Information Sciences 716 (2025) 122251

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Neural network compression using binarization and few full-precision weights

Franco Maria Nardini [a], Cosimo Rulli [a,*], Salvatore Trani [a], Rossano Venturini [b]

[a] ISTI-CNR, Pisa, Italy
[b] University of Pisa, Pisa, Italy

ARTICLE INFO

Keywords:
Deep neural networks
Model compression
Matrix multiplication
Image classification

ABSTRACT

Quantization and pruning are two effective Deep Neural Network model compression methods. In this paper, we propose *Automatic Prune Binarization* (APB), a novel compression technique combining quantization with pruning. APB enhances the representational capability of binary networks using a few full-precision weights. Our technique jointly maximizes the accuracy of the network while minimizing its memory impact by deciding whether each weight should be binarized or kept in full precision. We show how to efficiently perform a forward pass through layers compressed using APB by decomposing it into a binary and a sparse-dense matrix multiplication. Moreover, we design two novel efficient algorithms for extremely quantized matrix multiplication on CPU, leveraging highly efficient bitwise operations. The proposed algorithms are 6.9× and 1.5× faster than available state-of-the-art solutions. We extensively evaluate APB on two widely adopted model compression datasets, namely CIFAR-10 and ImageNet. APB shows to deliver better accuracy/memory trade-off compared to state-of-the-art methods based on i) quantization, ii) pruning, and iii) a combination of pruning and quantization. APB also outperforms quantization in the accuracy/efficiency trade-off, being up to 2× faster than the 2-bits quantized model with no loss in accuracy.

1. Introduction

Deep Neural Networks (DNNs) had an unprecedented impact on computer vision and achieve state-of-the-art performance in many different tasks, such as image classification [1], semantic segmentation [2], and object detection [3]. A key characteristic of deep neural networks is the positive effect of overparameterization [4] on their generalization capabilities. Overparameterization refers to the scenario where the number of learnable parameters surpasses the number of training instances; hence, scaling up the number of parameters has become a common recipe for improving the effectiveness of the neural model. This has driven the development of increasingly larger neural networks, with models reaching trillions of parameters. However, as the model size grows, so do the computational demands for training and deployment. This is in contrast with the need for pervasive distribution of neural networks, including their usage in resource-constrained and edge applications. Model compression [5] is the field of Deep Learning devoted to decreasing the computational requirements of Deep Neural Networks (DNNs) without hurting their effectiveness and generalization capabilities.

* Corresponding author.
E-mail address: cosimo.rulli@isti.cnr.it (C. Rulli).

https://doi.org/10.1016/j.ins.2025.122251
Received 17 June 2024; Received in revised form 24 April 2025; Accepted 24 April 2025

Available online 30 April 2025

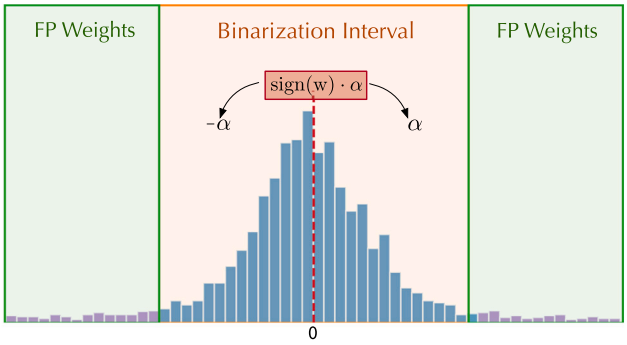

Fig. 1. Graphical illustration of Automatic Prune Binarization (APB). The weights in the binarization interval are converted to $\{-\alpha, \alpha\}$ according to their sign, while the remaining values are kept in full precision.

Among the various compression methods available, quantization and pruning stand out as two of the most effective. Quantization techniques reduce the number of bits required to represent the network parameters, thus offering compelling properties in terms of space saving and inference speedup.

Quantization is a non-differentiable operator, preventing its straightforward usage in neural network training based on gradient descent. The quantization step function can be approximated using a differentiable proxy function during the gradient computation in the backward pass [6]. For instance, in 1-bit quantization (binarization), the $sign$ function is used as a quantization, and the hyperbolic tangent or identity function are two common choices for the proxy function. Binarization reduces the memory burden of 32× with respect to its full-precision counterpart and converts floating-point multiplications into cheap bitwise operations [7]. Despite the work done in this field [8], binary networks still struggle to match the performance of the corresponding full-precision model. At the same time, 2 or 3-bits quantization approaches are closing the gap with full precision models [9], but do not ensure the compelling properties of binary networks, especially in terms of inference speedup on CPU. In fact, no efficient matrix multiplication implementation is available for these bit configurations, requiring to upcast the operands to the closest bit width natively supported by the CPU, usually 8 bits.

On the other hand, pruning techniques remove network parameters to produce sparse models as accurate as their original dense versions [10]. Pruning allows for consistent memory savings as only the non-zero parameters and their positions need to be saved. Despite the reduction of Floating Points OPerations (FLOPs), pruning a neural network does not imply a remarkable inference speedup until extreme sparsity levels are achieved, i.e., $> 95\%$, as CPUs and GPUs are optimized for dense computation. Pruning requires searching for the optimal sparsification pattern among 2^n possible solutions, where n is the number of parameters in the model. The problem rapidly becomes intractable, and, as for quantization, the step function modeling pruning is non-differentiable. Hence, pruning methods rely on heuristics to determine whether to prune or not a certain weight, such as its magnitude [11]. To summarize, binarization techniques allow for fast neural inference but do not match the performance of the equivalent full-precision models. On the other hand, pruning techniques can produce highly effective sparse networks but do not deliver consistent speedup until extreme sparsity ratios are reached.

Pruning and quantization follow two orthogonal approaches: while quantization distributes the representation capability of the network across a large set of low-precision values, pruning condenses it into a few full-precision values. Due to their complementary nature, researchers have been developing solutions that combine pruning and quantization. Existing approaches typically follow a prune-and-quantize pipeline, where weights that survive the sparsification step are quantized to a certain bit width, such as the methods proposed in Bayesian Bits (BB) [12], Multi Prized Lottery Ticket Hypothesis [13], and Single-path Bit Sharing [14]. We propose to combine pruning and quantization in a novel way. Instead of pruning *and* quantizing the network weights, we either binarize *or* keep them in full precision, with the purpose of maintaining the desirable properties of binary networks while empowering the representational capability of the model with few full-precision entries.

In this paper, we propose Automatic Prune Binarization (APB), a novel compression method that combines low-bit quantization (binarization) with pruning. APB allows each parameter to assume either a binary or a full-precision representation (Fig. 1). APB jointly maximizes the accuracy achieved by the network while minimizing its memory impact by identifying an optimal partition of the network parameters among these two sets. In detail, it works by identifying a binarization interval (centered in 0), and the parameters falling in this interval are represented using one bit. As done by other state-of-the-art binarization approaches, according to their sign, binarized parameters are converted to $\{-\alpha, \alpha\}$, where α is a learned layer-wise scalar value (Fig. 1). On the other hand, parameters outside the binarization interval are kept in full precision. By doing so, APB produces two different overlapping networks, i.e., a binary network and an extremely sparse full-precision network. Their combined expressiveness allows for improving the performance of binary networks without the need to double the bits required to represent the weights, as done in 2-bits quantization. This goal can be achieved when the number of full-precision weights is sufficiently low. In this case, APB also offers compelling inference properties. In fact, we experimentally show that the overhead given by the sparse network is negligible at our sparsity ratios. Moreover, we develop and present two novel matrix multiplication algorithms for CPU in extreme quantization scenarios. Our approach works by remapping a q-bits matrix in a set of $q + 1$ binary matrices. Then, these matrices can be efficiently multiplied by leveraging cheap bitwise operations. We provide a high-performance implementation of these novel algorithms on CPU and show that our

implementation can be much faster than currently available general-purpose solutions [15] for quantized matrix multiplication. To the best of our knowledge, these are the first techniques tailored for quantized matrices on CPU, except for the binary case. This allows the evaluation of the efficiency of highly quantized networks on CPU for the first time. Overall, we experimentally show that APB outperforms the performance of state-of-the-art quantization approaches in terms of memory/accuracy and efficiency/accuracy trade-offs.

In detail, the novel contributions of this work are:

- we introduce Automatic Prune Binarization (APB), a novel compression framework designed to simultaneously optimize network accuracy and minimize memory consumption. APB achieves this by automatically determining, for each weight, whether it should be binarized or retained in full precision. We show how the problem of partitioning the set of weights can be addressed using Stochastic Gradient Descent as in standard DNN training.
- we tackle the challenge of improving the efficiency of quantized neural networks on CPUs. We discuss the efficiency of matrix multiplication as a function of the operands' bit width. We then design two novel matrix multiplication routines based on efficient bitwise operations available on the CPU for extreme quantization scenarios. To exploit this innovative solution in conjunction with APB, we show how the forward pass through a layer compressed using APB can be decomposed into a binary multiplication and a sparse binary multiplication.
- we provide an extensive experimental evaluation on two public datasets used in model compression, i.e., CIFAR-10 and ImageNet [16], benchmarking APB against the state-of-the-art model compression techniques. Our competitors include quantization, pruning, and combinations of pruning and quantization. The evaluation showcases that APB offers better accuracy/memory trade-off than existing approaches.
- we assess the performance of our newly developed low-bit matrix multiplication routines in the forward pass of neural networks on the ImageNet dataset. Our methods are 6.85× and 1.5× faster than available state-of-the-art solutions. Moreover, APB, used together with these routines, shows superior performance even in the accuracy/efficiency trade-off, being 2× faster than the best 2-bits quantized model with no loss in accuracy.

The rest of the paper is organized as follows: in Section 2, we present the related work in binarization, low-bit quantization, pruning, and efficient inference on CPU for neural networks. In Section 3, we present our novel APB compression framework mixing full-precision and binary weights. In Section 4, we discuss the efficiency of matrix multiplication at different bit widths and we introduce our novel low-bit matrix multiplication routines. Section 5 presents an experimental evaluation of APB. First, we discuss the memory compression performance of APB (Section 5.2). Then, we evaluate the performance of our matrix multiplication algorithms and compare the execution time of the quantized networks against APB (Section 5.3). Finally, Section 6 concludes the work and draws some future lines of research.

2. Related work

Model compression is crucial for the practical usage of Deep Neural Networks, allowing the deployment of large and resource-consuming architectures in constrained environments such as edge devices, sensors, and smartphones [17]. Our method lies at the intersection of two families of compressors, i.e., *binarization/low-bit quantization* [18] and *pruning* [19]. In the following, we review the main contributions in these lines along with the ones investigating *efficient inference* algorithms on quantized networks [20].

Binarization. Pioneering work on binarization are BinaryConnect [18] and XNOR-Net [7]. These methods rely on the use of the *sign* function to constrain the weights in $\{-1, 1\}$ and to scale them by the mean of their absolute value. More recent work leverages advanced techniques to train highly accurate binary models. Recently, in that line, some works show that maximizing the entropy of the binarized weights is an effective approach as shown by Qin et al. [21] and Li et al. [22]. Xu et al. [23] show the importance of *latent weights*, i.e., full-precision weights used during backpropagation and weight update. The authors focus on *dead weights*, i.e., weights that are rarely updated due to their distance to the origin. Authors show that these weights are responsible for hampering the training process, and they propose a tailored Rectified Clamp Unit to revivify those weights. Liu et al. tackle the problem of frequent weight flipping, i.e., weights changing their sign, by employing two learnable scaling gradient factors for the activations, one for each of the binary states ($\{-1, 1\}$) [24]. Another family of approaches proposes architectural changes to the network to quantize. Bi-Real Net adds a double skip-connection on the ResNet architecture to sum the real-valued input with the features obtained after a binary convolution [25]. A well-known solution in this field is ReActNet [26]. Here, the authors duplicate the input channels of convolutional layers, introduce tailored activation functions for binary networks, and employ knowledge distillation to enhance the training phase. Hu et al. introduce *Squeeze* and *Expand* layers aiming at combining input and output activations [27]. For a comprehensive discussion on binary networks, we recommend the reading of BiBench [28], a recently released benchmark comparing the performance of binarization algorithms on different tasks and on different architectures. We point out that in our work, we will only focus on binarization algorithms for convolutional neural networks.

Low-bits Quantization. Quantization techniques often rely on the Straight-Through Estimator (STE) [6] to propagate the gradients through non-differentiable quantization functions. Lee et al. propose to overcome the limits of STE by exploiting an element-wise gradient correction method [9]. Their approach, named Element-Wise Gradient Scaling (EWGS), scales the gradient of each full-precision weight according to three factors: i) its sign, ii) the gap between full-precision and quantized value, iii) a scaling factor learned through the approximation of the Hessian. SLB employs a continuous relaxation strategy to overcome the gradient mismatch

problem [8]. Each weight of the network is mapped to a probability distribution that represents the values it can assume with a q-bits representation. The values associated with the highest probabilities are selected as quantization values. Recently, several works have focused on quantizing Large Language Models (LLMs) fueled by transformer-based architectures. GPTQ [29] is a post-training quantization method based on minimization of the reconstruction error for the network weights that show efficient and effective compression on GPT models. For example, QLoRA [30] introduces NormalFloat (NF4) to better capture the weight distribution in LLMs and utilizes double quantization to compress the quantization constants. Their results demonstrate that 4-bit compressed LLMs can perform on par with full-precision models across several benchmarks. We also recommend the reading of Jin et al. [31] for a comparison between existing quantization methods on LLMs. However, as this work addresses the compression of convolutional neural networks (CNNs), comparisons with methods used for LLMs are beyond the scope of this paper.

Pruning. Pruning techniques effectively sparsify neural networks with small/no accuracy degradation.[1] Recently, a plethora of different pruning methods has been developed. For a comprehensive discussion, we recommend the reading of dedicated surveys [32]. We highlight that the importance of high absolute-value weights in neural networks was first discovered in pruning techniques. In fact, magnitude-based heuristics save high absolute-value weights while zeroing out the others. Han et al. are the first to apply magnitude-based pruning in conjunction with re-training of the network to mitigate the possible performance degradation [33]. Lately, magnitude pruning has been improved by introducing re-winding, namely reassigning the surviving parameters to their initialization values, showing that it outperforms fine-tuning on several network architectures and datasets [34]. A lot of effort has been spent in training sparse networks (or pruning them at initialization), rather than pruning after training. This interest is motivated by the so-called "Lottery Ticket Hypothesis", namely the existence of highly effective sparse networks in randomly initialized dense networks [11].

Combination of Pruning and Quantization Several works explore the combination of pruning and quantization, to fruitfully exploit the advantages provided by both techniques. Han et al. [33] apply clustering algorithm on the weights surviving the pruning phase, then optimize the value of the centroids using the average of the weight gradients. Bayesian Bits [12] is an approach where mixed precision quantization is combined with pruning. In particular, network weights are either quantized to a power-of-two-bit width or zeroed out in a data-driven fashion. In "Multi-Prize Lottery Ticket" (MPT) [13], Diffenderfer et al. mix pruning and binarization by i) discovering highly-effective sub-networks in neural networks, ii) binarizing the surviving values. Single-path Bit Sharing (SBS) [14] is a method for automatic loss-aware model compression, focusing on joint pruning and quantization to achieve a highly compact model efficiently. SBS utilizes a single-path bit-sharing model to encode all bit widths in the search space, enabling the automatic determination of compression configurations for each layer while balancing between pruning and quantization. Recently, a combination of pruning and quantization approaches has been applied to transformer-based architectures [35]. This method, named SPQR, isolates outlier weights in each layer and keeps them in full precision. SPQR differs from `APB` for two main reasons: first, SPQR requires to isolated *block* of weights, such as rows or columns, while `APB` works on a single element of a tensor. Second, SPQR is a post-training quantization method, while `APB` leverages training-time gradients to optimally partition binary and full-precision weights.[2]

Efficient Inference. Nurvitadhi et al. study the efficiency of binary multiplication on different hardware platforms. Authors estimate a speedup of 2× of binary over single-precision multiplication on a CPU equipped with 64-bit bitwise instructions [20].

Regarding efficient inference of binary networks on CPU, a major contribution is BitFlow, a binary convolution algorithm (*PressedConv*) based on bit-packing on the channel dimension [36]. BitFlow provides 1.8× speedup compared to naïve binary convolution. In this line, DaBnn [37] and Larq [38] are efficient inference frameworks for binary neural networks on mobile platforms powered by ARM processors. Our work is the first one to study the efficiency of low-bit quantized neural networks on general-purpose CPUs. Moreover, we will release the optimized source code we developed for binary and low-bit multiplications on this kind of CPU.

Our Contribution. `APB` approach is orthogonal to other techniques mixing pruning and quantization. As an example, in works combining binarization and pruning [13], parameters are either zero or binary ($\{-1, +1\}$). `APB`, instead, enriches the expressiveness of binary networks with full-precision parameters in the same framework that effectively mixes binarization and pruning. This means that weights in `APB`-compressed networks are either binary or *full-precision*. To the best of our knowledge, we are the first to jointly optimize binary and full-precision parameters in a novel and end-to-end compression framework.

3. Automatic prune binarization

We now describe `APB`, our novel compression framework that adds a few full-precision values to binary networks. First, we introduce the role of binarization. Second, we show how large absolute-value weights play different roles in binarization and pruning. Third, we formally introduce `APB` and show how its parameters can be optimized by leveraging Stochastic Gradient Descent (SGD). Finally, we show how to decompose the matrix multiplication into dense-dense and sparse-dense matrix multiplications.

Binarization. Let us consider $W \in \mathbb{R}^n$ as the set of weights of a neural network. The scope of binarization is to employ a single bit to store each weight $w \in W$, forcing w to be in $\{-1, +1\}$. Previous studies show that it is possible to enhance the expressiveness of the model by scaling the weights with a scalar α [7] so as to remap them to $\{-\alpha, +\alpha\}$. We rely on re-scaled binarization that asks for the definition of a `Bin` operator defined as follows:

[1] With pruning, we always refer to element-wise pruning. See Liang et al. [32] for a complete analysis of the difference between element-wise and structured pruning.

[2] Our approach was developed at the same time and independently with respect to SPQR.

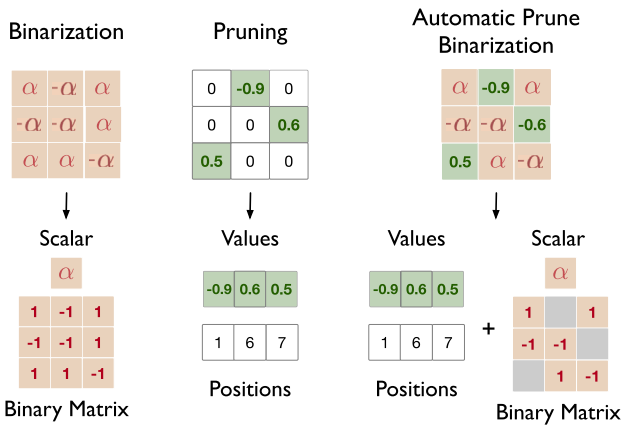

Fig. 2. Network weights representation in binarization (left), pruning (center), APB (right). In APB, binary, and full-precision weights coexist in the same matrix. For full-precision entries, we are required to store also the index inside the weight matrix.

$$\texttt{Bin}(w) = \alpha \cdot \text{sign}(w) = \begin{cases} +\alpha & \text{if } w \geq 0 \\ -\alpha & \text{if } w < 0. \end{cases} \tag{1}$$

The Bin operator is defined as a function of $sign(w)$, whose derivative is zero almost everywhere. This hinders the use of gradient-based approaches to train the model. For this reason, gradients are approximated with the Straight Through Estimator (STE) [6]. STE works by using a surrogate differentiable function $g(w)$ that approximates $sign(w)$. Hence, we can use the derivative of $g()$ in place of the derivative of $sign()$. In practice, STE imposes that:

$$\frac{\partial \texttt{Bin(w)}}{\partial w} = \frac{\partial g(w)}{\partial w}. \tag{2}$$

This derivative is used to update the full-precision weights W_l, which are referred to as *latent weights* [23] in the context of binarization. The final binary matrix W is then obtained by simply applying the Bin operator over W_l.

Large Absolute-value Weights. Given the STE training strategy, the binary weight w is updated only when there is a flip of sign in the corresponding latent weight $w_l \in W_l$, as a result of the gradient update. In a recent work [23], authors identified the problem of large absolute-value parameters that diverge from the zero-centered Laplacian distribution characterizing latent weights, namely *dead weights*. These weights interfere with the optimization phase, giving a reduced chance to change their sign. In practice, they freeze part of the network, hindering the training process.

Even in pruning, large absolute-value weights play a central role. Here, network layers are sparsified by zeroing out less important weights, and the importance of each parameter is chosen according to a heuristic. Interestingly, a simple yet very effective heuristic to determine the importance of a parameter is its magnitude [19]. Several works show that a small portion ($< 10\%$) of large absolute-value weights is enough to match the performance of the dense model [39].

Large absolute-value weights thus play contrasting roles in binarization and pruning techniques. In binarization, they hamper and slow down the training process due to the low likelihood of changing the sign. In pruning, they synthesize the expressiveness of the overall model. We build our approach on this discrepancy: we leverage the advantages that large-absolute weights provide in pruning while mitigating the drawbacks associated with binarization. This is the key intuition underlying APB, our novel method for effective compression of neural networks. APB keeps high absolute-value weights in full precision, and it binarizes the remaining ones. For this purpose, APB defines a symmetric binarization interval around zero (Fig. 1): weights falling outside this interval are kept in full-precision, while the others are mapped to $\{-\alpha, +\alpha\}$ according to their sign. In this regard, APB can be interpreted as a pruning technique where small absolute-value weights are binarized instead of being zeroed out [19]. The amplitude of the binarization interval and the value of the scalar α are optimized during training. As shown in Fig. 2, APB compresses the network by applying pruning and binarization in parallel, i.e., each weight is either full-precision or binary. Conversely, in classical approaches mixing pruning and quantization/binarization, each parameter is either zeroed-out or quantized/binarized (Section 2). Within APB, two different networks coexist during the training: a binary and a sparse network.

APB. Given the considerations on the role of large absolute-value weights, APB employs weight magnitude to determine whether weight should be binarized or kept in full precision. Our approach is partially inspired by [40], which introduces a trainable threshold to determine if a weight should be set to 0 or 1. The operator APB on a weight w is defined as:

$$\texttt{APB(w)} = \begin{cases} \text{sign}(w)\,\alpha & \text{if } |w| \leq \alpha + \delta \\ w & \text{otherwise,} \end{cases} \tag{3}$$

with δ being the amplitude of the binarization interval exceeding α. Fig. 3 graphically depicts how APB works. The weights whose absolute value ranges in $[0, \alpha + \delta]$ are binarized (orange area), while the parameters falling outside this interval are kept in full precision (green area).

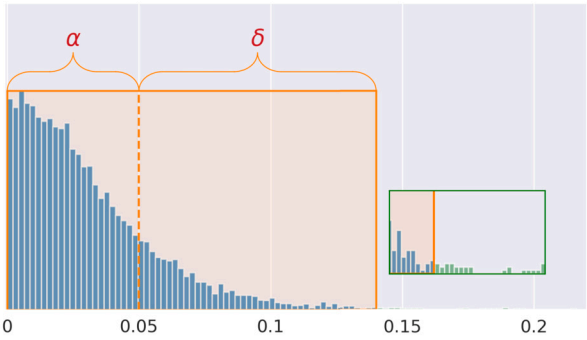

Fig. 3. Automatic Prune Binarization (APB) applied on the network parameters W. The width of the binarization interval (orange area) is defined by $\alpha + \delta$. A weight $w_i \in W$ is binarized if $|w_i| \leq \alpha + \delta$; otherwise, it is kept in full-precision (green area).

If w is within the binarization interval, APB is not differentiable. In this case, we apply STE by employing the identity function, i.e., $id(x) = x$, as $g(w)$ [6]. Thus, the derivative becomes:

$$\frac{\partial \text{APB}}{\partial w} = \begin{cases} g'(w) & \text{if } |w| \leq \alpha + \delta \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

Assuming $\delta \geq 0$, we can re-write

$$|w| \leq \alpha + \delta \quad \Rightarrow \quad \frac{|w| - \alpha}{\delta} \leq 1. \tag{5}$$

To ease the notation, we define $\hat{w} = \frac{|w|-\alpha}{\delta}$. This entails:

$$\text{APB}(w) = \begin{cases} \text{sign}(w)\,\alpha & \text{if } \hat{w} \leq 1 \\ w & \text{otherwise.} \end{cases} \tag{6}$$

We define the indicator function of the set of binarized weights as $\chi_B := \mathbb{1}(\hat{w} \leq 1)$. We can now define the gradients of α and δ by unrolling the derivatives of the loss function $\mathcal{L}$. We introduce the indicator function to constrain α and δ to depend exclusively on the binarized weights and by leaving them independent from those weights that APB leaves full precision.

$$\frac{\partial \mathcal{L}}{\partial \delta} = \frac{1}{n} \sum \frac{\partial \mathcal{L}}{\partial \hat{w}} \frac{\partial \hat{w}}{\partial \delta} \chi_B = \frac{1}{\delta^2 n} \sum \frac{\partial \mathcal{L}}{\partial \hat{w}} (\alpha - |w|) \chi_B. \tag{7}$$

$$\frac{\partial \mathcal{L}}{\partial \alpha} = \frac{1}{n} \sum \frac{\partial \mathcal{L}}{\partial \hat{w}} \frac{\partial \hat{w}}{\partial \alpha} = -\frac{1}{\delta n} \sum \frac{\partial \mathcal{L}}{\partial \hat{w}} \chi_B. \tag{8}$$

We also observe that

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{\delta} \frac{\partial \mathcal{L}}{\partial \hat{w}} \text{sign}(w). \tag{9}$$

We can compute $\frac{\partial \mathcal{L}}{\partial \alpha}$ and $\frac{\partial \mathcal{L}}{\partial \delta}$ using $\frac{\partial \mathcal{L}}{\partial w}$, which is the standard derivative of the weights w.r.t. the cost function, obtained using the backpropagation algorithm [41].

Memory Impact. We denote with $\text{Mem}(W)$ the memory impact of the matrix $W \in \mathbb{R}^n$ compressed using APB, expressed in bits. Assuming to have s full-precision surviving entries, we need to store $n - s$ binary weights and s full-precision values with their position. However, since $s \ll n$ and for easing the matrix multiplication, the binary matrix is fully represented. Hence:

$$\text{Mem}(W) = (n - s) + s(b_v + b_p) \simeq n + s(b_v + b_p) \tag{10}$$

where b_v is the bit width of the full precision values (32 for floating point) and b_p represents the bits needed to store their positions in the matrix. For each neural architecture under evaluation, we compute b_p as $\max_i \log_2(k_i - 1) + 1$, where k_i is the dimension of layer i.

Inference Considerations. We now discuss how to efficiently multiply a weight matrix A, compressed using APB, against an input activation matrix B.[3] For every $A_i \in A$ — where A_i is an element of the input matrix — we define the mask associated with A:

$$\text{Mask}(A_i) = \begin{cases} 1 & \text{if } A_i \in \{-\alpha, +\alpha\} \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

We decompose $A = A^{\text{bin}} + A^{\text{full}}$, where $A^{\text{bin}} \in \{-\alpha, +\alpha\}^n$ is a binary matrix and $A^{\text{full}} \in \mathbb{R}^n$ is a sparse full-precision matrix. A^{bin} is defined as $A^{\text{bin}}_i = \alpha \cdot \text{sign}(A_i)$, while A^{full} is given by

[3] Convolution can be converted to matrix multiplication using the *im2col* technique [42].

$$A_i^{\text{full}} = \begin{cases} 0 & \text{if } \texttt{Mask}(A_i) = 1 \\ A_i - A_i^{\text{bin}} & \text{otherwise.} \end{cases} \tag{12}$$

Given an input matrix B and the distributive property of matrix multiplication, we can write

$$C = A \cdot B = (A^{\text{bin}} + A^{\text{full}}) \cdot B = A^{\text{bin}} \cdot B + A^{\text{full}} \cdot B. \tag{13}$$

Since A^{bin} is a binary matrix, the only overhead introduced by `APB` is a sparse-dense matrix multiplication. Due to the extreme sparsity ratios of A^{full}, the sparse-dense multiplication can be efficiently performed with tailored implementations such as LIBXSMM [43].

4. Bitwise matrix multiplication

The efficiency of neural inference heavily relies on the efficiency of matrix multiplication (MM). MM has been widely studied [44] due to its paramount role in many scientific applications. Here, we discuss the efficiency of MM at different quantization levels and we introduce our novel routines for optimized matrix multiplication in low-bit configurations. In the following, we employ the notation `w/a` to specify the bit width of the weights and the activation matrices, respectively. In particular, we show how to implement efficient $1/2, 2/2$ MM using logical and bitwise operators. To the best of our knowledge, we are the first to investigate the efficiency of such low-bit configurations on CPU.

Efficiency of Matrix Multiplication. Matrix multiplication is known to be a memory-bounded problem. Indeed, several techniques have been developed to mitigate this aspect, and nowadays, its efficiency mostly depends on the available computational power.[4] The core operation of MM is the *update* function $c \leftarrow c + ab$, which is recursively applied on portions of the input matrices a, b to incrementally compute the output c. In the context of neural inference, c, a, and b represent the output, the weights, and the input of each layer, respectively. Modern CPUs allow performing the operation above with the *fused-multiply add* (`fma`) instruction, which computes the update with the same latency and throughput of an `add` instruction.[5] Furthermore, modern CPUs can rely on instruction-level parallelism (SIMD), which allows the processing of multiple inputs at the same time. As an example, the `_mm512_fmadd_ps` instruction computes the operation $c \leftarrow c + ab$ on three vectors of 16 full-precision values each. The theoretical peak performance (tpp) measures how many update functions (up) can be carried out per second (sec) in an ideal scenario assuming that the memory cost is negligible, i.e.,

$$tpp = cf \cdot tp \cdot v \ \frac{\text{up}}{\text{sec}}, \tag{14}$$

where cf is the clock frequency, tp is the throughput of the `fma` instruction, and v is the number of operands that can be stored on a CPU register. v is computed by dividing the width of the SIMD register, e.g., 512 for AVX-512, by the number of bits of the operand.

Network quantization exploits the performance gain obtained by reducing the bit width of weights and activations, hinging on the extreme flexibility of neural networks to model compression. Observe that every time the operand bit width halves, twice the data fits into the same CPU register (v). Consequently, tpp doubles each time the operand bit width halves. However, on modern CPUs, the `fma` instruction exists only for double/single/half-precision float and 8-bits integer values. For smaller bit widths, e.g., 2 or 3 bits, the use of the `fma` instruction requires to up-cast the operands to the closest supported data type, i.e., 8 bits. The situation is different for a binary network as it allows to implement MM by leveraging addition/subtraction or bitwise operations. This motivates us to investigate fast bitwise matrix multiplication techniques for efficient neural inference on CPU. We now discuss the $1/32$ and the $1/1$ cases, then we present our novel bitwise matrix multiplication routines for the $1/2$ and the $2/2$ scenarios.

1/32. In this quantization schema, weights w are constrained to assume values in $\{-1, 1\}$. Conversely, activations a are kept in full precision, i.e., they are represented by using a 32-bit floating point. This configuration is explored in several state-of-the-art quantization works, such as Qin et al. [21] and Gong et al. [45], as it features a 32× memory saving compared to the single-precision representation. In principle, binary weights convert multiplication into additions and subtractions [7]. We will now show that this conversion does not deliver a remarkable speedup over classical multiplication. Given $w \in \{-1, 1\}$, in Equation (15) we show how to rewrite the dot product between w and a into a series of additions and subtractions. We can write,

$$w \cdot a = \sum_{i=1}^{n} w_i a_i = \sum_{j \in I_+} a_j - \sum_{j \in I_-} a_j, \tag{15}$$

where $I_+ = \{i \mid w_i = 1\}$ and $I_- = \{i \mid w_i = -1\}$. In detail, I_+ is the set of indexes of positive weights ($w_i = 1$), while I_- keeps track of negative weights ($w_i = -1$). In this formulation, we simply accumulate the activations in correspondence with positive weights and then subtract the sum of the activations in correspondence with negative weights. Indeed, the `fma` operation achieves the same latency and throughput as the `add/sub` operations on modern CPUs.[6] This means that the update function $c \leftarrow c + ab$ (Equation (15)) costs 2× the `fma`-based full-precision matrix multiplication. We can think of a more efficient approach that reduces the update function to a single addition operation. Let us define

[4] https://en.algorithmica.org/hpc/algorithms/matmul/.
[5] https://www.agner.org/optimize/instruction_tables.pdf.
[6] https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

$$T = \sum_{i=1}^{n} a_i, \quad I_+ = \sum_{j \in I_+} a_j, \quad I_- = \sum_{j \in I_-} a_j. \tag{16}$$

with $T = I_+ + I_-$ by construction. T is the sum of the activations along the columns, and it does not depend on the weights w. This means that T can be computed at the beginning of the multiplication and then reused for all the columns of a. Moreover, it can be computed in $\Theta(n^2)$ time as it requires the sum of n values along n columns. Its impact is negligible compared to $\Theta(n^3)$ — time needed to run the matrix multiplication. T can be exploited to avoid the computation of one between I_+ or I_-. For example, we can obtain I_- as $I_- = 2\,I_+ - T$. Thus, we can compute $w \cdot a$ as

$$w \cdot a = I_+ - I_- = 2\,I_+ - T.$$

The cost of multiplying by 2 and subtracting T is negligible, as it does not depend on the size of the input. The cost of multiplying $w \in \{-1, 1\}$ and a is dominated by the computation of I_+. It can be efficiently implemented by leveraging a masked floating point addition, where the mask identifies the $i \in I_+$. On recent Intel's instruction set AVX512, the masked floating point addition can be implemented by using the `_mm512_add_ps` instruction. Anyway, this instruction has the same latency and throughput of `fma`. As a consequence, it does not offer any computational advantage compared to 32-bit floating-point MM. Moreover, the binary representation for the weights is useless in this case, as they require to be converted into 32-bit float numbers when performing the vectorized `_mm512_add_ps` on CPU. This can be done in two ways: i) the representation of the binary weights is expanded in memory with the consequence of achieving the same memory footprint of full-precision computation, or ii) they are directly expanded to 32 bits when moved to the CPU registers with the consequence of having an overhead due to bit extraction. We conclude that the 1/32 scenario does not offer strong computational advantages compared to 32 floating point implementation. The observations we draw also hold if activations are quantized to 8 or 16 bits.

1/1. Several works investigate how to effectively train fully binary neural networks (Section 2), where both weights w and activations a can assume two values, i.e., $\{-1, +1\}$. Besides the memory savings achieved, 1/1 also allows fast inference. In fact, given two binary vectors $u, v \in \{-1, 1\}^n$, their dot product can be computed as

$$u \cdot v = n - 2 \cdot \texttt{popcount}(\texttt{xor}(u, v)). \tag{17}$$

We observe that Equation (17) holds for every $u^\gamma = \{-\gamma, \gamma\}^n$, $v^\beta = \{-\beta, \beta\}^n$, as $u^\gamma \cdot v^\beta = \gamma\beta(u \cdot v)$. It is, in fact, a common practice to add scalar multipliers to enrich the expressiveness of binary networks.

We compute the *tpp* for binary multiplication to compare it with full-precision computation. This requires estimating the optimal execution flow for the three AVX-512 operators (`xor, popcount, add`) required to perform the update function. We assume to work on a modern CPU such as the Intel SkyLake architecture, equipped with 512-bit registers and the `vpopcntq` instruction that enables the computation of the `popcount` operation on 512-bit registers. Recall that modern architectures have different execution units, each associated with a different port.[7] Instructions employing different ports can be executed during the same clock cycle. Moreover, if an instruction uses k different ports, it is executed k times in the same clock cycle. In this architecture, `xor` and `add` work on port $\{0, 5\}$ (throughput 2), while `popcount` works exclusively on port 5 (throughput 1).[8] Hence, even in a perfectly pipelined execution flow, the computation of the update function in a single clock cycle, as done in the dense case using the `fma` operation, is unfeasible. This would be possible only if all the operations (`xor`, `popcount`, and `add`) were assigned to different ports. In the best-case scenario, we can perform 2 complete updates (6 instructions, 1024 values) every 3 clock cycle (2 instructions per clock cycle). The number of values processed per clock cycle, on average, is $\frac{1024}{3} \simeq 341$. The same processor, equipped with AVX-512 registers and the `fma` instruction with throughput 2, can process $\frac{512}{32} = 16$ 32-bits floating point values per port, namely 32 floating point values per clock cycle. This means that the theoretical speedup offered by binary multiplication is 10.5×. We show that our implementation of binary multiplication reaches the theoretical speedup compared to a state-of-the-art library for matrix multiplication, oneDNN.[9] In practice, the empirical speedup over the equivalent single-precision MM can be up to 16× on rectangular matrices. In fact, it has been shown that dense multiplication does not reach the *tpp* on non-squared matrices [46], while our bitwise routine can. We will detail this aspect in our experimental evaluation, Section 5.3.

1/2. This quantization schema achieves more accurate models than 1/1. State-of-the-art quantization approaches such as SLB [8] and EWGS [9] have shown that 1/2 delivers up to 9 points of Top1 accuracy gain on the Image Classification task on ImageNet compared to the 1/1 quantization. Despite the effectiveness improvements achieved, no efficient MM routines have been proposed for this configuration on CPU. In this regard, we design a novel matrix multiplication routine for 1/2 quantization that exploits bitwise operations as in the 1/1 case.

Let us consider a binary vector $w \in \{-\gamma, \gamma\}^n$ and a 2-bits uniformly quantized vector $a = \{a_0, a_1, a_2, a_3\}^n$, with $a_p = ps$, with $p \in \{0, 1, 2, 3\}$ and where s represents the distance between quantization levels and p is the index of the quantized value. We want to efficiently compute $w \cdot a$ using logical and bitwise operators, as in Equation (17). a stores the activations of a neural model, so it is generally quantized to positive values (asymmetric quantization[10]). This is due to the choice of the ReLU as an activation function,

[7] https://easyperf.net/blog/2018/03/21/port-contention.
[8] https://www.agner.org/optimize/instruction_tables.pdf.
[9] https://github.com/oneapi-src/oneDNN.
[10] https://pytorch.org/blog/quantization-in-practice/.

Table 1
Definition of the transformation T mapping a 2-bits vector a into its three binary vectors decomposition.

a_i	$-\frac{3}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	$\frac{3}{2}$
t_i	0	0	0	1
h_i	0	0	1	0
m_i	1	0	0	1

which zeros out negative inputs. For the purpose of our technique, it is useful to re-map a as a zero-centered symmetric distribution. We scale a by its mean. We obtain $\bar{a}$ as

$$\bar{a}_i = a_i - \mu = a_i - \frac{b_0 + b_3}{2} = a_i - \frac{3}{2}s$$

$\forall i = 0, \ldots, n$. This converts the computation of $w \cdot a$ into

$$w \cdot a = \sum_{i=1}^{n} w_i(\bar{a}_i + \mu) = \sum_{i=1}^{n} w_i \bar{a}_i + \sum_{i=1}^{n} w_i \mu.$$

The last term is the sum along the rows of the matrix weight scaled by μ. Thus, it can be computed *offline*, before the matrix multiplication starts, as both terms are known a priori. Let us consider the update function $c \leftarrow c + ab$. We can initialize the matrix c with these pre-computed row-wise terms rather than with zeros to avoid any impact of this computation on the performance. Observe that $\bar{a}_i \in \{-\frac{3s}{2}, -\frac{s}{2}, \frac{s}{2}, \frac{3s}{2}\}$. Given that the multiplication by γ and s do not impact the performance, we need to multiply a $\{-1, 1\}$ binary vector against a 2-bit vector whose values are in $\mathcal{D}_a = \{-\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}\}$. $\mathcal{D}_a$ is composed of two pairs of zero-symmetric values, namely $\{-\frac{3}{2}, \frac{3}{2}\}$ and $\{-\frac{1}{2}, \frac{1}{2}\}$. This suggests the possibility of employing a tailored representation made of multiple binary vectors. Hence, we represent a using three binary vectors computed using the transformation

$$\begin{aligned} T : \{\mathcal{D}_a\}^n &\rightarrow \{0,1\}^{n\times 3} \\ a &\rightarrow \{t, h, m\} \end{aligned} \tag{18}$$

The definition of T is reported in Table 1. Observe that, from now on, we will represent binary vectors using $\{0, 1\}$ instead of $\{-1, 1\}$. This will ease the explanation of our approach, and it is also coherent on how binary vectors are actually represented during computation. The transformation T generates two binary vectors t, h, and a binary mask m. The names of the vectors suggest that t (*t*hree halves) refers to $\{-\frac{3}{2}, \frac{3}{2}\}$, while h (one *h*alf) refers to $\{-\frac{1}{2}, \frac{1}{2}\}$.

Two binary vectors, i.e., t and h, are built according to the following rules.

- a bit set to 1 uniquely identifies the entry of the original vector a. This means that if $t_i = 1 \Rightarrow a_i = \frac{3}{2}$ and if $h_i = 1 \Rightarrow a_i = \frac{1}{2}$.
- a bit set to 0 is intentionally ambiguous: $t_i = 0 \Rightarrow a_i \in \{-\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}\}$, $h_i = 0 \Rightarrow a_i \in \{-\frac{3}{2}, -\frac{1}{2}, \frac{3}{2}\}$.

The mask m is introduced to determine the uncertain entries of t and h. We have that

- $m_i = 1 \wedge t_i = 0 \Rightarrow a_i = -3/2$;
- $m_i = 0 \wedge h_i = 0 \Rightarrow a_i = -1/2$.

To ease the notation, we define the set of *active entries* for t as $E_t = \{i \mid m_i = 1\}$. The set of active entries for h is $E_h = \{i \mid \bar{m}_i = 1\}$. We highlight that T is a bijective function that permits uniquely matching a with its ternary representation $\{t, h, m\}$. By using the mask m, we introduce a memory overhead of one bit for each value of a. On the other hand, m is necessary to split the $1/2$ multiplication into two $1/1$ multiplications. The idea is to multiply the binary vector w with t and h respectively by involving in the computation only the active entries identified by the mask m. The considerations above inherently define a ternary operator $\texttt{mbm}(x, y, z)$, which we define on two generic binary vectors x, y, and a binary mask z. In this case, x plays the role of the weight vector, y represents one between t and h, while z is the mask corresponding to y, i.e., m and $\bar{m}$ when masking t and h, respectively. The set of active entries of y is $E_y = \{i \mid z_i = 1\}$.

We now discuss how to implement the `mbm` operator. First, let us focus on Equation (17). It works by subtracting from the total number of bits involved in the computation, i.e., n, the number of times the bits of u and v present different values, multiplied by 2. Our approach to compute $\texttt{mbm}(x, y, z)$ is to multiply x and y using Equation (17), and then exploit the mask z to fix the result. There are two corrections to be applied. The first one is on the number of bits actually involved in the computation that is identified by the cardinality of the active entries: $|E_y| = \sum_i z_i$, which can be computed with a $\texttt{popcount}(z)$. Now we need to count how many times x and y present different values in correspondence of active entries. First, we compute $d = \texttt{xor}(x, y)$. $d_i = 1$ implies $x_i \neq y_i$. These entries should be taken into account only if $z_i = 1$, i.e., only in correspondence with active entries of the vector y. This can be computed with $\texttt{popcount}(\texttt{and}(\texttt{xor}(x, y), z))$.

Given these considerations, mbm is defined as

$$\texttt{mbm}(x, y, z) = \texttt{popcount}(z) - 2\ \texttt{popcount}(\texttt{and}(\texttt{xor}(x, y), z)). \tag{19}$$

The overall 1/2 routine based on bitwise operations consists in applying the mbm routine twice, first on the active entries of t identified by m, and second on the active entries of h identified by $\bar{m}$.

$$w \cdot a = \texttt{mbm}(w, t, m) + \texttt{mbm}(w, h, \bar{m}). \tag{20}$$

As mentioned in Section 4, mbm can be efficiently implemented using the _mm512_ternarylogic_epi64 instruction. This instruction allows to compute any ternary logic function, i.e., any logic function with three inputs, and has the same latency and throughput of the xor instruction. That said, the cost of executing the mbm operator is the same as the one of computing Equation (17). popcount(z) is the sum along the columns of a bit matrix. It runs in $\Theta(n^2)$ and its cost is negligible with respect to the cost of matrix multiplication, i.e., $\Theta(n^3)$. For this reason, we estimate that $tpp_{1/2} \simeq 2\, tpp_{1/1}$.

2/2. This quantization schema almost fills the effectiveness gap with full-precision networks (Section 2). In this quantization schema, both the weight vector and the activation vector are 2-bit quantized vectors. The naive 2/2 MM can be obtained via an up-cast to 8/8 that allows the use of the fma instruction as discussed before.

We design an efficient alternative solution where both the weights and the activations are decomposed into three binary vectors. The first step is to zero-center the two vectors by following the procedure described for the 1/2 case. By doing that, we obtain the following w and a.

$$w = s_w\{-\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}\}, \quad a = s_a\{-\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}\}$$

where s_w and s_a represent the quantization step for w and a, respectively. The idea behind the 2/2 routine is to apply the 1/2 routine twice. Thus, the transformation T is applied to both w and a. We obtain $T(w) = \{t_w, h_w, m_w\}$ and $T(a) = \{t_a, h_a, m_a\}$. The multiplication is decomposed as follows

- the active entries of t_w, identified by m_w, are multiplied by the active entries of t_a, identified by m_a;
- the active entries of t_w, identified by m_w, are multiplied by the active entries of h_a, identified by $\bar{m}_a$;
- the active entries of h_w, identified by $\bar{m}_w$, are multiplied by the active entries of t_a, identified by m_a;
- the active entries of h_w, identified by $\bar{m}_w$, are multiplied by the active entries of h_a, identified by $\bar{m}_a$.

Finally, the results are summed together. We observe that for each one of the four bullets above, we have two different sets of active entries. The actual set of active entries is given by their intersection. This is implemented by a logical and between the two masks involved. To summarize, our novel routine for 2/2 MM allows to compute $w \cdot a$ as

$$w \cdot a = \texttt{mbm}(t_w, h_w, m_w \wedge m_a) + \texttt{mbm}(t_w, h_a, m_w \wedge \bar{m}_a) + \texttt{mbm}(h_w, t_a, \bar{m}_w \wedge m_a) + \texttt{mbm}(h_w, h_a, \bar{m}_w \wedge \bar{m}_a). \tag{21}$$

Hence, $tpp_{2/2} \simeq 4\, tpp_{1/2} \simeq 8\, tpp_{1/1}$.

5. Experimental evaluation

In this section, we present our experimental evaluation of APB. Our empirical analysis consists of two different steps. In the former one (Section 5.2), we assess the memory compression capabilities of APB by comparing it against i) pruning, ii) quantization, and iii) pruning combined with quantization/binarization for convolutional neural networks. In the latter subsection (Section 5.3), we perform an efficiency analysis by evaluating the performance of our novel low-bits matrix multiplication algorithms against existing solutions on CPU. Moreover, we leverage our routines to compare the execution time of convolutional neural networks at different bit-widths and compare them to APB-compressed models.

5.1. Experimental setup

Datasets. We comprehensively evaluate APB against several state-of-the-art competitors on two widely adopted datasets for Image Classification, namely CIFAR-10[11] and ImageNet [16]. The CIFAR-10 dataset consists of a set of 60 K 32×32 images split into 50 K and 10 K training/test samples, respectively, labeled with 10 classes. The ImageNet dataset consists of 1.2M training images and about 50 K test images labeled with 1,000 classes.

Network Architectures. We evaluate the performance of APB in compressing different kinds of convolutional networks. In particular, these architectures are ResNet-18/20/56 [1] and VGG-Small [47] on the CIFAR-10 dataset, and ResNet-18/34/50 and WideResNet-50 on ImageNet, which are the benchmark architectures for quantization methods. We apply APB on all the layers of the networks, except the first, the last, and the downsampling layers, unless differently specified. The effectiveness of the models is measured in terms of Top1 classification accuracy. We leave the evaluation of APB on transformer-based architectures for future work. In fact,

[11] https://www.cs.toronto.edu/~kriz/cifar.html.

Table 2
Comparison between APB and state-of-the-art CL competitors in terms of weights bit width and Top1 accuracy. Note that CL methods do not compress the first, the last, and downsample layers. To ease the comparison with Table 3, we report both the CL and AL weights bit-width using the format CL (AL).

Dataset	Network	Method	Weights Bit width CL (AL)	Top1 (%)
CIFAR-10	ResNet-20	FP	32.0 (-)	93.6
		IR-Net	1.0 (1.4)	90.8
		SLB	2.0 (2.4)	92.0
		APB (Ours)	1.0 (1.4)	92.3
	VGG-Small	FP	32.0 (-)	94.5
		SLB	1.0 (1.6)	93.8
		SLB	2.0 (2.6)	94.0
		APB (Ours)	1.0 (1.6)	94.3
	ResNet-18	FP	32.0 (-)	95.4
		MPT	1.0 (1.5)	94.8
		APB (Ours)	1.0 (1.5)	95.0
ImageNet	ResNet-18	FP	32.0 (-)	69.6
		EWGS	1.0 (3.0)	67.3
		EWGS	2.0 (4.0)	69.3
		APB (Ours)	1.4 (3.4)	69.2
		APB (Ours)	1.7 (3.7)	69.7
	ResNet-34	FP	32.0 (-)	73.3
		EWGS	1.0 (2.1)	72.2
		APB (Ours)	1.4 (2.5)	73.2
	ResNet-50	FP	32.0 (-)	76.1
		APB (Ours)	1.1 (7.3)	75.8
	WideResNet-50	FP	32.0 (-)	78.5
		MPT	1.0 (3.3)	74.0
		APB (Ours)	1.1 (3.4)	77.6

it has been shown that binarization and extreme quantization, in general perform poorly when applied to networks exploiting the attention module. [28]

Training Details. We apply APB after initializing the network weights with pre-trained models. For each layer i, we set $\alpha_i = \mu_i$ and $\delta_i = 3\sigma_i$, where μ_i is the mean of $|w_i|$ and σ_i is the standard deviation of w_i. Under the assumption that $w_i \sim \mathcal{N}(\mu, \sigma^2)$ [19], this ensure high compression rate at initialization. We employ an SGD optimizer with a cosine annealing learning rate scheduler. On CIFAR-10, we train for 500 epochs, with a batch size of 128 and a learning rate of 1e-3. On ImageNet, we train for 100 epochs when using real-values activations and 150 for 2-bits activations, with batch size 256 and learning rate 1e-3. We freeze the value of α and δ when reaching η epochs of each training to allow fine-tuning of the surviving values. APB is implemented in PyTorch.[12]

5.2. Compression performance of APB with 32-bits activations

In this section, we evaluate the performance of APB as a memory compression framework. For this purpose, we consider models whose activations are kept in 32-bits. Recall that, even when weights are binarized, maintaining activations in full precision prevents any inference advantage, as discussed in Section 4, paragraph "1/32".

We compare APB against several different techniques, namely i) quantization, ii) pruning, iii) combination of pruning and binarization. Compression methods adopt two main approaches. The first one, which is adopted, for example, by pruning techniques, compresses all the network layers. We call these methods *All-Layers Compressors* (AL). The second one, which is adopted by binarization, low-bits quantization, and pruning + binarization, works by leaving in full precision the first, the last, and the downsample layers, if any. We name these solutions *Convolutional Layers Compressors* (CL).

As already mentioned in Section 5.1, APB belongs to CL methods. Despite that, in our experiments, we compare it against methods belonging to both families. To ease the comparison, we report the results in two different tables. Table 2 reports the results for AL methods, while Table 3 reports the results for CL methods. In Table 2, we report the CL weights bit-width and the AL weight bit-width to allow a direct comparison between the two tables. Both for AL and CL, we measure the memory impact as average *weights bit width*, namely the ratio between the number of bits required to store a model and its parameters. When comparing against CL methods, the first, the last, and the downsample layers of the network are excluded from the computation, while these are included when comparing against AL methods.

5.2.1. Compression of convolutional layers (CL)

We start our comparison with methods compressing only the Convolutional Layers (CL).

Comparison with quantization approaches. We compare APB against state-of-the-art neural quantization techniques on CIFAR-10 and ImageNet and report the results in Table 2. All the low-bit quantization techniques belong to the CL category described above. For

[12] https://pytorch.org/.

Table 3
Comparison between APB and pruning in terms of compression rate width and Top1 accuracy. Here, we list All Layers Compressors (AL) methods, which compress the first, the last, and downsample layers. BB [12] and SBS [14] uses mixed-precision activations. APB marked with ‡indicates the usage of 2-bit activations, while †indicates that the last fully connected layer is compressed as well.

Dataset	Network	Method	Weights Bit width AL	Top1 (%)
CIFAR-10	ResNet-20	FP	32.0	93.6
		Pruning [34]	4.8	91.1
		APB (Ours)	1.4	92.3
	ResNet-56	FP	32.0	94.6
		Pruning [34]	4.7	93.9
		Pruning [34]	1.0	91.9
		APB (Ours)	1.3	93.6
ImageNet	ResNet-18	FP	32.0	73.3
		BB	5.1	69.5
		BB	4.1	68.1
		SBS	4.0	69.6
		APB (Ours)	3.4	69.7
		APB + PTQ (Ours)	2.4	69.6
		APB ‡+ PTQ (Ours)	2.8	67.4
	ResNet-50	FP	32.0	76.1
		SBS	4.0	75.9
		Pruning [34]	5.3	75.8
		APB (Ours)	7.3	75.6
		APB + PTQ (Ours)	5.4	75.6
		APB †+ PTQ (Ours)	2.1	75.3

each quantization scheme, we report the best-performing competitor. The state-of-the-art on CIFAR-10 is SLB [8] for all quantization schemes except the 1/32 case on ResNet-20, where the best performance is achieved by IR-Net [21]. On ImageNet, the state-of-the-art solution is EWGS [9] for both ResNet-18 and ResNet-34.

Results show that APB outperforms all other state-of-the-art quantization approaches by providing a superior solution in terms of space and accuracy. Indeed, APB achieves higher accuracy than the 2-bits quantization while saving up to 2× memory on CIFAR-10, both for the ResNet-20 and the VGG-Small architecture. On ImageNet, APB allows to save up 1.4× compared to 2-bits quantization. In comparison to 1-bit quantization, APB presents a negligible memory overhead on CIFAR-10. In fact, the surviving full-precision values for these models account for about 0.1% of the total weights. This means that APB is substantially always to prefer to quantization when compressing small or mid-sized networks on CIFAR-10. On ImageNet, the number of surviving full-precision weights required by APB is larger, as demonstrated by the 0.4 more bits per weight. However, this memory overhead is strongly counterbalanced by 1.9 and 1.0 Top1 accuracy points of gain for ResNet-18 and ResNet-34, respectively, compared to 1-bit quantization. On ResNet-18, our models even outperform 2-bits quantization by 0.4 Top1 accuracy points despite its reduced memory footprint.

Comparison with combinations of pruning and binarization. Recently, some works explore the combination of pruning & binarization [22]. These methods adopt an orthogonal approach with respect to APB. While with APB weights are either binary or full-precision, with existing approaches combining pruning and binarization, the weights are either binary or *zero*. We compare the performance of APB with respect to Multi Prize Ticket (MPT) [13] in terms of memory compression. MPT works by discovering highly effective sparse sub-networks in sufficiently over-parameterized neural networks without the need for further training. Moreover, they apply the *sign* function to binarize the surviving weights, thus providing a sparse and binary network.

We argue that sparsification does not provide memory advantages if the non-zero weights are binary. Consider a tensor of size n with nnz non-zero entries. The cost of storing a non-zero entry is given by the number of bits to its value b_v plus the number of bits to save its position b_p. In this case, i.e., $b_v = 1$, as surviving weights are binarized. In MPT [13], the authors propose a sparsification ratio of 80% for this architecture. This means that one weight out of five is stored. Storing 5 weights in pure binary format only costs 5 bits, so if $b_p > 4$, the sparse-binary format does not provide memory footprint advantages compared to the pure binary one. Due to the large size of network layers, 4 bits are not enough to store the indexes of non-zero values. A more memory-efficient approach would be to store one bit for every weight to indicate whether the corresponding entry is pruned. In this case, the total memory impact would be $n + nnz$, which we approximate with n for simplicity.

Table 2 reports the evaluation results of APB against MPT. APB outperforms the performance of MPT both on ResNet-18 for CIFAR-10, and for WideResNet-50 on ImageNet. For ResNet-18 in CIFAR-10, APB matches the memory compression of MPT and also delivers a slight (0.2) effectiveness improvement. For WideResNet-50 on ImageNet, the model learned with APB achieves more than 3 points of Top1 Accuracy improvement w.r.t. MPT. with a memory overhead of only 0.1 bits per weight.

5.2.2. Compression of all layers (AL)

We now compare APB to methods compressing the whole network. These approaches also tackle the first, the last, and the downsample convolutional layers. Batch Normalization layers are left in full precision as their impact is negligible, as they account for less than 0.5% of network parameters in all the evaluated architectures.

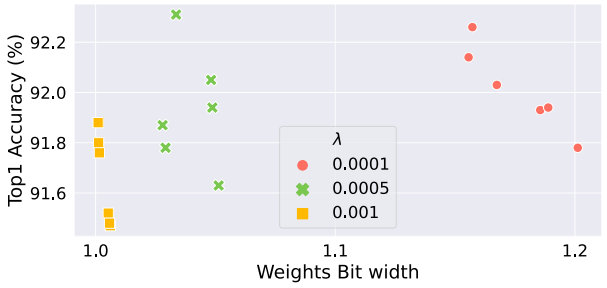

Fig. 4. The impact of weight decay (λ) on the Weights bit width and Top1 Accuracy λ. We employ ResNet-20 on CIFAR-10 as a benchmark. Larger λ yields a better compression rate at the cost of slightly reduced accuracy.

Table 2 shows that the impact of the layers left out by CL approaches can be considerable. On ImageNet, we observe that the non-compressed parameters account for an overhead of 2 bits per weight for ResNet-18 and 6.2 for ResNet-50. These numbers are obtained as the difference between the CL and the AL value in the "Weights Bit width CL (AL)" column of Table 2. On ResNet-18, we observe that the 70% of the overhead depends on the last fully connected layer. We experimentally verify that these last fully-connected layers can be quantized to 8-bit integers with a simple post-training quantization (PTQ) solution, that converts the weights from a floating point 32 representation to 8-bit integers[13] without harming the Top1 accuracy of the model. This halves the overhead of the non-compressed layers, reducing it to 0.9 bits per weight. For ResNet-50, quantizing the last fully connected layer to 8-bits allows reducing the overhead to 4.3 bit per weight.

Comparison with pruning. We compare the memory compression capabilities of APB against state-of-the-art pruning techniques, namely magnitude-based pruning with learning-rate rewinding [34]. We report the results of this comparison in Table 3. In the original article [34], the authors express the compression rate as the inverse of the sparsity ratio, i.e., 5% sparsity corresponds to 20× compression rate. Indeed, this does not account for the space required to store the indexes of the nonzero entries. We recompute the weights' bit width by taking it into account. We compare APB against pruning on ResNet-20, ResNet-56 on CIFAR-10, and on ResNet-50 on ImageNet. On CIFAR-10, APB vastly outperforms pruning in terms of memory/accuracy trade-off. When compressing ResNet-20, APB delivers a model which is 3.4× smaller but 1.2 points more accurate. Regarding ResNet-56, APB can deliver the same level of accuracy of a sparsified model with a 3.3× memory saving or the same level of memory compression with 1.7 points of Top1 accuracy improvement. On ImageNet, we observe that APB combined with Post Training Quantization (PTQ) on the last layer allows matching the performance of pruning using ResNet-50 as the backbone. Furthermore, we perform an experiment where we apply APB on the downsample convolutional layer of this architecture, marked with †in Table 3. For this model, the downsample convolutional layers account for the 10% of the total memory impact. Thus, we can obtain a model that only suffers from 0.5 accuracy degradation compared to pruning but allows saving up to 2.5× memory footprint.

Comparison with combinations of pruning and quantization. We compare with Bayesian Bits (BB) [12] and Single-path Bit Sharing (SBS) [14], state-of-the-art compression approaches mixing quantization at different bit widths with pruning. In detail, we compare in terms of bit width of their methods against our models learned with APB. As for the other experiments, we use ResNet-18 and ResNet-50 on Imagenet as the models for comparison and report the results in Table 3. As mentioned, we also apply PTQ on the final fully connected layer of ResNet18 APB can deliver the same accuracy but allows us to save up 2.1× compared to BB and 1.7× compared to SBS. For the sake of fairness, we point out that activations in these methods are quantized to mixed precision. In BB, they are quantized to 2/4 bits, while in SBS in 4/8. Hence, we compare to a model compressed with APB whose activations are quantized to 2-bits, marked with the ‡symbol in Table 3. Even if employing reduced bit width for activations, this model gains a 1.5× memory savings with reduced performance degradation over BB.

5.2.3. Ablation study

Weight Decay. The compression aggressiveness of APB can be controlled with the weight decay λ, namely the L_2 penalty applied on the weights of a neural network at training time. The larger the weight decay, the lower (on average) the absolute value of the trained parameters. We apply a different weight decay $\lambda_{\texttt{APB}}$ to α and δ, which is smaller than the λ for the standard weight parameters. Hence, the amplitude of the binarization interval is not directly affected by λ — just by $\lambda_{\texttt{APB}}$. As a consequence, increasing λ will force more weights to fall into the binarization interval, thus incrementing the percentage of binary values w.r.t. full-precision ones and delivering higher compression. Fig. 4 validates this hypothesis using ResNet-20 on the CIFAR-10 dataset. We run a grid search by fixing the batch size to 128 and the number of epochs to 500. We vary the weight decay $\lambda \in [1e\text{-}3, 5e\text{-}4, 1e\text{-}4]$, the freezing epoch η in $[250, 500]$ and $\lambda_{\texttt{APB}} \in [0, 1e\text{-}6, 1e\text{-}5]$. The plot shows how smaller values of λ allow the model to retain more full-precision weights, thus increasing the weights bit width. On average, this also increases the Top1 Accuracy. The experiment showcases how APB can span different points of the accuracy/memory trade-off by simply tuning the weight decay, without needing any additional hyperparameter.

[13] https://pytorch.org/docs/stable/quantization.html.

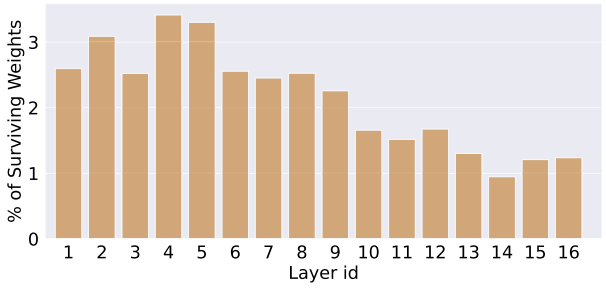

Fig. 5. Surviving full-precision weights per layer in ResNet-18 trained on ImageNet.

Surviving Weights. We study the distribution of surviving weights in the layers of a network compressed using APB. For this purpose, we use ResNet-18 on ImageNet as a reference. We train it using the same setting as in Table 2, 3 that allows it to reach 69.7 of Top1 Accuracy. We report the results of our evaluation in Fig. 5, showing the percentage of surviving full-precision weights (y-axis) per each layer of ResNet-18. As previously detailed, we only compress convolutional layers in ResNet18, excluding the first one. The plot shows that early layers tend to preserve more full-precision weights, while, as we go deeper in the network, their number decreases and stabilizes at around 1%. It is interesting to discuss the memory overhead caused by these few surviving weights. Table 2 reports an average of 1.7 bit per weight in this configuration. Note how a few surviving weight parameters can increase by %70 the memory overhead compared to pure binarization (1.0 bit per weight), as storing a full-precision weight costs 32-bit, and storing its position costs around 20 bits per non-binary entry. Hence, it is crucial to carefully select those weights that are worthed to keep in full-precision to yield competitive results with quantization. In future work, we plan to investigate more advances solutions to reduce the overhead of a single entry, both by reducing the bit width of its representation and by using delta encoding for the positions.

5.3. Efficiency evaluation

In this Section, we provide an extensive efficiency evaluation of our low-bits matrix multiplication routines and of APB-compressed networks. First, we assess the performance of our low-bits matrix multiplication routines, comparing them against state-of-the-art dense matrix multiplication frameworks. Then, we show that our APB compressed networks outperform existing quantization methods in terms of efficiency-accuracy trade-off.

Low-bits Matrix Multiplication. We now compare the efficiency of our Matrix Multiplication (MM) routines, i.e., 1/1, 1/2, 2/2, against state-of-the-art, highly optimized MM CPU libraries. In detail, we measure the achieved GFLOPs at different sizes of the matrices and we compare them to their *tpp*. *tpp* is computed according to Equation (14) and it is reported in Table 4 as dotted lines. Experiments are conducted on an Intel Xeon Gold 5318Y CPU, clocked at 3.4 GHz and equipped with the AVX-512 instruction set. Our novel inference routines are written in C++ and compiled with the -O3 option using GCC 11.2.0, with single-threaded execution. Table 4 reports an experimental comparison of square matrices. In the comparison, we include the performance of dense matrix multiplication as implemented in the oneDNN library. This library is the state-of-the-art solution for dense multiplication, employing industrial-level optimizations such as Just In Time (JIT) code compilation. Table 4 shows a general trend: large enough matrices allow all the tested algorithms, i.e., including the oneDNN library for dense MM, to achieve their best performance and get close to their *tpp*. The experimental results show that the 1/1, 1/2, and 2/2 routines reach up to 91%, 88%, and 77% of their theoretical peak performance, respectively. oneDNN GFLOPs peak at 87%. The results reported in Table 4 confirm that our bitwise multiplication routines are properly implemented, given that the gap with *tpp* is at most around the 20%. For the 1/2, and 2/2 cases the performance starts to decrease at $M = K = N = 2048$. This aspect can be tackled by implementing further optimizations, in particular to implement the so-called blocking strategy, which allows dealing with the memory-bounded nature of matrix multiplication. Further optimizations can be applied to our routines, such as micro-kernel parameters optimization according to the CPU architecture. These are optimization strategies that are commonly applied in the dense case [44]. Although interesting, these optimizations go beyond the scope of this work, and we leave their investigation as future work.

Square matrices allow MM routines to get close to their *tpp* at any bit width. Indeed, in DNN inference, rectangular matrices are much more common than squared ones. To evaluate the efficiency of our novel matrix multiplication algorithms on a real use case, we test them on the shapes obtained by applying the *im2col* transformation on the layers of ResNet-like architectures. We include the 8/8 MM in our analysis by employing the implementation provided by the FBGEMM [15] library. Results are reported in Fig. 6, where the x-axis indicates the shapes of the matrices under evaluation and the y-axis reports the speedup compared to the dense MM. We adopt two different implementations for dense MM. The first one is OneDNN, as in Table 4. BLIS [48] is an open-source GEMM library with assembly-level architecture-tailored optimizations and presents an optimization level closer to our C++ implementation. Observe that, on the shapes under evaluation, oneDNN is 60% faster than BLIS, which is a remarkable speedup considering that they employ the same algorithm for matrix multiplication.

Fig. 6 shows that our novel MM routines are significantly faster than dense multiplication. For example, the 1/1 routine delivers up to 15× speedup compared to oneDNN and up to 25× compared to BLIS. In this case, we witness a speedup larger than the theoretically estimated one, i.e., 10.5×. This is caused by the poor performance of dense MM on rectangular-shaped matrices [46]. In

Table 4
GFLOPs performance analysis of our novel bitwise MM routines against the oneDNN library used for dense MM. The analysis is performed on square matrices, i.e., $M = K = N$, where M is the number of rows of the first operand, K is the shared dimension, and N is the number of columns of the second operand. For each method, we report the *tpp* (theoretical peak throughput).

Method	*tpp*	Size ($M = K = N$)					
		64	128	256	512	1024	2048
		GFLOPs					
oneDNN	109	39	44	71	72	94	95
1 / 1	1160	264	404	899	959	1044	1060
1 / 2	580	154	213	418	468	496	259
2 / 2	145	28	36	77	95	112	85

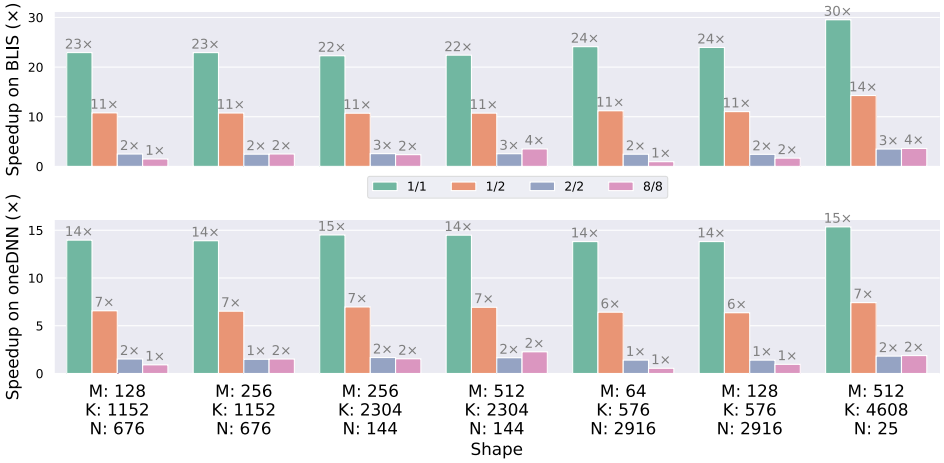

Fig. 6. Comparison between matrix multiplication routines at different quantization levels. M is the number of rows of the first operand, K is the shared dimension, and N is the number of columns of the second operand.

fact, rectangular matrices do not allow fully masking the memory-bounded nature of the MM operation.[14] This is evident in Fig. 6, where, on the shapes under evaluation, oneDNN reaches between the 50% and the 70% of *tpp*. Conversely, 1/1, 1/2 and 2/2 deliver respectively at least the 85%, 80% and 75% of their theoretical peak performance; The reason is that the bit width reduction provided by quantization mitigates memory-related issues when using the shapes under consideration.

Overall, the forward pass on ResNet-18 achieves 6.85× and 1.5× speedup compared to FBGEMM, the state-of-the-art MM library for quantized models on CPU, when employing the 1/2 and 2/2 MM routines, respectively. To the best of our knowledge, FBGEMM is the best available solution for MM at low bit widths, excluding our novel routines. Our solution allows, for the first time, to efficiently exploit the plethora of different quantization methods for 1/2 and 2/2 directly on CPU. We also observe that the performance ratio between different low-bits implementations precisely matches the predictions we made in Section 4: the 1/2 quantization schema is 2× slower than the 1/1, while the 2/2 is 8× slower.

Efficiency/Accuracy Trade-offs. We now compare the efficiency-accuracy trade-off that different low-bits quantization methods deliver compared to `APB`. We perform the analysis using the ResNet-18 architecture on the ImageNet dataset. We compute the inference time as the total MM time. The time spent on the first and the last layer, as well as on the batch-normalization and down-sampling layers, is ignored as these layers are not compressed in any of the approaches — as in Table 2. The comparison involves networks quantized with the 1/1, 1/2, and 2/2 configurations. The 1/1 accuracy is achieved by using SA-BNN [24], while 1/2 and 2/2 are achieved by using EWGS [9]. Both methods are state-of-the-art low-bit quantization approaches. We do not include methods such as ReActNet [26] or ElasticLink [27] that leverage custom architectures. We observe that `APB` could be easily employed in conjunction with these approaches, which is also part of our future work. In this analysis, `APB` is used to compress the weights of the network, while activations are quantized to 2 bits using the EWGS quantizer. We also experiment `APB` with 1-bit quantized activations, and we achieved worse performance than 1/2 quantization. This aspect is detailed in the subsequent paragraph. The inference time reported for `APB` is the sum of the time for 1/2 MM and the time for sparse-dense MM, as shown in Equation (13). LIBXSMM [43] is used for sparse-dense matrix multiplication.

Fig. 7 reports the results of the comparison in terms of speedup w.r.t. both BLIS (lower x-axis) and oneDNN (upper x-axis) and Top1 accuracy (y-axis). `APB` (green dots) largely dominates over 2/2 quantization provided by EWGS (orange dot). Our approach, in fact,

[14] https://en.algorithmica.org/hpc/algorithms/matmul/.

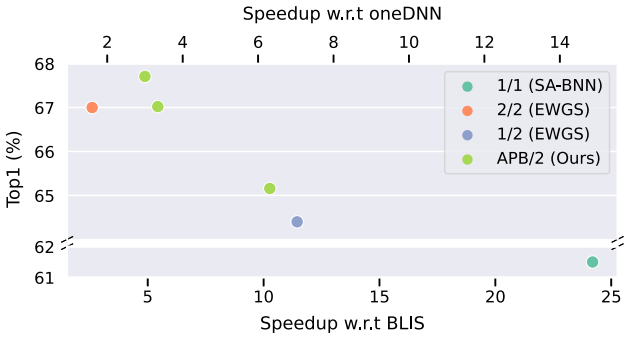

Fig. 7. Efficiency/Accuracy trade-offs of ResNet-18 on ImageNet using different quantization schema.

can yield a faster and sometimes more effective model compared to pure 2-bit quantization. The sparsity of surviving full-precision values per layer spans in 96-99%. In this range, we experimentally verify that sparse MM is always faster than the 2/2 routine, it matches the performance of 1/2 at about 97%, and it is faster than 1/1 at 99%. Moreover, thanks to our novel MM routine, the 1/2 models offer more than 11× / 7× speedup (BLIS/oneDNN) with a performance degradation of about 8% w.r.t. the full-precision model. Regarding 1/1, its extreme speedup asks for a significant accuracy degradation, i.e., 7 points of Top1 accuracy compared to the original full-precision model.

Comparison with pure binarization. We experiment with the combination of APB with 1-bit activations. We observe that the performance of our models outperforms by margin pure binary networks, by reaching 63.0 of Top1 accuracy. Anyway, this effectiveness improvement comes at the price of an elevated number of surviving full-precision weights, reaching 8% in some cases. The cost of these full-precision weights at inference time matches or sometimes surpasses the cost of binary multiplication. In practice, this means that the 1/2 scenario offers superior performance in terms of efficiency/accuracy tradeoff. This is coherent with the network quantization literature, where different works prove that reducing the precision of the activation worsens the effectiveness of the models more than reducing the precision of the weights.

6. Conclusions and future work

We proposed APB, a novel compression technique that merges binarization and pruning together to exploit the benefits provided by these two orthogonal techniques. We showed that APB jointly maximizes the accuracy achieved by the network while minimizing its memory impact by identifying an optimal partition of the network parameters among these two sets. Furthermore, we presented two novel matrix multiplication algorithms for extreme low-bit configurations, namely 1/2 and 2/2, where 1/2 refers to binary weights and 2-bit activations, while in the 2/2 configuration both weights and activations are quantized to 2 bits. We performed a comprehensive experimental evaluation on two widely adopted benchmark datasets, i.e., CIFAR-10 and ImageNet. Experiments show that APB achieves better accuracy/memory trade-off w.r.t. state-of-the-art compression methods based on i) quantization, ii) pruning, and, iii) the combination of pruning and quantization. Our novel matrix multiplication routines deliver a major speedup compared to the existing solution for low-bits matrix multiplication on CPU, ranging from 6.9× for the 1/2 configuration to 1.5× for the 2/2 configuration. Moreover, the experimental results show that APB is 2× faster than the 2-bit quantized model with no loss in accuracy. On the one hand, our novel matrix algorithms open up to exploiting quantized networks on the CPU. Also, they may boost the investigation of 1/2 quantization scenario, given that a very fast inference engine is available. On the other hand, we show that APB-compressed networks, where binary and full-precision weights are mixed in the same weight tensor, allow for better performance compared to fixed quantization, e.g., 2-bits. The importance of a reduced portion of full-precision weights, evidenced by pruning techniques in previous work, is stressed again in this new hybrid format.

Future Work. We plan to improve APB further to automatically identify the optimal quantization schema for each layer activation to improve the efficiency/accuracy trade-off. This would be possible due to the availability of efficient matrix multiplication routines covering several quantization schemas. Moreover, we are interested in applying APB in conjunction with highly effective custom convolutional architectures, such as ElasticLink [27].

The application of the innovations proposed in this paper beyond the convolutional neural networks is a compelling task. For example, several recent works, including Huang et al. [49] and Du et al. [50], have focused on training, fine-tuning, and evaluating transformer-based architectures, including Large Language Models, in extremely low-bit scenarios. Our new routines are suitable candidates to exploit in practice the bit width reduction provided by methods, especially when deploying the models on the CPU. In this perspective, we plan to test APB as a compression framework on attention-based architectures to compare its performance with the aforementioned pure quantization solutions. The major limitation of the proposed method is its limited applicability to CPUs, which are widely adopted in edge scenarios but are commonly replaced by GPUs or ASICs on server-side computation. For this purpose, we also plan to extend APB to be suitable for GPUs, taking into account the specific features of this computing platform.

CRediT authorship contribution statement

Franco Maria Nardini: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Resources, Methodology, Investigation, Funding acquisition, Conceptualization. **Cosimo Rulli:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Salvatore Trani:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Methodology, Investigation, Funding acquisition, Conceptualization. **Rossano Venturini:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partially supported by the Horizon Europe RIA “Extreme Food Risk Analytics” (EFRA), grant agreement n. 101093026, by the PNRR - M4C2 - Investimento 1.3, Partenariato Esteso PE00000013 - “FAIR - Future Artificial Intelligence Research” - Spoke 1 “Human-centered AI” funded by the European Commission under the NextGeneration EU program, by the PNRR ECS00000017 Tuscany Health Ecosystem Spoke 6 “Precision medicine & personalized healthcare” funded by the European Commission under the NextGeneration EU programme, and by the MUR-PRIN 2022 “Algorithmic Problems and Machine Learning”, grant agreement n. 20229BCXNW.

Data availability

Data is publicity available. The source code will be made available upon publication.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27–30, 2016, IEEE Computer Society, 2016, pp. 770–778.

[2] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, Ross B. Girshick, Masked autoencoders are scalable vision learners, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18–24, 2022, IEEE, 2022, pp. 15979–15988.

[3] Mingxing Tan, Ruoming Pang, Quoc V. Le, Efficientdet: scalable and efficient object detection, in: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13–19, 2020, Computer Vision Foundation / IEEE, 2020, pp. 10778–10787, https://openaccess.thecvf.com/content_CVPR_2020/html/, Tan_EfficientDet_Scalable_and_Efficient_Object_Detection_CVPR_2020_paper.html.

[4] Sébastien Bubeck, Mark Sellke, A universal law of robustness via isoperimetry, in: Marc’Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, Jennifer Wortman Vaughan (Eds.), Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, Virtual, December 6–14, 2021, 2021, pp. 28811–28822, https://proceedings.neurips.cc/paper/2021/hash/f197002b9a0853eca5e046d9ca4663d5-Abstract.html.

[5] Cristian Buciluǎ, Rich Caruana, Alexandru Niculescu-Mizil, Model compression, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2006, pp. 535–541.

[6] Yoshua Bengio, Nicholas Léonard, Aaron C. Courville, Estimating or propagating gradients through stochastic neurons for conditional computation, CoRR, arXiv:1308.3432, 2013, http://arxiv.org/abs/1308.3432.

[7] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, Ali Farhadi, Xnor-net: imagenet classification using binary convolutional neural networks, in: Bastian Leibe, Jiri Matas, Nicu Sebe, Max Welling (Eds.), Computer Vision - ECCV 2016 - 14th European Conference, Proceedings, Part IV, Amsterdam, the Netherlands, October 11–14, 2016, in: Lecture Notes in Computer Science, vol. 9908, Springer, 2016, pp. 525–542.

[8] Zhaohui Yang, Yunhe Wang, Kai Han, Chunjing Xu, Chao Xu, Dacheng Tao, Chang Xu, Searching for low-bit weights in quantized neural networks, in: Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, Hsuan-Tien Lin (Eds.), Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, Virtual, December 6–12, 2020, 2020, https://proceedings.neurips.cc/paper/2020/hash/2a084e55c87b1ebcdaad1f62fdbbac8e-Abstract.html.

[9] Junghyup Lee, Dohyung Kim, Bumsub Ham, Network quantization with element-wise gradient scaling, in: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, Virtual, 2021, June 19–25, Computer Vision Foundation / IEEE, 2021, pp. 6448–6457, https://openaccess.thecvf.com/content/CVPR2021/html/.

[10] Victor Sanh, Thomas Wolf, Alexander M. Rush, Movement pruning: adaptive sparsity by fine-tuning, in: Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, Hsuan-Tien Lin (Eds.), Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, Virtual, December 6–12, 2020, 2020, https://proceedings.neurips.cc/paper/2020/hash/eae15aabaa768ae4a5993a8a4f4fa6e4-Abstract.html.

[11] Jonathan Frankle, Michael Carbin, The lottery ticket hypothesis: finding sparse, trainable neural networks, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6–9, 2019, OpenReview.net, 2019, https://openreview.net/forum?id=rJl-b3RcF7.

[12] Mart Van Baalen, Christos Louizos, Markus Nagel, Rana Ali Amjad, Ying Wang, Tijmen Blankevoort, Max Welling, Bayesian bits: unifying quantization and pruning, Adv. Neural Inf. Process. Syst. 33 (2020) 5741–5752.

[13] James Diffenderfer, Bhavya Kailkhura, Multi-prize lottery ticket hypothesis: finding accurate binary neural networks by pruning a randomly weighted network, in: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3–7, 2021, OpenReview.net, 2021, https://openreview.net/forum?id=U_mat0b9iv.

[14] Jing Liu, Bohan Zhuang, Peng Chen, Chunhua Shen, Jianfei Cai, Mingkui Tan, Single-path bit sharing for automatic loss-aware model compression, IEEE Trans. Pattern Anal. Mach. Intell. (2023).

[15] Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, Mikhail Smelyanskiy, Fbgemm: enabling high-performance low-precision deep learning inference, arXiv preprint arXiv:2101.05615, 2021.
[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, Li Fei-Fei, Imagenet: a large-scale hierarchical image database, in: 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), Miami, Florida, USA, 20–25 June 2009, IEEE Computer Society, 2009, pp. 248–255.
[17] Muhammad Zawish, Steven Davy, Lizy Abraham, Complexity-driven model compression for resource-constrained deep learning on edge, IEEE Trans. Artif. Intell. (2024).
[18] Matthieu Courbariaux, Yoshua Bengio, Jean-Pierre David, Binaryconnect: training deep neural networks with binary weights during propagations, in: Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, Roman Garnett (Eds.), Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7–12, 2015, Montreal, Quebec, Canada, 2015, pp. 3123–3131, https://proceedings.neurips.cc/paper/2015/hash/3e15cc11f979ed25912dff5b0669f2cd-Abstract.html.
[19] Song Han, Jeff Pool, John Tran, William J. Dally, Learning both weights and connections for efficient neural network, in: Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, Roman Garnett (Eds.), Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7–12, 2015, Montreal, Quebec, Canada, 2015, pp. 1135–1143, https://proceedings.neurips.cc/paper/2015/hash/ae0eb3eed39d2bcef4622b2499a05fe6-Abstract.html.
[20] Ang Li, Simon Su, Accelerating binarized neural networks via bit-tensor-cores in Turing gpus, IEEE Trans. Parallel Distrib. Syst. 32 (7) (2021) 1878–1891, https://doi.org/10.1109/TPDS.2020.3045828.
[21] Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, Jingkuan Song, Forward and backward information retention for accurate binary neural networks, in: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13–19, 2020, Computer Vision Foundation / IEEE, 2020, pp. 2247–2256, https://openaccess.thecvf.com/content_CVPR_2020/html/, Qin_Forward_and_Backward_Information_Retention_for_Accurate_Binary_Neural_Networks_CVPR_2020_paper.html.
[22] Yunqiang Li, Silvia-Laura Pintea, Jan C. van Gemert, Equal bits: enforcing equally distributed binary network weights, in: Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, the Intelligence, EAAI 2022 Virtual Event, February 22 – March 1, 2022, AAAI Press, 2022, pp. 1491–1499.
[23] Zihan Xu, Mingbao Lin, Jianzhuang Liu, Jie Chen, Ling Shao, Yue Gao, Yonghong Tian, Rongrong Ji, Recu: reviving the dead weights in binary neural networks, in: 2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10–17, 2021, IEEE, 2021, pp. 5178–5188.
[24] Chunlei Liu, Peng Chen, Bohan Zhuang, Chunhua Shen, Baochang Zhang, Wenrui Ding, SA-BNN: state-aware binary neural network, in: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, the Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2–9, 2021, AAAI Press, 2021, pp. 2091–2099.
[25] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, Kwang-Ting Cheng, Bi-real net: enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm, in: Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, Yair Weiss (Eds.), Computer Vision - ECCV 2018 - 15th European Conference, Proceedings, Part XV, Munich, Germany, September 8–14, 2018, in: Lecture Notes in Computer Science, vol. 11219, Springer, 2018, pp. 747–763.
[26] Zechun Liu, Zhiqiang Shen, Marios Savvides, Kwang-Ting Cheng, Reactnet: towards precise binary neural network with generalized activation functions, in: Andrea Vedaldi, Horst Bischof, Thomas Brox, Jan-Michael Frahm (Eds.), Computer Vision - ECCV 2020 - 16th European Conference, Proceedings, Part XIV, Glasgow, UK, August 23–28, 2020, in: Lecture Notes in Computer Science, vol. 12359, Springer, 2020, pp. 143–159.
[27] Jie Hu, Ziheng Wu, Vince Junkai Tan, Zhilin Lu, Mengze Zeng, Enhua Wu, Elastic-link for binarized neural networks, in: Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, the Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 – March 1, 2022, AAAI Press, 2022, pp. 942–950.
[28] Haotong Qin, Mingyuan Zhang, Yifu Ding, Aoyu Li, Zhongang Cai, Ziwei Liu, Fisher Yu, Xianglong Liu, Bibench: benchmarking and analyzing network binarization, in: International Conference on Machine Learning, ICML 2023, 23–29 July 2023, Honolulu, Hawaii, USA, 2023, pp. 28351–28388.
[29] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, Dan Alistarh, OPTQ: accurate quantization for generative pre-trained transformers, in: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1–5, 2023, OpenReview.net, 2023, https://openreview.net/forum?id=tcbBPnfwxS.
[30] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, Luke Zettlemoyer, Qlora: efficient finetuning of quantized llms, in: Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, Sergey Levine (Eds.), Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10–16, 2023, 2023, http://papers.nips.cc/paper_files/paper/2023/hash/1feb87871436031bdc0f2beaa62a049b-Abstract-Conference.html.
[31] Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, Deyi Xiong, A comprehensive evaluation of quantization strategies for large language models, in: Lun-Wei Ku, Andre Martins, Vivek Srikumar (Eds.), Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and Virtual Meeting, 2024, August 11–16, Association for Computational Linguistics, 2024, pp. 12186–12215, https://aclanthology.org/2024.findings-acl.726.
[32] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, Xiaotong Zhang, Pruning and quantization for deep neural network acceleration: a survey, Neurocomputing 461 (2021) 370–403, https://doi.org/10.1016/J.NEUCOM.2021.07.045.
[33] Song Han, Huizi Mao, William J. Dally, Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding, in: Yoshua Bengio, Yann LeCun (Eds.), 4th International Conference on Learning Representations, ICLR 2016, Conference Track Proceedings, San Juan, Puerto Rico, May 2–4, 2016, 2016, http://arxiv.org/abs/1510.00149.
[34] Alex Renda, Jonathan Frankle, Michael Carbin, Comparing rewinding and fine-tuning in neural network pruning, in: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26–30, 2020, OpenReview.net, 2020, https://openreview.net/forum?id=S1gSj0NKvB.
[35] Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, Dan Alistarh, Spqr: a sparse-quantized representation for near-lossless LLM weight compression, in: The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7–11, 2024, OpenReview.net, 2024, https://openreview.net/forum?id=Q1u25ahSuy.
[36] Yuwei Hu, Jidong Zhai, Dinghua Li, Yifan Gong, Yuhao Zhu, Wei Liu, Lei Su, Jiangming Jin, Bitflow: exploiting vector parallelism for binary neural networks on CPU, in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, IEEE Computer Society, Vancouver, BC, Canada, May 21–25, 2018, pp. 244–253.
[37] Jianhao Zhang, Yingwei Pan, Ting Yao, He Zhao, Tao Mei, dabnn: a super fast inference framework for binary neural networks on ARM devices, in: Laurent Amsaleg, Benoit Huet, Martha A. Larson, Guillaume Gravier, Hayley Hung, Chong-Wah Ngo, Wei Tsang Ooi (Eds.), Proceedings of the 27th ACM International Conference on Multimedia, MM 2019, Nice, France, October 21–25, 2019, ACM, 2019, pp. 2272–2275.
[38] Lukas Geiger, Plumerai Team, Larq: an open-source library for training binarized neural networks, J. Open Source Softw. 5 (45) (2020) 1746, https://doi.org/10.21105/JOSS.01746.
[39] Trevor Gale, Erich Elsen, Sara Hooker, The state of sparsity in deep neural networks, CoRR, arXiv:1902.09574, 2019, http://arxiv.org/abs/1902.09574.
[40] Peisong Wang, Xiangyu He, Gang Li, Tianli Zhao, Jian Cheng, Sparsity-inducing binarized neural networks, in: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, the Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, the Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7–12, 2020, AAAI Press, 2020, pp. 12192–12199.
[41] David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, Learning representations by back-propagating errors, Nature 323 (6088) (1986) 533–536.
[42] Kumar Chellapilla, Sidd Puri, Patrice Simard, High performance convolutional neural networks for document processing, in: International Workshop on Frontiers in Handwriting Recognition, Suvisoft, 2006.

[43] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, Hans Pabst, LIBXSMM: accelerating small matrix multiplications by runtime code generation, in: John West, Cherri M. Pancake (Eds.), Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13–18, 2016, IEEE Computer Society, 2016, pp. 981–991.
[44] Kazushige Goto, Robert A. van de Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Softw. 34 (3) (2008) 12, https://doi.org/10.1145/1356052.1356053.
[45] Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, Junjie Yan, Differentiable soft quantization: bridging full-precision and low-bit neural networks, in: 2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 – November 2, 2019, IEEE, 2019, pp. 4851–4860.
[46] Franco Maria Nardini, Cosimo Rulli, Salvatore Trani, Rossano Venturini, Distilled neural networks for efficient learning to rank, IEEE Trans. Knowl. Data Eng. 35 (5) (2023) 4695–4712, https://doi.org/10.1109/TKDE.2022.3152585.
[47] Karen Simonyan, Andrew Zisserman, Very deep convolutional networks for large-scale image recognition, in: Yoshua Bengio, Yann LeCun (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings, San Diego, CA, USA, May 7–9, 2015, 2015, http://arxiv.org/abs/1409.1556.
[48] Jianyu Huang, Robert A. van de Geijn, Blislab: a sandbox for optimizing GEMM, CoRR, arXiv:1609.00076, 2016, http://arxiv.org/abs/1609.00076.
[49] Wei Huang, Xingyu Zheng, Xudong Ma, Haotong Qin, Chengtao Lv, Hong Chen, Jie Luo, Xiaojuan Qi, Xianglong Liu, Michele Magno, An empirical study of llama3 quantization: from llms to mllms, https://arxiv.org/abs/2404.14047, 2024.
[50] Dayou Du, Yijia Zhang, Shijie Cao, Jiaqi Guo, Ting Cao, Xiaowen Chu, Ningyi Xu, Bitdistiller: unleashing the potential of sub-4-bit llms via self-distillation, https://arxiv.org/abs/2402.10631, 2024.

Franco Maria Nardini is currently a Senior Researcher with ISTI-CNR, Pisa, Italy. His research interests include web information retrieval, machine learning, and data mining. He is a member of the Editorial Board of ACM Transactions on Information Systems and a Program Committee Member of SIGIR, ECIR, SIGKDD, CIKM, WSDM, IJCAI, and ECMLPKDD. He was a co-recipient of the ECIR 2025 Best Short Student Paper Award, the ACM SIGIR 2024 Best Paper Runner-Up Award, the ECIR 2022 Industry Impact Award, the ACM SIGIR 2015 Best Paper Award, the ECIR 2014 Best Demo Paper Award. He has been General Co-Chair of ECIR 2025, Program Committee Co-Chair of SPIRE 2023, Tutorial Co-Chair of ACM WSDM 2021, Demo Papers Co-Chair of ECIR 2021.

Cosimo Rulli is a researcher with the National Research Council of Italy. He received the Ph.D. degree in 2023, with a thesis on deep neural network compression. His research interests include deep learning, model compression, and efficiency in information retrieval. He was a co-recipient of the ACM SIGIR 2024 Best Paper Runner-Up Award and the ECIR 2025 Best Student Short Paper Award. He is a Reviewer of ACM TOIS, IEEE TKDE, and PMC, and he is a Committee Member of SIGIR, ECIR, CIKM, and WSDM.

Salvatore Trani is a Senior Researcher with the National Research Council of Italy. He received his Ph.D. in Computer Science from the University of Pisa in 2017. His research interests range from Information Retrieval to Data Mining and Machine Learning. He has authored more than 15 papers on these topics, published in peer-reviewed international journals and conferences. He is a Reviewer of ACM TOIS, IEEE TKDE, and Machine Learning and a Program Committee Member of SIGIR, ECIR, SIGKDD, CIKM, WSDM, ECAI, AAAI, and ICML. He has been Proceeding Chair of ECIR 2025.

Rossano Venturini received his Ph.D. degree from the University of Pisa, in 2010. He is an Associate Professor at the Computer Science Department of the University of Pisa. His research interests mainly focus on designing and analyzing algorithms and data structures for large datasets with applications in Information Retrieval and Machine Learning. He received the Best Paper Awards at ACM SIGIR in 2014 and 2015, the ACM SIGIR 2024 Best Paper Runner-Up Award and the ECIR 2025 Best Student Short Paper Award.