Mortar-FP8: Morphing the Existing FP32 Infrastructure for High-Performance Deep Learning Acceleration

Hongyan Li, Hang Lu, and Xiaowei Li, *Senior Member, IEEE*

***Abstract*—Vanilla deep neural networks (DNNs) after training are represented with native floating-point 32 (fp32) weights. We observe that the bit-level sparsity of these weights is very abundant in the mantissa and the distribution of exponent is aggregated, which can all be directly exploited to speed up model inference. In this article, we propose Mortar and Mortar-FP8, the offline/online software and hardware collaborative approaches for fp32 DNN acceleration. The proposed methods include the software algorithms to morph the mantissa and convert fp32 weights to fp8 format, as well as associated hardware accelerator architecture to accelerate general-purpose deep learning through optimized algorithm and specialized hardware. We highlight the following results by evaluating various deep learning tasks, including image classification, object detection, video understanding, video, and image super-resolution: 1) Mortar increase mantissa sparsity up to 1.58×–2.09× with only a negligible ∼0.2% accuracy loss; 2) Mortar-FP8 morph the fp32 weights to fp8 format with a minimal accuracy loss of ∼0.3%; and 3) the corresponding hardware accelerator significantly outperforms baselines, achieving up to 6.032× and 6.99× performance improvements. The area and power of Mortar are 0.031 mm^2 and 68.58 mW. Those metrics are 0.0505 mm^2 and 25.16 mW for Mortar-FP8.**

***Index Terms*—Deep learning accelerator, deep neural network (DNN), fp8 format.**

I. Introduction

DEEP learning has made significant progress in the past few years, bringing about a paradigm shift in numerous domains, such as computer vision, natural language processing, and speech recognition. Artificial neural networks have demonstrated remarkable success in solving complex problems and performing a variety of tasks with high accuracy.

As deep learning models become larger and more complex, accelerating their computations has become crucial to making them practical for real-world applications. Typically, deep neural networks (DNNs) are represented in floating-point 32 (fp32) precision because they exhibit high native model accuracy. However, typical fp32 DNNs are slow during computation compared to lower precision models such as fp16 or int8. Ideally, developers require a method that accurately represents vanilla models and obtains a fast inference speed that is on par with low-precision models.

From an architectural standpoint, certain existing DNN accelerators fail to optimize their designs specifically for fp32 precision, thereby disregarding the unique attributes of fp32 operands. General-purpose accelerators like TPU [4], KunLun [6], Enflame DTU [7], and MLU290 [8] predominantly employ conventional fp32 multiply-and-accumulation (MAC) operations as the fundamental micro-operation for conducting convolutions and matrix multiplications. However, this cumbersome floating-point arithmetic inevitably hampers the speed of inference. It is worth highlighting that these mentioned accelerators have gradually started supporting various reduced precision formats, such as INT8, for computational purposes. Furthermore, some accelerators even provide support for formats like FP8 and BF16. Leveraging these reduced precision formats allows accelerators to optimize the speed of inference.

The utilization of reduced precision techniques in deep learning has gained significant attention due to their potential to mitigate memory requirements, computational complexity, and energy consumption. With the exponential growth of DNNs and the increasing need for efficient deployment on resource-constrained devices, researchers have been motivated to explore lower-bit precision formats in both algorithm design and hardware implementation.

One such format is fp8, which employs 8-bit floating-point numbers to represent parameters in neural networks. This approach offers a means of conserving energy consumption of by reducing the storage resources by up to 75% when compared to the traditional fp32 representation [9]. However, the reduction in bit width of floating point in the fp8 format poses a significant challenge as it impacts the precision and range of floating-point digits. Notably, techniques, such as S2FP8 [10], hybrid fp8 [11], and loss scaling [12], have been introduced to address this issue. Micikevicius et al. [13] suggested two encodings: E4M3 and E5M2, for fp8 formats. These techniques aim to mitigate the impact of reduced bit width and ensure the network performance during computation in the fp8 format.

Manuscript received 1 May 2023; revised 28 July 2023 and 8 October 2023; accepted 28 October 2023. Date of publication 6 November 2023; date of current version 21 February 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62172387, and in part by the Youth Innovation Promotion Association of Chinese Academy of Sciences (CAS) under Grant 2021098. An earlier version of this article was presented in part at the 28th Asia and South Pacific Design Automation Conference [DOI: 10.1145/3566097.3567868]. This article was recommended by Associate Editor M. Shafique. *(Corresponding author: Hang Lu.)*

Hongyan Li and Xiaowei Li are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China.

Hang Lu is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, the Zhongguancun Laboratory, and the Shanghai Innovation Center for Processor Technologies, Chinese Academy of Sciences, Beijing 100190, China (e-mail: luhang@ict.ac.cn).

Digital Object Identifier 10.1109/TCAD.2023.3329778

Overall, these approaches show the potential benefits of fp8 computing in accelerating deep learning computations. However, most of the proposed methods do not consider the ease of implementation in hardware design. Some existing methods require complex tricks such as loss scaling [12] or log computation [10] to fit the representation range of fp8 into the representation range of fp32, which is not easy to implement on hardware.

To address the issue, in this article, we propose the software and hardware co-design method for representing fp32 representation into fp8 format. By using simple and hardware-friendly operations, fp32 weights can be represented as fp8, accelerating the computation of neural networks without compromising network performance, and reducing the storage size of neural networks.

This article introduces Mortar and Mortar-FP8 as novel DNN acceleration methodologies for faster and higher-performing inference during deep learning.

Aiming to reduce the mantissa length, Mortar reorganizes the mantissa of floating-point numbers by eliminating less important bit 1s. This technique complements accuracy loss and shortens the valid mantissa bit length. Building upon the principles of Mortar, Mortar-FP8 extends its scope by not only reducing the mantissa length but also narrowing the exponent width. As a result, Mortar-FP8 processes fp32 weights to only 8 bits. By processing fp32 weights to only 8 bits, Mortar-FP8 achieves a substantial reduction in bit width, leading to a significant decrease in computational cost and memory requirements for floating-point operations. It is worth noting that Mortar-FP8 employs the straightforward optimization techniques to manipulate the target representation efficiently. And the proposed accelerator utilizes low-cost hardware architectures, making it efficient and practical for resource-constrained environments.

The proposed approaches offer the practical solutions for implementing model accelerating techniques and low bit precision in hardware design, which is essential for the efficient deployment of neural networks. Overall, Mortar and Mortar-FP8 provide innovative solutions for accelerating DNN inference while maintaining accuracy and minimizing the use of resources. This article makes the following contributions.

1) We propose Mortar and Mortar-FP8, two novel software and hardware collaborative approaches for efficient general-purpose deep learning acceleration. The associated hardware accelerator architecture is as well as designed. Our method targets the abundant mantissa redundancy and aggregated distribution of exponent presented in fp32 weights in neural networks, allowing for parameter reduction to an 8-bit length, creating more efficient methodologies and hardware-friendly accelerator.
2) To evaluate the proposed approach, we conducted a thorough comparison with several state-of-the-art (SOTA) baselines, The following results are highlighted.

Mortar achieves 1.58×–2.09× sparsity improvement with negligible model accuracy loss of 0.2%. Compared with baseline BitX [14], Mortar can achieve higher accuracy while improving average 1.30× bit-sparsity than BitX. Mortar-FP8 achieves even greater reductions in computational costs, with a drastic reduction of the mantissa bit width from 23 to an average of 2.25 bits, resulting in a total bit width decrease to just 8 bits. The accuracy loss is minimal and barely noticeable.

Compared with other SOTA accelerators, Mortar achieves 4.607× and 6.032× performance improvement over Pragmatic [15] as the baseline. Mortar-FP8 achieves greater efficiency with 6.99× and 6.5× performance enhancement. The area and power consumption of Mortar-FP8 are 0.0505 mm^2 and 25.16 mW, respectively. And the area and power of Mortar are 0.031 mm^2 and 68.58 mW.

This article presents the journal extension version of our work at the Asia and South Pacific Design Automation Conference (ASP-DAC) 2023 [16]. In this version, we have significantly expanded our research, with particular focus on enhancing the offline Mortar method for the Mortar-FP8 converting algorithm. By employing this algorithm, we are able to effectively reduce the fp32 weights to an fp8 width, resulting in a narrower mantissa targeted in Mortar and a more efficient exponent width. Consequently, we achieve a highly efficient algorithm for accelerating neural network computation. Moreover, our study includes an in-depth analysis of the performance implications associated with the Mortar-FP8 conversion technique. As part of our contribution, we introduce an updated accelerator that effectively utilizes the Mortar-FP8. Based on the comprehensive evaluation results, we demonstrate the remarkable effectiveness and efficiency of our proposed approach.

This article is structured as follows. Section II provides the related work. Section III outlines the potential of our proposed method. Section IV presents the methodology employed in this study. Section V describes the hardware architecture developed for the implementation. In Section VI, we present the experimental results of both Mortar and Mortar-FP8, accompanied by thorough analyses. Finally, Section VII summarizes the work and provides the conclusion.

II. Related Work

A. Reduced Precision in Neural Networks

In recent years, reduced precision techniques have received considerable attention as effective means of improving the efficiency and computational performance of neural networks. Various floating-point formats are predominantly used in deep learning, such as fp32 and bfloat16. Additionally, researchers have explored extreme bit-reduction approaches in their work. For example, [17] proposed BinaryConnect, which trains networks with binary weights, achieving impressive results with only 1-bit weight precision. Similarly, [18] introduced the XNOR-Net model, which utilizes binary weights and activations. However, these approaches often suffer from accuracy degradation due to the highly limited representation capacity. Other formats, such as int8 [19] and log format [20], have also been proposed in quantization research studies, but some of them may encounter challenges in terms of accuracy or hardware deployment. Additionally, some neural network quantization methods require additional steps for target data mapping and cannot be directly quantized through online processing on hardware.

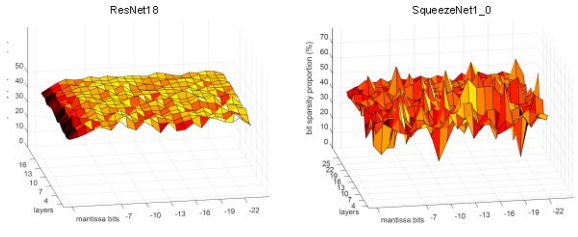

Fig. 1. Bit-level mantissa sparsity of fp32 weights for different DNNs pretrained with ImageNet.

B. FP8 Format

The FP8 format has gained growing interest in exploring lower-precision floating-point formats, aiming to strike a balance between accuracy and computational efficiency. It offers advantages compared to previously proposed formats [13]. Chunk-based accumulation and stochastic rounding were introduced in [9] to mitigate the problem of swamping [10]. Additionally, [11] put forth a hybrid 8-bit floating-point approach for training DNNs. S2FP8 [12] presented a novel method involving the storage of N fp8 values accompanied by two factors (squeeze and shift). Furthermore, [13] proposed a technique called loss scaling to ensure weights fit within the fp8 range. In pursuit of the tradeoff between accuracy and data representation, [9] put forward multilevel scaling. While these approaches contribute valuable insights, it is noteworthy that some of them may not thoroughly consider the ease of hardware implementation. To bridge this gap, this article introduces Mortar-FP8, a technique specifically designed to facilitate the conversion of weights into fp8 formats by ensuring hardware-friendliness.

III. Opportunity of Direct Conversion From FP32 to FP8

A. Mantissa Redundancy

Without loss of generality, a floating-point operand following IEEE-754 [21] standards consists of three parts: 1) signed bit (S); 2) mantissa (M); and 3) exponent (E). Bit sparsity is present when not all bits in the representation are 1. The mantissa, which is 23 bits long, exhibits unique features that can be used for general-purpose acceleration due to its redundant bit sparsity. Fig. 1 displays the bit sparsity distribution of mantissa in vanilla DNN weights, with the x-axis representing the mantissa bit positions, the y-axis representing sequential layers, and the z-axis displaying the bit sparsity proportion calculated as the percentage of "1" bits over the total number of binary bits. The figure demonstrates that the sparsity distribution is uniform ($\sim$50%) throughout all bit positions because each bit has an equal probability of being 0 or 1, resulting in abundant mantissa bit sparsity.

Targeting the bit-level sparsity for DNN acceleration is not a new idea. Many schemes in the literature have directly leveraged in-situ zero-bit skipping mechanisms to avoid ineffectual computations [15], [22], or special encoding methods to create more bit sparsity headroom [23], [24]. However, these approaches mostly focus on the fixed-point or integer operands. Very few works try to accelerate MACs by targeting the bit sparsity in the fp32 operands.

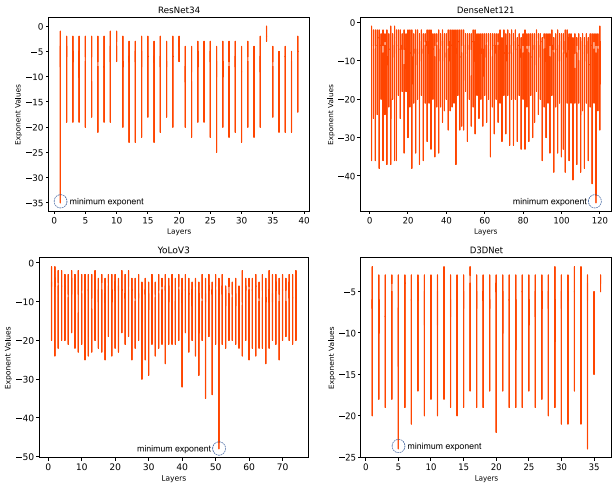

Fig. 2. Exponent distribution of fp32 weights in various networks.

Due to the nature of floating-point numbers, the significance of bits decreases from higher to lower positions in the mantissa. However, previous works rarely utilize the characteristic. From the figure above, lower significant bit positions have a similar bit sparsity, while playing only a negligible role in the final product. This observation implies a potential opportunity to compute fewer bit 1s while still maintaining target accuracy. By reorganizing the mantissa bits with each bit's significance, we can safely cut off several rear bit 1 s (setting rear bit 1 s to 0 s) to compensate for the change of the front bits. Utilizing bit sparsity helps to reduce the bit width of mantissa, resulting in a reduction in computational overheads of the DNN.

Therefore, reorganizing all layers in a DNN in this manner allows the accelerator only to compute the narrow essential bits, effectively reducing computational overhead. However, achieving this objective is complicated, involving an accuracy constraint and the associated hardware design. In Section IV, we will elaborate on how Mortar and Mortar-FP8 are designed for these purposes.

B. Aggregated Distribution of Exponent

The redundant bit sparsity characteristic of the mantissa in weights presents an opportunity to reduce the bit width of the weights. However, solely reducing the bit width of the mantissa cannot represent a 32-bit floating-point number using only 8 bits. Consequently, reducing the exponent's bit width is also necessary.

Since a small change in the exponent can result in a significant impact in the value of the weight, reducing the bit width of the exponent can be more challenging than reducing that of the mantissa. This is because decreasing the exponent's bit width may cause a significant loss of range, which can have a greater impact on the accuracy of the result compared to reducing the mantissa's bit width. To address this issue, we analyzed weight exponent numerical characteristics in neural networks.

Fig. 2 visualizes exponent distribution in neural networks, with layers on the x-axis, and the layer's weight exponent on the y-axis. As depicted in the Fig. 2, typically, the smallest

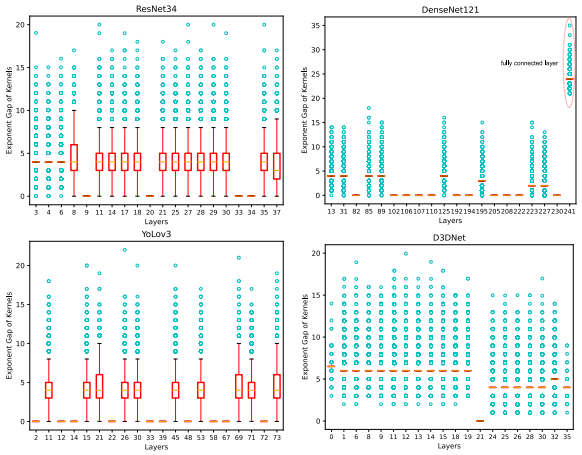

Fig. 3. Exponent gap of kernels in different networks, the gap refers to the subtraction between maximum and minimum of exponent in a kernel.

exponent in these networks is smaller than −30, and in YoLov3, the minimum weight is approximately −50. Since the exponent ranges from −50 to 0, using an 8-bit representation for the exponent would be inefficient. On the other hand, a 4 or 5-bit representation for exponents is insufficient. Finding a unified method mapping exponents to 4 or 5 bits while maintaining the accuracy of different layers is almost impossible, due to significant differences in the exponent distributions between different layers. Therefore, further optimization is required to successfully reduce the exponent's bit width while preserving model accuracy.

The number of bits allocated to the exponent determines the number of accurately represented exponents, rather than the exponent range. In our research, we examined the gap between maximum and minimum exponents, as it reflects the number of different exponents within the given range, consistent with the meaning of exponent bit width. Since different layers exhibit varying exponent distributions. Our study focused on a smaller granularity—convolution kernels.

Fig. 3 presents the boxplot of the distribution of kernel exponent gaps, where the x-axis represents the randomly selected 20 layers in the network and the y-axis denotes the exponent gap of kernels in each layer. The boxplot being closer to the bottom indicates that smaller gaps are more prevalent. The boxplot whiskers are mostly below 10, which means that most gaps are widely distributed around 10, and the majority of outliers are also below 15. Only a small portion of the outliers exceed 20.

Based on the boxplot, we can infer that the kernel exponent gap of varying networks all falls within the range of 0 to 15. Given that 4 bits can accurately represent 16 exponents, narrow bit widths such as 4 bits can be used to represent exponents by covering the gap range of every kernel. This provides an opportunity to shorten the bit width of the exponent. We will elaborate on how to achieve this in Section III.

However, in the boxplot of DenseNet121, there is a notable dissimilarity in the distribution of kernel gaps for last layer. The gaps of the last layer are distributed beyond 20 and can reach up to 35 at the highest. Our analysis revealed the layer is the fully connected layer. This gap exceeds the representation capacity using only four bits. Hence, for certain image recognition models, abstaining from processing the fully connected layer to fp8 may be necessary. This measure can safeguard against information loss and ensure model accuracy maintenance. Further about this will be found in Section VI.

C. General Concept

Regarding the floating point formats during computation, we consider a series of fp32 MACs in computing the partial sum of convolutions, and the expression [25] can be transformed from

$$\begin{aligned}\sum_{i=0}^{N-1} A_i \times W_i &= \sum_{i=0}^{N-1}\left[(-1)^{S_{w_i}} A_i \times M_{W_i} \times 2^{E_{w_i}}\right] \\ &= \sum_{i=0}^{N-1} \sum_{b=E_i-E_{\max}}^{E_i-E_{\max}-23} \left[(-1)^{S_{w_i}\oplus S_{A_i}} \cdot \left(M_{A_i} \times M_{W_i}^{b}\right)\right] \\ &\quad \times 2^{E_{\max}+b}. \end{aligned} \tag{1}$$

Hence, we can deduce that floating-point MAC operations can be decomposed into a sequence of bit-level operations on the corresponding mantissa and exponent components.

Our observations reveal that the presence of redundant mantissa bits and aggregated exponent can be effectively utilized to enhance the efficiency of DNN computations. By leveraging bit-level manipulation techniques, it becomes possible to achieve faster and more efficient computations by reducing the number of bits involved in the operations.

However, it is crucial to strike a balance when reducing the number of bits, as an excessive reduction can lead to compromised accuracy and restricted numerical range. This can ultimately result in a decline in network performance. Therefore, the carefully designed approach is needed to achieve an optimal tradeoff between computational speed and accuracy preservation. Subsequently, in the subsequent section, we shall delve into a detailed elaboration on how this objective is effectively accomplished.

IV. Methodology

Based on the analysis of the computational characteristics of floating-point numbers and the characteristics of neural network floating-point weights, we propose Mortar and Mortar-FP8.

Mortar uses a simple method to increase the bit sparsity of the mantissa, effectively reducing the width of the mantissa bits. Furthermore, Mortar-FP8 proposes a method for converting fp32 data into any floating-point representation and representing fp32 format as fp8. We further propose the corresponding accelerator that shortens the neural network weight representation without significantly affecting network performance, thus reducing the computational complexity and storage requirements of the network.

In the following sections, we will provide detailed explanations of both methods and the corresponding accelerator design.

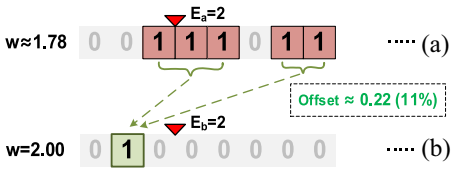

Fig. 4. Example of mantissa morphing. (a) and (b) depicts the original weight and morphed weight respectively. The sparsity of weight's mantissa has been significantly improved: The bit-level sparsity of (a) is five times that of (b) with a negligible error of merely 0.22 (11%).

A. Mortar

The first proposed solution, Mortar, focuses on reducing the bit width of the mantissa component. Mortar is a comprehensive hardware and software co-designed approach aiming at accelerating the inference process. To achieve this, Mortar employs a specialized technique called "mantissa morphing" that effectively sparsifies the bits at the mantissa level, thereby maximizing bit-level sparsity in the model. This sparsity optimization technique significantly enhances the efficiency of DNN acceleration at the software level.

To enhance clarity and facilitate understanding of the concept of mantissa morphing, we have included Fig. 4, which provides a graphical illustration of the core concept behind this approach. The initial weight is shown in Fig. 4(a). Through analyzing the bit significance, we infer that the overall value represented by multiple valid bits of figure (a) is very similar to the single valid bit located in figure (b). This leads us to design an algorithm locating the most significant "1" bit that can be optimized, i.e., a "1" bit preceded by a "0." We turn the preceding "0" into a "1" bit and clear all succeeding bits.

Intuitively, the original weight is only transformed to the new weight if their difference falls within an acceptable range. Consequently, deciding which bit to compensate into a "1" is essential to mantissa morphing. A precision algorithm is introduced to establish the error range P to control the difference between the morphing weight W' and the initial weight W. Below is the precise formula for Precision

$$\text{Precision} = \left(\left| \frac{W_i' - W_i}{W_i} \right| = \left| \frac{\varepsilon_i}{W_i} \right| < P \right) \ ? \ 1 : 0. \quad (2)$$

The hyperparameter P plays a crucial role in the mantissa morphing process as it determines the tradeoff between sparsity and accuracy. For each optimizing "1" bit, the precision function is called, and if the result is smaller than P, then mantissa morphing is applied at that position, replacing the original weight W with the morphed weight W'. However, if the error is greater than P, the compensation effect exceeds the appropriate range, and the search for the next valid bit is necessary. This approach avoids over and under-compensation, allowing for flexibility in adjusting the tradeoff between accuracy and compensation. If P is set to a large value, the morphing conditions are looser, and the algorithm will delete more bit 1s, increasing sparsity at the cost of model accuracy. Conversely, a lower value of P will be more restrictive when selecting morphing bits, preserving more information for accuracy.

At the offline level, Mortar processes the trained fp32 weights using mantissa morphing, which establishes a hardware-friendly approach to utilizing the abundant insignificant bits while preserving the original accuracy. The detailed technique of Mortar's offline algorithm is elaborated below.

Algorithm 1 Mantissa Morphing

Input: Original fp32 weight, W_i
Output: New weight after mantissa morphing, W_i',

1: Interpret the n-bit exponent $E = [e_1, \ldots, e_n]$ and mantissa
2: $M = [m_1, \ldots m_n]$, the actual position of E is determined.
3: Set the value for parameter 'P' in Precision function
4: foreach column j in W:
5: if $W_j = 1$ and $W_{j-1} = 0$:
6: $W'_{j-1} = 1$
7: foreach column k in $W\ [j : c]$:
8: $W'_k = 0$
9: if (Precision(W, W', P)) # precision judge
10: Return $W', j + 1$
11: else $W' = W$;
12: continue

*Loop 7 can be parallelized for speedups

1) Preprocessing: Consider the example of the mantissa of six fp32 weights in Fig. 5(a). We obtain a bit matrix displaying the binary mantissa stored in memory. The example shows 23 bit-width mantissas with the leftmost bit having the largest significance and the rightmost having the smallest significance that corresponds to values 2^{-1} to 2^{-23}. Each row represents a particular weight and is marked with a different color. According to IEEE 754, a hidden "1" is inserted into the mantissa's leftmost bit. The triangle mark represents the true relevance of weight by the value of the exponent.

2) Mantissa Morphing: Fig. 5(b) describes the core operation for mantissa morphing, and the pseudocode is provided in Algorithm 1. The initial weights are adjusted offline, and the parameter P is the threshold for morphing. For each weight W_i, the conditioned search begins from the most significant bit and progresses to the least important bit, and Mortar finds a j such that $W_{i,j}$ is valued "1" (lines 4 and 5). Then, the preceding $W_{i,j-1}$ bit is converted to "1" and all subsequent bits to "0," which we declare as the new weight W_i' (lines 6–8). Finally, W_i, W_i', and P are input into the Precision function. In Fig. 5(b), "0" bits with green backgrounds represent positions satisfying the morphing requirements. Finally, the new W_i is returned along with the morphing bit's location. Contrarily, 0 bits in red are positions failing the precision function test and where the weight is preserved until the next suitable bit (lines 9–12).

3) Mortar Encoding: Fig. 5(c) shows the mortar encoding process. Due to the mechanism of the mortar algorithm, we can automatically spot the specific location $j - 1$ of the last valid bit in each weight, followed by continuous "0" bits. Since the first valid bit is always the hidden "1" of each weight, the computing interval of each weight can be readily obtained from these data. This interval determines

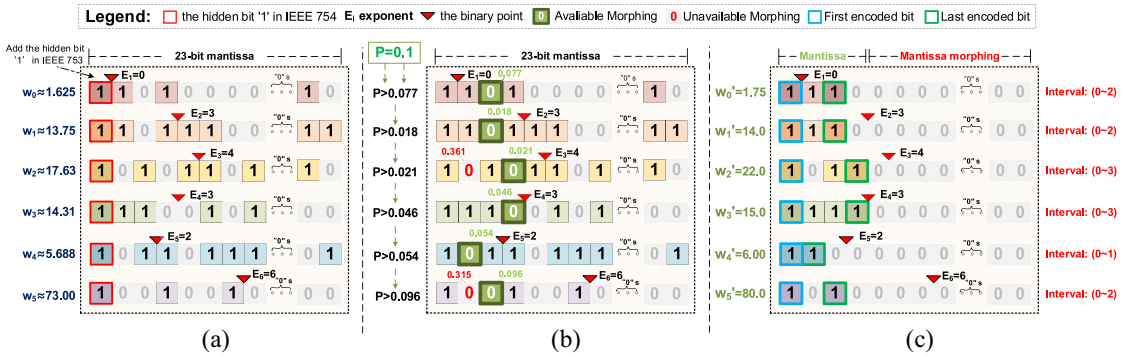

Fig. 5. Offline procedure of Mortar. Bit matrix for preprocessing is shown in (a). (b) demonstrates the process of selecting the bits to apply mantissa morphing with $P = 0.1$. And Mortar encoding, as shown in (c) to infer the data interval for each weight.

the specific computation range for the associating online accelerator design, substantially reducing the computational cycles and avoiding invalid operations to nontrivially improve the computation's efficiency.

B. Mortar-FP8

Mortar, as a technique, employs a morphing approach to effectively reduce the bit width of the mantissa. Building upon the established foundations of Mortar, our methodology further explores techniques to decrease the mantissa bit width, thereby transforming the process of bit width reduction from a morph compensation mechanism into a unified optimization problem. This advancement results in achieving shorter mantissa bit widths.

Furthermore, based on previous analyses, we propose a novel mechanism for reducing the exponent. Leveraging this mechanism, the study presents a converting algorithm and a detailed process for processing fp32 weights to E4M3 format, which serves as a specific case of this method. This conversion provides an efficient solution for accelerating DNN inference while preserving high accuracy. The refined and extended technique, aptly named Mortar-FP8, not only facilitates faster computations but also possesses highly desirable hardware-friendly characteristics.

1) Mantissa Processing: Manipulating the mantissa of a weight within a specific range has minimal impact on the actual value of the weight. It can consequently result in insignificant changes to both the MAC results and neural network performance. In the Mortar method, a suitable zero bit in the mantissa is searched for from front to back in a single weight, then converted to one and the subsequent bits truncated to shorten the bit width of the mantissa. This technique helps accelerate the calculation. However, if all weights are to be converted to the unified fp8 format, the number of bits in the mantissa must be fixed. Thus, the Mortar-FP8 method proposes a novel approach to this problem.

To achieve shorter and a fixed bit width of the mantissa, (3) is applied to the processing of the weight mantissa, ultimately leading to improved hardware design performance

$$\min\left|M_{\mathrm{fp}_n} - M_{\mathrm{fp}_{32}}\right|. \tag{3}$$

In (3), $M_{\mathrm{fp}_{32}}$ and M_{fp_n}, respectively, represent the mantissa bit of original fp_{32} and processed fp_n. The n represents the

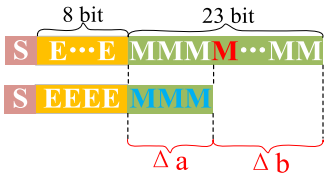

Fig. 6. Illustration of mantissa processing of Mortar-FP8. The mantissa bit width of converted floating-point format is l.

processed weight's bit width. To clearly illustrate how to minimize the difference between the $M_{\mathrm{fp}_{32}}$ and M_{fp_n}, Fig. 6 is illustrated.

According to Fig. 6, (3) can be expressed as the sum of Δa and Δb. The mantissa length of fp_n's is denoted by l. The value of Δb is obtained by taking the minus sum of the original fp_{32} bit mantissa ranging from the $(l+1)$th to the 23rd position. Conversely, the value of Δa can be determined by computing the difference between the first l original mantissa bits of fp_{32} and the corresponding first l mantissa bits of fp_n. Equation (3) is rewritten as

$$\min|\Delta a + \Delta b|. \tag{4}$$

Δb is 0 or a negative number. To minimize the difference between fp_n and fp_{32}, the value of Δa is set to either 0 or a positive number. Furthermore, for a fixed length of mantissa, the value of Δa is always a multiple of 2^{-l}. Therefore, the value of Δa can be expressed as $x * 2^{-l}$, where x is a non-negative integer. The equation is written as

$$\min\left|x * \ 2^{-l} + \ \Delta b\right|. \tag{5}$$

In the original fp_{32} weight, if all bits ranging from the $(l+1)$th to the 23rd of the mantissa are zero, then the variable Δb is equal to zero. Conversely, if all these bits are one, the Δb becomes $-(2^{-l} - 2^{-24})$. Consequently, the magnitude of Δb lies in the range between $-(2^{-l} - 2^{-24})$ and 0, which is always less than 2^{-l}. Therefore, the value of x can take only the 0 or 1.

When the solution of x in (5) is zero, it implies that (5) is minimized when x equals zero rather than one. In this situation, (5) represents the absolute value of Δb is less than $2^{-(l+\mathbf{1})}$

$$\begin{aligned} |2^{-l} + \Delta b| &> |\Delta b| \\ 2^{-l} - |\Delta b| &> |\Delta b| \\ \Delta b &< 2^{-(l+\mathbf{1})}. \end{aligned}$$

Based on the properties of binary bits, if the $(l+1)$th bit of the mantissa in the original fp_{32} weight is equal to zero, then the magnitude of $|\Delta b|$ is definitely less than $2^{-(l+1)}$; otherwise, if this bit is one, then the magnitude of $|\Delta b|$ is greater than $2^{-(l+1)}$, and the solution for x will be one. As a result, the value of x can be either zero or one, depending on whether the $(l+1)$th bit is 0 or 1, respectively. In the special case where the solution for x is equal to one, which occurs when the first l mantissa bits of the original number are all ones, setting x to one would result in an overflow of fp_n's mantissa representation. Therefore, in this scenario, the mantissa bits of fp_n would remain the same as the first l mantissa bits of fp32. Therefore, the solution to (3) can be written as

$$\begin{cases} M_{\text{fp}_n} = {M_{\text{fp}_{32}}}^{[0:l]}, & {M_{\text{fp}_{32}}}^{i} = 1 \text{ and } i \subset (0,\ l+1) \\ M_{\text{fp}_n} = {M_{\text{fp}_{32}}}^{[0:l]}, & {M_{\text{fp}_{32}}}^{l} = 0 \\ M_{\text{fp}_n} = {M_{\text{fp}_{32}}}^{[0:l]} + 1, & {M_{\text{fp}_{32}}}^{l} = 1. \end{cases} \tag{6}$$

Formula (6) determines how the mantissa of fp_n is processed, which considers both the length of the mantissa and the first $l+1$ bits of the mantissa in the original fp_{32} representation. If the initial l bits of the mantissa in the original fp_{32} representation are all 1 s, then the mantissa of fp_n will also be all 1 s. On the other hand, if the first l bits are not all ones, then the mantissa of fp_n is processed based on the value of the $(l+1)$th bit in the original fp_{32} formats. If the $(l+1)$th bit is 1, the mantissa of fp_n is derived by adding one to the first l bits of the mantissa in the original fp_{32}. However, if the $(l+1)$th bit is 0, the mantissa of fp_n will be identical to the first l bits of the mantissa in the original fp_{32}.

2) Dynamic Bias Mechanism: Lower precision in the floating-point format not only shortens the mantissa bits but also leads to a shorter exponent representation. Limited bit width in exponents results in decreased precision due to overflow or underflow errors. Conversely, too many exponent bits lead to accurate representation at the expense of slow speed in network computation and increased hardware overheads. Therefore, it is crucial to determine the appropriate exponent bit width for the data of the neural network.

The analysis in previous indicates that the distribution range of neural network weight exponents is significant, while their distribution gap within convolution kernels is relatively smaller and consistent. By denoting the gap between the maximum and minimum exponent as G, $\log_2^G$ bits are sufficient to accurately represent each exponent within the given kernel. Therefore, exponent bit width can be reduced while maintaining precise representation, accelerating network operations without affecting neural network's performance.

However, the exponent distribution ranges of different convolution kernels are varied, requiring a method that shortens bit width while representing various exponent ranges. In floating-point representation, bias provides an additional degree of freedom to address this issue. By subtracting a fixed bias value from the exponent, the exponent representation range can be shifted along with the offset. In the fp_{32} representation, 8-bit exponent representation ranges from 0 to 255, and by subtracting a fixed bias of 127, the range can be shifted to -127 to 128. Unlike bias in fp_{32}, our proposed method does not use a fixed bias. Instead, we set different biases for different kernels, which is called the dynamic bias mechanism. This approach achieves the consistent gap distribution within kernels of different ranges using the same lower bit-width.

Algorithm 2 FP8 Converting

Input: The $n \times n$ kernel original fp32 weights W_k
Output: New fp8 converted weight $W_k{}'$, dynamic bias B_k of $W_k{}'$

1: Interpret the n^2 sign bits $S_k = [s_1, \ldots, s_{n^2}]$, exponents bits $E_k = [e_1, \ldots, e_{n^2}]$ and mantissa bits $M_k = [m_1, \ldots m_{n^2}]$.
2: $B_k = \min(E_k)$
3: for e_i in E_k:
4: $\quad e_i' = e_i - B_k$
5: $\quad$ If $e_i' > 15$:
6: $\quad\quad e_i' = 15$ # *The maximum mantissa under E4M3 is 15.*
7: $\quad E_k' = [e_1{}', \ldots, e_{n^2}{}']$
8: for m_j in M_k:
9: $\quad$ if $m_j[3] = 1$
10: $\quad\quad$ if $m_i[0]$ and $m_i[1]$ and $m_i[2]$:
11: $\quad\quad\quad m_j' = m_j[:3]$
12: $\quad\quad$ else:
13: $\quad\quad\quad m_j' = m_j[:3] + 1$
14: $\quad$ else:
15: $\quad\quad m_j' = m_j[:3]$
16: $\quad M_k' = [m_1{}', \ldots, m_{n^2}{}']$
17: $W_k{}' = concat(S_k, E_k', M_k')$

Suppose the gap between weight exponent distributions within a convolution kernel is G, and the corresponding exponent value's maximum width is W ($W \leq \log_2^G$). The range for W bit representation is 0 to $2^W - 1$, and the bias for this kernel is the minimum of its exponent values. By subtracting the bias from all exponents, the kernel's exponent representation range can be shifted to 0 to $2^W - 1$. If the bit width W is less than $\log_2^G$, any data beyond $2^W - 1$ will be truncated to $2^W - 1$. In the actual calculation, the exponents within the kernel are the bit values plus the corresponding bias. Therefore, the exponent values for a given kernel can be represented using only W bit with this approach.

3) fp8 Converting: We propose to apply the exponent and mantissa compression described above to the fp_8 format. The first bit represents the sign in fp_8 presentation. By setting the exponent bit-width to 4 and the mantissa bit-width to 3, we achieve the E4M3 fp_8 format, which leverages the fact that most of the gap distribution in kernels' exponents is within 16, as shown in Fig. 3.

Algorithm 2 outlines the procedure for converting fp_{32} weights into the fp_8 format. In line 2, we determine the bias for a given $n \times n$ size convolution kernel W_k as the minimum exponent of the n^2 weights. Lines 3–7 describe how exponents are processed when converting weights to fp_8. The algorithm subtracts the bias in line 2 from the original exponent to obtain the converted fp_8 exponent. If the converted exponent is greater than 15 (the maximum exponent representation for E4M3), it is set to 15.

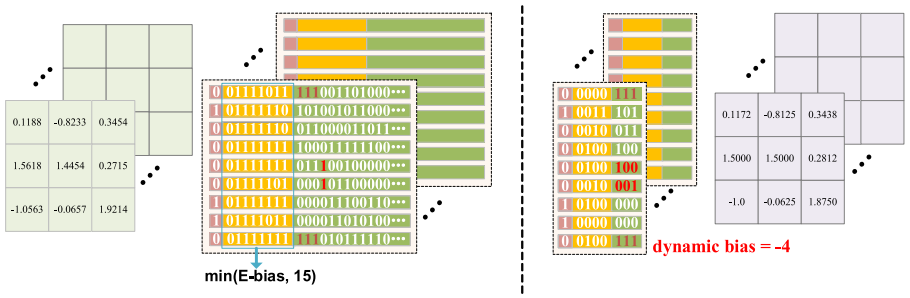

Fig. 7. Example of converting a 3×3 kernel fp32 weights to fp8 formats. (a) Original kernel weights and its fp32 bits representation. (b) Converted E4M3 fp8 formats of the kernel weights.

Lines 8–16 describe how to shorten the 23-bit mantissa to 3 bits in the fp_8 format. If the fourth bit of the original mantissa is 1, and the first three bits are all 1 s, then the converted fp_8 mantissa should be set as all 1 s. For other cases where the first three bits are not all 1 s, the converted fp_8 mantissa should be calculated by adding one to the first three bits of the original mantissa. Otherwise, the converted fp8 mantissa should be consistent with the first three bits of the original mantissa. Additionally, we need to save the dynamic bias of the corresponding kernel during the conversion process.

By following this approach, we can convert fp_{32} weights to fp_8 while optimizing the exponent and mantissa bit-widths. This will effectively accelerate neural network computation without compromising performance quality.

To intuitively illustrate our proposed method, an example of processing a 3×3 kernel weight into fp8 formats is given in Fig. 7. Fig. 7(a) displays the original weights and their corresponding fp32 representation. The yellow-colored portion represents the exponent part, while the green-colored segment denotes the mantissa of each weight. The first three bits of the mantissa that are equal to 1 are represented by a dark red color, while the fourth bit that is equal to 1 is represented by a bright red color.

For this kernel, the minimum value of the exponent is –4. Thus, in Fig. 7(b), the dynamic bias is –4. To obtain the converted fp8 exponent for each weight, we subtract the dynamic bias from the original exponents. If the converted exponent is greater than 15, it is set as 15 in line with the maximum exponent representation for E4M3.

In the mantissa part, we observe that the first and ninth weights have the first three bits equal to 1, indicated in dark red. Consequently, the mantissa of the converted fp8 weight is also all equal to 1. For weights where the fourth bit is equal to 1 (e.g., the fifth and sixth weights), we add 1 to the first three bits of the original bit when converting it to fp8. For other weights, the tails are consistent with the first three bits of the original weight.

Fig. 7 illustrates that the numerical difference between the original kernel and converted kernel is minor, but the representation for each weight has significantly decreased from 32 bits to 8 bits. This has clear benefits for both computing and storing neural networks, making our proposed method highly advantageous.

V. Mortar Accelerator

This section describes our accelerator design that supports both fp32 and fp8 modes, based on our proposed methods. For fp32 mode, the model weights must be processed online with mantissa morphing. However, this requirement is unnecessary for fp8 mode because Mortar-FP8 conversion is easy and efficient to deploy on our accelerator. The user can configure the accelerator to select between the two modes based on different scenarios.

As shown in Section VI, 32-bit mode has slightly higher accuracy with Mortar acceleration method, while Mortar-FP8 processing requires lower computational and storage requirements. Therefore, users can choose the fp32 mode for tasks such as medical image diagnosis and autonomous driving, where accuracy is critical. Alternatively, fp8 mode can be selected for tasks such as some image processing to achieve high-efficiency calculation with low storage demand.

A. Fp32 Mode for Offline Mortar

The fp32 mode in the accelerator is designed to speed up fp-32 inference using the mantissa morphing algorithm. Fig. 8 shows the overall architecture of Mortar accelerator, while Table V provides an area and energy breakdown.

In our accelerator, memory access is through DMA, and data fetched from memory is stored in a local buffer. The accelerator integrates a memory of 2 GB. To facilitate efficient computation, registers are specifically allocated for storing the weights and activations within the architecture. Likewise, registers are also utilized for holding the intermediate values generated during the preprocess stage and within the compute unit (CU) component. The Mortar processing element (PE) consists of an array of Mortar CUs, which receives input activations $A_0 \sim A_{N-1}$ and weights $W_0 \sim W_{N-1}$. Within each CU, a serial architecture is used to perform ib × ib MAC operations per cycle, where i represents the ith input of A and W.

Once the weights have been processed offline by the Mortar accelerator, the preprocessing module separates the weights into two parts: the 2–9 bits and the 10–32 bits. These bits are then stored in registers. This separation enables the division of each weight into its mantissa and exponent components. Subsequently, these preprocessed weights are

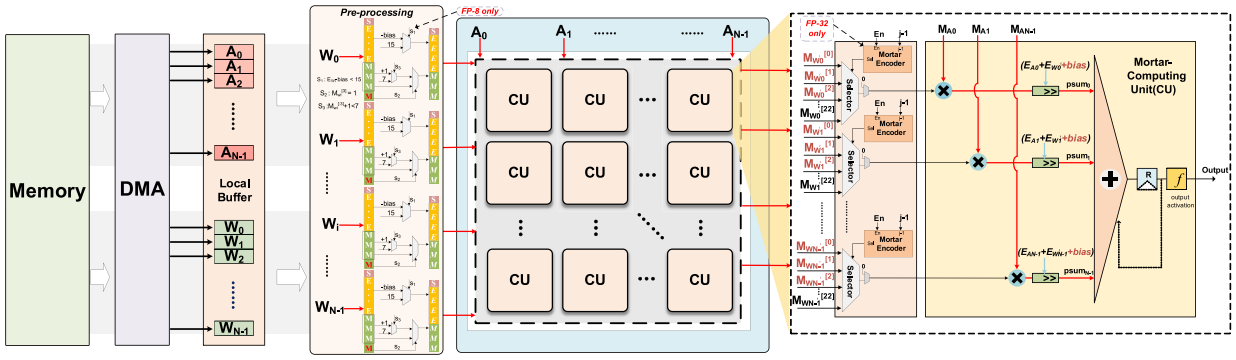

Fig. 8. Overview of Mortar accelerator architecture and the microarchitecture of mortar CU.

then fed into the CU. Simultaneously, the mantissa bits of the activations, denoted as $M_{A0\sim N-1}$ are serially inputted into the CU alongside the weight mantissa bits $M_{W0\sim N-1}$. Each selector receiving the weight mantissa is governed by the "mortar encoding" signal, which ensures that only the relevant data interval for each weight is selected. Any data falling outside this interval is automatically assigned a value of 0. This mechanism aids in accurately choosing the appropriate bits while discarding unnecessary information.

By utilizing a serial computation of M_{Wi} and M_{Ai} $(i = 0, 1 \ldots, N-1)$, Mortar can output the multiplication value using simple combinational logic in each cycle, reducing hardware design overhead. The shifter then shifts the output value of each cycle with the corresponding bit significance to ensure the correctness of the result. For floating point numbers, the shifting operation is performed by the exponent accumulation in A, not introducing much overhead. The corresponding shift for MACs is $E_{W_i'} + E_{A_i}$, representing the multiplication of activation and weight bits in the cycle. Lastly, the adder tree performs final partial-sum accumulation.

Overall, Mortar provides a high-performance hardware accelerator for fp-32 inference, leveraging the benefits of the mantissa morphing algorithm.

B. FP8 Mode for Online Mortar-FP8

Due to the simplicity and effectiveness of the Mortar-FP8 methodology, the accelerator incorporates the online components, denoted as "fp8 only" in Fig. 8, to convert weights from fp32 to fp8, and perform computations with these components.

The preprocessing unit responsible for handling 32-bit weights in the Mortar-FP8 method consists of multiple modules aimed at performing mantissa reduction and dynamic exponent adjustment online. In addition to just interpreting fp32 weights to mantissa and exponent in Mortar, this unit employs simple 2-way selectors to process the exponent mantissa online, ultimately transforming them into the E4M3 format.

Regarding the exponent processing of the weight, the accelerator utilizes a 2-way selector. It compares the result obtained by subtracting the dynamic bias from the exponent with 15 and outputs the converted fp8 exponent accordingly.

When it comes to the mantissa processing, the preprocessing unit employs a series of 2-way selectors. If the fourth bit of the original weight mantissa is not equal to 1, the first 2-way selector generates the original first three bits of the mantissa. However, if the fourth bit is 1, the next 2-way selector comes into play. This selector compares the value obtained by adding 1 to the first three bits of the mantissa with 7 before outputting the processed mantissa value. The selectors in the Mortar-FP8 accelerator have a minimal impact on overhead, as they are designed to be simple. This can be observed in Table VI, which presents the overhead of the preprocessing unit of both the Mortar and the Mortar-FP8 accelerators.

With the steps in the preprocessing unit, the Mortar accelerator performs online processing of 32-bit weights into 8-bit weights, which are then sent to the CU component for computation. When fed into the CU, the mantissa bits are compressed into 3 bits, which is different from the 23 bits in fp32 mode. The compressed bits are colored with dark pink in Fig. 8. The encoder component, which is unnecessary in the fp8 mode CU design, is depicted as "fp32 only." Since all weight exponents subtract the shared bias, the corresponding shift for MACs in the part-sum calculation is $E_{W_i'} + E_{A_i} + \mathrm{bias}$.

Overall, the Mortar accelerator with fp8 mode employs the simple components to swiftly convert fp32 weights to fp8 format online, resulting in accelerated neural network computation at a relatively low cost. The reduction in bit-width of the weights significantly alleviates the memory requirements and computational complexities, ultimately accelerating inference in neural networks.

VI. Evaluation and Discussion

A. Benchmark and Hardware Implementation

The deep learning models and the pretrained parameters on ImageNet [26] dataset are directly obtained from PyTorch [27]. The benchmark models are shown in Table I to demonstrate the ability of Mortar and Mortar-FP8 for general-purpose DNN task acceleration. We choose the tasks from different domains including Image Classification, Object Detection, Video Understanding, Video and Image Super Resolution, and style transfer. The benchmarks cover "large" models with the parameter size, such as 88.79M (ResNext101 [28]), 86.61M (ViT [29]), as well as "small" models with the

TABLE I
BENCHMARK DNNS AND THEIR ORIGINAL SPECS FOR EVALUATING MORTAR'S AND MORTAR-FP8'S PERFORMANCE

Models	Domain	Type	Dataset	Metric	GFLOPS	Weights (M)	Orig. Accuracy
DenseNet161	Image Classification	2D Convolution	ILSVRC2012	Top-1 %	15.64	28.68	75.28
ResNext101	Image Classification	2D Convolution	ILSVRC2012	Top-1 %	33.02	88.79	78.24
ResNet18	Image Classification	2D Convolution	ILSVRC2012	Top-1	3.64	11.69	67.28
Yolov3	Object Detection	2D Convolution	COCO	mAP	25.42	61.95	53.30
FCOS [1]	Object Detection	Feature Pyramid	COCO	mAP	80.14	32.02	0.382
ViT	Video Understanding	Transformers	ILSVRC2012	Top_1(%)	29.42	86.61	83.89
D3DNet	Video Super Resolution	3D Deformable	Viemo-90k [2]	PSNR	408.82	2.58	36.05
				SSIM			0.94
LapSRN	Image Super Resolution	2D De-Convolution	SET14 [3]	/	736.73	0.87	See Figure 11 (a)
CartoonGAN	Style Transfer	GAN	Flickr [5]	/	108.98	11.69	See Figure 11 (b)

TABLE II
ACCURACY OF MORTAR UNDER $P = 0.1$

Model	Baseline	Mortar(P=0.1)
DenseNet161	75.28	**75.31**
ResNext101_32x8d	78.24	**78.26**
ResNet18	67.28	**67.22**
Yolov3	53.30	**53.10**
FCOS	0.382	**0.378**
Vit	83.89	**83.66**
D3DNet	36.05	**36.05**
	0.94	**0.94**

TABLE III
ACCURACY OF MORTA-FP8

Model	Baseline	Mantissa processing	Mortar-FP8
ResNet18	67.28	66.91	**66.94**
ResNet34	71.32	70.85	**70.86**
ResNet50	74.50	74.43	**74.13**
DenseNet121	71.97	71.82	**71.81**
DenseNet161	75.28	74.88	**74.69**
ResNext101_32x8d	78.23	77.92	**77.97**
YoLov3	53.30	53.50	**53.40**
D3DNet	36.05	36.03	**36.02**
	0.937	0.936	**0.936**

parameter size of 2.58M (D3DNet [30]). To further demonstrate the generalization capability of Mortar and Mortar-FP8, Yolov3 [31] is trained on the COCO [32] dataset. Our experiments extensively evaluate Mortar's and Mortar-FP8's effect on model accuracy and bit width decrease, and we compare the performance with SOTA hardware pruning accelerator BitX.

At the RTL, we utilize Vivado HLS (v2018.2) for post-synthesis simulation on Xilinx Virtex-7 FPGA. Our design consists of 16 CUs within PEs, operating at a clock frequency of 200 MHz. To evaluate the runtime memory access data and estimate the energy consumption associated with memory accesses, we record the data and utilize the DRAMsys tool [33]. To measure power and area during RTL synthesis, we employ Synopsys Design Compiler (v2016). The design is synthesized using the TSMC 28nm technology library, with the frequency set at 1 GHz.

B. Accuracy

Mortar (fp32): To validate the effectiveness of the Mortar method, many experiments are conducted on a large number of neural network models. As discussed in the methodology section, the parameter P plays a crucial role in striking a balance between the precision of the neural network model and the acceleration achieved through Mortar. By increasing P, more sparsity is introduced, which in turn leads to faster inference speed. To determine the optimal value for P that maximizes network performance, we conducted thorough space exploration. Although only some of the results are presented in Fig. 9 due to space limitations, more comprehensive findings can be found in the conference version, further demonstrating the rationale behind our choice of P.

From Fig. 9, it can be observed that, for the majority of models, maintaining a range of $0.0001 < P < 0.05$ yields near-identical performance. However, as P exceeds 0.1, a notable decline in accuracy is observed. Moreover, when P surpasses 0.5, there is a significant decrease in accuracy. To strike a balance between acceleration effects and neural network performance, we set the value of P to 0.1 without compromising the overall performance of the neural network model.

Table II presents the accuracy of various models with P set to 0.1. It is evident from the table that the majority of models maintain the same level of accuracy as the original model, with only minor accuracy degradation in some instances. Notably, DenseNet161 and ResNext101_32x8d demonstrate improved accuracy when the Mortar method is applied. This surprising result suggests that Mortar not only preserves the performance of neural networks but can even enhance accuracy in certain cases.

Mortar-FP8: Massive experiments are also conducted to evaluate the effectiveness of Mortar-FP8 and investigate the contributions of each component of the proposed method. Specifically, we analyzed the impact of separately processing only the mantissa to 3 bits and converting the entire weights to fp8. The comprehensive results are reported in Table III. As highlighted in Section III, the weights' exponent gap surpasses the range of 4 bits. Hence, we refrain from processing the weights of the fully connected layer of the image recognition models to fp8.

Based on an analysis of the results presented in Table III, it is evident that reducing the mantissa to 3 bits for all models leads to an average decrease in accuracy of less than 0.4%. In comparison with Mortar, when employing the Mortar-FP8 approach, which further reduces the mantissa bits, a slightly higher reduction in accuracy is observed cause more bits are eliminated. Nevertheless, this reduction in accuracy is minimal considering the significant reduction in bit width from 23 to 3 bits.

Furthermore, when converting all weights to fp8 using the proposed method, there is no significant reduction in network performance. Similarly, the increased accuracy is also

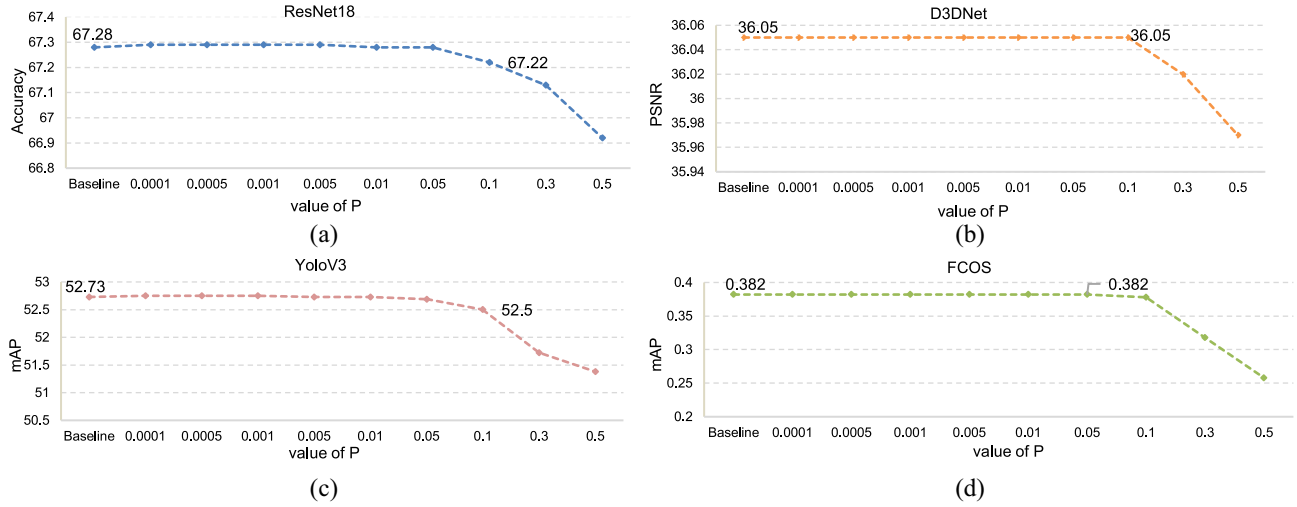

Fig. 9. Design space exploration of key design parameter P on various models and P = 0.1 is the turning point for most models. (a) Design space exploration of P in ResNet18. (b) Design space exploration of P in D3DNet. (c) Design space exploration of P in YoLoV3. (d) Design space exploration of P in FCOS.

TABLE IV
ACCURACY/SPARSITY COMPARISON BETWEEN MORTAR AND BITX

Models	Original	BitX-mild	Mortar
DenseNet161	75.28/1x	75.20/1.32x	**75.37/1.58x**
ResNext101_32x8d	78.24/1x	78.20/1.27x	**78.26/1.81x**
ResNet18	67.28/1x	67.09/1.64x	**67.22/2.09x**
Avg. loss / sparsity	0.00/1x	-0.10/1.41x	**+0.01/1.83x**

seen in YoLov3. The improved accuracy observed in these cases can be primarily attributed to the benefits offered by the low-precision representation formats employed in both Mortar-FP8 and Mortar. Specifically, the decreased mantissa bit length and the utilization of a reduced precision format (FP8) for representing weights may introduce some level of noise. In certain cases, this noise can act as a regularizer to prevent overfitting. Consequently, the models exhibit improved accuracy compared to the baselines.

Taken together, these experimental findings provide clear evidence that the dynamic bias mechanism employed in the method is both effective and accurate in handling floating-point values with the exponent processed to 4 bits. And the Mortar-FP8 acceleration method demonstrates effectiveness. The results highlight its potential to be widely applied in various real-world applications where the reduction in bit width has minimal impact on network performance.

Comparison With Relevant SOTA Design: As a method that reduces the bit width of the mantissa based on mantissa sparsity, we present the effectiveness of Mortar by analyzing the bit-level sparsity of the neural network model after applying the Mortar method. Additionally, we compared Mortar with BitX [14], a novel DNN accelerator using hardware pruning to increase bit-level sparsity for inference acceleration.

Table IV shows the accuracy and sparsity changes for different types of datasets that apply mantissa morphing using the threshold $P = 0.1$. The results show that in general cases, the sparsity improvement of a model can reach 2× with negligible accuracy degradation.

We selected several models, including DenseNet161 [34], ResNext101, and ResNet18 [35], for the image classification task. Table IV shows that Mortar maintains improved sparsity while achieving better accuracy in all three models. When

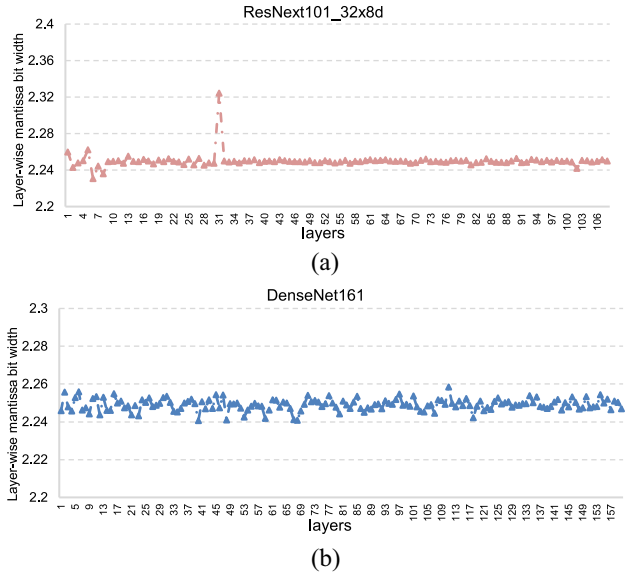

Fig. 10. Mantissa bit width of (a) ResNext101_32x8d and (b) DenseNet161.

comparing the other data reported in BitX, Mortar outperforms BitX in accuracy while maintaining a slightly improved sparsity at similar levels of sparsity improvement. The difference in model accuracy is remarkably significant, with a nontrivial 5% improvement for ResNext101 and ResNet18.

Although BitX effectively improves the sparsity of model weights, it fails to consider the difference of each weight in the model. Its fixed-position pruning leads to over/under-pruning. However, Mortar accounts for the different sparsity proportions of each weight in mantissa morphing. It compresses the weights based on bit significance and uses bitwise compensation to achieve fine-grained compression that preserves high accuracy. Therefore, Mortar addresses sparsification more precisely by utilizing bit-level sparsity and significantly extending the cost-benefit tradeoff between accuracy and compression. By utilizing these techniques, Mortar achieves superior performance compared to BitX, highlighting its potential as an effective and efficient solution for compression in neural networks.

Fig. 11. Visual demonstrations of (a) 4× super resolution inference via Mortar and Mortar-FP8 and (b) cartoon style transfer via Mortar and Mortar-FP8. The results are extremely identical to the style transfer for the original image. The visual demonstrations support the effectiveness of Mortar and Mortar-FP8 as powerful solutions for fast and efficient inference with preserved quality.

C. Mortar-FP8 Specifics

Regarding Mortar-FP8 specifically, the mantissa processing proposed in Section IV reduces the bit width of the mantissa to a maximum of 3 bits, which provides a more obvious indication of the actual computation cycle compared to bit sparsity. Fig. 10 illustrates the layer-wise bit-width of the mantissa for DenseNet161 and ResNext101 applied with Mortar-FP8. The bit-width of a weight is represented by the position of last bit one in the mantissa. Notably, Mortar-FP8 achieved an outstanding reduction in the average layer bit-width of the mantissa, from 23 bits to approximately 2.25 bits. The largest bit width in ResNext101 is just about 2.3. Furthermore, this reduction does not result in a significant decrease in accuracy of DenseNet161 and ResNext101, with only a negligible decrease observed.

This figure indicates the effectiveness of our proposed approach in shortening the bit-width of floating-point numbers. The lower bit-width implies fewer computations, leading to faster neural network performance. Mortar-FP8 significantly reduces the bit-width of floating-point numbers without affecting the performance of neural networks. This demonstrates the effectiveness of Mortar-fp8 as a powerful solution for reducing computation costs while preserving high-level model accuracy.

D. Visual Comparison

To qualitatively analyze Mortar and Mortar-FP8, we apply our approaches on multiple image processing tasks to visually display their effect on image outputs. In Fig. 11, we apply Mortar and Mortar-FP8 on both 4× Super Resolution with LapSRN [36] and CartoonGAN [37], showing results for both original and enhanced models.

The results show that Mortar's and Mortar-FP8's effect on the original model is both quantitatively minimal and qualitatively imperceptible to the end user. Mortar maintained a high-level quality of its outputs, indicating its effectiveness in reducing bit-width while maintaining high accuracy and quality.

E. Accelerator Performance

To evaluate the performance of Mortar and Mortar-FP8 hardware accelerator against other accelerators, we conducted a concrete analysis in Fig. 12. Specifically, we selected Pragmatic [15] and Stripes [38] as representatives for bit-serial accelerators, along with ResNet50 and SqueezeNet1_1 [39] as inference models for the comparison.

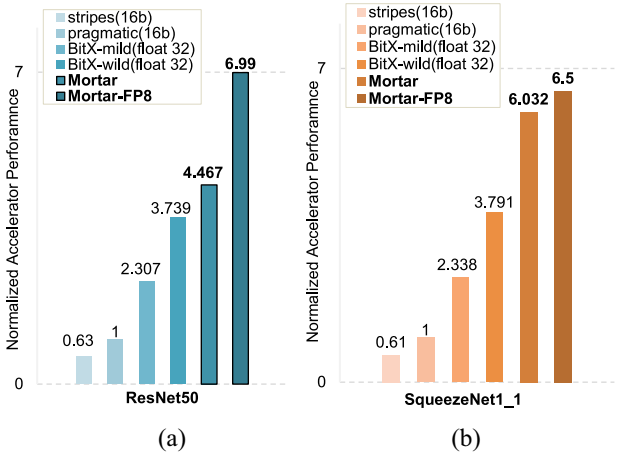

Fig. 12. Speedup comparison of Mortar and Mortar-FP8 with other SOTA accelerators in (a) for ResNet50 and (b) for SqueezeNet1_1.

While Stripes implements MAC computation using bit-level arithmetic, it does not consider bit sparsity. Pragmatic builds on stripes by dynamically skipping zero bits, thereby exploiting bit sparsity. However, they are not designed to fully exploit bit sparsity or to shorten the bit-width of floating point. In contrast, Mortar and Mortar-FP8 focus on these issues to achieve superior performance.

Through our proposed methods and accelerator, we address these challenges by mitigating mantissa sparsity and shortening the bit-width of floating-point numbers. Overall, our experimental results demonstrate that Mortar and Mortar-FP8 outperform the other architectures, highlighting their potential as powerful solutions for efficient inference in neural networks.

Speedup: Fig. 12 demonstrates the speedup achieved by Mortar and Mortar-FP8 over Pragmatic (the normalized baseline) and Stripes.

Mortar achieves a speedup of 4.467× over Pragmatic for ResNet50, while Mortar-FP8 achieves a speedup of 6.99×. For SqueezeNet1_1, Mortar achieves a speedup of 6.302× over Pragmatic, while Mortar-FP8 achieves a speedup of 6.5×. Our methods also outperform BitX in terms of inference speedup. As the experiment does not consider the factor of bandwidth reduction, the acceleration effect of our approach would be even better in reality.

This superior performance can be attributed to the effective utilization of bit sparsity in the mantissa by the proposed accelerator. Specifically, Mortar's encoding mechanism reduces cycles that do not significantly contribute

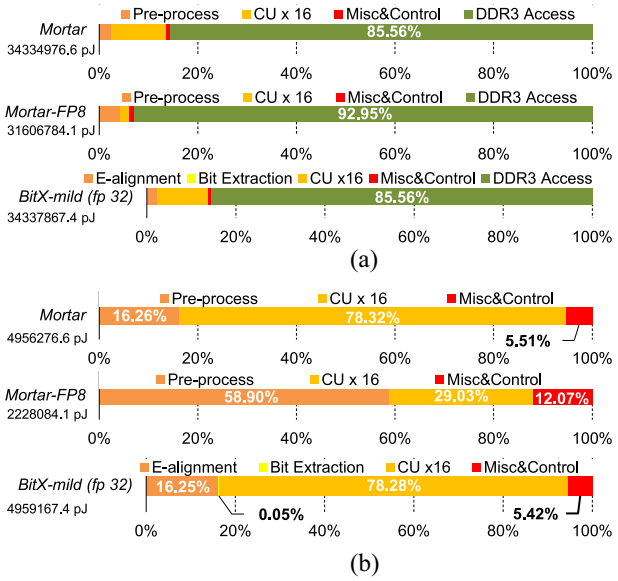

Fig. 13. (a) Full-system energy breakdown and (b) PE-only energy breakdown.

TABLE V
PE AREA AND POWER BREAKDOWN @TSMC 28NM

Accelerator	Mortar		Mortar-FP8	
Item	Area (mm^2)	Power (mW)	Area (mm^2)	Power(mW)
Pre-processing	0.017 (54.84%)	11.15 (16.26%)	0.039 (77.23%)	18.16 (49.64%)
16 CUs	0.003 (9.68%)	53.71 (78.32%)	0.0005 (0.99%)	8.95 (35.57%)
Misc and Control	0.011 (35.48%)	3.72 (5.42%)	0.011 (21.78%)	3.72 (14.79%)
Total	0.031	68.58	0.0505	25.16

to model performance by focusing solely on the encoded intervals, while the bit width of the exponent in Mortar-FP8 is drastically reduced. These approaches result in a significant reduction of computation complexity, leading to faster inference times. Moreover, since the bandwidth required for data transfer and storage is significantly reduced when weights are morphed from fp32 to fp8 format, the actual acceleration should be even greater.

Energy Breakdown: Our Xilinx V7 FPGA platform involves the DDR3 memory. We use DRAMsys to estimate the runtime memory access energy. As shown in Fig. 13, the energy breakdown can be analyzed from two aspects. First, Fig. 13(a) shows the full-system energy breakdown, revealing that memory accesses dominate the energy consumption. For example, the memory access energy of Mortar-FP8 makes up 92.95%, while the PE energy only occupies about 2%. Second, in Fig. 13(b), we decompose the PE-only energy. In Mortar, CU energy dominates at 78.32%, as we have 16 CUs with a large number of buffers to store the bit-pruned weights. However, for Mortar-FP8, the preprocess dominates because the conversion from fp32 to fp8 occurs online during the process.

Area and Power Breakdown: Under TSMC's 28 nm technology node, the area of Mortar and Mortar-FP8 is 0.031 mm^2 and 0.0505 mm^2 respectively. According to Table V the preprocess phase accounts for the majority of the area, representing 54.84% of Mortar and a staggering 77.23% of Mortar-FP8. As the weight processing phase in Mortar is performed offline, it consumes less area and power compared to Mortar-FP8. It is worth noting that the CU used in the design of Mortar-FP8 has been significantly reduced due to the decreased bit length of the weights (8 bit) when compared to Mortar (32 bit). This reduction in CU size has contributed to an impressive decrease in overall power consumption, resulting in a mere 25.16 mW. Additionally, we have considered other floating-point accelerators for comparison under TSMC's 28 nm technology node in Table VI, such as BitX [14], ReDCIM [40], and VLSI'21 [41]. BitX proposed a hardware pruning accelerator specifically targeting fp32, while ReDCIM and VLSI'21 support FP32, FP16, and BF16 formats separately. The proposed Mortar-FP8 supports shorter floating-point formats and achieves smaller power consumption.

TABLE VI
COMPARISON WITH OTHER FLOATING-POINT ACCELERATORS

Accelerator	BitX	ReDCIM	VLSI'21	Mortar	Mortar-FP8
FP precision	FP32	FP16, FP32	BF16	FP32	**FP8**
Area(mm^2)	0.039	-	-	**0.031**	**0.0505**
Power (mW)	68.62	12.5~69.4	1.2~156.1	**68.58**	**25.16**

VII. CONCLUSION

This article introduces novel offline/online collaborative approaches for accelerating general-purpose deep learning—software optimization called mantissa morphing and an fp8 conversion algorithm, along with hardware accelerator design. Our approach, Mortar, employs bit compensation when optimizing bit-level operations, significantly increasing the mantissa's sparsity to accelerate deep learning models based on fp32. Mortar-FP8 successfully converts fp32 weights to fp8 representation, reducing computation and memory requirements while maintaining high inference accuracy. Furthermore, our methods exhibit robust generalization capabilities across different model tasks and datasets, outperforming existing hardware accelerators. We hope this work will inspire future accelerator designs to become more efficient and versatile.

REFERENCES

[1] Z. Tian, C. Shen, H. Chen, and T. He, "FCOS: Fully convolutional one-stage object detection," in *Proc. ICCV*, 2019, pp. 9627–9636.
[2] T. Xue, B. Chen, J. Wu, D. Wei, and W. T. Freeman, "Video enhancement with task-oriented flow," *Int. J. Comput. Vision*, vol. 127, pp. 1106–1125, Feb. 2019.
[3] R. Zeyde, M. Elad, and M. Protter, "On Single image scale-up using sparse-representations," in *Proc. Int. Conf. Curves Surfaces*, 2010, pp. 711–730.
[4] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. ISCA*, 2017, pp. 1–12.
[5] "Flickr image dataset." Accessed: Apr. 16, 2023. [Online]. Available: https://www.kaggle.com/hsankesara/flickr-image-dataset
[6] J. Ouyang, X. Du, Y. Ma, and J. Liu, "3.3 Kunlun: A 14nm high-performance AI processor for diversified workloads," in *Proc. ISSCC*, 2021, pp. 50–51.
[7] E. Technology. "Enflame DTU." Accessed: Apr. 16, 2023. [Online]. Available: https://www.servethehome.com/enflame-dtu-1-0-ai-compute-chip-at-hot-chips-33/

[8] Cambricon. "CambriconMLU290." Accessed: Apr. 19, 2023. [Online]. Available: https://www.cambricon.com/index.php?m=content&c=index&a=lists&catid=340
[9] K. Zhong et al., "Exploring the potential of low-bit training of convolutional neural networks," 2020, *arXiv:2006.02804*.
[10] L. Cambier, A. Bhiwandiwalla, T. Gong, M. Nekuii, O. H. Elibol, and H. Tang, "Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks," in *Proc. ICLR*, 2020, pp. 1–12.
[11] X. Sun et al., "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks," in *Proc. NIPS*, 2019, pp. 1–10.
[12] N. Mellempudi, S. Srinivasan, D. Das, and B. Kaul, "Mixed precision training with 8-bit floating point," in *Proc. ICLR*, 2019, pp. 1–10.
[13] P. Micikevicius et al., "FP8 formats for deep learning," 2022, *arXiv:2209.05433*.
[14] H. Li et al., "BitX: Empower versatile inference with hardware runtime pruning," in *Proc. ICPP*, 2021, pp. 1–12.
[15] J. Albericio, P. Judd, A. Delmas, S. Sharify, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proc. MICRO*, 2017, pp. 382–394.
[16] Y. Gao, H. Li, K. Zhang, X. Yu, and H. Lu, "Mortar: Morphing the bit level sparsity for general purpose deep learning acceleration," in *Proc. ASP-DAC*, 2023, pp. 739–744.
[17] M. Courbariaux, Y. Bengio, and J. P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in *Proc. NIPS*, 2015, pp. 1–9.
[18] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Proc. ECCV*, 2016, pp. 525–542.
[19] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. CVPR*, 2017, pp. 2704–2713.
[20] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016, *arXiv:1603.01025*.
[21] IEEE. "IEEE standard for floating-point arithmetic (754-2019)." 2019. [Online]. Available: https://standards.ieee.org/standard/754-2019.html
[22] H. Lu, X. Wei, N. Lin, G. Yan, and X. Li, "Tetris: Re-architecting convolutional neural network computation for machine learning accelerators," in *Proc. ICCAD*, 2018, pp. 1–8.
[23] S. Sharify et al., "Laconic deep learning inference acceleration," in *Proc. ISCA*, 2019, pp. 304–317.
[24] F. Tu et al., "A 28nm 29.2TFLOPS/W BF16 and 36.5TOPS/W INT8 reconfigurable digital CIM processor with unified FP/INT pipeline and bitwise in-memory booth multiplication for cloud deep learning acceleration," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2022, pp. 1–3.
[25] H. Lu et al., "Distilling bit-level sparsity parallelism for general purpose deep learning acceleration," in *Proc. MICRO*, 2021, pp. 963–976.
[26] J. Deng et al., "ImageNet: A large-scale hierarchical image database," in *Proc. CVPR*, 2009, pp. 248–255.
[27] Facebook. "Pytorch." Accessed: Apr. 13, 2023. [Online]. Available: https://pytorch.org/
[28] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proc. CVPR*, 2017, pp. 1492–1500.
[29] A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. ICLR*, 2020, pp. 1–21.
[30] X. Ying, L. Wang, Y. Wang, W. Sheng, W. An, and Y. Guo, "Deformable 3D convolution for video super-resolution," 2020, *arXiv:2004.02803*.
[31] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," in *Proc. CVPR*, 2018, pp. 1–6.
[32] T.-Y. Lin et al., "Microsoft COCO: Common objects in context," in *Proc. ECCV*, 2014, pp. 740–755.
[33] C. W. M. Jung and N. Wehn, "DRAMSys: A flexible DRAM subsystem design space exploration framework," *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 8, pp. 63–74, Feb. 2015.
[34] G. Huang, Z. Liu, V. Laurens, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. CVPR*, 2017, pp. 4700–4708.
[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016, pp. 770–778.
[36] W.-S. Lai, J.-B. Huang, N. Ahuja, and M.-H. Yang, "Deep Laplacian pyramid networks for fast and accurate super-resolution," in *Proc. CVPR*, 2017, pp. 5835–5843.
[37] Y. Chen, Y.-K. Lai, and Y.-J. Liu, "CartoonGAN: Generative adversarial networks for photo cartoonization," in *Proc. CVPR*, 2018, pp. 9465–9474.
[38] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proc. MICRO*, 2016, pp. 1–12.
[39] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," in *Proc. ICLR*, 2017, pp. 1–13.
[40] F. Tu et al., "ReDCIM: Reconfigurable digital computing-in-memory processor with unified FP/INT pipeline for cloud AI acceleration," *IEEE J. Solid-State Circuits*, vol. 58, no. 1, pp. 243–255, Jan. 2023.
[41] J. Lee et al., "A 13.7 TFLOPS/W floating-point DNN processor using heterogeneous computing architecture with exponent-computing-in-memory," in *Proc. Symp. VLSI Circuits*, 2021, pp. 1–2.

Hongyan Li is currently pursuing the Ph.D. degree with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

Her research interests include software–hardware co-design acceleration of neural networks, deep learning network pruning, video understanding, and dynamic inference.

Hang Lu received the B.S. and M.S. degrees in electronic and information engineering from Beihang University, Beijing, China, in 2008 and 2011, respectively, and the Ph.D. degree in computer architecture from the University of Chinese Acedemy of Sciences, Beijing, in 2015.

He is currently an Associate Professor and a Master Tutor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. He is also a Research Scientist with the Shanghai Innovation Center for Processor Technologies, CAS. He is a member of the Youth Innovation Promotion Association of CAS, and the New Best Star of ICT. His research interests include power-efficient computing platforms, AI chip design, and deep learning algorithm optimization.

Xiaowei Li (Senior Member, IEEE) received the B.Eng. and M.Eng. degrees in computer science from the Hefei University of Technology, Hefei, China, in 1985 and 1988, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1991.

He was an Associate Professor with the Department of Computer Science and Technology, Peking University, Beijing, from 1991 to 2000. In 2000, he joined ICT, CAS, as a Professor, where he is currently the Deputy Director of the State Key Laboratory of Computer Architecture. He has coauthored over 280 papers in journals and international conferences, and he holds 60 patents and 30 software copyrights. His current research interests include VLSI testing, design for testability, design verification, dependable computing, and wireless sensor networks.

Prof. Li has been the Vice Chair of the IEEE Asia and Pacific Regional Test Technology Technical Council since 2004. He was the Chair of the Technical Committee on Fault-Tolerant Computing, China Computer Federation from 2008 to 2012, and the Steering Committee Chair of IEEE Asian Test Symposium from 2011 to 2013. He was the Steering Committee Chair of IEEE Workshop on RTL and High Level Testing from 2007 to 2010. He serves as an Associate Editor for the *Journal of Computer Science and Technology*, the *Journal of Low Power Electronics*, the *Journal of Electronic Testing: Theory and Applications*, and the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS.