A completely custom System-on-Chip (SoC) on an FPGA requires a significant development effort. RISC-V ISA extensions [4] in the form of custom functional units (CFUs) [6] offer an attractive tradeoff. Here, an SoC consisting of a general-purpose soft RISC-V core is used in

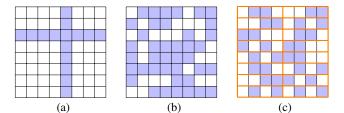


Fig. 1: Different sparsity structures: (a) Structured sparsity, resulting from structured pruning, which removes whole columns or rows from, e.g., a convolution matrix. (b) Unstructured sparsity, resulting from unstructured pruning, which removes arbitrary weights. (c) Semi-structured sparsity, resulting from semi-structured pruning, a.k.a. n:m pruning that zero-outs n weights every m elements; shown is a 2:4 pattern.

combination with instruction extensions designed as hardware accelerators with custom logic that do not require a significant design effort but, at the same time, provide acceleration for DNN workloads [6, 7].

DNNs are typically over-parameterized; this means that pruning a DNN can often be carried out significantly without impacting its performance. The pruning of DNNs has been the subject of significant prior research [8, 9]. Pruning a neural network can involve removing neurons, filters, weights, or layers, typically leading to sparser DNNs. The main goals are decreasing memory utilization, latency, and energy consumption [10]. While DNN pruning approaches can be classified from various perspectives, one classification is based on *structure*. Based on the structure of sparsity, the DNN pruning method can be fully structured, semi-structured, or unstructured. The different types of sparsity in a DNN model resulting from the different pruning methods are illustrated in Figure 1.

Fully structured pruning, such as layer or filter pruning (in the case of CNNs), is often accompanied by accuracy degradation. In comparison, the accuracy degradation for unstructured or semi-structured pruning is significantly less. Many popular DNN architectures have been pruned using unstructured pruning with high sparsity ratios [11]. A *sparsity ratio x* is defined as the percentage of zeros in a model. Unstructured or semi-structured pruning has a drawback; obtaining acceleration from it on general-purpose processors is infeasible.

CPU-based architectures most often only support the execution of dense computations and uniform data structures and perform the processing of sparse computations much less efficiently. One approach to performing sparse computations is to store sparse matrices in compressed format, which retains non-

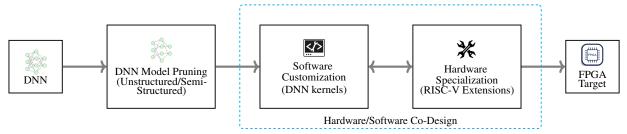


Fig. 2: Overview of our method for hardware/software co-design of RISC-V extensions to accelerate sparse DNNs on FPGAs. The process starts with a DNN model, which is pruned using unstructured or semi-structured pruning. Software customization of DNN kernels and hardware specialization of RISC-V extensions are jointly performed according to the co-design approach.

zero elements and indexing information [12]. This approach is useful for very high sparsity (90%–99%). For DNNs, the sparsity is high but not nearly as high as this. Therefore, the compressed storage approach becomes unsuitable. This strongly motivates exploiting customizable hardware to accelerate modern DNNs pruned with semi-structured or unstructured pruning.

Literature on accelerating sparse DNNs (unstructured or semi-structured) can be classified into two categories. The first category targets general-purpose processors, including CPUs and GPUs. For example, NVIDIA introduced a 2:4 pruning scheme that can be accelerated on NVIDIA Ampere architecture GPUs [13]. In [14], the authors propose a code generator for accelerating sparse DNN models on GPUs. In the second category of works, customized accelerator architectures are proposed for accelerating sparse DNNs, such as SNAP [15] and DANNA [16]. To the best of our knowledge, existing work on extending the instruction set architectures for accelerating sparse models is limited. In IndexMAC [17], the authors accelerated models with structured sparsity by extending a RISC-V CPU and demonstrated a speedup of 1.80–2.14×. Utilizing completely general-purpose processors is less efficient, while using solely custom accelerators is less flexible and not readily available. The advantage of proposing instruction extensions of a RISC-V is that it offers a decent tradeoff between the two cases. In a hardware/software co-design approach, hardware customization, software specialization, and model (DNN) optimization are jointly approached. The result is a modest increase in hardware resources and substantial performance speedups over CPU-only implementations as will be demonstrated.

With this motivation, we propose accelerating DNNs by exploiting semi-structured and unstructured sparsity using a hardware/software co-design approach and implementing this approach by providing instruction set extension units in hardware for RISC-V CFUs (illustrated in Fig. 2). Our contributions are as follows:

- We propose instruction set extensions for RISC-V CPUs to support unstructured sparsity for accelerated DNN processing at a modest increase in FPGA resource usage. These can be used for both TinyML and normal DNN workloads. In contrast to other approaches, our design makes no assumptions on the structure and number of zeros.
- Instruction set extensions for RISC-V CPUs are proposed to support semi-structured sparsity with accompanied software specialization. A novel lookahead encoding scheme is pro-

- posed here that does not compromise the DNN's performance.
- 3) We introduce a combined design for accelerating a DNN to support both unstructured pruning and semi-structured pruning. This dual-pruning capability is beneficial because it allows the model to simultaneously leverage each pruning method's distinct degrees of freedom, thereby enhancing computational efficiency and reducing model complexity.

#### II. BACKGROUND AND PRELIMINARIES

This section provides a brief overview of the RISC-V instruction set architecture (ISA) [4] and CFU Playground [6].

#### A. Custom Functional Units

The RISC-V ISA [4] allows for tailored instruction set extensions and accelerator design. Instruction encoding spaces and variable-length encoding make this accessible, letting developers customize processors while still using the standard ISA toolchain. Customization of the RISC-V processor architecture is achieved with so-called CFUs. These CFUs refer to custom logic added in hardware that is tightly coupled with the processor to provide specialized functionality. A CFU is addressed by the RISC-V ISA using the *R-type* instruction. The format of an R-type 32-bit instruction is shown in Figure 3 and can be described as follows: The *opcode* (7 bits) together with 3-bit *funct3* and 7-bit *funct7* specify the type of the instruction format and the operation to be performed. Fields *rs1* and *rs2* denote two source registers and *rd* the destination register, each addressed by 5 bits.

Notably, CFUs do not have direct access to the main memory. Therefore, the CPU acts as an intermediate node to transfer the

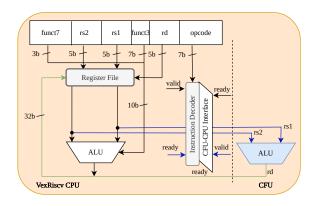


Fig. 3: CPU-CFU interface using R-type instruction of RISC-V.

data in memory to the CFU. Handshaking between CPU and CFU is handled using valid and ready signals. General pipeline stages can be described as follows. When the field opcode matches *custom-0* (predefined for custom instructions [4]), the CPU recognizes that the instruction needs to be forwarded to the CFU. At the same time, the register file resolves the *rs1* and *rs2* addresses to two 32-bit values. These bits form the input to the CFU along with the *funct7* and *funct3* values. Once a computation has finished inside the CFU, which can take one or multiple clock cycles, a valid signal is set to notify the CPU, and the result is written back from the CFU to the register file.

## B. CFU Playground

Building custom instructions can be a tedious process as it requires hardware/software co-design along with the ability to profile, modify, and rapidly prototype. With the goal of enabling a rapid exploration and design of CFUs, CFU Playground [6] was developed. The CFU Playground workflow can be divided into three main stages: deployment, profiling, and optimization. A TensorFlow Lite model is given to CFU Playground, which uses a RISC-V compiler alongside SymbiFlow<sup>1</sup> or Vivado for synthesis in the *deployment* stage. Then, in the *profiling* stage, various options are available to identify suitable candidates for optimization using custom functional units. Next, in the optimization phase, custom instructions are designed, and the three stages are repeated. CFU Playground uses an open-source implementation of a RISC-V processor known as VexRiscv [18]. A LiteX SoC configuration is placed within VexRiscv. The CPU-CFU interface provides a very tight coupling between the CPU and the added custom functionality. During FPGA synthesis, place, and route, the interface disappears, and the CFU essentially becomes a CPU pipeline component. The new instructions can be utilized in C or C++ application programs through an inline assembly macro provided. No adjustments to the RISC-V GCC toolchain are necessary.

#### III. PROPOSED APPROACH

This section provides a comprehensive breakdown of our proposed approach. First, we describe the basics (Section III-A), followed by proposing hardware designs to support semi-structured (Section III-B), unstructured pruning (Section III-C) and finally the combined design (Section III-D).

### A. Baseline

Our starting point is a VexRiscv soft-core with five pipeline stages and a CFU implementing a Single Instruction, Multiple Data (SIMD) MAC instruction (cfu\_simd\_mac) that takes four INT8 weights (filter[i]) and four INT8 activations/inputs (input[i]) as two inputs of a custom instruction and returns the multiply-and-accumulate result. This initial design is provided by TFLite<sup>2</sup> included within CFU Playground.

Listing 1 displays the baseline pseudo-code of a convolutional kernel with kernel height output\_height, kernel width output width, and output channels output channels, which

utilizes this baseline CFU design as a SIMD MAC instruction (cfu simd mac).

Listing 1: Pseudo-code of baseline convolutional kernel.

```
for (output_height) {
  for (output_width) {
    for (out_channel) {
      for (int i=0; i<in_channel; i+=4) { // 4x4 MAC
            cfu_simd_mac(filter[i], input[i]); }}}}</pre>
```

## B. Semi-Structured Sparsity Accelerator (SSSA)

In the baseline design (see Listing 1), a single call to cfu\_mac multiplies four INT8 values and returns the accumulated sum. In semi-structured pruning, sparsity manifests as blocks of zero weights. Therefore, to efficiently exploit sparsity, blocks of consecutive zeros must be skipped. One possible approach is to design a CFU that only processes non-zero blocks of weights and simply skips all other blocks; however, this introduces overhead within the innermost loop.

Our approach involves co-designing hardware and software components tailored to the DNN application. Notably, DNN weights remain static at runtime, and FPGAs provide bit-level granularity. Exploiting these two characteristics, we propose a pre-processing step of the list of weights for calculating the number of consecutive all-zero blocks following each non-zero block and then encoding this number into the non-zero block weights (see Algorithm 1). At execution time, this encoded counter is extracted in hardware and used for incrementing an induction variable in the innermost loop through custom instructions (Listing 2).

Algorithm 1 processes a given 3D matrix of CNN weights (kerne1) with dimensions corresponding to the number of input channels (C), height (H), and width (W). The algorithm iterates over each kernel element in the height and width dimensions,

**Algorithm 1** Encode CNN Kernel Weights with Lookahead Information

```
Require: CNN kernel represented as a 3D matrix (kernel)
Ensure: Encoded CNN kernel with lookahead information
 1: Initialize kernel dimensions: C (number of input channels), H (height),
     W (width)
    for h = 0 to H-1 do
 3:
         for w = 0 to W-1 do
 4:
             for c = 0 to C-1 step 4 do
 5:
                 i nxt \leftarrow c + 4
 6:
                 \overline{\text{skip}} blocks \leftarrow 0
 7:
                 while i nxt < C and skip blocks < 4 do
 8:
                    if checkBlkSkip(kernel[h][w][i_nxt]) then
 9.
                         skip blocks \leftarrow skip blocks + 1
10:
                        i \text{ nxt} \leftarrow i \text{ nxt} + 4
11:
                    else
12:
                        break
13:
                     end if
14:
                end while
15:
                for k = 0 to 3 do
16:
17:
                        encodeLastBits(kernel[h][w][c+k], skip blocks)
18:
                    end if
19:
                end for
             end for
20:
21:
         end for
22: end for
23: return Encoded CNN kernel
```

<sup>&</sup>lt;sup>1</sup>Open-source flow for generating bitstreams from Verilog (https://github.com/SymbiFlow)

<sup>&</sup>lt;sup>2</sup>TensorFlow Lite (TFLite) is a collection of tools to convert and optimize TensorFlow [19] models to run on mobile and edge devices

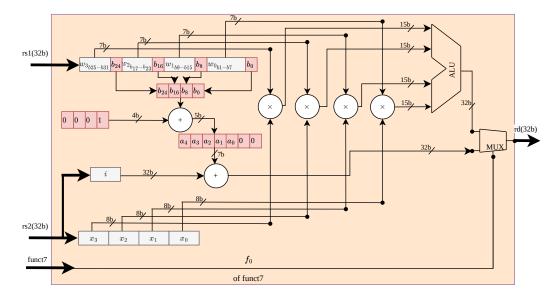


Fig. 4: RTL diagram of the proposed hardware SSSA for exploiting semi-structured sparsity.

block1	blo	ck2	·	]	blo	ck3	3		blo	ck4	4		blo	ck5			blo	ck(	ò	!	blo	ck7	7	;
4 7 3	1 0	0	0	0	0	0	0	0	11	7	12	4	0	0	0	0	13	0	12	4	0	1	0	0
0 0 1	0				·				0	0	1 0	1					0	0	0	0				

Fig. 5: The first row shows 7 blocks of DNN weights, each containing four INT8 weights. Algorithm 1 (encoding) performs a pass over all blocks, annotates each non-zero block with the information of the number of succeeding all-zero blocks, and encodes their number. The 2nd row shows the calculated code.

Fig. 6: The first row shows four weights along with their binary representations, and the second row shows the corresponding 4-bit code obtained in Figure 5. The sign bit of the weight is saved, and each bit in the code is appended to the LSB of the corresponding weight.

testing four consecutive blocks of weights along the input channel dimension for being zero or not.

When a block of four consecutive zeros is detected, the skip\_blocks counter is incremented, which can range from 0 to 15. This counter indicates how many all-zero blocks can be skipped during computation, thus optimizing the execution by reducing unnecessary operations. An example of this step is illustrated in Figure 5.

The dynamic range of INT8 weights is limited to [-64, 63] so as to not use the most significant bit after the signed bit, effectively simulating INT7 precision. The EncodeLastBits function (see Algorithm 2) embeds the 4-bit skip\_blocks value into each block of four DNN weights by shifting the bits of each weight to the left, making space for the skip bit, and then inserting the skip bit into the least significant bit (LSB) of each weight. The encoding procedure is visualized in Figure 6. By encoding the skip information directly within the weights, our hardware design, as shown in Figure 4, can directly use the extracted information of as many blocks to skip to increment an induction variable (i) in the innermost loop, as shown in Listing 2. Thereby, we can avoid any runtime overhead in software tests.

According to Line 12 of Algorithm 2, the LSB of a given weight is not overwritten directly but appended as the LSB with all higher significant bits shifted left. Our experiments (see Section IV-G) show that "sacrificing" one bit does not adversely affect the DNN accuracy for the considered applications. Next, we describe 1) the details of the hardware design and 2) the kernel customization more closely.

# Algorithm 2 encodeLastBits

```
1: function ENCODELASTBITS(weights[4], skip_blocks)
 2:
         for i = 0 to 3 do
 3:
             /* Isolate the sign bit */
 4:
             sign bit \leftarrow (weights[i] >> 7) & 0b1
 5:
             /* Extract skip bit */
 6:
             skip\_bit \leftarrow (skip\_blocks >> i) \& 0b1
 7:
             /* Remove the MSB after the sign bit *
 8:
             weights[i] \leftarrow weights[i] & 0b10111111
 9:
             /* Shift bits one position to the left */
10:
             \texttt{weights[i]} \leftarrow (\bar{\texttt{weights[i]}} << 1) \; \& \; 0b011111110
             /* Insert skip bit */
12:
             weights[i] \leftarrow weights[i] \mid skip\_bit
13:
             /* Restore the sign bit */
14:
             weights[i] \leftarrow weights[i] | (sign bit << 7)
15:
         end for
16:
         return weights
17: end function
```

1) Hardware Design: The CFU, as shown in Figure 4, implements two instructions (sssa\_mac and sssa\_inc\_indvar): sssa\_inc\_indvar takes the LSB (Least Significant Bit) of each of the four weights (first operand of the instruction) to offset the induction variable (i), while sssa\_mac performs a MAC operation on four 7-bit weights and four 8-bit input values. The LSB of the funct7 field is used to choose between these two instructions.

In Figure 4, the first operand, given by rs1, a 32-bit input register, provides four consecutive 7-bit weights  $(w_3, w_2, w_1, w_0)$ augmented by the corresponding encoding information bits  $(b_{24}, b_{16}, b_8, b_0)$ . The latter are extracted to form a 7-bit increment value. The second operand, given by rs2, a 32-bit input register, is either interpreted as the value of the induction variable i or four 8-bit input values  $(x_3, x_2, x_1, x_0)$  depending on the particular instruction used, which is differentiated by the LSB of funct7 ( $f_0$ ). If funct7 indicates a MAC operation, four 7-bit weights and four 8-bit values are multiplied and accumulated. Otherwise, the lookahead information contained in  $(b_{24}, b_{16}, b_8, b_0)$  of rs1 is used to increment the induction variable i by multiples of four. This is achieved by adding one to the bits encoding skip blocks information  $(b_{24}, b_{16}, b_8, b_0)$  and left shifting by two to multiply by four, obtaining the eventual 7-bit increment  $(a_4, a_3, a_2, a_1, a_0, 0, 0)$ , which leads to skipping the specified number of zero-blocks in the innermost loop, as summarized in Listing 2.

Listing 2: Pseudo-code of specialized kernel for SSSA.

```
for (output_height){
  for (output_width){
    for (out_channel){
      int i = 0;
      while (i<in_channel){
        sssa_mac(filter[i], input[i]); // 4x4 MAC
      i = sssa_inc_indvar(filter[i], i);}}}</pre>
```

2) Kernel customization: In order to use the design unit, the kernel shown in Listing 1 is modified to Listing 2 as follows: we replace the for loop with a while loop in the innermost loop of the baseline code for a convolutional kernel. Our two custom instructions are then inserted in this while loop. Instruction sssa\_inc\_indvar processes four weights packed as one 32-bit operand (including four bits for lookahead information) filter[i] and the current value of the induction variable (i), returning the updated value of the induction variable. Based on the encoded lookahead information in the block of weights, up to a maximum of 15 subsequent blocks can be skipped. The other instruction, sssa\_mac, multiplies four 7-bit weights with four 8-bit input values as described in Section III-B1.

# C. Unstructured Sparsity Accelerator (USSA)

While semi-structured sparsity will be shown to offer a notable acceleration, it still has the finest granularity of only full blocks and limits the number of blocks to be skipped. In the following, we suggest a co-design solution for fully unstructured pruning. The corresponding design can be combined and integrated with previous approaches to accelerate both unstructured and semi-structured pruning. The following sections outline our proposed approach for supporting the acceleration of unstructured sparsity.

- 1) Baseline: Our approach employs a baseline single sequential MAC unit that multiplies four input values by four weight values over four cycles, returning the accumulated sum. This baseline sequential design consistently requires four clock cycles regardless of the presence of zeros in the weights.
- 2) Hardware Design: Proposed is a variable-cycle MAC unit that takes only as many cycles as non-zero weight elements in a block except a single cycle for an all-zero block. In our RISC-V design shown in Figure 7, each block contains four INT8 weights. The input to the CFU comes through two 32-bit input registers (shown on the left side in the figure); these are four weights  $(w_3, w_2, w_1, w_0)$  and four input values  $(x_3, x_2, x_1, x_0)$ . Each of the four weights is compared in parallel to zero, generating 4-bit case signal  $(c_3, c_2, c_1, c_0)$ . The case signal control logic then processes this signal to produce four control signals  $(cl_0, cl_1, cl_2, cl_3)$ , which dictate the selection logic of the following two sets of multiplexers. The multiplexers, each controlled by the control signals, selectively pass and align the input weights  $(w_3', w_2', w_1', w_0')$  and input values  $(x_3', x_2', x_1', x_0')$ . After the data is aligned, it is passed to a sequential MAC unit that takes as many cycles as non-zero weights in the block, with the exception of an all-zero block, in which case one cycle is taken.
- 3) Kernel Customization: Utilizing unstructured sparsity requires a simple modification: replacing the baseline MAC (see Listing 1) instruction with the usss\_vcmac, which exploits the unstructured sparsity via our variable-cycle sequential MAC design. The considered baseline performs the multiplications sequentially since we consider the case of a *single* multiplier. Whereas the baseline MAC unit always takes *four* clock cycles per block. In contrast, usss\_vcmac takes a *variable* number of clock cycles depending on the number of zeros.

# D. Combined Sparsity Accelerator (CSA)

Finally, we propose the Combined Sparsity Accelerator (CSA), which integrates the functionalities of both prior designs. In practice, a combined approach using both semi-structured and unstructured sparsity techniques maximizes performance gains because DNN models often exhibit both sparsity types and can be optimized for both types of sparsities. This dual capability allows for more degrees of freedom during pruning a DNN. The combined design has two instructions: csa inc indvar and csa vcmac (shown in Listing 3). csa inc indvar behaves in the same way as sssa inc indvar instruction introduced before our SSSA design to accelerate the semi-structured sparsity while csa vcmac is just a variable-cycle MAC unit similar to the usss vcmac in our USSA design to accelerate the unstructured pruning except that the weights are considered to be of 7 bits. The CSA combines the advantages of both the USSA and the SSSA.

Listing 3: Pseudo-code of the specialized kernel for CSA.

```
for (output_height){
  for (output_width){
    for (out_channel){
      int i = 0;
      while (i<in_channel){
        csa_vcmac(filter[i], input[i]);
      // 4x4 variable-cycle sequentialMAC
      i = csa_inc_indvar(filter[i], i);}}}</pre>
```

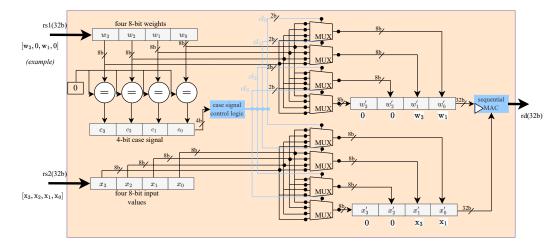


Fig. 7: RTL diagram of the proposed USSA for exploiting unstructured sparsity.

## IV. EXPERIMENTS

In this section, we assess our proposed designs in terms of achievable speedup and FPGA resource utilization.

#### A. Evaluation Methodology

Our designs were implemented on an Arty A7-35T FPGA board. Although our experiments primarily focus on convolutional and fully connected layers within CNNs, our designs can be seamlessly adapted to support other layer types as well, such as LSTMs and fully connected layers without any further instruction set extension or modification.

## B. Models and Datasets

Three datasets and four models are used in our evaluation. Two models are used with CIFAR-10 [20], which is a dataset with 50,000 training and 10,000 testing 32x32 color images divided into ten classes. Keyword Spotting focuses on recognizing specific words. The Google Speech Commands (GSC) v2 dataset [21] contains 65,000 audio signals spoken by a diverse group of individuals, and it consists of 30 audio classes such as "up," "down," "yes," "no," etc. Person detection refers to a machine learning task where the goal is to detect the presence or absence of a person in an image. The Visual Wake Words (VWW) dataset [22] was derived from the COCO dataset [23] and includes around 115,000 training and validation images. We evaluate four models: VGG16, ResNet-56, MobileNetV2, and DSCNN.

# C. Pruning Methodology

We have not delved into training any pruned DNN model and optimizing for accuracy due to space constraints and because our primary focus is on proposing RISC-V extensions to accelerate structured and unstructured sparsity. In principle, any pruning method that generates a model as input with unstructured sparsity or semi-structured sparsity conforming to our sparsity pattern can be utilized. For our purposes, we applied an iterative pruning approach with explainable-AI-based ranking similar to the methods described in [24], [25], and [26].

# D. Speedup for USSA

The USSA (Unstructured Sparsity Accelerator) is used to accelerate models with unstructured sparsity. No constraints are enforced on the structure of the sparsity. In the baseline scenario, no computational reductions are made to exploit the sparsity.

The impact of sparsity  $(x \in [0,1])$  on speedup can be quantified in general by examining the probability distribution of zeros and ones within DNN weights assuming an IID<sup>3</sup> distribution. In that case, the analytical average number of clock cycles  $(c_a)$  can be computed as:

$$c_a = \sum_{k=0}^{4} {4 \choose k} \cdot x^k \cdot (1-x)^{4-k} \cdot (4-k)$$

In this equation,  $\binom{4}{k}$  is the binomial coefficient representing the number of ways to choose k ones in a block of four elements, and 4-k denotes the number of cycles required for each configuration. In the ideal case, zero clock cycles are needed for a block of four zeros. In contrast, our design still requires one clock cycle for a block of four zeros. The observed average number of clock cycles  $(c_0)$  can then be given as:

$$c_o = \sum_{k=0}^{3} {4 \choose k} \cdot x^k \cdot (1-x)^{4-k} \cdot (4-k) + {4 \choose 4} \cdot x^4 \cdot (1-x)^0$$

The analytical speedup  $(s_a)$  and observed speedup  $(s_o)$  can be obtained as  $s_a = 4/c_a$  and  $s_o = 4/c_o$ , respectively. A comparison of the two speedups is shown in Figure 8. The effect of a single cycle overhead when all four weights in a block are zero becomes only noticeable at very high sparsities. However, this additional cycle can be avoided using CSA. Additionally, our approach can be extended to cases involving INT4 and INT2 weights, where the speedup over the baseline would be higher. For example, one 32-bit register can contain eight INT4 weights, and if seven of them are zeros, then the USSA will take a single clock cycle, whereas the baseline will take eight clock cycles.

<sup>&</sup>lt;sup>3</sup>independent, identically distributed (IID)

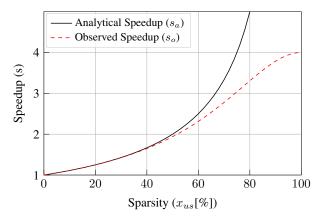


Fig. 8: Analytical and observed speedups for USSA (Unstructured Sparsity Accelerator).

# E. Speedup for SSSA

The SSSA (Semi-Structured Sparsity Accelerator) accelerates DNN models by exploiting semi-structured sparsity. The speedup is benchmarked against a baseline where every block of weights and inputs is processed, irrespective of their values. The analytical speedup  $(s_a)$  is calculated by the ratio of the total number of weights to the number of zero weights while the observed speedup  $(s_o)$  is measured by comparing the clock cycles required to process a convolutional layer in the baseline configuration versus the number of clock cycles needed by the SSSA.

Figure 9 provides a comparison of the two speedups. Note that the observed speedup  $(s_o)$  sometimes exceeds the analytical speedup  $(s_a)$  due to reduced overhead, as the accelerator bypasses entire blocks of zero values in a loop, eliminating unnecessary iterations. Speedups were measured for a convolutional layer. It is to be noted that even higher speedups should be achievable for models with higher sparsity than considered in the paper.

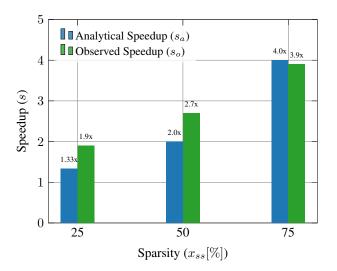


Fig. 9: Analytical and observed speedups for SSSA (Semi-Structured Sparsity Accelerator).

## F. Speedup of DNN models using CSA

The Combined Sparsity Accelerator (CSA) integrates the functionalities of both prior designs. Observed speedups for four DNN models using the CSA are presented in Figure 10.

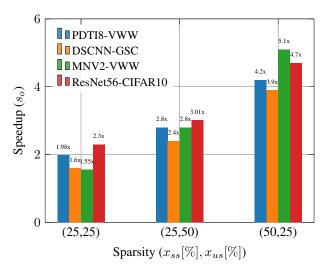


Fig. 10: Speedups of considered DNN models with CSA (Combined Sparsity Accelerator) for three different configurations of unstructured sparsity  $(x_{us})$  and semi-structured sparsity  $(x_{ss})$ . Even higher speedups should be achievable for models with higher sparsity than considered in the paper.

#### G. Impact of Losing One Information Bit

In our design for semi-structured pruning, one bit per weight is used to encode sparsity information. Therefore, the model performance loss from sacrificing one bit needs to be analyzed. In our experiments, we found that effectively reducing the precision from INT8 to INT7 for each of the three considered applications did not impact the accuracy. Concrete results are shown in Table II.

TABLE II: Accuracy comparison between INT8 and INT7 precision.

Model-Dataset	Accuracy (INT8)	Accuracy (INT7)				
ResNet-56 on CIFAR10	93.51%	93.53%				
MobileNetV2 on VWW	91.53%	91.42%				
DSCNN on GSC	95.17%	95.10%				

Our results are consistent with observations in previous works [28], where it was found that quantizing to INT7 weights does not incur any noticeable loss in accuracy and may, in some cases, increase the performance on the test dataset. A slight increase in some cases may occur due to better generalization on the test dataset.

#### H. Comparison with the State-of-the-Art

Though not easily comparable, we selected two other works for comparison: IndexMAC [17] and a fully parallel sparse convolution accelerator by Lu et al. [27]. Table I summarizes the comparison. The approach by Lu et al. employs a fully hardware-parallel architecture, while IndexMAC is a CPU architecture

TABLE I: Comparison of different methods for accelerating sparse DNNs.

Method	Semi-Structured	<b>Unstructured Sparsity</b>	Sparsity Pattern	Speedup	Sparsity Ratios	Architecture
Ours (USSA)	×	✓	NA	2–3×	High	CPU+HW
Ours (SSSA)	✓	×	4:4	$2-4\times$	Low	CPU+HW
Ours (CSA)	✓	✓	4:4, random	4–5×	Moderate	CPU+HW
IndexMAC [17]	✓	Х	2:4	2-3×	Moderate	CPU+HW
Lu et al. [27]	NA	✓	Low	2.4-12.9×	NA	HW

with instruction extensions added to a RISC-V core to accelerate semi-structured pruning. The tradeoff between a full hardware accelerator and a CPU extension is that the former offers higher throughput and lower latency due to the simultaneous processing of operations. In contrast, the latter is more resource-efficient, utilizing hardware resources more effectively by reusing them across multiple operations.

IndexMAC employs specific sparsity patterns, such as 1:4 or 2:4 pruning. This sparsity pattern lies "mid-way" between the two of our designs. Note that our unstructured pruning approach imposes no constraints on sparsity patterns. This leads to higher levels of sparsity. Our semi-structured pruning approach demands that a block of four weights be zero, which we refer to as a 4:4 sparsity pattern.

In terms of speedup, our unstructured pruning design (USSA) is superior to the IndexMAC approach for the reason that it provides similar speedup for similar sparsity but imposes no constraint on the specific sparsity pattern, leading to higher pruneability.

Moreover, our semi-structured pruning approach (SSSA) is both faster and simpler due to our novel lookahead encoding. By reserving one bit per weight for lookahead information, we efficiently skip blocks with zero weights, which accelerates the processing of DNNs. This novel method of leveraging embedded information within DNN weights to enhance acceleration is, to the best of our knowledge, not found in previous work. Another key aspect is the combination of semi-structured and unstructured pruning in a single design. This hybrid approach offers the best of both of our designs.

#### I. Resource Usage

We provide FPGA resource usage for our designs for accelerating semi-structured and unstructured pruning in Table III. The resource usage was obtained for a Xilinx XC7A35T FPGA comprising 33,280 logic cells and 90 DSP slices. In both cases, the increase in LUT utilization is less than 4%, and the usage of flip-flops (FF) is around 6% while one additional DSPs are used. The two designs may be used independently or merged to accelerate both unstructured and semi-structured sparsity simultaneously at the cost of slightly more FPGA resource utilization. The combined design utilizes an additional 4.39% more LUTs, 8.23% more FFs, and two additional DSPs. The standard clock frequency of 100 MHz of the RISC-V-based system (LiteX SoC [6]) was considered throughout all our experiments. The addition of our proposed CFUs did not affect the frequency. Note that if a CFU design would be on the critical path, there is the option to pipeline the CFU datapath, as CFU Playground allows for multicycle custom instructions.

TABLE III: FPGA resource usage comparison for different sparsity accelerator CFUs.

Design Type	RIS	SC-V	Cost Increment [%				
	w/o CFU	with CFU					
USSA							
LUTs	2,482	2,516	1.36%				
Slice FF	1,470	1,563	6.32%				
BRAMs	9	9	0%				
DSPs	4	5	25%				
SSSA							
LUTs	2,473	2,568	3.84%				
Slice FF	1,481	1,578	6.55%				
BRAMs	9	9	0%				
DSPs	4	5	25%				
CSA							
LUTs	2,459	2,567	4.39%				
Slice FF	1,470	1,591	8.23%				
BRAMs	9	9	0%				
DSPs	4	6	50%				

#### V. CONCLUSION AND FUTURE WORK

In this paper, we proposed novel RISC-V extensions aimed at accelerating DNNs by exploiting both unstructured and semi-structured sparsity. Our designs are customizable to fit various use cases and can benefit from FPGA reconfigurability. The novel aspect of our design to accelerate semi-structured pruning (SSSA) is the lookahead approach that encodes sparsity information into DNN weights by using the last bit of each INT8 weight. This strategy simplifies the design and requires minimal extra FPGA resources. The novel aspect of our design to accelerate unstructured sparsity (USSA) is a variable-cycle sequential multiply-accumulate unit. We then combined both designs and achieved up to  $5\times$  speedup for selected DNN-based applications, with the potential for more significant speedups in other scenarios.

In order to overcome the limitation imposed by the register-toregister (64 bit) CFU-CPU interface, we also plan to investigate co-processor architectures in the future.

### ACKNOWLEDGMENT

This work was partly supported by the Fraunhofer Institute for Integrated Circuits IIS, Erlangen, Germany, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under project number 524986327 (NA<sup>3</sup>Os), and the Federal Ministry for Education and Research (BMBF) within project "DI-EDAI" (16ME0992).

#### REFERENCES

- [1] I. Lee and K. Lee. "The Internet of Things (IoT): Applications, Investments, and Challenges for Enterprises". In: *Business Horizons* 58.4 (2015), pp. 431–440. DOI: 10.1016/j.bushor.2015.03.008.
- [2] R. Singh and S. S. Gill. "Edge AI: A Survey". In: *Internet of Things and Cyber-Physical Systems* 3 (2023), pp. 71–92. DOI: 10.1016/j.iotcps.2023.02.004.
- [3] R. Kallimani, K. Pai, P. Raghuwanshi, S. Iyer, and O. L. A. López. "TinyML: Tools, Applications, Challenges, and Future Research Directions". In: *Multimedia Tools and Applications* 83 (2023), pp. 29015–29045. DOI: 10.1007/s11042-023-16740-9.
- [4] A. Waterman and K. Asanovic. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA. 2017. URL: https://riscv. org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.
- [5] S. Ha and J. Teich, eds. Handbook of Hardware/Software Codesign. Springer, 2017. DOI: 10.1007/978-94-017-7267-9.
- [6] S. Prakash, T. Callahan, J. Bushagour, C. Banbury, A. V. Green, P. Warden, T. Ansell, and V. J. Reddi. "CFU Playground: Want a Faster ML Processor? Do it Yourself!" In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE) (Antwerp, Belgium). IEEE, Apr. 17–19, 2023. DOI: 10.23919/ DATE56975.2023.10137093.
- [7] M. Sabih, B. Sesli, F. Hannig, and J. Teich. "Accelerating DNNs using Weight Clustering on RISC-V Custom Functional Units". In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)* (Valencia, Spain). Mar. 25–27, 2024. DOI: 10.23919/DATE58400.2024.10546844.
- [8] C. Hongrong, Z. Miao, and Q. Javen. "A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations". In: *The Computing Research Repository* (CoRR) (2023). arXiv: 2308.06767 [cs.LG].
- [9] C. Heidorn, M. Sabih, N. Meyerhöfer, C. Schinabeck, J. Teich, and F. Hannig. "Hardware-Aware Evolutionary Explainable Filter Pruning for Convolutional Neural Networks". In: *International Journal of Parallel Programming* 52 (Feb. 2024), pp. 40–58. DOI: 10.1007/s10766-024-00760-5.
- [10] M. Streubühr, R. Rosales, R. Hasholzner, C. Haubelt, and J. Teich. "ESL Power and Performance Estimation for Heterogeneous MPSoCs using SystemC". In: *Proceedings of the Forum on Specification and Design Languages (FDL)* (Oldenburg, Germany). IEEE, Sept. 13–15, 2011, pp. 202–209.
- [11] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag. "What is the State of Neural Network Pruning?" In: *Proceedings of the Conference on Machine Learning and Systems (MLSys)* (Austin, TX, USA). Mar. 2–4, 2020, pp. 129–146. URL: https://proceedings.mlsys.org/paper\_files/paper/2020/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf.
- [12] Y. Saad. Iterative Methods for Sparse Linear Systems. Other Titles in Applied Mathematics. SIAM, 2003. DOI: 10.1137/1. 9780898718003.
- [13] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius. "Accelerating Sparse Deep Neural Networks". In: *The Computing Research Repository (CoRR)* (2021). arXiv: 2104.08378 [cs.LG].
- [14] Z. Wang. "SparseRT: Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference". In: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT) (Virtual Event). ACM, Oct. 3–7, 2020, pp. 31–42. DOI: 10.1145/3410463.3414654.
- [15] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang. "SNAP: An Efficient Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference". In:

- *IEEE Journal of Solid-State Circuits* 56.2 (2021), pp. 636–647. DOI: 10.1109/JSSC.2020.3043870.
- [16] X. Liu and H. Feng. "DANNA: A Dimension-Aware Neural Network Accelerator for Unstructured Sparsity". In: Proceedings of the 5th International Conference on Communications, Information System and Computer Engineering (CISCE) (Guangzhou, China). IEEE, Apr. 14–16, 2023, pp. 306–310. DOI: 10.1109/CISCE58541. 2023.10142599.
- [17] V. Titopoulos, K. Alexandridis, C. Peltekis, C. Nicopoulos, and G. Dimitrakopoulos. "IndexMAC: A Custom RISC-V Vector Instruction to Accelerate Structured-Sparse Matrix Multiplications". In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE) (Valencia, Spain). IEEE, Mar. 25–27, 2024. DOI: 10.23919/DATE58400.2024.10546747.
- [18] SpinalHDL. VexRiscv Core. 2023. URL: https://github.com/ SpinalHDL/VexRiscv (visited on 06/14/2024).
- [19] Martín Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.
- [20] A. Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". Master Thesis. University of Toronto, Canada, 2009.
- [21] P. Warden. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition". In: *The Computing Research Repository* (CoRR) (2018). arXiv: 1804.03209 [cs.CL].
- [22] A. Chowdhery, P. Warden, J. Shlens, A. G. Howard, and R. Rhodes. "Visual Wake Words Dataset". In: *The Computing Research Repository (CoRR)* (2019). arXiv: 1906.05721 [cs.CV].
- [23] T.-Y. Lin et al. "Microsoft COCO: Common Objects in Context". In: The Computing Research Repository (CoRR) (2014). arXiv: 1405.0312 [cs.cV].
- [24] M. Sabih, F. Hannig, and J. Teich. "Utilizing Explainable AI for Quantization and Pruning of Deep Neural Networks". In: *The* Computing Research Repository (CoRR) (Aug. 20, 2020). arXiv: 2008.09072 [cs.CV].
- [25] M. Sabih, F. Hannig, and J. Teich. "DyFiP: Explainable AI-based Dynamic Filter Pruning of Convolutional Neural Networks". In: Proceedings of the 2nd European Workshop on Machine Learning and Systems (EuroMLSys) (Rennes, France). ACM, Apr. 5–8, 2022, pp. 109–115. DOI: 10.1145/3517207.3526982.
- [26] M. Sabih, A. Mishra, F. Hannig, and J. Teich. "MOSP: Multi-Objective Sensitivity Pruning of Deep Neural Networks". In: Proceedings of the IEEE 13th International Green and Sustainable Computing Conference (IGSC) (Pittsburgh, PA, USA). IEEE, Oct. 24–25, 2022, pp. 1–8. DOI: 10.1109/IGSC55832.2022.9969374.
- [27] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang. "An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs". In: Proceedings of the IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (San Diego, CA, USA). Apr. 28–May 1, 2019, pp. 17–25. DOI: 10.1109/FCCM.2019. 00013
- [28] D. Wu, Q. Tang, Y. Zhao, M. Zhang, Y. Fu, and D. Zhang. "EasyQuant: Post-training Quantization via Scale Optimization". In: *The Computing Research Repository (CoRR)* (2020). arXiv: 2006.16669 [cs.cv].