AIMS Mathematics, 10(7): 15755–15784.
DOI: 10.3934/math.2025706
Received: 06 April 2025
Revised: 30 June 2025
Accepted: 03 July 2025
Published: 10 July 2025

https://www.aimspress.com/journal/Math

Research article

Optimizing Tensor-Train Decomposition for efficient edge AI: Accelerated decoding via GEMM and reshape minimization

Hyunseok Kwak, Sangmin Jeon, Kyeongwon Lee and Woojoo Lee*

Department of Intelligent Semiconductor Engineering, Chung-Ang University 84, Heukseok-ro, Dongjak-gu, Seoul, 06974, Korea

* **Correspondence:** Email: space@cau.ac.kr.

Abstract: Tensor-Train Decomposition (TTD) has emerged as a powerful mathematical framework for compressing neural network models in edge-oriented deployments, significantly reducing communication overhead between cloud environments and resource-constrained edge devices. However, its widespread adoption is hindered by the substantial computational overhead of decoding compressed parameters on edge hardware. In this paper, we experimentally demonstrate and mathematically validate that TTD achieves superior compression efficiency and better accuracy retention compared to conventional pruning methods, particularly when fine-tuning is impractical. To overcome the critical decoding bottleneck, we propose a mathematically rigorous yet hardware-aware optimization framework specifically tailored for efficient TTD-based deployments. Our approach leverages existing General Matrix Multiplication (GEMM) accelerators, commonly available in modern edge processors, to substantially accelerate the computationally intensive *Einsum* operations inherent in TTD decoding. Furthermore, we analytically identify redundant *reshape* operations between the decoding and inference stages, introducing a novel merging strategy that significantly reduces memory-bound overhead. Evaluations on a Field-Programmable Gate Array (FPGA)-based edge inference processor show substantial improvements, including a 3× speedup in reshape operations and a 69.3% decrease in decoding time. By seamlessly integrating rigorous mathematical formulation, analytical justification, and practical hardware optimization, this work paves the way for the efficient real-world deployment of TTD-compressed models on edge devices.

Keywords: Tensor-Train Decomposition (TTD); edge AI; model compression; GEMM accelerator; reshape optimization; Cloud-to-Edge deployment; FPGA prototyping
Mathematics Subject Classification: 15A69, 65F30, 68T07

1. Introduction

Artificial Intelligence (AI) has rapidly solidified its status as a core technology in diverse areas—including autonomous driving, healthcare, and mobile applications—owing to its remarkable capability to learn from massive data and make real-time decisions [1, 2]. While cloud-based AI models have traditionally handled both training and inference tasks in centralized data centers, ever-growing workloads and data transfers pose critical challenges in terms of network latency, energy consumption, and data privacy [3, 4]. To address these issues, a paradigm shift toward on-device inference has become essential for systems such as autonomous vehicles, wearable devices, and Internet of Things (IoT) sensors that demand immediate responses under strict resource constraints [5, 6]. More recently, research efforts have progressed beyond on-device inference to on-device training, where even the learning process is carried out locally on edge devices [7, 8]. Although this approach minimizes interactions with the cloud—enhancing real-time responsiveness and safeguarding user privacy—it remains hampered by the limited computational power and storage space of edge devices, which restricts the size of trainable models and degrades learning speeds [9, 10].

By contrast, a hybrid scheme that trains large-scale models on the cloud and performs only inference on edge devices is still widely adopted. It allows for leveraging computationally intensive resources in the cloud to train more complex and accurate models while flexibly distributing workloads between the cloud and edge [11, 12]. Nevertheless, this cloud-training–edge-inference approach continues to incur substantial transmission overhead when periodically distributing trained parameters to edge devices. Specifically, the energy budget of an edge device is often too limited to accommodate repeated exchanges of large amounts of data. Hence, minimizing communication energy consumption remains a pressing challenge in cloud-to-edge model deployment [13, 14].

Various parameter compression techniques have been investigated to reduce the size of transmitted models, ranging from classical pruning [15–17] and tensor decomposition methods [18–20], to emerging quantization techniques [21–23], knowledge distillation [24–26], and even hybrid approaches combining multiple strategies [28–30]. Among these techniques, the *Tensor-Train Decomposition* (*TTD*) [31] stands out due to its mathematical elegance and effectiveness in significantly reducing model size while preserving the original structure of the neural network. By approximating large tensors through products of lower-order tensors, TTD can achieve high compression ratios with minimal performance loss—making it particularly well-suited for resource-constrained edge devices [32–34]. However, deploying TTD in practice requires an additional decoding step at the device end, which recreates the original model parameters from compressed factors. This decoding overhead is the cost paid to save transmission energy, and its mitigation through efficient decoding algorithms is paramount for practical use of TTD-based compression.

To address these challenges, this paper provides a mathematically rigorous yet practically motivated approach that leverages existing *General Matrix Multiplication* (*GEMM*) hardware units commonly found in modern AI edge processors. We propose utilizing GEMM accelerators to speed up the *Einsum* operations inherent in the TTD decoding process. Furthermore, we introduce a novel optimization strategy that analytically identifies and merges redundant *reshape* operations occurring between decoding and inference phases. This optimization significantly reduces memory-bound overhead, achieving substantial computational savings. Since our proposed optimization operates directly on the Tensor-Train (TT) format structure, it can be readily combined with hybrid compression methods,

including those that integrate TT-compatible quantization or pruning.

To validate the theoretical foundations and effectiveness of our method, we implement a fully customized AI edge inference processor integrating a RISC-V Rocket core with a GEMM accelerator and verify its performance via FPGA prototyping. Experimental results confirm the proposed reshape optimization alone accelerates reshape operations by approximately threefold, while the combined use of GEMM acceleration and reshape optimization reduces the overall decoding time by up to 69.3%. Thus, our contributions can be succinctly summarized as follows:

- We establish theoretical underpinnings and provide experimental analyses demonstrating the balance between compression efficiency and accuracy achievable by TTD in cloud-to-edge deployments. Based on these insights, we design a rigorous optimization framework to enhance the decoding efficiency of TTD-compressed models.
- We demonstrate the reuse of existing GEMM hardware to significantly accelerate the Einsum operations critical to TTD decoding, thereby alleviating computational burdens without additional specialized hardware.
- We mathematically identify and analytically prove the redundancy in reshape operations between decoding and inference stages, proposing an optimized approach to merge these operations, which dramatically reduces memory-bound overhead.

Overall, this paper bridges the theoretical foundations of tensor decomposition techniques and practical hardware optimizations, offering an efficient and robust solution for the real-world deployment of TTD-compressed AI models on resource-constrained edge devices.

2. Comparative analysis of model compression via Tensor-Train decomposition: A cloud-to-edge perspective

In a cloud-based training and edge-based inference (cloud-to-edge) AI paradigm, large volumes of model parameters must be periodically distributed from the cloud to edge devices. To reduce transmission time and energy consumption during these deployments, numerous model-reduction and compression techniques have been investigated. Representative examples include *weight pruning*, which removes redundant or low-importance weights from neural networks [15–17], and *tensor decomposition*, which reduces the dimensionality of model parameters [18–20]. Additionally, techniques such as *quantization*, which reduce the precision of weights and activations to lower bit-width representations (e.g., 8-bit or 4-bit integers) [21–23], and *knowledge distillation*, which transfers learned representations from a large (teacher) model to a smaller (student) model while maintaining accuracy [24–26], have also been widely studied.

Unlike cloud environments, however, edge devices typically have stringent constraints on hardware performance and memory capacity. Thus, effective parameter compression for cloud-to-edge inference demands not only efficient transmission but also the mitigation of computational overhead arising from decoding on the device.

Weight pruning is conventionally categorized into *unstructured* and *structured* pruning [35]. Unstructured pruning removes individual weights deemed unimportant, achieving high compression ratios but resulting in irregularly sparse matrices. Such irregular sparsity hinders efficient matrix

operations unless specialized accelerators are utilized, making it less ideal for edge environments [36]. By contrast, structured pruning eliminates weights in coarse-grained units (e.g., channels, layers, or blocks), producing structurally sparse matrices that facilitate efficient hardware implementation [35]. Notably, *channel pruning*, which compresses convolutional layers by removing specific input or output channels, retains a regular shape conducive to optimized edge inference [37].

Quantization techniques typically include Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ). QAT employs "fake quantization", representing weights and activations in lower-precision integer formats during training, while retaining floating-point computations and full-precision gradients to minimize accuracy degradation [28]. By contrast, PTQ quantizes pretrained models without fine-tuning but requires a calibration step with a small dataset to collect activation statistics [22]. Post-quantization, weights and activations are stored as low-precision integers, often with distinct scaling factors across tensors. This necessitates a computationally expensive requantization step (dequantization–rescaling–quantization) when merging tensors, significantly degrading inference performance unless dedicated hardware accelerators are used, which may be impractical in edge scenarios due to hardware complexity [23, 38].

Knowledge Distillation (KD) trains a lightweight student model by transferring knowledge from a larger pretrained teacher model, typically combining soft labels (KL divergence) and hard labels (cross-entropy loss) [39]. Despite improved accuracy, KD imposes substantial memory and computational overhead by requiring the large teacher model during training [25, 26]. Additionally, the student model's performance is inherently limited by the teacher model's capabilities, potentially inheriting biases detrimental to generalization [27].

Tensor decomposition methods, in parallel, have also been extensively explored. Representative examples include Canonical Polyadic Decomposition (CPD), Tucker decomposition, and Tensor-Train Decomposition (TTD) [40–43]. CPD approximates high-dimensional tensors as sums of outer products of vectors, theoretically offering high compression ratios. However, CPD often struggles to converge reliably, complicating optimal rank determination. Its decoding process involves repeated outer-product computations, introducing significant complexity for edge implementations [44]. Tucker decomposition uses a core tensor and associated mode matrices, allowing flexible dimension-wise compression. Nevertheless, the retained high-dimensional core tensor leads to substantial memory and decoding overhead on edge devices [45, 46].

By contrast, TTD expresses a large-scale tensor as a product of lower-dimensional *core tensors* [31]. In addition to generally delivering superior compression ratios among decomposition methods [47], TTD preserves essential structural properties of the data while yielding a compact representation. This makes TTD highly effective in large-scale data handling, storage, and transmission [48]. Importantly, decoding from TTD does not require maintaining a high-dimensional core tensor as in Tucker decomposition; one may recover the original tensor by sequentially multiplying several small core tensors, thereby reducing both computational and memory requirements on the edge device.

Based on these advantages, we selected *channel pruning* and *TTD*—both recognized for their suitability in edge devices—as main candidates for parameter compression. We then evaluated each method's compression effectiveness and its effect on accuracy using practical image classification tasks. Because compression ratio directly influences the on-device memory usage, it is a key factor for assessing whether a compression technique is viable on resource-constrained edge hardware. In our experiments, we used the Canadian Institute for Advanced Research (CIFAR)-10 and CIFAR-100

datasets and employed ResNet18, SENet18, and GoogLeNet as representative convolutional neural networks.

Model parameter compression performance is typically evaluated after a fine-tuning phase that recovers accuracy lost during compression [49, 50]. However, even under cloud environments, fine-tuning often demands substantial time and computational resources; consequently, techniques that omit fine-tuning yet minimize accuracy degradation have received increasing attention [51]. This consideration becomes particularly relevant in cloud-based model deployment, where the feasibility of fine-tuning can vary widely. Indeed, whether one can afford fine-tuning or not significantly influences the effectiveness of a compression scheme. Hence, we evaluate both the scenario that includes fine-tuning and the scenario that omits it (anticipating heavy computational loads in the cloud) for channel pruning and TTD, comparing compression ratios and accuracy across both conditions.

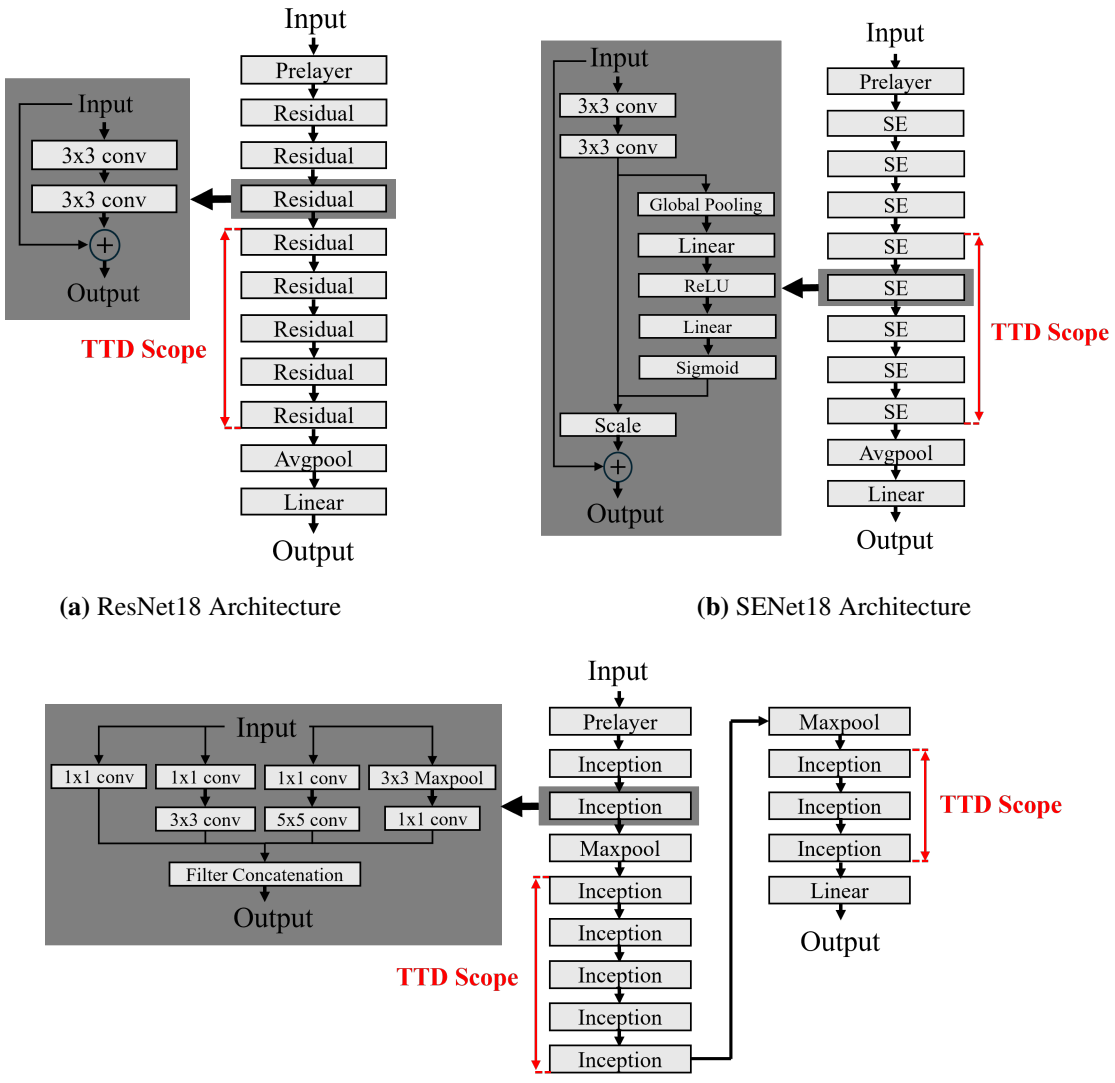
Figure 1 illustrates the ResNet18, SENet18, and GoogLeNet architectures employed in our experiments. Each model is built primarily on convolutional layers and contains one or more key structural blocks: the *Residual* block (ResNet), the *Squeeze-and-Excitation* block (SENet), and the *Inception* block (GoogLeNet). We apply channel pruning to every convolutional layer, removing channels of the lowest importance based on a specified pruning ratio. In contrast, TTD is most beneficial for layers with significantly more parameters than preceding layers; targeting layers with smaller parameter counts yields minimal additional compression. Accordingly, we define a *TTD scope* in Figure 1 that includes continuous sets of parameter-heavy convolutional layers for decomposition.

Algorithm 1 outlines our proposed approach for determining the optimal TTD Scope. The algorithm starts from the last convolutional layer—typically containing the most parameters—and incrementally expands the TTD Scope towards earlier layers. This expansion continues until no further compression gains are observed, thus identifying the optimal set of layers for applying TTD. Figure 2 illustrates how incremental expansion of the TTD Scope (denoted as B1-x, starting from the last convolutional block) affects the compression ratio. As the scope expands upwards through layers, compression gains initially increase but eventually diminish, clearly indicating the optimal point at which to define the TTD Scope for each architecture.

Algorithm 1 Determining the Optimal TTD Scope for Maximum Compression.

Input M : a Convolutional Neural Network (CNN) model with B basic blocks $\{\text{block}_1, \ldots, \text{block}_B\}$
Input ϵ_{init} : initial truncation tolerance for TTD
Output S : set of blocks guaranteeing maximal compression ratio under the given ϵ_{init}

$S \leftarrow \{\}$
$CR_{prev} \leftarrow 1$
Count the parameters in the entire model M to obtain P_{orig}.
for $i : 1$ **to** B **do**
    $S' \leftarrow \{S, \text{block}_i\}$
    Apply TTD to S' with ϵ_{init} and count the compressed parameters P_{comp}.
    $CR \leftarrow P_{\text{orig}} / P_{\text{comp}}$
    if $CR \leq CR_{prev}$ **then**
        break // No further compression gain
    end if
    $CR_{prev} \leftarrow CR$
    $S \leftarrow S'$
end for
return S



(a) ResNet18 Architecture

(b) SENet18 Architecture

(c) GoogLeNet Architecture

Figure 1. Image classification model architectures utilized in our experiments to systematically evaluate and compare compression performance between Channel Pruning and TTD.

The proposed Algorithm 1 can generally be applied to most models composed of consecutive convolutional layers, quickly determining the TTD scope based solely on structural information without additional accuracy assessments or sensitivity analyses. However, in lightweight architectures like MobileNet, solely relying on the compression ratio for TTD scope selection could include accuracy-sensitive blocks, potentially resulting in significant accuracy degradation. Therefore, additional steps such as accuracy evaluation or sensitivity analysis might be required. In contrast, deeper architectures such as ResNet50 typically include structurally redundant blocks, making the proposed approach comparatively more stable.

Once the optimal TTD Scope is identified, the next important step is to determine an appropriate

truncation tolerance (ϵ), which directly controls the compression effectiveness and accuracy. Unlike channel pruning, whose compression ratio is straightforward to predict from the predefined pruning ratio, TTD's compression ratio hinges on the extent of *singular-value truncation* performed in the iterative *Singular Value Decomposition* (*SVD*) procedure. An input threshold ϵ determines how aggressively small singular values are truncated in each step, thus affecting the resultant compression ratio. Although a larger ϵ is expected to result in higher compression, it is difficult to determine *a priori* the exact ratio. Regarding this, we performed a detailed mathematical analysis of how ϵ affects approximation error and compression ratio, which is deferred to Section 4.

Figure 3 compares channel pruning (with different pruning ratios) and TTD (with different values of ϵ) for each network architecture, reporting both accuracy and compression ratio. We observe that increasing the pruning ratio or ϵ generally improves compression at the cost of accuracy reduction, albeit with differing behaviors between the two approaches. In channel pruning, the accuracy drop is fairly linear with respect to the pruning ratio. By contrast, TTD sometimes exhibits certain threshold points of ϵ beyond which accuracy degrades more sharply. For example, ResNet18 and SENet18 maintain a modest loss (below 2%) until $\epsilon \approx 0.1$, at which point compression ratios around 3× are reached. GoogLeNet shows a similar behavior, sustaining around 2.4× compression at $\epsilon \approx 0.14$ with only 1% accuracy loss. This underlines the importance of selecting an appropriate ϵ for TTD-based compression.

Table 1 summarizes comparative results for TTD and channel pruning across three architectures. Top-1 accuracy for CIFAR-10 and both top-1 and top-5 accuracies for CIFAR-100 are reported. Particularly notable are the cases where *fine-tuning* is omitted, as fine-tuning is often infeasible in practical cloud-to-edge scenarios due to significant computational and temporal overhead. We particularly highlight the following findings on CIFAR-100 without fine-tuning:

- **ResNet18**: TTD achieves 1.81× compression while sustaining 70.17% top-1 accuracy, a mere 1.25% drop from baseline. By contrast, channel pruning with 1.02× compression degrades accuracy by 3.33%.

- **SENet18**: TTD attains 1.75× compression and preserves top-1 (69.97%) and top-5 (90.20%) accuracy within 0.91% and 0.47% of baseline, respectively. Channel pruning manages only 1.03× compression yet drops top-1 accuracy by 3.64%.

- **GoogLeNet**: TTD provides 1.54× compression while achieving 70.44% top-1 and 90.58% top-5 accuracy, only 1.70% and 0.57% below baseline. Channel pruning, in contrast, compresses to 1.03× but suffers losses of 5.04% (top-1) and 2.77% (top-5).

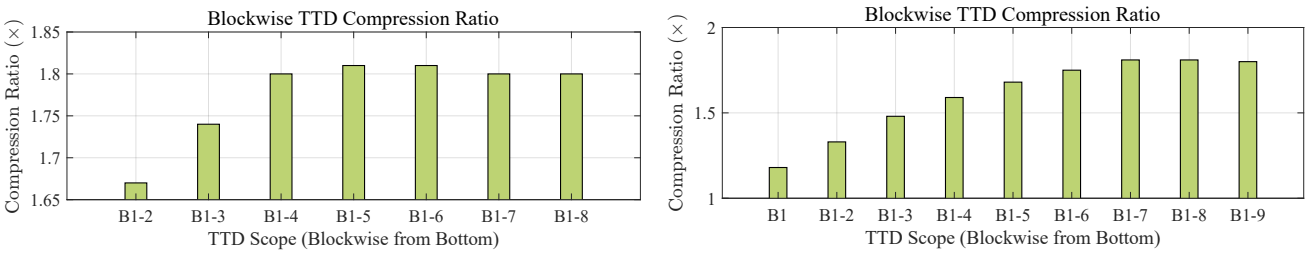
In particular, for lightweight applications with fewer classification categories, such as CIFAR-10, the advantages of TTD become even more pronounced in scenarios without fine-tuning. For instance, ResNet18 achieves a significant compression ratio of 2.87× while incurring an accuracy loss of only 0.76% compared to the baseline. Similarly, SENet18 and GoogLeNet maintain accuracy losses below 1% at compression ratios of 2.80× and 2.62×, respectively. These results consistently demonstrate that TTD provides higher compression efficiency and smaller accuracy degradation than channel pruning when fine-tuning is not performed.

The experimental findings in Table 1 clearly illustrate that TTD consistently outperforms channel pruning, not only in scenarios where fine-tuning is possible, but especially in realistic cloud-to-edge

deployment contexts, where fine-tuning might be impractical due to computational and resource constraints. Thus, TTD enables significant parameter reduction while minimizing accuracy loss, thereby effectively alleviating the computational and temporal overhead associated with frequent model deployment to edge devices. This is in stark contrast to pruning-based compression techniques, which rely heavily on fine-tuning to ensure acceptable accuracy after compression [52, 53]. Motivated by these experimental results, subsequent sections of this paper focus primarily on TTD-based model compression, investigating strategies optimized specifically for efficient model transmission and inference in edge device environments.

Table 1. Comparative performance of TTD and channel pruning across the three different architectures on CIFAR-10 and CIFAR-100 datasets, highlighting compression ratio, top-1 accuracy, and top-5 accuracy, with and without fine-tuning.

Model	Fine-tuning	Compression Ratio	CIFAR-10 Top-1 ACC.	Compression Ratio	CIFAR-100 Top-1 ACC.	CIFAR-100 Top-5 ACC.
ResNet18-Baseline	-	1.0×	93.74	1.0×	71.42	90.96
ResNet18-TTD	◯	4.09×	93.26	2.76×	69.45	90.76
ResNet18-Pruning	◯	1.49×	92.60	1.43×	69.20	89.72
ResNet18-TTD	×	2.87×	92.98	1.81×	70.17	90.89
ResNet18-Pruning	×	1.06×	92.89	1.02×	68.09	88.80
SENet18-Baseline	-	1.0×	93.10	1.0×	70.88	90.67
SENet18-TTD	◯	3.71×	93.16	2.70×	69.50	89.66
SENet18-Pruning	◯	1.42×	92.50	1.42×	69.77	89.85
SENet18-TTD	×	2.80×	92.22	1.75×	69.97	90.20
SENet18-Pruning	×	1.07×	92.43	1.03×	67.24	88.66
GoogLeNet-Baseline	-	1.0×	94.45	1.0×	72.14	91.15
GoogLeNet-TTD	◯	3.69×	93.83	2.27×	71.05	90.88
GoogLeNet-Pruning	◯	1.58×	92.75	1.64×	69.84	90.70
GoogLeNet-TTD	×	2.62×	93.52	1.54×	70.44	90.58
GoogLeNet-Pruning	×	1.14×	93.53	1.03×	67.10	88.38



(a) ResNet18 **(b)** GoogLeNet

Figure 2. Compression ratio variation with incremental expansion of TTD Scope in different CNN architectures.

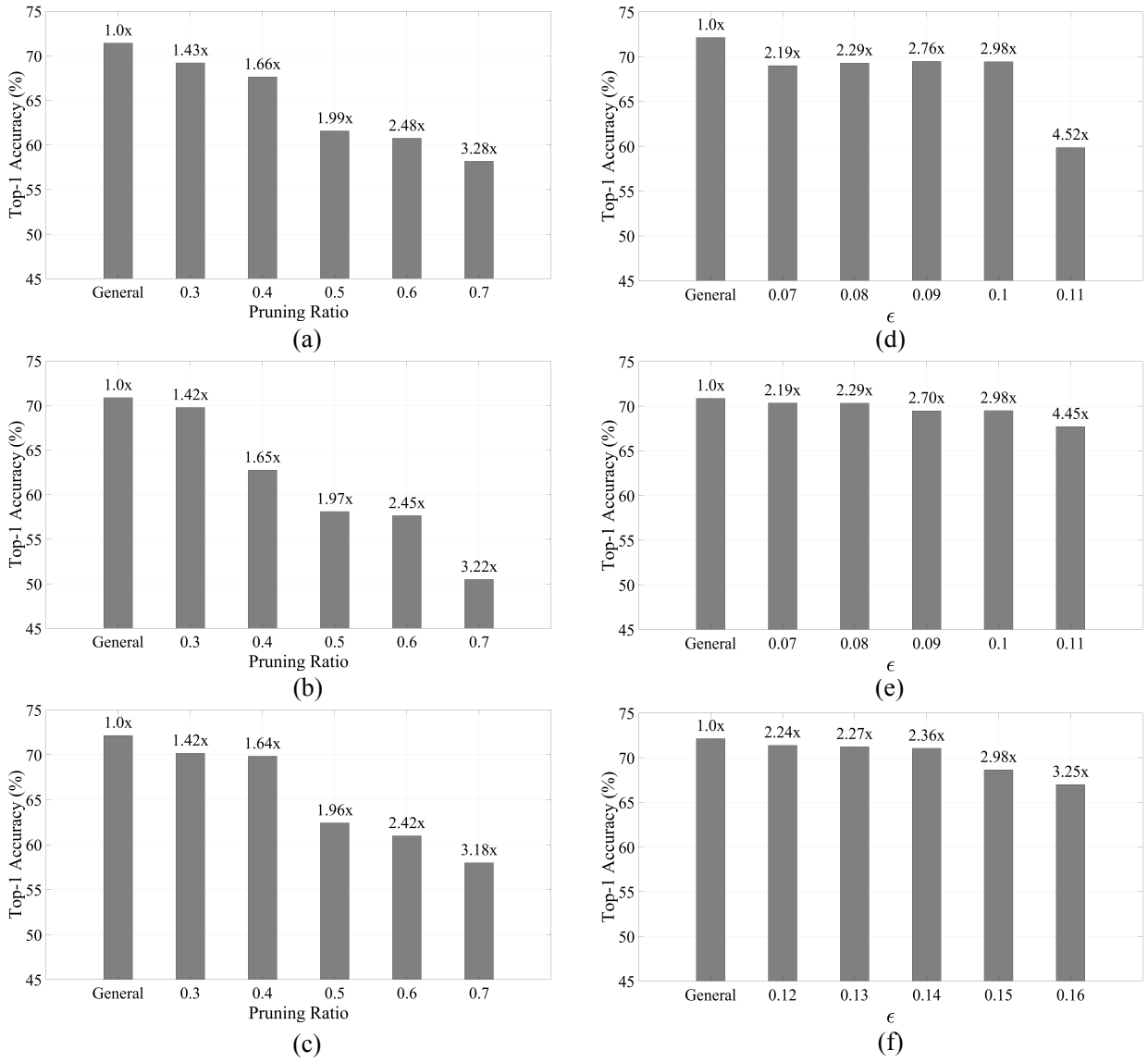


Figure 3. Experimental results comparing accuracy and compression ratio under various compression hyperparameters: (a)–(c) Channel pruning and (d)–(f) TTD applied to ResNet18, SENet18, and GoogLeNet models, respectively.

3. Optimization of TTD decoding for edge deployment

In the TTD framework for cloud-to-edge model deployment, the cloud first compresses a trained model's parameters using TTD and then transmits the compressed tensors to an edge device. Upon receiving these factors, the device decodes them back into the original model parameters and proceeds with inference. Figure 4 illustrates this sequence of operations, highlighting that the *decoding* step performed at the edge is the major computational overhead associated with TTD. This overhead not only counteracts the energy savings gained by TTD's reduced communication footprint but also increases the computational burden on edge devices. As such, efficient decoding is a critical

prerequisite for practical utilization of TTD-based model compression.

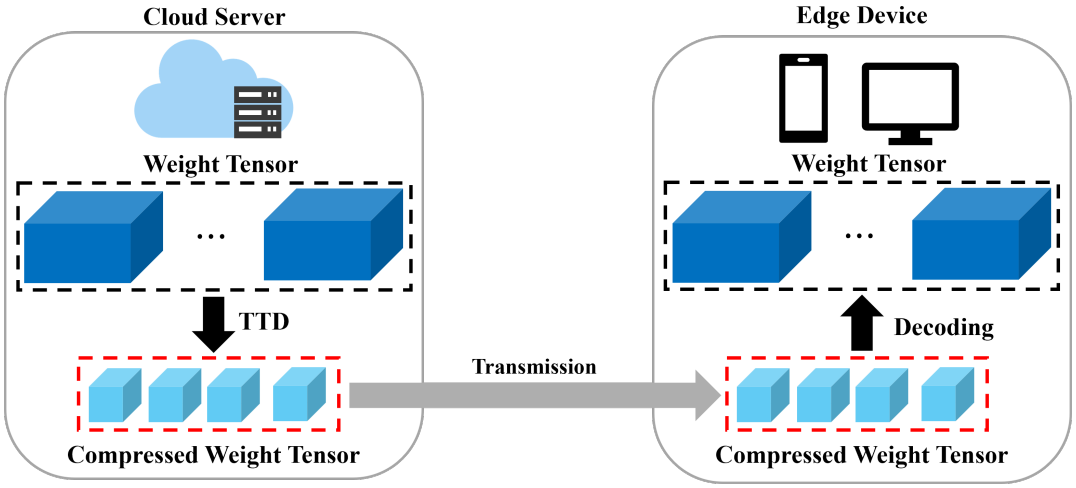


Figure 4. Cloud-to-edge model deployment using TTD. At the edge device, *decoding* introduces substantial computational overhead.

In this section, we focus on an image-classification task running on a resource-constrained edge device. Specifically, we examine how TTD-compressed parameters, transmitted from the cloud, are decoded and subsequently used for inference, and we propose a novel optimization framework to accelerate this decoding process.

Decoding Overhead and Its Impact. Convolutional layer parameters targeted for TTD compression are typically represented as a four-dimensional (4D) tensor $\mathbf{W} \in \mathbb{R}^{D\times C\times H\times W}$, where D denotes the number of output channels, C the number of input channels, and H, W the height and width of the convolution kernel, respectively. Once TTD is applied to a convolution layer, the original tensor can be approximated by a product of four core tensors,

$$\begin{aligned} &\mathbf{W} \approx \mathbf{G}_1 \cdot \mathbf{G}_2 \cdot \mathbf{G}_3 \cdot \mathbf{G}_4, \\ &\mathbf{G}_1 \in \mathbb{R}^{r_1'\times D\times r_1}, \quad \mathbf{G}_2 \in \mathbb{R}^{r_1\times C\times r_2}, \quad \mathbf{G}_3 \in \mathbb{R}^{r_2\times H\times r_3}, \quad \mathbf{G}_4 \in \mathbb{R}^{r_3\times W\times r_4}, \end{aligned} \tag{3.1}$$

where r_k represents the *TT-rank*, determined by the singular-value truncation threshold ϵ.

By mathematical construction, the boundary ranks are always set to $r_1' = r_4 = 1$ to ensure correct tensor reconstruction. Figure 5 illustrates how intermediate TT-ranks r_1, r_2, and r_3 vary with respect to the truncation threshold ϵ. As ϵ increases, each TT-rank decreases at a distinct rate.

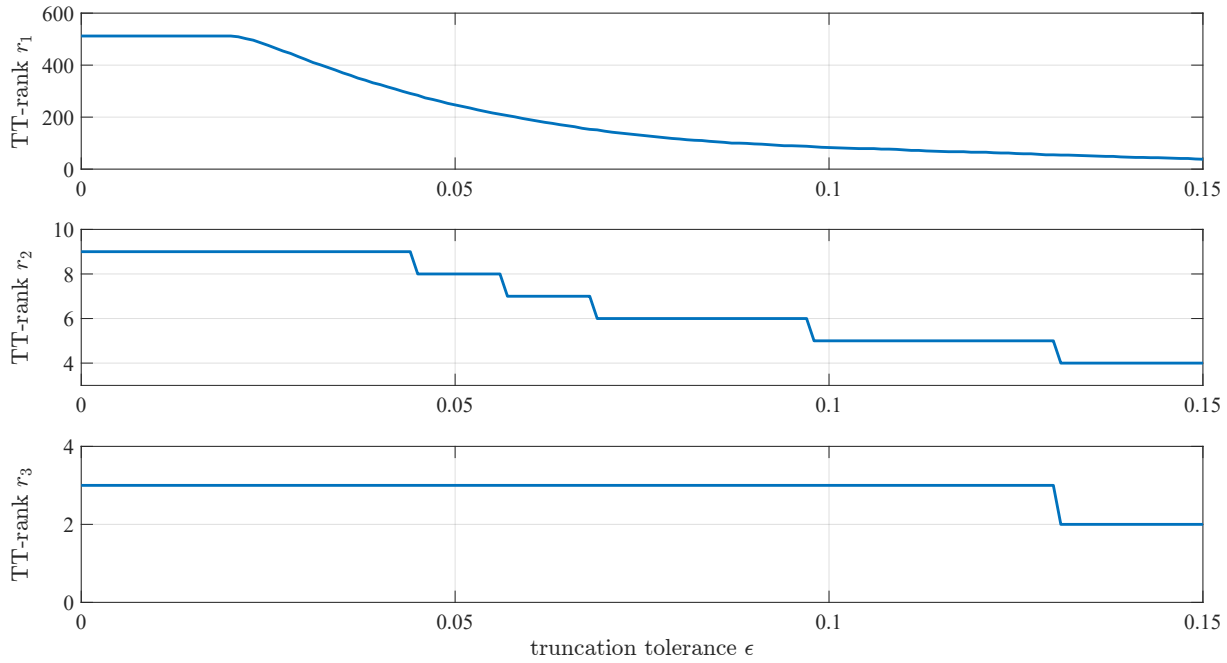


Figure 5. Variation of intermediate TT-ranks (r_1, r_2, and r_3) with respect to the truncation tolerance ϵ.

Meanwhile, the impact of varying ϵ on model accuracy is presented earlier in Figure 3 (d)–(f). These figures demonstrate a clear trade-off relationship: as ϵ increases, compression ratios improve but accuracy gradually decreases. By referencing both Figure 5 (TT-rank variations) and Figure 3 (accuracy results), readers can comprehensively understand how changes in ϵ simultaneously influence compression efficiency and model performance.

On the edge device, the decoding phase reconstructs the original model parameters by multiplying these core tensors together, culminating in an overhead that can negate the energy savings derived from TTD's lighter communication. Thus, to make TTD truly viable in real-world scenarios, we must minimize this computational overhead at the edge.

Einsum and Reconstruction. Decoding of TTD-compressed parameters on edge devices consists of two primary stages:

(i) *Einsum Stage*: This stage involves reconstructing the original model parameters through repeated matrix multiplications (Einsum operations) applied to the core tensors ($\mathbf{G}_1$, $\mathbf{G}_2$, $\mathbf{G}_3$, and $\mathbf{G}_4$) obtained from the TTD compression. To facilitate these multiplications, each core tensor is reshaped into a suitable 2D matrix form.

(ii) *Reconstruction Stage*: Once the original parameters have been recovered in matrix form, they are reshaped back into the original high-dimensional tensor format (e.g., 4D tensor for convolutional layers), making them ready for subsequent inference.

These repeated reshape operations, as illustrated in Figure 6, do not alter the underlying data but reorganize it into new shapes. Although reshaping itself is computationally inexpensive, it is memory-bound, potentially becoming a significant performance bottleneck when performed frequently during

decoding. Additionally, Figure 7 conceptually demonstrates that even decoding a single convolutional layer requires multiple matrix multiplications. Extending this across the entire neural network leads to considerable computational overhead. Therefore, accelerating these repeated Einsum operations is essential to improve the overall decoding efficiency on resource-constrained edge devices.

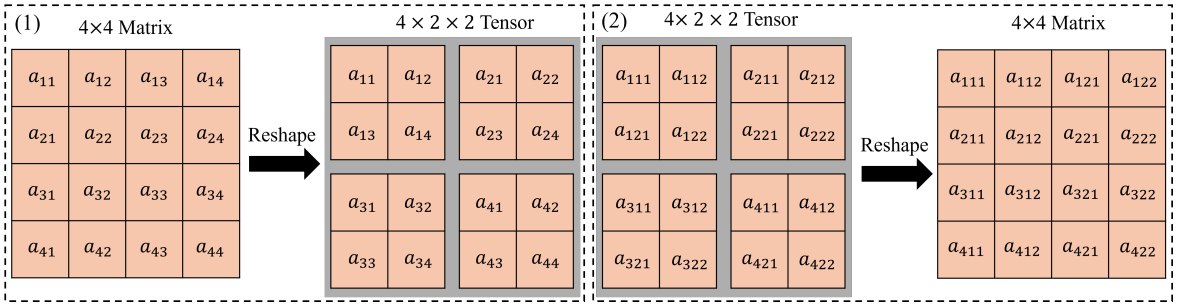


Figure 6. Illustration of tensor reshaping operations: (1) Reshaping a matrix into a three-dimensional tensor, and (2) Reshaping a three-dimensional tensor back into a matrix.

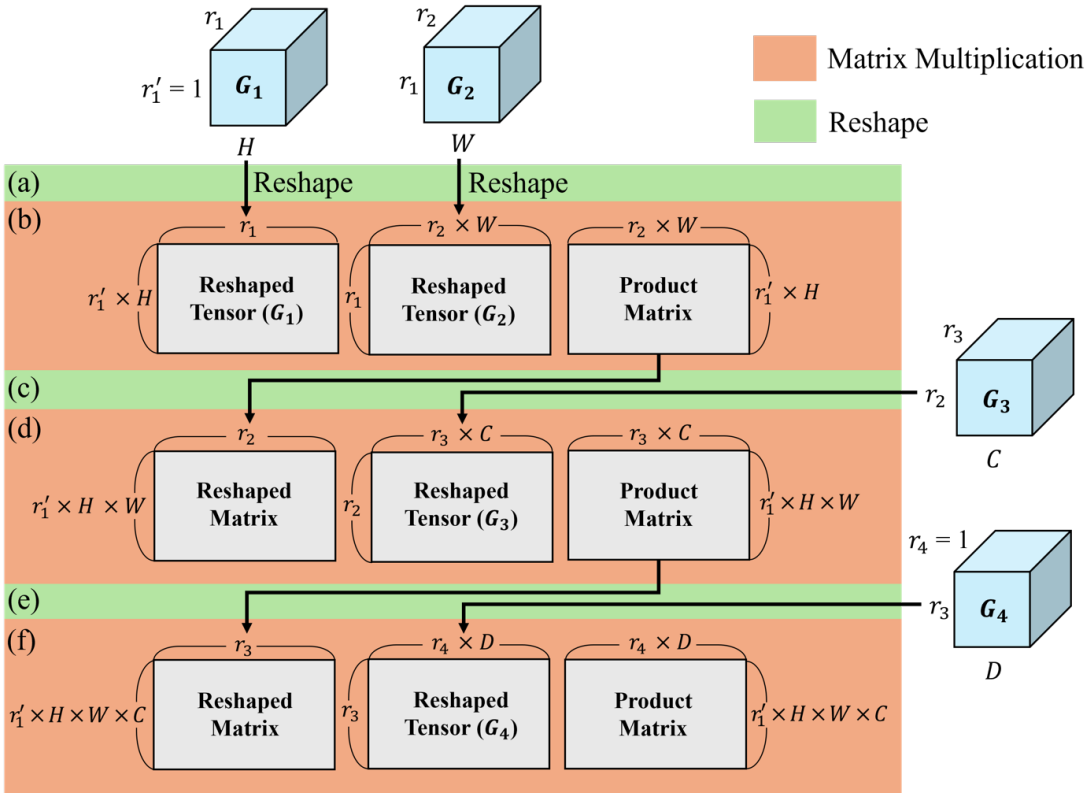


Figure 7. Conceptual illustration of the Einsum operation during decoding: (a) Reshaping G_1 and G_2, (b) Performing a matrix multiplication, (c) Reshaping the result with G_3, (d) Another matrix multiplication, (e) Reshaping the result with G_4, and (f) Final matrix multiplication.

Leveraging GEMM at the Edge. A straightforward yet effective way to accelerate these repeated matrix multiplications is to exploit the GEMM hardware unit already present in many edge AI processors. Modern accelerators include specialized GEMM Intellectual Property (IP) blocks for high-throughput processing of deep networks such as CNNs and Transformers. Our key insight is that the same GEMM engine can be reused for TTD decoding. Since inference and decoding do not occur simultaneously, the device can dedicate this GEMM engine exclusively to decoding during deployment, thus avoiding the need for additional custom hardware.

Figure 8 illustrates the end-to-end inference process using TTD-compressed model parameters on an edge device. In the proposed method, both the Einsum operation during decoding (**D1**) and the matrix multiplication in the inference stage (**I2**) utilize the GEMM IP to enhance computational efficiency. Notably, during inference, the reconstructed weight tensor from the decoding stage is convolved with the input tensor using an image-to-column (*IM2COL*) transformation.

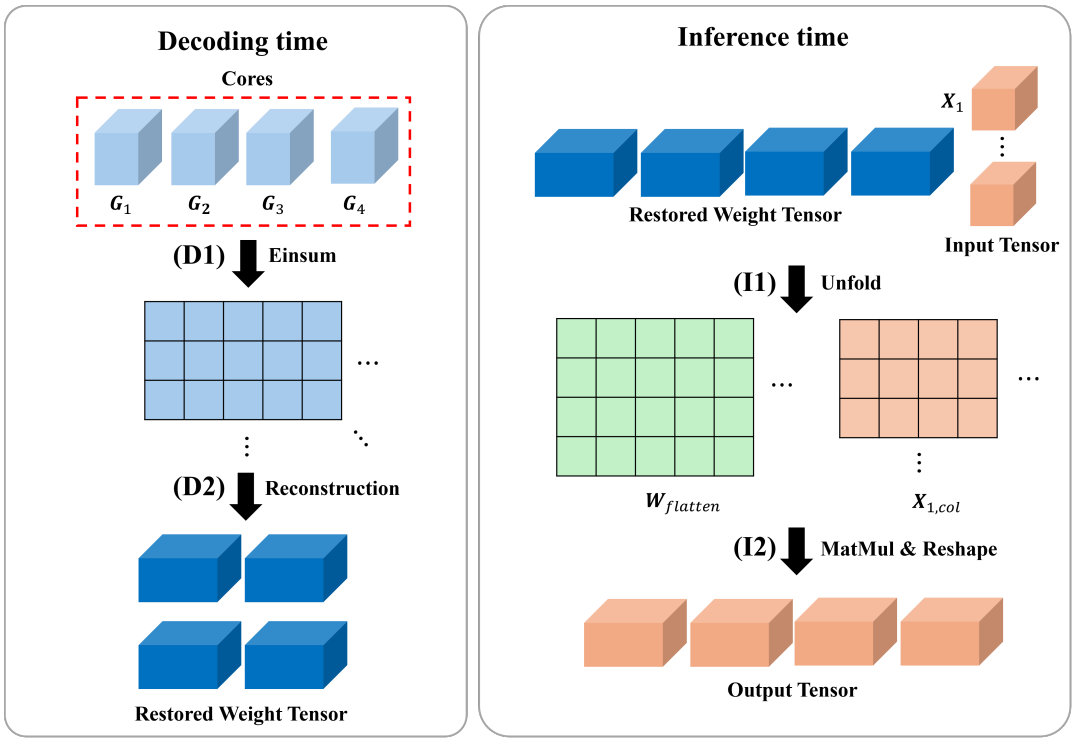


Figure 8. Overview of the cloud-to-edge inference procedure under TTD-based compression: (**D1**) Einsum for decoding, (**D2**) Reconstruction to original tensor, (**I1**) IM2COL for matrix transformation, and (**I2**) Matrix multiplication-based convolution.

IM2COL is a software-based technique that transforms convolution operations into GEMM. It is widely adopted in major deep learning frameworks such as TensorFlow and PyTorch. By flattening high-dimensional input tensors into 2D matrices, *IM2COL* allows complex tensor-based convolutions to be executed as simpler and highly optimized matrix multiplications [54]. Since many edge devices are already equipped with built-in GEMM IPs capable of performing matrix multiplications in parallel, *IM2COL* enables efficient convolution execution without requiring dedicated hardware units. Another advantage of *IM2COL* is its generality—it consistently converts convolution layers into a unified matrix

multiplication pattern regardless of kernel size, stride, or padding. This property simplifies hardware design and ensures better resource utilization by providing a consistent execution path across various convolution settings. As such, *IM2COL* offers a practical and essential approach for achieving both computational efficiency and hardware reusability on resource-constrained edge devices.

Challenge: Redundant Reshape Operations. While decoding with a GEMM engine is advantageous, it introduces a subtle issue when combined with *IM2COL*. As shown in Figure 8, after the Einsum stage, we reshape the result back into a 4D weight tensor $\tilde{\mathbf{W}}$ in step (**D2**). In the subsequent inference step, (**I1**), *IM2COL* again reshapes $\tilde{\mathbf{W}}$ to a 2D matrix in order to perform convolution. Hence, the same tensor is reshaped twice in a row, effectively duplicating effort and wasting memory bandwidth. To alleviate this, we propose merging these two reshape operations into a single transformation—effectively omitting the 4D reshaping step during decoding. The result of the Einsum stage can be directly cast into the 2D format required by *IM2COL*, as illustrated in Figure 9.

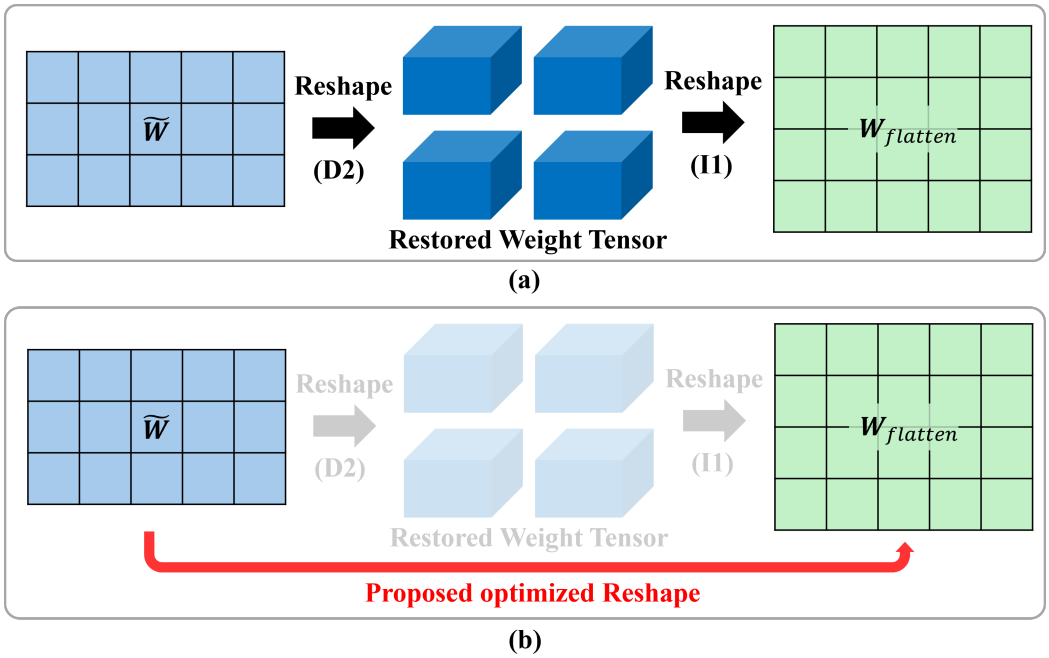


Figure 9. Proposed reshape optimization to eliminate redundant operations between decoding and inference stages: (a) Baseline approach involving two separate reshape operations, and (b) Optimized approach merging them into a single reshape step.

Summary of Proposed Optimization. Summarizing the above discussion, our approach provides three main contributions:

(i) We propose leveraging the GEMM engine present on edge devices to significantly accelerate the Einsum stage of TTD decoding, thus reducing computational overhead without additional hardware.

(ii) We identify redundant reshape operations between the decoding (Einsum) and inference (*IM2COL*) stages, analyzing how these repeated operations inflate computational overhead and

execution time.

(iii) To address this, we introduce a novel reshape optimization strategy that merges consecutive reshape steps into a single transformation, entirely implementable in software. As illustrated in Figure 9, this approach removes unnecessary reshaping, reducing reshape operation time by up to 66% according to our experimental results.

This optimization preserves the primary benefit of TTD—reduced communication cost—while substantially enhancing decoding and inference performance on resource-constrained edge devices. The following section rigorously analyzes the mathematical foundations underpinning this approach, providing theoretical justification for its general applicability to typical network architectures.

4. Mathematical justification of proposed optimizations

This section provides a mathematical foundation of TTD, focusing on two main aspects. First, we investigate how the user-defined truncation threshold ϵ determines the global reconstruction error and overall compression ratio, thereby illuminating the role of ϵ as a fundamental tuning parameter in TTD. Second, we formally analyze the computational flow of TTD-based model compression and demonstrate that repeated *reshape* operations in both the decoding and inference stages can be merged into a single transformation. From a mathematical perspective, this confirms that the proposed reshape optimization goes beyond a mere heuristic and is valid for a broad class of neural network architectures and convolution operations.

1) TTD and Singular Value Decomposition. TTD factorizes a high-dimensional tensor into a product of lower-dimensional *core tensors*. To illustrate, consider first reshaping the original tensor into a 2D matrix, then applying *Truncated Singular Value Decomposition* (TSVD). After truncation, the partial results are appropriately reshaped into core tensors and the next-stage input tensor; this procedure repeats until the entire original tensor is expressed as a chain of core tensors.

(a) Bidiagonalization via Householder Transformations. Following the notation of [55], let $\mathbf{A} \in \mathbb{R}^{M\times N}$ $(M > N)$ be the matrix obtained by reshaping part of the original high-dimensional tensor. To transform $\mathbf{A}$ into a bidiagonal form, we require N Householder vectors derived by incrementing the column index, and $N-2$ additional Householder vectors derived by incrementing the row index. Concretely, for $k \in \{1,\ldots,N\}$,

$$a_{col}^{(k)} = [\mathbf{A}[k:M,\, k],\, 0,\ldots,0]^T, \tag{4.1}$$

$$u^{(k)} = a_{col}^{(k)} + \operatorname{sign}(a_{col}^{(k)}[1])\,\|a_{col}^{(k)}\|_2\, e_1, \tag{4.2}$$

$$x^{(k)} = \frac{u^{(k)}}{\|u^{(k)}\|_2}, \tag{4.3}$$

where $e_1 = [\,1,\,0,\ldots,0\,]^T$, and zeros are appended to match the dimension $\mathbb{R}^M$, and

$$\operatorname{sign}(x) = \begin{cases} +1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0. \end{cases} \tag{4.4}$$

Similarly, for $\ell \in \{1, \ldots, N-2\}$, one obtains a set of Householder vectors $\{y^{(\ell)}\}$ by processing rows of $\mathbf{A}$. From each Householder vector $x^{(k)}$ or $y^{(\ell)}$, we construct Householder matrices of the form

$$\mathbf{P}^{(k)} = \mathbf{I} - 2\frac{x^{(k)}\,x^{(k)T}}{x^{(k)T}\,x^{(k)}}, \quad \mathbf{Q}^{(\ell)} = \mathbf{I} - 2\frac{y^{(\ell)}\,y^{(\ell)T}}{y^{(\ell)T}\,y^{(\ell)}}. \tag{4.5}$$

By multiplying these Householder matrices in sequence, we obtain a bidiagonal matrix $\mathbf{B}^{(0)}$:

$$\begin{aligned} \mathbf{B}^{(0)} &= \mathbf{P}^{(N)} \cdots \mathbf{P}^{(1)}\,\mathbf{A}\,\mathbf{Q}^{(1)} \cdots \mathbf{Q}^{(N-2)}, \\ \mathbf{U}_1 &= \mathbf{P}^{(1)} \cdots \mathbf{P}^{(N)}, \qquad \mathbf{V}_1 = \mathbf{Q}^{(1)} \cdots \mathbf{Q}^{(N-2)}. \end{aligned} \tag{4.6}$$

Here, $\mathbf{U}_1$ and $\mathbf{V}_1$ are orthogonal matrices that normalize the rows and columns of $\mathbf{A}$, respectively, satisfying $\mathbf{B}^{(0)} = \mathbf{U}_1^T \mathbf{A}\,\mathbf{V}_1$.

(b) Diagonalization via Givens Rotations. Having obtained the bidiagonal matrix $\mathbf{B}^{(0)}$, we next apply *Givens rotations* [56] to reduce $\mathbf{B}^{(0)}$ to a diagonal matrix $\mathbf{\Sigma}$. Concretely, for $\mathbf{B}^{(0)} \in \mathbb{R}^{N\times N}$, we multiply by a sequence of left-rotation matrices $\mathbf{S}(n, n-1)$ and right-rotation matrices $\mathbf{T}(n, n-1)$, where n descends from N to 1. Each left-rotation matrix $\mathbf{S}(n, n-1)$ is associated with a rotation angle θ_n and takes the form:

$$\mathbf{S}(n, n-1) = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \cos\theta_n & \sin\theta_n & \cdots & 0 \\ 0 & 0 & \cdots & -\sin\theta_n & \cos\theta_n & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 1 \end{bmatrix}, \tag{4.7}$$

where the $(n-1, n)$-th block of $\mathbf{S}(n, n-1)$ incorporates $\cos\theta_n$ and $\sin\theta_n$ to eliminate subdiagonal elements in the n-th column of $\mathbf{B}^{(0)}$. In particular, if q_n denotes the diagonal element and e_n the subdiagonal element at step $n+1$, then

$$\cos\theta_n = \frac{q_n}{\sqrt{q_n^2 + e_n^2}}, \qquad \sin\theta_n = \frac{e_n}{\sqrt{q_n^2 + e_n^2}}. \tag{4.8}$$

For right-rotation matrices $\mathbf{T}(n, n-1)$, one employs a separate rotation angle ϕ_n but otherwise proceeds analogously to (4.7) and (4.8).

By applying these Givens rotations in sequence from $n = N$ down to $n = 1$, we generate intermediate transformations such that

$$\mathbf{B}^{(n+1)} = \mathbf{S}(n+1,\, n)^T \cdots \mathbf{S}(N,\, N-1)^T\,\mathbf{B}^{(0)}\,\mathbf{T}(N,\, N-1) \cdots \mathbf{T}(n+1,\, n), \tag{4.9}$$

and ultimately arrive at a diagonal matrix $\mathbf{\Sigma} \in \mathbb{R}^{N\times N}$. Specifically, after performing all left and right rotations, we obtain

$$\begin{aligned} \Sigma &= \mathbf{S}(1,\, 0)^T \cdots \mathbf{S}(N,\, N-1)^T\,\mathbf{B}^{(0)}\,\mathbf{T}(N,\, N-1) \cdots \mathbf{T}(1,\, 0), \\ \mathbf{U}_2^T &= \mathbf{S}(1,\, 0)^T \cdots \mathbf{S}(N,\, N-1)^T, \quad \mathbf{V}_2 = \mathbf{T}(N,\, N-1) \cdots \mathbf{T}(1,\, 0). \end{aligned} \tag{4.10}$$

Here, $\mathbf{U}_2^T$ and $\mathbf{V}_2$ are orthogonal matrices formed by concatenating the respective Givens rotations in their left and right transformations.

Recalling from the Bidiagonalization phase (cf.(4.6)) that $\mathbf{B}^{(0)} = \mathbf{U}_1^T \mathbf{A}\, \mathbf{V}_1$, we conclude that the net effect of Bidiagonalization and Diagonalization yields the SVD of $\mathbf{A}$:

$$\Sigma \;=\; \mathbf{U}_2\, \mathbf{U}_1\, \mathbf{A}\, \mathbf{V}_2\, \mathbf{V}_1, \quad \mathbf{U} \;=\; \mathbf{U}_2\, \mathbf{U}_1, \quad \mathbf{V} \;=\; \mathbf{V}_1\, \mathbf{V}_2, \quad \text{so that} \quad \mathbf{A} \;=\; \mathbf{U}\, \Sigma\, \mathbf{V}^T.$$

Thus, the combination of Householder-based Bidiagonalization and Givens-based Diagonalization fully characterizes the Golub–Reinsch procedure for obtaining $\mathbf{U}, \Sigma, \mathbf{V}^T$ from an arbitrary matrix $\mathbf{A}$. In TTD, iterating this procedure on reshaped subtensors and performing a subsequent *truncation* step (cf. Section 2) leads to compression by discarding minor singular values.

(c) Truncation. After computing the full SVD of $\mathbf{A} \in \mathbb{R}^{M \times N}$, the diagonal entries of $\mathbf{\Sigma}$ (i.e., the singular values) are sorted in descending order. For consistency, we permute the columns of $\mathbf{U}$ and the rows of $\mathbf{V}^T$ to match this sorted arrangement so that each singular value corresponds to the appropriate pair of left and right singular vectors. Because these singular values quantify the "information content" of $\mathbf{A}$, sorting them before truncation ensures that the most significant components are retained to maximize approximation fidelity.

Let $\delta > 0$ be a truncation threshold. We scan the sorted singular values until we find the first index i such that the i-th singular value falls below δ. All singular values (and their associated vectors) beyond index i are discarded. Hence, the matrices $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}^T$ are truncated according to:

$$\mathbf{U}_{\text{truncated}} \;=\; \mathbf{U}[\,1\!:\!M,\; 1\!:\!i], \quad \mathbf{\Sigma}_{\text{truncated}} \;=\; \Sigma[\,1\!:\!i,\; 1\!:\!i], \quad \mathbf{V}^T_{\text{truncated}} \;=\; \mathbf{V}^T[\,1\!:\!i,\; 1\!:\!N], \tag{4.11}$$

where the notation $\mathbf{A}[\,j\!:\!k,\; m\!:\!n]$ refers to the submatrix of $\mathbf{A}$ that spans rows j through k and columns m through n. Consequently, the original matrix $\mathbf{A}$ can be approximated by

$$\mathbf{A} \;=\; \mathbf{U}_{\text{truncated}}\, \mathbf{\Sigma}_{\text{truncated}}\, \mathbf{V}^T_{\text{truncated}} \;+\; \mathbf{E}, \tag{4.12}$$

where $\mathbf{E}$ denotes the residual error. Repeatedly reshaping the original high-dimensional data into 2D form, applying TSVD, and then reshaping partial results into core tensors yields the multi-stage TTD for the entire tensor.

2) The Role of ϵ: Balancing Compression and Accuracy. In TTD, a user-specified ϵ determines how aggressively small singular values are discarded at each TSVD step. Then each step $k \in \{1, \ldots, d-1\}$ introduces a local error $\mathbf{E}_k$ with $\|\mathbf{E}_k\|_2 \leq \delta$. If the original tensor is d-dimensional, TTD typically performs $d-1$ such TSVD steps, so the global reconstruction error satisfies

$$\|A - \tilde{A}\|_2 \;\leq\; \sqrt{\sum_{k=1}^{d-1} \|\mathbf{E}_k\|_2^2} \;\leq\; \sqrt{(d-1)\,\delta^2} \;=\; \delta\, \sqrt{d-1},$$

where A is the matrix representation of the original tensor and $\tilde{A}$ is its reconstructed approximation. As discussed in Section 2, to limit this cumulative error to be at most ϵ, we may conservatively require

$$\delta\, \sqrt{d-1} \;\leq\; \epsilon \quad \Longrightarrow \quad \delta \;\leq\; \frac{\epsilon}{\sqrt{d-1}}.$$

Many applications employ a relative error measure, scaling by $\|A\|_2$. In that case,

$$\delta \;=\; \frac{\epsilon}{\sqrt{d-1}}\,\|A\|_2, \tag{4.13}$$

which precisely characterizes the trade-off: smaller ϵ yields lower approximation error but also lowers compression ratio, while larger ϵ induces higher compression at the cost of greater reconstruction error. This relationship underpins the phenomenon observed empirically in Figure 3, where increasing ϵ leads to more aggressive singular-value truncation, higher compression ratios, and correspondingly larger accuracy loss.

3) Mathematical Analysis of Reshape Operations in TTD. The TTD procedure relies heavily on repeatedly reshaping high-dimensional tensors into 2D matrices for TSVD, then reshaping the decomposed factors back into lower-dimensional core tensors for subsequent iterations. We formalize the reshape definition, illustrate its role in both compression and decoding, and explore how *IM2COL*-based convolution can benefit from merging consecutive reshape steps.

(a) Reshape Definition. Formally, let $\mathbf{X} \in \mathbb{R}^{I_N \times \cdots \times I_1}$ be an N-dimensional tensor, and let $\mathbf{Y} \in \mathbb{R}^{J_M \times \cdots \times J_1}$ be an M-dimensional tensor. We define

$$\mathbf{Y} \;=\; \text{reshape}(\mathbf{X},\, [\,J_M,\, J_{M-1},\, \ldots,\, J_1]) \quad \text{if and only if} \quad \text{numel}(\mathbf{X}) \;=\; \text{numel}(\mathbf{Y}), \tag{4.14}$$

where numel$(\cdot)$ computes the total number of elements in a tensor. For instance, a 3D tensor $\mathbf{T} \in \mathbb{R}^{I_3 \times I_2 \times I_1}$ satisfies

$$\text{numel}(\mathbf{T}) \;=\; I_1 \times I_2 \times I_3. \tag{4.15}$$

Because reshape merely reinterprets the same contiguous memory with a different shape, it entails no actual data rearrangement. Hence, reshaping is generally a memory-bound operation rather than a compute-bound operation.

(b) TTD Workflow. Consider a d-dimensional original tensor $\mathbf{W}$. TTD compresses $\mathbf{W}$ by performing $d-1$ iterations of TSVD, reshaping intermediate tensors as necessary. Let C_k denote the input tensor at the k-th iteration. We first reshape C_k into a 2D matrix $\mathbf{C}_k$ as:

$$\mathbf{C}_k \;=\; \text{reshape}\Big(C_k,\, \Big[r_{k-1},\, \frac{\text{numel}(C_k)}{r_{k-1}\, n_k}\Big]\Big), \tag{4.16}$$

where n_k is the size of the k-th dimension of $\mathbf{W}$, and r_{k-1} is the TT-rank carried over from the previous iteration. We then apply TSVD to $\mathbf{C}_k$:

$$\text{TSVD}_k\,(\mathbf{C}_k) \;=\; \mathbf{U}^{(k)}_{\text{truncated}}\, \mathbf{\Sigma}^{(k)}_{\text{truncated}}\, \mathbf{V}^{(k)T}_{\text{truncated}} \;+\; \mathbf{E}_k. \tag{4.17}$$

The left factor $\mathbf{U}^{(k)}_{\text{truncated}}$ is then reshaped into the next core tensor:

$$\mathbf{G}_k \;=\; \text{reshape}\,(\mathbf{U}^{(k)}_{\text{truncated}},\, [r_{k-1},\, n_k,\, r_k]), \tag{4.18}$$

$$C_{k+1} = \text{reshape}\Big(\mathbf{\Sigma}^{(k)}_{\text{truncated}}\mathbf{V}^{(k)T}_{\text{truncated}}, [r_k, \frac{\text{numel}(C_{k+1})}{r_k}]\Big), \tag{4.19}$$

where numel$(C_{k+1}) = r_k \times n_{k+1} \times \cdots \times n_d$. Repeating this process for $k = 1$ to $d-1$ yields, after reshaping and truncating each 2D matrix, a chain of core tensors that together approximate $\mathbf{W}$. Conversely, during *decoding*, the same reshapes are applied in reverse to multiply the core tensors (via repeated Einsum operations) and recover an approximation $\tilde{\mathbf{W}}$ of the original tensor.

Let $\mathbf{M}_{\text{temp}}^{(k)}$ denote the intermediate matrix multiplication result at step k. The Einsum stage then proceeds as:

$$\begin{aligned} \mathbf{M}_{\text{temp}}^{(1)} &= \text{reshape}(\mathbf{G}_1, [r_0\, n_1, r_1]) \times \text{reshape}(\mathbf{G}_2, [r_1, n_2\, r_2]), \\ \mathbf{M}_{\text{temp}}^{(2)} &= \text{reshape}(\mathbf{M}_{\text{temp}}^{(1)}, [r_0\, n_1\, n_2, r_2]) \times \text{reshape}(\mathbf{G}_3, [r_2, n_3\, r_3]), \\ &\vdots \\ \mathbf{M}_{\text{temp}}^{(d-1)} &= \text{reshape}(\mathbf{M}_{\text{temp}}^{(d-2)}, [r_0\, n_1\, n_2 \ldots n_{d-1}, r_{d-1}]) \times \text{reshape}(\mathbf{G}_d, [r_{d-1}, n_d\, r_d]). \end{aligned} \tag{4.20}$$

Finally, we perform a Reconstruction reshape

$$\tilde{\mathbf{W}} = \text{reshape}(\mathbf{M}_{\text{temp}}^{(d-1)}, [r_0\, n_1, n_2, \ldots, n_d]), \tag{4.21}$$

obtaining the approximation $\tilde{\mathbf{W}}$ of the original d-dimensional tensor $\mathbf{W}$.

(c) *IM2COL* Process and Reshape Merging. We next turn to the *IM2COL* transformation. Suppose $\mathbf{X} \in \mathbb{R}^{D\times C\times H\times W}$ is the input tensor, where D is the batch size, C the input channels, and H, W the height and width of the feature maps. Let $\mathbf{W}_{\text{conv}} \in \mathbb{R}^{out\_c\times in\_c\times HF\times WF}$ be the convolution kernels. Here out_c denotes the output channels, in_c the input channels (matching C), and HF, WF the height and width of the convolution filters. Define the output feature map size as (OH, OW), given by

$$OH = \frac{H + 2p - HF}{s}, \quad OW = \frac{W + 2p - WF}{s},$$

where p and s are the padding and stride, respectively. The *IM2COL* method unfolds both $\mathbf{X}$ and $\mathbf{W}_{\text{conv}}$ into 2D matrices, then performs a matrix multiplication.

First, we slice $\mathbf{X}$ by the filter window and stack each patch as a column, forming a 6D tensor $\mathbf{X}_{\text{patch}} \in \mathbb{R}^{D\times C\times OH\times OW\times HF\times WF}$. Each slice index i, j, u, v runs from 1 to OH, OW, HF, WF, respectively:

$$\mathbf{X}_{\text{patch}}[1\!:\!D,\ 1\!:\!C,\ i,\ j,\ u,\ v] = \mathbf{X}[1\!:\!D,\ 1\!:\!C,\ (i-1)\,s+u,\ (j-1)\,s+v].$$

We then flatten $\mathbf{X}_{\text{patch}}$ into $\mathbf{X}_{\text{col}} \in \mathbb{R}^{D\times XH\times XW}$ by

$$XH = C \times HF \times WF, \quad XW = OH \times OW, \quad \mathbf{X}_{\text{col}} = \text{reshape}(\mathbf{X}_{\text{patch}}, [D, XH, XW]). \tag{4.22}$$

Likewise, we unfold $\mathbf{W}_{\text{conv}}$ into a 2D matrix $\mathbf{W}_{\text{flatten}} \in \mathbb{R}^{out\_c\times WW}$, where

$$WW = in\_c \times HF \times WF, \quad \mathbf{W}_{\text{flatten}} = \text{reshape}(\mathbf{W}_{\text{conv}}, [out\_c, WW]). \tag{4.23}$$

Convolution then becomes a matrix multiplication:

$$\mathbf{Y}[i,\ 1\!:\!out\_c,\ 1\!:\!OH\times OW] = \mathbf{W}_{\text{flatten}} \times \mathbf{X}_{\text{col}}[i,\ 1\!:\!XH,\ 1\!:\!XW], \quad \mathbf{Y} \in \mathbb{R}^{D\times out\_c\times(OH\times OW)}, \tag{4.24}$$

where i indexes the batch dimension $1 \le i \le D$. A final reshape of $\mathbf{Y}$ matches the conventional convolution-output format,

$$\mathbf{Y} = \text{reshape}(\mathbf{Y}, [D, out\_c, OH, OW]).$$

(d) Mergeability of Consecutive Reshapes. Figure 8 outlines how decoding (**D1**, **D2**) and inference (**I1**, **I2**) each employs reshaping. In a naive approach, once we finish decoding at step **D2**, the weight tensor is reshaped into a 4D array $\tilde{\mathbf{W}}$. Then, **I1** (*IM2COL* initialization) again reshapes $\tilde{\mathbf{W}}$ into a 2D matrix. Formally,

$$\tilde{\mathbf{W}} = \text{reshape}(\mathbf{M}_{\text{temp}}^{(3)}, [r_0\, n_1,\, n_2,\, n_3,\, n_4]) = \mathbf{W}_{\text{conv}}, \quad \mathbf{W}_{\text{flatten}} = \text{reshape}\,(\mathbf{W}_{\text{conv}}, [out\_c,\, WW]).$$

However, since reshaping does not rearrange the underlying memory, it is possible to merge these consecutive 2D→4D and 4D→2D reshapes into one step. By mapping the TTD-decoding variables $\{r_0, n_1, \dots\}$ to the convolution parameters $\{out\_c, WW\}$, we can directly reshape:

$$\mathbf{W}_{\text{flatten}} = \text{reshape}(\mathbf{W}_{\text{temp}}, [out\_c, WW]), \tag{4.25}$$

thus bypassing the intermediate 4D form. This merged reshape is mathematically valid whenever numel($\tilde{\mathbf{W}}$) and numel($\mathbf{W}_{\text{flatten}}$) match that of $\mathbf{W}_{\text{temp}}$. Consequently, the extra memory-bound operations are eliminated, enabling higher efficiency on resource-constrained edge devices without altering any data or requiring specialized hardware modifications.

(e) Conclusion and Implications. Summarizing the above results, we establish that:

- Reshape in TTD: High-dimensional tensors are systematically reshaped into 2D matrices for TSVD, then reshaped back to produce either core tensors or subsequent input tensors. This same pattern appears in decoding via repeated Einsum operations.
- *IM2COL* Mergeability: Consecutive reshapes in TTD decoding and *IM2COL*-based convolution can be merged into a single index transformation, reducing memory-bound overhead by removing redundant 2D→4D→2D conversions.

This analysis rigorously establishes that the proposed reshape optimization is mathematically justified and not merely an implementation detail. In the next section, we experimentally demonstrate how it yields significant performance benefits on an FPGA-based edge AI system, aligning well with the goals of efficient and flexible model deployment in resource-constrained environments.

5. Experimental results

5.1. Experimental setup and FPGA prototype

We established an experimental setup to quantitatively analyze the acceleration effects of the Einsum operations in TTD decoding using a GEMM IP, as well as the optimization effects achieved by eliminating redundant reshape operations during inference. The core component of our experimental environment, the GEMM IP, was designed based on the Versatile Tensor Accelerator (VTA) [57], featuring a 320kB on-chip scratchpad memory (SPM) and supporting up to 16×16 matrix

multiplications. For matrices exceeding this dimension, the core divides them into multiple 16×16 blockwise matrices, sequentially transfers them into the GEMM IP's SPM, and aggregates the partial multiplication results to complete the overall matrix operation.

We developed an edge AI inference processor incorporating this GEMM IP. To achieve this, we designed a full-custom Register Transfer Level (RTL) implementation utilizing the RISC-V eXpress EDA tool [58], which has been widely adopted for various RISC-V processor designs—such as for anomaly detection [59, 60], security-oriented architectures [61], lightweight floating-point support [62], and processing-in-memory (PIM) integrated edge processors [63,64]. Figure 10 illustrates the architecture of our proposed edge AI processor. The core consists of a RISC-V Rocket Core [65], coupled with 64kB of on-chip SRAM and interconnected through a lightweight network-on-chip (μNoC) [66].

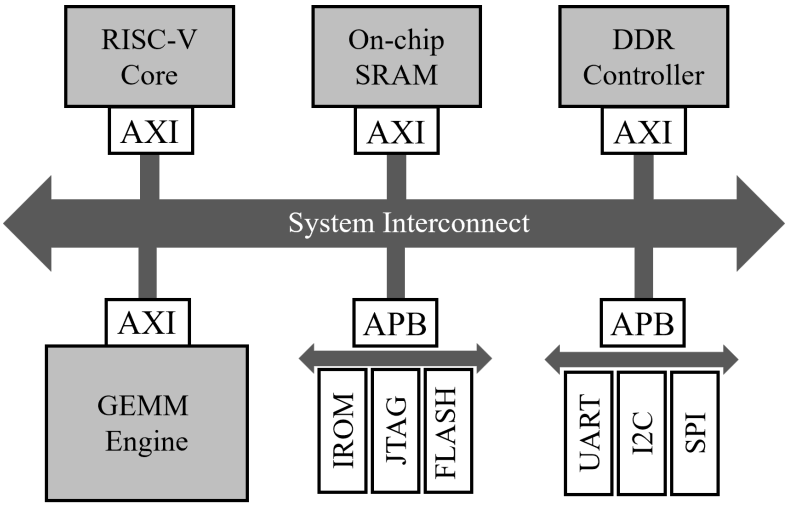


Figure 10. Architecture of the developed full-custom edge AI inference processor based on a RISC-V Rocket core integrated with a GEMM accelerator.

Subsequently, we validated the functionality of our proposed inference processor through FPGA prototyping, performing practical inference tasks with real image classification applications. The FPGA board used for prototyping is the Genesys2 Kintex-7 [67], operating at a clock frequency of 100MHz. Table 2 summarizes the resource utilization of the developed FPGA prototype. The total utilization comprises 122,113 Look-Up Tables (LUTs) and 78,482 Flip-Flops (FFs), of which the Rocket RISC-V core accounts for approximately 12.3% (15,041 LUTs) and 12.6% (9,890 FFs), respectively. Furthermore, the GEMM accelerator designed to expedite inference and decoding processes consumes 68.9% (84,150 LUTs) and 42.0% (32,939 FFs) of the total resources.

Leveraging the developed processor prototype, we established a cloud-training and edge-inference model deployment environment. Figure 11 depicts an exemplary demonstration of a CIFAR-10 image classification application running on our FPGA prototype.

Table 2. FPGA resource utilization breakdown of the prototyped edge AI inference processor.

IPs	LUTs	FFs
Rocket RISC-V Core	15,041	9,890
SRAM	166	323
DDR Controller	7,961	7,581
Peripherals	5,047	10,373
System Interconnect	9,748	17,376
GEMM Accelerator	84,150	32,939

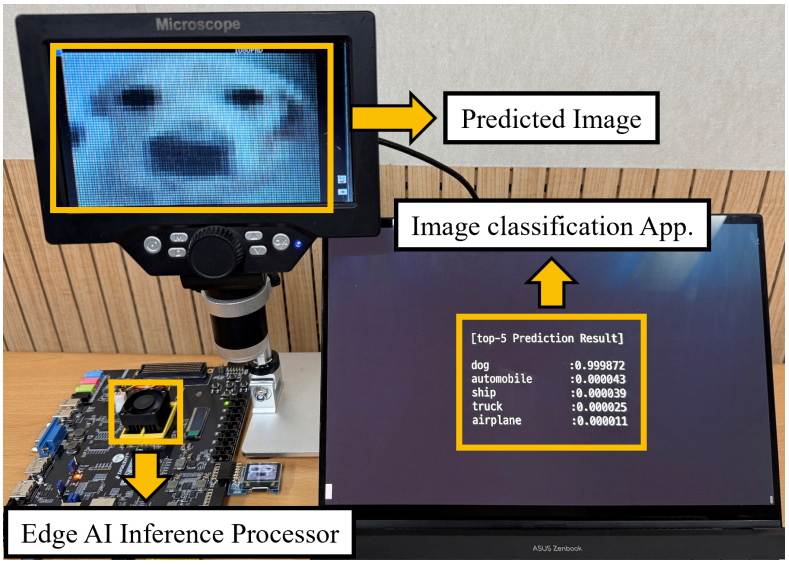


Figure 11. Demonstration of CIFAR-10 image classification running on the FPGA-prototyped edge AI inference processor.

For our experiments, we utilized a ResNet20-based lightweight image classification model optimized for smooth real-time inference on the prototype edge processor. To independently assess the performance gains from the GEMM IP-based acceleration of TTD decoding and reshape optimizations, we integrated tracking capabilities into the application that enable monitoring of TTD decoding operations and GEMM IP activation. The application was executed on our prototyped edge processor to collect and analyze experimental results.

5.2. Performance evaluation of optimized TTD decoding

We first measured and compared the reshape operation times for two scenarios: the baseline, which separately executes both reshaping steps (**D2** and **I1** in Figure 9), and the optimized approach, which performs a single, merged reshape to directly produce $\mathbf{W}_{\text{flatten}}$. Table 3 summarizes the measured reshape times across various tensor shapes. Intuitively, merging two reshape operations into one might yield a twofold improvement. However, our experimental results demonstrate approximately

a threefold improvement across different tensor shapes. This performance gain arises because the optimized method directly reshapes from a two-dimensional tensor to another two-dimensional tensor ($\mathbf{W}_{\text{flatten}}$), whereas the baseline first reshapes into a higher-dimensional intermediate tensor ($\tilde{\mathbf{W}}$) and subsequently reshapes again into a two-dimensional tensor, incurring additional dimensional transformation overheads. Furthermore, while the absolute reshape times naturally increase with larger tensor shapes, the optimized method consistently maintains an approximately threefold performance improvement.

Table 3. Performance comparison of reshape operations with and without proposed optimization for different tensor sizes.

Tensor shape	Baseline (ms)	Reshape opt. (ms)	Performance improvement
$16 \times 16 \times 1 \times 1$	2.248	0.748	3.005 ×
$32 \times 32 \times 1 \times 1$	8.739	2.911	3.002 ×
$16 \times 16 \times 3 \times 3$	14.268	4.775	2.988 ×
$32 \times 32 \times 3 \times 3$	56.966	19.065	2.988 ×
$64 \times 64 \times 3 \times 3$	227.627	76.206	2.987 ×

Subsequently, we evaluated the overall TTD decoding performance improvement by combining the reshape optimization (skipping step **D2** from Figure 8) with the acceleration provided by the GEMM accelerator for matrix multiplication operations. The baseline scenario, in this case, was defined as performing the TTD decoding process entirely on the core processor without employing reshape optimizations. The compression ratio was uniformly set to 2× for consistency across all tensor shapes. Table 4 presents the decoding times for three distinct scenarios: (1) Baseline without acceleration, (2) GEMM accelerator only, and (3) Proposed solution combining GEMM accelerator and reshape optimization.

Table 4. Decoding latency comparison for TTD with GEMM acceleration and reshape optimization across different tensor shapes. GEMM block size is 16×16.

Tensor shape	Baseline (ms)	GEMM Acc. only (ms)	GEMM Acc. + Reshape Opt. (ms)
$16 \times 16 \times 1 \times 1$	1.886	1.458	0.710
$32 \times 32 \times 1 \times 1$	8.044	5.764	2.853
$16 \times 16 \times 3 \times 3$	18.401	13.626	6.043
$32 \times 32 \times 3 \times 3$	92.886	49.678	30.613
$64 \times 64 \times 3 \times 3$	510.827	232.907	156.701

Comparing the baseline and the GEMM-accelerated scenario without reshape optimization, we observed a 22.7% decoding time reduction (1.886 ms to 1.458 ms) for the smallest tensor shape ($16 \times 16 \times 1 \times 1$). The performance improvement increased significantly with tensor size, achieving reductions of 28.3%, 41.2%, 46.5%, and 54.4% for tensor shapes of $32 \times 32 \times 1 \times 1$, $16 \times 16 \times 3 \times 3$, $32 \times 32 \times 3 \times 3$, and $64 \times 64 \times 3 \times 3$, respectively. This indicates the greater efficacy of GEMM acceleration for larger tensor computations.

When combining GEMM acceleration with reshape optimization, the decoding times decreased even further, achieving performance improvements of 62.3%, 64.5%, 67.1%, 67.0%, and 69.3% for the respective tensor shapes listed above. These results show our approach effectively accelerates both matrix multiplication and reshaping, substantially reducing overhead in deploying TTD models on resource-limited edge devices. Finally, to demonstrate the generalizability of the proposed approach across diverse hardware constraints, we conducted additional experiments using a GEMM IP with smaller (8×8) block sizes instead of the original (16×16). As shown in Table 5, reducing the block size naturally decreases parallelism, resulting in increased overall decoding times. Nevertheless, the proposed method consistently achieved up to 61% performance improvement over both baseline and GEMM-only scenarios. This confirms that our optimization technique can be effectively applied across various edge platforms with differing hardware constraints.

Table 5. Decoding latency comparison for TTD with GEMM acceleration and reshape optimization across different tensor shapes. GEMM block size is 8×8.

Tensor shape	Baseline (ms)	GEMM Acc. only (ms)	GEMM Acc. + Reshape Opt. (ms)
$16 \times 16 \times 1 \times 1$	1.886	1.523	0.775
$32 \times 32 \times 1 \times 1$	8.044	6.114	3.203
$16 \times 16 \times 3 \times 3$	18.401	14.788	7.205
$32 \times 32 \times 3 \times 3$	92.886	56.303	37.238
$64 \times 64 \times 3 \times 3$	510.827	275.521	199.315

6. Conclusions

In this paper, we have experimentally and analytically demonstrated that TTD achieves superior accuracy retention and compression efficiency compared to traditional pruning methods, especially when fine-tuning is impractical in cloud-to-edge deployments. To address the critical challenge posed by the computational overhead of decoding TTD-compressed parameters at resource-constrained edge devices, we have introduced a mathematically rigorous yet practically viable optimization framework. Our approach exploits existing GEMM hardware accelerators commonly integrated into edge processors to dramatically accelerate the performance-critical Einsum operations inherent in TTD decoding, without necessitating additional specialized hardware. Furthermore, through careful analytical investigation, we identified redundant reshape operations that span the decoding and inference phases and developed a novel reshape-merging strategy, significantly reducing memory-bound overhead. Extensive experimental evaluations on our FPGA-based RISC-V edge inference processor validate these innovations, demonstrating approximately a threefold acceleration in reshape operations and an overall decoding performance improvement of up to 69.3%. While our experiments primarily demonstrate the proposed optimization in image classification scenarios, the underlying computational pattern—decoding TTD-compressed convolution layers into GEMM operations—is commonly found across various deep learning applications, including speech recognition and object detection. Thus, our optimization techniques can be broadly applicable to other edge AI scenarios involving TTD-based compression and GEMM accelerators. Exploring and validating these extensions represent promising directions for future research. Beyond these immediate benefits, our work

demonstrates the synergy between advanced tensor mathematics, algorithmic optimizations, and practical hardware constraints, paving the way for efficient and scalable cloud-to-edge AI deployments. We anticipate that this interdisciplinary approach will inspire further research at the convergence of mathematical optimization, hardware-aware algorithm design, and edge computing, ultimately enabling broader and more efficient real-world adoption of advanced AI technologies.

Author contributions

H. K., S. J., K. L., and W. L. were the main researchers who initiated and organized research reported in the paper. All the authors were responsible for analyzing the simulation results and writing the paper, and all authors have read and agreed to the published version of the manuscript.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00345668), in part by Korea Institute for Advancement of Technology (KIAT) grant funded by the Korea Government (MOTIE) (P0017011, HRD Program for Industrial Innovation), and in part by the Chung-Ang University Graduate Research Scholarship Grants in 2024.

Conflict of interest

The authors declare no conflicts of interest.

References

1. S. khan, M. Naeem, M. Qiyas, Deep intelligent predictive model for the identification of diabetes, *AIMS Math.*, **8** (2023), 16446–16462. https://doi.org/10.3934/math.2023840
2. K. Tarmissi, H. A. Mengash, N. Negm, Y. Said, A. M. Al-Sharafi, Explainable artificial intelligence with fusion-based transfer learning on adverse weather conditions detection using complex data for autonomous vehicles, *AIMS Math.*, **9** (2024), 35678–35701. https://doi:10.3934/math.20241693
3. J. Chen, H. Yan, Z. Liu, M. Zhang, H. Xiong, S. Yu, When federated learning meets privacy-preserving computation, *ACM Comput. Surv.*, **56** (2024), 1–36. https://10.1145/3679013
4. H. Zhang, S. Huang, M. Xu, D. Guo, X. Wang, X. Wang, et al., Large-scale measurements and optimizations on latency in edge clouds, *IEEE T. Cloud Comput.*, **12** (2024), 1218–1231. https://doi.org/10.1109/TCC.2024.3452094
5. A. Zhou, J. Yang, T. Qiao, Y. Qi, Z. Yang, W. Zhao, et al., Graph neural networks automated design and deployment on device-edge co-inference systems, *ACM/IEEE Design Autom. Conf.*, **61** (2024), 1–6. https://doi.org/10.1145/3649329.3655938

6. X. Wang, M. Shen, K. Yang, On-edge high-throughput collaborative inference for real-time video analytics, *IEEE Internet Things*, **11** (2024), 33097–33109. https://doi.org/10.1109/JIOT.2024.3424235
7. S. Zhu, T. Voigt, F. Rahimian, J. Ko, On-device training: A first overview on existing systems, *ACM T. Sensor Network*, **20** (2024), 1–39. https://doi.org/10.1145/3696003
8. A. Quélennec, E. Tartaglione, P. Mozharovskyi, V. T. Nguyen, Towards on-device learning on the edge: Ways to select neurons to update under a budget constraint, *IEEE/CVF Winter Conf. Appl. Comput. Vis.*, 2024, 685–694. https://doi.org/10.1109/WACVW60836.2024.00080
9. J. Lin, L. Zhu, W. M. Chen, W. C. Wang, C. Gan, S. Han, On-device training under 256KB memory, *Adv. Neural Inf. Process. Syst.*, **35** (2022), 22941–22954.
10. S. Sai, M. Prasad, G. Dashore, V. Chamola, B. Sikdar, On-device generative AI: The need, architectures, and challenges, *IEEE Consum. Electr. M.*, (In press). https://doi.org/10.1109/MCE.2024.3518761
11. Z. Dong, Q. He, F. Chen, H. Jin, T. Gu, Y. Yang, Edgemove: Pipelining device-edge model training for mobile intelligence, *ACM Web Conf.*, 2023, 3142–3153. https://doi.org/10.1145/3543507.3583540
12. H. Wen, Y. Li, Z. Zhang, S. Jiang, X. Ye, Y. Ouyang, et al., Adaptivenet: Post-deployment neural architecture adaptation for diverse edge environments, *ACM Mobicom.*, 2023, 1–17. https://doi.org/10.1145/3570361.3592529
13. Y. Shi, K. Yang, T. Jiang, J. Zhang, K. B. Letaief, Communication-efficient edge AI: Algorithms and systems, *IEEE Commun. Surv. Tutor.*, **22** (2020), 2167–2191. https://doi.org/10.1109/COMST.2020.3007787
14. X. Li, S. Bi, Optimal AI model splitting and resource allocation for device-edge co-inference in multi-user wireless sensing systems, *IEEE Trans. Wireless Commun.*, (In press). https://doi.org/10.1109/TWC.2024.3378418
15. Y. Guo, A. Yao, Y. Chen, Dynamic network surgery for efficient DNNs, *Adv. Neural Inf. Process. Syst.*, **29** (2016).
16. X. Dong, S. Chen, S. Pan, Learning to prune deep neural networks via layer-wise optimal brain surgeon, *Adv. Neural Inf. Process. Syst.*, **30** (2017).
17. X. Ding, G. Ding, Y. Guo, J. Han, Centripetal SGD for pruning very deep convolutional networks with complicated structure, *IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2019, 4943–4953. https://doi.org/10.1109/CVPR.2019.00508
18. O. A. Malik, S. Becker, Low-rank Tucker decomposition of large tensors using TensorSketch, *Adv. Neural Inf. Process. Syst.*, **31** (2018).
19. Y. Gao, G. Zhang, C. Zhang, J. Wang, L. T. Yang, Y. Zhao, Federated tensor decomposition-based feature extraction approach for industrial IoT, *IEEE Trans. Ind. Inform.*, **17** (2021), 8541–8549. https://doi.org/10.1109/TII.2021.3074152
20. K. Han, W. Xiang, Multiscale tensor decomposition and rendering equation encoding for view synthesis, *IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2023, 4232–4241. https://doi.org/10.1109/CVPR52729.2023.00412

21. M. Nagel, M. Fournarakis, Y. Bondarenko, T. Blankevoort, Overcoming oscillations in quantization-aware training, *Int. Conf. Mach. Learn.*, 2022, 16318–16330.

22. Z. Liu, Y. Wang, K. Han, W. Zhang, S. Ma, W. Gao, Post-training quantization for vision transformer, *Adv. Neural Inf. Process. Syst.*, **34** (2021), 28092–28103.

23. A. Marchisio, D. Dura, M. Capra, M. Martina, G. Masera, M. Shafique, Swifttron: An efficient hardware accelerator for quantized transformers, *Int. Jt. Conf. Neural Netw.*, 2023, 1–9. https://doi.org/10.1109/IJCNN54540.2023.10191521

24. L. Beyer, X. Zhai, A. Royer, L. Markeeva, R. Anil, A. Kolesnikov, Knowledge distillation: A good teacher is patient and consistent, *IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2022, 10925–10934. https://doi.org/10.1109/CVPR52688.2022.01065

25. C. Blakeney, X. Li, Y. Yan, Z. Zong, Parallel blockwise knowledge distillation for deep neural network compression, *IEEE T. Parallel Distr.*, **32** (2020), 1765–1776. https://doi.org/10.1109/TPDS.2020.3047003

26. O. T. C. Chen, Y. X. Chang, C. Y. Chung, Y. Cheng, M. H. Ha, Hardware-aware iterative one-shot neural architecture search with adaptable knowledge distillation for efficient edge computing, *IEEE Access*, (In press). https://doi.org/10.1109/ACCESS.2025.3554185

27. T. Huang, S. You, F. Wang, C. Qian, C. Xu, Knowledge distillation from a stronger teacher, *Adv. Neural Inf. Process. Syst.*, **35** (2022), 33716–33727.

28. X. Sun, E. Zhu, G. Qu, F. Zhang, QuaDCNN: Quantized compression of deep CNN based on tensor-train decomposition with automatic rank determination, *Neurocomputing*, **638** (2025), 130047. https://doi.org/10.1016/j.neucom.2025.130047

29. N. Aghli, E. Ribeiro, Combining weight pruning and knowledge distillation for CNN compression, *IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2021, 3191–3198.

30. S. Muralidharan, S. T. Sreenivas, R. Joshi, M. Chochowski, M. Patwary, M. Shoeybi, et al., Compact language models via pruning and knowledge distillation, *Adv. Neural Inf. Process. Syst.*, **37**, 41076–41102, 2024.

31. I. V. Oseledets, Tensor-train decomposition, *SIAM J. Sci. Comput.*, **33** (5), 2295–2317, 2011. https://doi.org/10.1137/090752286

32. C. Yin, B. Acun, C. J. Wu, X. Liu, TT-Rec: Tensor train compression for deep learning recommendation models, *Proc. Mach. Learn. Syst.*, **3** (2021), 448–462.

33. C. Yin, D. Zheng, I. Nisa, C. Faloutsos, G. Karypis, R. Vuduc, Nimble GNN embedding with tensor-train decomposition, *ACM SIGKDD Conf. Knowl. Discov. Data Min.*, 2022, 2327–2335. https://doi.org/10.1145/3534678.3539423

34. D. Lee, R. Yin, Y. Kim, A. Moitra, Y. Li, P. Panda, TT-SNN: Tensor train decomposition for efficient spiking neural network training, *Des. Autom. Test Eur. Conf. Exhib.*, 2024, 1–6. https://doi.org/10.23919/DATE58400.2024.10546679

35. Y. He, L. Xiao, Structured pruning for deep convolutional neural networks: A survey, *IEEE T. Pattern Anal.*, **46** (5), 2900–2919, 2023. https://doi.org/10.1109/TPAMI.2023.3334614

36. Z. Wang, Sparsert: Accelerating unstructured sparsity on GPUs for deep learning inference, *ACM Int. Conf. Parallel Archit. Compil. Techn.*, 2020, 31–42. https://doi.org/10.1145/3410463.3414654

37. X. Ma, S. Lin, S. Ye, Z. He, L. Zhang, G. Yuan, et al., Non-structured DNN weight pruning—Is it beneficial in any platform? *IEEE Trans. Neural Netw. Learn. Syst.*, **33** (2021), 4930–4944. https://doi.org/10.1109/TNNLS.2021.3063265

38. J. Lee, W. Lee, J. Sim, Tender: Accelerating large language models via tensor decomposition and runtime requantization, *ACM/IEEE Int. Symp. Comput. Archit.*, 2024, 1048–1062. https://doi.org/10.1109/ISCA59077.2024.00080

39. J. Gou, B. Yu, S. J. Maybank, D. Tao, Knowledge distillation: A survey, *Int. J. Comput. Vis.*, **129** (2021), 1789–1819. https://doi.org/10.1007/s11263-021-01453-z

40. D. Hong, T. G. Kolda, J. A. Duersch, Generalized canonical polyadic tensor decomposition, *SIAM Rev.*, **62** (2020), 133–163. https://doi.org/10.1137/18M1203626

41. X. Liu, K. K. Parhi, Tensor decomposition for model reduction in neural networks: A review, *IEEE Circ. Syst. Mag.*, **23** (2023), 8–28. https://doi.org/10.1109/MCAS.2023.3267921

42. J. G. Jang, U. Kang, Static and streaming Tucker decomposition for dense tensors, *ACM T. Knowl. Discov. D.*, **17** (2023), 1–34. https://doi.org/10.1145/3568682

43. X. Gao, L. Lu, Q. Liu, Non-negative Tucker decomposition with double constraints for multiway dimensionality reduction, *AIMS Math.*, **9** (2024), 21755–21785. https://doi.org/10.3934/math.20241058

44. M. Astrid, S. I. Lee, CP-decomposition with tensor power method for convolutional neural networks compression, *IEEE Int. Conf. Big Data Smart Comput.*, 2017, 115–118. https://doi.org/10.1109/BIGCOMP.2017.7881725

45. S. Fang, R. M. Kirby, S. Zhe, Bayesian streaming sparse Tucker decomposition, *Uncertainty Artif. Intell.*, 2021, 558–567.

46. Z. C. Wu, T. Z. Huang, L. J. Deng, H. X. Dou, D. Meng, Tensor wheel decomposition and its tensor completion application, *Adv. Neural Inf. Process. Syst.*, **35** (2022), 27008–27020.

47. C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, B. Yuan, TIE: Energy-efficient tensor train-based inference engine for deep neural network, *Int. Symp. Comput. Archit.*, 2019, 264–278. https://doi.org/10.1145/3307650.3322258

48. L. Li, W. Yu, K. Batselier, Faster tensor train decomposition for sparse data, *J. Comput. Appl. Math.*, **405** (2022), 113972. https://doi.org/10.1016/j.cam.2021.113972

49. P. V. Dantas, W. S. da Silva Jr, L. C. Cordeiro, C. B. Carvalho, A comprehensive review of model compression techniques in machine learning, *Appl. Intell.*, **54** (2024), 11804–11844, 2024. https://doi.org/10.1007/s10489-024-05747-w

50. Y. Wang, C. Yang, S. Lan, L. Zhu, Y. Zhang, End-edge-cloud collaborative computing for deep learning: A comprehensive survey, *IEEE Commun. Surv. Tutor.*, (In press). https://doi.org/10.1109/COMST.2024.3393230

51. B. J. Eccles, P. Rodgers, P. Kilpatrick, I. Spence, B. Varghese, DNNshifter: An efficient DNN pruning system for edge computing, *Future Gener. Comput. Syst.*, **152** (2024), 43–54. https://doi.org/10.1016/j.future.2023.09.025

52. A. Koubaa, A. Ammar, A. Kanhouch, Y. AlHabashi, Cloud versus edge deployment strategies of real-time face recognition inference, *IEEE Trans. Netw. Sci. Eng.*, **9** (2021), 143–160. https://doi.org/10.1109/TNSE.2021.3055835

53. S. Manzoor, E. J. Kim, S. H. Joo, S. H. Bae, G. G. In, K. J. Joo, et al., Edge deployment framework of GuardBot for optimized face mask recognition with real-time inference using deep learning, *IEEE Access*, **10** (2022), 77898–77921. https://doi.org/10.1109/ACCESS.2022.3190538

54. S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, et al, cuDNN: Efficient primitives for deep learning, *arXiv Preprint*, arXiv:1410.0759, 2014. https://doi.org/10.48550/arXiv.1410.0759

55. G. Golub, W. Kahan, Calculating the singular values and pseudo-inverse of a matrix, *J. Soc. Ind. Appl. Math. Ser. B Numer. Anal.*, **2** (1965), 205–224. https://doi.org/10.1137/0702016

56. G. H. Golub, C. F. Van Loan, *Matrix Computations*, Johns Hopkins Univ. Press, 4th ed., 2013.

57. T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, et al., A hardware–software blueprint for flexible deep learning specialization, *IEEE Micro*, **39** (2019), 8–16. https://doi.org/10.1109/MM.2019.2928962

58. K. Han, S. Lee, K. I. Oh, Y. Bae, H. Jang, J. J. Lee, et al., Developing TEI-aware ultralow-power SoC platforms for IoT end nodes, *IEEE Internet Things*, **8** (6) (2020), 4642–4656. https://doi.org/10.1109/JIOT.2020.3027479

59. J. Park, E. Choi, K. Lee, J. J. Lee, K. Han, W. Lee, Developing an ultra-low power RISC-V processor for anomaly detection, *Des. Autom. Test Eur. Conf. Exhib.*, 2023, 1–2. https://doi.org/10.23919/DATE56975.2023.10137003

60. E. Choi, J. Park, K. Lee, J. J. Lee, K. Han, W. Lee, Day–Night architecture: Development of an ultra-low power RISC-V processor for wearable anomaly detection, *J. Syst. Archit.*, **152** (2024), 103161. https://doi.org/10.1016/j.sysarc.2024.103161

61. E. Choi, J. Park, K. Han, W. Lee, AESware: Developing AES-enabled low-power multicore processors leveraging open RISC-V cores with a shared lightweight AES accelerator, *Eng. Sci. Technol. Int. J.*, **60** (2024), 101894. https://doi.org/10.1016/j.jestch.2024.101894

62. J. Park, K. Han, E. Choi, J. J. Lee, K. Lee, W. Lee, et al., Designing low-power RISC-V multicore processors with a shared lightweight floating point unit for IoT endnodes, *IEEE Trans. Circuits Syst. I Regul. Pap.*, **71** (2024), 4106–4119. https://doi.org/10.1109/TCSI.2024.3427681

63. K. Lee, S. Jeon, K. Lee, W. Lee, M. Pedram, Radar-PIM: Developing IoT processors utilizing Processing-in-Memory architecture for ultrawideband-radar-based respiration detection, *IEEE Internet Things J.*, **12** (2025), 515–530. https://doi.org/10.1109/JIOT.2024.3466228

64. S. Jeon, K. Lee, K. Lee, W. Lee, HH-PIM: Dynamic optimization of power and performance with heterogeneous-hybrid PIM for edge AI devices, *arXiv preprint arXiv:2504.01468*, 2025. https://arxiv.org/abs/2504.01468

65. SiFIVE, Available from: `https://github.com/chipsalliance/rocket-chip`. Accessed Thursday 10th July, 2025.

66. K. Han, S. Lee, J. J. Lee, W. Lee, M. Pedram, TIP: A temperature effect inversion-aware ultra-low power system-on-chip platform, In *Proc. IEEE/ACM Int. Symp. Low Power Electron. Des.*, 1–6, 2019. https://doi.org/10.1109/ISLPED.2019.8824925

67. Genesys2 FPGA Board, Available from: `https://digilent.com/shop/genesys-2-kintex-7-fpga-development-board`. Accessed Thursday 10th July, 2025.

AIMS Press