BiSon-e: A Lightweight and High-Performance Accelerator for Narrow Integer Linear Algebra Computing on the Edge

Enrico Reggiani
Polytechnic University of Catalonia
Barcelona Supercomputing Center
Barcelona, Spain
enrico.reggiani@bsc.es

Cristóbal Ramírez Lazo
Polytechnic University of Catalonia
Barcelona Supercomputing Center
Barcelona, Spain
cristobal.ramirez@bsc.es

Roger Figueras Bagué
Barcelona Supercomputing Center
Barcelona, Spain
roger.figueras@bsc.es

Adrián Cristal
Polytechnic University of Catalonia
Barcelona Supercomputing Center
Barcelona, Spain
adrian.cristal@bsc.es

Mauro Olivieri
Sapienza University of Rome
Roma, Italy
Barcelona Supercomputing Center
Barcelona, Spain
mauro.olivieri@uniroma1.it

Osman Sabri Unsal
Barcelona Supercomputing Center
Barcelona, Spain
osman.unsal@bsc.es

ABSTRACT

Linear algebra computational kernels based on byte and sub-byte integer data formats are at the base of many classes of applications, ranging from Deep Learning to Pattern Matching. Porting the computation of these applications from cloud to edge and mobile devices would enable significant improvements in terms of security, safety, and energy efficiency. However, despite their low memory and energy demands, their intrinsically high computational intensity makes the execution of these workloads challenging on highly resource-constrained devices. In this paper, we present *BiSon-e*, a novel RISC-V based architecture that accelerates linear algebra kernels based on narrow integer computations on edge processors by performing Single Instruction Multiple Data (SIMD) operations on off-the-shelf scalar Functional Units (FUs). Our novel architecture is built upon the *binary segmentation* technique, which allows to significantly reduce the memory footprint and the arithmetic intensity of linear algebra kernels requiring narrow data sizes. We integrate *BiSon-e* into a complete System-on-Chip (SoC) based on RISC-V, synthesized and Place&Routed in 65nm and 22nm technologies, introducing a negligible 0.07% area overhead with respect to the baseline architecture. Our experimental evaluation shows that, when computing the Convolution and Fully-Connected layers of the AlexNet and VGG-16 Convolutional Neural Networks (CNNs) with 8-, 4-, and 2-bit, our solution gains up to 5.6×, 13.9× and 24× in execution time compared to the scalar implementation of a single RISC-V core, and improves the energy efficiency of string matching tasks by 5× when compared to a RISC-V-based Vector Processing Unit (VPU).

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; **Embedded systems**.

KEYWORDS

Binary Segmentation, Edge Computing, Convolutional Neural Network, String Matching, RISC-V, Hardware Accelerator, Narrow Integer Arithmetic, Low-power design, Number Representation

1 INTRODUCTION

Contemporary Internet-of-Things (IoT), edge and mobile computing applications require high-performance. This demand is fueling a large research effort in low-power, high-performance embedded processors [25, 43]. Such devices, mainly constrained by power and cost, have to fulfill the performance and memory requirements of a vast collection of important application domains, such as deep learning, robotics, graph processing, and cryptography. Most of these application classes represent data as matrices and vectors, and express their computation through a set of linear algebra kernels. When targeting edge platforms, a popular approach to reduce energy demands and memory footprint requirements is to compact the data layout using a smaller data format while preserving the application accuracy. On the one hand, expressing and computing data exploiting low-precision floating-point formats [3] is gaining traction in the High Performance Edge Computing (HPEC) community, as they represent a good trade-off between data size and accuracy. On the other hand, narrow fixed-point and integer data representations (*i.e.*, byte and sub-byte) offer a better alternative in terms of Performance per Watt ratio, although they feature smaller number representations. One of the dominant applications of edge computing that leverages these compressed data formats is the Quantized Convolutional Neural Network (QCNN) inference, which exploits *quantization* to represent data and weights with data sizes typically ranging from eight to one bit with tolerable accuracy penalties[30, 33]. Other kernels belonging to important application classes for edge computing, such as graph computing and cryptography, widely rely on boolean matrices and vectors computations to traverse a graph or to encrypt/decrypt a message. These applications would greatly benefit from hardware and software solutions that efficiently compute narrow integer linear algebra kernels.

Accordingly, we present *BiSon-e* [1], a high-performance and lightweight architecture aimed at increasing the efficiency of linear algebra, narrow integer computations on edge processors. The proposed solution relies on a mathematical technique called *binary segmentation* [37], which reduces the memory footprint of matrices and vectors consisting of narrow integers, and considerably

[1]Binary Segmentation on-edge

The final publication is available at ACM via http://dx.doi.org/10.1145/3503222.3507746

decreases the arithmetic complexity of linear algebra computations. To the best of our knowledge, this is the first work developing a *binary segmentation* based architecture. *BiSon-e* is motivated by the lack of sufficient support for efficient narrow computations in current edge processors and Instruction Set Architectures (ISAs), as most of them do not implement memory and arithmetic instructions for data formats smaller than 8-bit. For example, compressing sub-byte data in memory needs a conversion to standard bitwidths before and after each computation, leading to performance and energy consumption inefficiencies. Moreover, processor Functional Units (FUs) are overprovisioned for computations involving narrow data sizes, and exhibit an Energy per Instruction (EPI) that does not scale with the input data size. Our key contribution is to increase the efficiency of narrow data formats in terms of data storage and linear algebra kernel computations, scaling their performance with the decrease of the data size. Instead of extending standard RISC-V ISA for new sub-byte data sizes, and designing custom hardware supporting them, we rely on data segmentation to fuse multiple arithmetic operations in a single instruction, performing Single Instruction Multiple Data (SIMD) computations on off-the-shelf scalar FUs.

The main contributions of this paper are summarized as follows:

- We perform a Design Space Exploration (DSE) of *binary segmentation* on 64-bit architectures. Guided by DSE, we design the *BiSon-e* architecture which features a *binary segmentation* enhanced Central Processing Unit (CPU) pipeline. We analyze a set of linear algebra computational kernels that can leverage *binary segmentation* to increase the performance of edge based narrow integer applications;
- We benchmark *BiSon-e* with three algorithms belonging to two demanding edge computing application classes, namely deep learning, and string matching, considering both performance and energy efficiency. Our solution improves the back-to-back runtime performance of the AlexNet and the VGG-16 Convolutional Neural Networks (CNNs) by a factor that ranges from 5.6× to 24× on 8-bit and 2-bit data sizes with respect to the single-core scalar implementation, and outperforms the string matching use-case vectorized implementation by a factor of 5× in terms of energy efficiency;
- We integrate, design, and fully implement the proposed architecture, including Place and Route (P&R), on a RISC-V based System-on-Chip (SoC), on both 65nm and 22nm technologies. We show that *BiSon-e* can be integrated into modern edge processors with a negligible 0.07% area overhead, and without any performance loss;

The organization of the paper is as follows. Section 2 presents the *binary segmentation* technique. Section 3 performs a DSE of *binary segmentation* on 64-bit CPUs. Section 4 details the *BiSon-e* architecture, discussing its design choices and features. Section 5 evaluates the experimental results obtained with the proposed solution. Section 6 reviews the main related work. Finally, Section 7 summarizes *BiSon-e*.

2 BINARY SEGMENTATION

In the class of applications requiring narrow integer computations, the data size needed by the algorithm is typically lower than the one allowed by the underlying architecture. Modern processors datapaths are often based on 32-bit or 64-bit, and thus byte and sub-byte computations underutilize both their arithmetic capabilities and data movements efficiency. Moreover, the current ISAs and programming languages typically lack adequate support to handle narrow data bitwidths. Consequently, the performance of workloads featuring narrow integer computations does not scale in concert with the data size. This work explores the *binary segmentation* technique to reduce these limitations. This technique interpolates data within the processor bitwidth [37] and improves the memory footprint and the arithmetic complexity of several fundamental linear algebra kernels.

An *n-dimensional* vector $v = [v_0, \ldots, v_{n-1}]$ populated with integers in the $[0, 2^b)$ range, with b denoting the element bitwidth can be represented by the single integer V_b:

$$V_b = \sum_{i=0}^{n-1} v_i 2^{bi} \tag{1}$$

This interpolation allows creating a compact storage scheme for matrices and vectors populated with bounded integers, as a single computer word can be composed of several elements belonging to v. Enhanced support for lower data sizes would dramatically decrease the applications memory footprint. However, in modern processors, the lower bound of data sizes that inherently support Equation (1) is often in the byte range. As a result, the advantages offered by the compact storage scheme described in Equation (1) for sub-byte data sizes can be mitigated by the overhead needed to pack and extract data from non-standard data sizes before and after their computation. Moreover, adding the support for narrow data sizes could be a demanding task, as it would imply changes at hardware, ISA, compiler, and software level. In this paper, we propose a novel approach to efficiently represent sub-byte data sizes via *binary segmentation*, and to compute linear algebra arithmetics with minimal changes in hardware and ISA, without the need to define new data formats at the software level. Indeed, the *binary segmentation* technique has been successfully explored, from a theoretical perspective, to decrease the arithmetic complexity of several arithmetic expressions: polynomial multiplication [16], multiplication of two complex numbers [36], Discrete Fourier Transform (DFT) [42], inner and outer product of two vectors [37], polynomial division [10], and polynomial Greatest Common Divisor (GCD) [12], and supports both signed and unsigned computations [42].

As a simple example on how this technique can be used to compute arithmetic operations on matrices and vectors, we can consider the sum of two vectors u and v, both compised of elements in the $[0, 2^b)$ range. Following Equation (1), and defining the *clustering width* (cw) as $cw = b+1$, we can create two integers U_{cw} and V_{cw} via *binary segmentation*, sum them as a single sum of integers, and recover the output vector, obtaining the element-wise sum of the two vectors. The *clustering width* is defined to be greater than the actual bitwidth of the input elements b, as it includes extra guard-band bits to avoid overflows in the segmented data due to carry propagation. This allows performing the summation of n narrow integers with only one summation of two long integers, instead of n summations of short integers. It is worth noticing that *binary segmentation* is not an approximate computing technique, since it guarantees exact computations, as the *clustering width* dimension already accounts for the number of bits needed to represent the computation output without introducing precision losses.

Below, we describe how *binary segmentation* can improve the efficiency of more representative kernels belonging to the linear algebra field, namely Inner Product (IP) and Linear Convolution (LC) of two vectors.

2.1 Inner Product of Two Vectors via Binary Segmentation

The IP of two vectors composed of n elements $u = [u_0, \ldots, u_{n-1}]$ and $v = [v_0, \ldots, v_{n-1}]$, having bitwidths b_u and b_v, can be obtained by the following expression:

$$IP = \sum_{i=0}^{n-1} u_i v_i \tag{2}$$

To compute the IP via *binary segmentation*, it is first necessary to reverse the vector v such that:

$$v'_i = v_{n-1-i} \qquad , i = 0, 1, \ldots, n-1 \tag{3}$$

According to Equation (1), we create the integers U_{cw} and V_{cw} from u and v', with the following *clustering width*:

$$cw \geq b_u + b_v + \lceil \log_2(n) \rceil \tag{4}$$

Then, the IP computed via *binary segmentation* is the multiplication between U_{cw} and V_{cw}, resulting in the integer W_{cw}. The IP result is derived from W_{cw} by extracting the bits expressed as follows:

$$IP = W_{\{(n-1)cw+cw,\ (n-1)cw\}} \tag{5}$$

Considering the reference example depicted in Figure 1a, we can evaluate the IP between $u = [7, 5]$ and $v = [4, 2]$ via *binary segmentation* employing a single integer multiplication. Specifically, we represent each element of the input vector with a bitwidth equal to the *clustering width* defined in Equation (4) (*i.e.*, 7-bit), and we revert the order of the elements belonging to v (blue). Then, we express the resulting vectors as single integers (green), and we perform their multiplication (yellow). Finally, we extract the IP result from the seven bits resulting from Equation (5) (red). For this example, we employed a single integer multiplication in place of two multiplications and one sum to obtain the final result. As detailed in Section 3, the same approach can be used to compute the IP between three to ten elements concurrently on a 64-bit architecture, for input sizes ranging from 8-bit to 1-bit.

2.2 Convolution of Two Vectors via Binary Segmentation

Given $u = [u_0, \ldots, u_{m-1}]$ and $v = [v_0, \ldots, v_{n-1}]$, we compute the vector $w = [w_0, \ldots, w_K]$, having length $K = m+n-1$, as the LC between u and v:

$$w_k = \sum_{i=0}^{K} u_i v_{k-i} \qquad , k = 0, 1, \ldots, K-1 \tag{6}$$

The same expression can be computed via *binary segmentation* by representing U_{cw} and V_{cw} as in Equation (1), with a *clustering width* of:

$$cw \geq b_u + b_v + \lceil \log_2(min\{m, n\}) \rceil \tag{7}$$

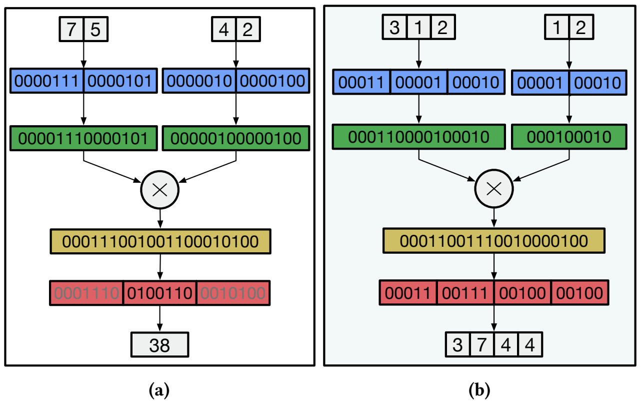


Figure 1: Examples of IP (a) and LC (b) kernel computations via *binary segmentation*, with *clustering widths* of 7-bit and 5-bit, respectively. The input vectors are represented with *clustering widths* bits (blue), merged into single variables (green), and multiplied (yellow). The final result is then extracted from the multiplication output (red).

Then, we recover w from the output of the integer multiplication between U_{cw} and V_{cw}. An example of LC between two vectors $u = [3, 1, 2]$ and $v = [1, 2]$ is shown in Figure 1b. First, the bitwidth of each element belonging to u and v is represented as a 5-bit number, according to Equation (7) (blue). Then, the two input vectors are converted to single integers (green) and multiplied (yellow). By segmenting the multiplication result into four 5-bit binary numbers, it is possible to recover the LC result[2] (red). This example only uses one multiplication to compute the LC between u and v, which would have required six multiplications and two additions to be computed with Equation (6). Figure 1a and Figure 1b highlights the steps required to obtain the final result.

The IP and LC examples reported in Figure 1 reduce their arithmetic complexity by a factor of 3× and 7×, respectively. Certainly, the ratio between the processor word size and the data size highly impacts the achievable arithmetic complexity reduction. Moreover, if the vector length does not fit the processor word size, the arithmetic problem must be partitioned into smaller-size subproblems. It is also worth considering the effort required to convert a set of vector elements into a single integer, and to extract the output elements from the integer multiplication result. These requirements increase the overall arithmetic complexity of *binary segmentation* if the underlying architecture is not efficient in clustering, extracting, and masking data, leading to a decrease in the overall performance gain. We deeply explore and quantify these considerations in Section 3, while in Section 4 we show how our architecture overcomes these limitations, allowing narrow integer linear algebra kernels to fully benefit from the advantages that *binary segmentation* offers.

3 DESIGN SPACE EXPLORATION

The proposed DSE aims to explore the benefits and the pitfalls of implementing *binary segmentation* on edge processors, exploiting

[2] $LC_{out} = [3 \times 1, 1 \times 1 + 3 \times 2, 2 \times 1 + 1 \times 2, 2 \times 2]$

standard CPU architectures. The efficiency of *binary segmentation* strictly depends on the ratio between the CPU registers size and the application data bitwidths. On the one hand, the greater this ratio is, the larger the number of elements embedded in a single operation. On the other hand, increasing the number of elements clustered in a single register implies a higher overhead required to pack data into single integers, or to extract the results from the multiplication output. Following Section 2, our evaluation mainly focuses on the IP and LC, as they represent the core kernels of our benchmarks. However, the proposed methodology can be extended and applied to other arithmetic operators that would benefit from this approach, such as the one listed in Section 2.

To characterize the *binary segmentation* technique on CPU architectures, it is important to define the number of elements that can be computed concurrently. We denote this set of elements as *input-cluster*, and we evaluate the *input-cluster* dimension on both 32-bit and 64-bit CPU architectures. As a reference, Figure 1a has *input-clusters* composed of two elements, while Figure 1b features asymmetric *input-clusters* of three and two elements. In this work, we consider the same *input-cluster* dimension for each operand of the FU. This choice is optimal for CPUs architectures, as they are equipped with symmetric FUs. Moreover, the *clustering width* of the IP and LC *input-clusters*, defined in Equation (4) and Equation (7), is reduced to the same expression if the *input-cluster* dimension of the two input vectors is the same. However, architectures featuring asymmetric multipliers, like Field Programmable Gate Arrays (FPGAs) [27], could benefit from having asymmetric *input-cluster* dimensions.

The maximum number of elements composing an *input-cluster* can be derived as the ratio between the CPU register bitwidth and the *clustering width*:

$$\text{input-cluster}_{dim} = \frac{\text{CPU}_{\text{bitwidth}}}{ClusteringWidth} \tag{8}$$

Figure 2 reports the maximum *input-cluster* dimension on 32-bit and 64-bit registers, for input bitwidths ranging from 1-bit to 16-bit. As Figure 2 shows, the *input-cluster* dimension is inversionally proportional to the input data size. Specifically, a 32-bit architecture handles from two 7-bit to six 1-bit input data concurrently for the selected kernels, while a 64-bit architecture can compute from two 15-bit to ten 1-bit elements concurrently. Thus, when the ratio between the underlying hardware architecture and the target data size is wide enough, using *binary segmentation* allows computational concurrency. Specifically, data concurrency is exploited when the *input-cluster* dimension is equal or greater than two, as multiple data are processed in parallel using a single operation. According to Figure 2, this concurrency starts to be effective for 15-bit data sizes on 64-bit architectures, and it is supported on 32-bit architectures for data ranging from 1-bit to 7-bit.

Figure 2 also shows that the number of elements composing the *input-cluster* is rarely improving the kernels memory footprint. Indeed, only 1-bit *input-clusters* can hold more elements than a conventional byte-based allocation. For this reason, creating the *input-clusters* before each computation while keeping data compressed in memory would be beneficial from a memory consumption perspective. However, such data manipulation would increase the computational cost of performing *binary segmentation*, reducing

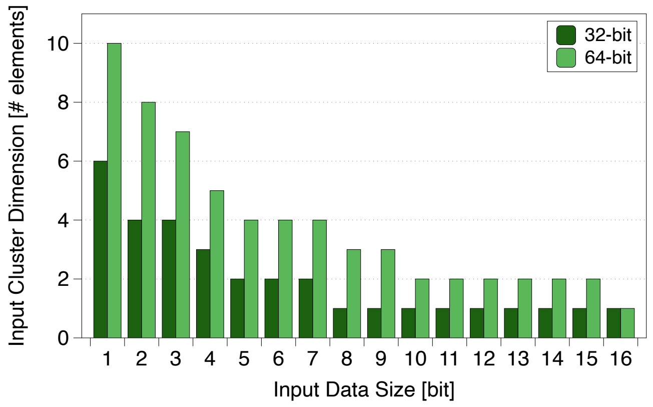


Figure 2: Maximum *input-cluster* dimension achievable on 32-bit and 64-bit architectures, for data sizes ranging from 1-bit to 16-bit. The *input-cluster* dimension is defined as the number of elements that can be packed in a single register, following the *binary segmentation* constraints.

its overall arithmetic complexity improvement. These considerations are analyzed in detail in Section 3.1 and Section 3.2. We focus our study on 64-bit architectures, as they are capable of supporting more data sizes than the 32-bit ones.

3.1 Inner Product Kernel Analysis

From Equation (2), we can notice that computing the IP of two vectors having length *n* requires *n* multiplications and *n-1* additions. As discussed in Section 2, we can perform the same computation by means of a single multiplication, as long as the *input-cluster* can hold *n* elements. We used the maximum *input-cluster* dimension for every considered data size, reported in Figure 2, to derive the arithmetic complexity decrease, defined as the ratio between the arithmetic operations (*i.e.*, multiplications and additions) needed to compute the IP kernel by using either Equation (2) or *binary segmentation*. As shown in Figure 3a, the analyzed technique can save a considerable number of arithmetic operations to compute the IP kernel. Specifically, the number of multiplications and additions required by the *binary segmentation* technique is reduced from a 3× for 15-bit computations, to a 19× for 1-bit computations.

Although this reduction can have a huge impact on linear algebra kernels computing IPs, the implementation of this technique exploiting standard ISAs leads to sub-optimal results, mostly due to their inability to efficiently support non-standard bitwidth data manipulations. Indeed, each element of the *input-cluster* needs to be converted to non-standard bitwidths (*i.e.*, the *clustering width*) to respect Equation (4). As a result, the *input-cluster* creation becomes the bottleneck of the IP kernel using *binary segmentation*. Figure 4a reports the profiling of the IP kernel execution, computed via *binary segmentation* on 64-bit architectures, for the *input-cluster* widths defined in Figure 2. We split the arithmetic operations in three main categories: data are firstly pre-processed through *Pack* instructions to create the *input-clusters*, then a *Multiply* operation performs their IP computation, whose result is filtered by the *Extract* operation.

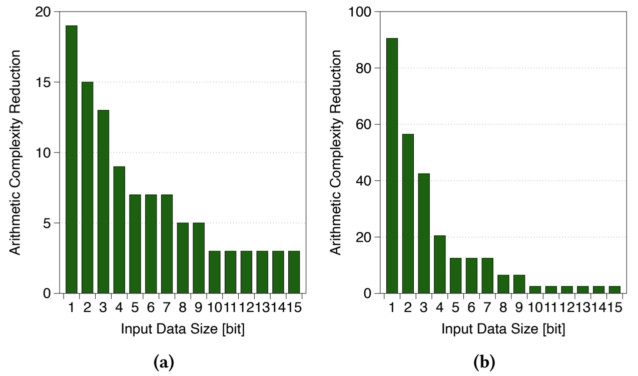
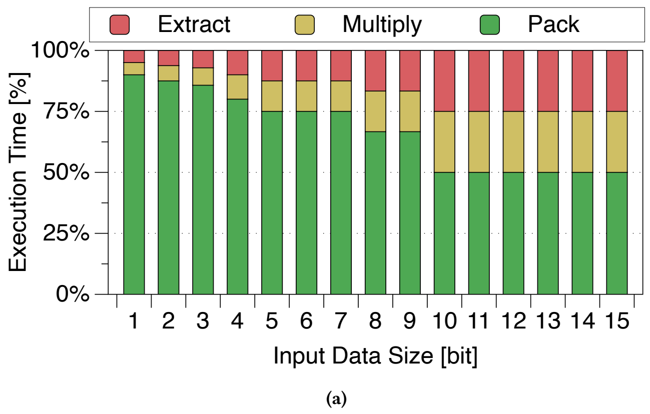


Figure 3: Arithmetic complexity reduction when computing IP (a) and LC (b) kernels on 64-bit architectures, accounting for multiplications and additions.



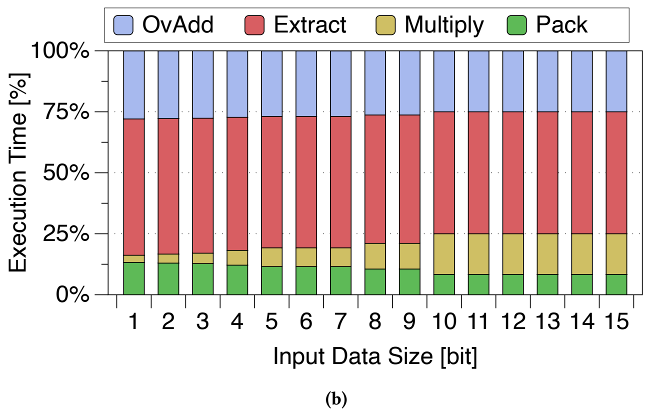


Figure 4: Amount of time spent in Pre-Processing, Processing and Post-Processing phases of the IP (a) and LC (b) kernels computed via *binary segmentation* on 64-bit architectures.

Ideally, the *Multiply* phase of Figure 4a should cover the whole execution time percentage, enabling the performance improvements of Figure 3a. However, Figure 4a shows that the number of instructions required for the actual IP computation using *binary segmentation* is minimal, while the greatest contribution is attained by the pre-processing phase, whose purpose is to create the *input-clusters*. Aiming to alleviate the data pre-processing overhead introduced in the *Pack* phase, we also implemented custom bit-manipulation instructions, namely *PACK* and *MASK*, to quickly create the *input-cluster* and extract a specific data slice from the output result. These instructions are common in ISAs bit-manipulation extensions [21]. However, as further discussed in Section 5, our evaluation reports that these bit-manipulation instructions can slightly increase the time spent in the *Multiply* phase of Figure 4a, by a factor ranging from 4.5% to 15% for 1-bit and 15-bit data, respectively. We tackle this challenge by proposing an enhanced architecture to compute the IP kernel via *binary segmentation*. As detailed in Section 4.1, we implemented *BiSon-e* to fuse both the *input-cluster* creation, the multiplication, and the output extraction into a single operation. We also exploit the data compression scheme offered by *binary segmentation* in Equation (1) to reduce the memory movements of this kernel.

3.2 Linear Convolution Kernel Analysis

Figure 3b shows the LC arithmetic reduction on 64-bit architectures, for the *input-cluster* dimensions defined in Figure 2, with data sizes ranging from 1-bit to 15-bit. From Figure 3b, it can be noted that computing a 2×2 LC among 15-bit integers induces a 2.5× arithmetic saving, while a 10×10 LC of boolean data obtains a 90.5× arithmetic reduction with respect to a standard implementation.

As the number of output elements of each LC computation, called *output-cluster*, is greater than the *input-cluster* (*i.e.*, *n+m-1*), to recover each *output-cluster*, for the *input-cluster* dimensions reported in Figure 2, it is necessary to perform two separate multiplications, computing the low and high slices of the multiplication, respectively.

Listing 1 reports the pseudo-code of the *binary segmentation*-based LC between two input vectors *v_in0* and *v_in1*, whose lengths are *m* and *n*.

```
for (i = 0; i < m/ic_dim; i++) {
  create_ic(ic0, &v_in0[i*ic_dim]);
  for (j = 0; j < n/ic_dim; j++) {
    // buffer input-cluster
    if(i == 0)
      create_ic(ic1_v[j], &v_in1[j*ic_dim]);
    // actual C1D computation
    m_out_l = ic0 * ic1_v[j];
    m_out_h = MULH(ic0, ic1_v[j]);
    // extract the output-cluster
    create_oc(oc_v, m_out_l, m_out_h);
    // accumulate using overlap-add
    overlap_add(c1d_v, oc_v);
  }
}
```

Listing 1: Pseudo-code of the LC kernel using *binary segmentation.*

The inner-most loop of Listing 1 computes the LC among the *i*-th *input-cluster* belonging to *v_in0* (*i.e.*, *ic0*) and the whole *v_in1* vector. The *create_ic* function creates both the *input-clusters*, exploiting either bitwise instructions (*i.e.*, left-shift and OR) or a set of *PACK* instructions. Since the resulting *output-cluster* is divided among the two multiplications output, the *create_oc* function extracts the result of each *ic_dim* × *ic_dim* convolution, composed of *ic_dim+ic_dim-1* elements, from the two multiplications output, creating the *oc_v* vector. Finally, the overlap-add method [22] composes the LC output vector, called *c1d_v*. Since each *output-cluster* represents a segment of *c1d_v*, the overlap-add method accumulates each segment into a given position of *c1d_v*.

From Listing 1, we can notice that the LC offers more data-reuse possibilities at the *input-cluster* level than the IP kernel, as the first inner-most loop iteration creates the *c1d_v*, that is reused till the program ends. Instead, the advantages offered by *binary segmentation* for LC are mitigated by the extra-computation required to extract *c1d_v* and to perform overlap-add. Indeed, for every inner-most loop iteration, it is required to extract *ic_dim+ic_dim-1* elements from the multiplication results, storing them into the *c1d_v* vector, and compute the element-by-element addition between *c1d_v* and a segment of *oc_v*. This overhead is reported in Figure 4b, showing the percentage of time spent on each phase of the computation. Specifically, we can note that extracting the *output-cluster* from *c1d_v* takes roughly the 50% of the total execution time, and that the overlap-add kernel takes averagely 26% of the overall time to be computed. As detailed in Section 4.2, we propose an enhanced implementation of the LC algorithm, that improves the analyzed limits by properly creating the *output-cluster*, allowing to skip the *c1d_v* extraction and to compute the overlap-add kernel exploiting *binary segmentation*.

4 BISON-E ARCHITECTURE

The architecture proposed in this paper has been built on top of the *binary segmentation* technique, presented in Section 2. Although this technique has proved its strength to optimize memory compactness and arithmetic complexity of integer linear algebra kernels, to the best of our knowledge, this is the first work that investigates it from a computer architecture perspective. As Section 3 shows, exploiting *binary segmentation* naïvely leads to practical inefficiencies, mainly due to the standard ISAs and architectures lack of support for non-standard data sizes bit-manipulation operations. In this Section, we tackle this problem by presenting *BiSon-e*, a lightweight architecture that enables exploiting *binary segmentation* on resource-constrained devices. *BiSon-e* extends general-purpose ISAs with instructions facilitating narrow-integer computations by leveraging the extremely area-efficient *binary segmentation* idea. The insight behind *BiSon-e* is to fill the gap between application-specific accelerators and SIMD/Vector units for die-area sensitive edge computing use-cases. On the one hand, as detailed in Section 5, *BiSon-e* is comparably more efficient than a high-performance Vector Processing Unit (VPU) for narrow integer computations, while featuring 600× less area overhead. On the other hand, *BiSon-e* features more flexibility than an application-specific accelerator, as it can be used for any kernel exploiting SIMD-style narrow computations. *BiSon-e* is efficient for modern edge computing systems for the following reasons. It leverages existing FUs on scalar architectures to provide a more flexible integer compute fabric than SIMD architectures. Its flexibility implies a better fine-tuning of the data sizes

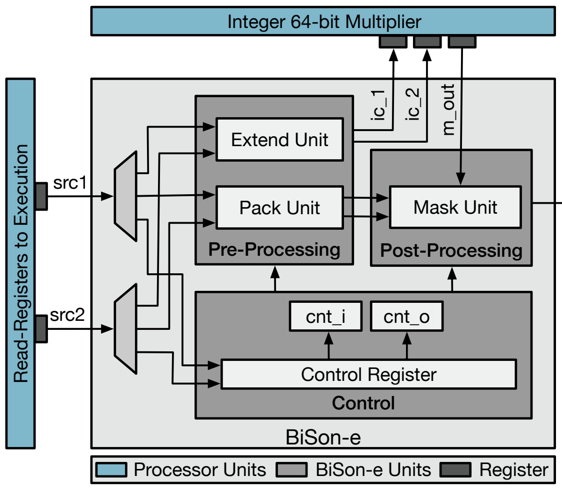


Figure 5: *BiSon-e* block diagram.

involved in the computation than standard narrow-SIMD units, that rarely support arithmetic and memory instructions for data formats below 8-bits, and typically neither cover all the possible data size granularities nor support mixed-precision computations. As an example, the current RISC-V vector extension [41] has deprecated its support for narrow-SIMD computations (*i.e.*, Zvediv), whose initial specifications only accounted for 8-bit, 4-bit, 2-bit, and 1-bit data types. As opposed to standard SIMD units, *BiSon-e* allows every data size discussed in Section 3 to be kept compressed in memory, and computed in a SIMD fashion, without incurring data manipulation related area overheads. *BiSon-e* also features mixed-precision computation support by design, as the *clustering widths* of Equation (4) and Equation (7) already account for different data sizes between the data sources. Thus, *BiSon-e* is capable of computing compressed data and performing flexible SIMD-style computations, whose width is proportional to the data size of every operation operand, without associated overheads. Moreover, implementing *BiSon-e*, whose key novelty relies on hardware reutilization, does not require any additional datapath, or a separate Register File (RF) or FU, leading to a negligible area and power overheads.

As detailed in Section 3.1 and Section 3.2, the main bottleneck of implementing *binary segmentation* on standard architectures is either represented by the pre-processing phase, responsible of the *input-clusters* creation, or by the post-processing one, whose main purpose is to extract the output from the segmented data and perform accumulations using the overlap-add method. On the contrary, *binary segmentation* does not affect the computation complexity, as it can rely on standard arithmetic units (*e.g.*, integer multipliers), whose datapaths and implementations are already implemented in processors supporting integer computations. Thus, the principal aim of the proposed architecture is to efficiently cluster data before the multiplication, and to optimize the data extraction on the multiplier output side. *BiSon-e*, whose main functional blocks are depicted in Figure 5, tackles these problems by means of two stages, called *pre-processing* and *post-processing*. The *pre-processing* stage functionality is twofold. Its *extend-unit* converts *ic_dim* elements

Table 1: *BiSon-e* Control parameters list

	Input Data		Input Cluster		Iterations	
	Bitwidth	N.Elements	Bitwidth	N.Elements	Pre-Proc	Post-Proc
Extend	8	8	21	3	3	1
	7	9	16	4	3	1
	6	10	16	4	3	1
	5	12	16	4	3	1
	4	16	12	5	4	1
	3	21	9	7	3	1
	2	32	8	8	4	1
	1	64	6	10	7	1
Pack	8	8	1	64	1	8
	8	8	2	32	1	4
	8	8	4	16	1	2

Table 2: Overview of the BiSon-e custom instructions

Instruction	Description
bs.set	control registers configuration
bs.pack	pack *n* elements from Rs1 and Rs2
bs.ip	returns the Inner Product
bs.lc.l	returns the lower slice of the Linear Convolution
bs.lc.h	returns the higher slice of the Linear Convolution

belonging to *src1* and *src2* into the *input-clusters*, and forwards them to the processor multiplier to perform the actual computation. The number of elements to be clustered, as well as the *src1* and *src2* bitwidths, are specified in the *control register*, whose configurations[3] are defined in the *Extend* part of Table 1, and programmed through the *bs.set* custom instruction reported in Table 2.

As an example, when configuring the *control register* with the parameters listed in the first row of Table 1, the *extend-unit* expects eight 8-bit elements in both *src1* and *src2*, and creates the *input-clusters* composed of three 21-bit elements. The *Pre-Proc* parameter, defined in Table 1, is used to cyclically offset the *src1* and *src2* registers content, depending on the *cnt_i* value, spanning from 0 to *Pre-Proc* - 1. Indeed, as expressed in the first row of Table 1, a single 64-bit register containing eight elements requires three iterations (i.e., clock cycles) to be completely processed, each one computing three elements of the input registers. The *input-clusters* are forwarded to the multiplier through the *ic_1* and *ic_2* output busses. Then, the multiplication result is processed by the *mask-unit*, that composes the final result depending on the instruction opcode. Specifically, the *mask-unit* extracts data in the range expressed in Equation (5) if a *bs.ip* instruction is decoded, while it outputs either the lower or the higher part of the convolution in case of a *bs.lc.l* or a *bs.lc.h* instruction.

To speed-up the data compression phase, *BiSon-e* implements the *pack-unit*, which compress its input data into their actual data sizes. The *pack-unit* functionality is inferred by the *bs.pack* instruction, listed in Table 2. The source operands of this instruction contain the input register to be compressed (*i.e.*, *src1*) and the final compressed register (*i.e.*, *src2*).

[3]For space reasons, we omit part of the possible configurations allowed by *BiSon-e*, including the one concerning mixed-precision computations.

Each instruction call converts the *src1* elements into the target output bitwidth, and forwards both the results and *src2* to the *mask-unit* to merge them. As an example, the first row of the Table 1 *Pack* block could convert eight 8-bit input elements to a single register, composed of sixty-four 1-bit elements. To do that, *BiSon-e* only requires eight iterations, more precisely, eight *bs.pack* instructions. On every iteration, the *pack-unit* converts the eight elements of *src1* into 1-bit format, while the *mask-unit* concatenates the created data slice into *src2*. The concatenation offset used by the *mask-unit* on every iteration depends on the *cnt_o* value, ranging from 0 to *Post-Proc* - 1. Therefore, the *post-processing* stage either acts as a filter to extract the meaningful slice of data from the multiplier output, or it is used to compress data in case of *bs.pack* instructions. Its behaviour depends on the decoded instruction, as well as on the values set in the *control register*.

As detailed in Section 5, the proposed solution introduces a minimal impact on power and area consumption, as the actual computation is performed by the existing processor multiplier. Moreover, we designed the proposed architecture to avoid a latency overhead increase. Indeed, as in the case of standard multiplication operations on the considered target processor, both *bs.ip* and *bs.lc* instructions feature a latency of three clock cycles. In the first clock cycle, data are read from the RF, processed by the *pre-processing* stage, and forwarded to the multiplier input registers. The second clock cycle performs the multiplication, while the third one stores the result into the RF, after being properly extracted by the *post-processing* stage. As Table 1 reports, the number of *Pre-Proc* iterations required by all the *Extend* configurations is greater than one. As detailed in Section 4.1, this implies that consecutive *bs.ip* or *bs.lc* instructions share the source operands, allowing to pipeline the execution of multiple iterations.

From Table 2, we can also note that *BiSon-e* requires a minimal set of simple instructions. As today's compilers can optimize, through vectorization, computations such as IP and LC, the proposed methodology can be exploited by a compiler, as soon as the target language supports sub-byte data types. Alternatively, as performed in this work and as a current trend in many fields like deep learning, users can define high-level libraries, optimized with the low-level instructions of Table 2.

4.1 Enhanced Inner Product Computation

The pseudo-code of the IP computation exploiting *BiSon-e* is reported in Listing 2.

Firstly, the *control register* is configured according to Table 1. Once the parameters have been loaded into the *control register*, the main loop computes the IP between two vectors having length *v_dim*. Note that the number of iterations required to perform the computation is given by the ratio between *v_dim* and the number of elements packed in the register (*i.e.*, *el_in_reg*). Indeed, every loop iteration computes the IP of *el_in_reg* elements belonging to *v_in0* and *v_in1*, and each *bs.ip* instruction processes *input-cluster* elements. For example, considering the *BiSon-e* configuration for 1-bit input data in Table 1, a single iteration of the loop contains seven *bs.ip* instructions, each tackling ten elements of *v_in0* and *v_in1*. The partial IP is then further accumulated into the final result.

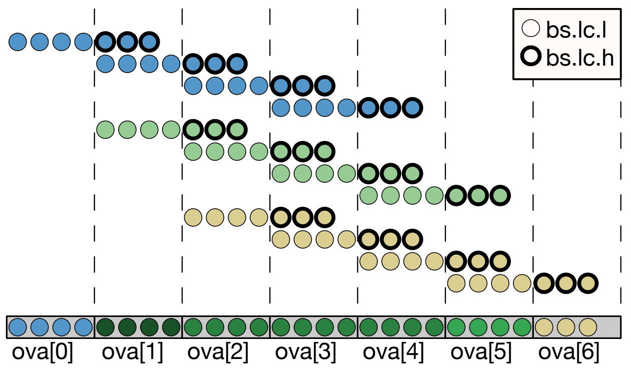


Figure 6: Improved overlap-add using *BiSon-e*. Different colors represent different outer loop iterations.

```
// Configure Control Register
bs.set(config.params);
for (i = 0; i < v_dim/el_in_reg; i++){
  for (j = 0; j < Pre-Proc; j++){
    // Compute Partial Inner Products
    bs.ip(ip_tmp, v_in0[i], v_in1[i]);
    // Accumulate Final Inner Product
    ip += ip_tmp
  }
}
```

Listing 2: Pseudo-code of the IP kernel using *BiSon-e*.

As Listing 2 shows, the IP computation exploiting *BiSon-e* has several advantages with respect to both the naïve algorithm and the one exploiting *binary segmentation* on standard CPUs. Firstly, it allows to fully take advantage of the *binary segmentation* benefits of reducing the memory footprint and the computation arithmetic complexity. Indeed, *BiSon-e* supports compressed data as inputs, that do not require extra manipulation to be extracted into a standard bitwidth before performing the computation. Secondly, it reduces the number of iterations required for the computation by a factor that scales from eight to sixty-four, for 8-bit and 1-bit data sizes, respectively. Finally, as every *bs.ip* instruction belonging to the same loop iteration deals with the same input registers, its execution can be pipelined at the hardware level, allowing reducing the overall computation latency. As Figure 5 shows, we used the input and output registers of the two-stage multiplier to pipeline the instructions execution, allowing to execute up to two instructions concurrently.

4.2 Fused Overlap-Add

As detailed in Section 3.2, the main bottleneck of performing the LC kernel with *binary segmentation* relates with the *post-processing* phase, since the result of every convolution has to be extracted and accumulated into the output vector. The proposed solution exploits *BiSon-e* and *binary segmentation* to perform fused overlap-add accumulations. The *bs.lc.l* and *bs.lc.h* custom instructions compute and extract the whole *ic_dim+ic_dim-1* result of the convolution. As in the *bs.ip* case, the two instructions share their inputs. However, *bs.lc.l* returns the lower *ic_dim* elements of the result, while *bs.lc.h* returns the higher *ic_dim-1* elements. Thus, differently from the LC reference implementation of Listing 1, the LC result does not require to be further manipulated before the overlap-add phase, as it is possible to perform overlap-add via *binary segmentation* without extracting the data. Indeed, the configurations of Table 1 allow for extra computation in the segmented data format. As an example, considering the first row of Table 1, we can notice that the *input-cluster* bitwidth (*i.e.*, 21 bits) is greater than the *clustering width* resulting from Equation (7), which is equal to 18 bits for *input-clusters* of three elements and 8-bit data sizes. We accounted for the remaining three bits to compute overlap-add via *binary segmentation*. As an example, Figure 6 illustrates the fused overlapp-add of a LC performed on two 16×12 input vectors having 4-bit data size, and with *BiSon-e* configured to produce *input-clusters* with four 4-bit elements. With that configuration, after every *bs.lc.l* and *bs.lc.h* instructions sequence, the *post-processing* stage outputs seven elements divided into two registers. Specifically, the first register includes four 16-bit elements corresponding to the outcome of the *bs.lc.l* instruction, while the second register contains three 16-bit elements created by the *bs.lc.h* instruction. As can be seen from Figure 6, the overlap-add can be reduced to two additions per iteration, the first adding the *bs.lc.l* result into the current *ova* register (*i.e.*, the current output register), and the second adding the *bs.lc.h* result into the next *ova* register. The twenty-seven elements are accumulated via *binary segmentation*, and stored in seven *ova* registers. Thus, the overlap-add phase can be computed without pre-extracting the result of every multiplication, and computing only twenty-four additions, instead of the eighty-four needed by the standard overlap-add implementation to create the twenty-seven elements final result.

5 EXPERIMENTAL EVALUATION

This Section evaluates *BiSon-e* in terms of performance, energy efficiency, and area consumption. We also perform a comparison with an embedded VPU, integrated in the target SoC, to show the efficiency of *BiSon-e* when compared with a more conventional high-performance embedded architecture.

5.1 Experimental Setup

Performance numbers have been measured using the gem5 simulator, configured with a 5-stage, single-issue in-order pipeline, supporting the 64-bit RV64IM RISC-V ISA. The cache hierarchy comprises 4-way 16 KB L1D and L1I caches, having 2-cycle access latency, and a unified 8-way 64 KB L2 cache with 20-cycle access latency. Moreover, the processor is equipped with a VPU processor, exploiting 2-lanes and a maximum vector length of 4096 bits. We use the gem5 VPU proposed in [39], implementing the RISC-V-V v0.7.1 vector extension [41] to run the vectorized implementation of the workloads implemented with vector intrinsics instructions. We extended the RISC-V GNU Compiler Toolchain [40] with the custom instructions of Table 2 to support *binary segmentation* and *BiSon-e*, integrated in C/C++ implementations of the benchmarks through intrinsic instructions. We used the MacPat simulator [28] to extract energy efficiency metrics.

5.2 Workload Description

Convolutional Neural Networks. CNNs represent one of the most widespread models used in computer vision deep learning algorithms. Being a CNN composed of a variety of kernels, the most time-consuming ones are represented by the Convolutional and the Fully-Connected layers. To optimize the performance of these layers through customized libraries [31, 46], data is usually reshaped to convert them in linear algebra computations [13, 47]. Moreover, to further cope with the runtime requirements of CNNs *inference*, one of the most attractive solutions is quantization [23, 30, 33], a technique that converts CNN data and parameters from floating-point to integer formats, whose size typically ranges from 8-bit down to 1-bit [38] featuring negligible accuracy losses when compared to the floating-point baseline [8, 15, 29, 34, 49]. Investigating the optimization of QCNN computations is an important research topic, as they represent a critical application for CPUs, Graphics Processing Units (GPUs), and hardware accelerators [1, 24, 45]. We leverage on *BiSon-e* to improve the efficiency of the AlexNet [26] and VGG-16 [44] QCNNs. For these benchmarks, we exploit the IP kernel computed on *BiSon-e*, improving its execution by reducing the overall number of required multiplications and additions. To test the performance scalability of *BiSon-e*, we explored different data sizes of both data and weights of the selected networks.

Approximate String Matching. Research in pattern matching applies to many important use cases, ranging from biological sequence alignments and genome pre-alignment filters [5, 6, 9], to web search engines and data compression. One of the principal fields of pattern matching, called *string matching*, verifies if a sequence of characters belonging to a given alphabet (*i.e.*, the pattern) matches into a reference string (*i.e.*, the text). The *string matching* problem can be solved by following many different methods [4]. One widespread algorithm breaks up both the text and the pattern into boolean vectors, one for every letter of the alphabet, and computes boolean LCs among each vector pair. Each LC result is then accumulated into the output vector, whose elements identify the number of mismatches among the pattern and the text, starting from each text position. This solution, firstly proposed in [16], is an extension of the Knuth-Morris-Pratt (KMP) algorithm [32], and represents a valid candidate to solve the problem of *approximate string matching* with don’t care conditions. Although this solution has proved its efficiency on several works in the literature [7, 17], its performance can be further improved by applying data compression, as well as by decreasing the arithmetic complexity of boolean LCs. Both of these optimizations can be efficiently achieved by the enhanced LC computation enabled by *BiSon-e*.

5.3 Performance

As a first performance analysis, we study the IP and the LC kernels exploiting *binary segmentation*, and we compare their run-time with their naïve implementations of Equation (2) and Equation (6). In Figure 7, we benchmark three implementations using *binary segmentation*: one exploiting the standard 64-bit integer RISC-V ISA (*i.e.*, RV64IM), one featuring bit-manipulation instructions, and one computing with *BiSon-e*.

For the IP, the speed-ups obtained with respect to the naïve kernel are reported in Figure 7a, and ranges from 1.3× to 1.5×,

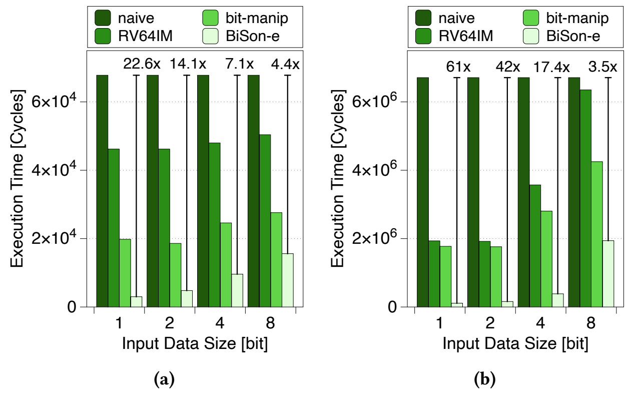


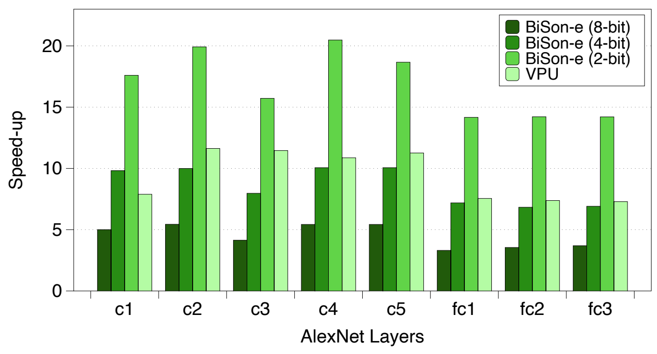
Figure 7: Execution time of IP (a) and LC (b) kernels using naïve implementation and binary segmentation exploiting either standard ISA, bit-manipulation instructions, and *BiSon-e*.

and from 2.5× to 3.4×, for the *binary segmentation* implementation featuring standard and bit-manipulation instructions, respectively. These results are in line with our analysis in Figure 3a, as the instructions needed to pack and extract data typically exploit a smaller latency than the multiplication one, and because of the smaller number of loop iterations required by the *binary segmentation* implementations. Figure 7a also shows that, in contrast to Figure 3a, the achieved speed-up does not scale with the number of elements composing the *input-cluster*. As analyzed in Figure 4a, this behavior is due to the *input-cluster* creation complexity, which grows with the number of elements composing the cluster. While the speed-ups obtained in Figure 7a highly differ from the theoretical performance gain that *binary segmentation* could exploit, the implementation exploiting *BiSon-e* leads to a 4.4×, 7.1×, 14.1× and 22.6× speed-up with respect to the reference implementation of Figure 7a, for data sizes of 8-bit, 4-bit, 2-bit, and 1-bit, accordingly. Note that the performance of *BiSon-e* are comparable with the maximum theoretical improvement allowed by *binary segmentation* for the IP kernel, reported in Figure 3a.

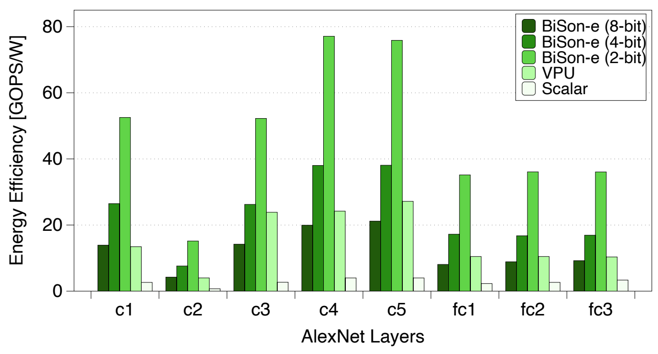
The experimental evaluation of the LC kernel is reported in Figure 7b. The RV64IM-based implementation improves the reference by a factor ranging from 1.1× to 3.5×, while the implementation exploiting bit-manipulation instructions reaches from 1.6× to 3.8× with respect to the reference. When compared to Figure 7b, our implementation leveraging *BiSon-e* features a speed-up of 3.5×, 17.4×, 42× and 61× for input bitwidth of 8-, 4-, 2- and 1-bit, respectively, proving that the proposed architecture can significantly improve the naïve *binary segmentation* computation reported in Figure 7b.

We implemented three workloads belonging to two different application classes, namely deep learning and approximate string matching.

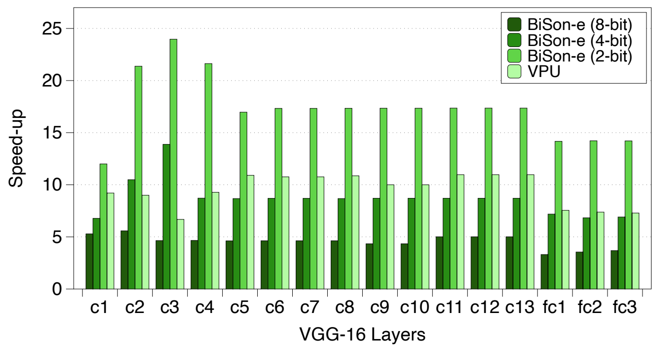
For the deep learning benchmark analysis, we focus our evaluation on the Convolutional and Fully-Connected layers of both the AlexNet and the VGG-16 CNNs, for input and weights data sizes ranging from 8-bit to 2-bit. All the kernels (*i.e.*, scalar, vector, and featuring *BiSon-e*) have been implemented using img2col, reshaping



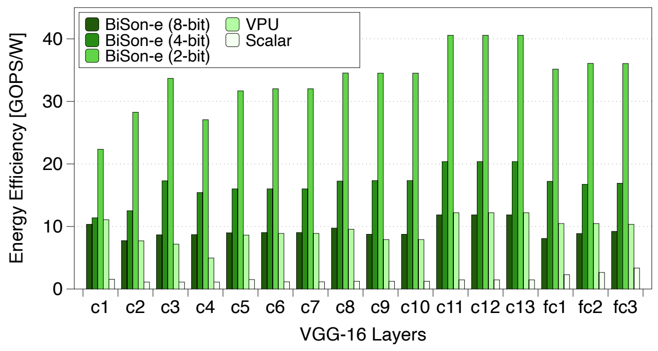
(a)



(b)



(c)



(d)

Figure 8: Speed-ups (a, c), and energy efficiency (b, d) of the AlexNet and the VGG-16 CNNs with respect to the scalar implementation, exploiting either *BiSon-e* or the VPU.

the Convolutional layers as blocked matrix-matrix multiplications, and the Fully-Connected layers as matrix-vector multiplications. The obtained results, in terms of performance and energy efficiency, are summarized in Figure 8. On the performance side, Figure 8a reports the speed-ups of the vectorized and the *BiSon-e* implementations of the AlexNet CNN, with respect to the scalar reference. As Figure 8a shows, the *BiSon-e* implementation performance scales with the decrease of the input data size. Specifically, the proposed solution runs up to 5.4×, 10.1×, and 20.5× faster than the scalar implementation for Convolutional layers, and up to 3.7×, 7.2×, and 14.2× for the Fully-Conected layers, for 8-, 4- and 2-bit data sizes, respectively. Averagely, when compared with the VPU implementation, *BiSon-e* exhibits comparable performance on the 4-bit test, and outperforms it on the 2-bit test by a factor of 1.9×. *BiSon-e* shows a 1.8× higher run-time than the VPU only for the 8-bit test. However, as Figure 8b illustrates, the 8-bit AlexNet performed with *BiSon-e* shows comparable energy efficiency with respect to the vectorized counterparts, and exhibits better efficiency in the 4-bit and 2-bit layers, by a factor that ranges from 1.5× to 3.1×.

Similarly, Figure 8c shows the performance improvement of *BiSon-e* and the VPU of the VGG-16 network, with respect to the scalar baseline. Specifically, a back-to-back execution of the network performed with *BiSon-e* peaks a 4.7×, 9.1×, and 18.5× with respect to the scalar implementation, for 8-, 4-, and 2-bit computations, also showing comparable and better performance than the VPU, by a factor up to 1.9× for 2-bit data sizes. In terms of energy efficiency, as detailed in Figure 8d, the VGG-16 network is computed with *BiSon-e* gains averagely 1.1×, 1.8×, and 3.6× with respect to the VPU implementation, for 8-, 4-, and 2-bit computations. For both the AlexNet and the VGG-16 networks, the performance scalability of *BiSon-e* is guaranteed by the increasing number of operations performed concurrently, as well as by the compressed input format on both data and weights, which allows decreasing the overall memory transfers.

Concerning the approximate string matching workload, we consider a pattern of 256 characters, a text whose length ranges from 4K to 128K characters, and an alphabet of 4 (*e.g.*, A, T, G, C) and 256 letters. Figure 9 reports the speed-ups of *BiSon-e* and the VPU with respect to the scalar reference. *BiSon-e* outperforms both the scalar and the VPU implementations for all the considered datasets. Concerning the 4-letters alphabet benchmark, *BiSon-e* gains from 21.8× to 51.6×, and from 1.3× to 3.2× execution time with respect to the scalar and the VPU implementations. We obtain similar results for the 256-letters alphabet benchmark, where *BiSon-e* outperforms both the scalar and the VPU execution time by a factor ranging

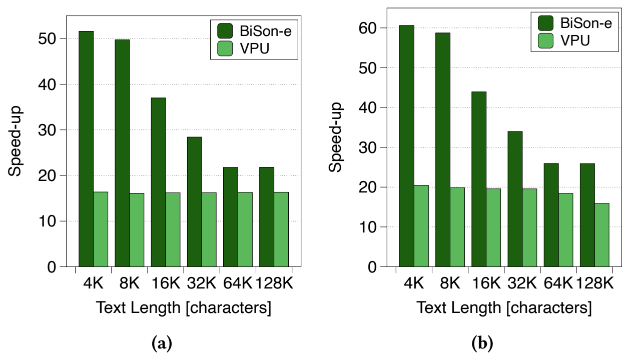


Figure 9: Approximate string matching kernel speed-up exploating either *BiSon-e* (dark-green bars) or the VPU (light-green bars) with respect to the scalar implementation, featuring a 4 (a) and a 256 (b) letters alphabet.

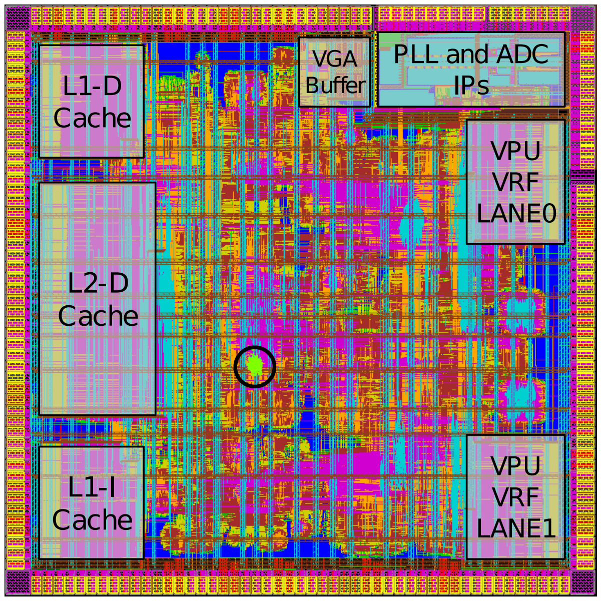


Figure 10: Layout of the DRAC SoC including *BiSon-e*, highlighted in green and circled in black, for the 65nm technology.

from 25.9× to 60.6×, and from 1.4× to 3×, respectively. Furthermore, for the approximate string matching benchmarks of Figure 9, *BiSon-e* gains an average energy improvement of 40× and 5× when compared to the scalar and the VPU implementations.

5.4 Area and Power Analysis

We integrated *BiSon-e* into the DRAC SoC design [2], using the Cadence tool flow (Genus/Innovus), to obtain the layout and main performance metrics of the overall microarchitecture. We implemented the design in two different technologies, namely TSMC 65nm bulk CMOS and GlobalFoundries 22 nm FDSOI. The 65nm layout, reported in Figure 10, is ready for production, and includes the SoC along with the VPU, peripheral controllers, a PLL, an ADC, and the IO pad-ring. The design employs standard- and low-threshold cells, and the synthesis and P&R target frequency is set to 600 MHz. The 22nm layout includes the SoC along with the VPU and peripheral controllers, as an IP block ready to be connected to a PLL and an IO pad-ring. The design employs 8-track standard cells without using body-biasing, with a target frequency of 1GHz. For both implementations, we analyzed the physical impact of *BiSon-e* incorporation on area, timing and power figures, referring to the above-defined timing constraints. Regarding the timing performance, target constraints are met in the typical corner, and it has been demonstrated that *BiSon-e* does not introduce new critical paths in the processor datapath. Regarding area and power consumption, results are summarized in Table 3. In both technologies, the area overhead of *BiSon-e* in the whole layout is below 0.07%, and the contribution to the total power consumption is lower than 0.04%. The cell count of *BiSon-e* is 1210 in the 65 nm library, while it is 1081 in the 22 nm library. For comparison with *Bison-e*, we implemented and synthesized a narrow-SIMD unit in 22nm, capable of computing 8-,4-,2-,1-bit data on a 64-bit datapath, and our evaluation reported a 10× area increase with respect to *Bison-e*, whose key novelty relies on reutilizing hardware, featuring low area-overhead and high flexibility.

Table 3: Area and Power Consumption

Component	Area [mm2]		Total Power [mW]	
	65nm	22nm	65nm	22nm
Scalar Core	4.167	0.383	1419	283.4
VPU	2.277	0.236	1757	309.3
BiSon-e	0.003704	0.000419	1.089	0.236

6 RELATED WORK

Enhanced Processing Units. Although, to the best of our knowledge, this is the first work investigating the application of *binary segmentation* on computer architecture, several works have analyzed the reduction of arithmetic complexity by packing multiple computations in a single arithmetic operation. [18], exploits the Xilinx FPGA DSP48E2 slices to pack two 8-bit multiplications, both sharing one of the multiplicands, into a single DSP slice, achieving a 1.75× speed-up compared to a naïve multiplication on the same device. The same approach has been improved in [11], where the authors propose an enhanced DSP slice architecture able to compute four 9-bit concurrent multiplication with 0.6% area overhead.

Low-Area Application Specific Accelerators. Among the works investigating the optimization of narrow integer computing on edge processors targeting CNNs, [19] proposes SIMD Multiply-Accumulates (MACs) units and bit manipulation instructions to improve the computation of QCNNs on a multi-core embedded processor. However, their architecture performance does not scale with the input data size decrease, as their SIMD units are only optimized for 8-bit computations, and require additional instructions for 4-bit and 2-bit data computations. The authors tackled this limitation in [20], proposing dot product units and custom RISC-V instructions to improve the performance of byte and sub-byte computations, reducing the runtime of a small Convolutional layer by a factor of 5.3× and 8.9× with respect to their baseline, for 4-bit and 2-bit data sizes. We modeled in gem5 the CPU architecture described in [20], and we integrate it with the proposed architecture. The results show that our solution is 4× more efficient in terms of area, and 3.14×, 3.19×, and 2.53× more performant in terms of Performance-Per-Unit-Area[Cycles/Area] for 8-bit, 4-bit, and 2-bit data sizes. The work in [35] proposes SIMD instructions and custom FUs to accelerate CNN Convolutional layers with narrow mixed-precision data types, on a RISC-V based edge processor. [35] runs 1.3×, 1.1×, and 1.4× faster than *BiSon-e* for 8-,4-, and 2-bit data types, while occupying 11.6× its area. Thus, *BiSon-e* outperforms [35] by 8.9×, 11.2×, and 8.4× for 8-,4-, and 2-bit computations in terms of Performance-Per-Unit-Area, while natively supports mixed-precision computation, as detailed in Equation (4) and Section 4. Moreover, while [19, 20, 35] proposes accelerators that are tied to the deep learning domain, *BiSon-e*, as detailed in Section 2, can be exploited for a wide range of computational kernels.

High-Performance Application Specific Accelerators. The methodology proposed in this work through *BiSon-e* features even more flexibility than high-performance application-specific accelerators like [14, 48]. For example, [14], represents a state-of-the-art DNN accelerator for mobile devices, featuring 192 processing elements and line buffers for a total area of 36mm2 on the TSMC 65nm technology node. However, the architecture proposed in [14] only supports computations based on 8-bit data and weights. Certainly, [14] provides better performance than a single *BiSon-e* instance featuring a single multiplier and integrated on an off-the-shelf processor, mainly because of its tailored design and its demanding area, roughly 2227× larger than *BiSon-e* in 65nm. Indeed, [14] outperforms *BiSon-e* by a factor of 39.9×, 21.4×, and 10.9× in terms of Performance-Per-Unit-Area, for the computation of AlexNet on 8-, 4-, and 2-bit data types. Considering the perceived 100× efficiency gap between CNN accelerators based on ASICs and CPUs [50], this work goes toward closing this gap. Moreover, the same area budget of [14], would enable the integration of up to 8 scalar cores, each featuring one *BiSon-e* unit, or up to 4 scalar cores, featuring one VPU and one *BiSon-e* unit. As a result, a specific solution can be chosen depending on the latency, throughput, area, and power constraints of the target processor, as well as on the variety of workloads it has to execute. Moreover, the proposed methodology can be exploited to design application-specific accelerators based on the *binary segmentation*, and *BiSon-e* can be integrated on SIMD or Vector processors to scale their performance on narrow-precision computations, exploiting the same benefits in terms of area, reduced memory footprint, and flexibility on the employed data types, with a minimal ISA extension, and without designing application-specific and area-consuming FUs.

7 CONCLUSIONS

This work proposes a novel methodology to accelerate linear algebra kernels based on narrow integers. We present *BiSon-e*, a lightweight and high-performance accelerator targeting narrow integer linear algebra computing on resource-constrained processors. We built the proposed solution upon the *binary segmentation* mathematical technique, which reduces both the memory footprint and the arithmetic complexity of integer linear algebra computations, and whose efficiency is proportional to the ratio between the target data sizes and the architecture bitwidth. We perform a detailed DSE of *binary segmentation* on 64-bit architecture, highlighting its strengths and pitfalls. Then, we propose *BiSon-e* to overcome the main limitations of the analyzed technique on standard CPU architectures and ISAs, and showing that the proposed engine can run important linear algebra kernels such as IP and LC from 3.5× to 61× faster than a scalar RISC-V edge processor. We benchmark the proposed solution on three algorithms belonging to two key edge computing application domains, namely deep learning and approximate string matching. We integrate the proposed architecture into a complete SoC, based on RISC-V, past the P&R step. Our analysis shows that *BiSon-e* considerably enhances the performance of narrow integer computations, introducing a negligible 0.07% impact on the overall processor area. Specifically, our experimental evaluation shows that *BiSon-e* outperforms the scalar processor from 4.7× to 19.3× on the AlexNet and the VGG-16 CNN benchmarks in terms of execution time, and shows comparable or higher energy efficiency than a VPU on the same tasks. Moreover, *BiSon-e* on approximate string matching tasks reaches execution speed-up from 1.4× to 3× when compared to the VPU implementation, showing an avarage 5× improvement in terms of energy efficiency.

ACKNOWLEDGEMENTS

This research was supported by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of the total cost eligible, under the DRAC project [001-P-001723], and from the Spanish State Research Agency - Ministry of Science and Innovation (contract PID2019-107255GB). This research was also supported by the grant PRE2020-095272 funded by MCIN/AEI/ 10.13039/501100011033 and, by “ESF Investing in your future”

REFERENCES

[1] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, and François Berry. 2018. Accelerating CNN inference on FPGAs: A Survey. *arXiv e-prints*, Article arXiv:1806.01683 (May 2018), arXiv:1806.01683 pages. arXiv:1806.01683 [cs.DC]

[2] J. Abella, C. Bulla, G. Cabo, F. J. Cazorla, A. Cristal, M. Doblas, R. Figueras, A. González, C. Hernández, C. Hernández, V. Jiménez, L. Kosmidis, V. Kostalabros, R. Langarita, N. Leyva, G. López-Paradís, J. Marimon, R. Martínez, J. Mendoza, F. Moll, M. Moretó, J. Pavón, C. Ramírez, M. A. Ramírez, C. Rojas, A. Rubio, A. Ruiz, N. Sonmez, V. Soria, L. Terés, O. Unsal, M. Valero, I. Vargas, L. Villa, and C. Ramíìez. 2020. An Academic RISC-V Silicon Implementation Based on Open-Source Components. In *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*. 1–6. https://doi.org/10.1109/DCIS51330.2020.9268664

[3] A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan. 2019. DLFloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 92–95. https://doi.org/10.1109/ARITH.2019.00023

[4] Koloud Al-Khamaiseh and Shadi ALShagarin. 2014. A Survey of String Matching Algorithms. *International Journal of Engineering Research and Applications* 4 (08 2014), 144–156.

[5] Mohammed Alser, Hasan Hassan, Akash Kumar, Onur Mutlu, and Can Alkan. 2019. Shouji: A fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics (Oxford, England)* 35 (11 2019), 4255–4263. https://doi.org/10.1093/bioinformatics/btz234

[6] Mohammed Alser, Taha-Michael Shahroodi, Juan Gómez-Luna, Can Alkan, and Onur Mutlu. 2020. SneakySnake: a fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs. *Bioinformatics* 36 (12 2020). https://doi.org/10.1093/bioinformatics/btaa1015

[7] Amihood Amir, Avivit Levy, and Liron Reuveni. 2008. The Practical Efficiency of Convolutions in Pattern Matching Algorithms. *Fundam. Inf.* 84, 1 (jan 2008), 1–15.

[8] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. 2018. Post-training 4-bit quantization of convolution networks for rapid-deployment. *arXiv e-prints*, Article arXiv:1810.05723 (Oct. 2018), arXiv:1810.05723 pages. arXiv:1810.05723 [cs.CV]

[9] Zülal Bingöl, Mohammed Alser, Onur Mutlu, Ozcan Ozturk, and Can Alkan. 2021. GateKeeper-GPU: Fast and Accurate Pre-Alignment Filtering in Short Read Mapping. 209–209. https://doi.org/10.1109/IPDPSW52791.2021.00039

[10] Dario Bini and Victor Pan. 1986. Polynomial division and its computational complexity. *Journal of Complexity* 2, 3 (1986), 179 – 203. https://doi.org/10.1016/0885-064X(86)90001-4

[11] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2018. Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs. 35–357. https://doi.org/10.1109/FPL.2018.00014

[12] Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. 1989. GCDHEU: Heuristic polynomial GCD algorithm based on integer GCD computation. *Journal of Symbolic Computation* 7, 1 (1989), 31 – 48. https://doi.org/10.1016/S0747-7171(89)80004-5

[13] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Université de Rennes 1, Suvisoft, La Baule (France). https://hal.inria.fr/inria-00112631 http://www.suvisoft.com.

[14] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9 (06 2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232

[15] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. 2019. Low-bit Quantization of Neural Networks for Efficient Inference. 3009–3018. https://doi.org/10.1109/ICCVW.2019.00363

[16] Michael J Fischer and Michael S Paterson. 1974. String matching and other products. In *Complexity of Computation, RM Karp (editor), SIAM-AMS Proceedings*, Vol. 7. 113–125.

[17] Kimmo Fredriksson and Szymon Grabowski. 2009. Fast Convolutions and Their Applications in Approximate String Matching. 254–265. https://doi.org/10.1007/978-3-642-10217-2_26

[18] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. 2016. Deep learning with int8 optimization on xilinx devices. *White Paper* (2016).

[19] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. 2020. PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378 (02 2020), 20190155. https://doi.org/10.1098/rsta.2019.0155

[20] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini. 2020. XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 186–191. https://doi.org/10.23919/DATE48585.2020.9116529

[21] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. 2017. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2700–2713. https://doi.org/10.1109/TVLSI.2017.2654506

[22] D. Griffin and Jae Lim. 1984. Signal estimation from modified short-time Fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 32, 2 (1984), 236–243. https://doi.org/10.1109/TASSP.1984.1164317

[23] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *arXiv e-prints*, Article arXiv:1609.07061 (Sept. 2016), arXiv:1609.07061 pages. arXiv:1609.07061 [cs.NE]

[24] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. 2019. A Survey of the Recent Architectures of Deep Convolutional Neural Networks. *arXiv e-prints*, Article arXiv:1901.06032 (Jan. 2019), arXiv:1901.06032 pages. arXiv:1901.06032 [cs.CV]

[25] Wazir Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. 2019. Edge computing: A survey. *Future Generation Computer Systems* 97 (02 2019). https://doi.org/10.1016/j.future.2019.02.050

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems* 25 (01 2012). https://doi.org/10.1145/3065386

[27] Steve Leibson and Nick Mehta. 2013. Xilinx ultrascale: The next-generation architecture for your next-generation architecture. *Xilinx White Paper WP435* 143 (2013).

[28] Sheng Li, Jung Ho Ahn, Richard Strong, Jay Brockman, Dean Tullsen, and Norman Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 469–480. https://doi.org/10.1145/1669112.1669172

[29] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. 2021. BRECQ: Pushing the Limit of Post-Training Quantization by Block Reconstruction. *arXiv e-prints*, Article arXiv:2102.05426 (Feb. 2021), arXiv:2102.05426 pages. arXiv:2102.05426 [cs.LG]

[30] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 2849–2858.

[31] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1025–1040. https://www.usenix.org/conference/atc19/presentation/liu-yizhi

[32] Xiangyu Lu. 2019. The Analysis of KMP Algorithm and its Optimization. *Journal of Physics: Conference Series* 1345 (11 2019), 042005. https://doi.org/10.1088/1742-6596/1345/4/042005

[33] B. Moons, K. Goetschalckx, N. Van Berckelaer, and M. Verhelst. 2017. Minimum energy quantized neural networks. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 1921–1925. https://doi.org/10.1109/ACSSC.2017.8335699

[34] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. 2020. Up or down? adaptive rounding for post-training quantization. In *International Conference on Machine Learning*. PMLR, 7197–7206.

[35] Gianmarco Ottavi, Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. 2020. A Mixed-Precision RISC-V Processor for Extreme-Edge DNN Inference. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 512–517. https://doi.org/10.1109/ISVLSI49217.2020.000-5

[36] Victor Pan. 1984. *How to Multiply Matrices Faster*. Springer-Verlag, Berlin, Heidelberg.

[37] V. Pan. 1993. Binary segmentation for matrix and vector operations. *Computers and Mathematics with Applications* 25, 3 (1993), 69 – 71. https://doi.org/10.1016/0898-1221(93)90144-K

[38] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. 2020. Binary neural networks: A survey. *Pattern Recognition* 105 (2020), 107281. https://doi.org/10.1016/j.patcog.2020.107281

[39] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. 2020. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures. *ACM Trans. Archit. Code Optim.* 17, 4, Article 38 (Nov. 2020), 30 pages. https://doi.org/10.1145/3422667

[40] RISC-V GNU Compiler Toolchain [n.d.]. *RISC-V GNU Compiler Toolchain*. https://github.com/riscv/riscv-gnu-toolchain

[41] RISC-V "V" Vector Extension [n.d.]. *RISC-V "V" Vector Extension*. https://github.com/riscv/riscv-v-spec/releases

[42] Arnold Schönhage. 2006. *Asymptotically fast algorithms for the numerical muitiplication and division of polynomials with complex coefficients*. 3–15. https://doi.org/10.1007/3-540-11607-9_1

[43] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[44] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[45] Marc Solé-Bonet and Leonidas Kosmidis. 2022. SPARROW: A Low-Cost Hardware/Software Co-designed SIMD Microarchitecture for AI Operations in Space Processors. *2022 Design, Automation Test in Europe Conference Exhibition (DATE)* (2022).

[46] Aravind Vasudevan, Andrew Anderson, and David Gregg. 2017. Parallel Multi Channel Convolution using General Matrix Multiplication. *arXiv e-prints*, Article arXiv:1704.04428 (April 2017), arXiv:1704.04428 pages. arXiv:1704.04428 [cs.CV]

[47] A. Vasudevan, A. Anderson, and D. Gregg. 2017. Parallel Multi Channel Convolution using General Matrix Multiplication. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 19–24. https://doi.org/10.1109/ASAP.2017.7995254

[48] S. Venkataramani, V. Srinivasan, W. Wang, S. Sen, J. Zhang, A. Agrawal, M. Kar, S. Jain, A. Mannari, H. Tran, Y. Li, E. Ogawa, K. Ishizaki, H. Inoue, M. Schaal, M. Serrano, J. Choi, X. Sun, N. Wang, C. Chen, A. Allain, J. Bonano, N. Cao, R. Casatuta, M. Cohen, B. Fleischer, M. Guillorn, H. Haynie, J. Jung, M. Kang, K. Kim, S. Koswatta, S. Lee, M. Lutz, S. Mueller, J. Oh, A. Ranjan, Z. Ren, S. Rider, K. Schelm, M. Scheuermann, J. Silberman, J. Yang, V. Zalani, X. Zhang, C. Zhou, M. Ziegler, V. Shah, M. Ohara, P. Lu, B. Curran, S. Shukla, L. Chang, and K. Gopalakrishnan. 2021. RaPiD: AI Accelerator for Ultra-low Precision Training and Inference. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 153–166. https://doi.org/10.1109/ISCA52012.2021.00021

[49] Peisong Wang, Qiang Chen, Xiangyu He, and Jian Cheng. 2020. Towards accurate post-training network quantization via bit-split and stitching. In *International Conference on Machine Learning*. PMLR, 9847–9856.

[50] Qianru Zhang, Meng Zhang, Tinghuan Chen, Zhifei Sun, Yuzhe Ma, and Bei Yu. 2019. Recent advances in convolutional neural network acceleration. *Neurocomputing* 323 (2019), 37–51. https://doi.org/10.1016/j.neucom.2018.09.038