

TT-Edge: A Hardware–Software Co-Design for Energy-Efficient Tensor-Train Decomposition on Edge AI

Hyunseok Kwak^{†‡}, Kyeongwon Lee^{†‡}, Kyeongpil Min[†], Chaebin Jung[†], and Woojoo Lee^{†*}
[†]Department of Intelligent Semiconductor Engineering, Chung-Ang University, Seoul, Korea

Abstract—The growing demands of distributed learning on resource-constrained edge devices underscore the importance of efficient on-device model compression. Tensor-Train Decomposition (TTD) offers high compression ratios with minimal accuracy loss, yet repeated singular value decompositions (SVDs) and matrix multiplications can impose significant latency and energy costs on low-power processors. In this work, we present *TT-Edge*, a hardware–software co-designed framework aimed at overcoming these challenges. By splitting SVD into two phases—bidiagonalization and diagonalization, TT-Edge offloads the most compute-intensive tasks to a specialized *TTD-Engine*. This engine integrates tightly with an existing GEMM accelerator, thereby curtailing the frequent matrix–vector transfers that often undermine system performance and energy efficiency. Implemented on a RISC-V-based edge AI processor, TT-Edge achieves a $1.7\times$ speedup compared to a GEMM-only baseline when compressing a ResNet-32 model via TTD, all while reducing overall energy usage by 40.2%. Notably, these gains come with only a 4% increase in total power and minimal hardware overhead—enabled by a lightweight design that reuses GEMM resources and employs a shared floating-point unit. Our experimental results on both FPGA prototypes and post-synthesis power analysis at 45 nm demonstrate that TT-Edge effectively addresses the latency/energy bottlenecks of TTD-based compression in edge environments.

I. INTRODUCTION

The rapid growth of AI applications on edge devices has generated a strong demand for both privacy-preserving and domain-specific model development. A key paradigm meeting these requirements is distributed learning, where models are trained locally on edge devices, and only the resulting parameters—rather than raw data—are periodically shared and aggregated to update a global model [1]–[5]. Approaches like Decentralized Learning and Federated Learning exemplify this trend, offering clear advantages in safeguarding data ownership while allowing models to adapt to specific domains.

However, as distributed learning gains traction in real-world scenarios, the frequent exchange of model parameters between edge–cloud and edge–edge nodes has significantly increased communication overhead [6]–[8]. This bottleneck has prompted extensive research into on-device model compression techniques aimed at reducing the volume of parameters to be transmitted [9]–[11]. Among these strategies, tensor decomposition (TD) has received special attention for its ability to reshape high-dimensional tensors into more compact,

This work was supported in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (No. RS-2023-00277060), and in part by the National Research Foundation of Korea (NRF) grant funded by MSIT (No. RS-2024-00345668).

[‡] Hyunseok Kwak and Kyeongwon Lee contributed equally to this work.

*Woojoo Lee is the corresponding author.

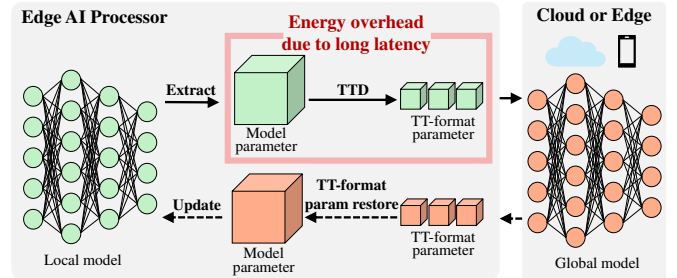


Fig. 1: Compressed model parameter transmission via TTD.

lower-dimensional representations [12], [13]. Unlike methods that yield sparse parameter matrices, TD-based compression preserves small but dense matrix multiplications, making it more computationally efficient across diverse hardware.

Notable TD approaches include Tucker Decomposition [14], Tensor-Ring Decomposition (TRD) [15], and Tensor-Train Decomposition (TTD) [16]. TTD has garnered particular interest for achieving high compression ratios with minimal accuracy loss [17]–[19] and for its compatibility with hardware accelerators [20]–[22]. In our study, we evaluated several TD-based methods on a lightweight ResNet model (ResNet-32 [23]) trained on the CIFAR-10 dataset. As summarized in Table I, TTD attained a $3.4\times$ compression ratio relative to the uncompressed model, while preserving 92.09% accuracy. These promising results motivated our focus on TTD to mitigate communication overhead in distributed learning.

Despite its demonstrated effectiveness, deploying TTD-based compression on resource-constrained edge AI processors poses nontrivial challenges. Fig. 1 illustrates a typical distributed learning workflow, in which each edge device compresses its local model parameters into TT-format before transmitting them for global model updates; the receiving node then reconstructs those parameters to update its local model. While the reconstruction phase of TTD has been optimized extensively for years—particularly in cloud-server training environments with abundant computational resources—the compression phase is comparatively understudied in the context of edge deployment.

On-device TTD compression presents two main hurdles.

TABLE I: Simulation results comparing the performance of different TD methods for ResNet-32 on CIFAR-10.

Method	Accuracy (%)	Comp. ratio	Final #params
Uncompressed	92.49	1.0×	0.47M
Tucker Decomposition [14]	92.18	2.8×	0.16M
TRD [15]	91.44	2.7×	0.17M
TTD (Focus of this work)	92.09	3.4×	0.14M

First, its repeated use of singular value decomposition (SVD) imposes high computation and memory demands—especially prohibitive for low-power edge AI processors. Second, repeated matrix multiplications during the compression process can overwhelm edge AI processors equipped with dedicated GEMM accelerators, creating high latency and energy overhead from frequent data transfers between the main core and the accelerator [24].

In this work, we introduce *TT-Edge*, a framework that fundamentally rethinks TTD compression to better suit edge AI processors. Our approach consists of two key innovations:

- 1) **Efficient SVD through Bidiagonalization:** Instead of relying on QR iteration, we adopt a bidiagonalization technique inspired by ScaLAPACK [25], breaking SVD into two phases (bidiagonalization and diagonalization). Profiling on edge AI processors revealed that the bidiagonalization phase dominates SVD computation—about $3.6\times$ more time-consuming than diagonalization. We thus propose a specialized hardware accelerator for Householder-based bidiagonalization, carefully adapting the algorithm to edge constraints for seamless and efficient hardware integration.
- 2) **TTD-Engine Architecture:** To address the performance bottlenecks caused by frequent matrix–vector operations, we design a *TTD-Engine* that integrates directly with existing GEMM accelerators. This allows matrix operations to continue benefiting from GEMM hardware while substantially reducing the latency and energy penalties typically incurred by shuttling data back and forth between the main core and accelerator.

We implemented our *TT-Edge* framework in a RISC-V-based edge AI processor equipped with the proposed *TTD-Engine*. We designed the full RTL of the processor and performed FPGA prototyping to verify its functionality. Subsequently, we conducted performance evaluations on the prototype, confirming that the *TT-Edge* processor achieved a $1.7\times$ speedup compared to a conventional GEMM-based edge AI processor (i.e., baseline processor). Next, we synthesized our design using a 45nm PDK and carried out power simulations. Owing to *TT-Edge*’s lightweight design, the total processor power consumption increases by only 4% relative to the baseline. Moreover, because the *TTD-Engine* offloads compression from the processor core—enabling partial clock gating when not actively needed—the power overhead during TTD runs is, on average, only about 1% higher than a baseline in which the core remains continuously active. Overall, our solution yields a 40.2% energy reduction compared to the baseline, affirming that *TT-Edge* provides an efficient, low-power foundation for TTD-based model compression in distributed edge learning.

II. TTD ON EDGE AI: ALGORITHM, HOUSEHOLDER SVD ADAPTATION, AND KEY CHALLENGES

A. TTD-based Model Compression

1) *TTD Algorithm Overview:* TTD compresses an N -dimensional tensor $\mathbf{W} \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_N}$, composed of model parameters, into a set of 3D core tensors $\{\mathbf{G}_k\}_{k=1}^N$. Each core tensor $\mathbf{G}_k \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}$ satisfies boundary conditions $r_0 = r_N = 1$. The general TTD algorithm is described in Algorithm 1; it iteratively performs: *Reshape*, *SVD*, *Sorting*,

Truncation, and a set of matrix multiplications. We outline the key steps below.

a) *Reshape (line 7):* To begin, \mathbf{W} is reshaped into a matrix W_{temp} . For any tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$, one can rearrange its dimensions to $[J_1, J_2, \dots, J_M]$ provided that $\prod_{k=1}^N I_k = \prod_{m=1}^M J_m$. This operation preserves the overall ordering of elements while changing their dimensional layout.

b) *SVD (line 8):* Next, we compute the singular value decomposition (SVD) of W_{temp} , yielding $W_{\text{temp}} = U \Sigma V^T$. Here, $U \in \mathbb{R}^{M \times M}$ and $V \in \mathbb{R}^{N \times N}$ are orthogonal, whereas $\Sigma \in \mathbb{R}^{M \times N}$ contains the singular values on its diagonal (for $M > N$).

c) *Sorting (line 9):* We then sort the singular values in Σ (via bubble sort in line 19 of Algorithm 1), forming Σ_s . In this process, the column indices of Σ are recorded in an index vector *Ind*. Applying *Ind* reorders U and V^T accordingly, yielding U_s and V_s^T .

d) *Truncation (line 10):* In line 29 (δ -TRUNCATION), we form the truncated matrices U_t , Σ_t , and V_t^T . Given a threshold δ , let i be the smallest integer such that $\|\Sigma_s[i : \text{rank}(\Sigma_s), :]\|_F < \delta$, where $\|\cdot\|_F$ denotes the Frobenius norm. Columns of U_s and rows of V_s^T beyond that index are discarded, retaining only the dominant singular values. Finally, we multiply Σ_t by V_t^T (lines 11–12) and reshape U_t into a new core tensor \mathbf{G}_k .

e) *TTD Decoding:* Once the set of core tensors $\{\mathbf{G}_k\}_{k=1}^N$ is obtained, the original tensor \mathbf{W} can be approximated via

$$\mathbf{W}_R = \mathbf{G}_1 \times_1 \mathbf{G}_2 \times_1 \dots \times_1 \mathbf{G}_N, \quad (1)$$

where \times_1 denotes the tensor contraction operator.

In particular, for $\mathbf{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ and $\mathbf{Y} \in \mathbb{R}^{J_1 \times \dots \times J_M}$ with $I_N = J_1$, let $\mathbf{T} = \mathbf{X} \times_1 \mathbf{Y}$. Then, the contraction can be computed as:

$$\begin{aligned} \mathbf{T} &= \text{Reshape}(\mathbf{X}, [I_1 \dots I_{N-1}, I_N]) \cdot \text{Reshape}(\mathbf{Y}, [J_1, J_2 \dots J_M]), \\ \mathbf{T} &= \text{Reshape}(\mathbf{T}, [I_1, \dots, I_{N-1}, J_2, \dots, J_M]), \end{aligned} \quad (2)$$

which computes matrix multiplication and reshapes the result back into a tensor [26]. Repeating (2) across all cores in (1) reconstructs (approximately) the original parameters, completing the TTD-based compression and reconstruction procedure.

2) *Adapting Householder-based SVD Bidiagonalization:* SVD is a central step in TTD, as it decomposes a matrix into its singular values and corresponding basis matrices. To enable efficient SVD on resource-constrained platforms, we adopt a Householder Bidiagonalization (HBD) approach, which first reduces a matrix to an upper bidiagonal form and thereby lowers the complexity of the subsequent diagonalization [27].

a) *Bidiagonalizing A:* Given an $M \times N$ matrix A , the HBD process produces an upper bidiagonal matrix B . For each $i = 1, 2, \dots, \min(M, N)$, we consider the submatrix $A[i : M, i : N]$ and let $\mathbf{x} = A[i : M, i]$. We form a Householder vec. \mathbf{v} by

$$\mathbf{v} = \mathbf{x} - \text{sign}(x_1) \|\mathbf{x}\| e_1, \quad e_1 = [1, 0, \dots, 0]^T. \quad (3)$$

Then, the Householder reflector is defined as

$$H = I - 2 \frac{\mathbf{v} \mathbf{v}^T}{\mathbf{v}^T \mathbf{v}}, \quad (4)$$

which is applied from the left ($A \leftarrow HA$) to eliminate subdiagonal entries in $A[i : M, i : N]$. We refer to this step as the *left Householder transform*.

Next, we focus on the first row of $A[i : M, i+1 : N]$, denoted \mathbf{y} . We form another Householder vector:

$$\mathbf{v} = \mathbf{y} - \text{sign}(y_1) \|\mathbf{y}\| e_1, \quad e_1 = [1, 0, \dots, 0], \quad (5)$$

Algorithm 1 Tensor Train Decomposition

```

1: function TTD(W, ε)
2:   W : Input_Tensor ∈ ℝn1 × ... × nN
3:   ε : prescribed accuracy
4:   δ ←  $\frac{\epsilon}{\sqrt{d-1}} \times \|W\|_F$  : threshold of truncation
5:   Wtemp ← W, r0 ← 1
6:   for k = 1 to N-1 do
7:     Wtemp = Reshape(Wtemp, [rk-1nk,  $\frac{\text{numel}(W_{temp})}{r_{k-1}n_k}$ ])
8:     U, Σ, VT ← SVD(Wtemp)
9:     Us, Σs, VsT ← Sorting_Basis(U, Σ, VT)
10:    Ut, Σt, VtT ← δ-Truncation(Us, Σs, VsT, δ)
11:    Wtemp ← ΣtVtT, rk ← rank(Σt)
12:    New Core: Gk ← Reshape(Ut, [rk-1, nk, rk])
13:  end for
14:  GN = Reshape(Wtemp, [rN-1, nN, rN])
15:  return G1...GN
16: end function
17:
18: function Sorting_Basis(U, Σ, VT)
19:   Ind : Bubble Sorted index array
20:   Σs, Ind ← Bubble_Sort(Σ)
21:   for i = 1 to rank(Σ) do
22:     Us[:, Ind[i]] ← U[:, i], VsT[Ind[i], :] ← VT[i, :]
23:   end for
24:   return Us, Σs, VsT
25: end function
26:
27: function δ-Truncation(Us, Σs, VsT, δ)
28:   k ← min{i ∈ {1, ..., rank(Σs)} | ||Σs[i : rank(Σs), :]||F < δ}
29:   Ut, Σt, VtT ← Us[:, 1 : k], Σs[1 : k, 1 : k], VsT[1 : k, :]
30:   return Ut, Σt, VtT
31: end function

```

and compute

$$H = I - 2 \frac{v^T v}{v v^T}. \quad (6)$$

Applying H from the right ($A \leftarrow AH$) removes off-diagonal entries in that row. We call this the *right Householder transform*. Repeating the above left and right transforms for $i = 1, \dots, N$ yields an upper bidiagonal matrix B .

b) *Forming B*: By aggregating all left Householder reflectors, we obtain U_B , while those used on the right form V_B^T . Thus, $A = U_B B V_B^T$. B is the upper bidiagonal matrix.

c) *Diagonalizing B*: In the final step, B is diagonalized by a standard QR-based procedure: $B = U_D \Sigma V_D^T$, where U_D and V_D are orthonormal and Σ is a diagonal matrix of singular values. Combining these results, the SVD of A is expressed as $A = U \Sigma V^T$, where $U = U_B U_D$ and $V^T = V_D^T V_B^T$.

B. TTD Compression Challenges in Edge AI Processors

To identify the key challenges of performing TTD-based model compression on edge AI processors, we first assume a practical baseline architecture and then analyze the TTD compression workflow in detail. The baseline processor under consideration has the following main components:

- **GEMM Accelerator**: A hardware module supporting 16×16 matrix multiplication with a 320 KB on-chip scratchpad memory (SPM).
- **Processor Subsystem**: A Rocket RISC-V core [28] equipped with DDR3 DRAM, 128 KB of system SRAM, and various peripherals.
- **DMA Engine**: An integrated DMA unit that enables high-throughput data transfers between the core, GEMM accelerator, and memory without constant core intervention.

On this baseline processor, TTD-based compression can be split into operations handled solely by the core and those where the core collaborates with the GEMM accelerator. For instance, HBD, one key step in TTD, consists of generating

Householder vectors as defined in (3) and (5), followed by creating and applying the corresponding reflection matrices in (4) and (6). Taking the left transform as an example, $2 \frac{v}{v^T v} (v^T A)$ can be viewed as a combination of one vector–scalar division and two matrix multiplications. While Householder vector generation, scalar division, sorting, and truncation must be performed on the core (because the GEMM accelerator does not directly support them), large-scale matrix multiplications in TTD can be offloaded to the accelerator. However, the accelerator’s 16×16 processing limit requires dividing the entire matrix into multiple blocks—an approach known as blockwise matrix multiplication. In this scenario, the core is responsible for calculating each block’s address, dimensions, and data layout, sending these parameters to the accelerator, and transferring block data to the SPM.

These architectural constraints introduce several bottlenecks during TTD-based compression:

- 1) **Limited operation support**: The GEMM accelerator does not handle Householder vector generation, scalar division, sorting, and truncation, leaving the core to perform all of these tasks and causing significant computational load and latency.
- 2) **Communication overhead in blockwise multiplication**: Because the accelerator handles only 16×16 matrix blocks, the core frequently needs to compute block parameters and relay them to the accelerator, incurring cumulative latency from repeated communication.
- 3) **Frequent data movement**: Repeatedly moving large blocks between DRAM and the SPM for every matrix multiplication step substantially increases memory and interconnect latency.

In addition to these latency-related issues, such bottlenecks have a negative impact on power consumption and overall energy efficiency—key concerns in edge AI environments. The core must sustain a high operational load to manage non-GEMM operations, while also coordinating blockwise GEMM tasks and continuously interacting with the accelerator. Consequently, the system’s energy efficiency suffers due to the compounding overhead of intensive core activity and frequent accelerator communication.

In this paper, we present an integrated HW–SW approach for efficient TTD on resource-constrained edge AI processors.

III. PROPOSED TT-EDGE SOLUTION

To tackle the latency and energy challenges, we propose the *TT-Edge* solution to mitigate the latency issues while also addressing power and energy consumption in edge AI processors. First, the main ideas for reducing latency can be summarized as follows:

- 1) **Dedicated TTD compression accelerator**: To reduce the core’s high computational latency, we introduce a lightweight accelerator specialized for TTD-based compression. Offloading tasks such as Householder transformations and SVD operations from the core significantly lowers end-to-end latency.
- 2) **Direct interconnection with the GEMM accelerator**: To alleviate the repeated round trips through the system interconnect, we enable a direct connection between the

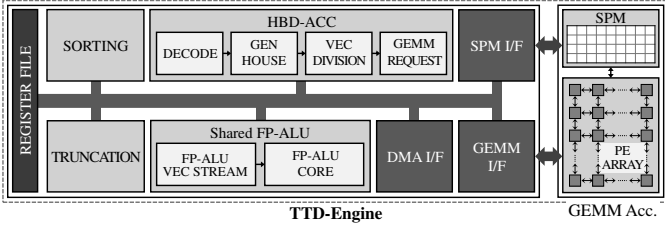


Fig. 2: System-level architecture of the proposed TTD-Engine.

TTD accelerator and the GEMM accelerator, allowing blockwise GEMM parameters to be computed on the new accelerator rather than the core.

- 3) **On-Chip retention of Householder vectors:** A fundamental solution to memory-interconnect latency (e.g., redesigning the entire memory and network systems) lies beyond the scope of this TTD-focused study. Instead, we propose a practical approach: store the Householder vectors within the SPM during the bidiagonalization process, preventing repeated DRAM access for these vectors.

Next, the ideas to address energy-related issues are:

- 1) **Lightweight hardware design for energy efficiency:** Beyond reducing latency, TT-Edge must also tackle power and energy constraints. Through a HW-SW co-design approach, we modify the TTD algorithm to reuse common hardware resources with minimal redundancy. This careful optimization keeps the accelerator’s power consumption low.
- 2) **Maximizing core clock gating:** To increase the time during which the core can remain clock-gated, the dedicated accelerator is designed to support as many TTD operations as possible. This approach maximizes offloading to the accelerator and further reduces power consumption.
- 3) **Integrating the existing GEMM accelerator into the dedicated accelerator:** Since TTD and AI processes are mutually exclusive, the existing GEMM accelerator typically remains in standby during TTD execution. To take full advantage of its capabilities, we incorporate the GEMM accelerator inside our specialized design, enabling it to be fully utilized during TTD processing.

We collectively refer to this dedicated accelerator as the *TTD-Engine*. Fig. 2 shows the overall TTD-Engine architecture, which is centered on the HBD-ACC module for accelerating Householder Bidiagonalization. Supporting components include the Shared FP-ALU for floating-point operations, as well as SORTING and TRUNCATION modules to complete the TTD workflow. In the following subsections, we detail the design of each module and demonstrate how they address the latency and energy bottlenecks.

A. Design of the HBD-ACC

The HBD-ACC is the core component for efficiently accelerating the HBD process with minimal resource usage. Because HBD involves different procedures (left or right transforms) depending on the input vector orientation, we first unify these steps into a single flow to enable reuse of low-power hardware blocks. Algorithm 2 depicts our modified Householder transform procedure, combining:

- Householder Reduction (lines 4–13): Applies Householder transforms to the original matrix.

Algorithm 2 Proposed Householder Bidiagonalization for the HBD-ACC in the TTD-Engine.

```

1: function HOUSEHOLDER_BIDIAGONALIZE(A)
2:   Set  $B$  to an  $N \times N$  zero matrix
3:   Set  $U_B, V_B$  to  $M \times N, N \times N$  identity matrix
4:   for  $i = 1$  to  $N$  : do
5:      $B[i, i], v_L \leftarrow \text{HOUSE}(A[i : M, i])$ 
6:      $\text{HOUSE\_MM\_UPDATE}(B[i, i], v_L, A[i : M, i + 1 : N], 0)$ 
7:      $A[i, i] \leftarrow v_L[1]$ 
8:     if  $i < N$  then
9:        $B[i, i + 1], v_R \leftarrow \text{HOUSE}(A[i, i + 1 : N])$ 
10:       $\text{HOUSE\_MM\_UPDATE}(B[i, i + 1], v_R, A[i + 1 : M, i + 1 : N], 1)$ 
11:       $A[i, i + 1] \leftarrow v_R[1]$ 
12:    end if
13:  end for
14:  for  $i = N$  to  $1$  : do
15:     $v_L \leftarrow A[i : M, i], v_R \leftarrow A[i, i + 1 : N]$ 
16:     $\text{HOUSE\_MM\_UPDATE}(B[i, i], v_L, U_B[i : M, i + 1 : N], 0)$ 
17:     $\text{HOUSE\_MM\_UPDATE}(B[i, i + 1], v_R, V_B^T[i + 1 : N, i + 1 : N], 1)$ 
18:  end for
19:  return  $U_B, B, V_B^T$ 
20: end function
21:
22: function HOUSE( $x$ )
23:    $v \leftarrow x, q \leftarrow -\text{sign}(v[1]) * \|v\|, v[1] \leftarrow v[1] + \text{sign}(v[1]) * \|v\|$ 
24:   return  $q, v$ 
25: end function
26:
27: procedure HOUSE_MM_UPDATE( $q, v, \text{SubArray}, \text{order}$ )
28:    $\beta \leftarrow v[1] * q$ 
29:    $\text{vec}_1 \leftarrow (\text{order} = 0)? v / \beta : v \times \text{SubArray}^T$ 
30:    $\text{vec}_2 \leftarrow (\text{order} = 0)? v^T \times \text{SubArray} : v / \beta$ 
31:    $\text{SubArray} \leftarrow \text{SubArray} + \text{vec}_1 \times \text{vec}_2$ 
32: end procedure

```

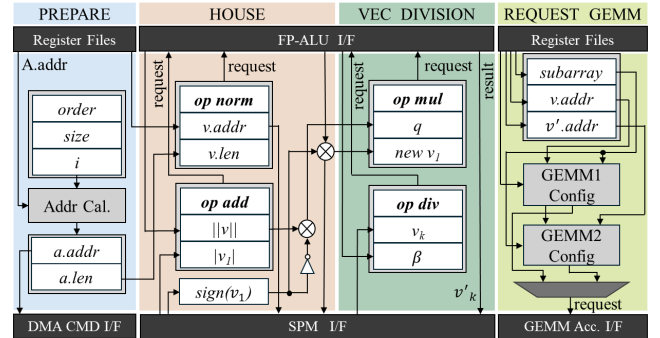


Fig. 3: Architecture of the HBD-ACC.

- Householder Accumulation (lines 14–18): Updates and accumulates the Householder matrices.

In this process, the *HOUSE* function corresponds to generating Householder vectors (as in (3) and (5)), while the creation and application of reflection matrices are consolidated into a single procedure, *HOUSE_MM_UPDATE*.

Leveraging this unified algorithm, we design the HBD-ACC to independently execute the entire HBD routine and minimize frequent interaction with the main core. Fig. 3 illustrates its architecture, which implements the *HOUSE* and *HOUSE_MM_UPDATE* steps in four stages: PREPARE, HOUSE, VEC DIVISION, and REQUEST GEMM.

During PREPARE, the address calculator determines the address for the Householder vector ($a.addr = A.width + 1 + order$) based on input parameters *order*, *size* and *i*, where *size* contains the matrix dimensions (width and height), then issues a DMA request to fetch vector *a* from external memory into the SPM (as vector *v*). In the HOUSE stage, the shared FP-ALU computes the norm of *v* and combines it with *v*[1] to generate *q*, which defines the Householder transformation.

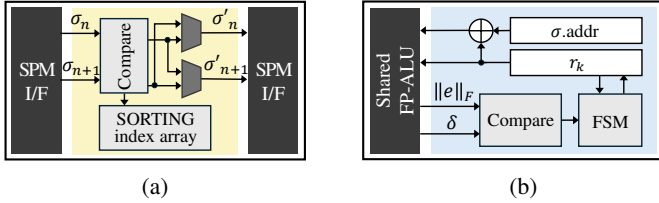


Fig. 4: Architecture of (a) SORTING and (b) TRUNCATION modules.

HOUSE_MM_UPDATE is common to both Householder Reduction and Accumulation. In the *VEC DIVISION* stage, the FP-ALU calculates the scaling factor β by multiplying q and $v[1]$ from the preceding stage (or reading $v[1]$ from the SPM, depending on the specific phase). It then computes each element of v' by dividing the corresponding element of v by β and storing the result back in the SPM. Finally, the *REQUEST GEMM* stage issues two consecutive GEMM operations: the first multiplies v with a submatrix of A (either transposed or not, depending on *order*), while the second multiplies that result by v' . The accelerator repeats these operations, incrementing i and toggling *order* as needed, until the HBD process completes.

B. Design of the SORTING and TRUNCATION Modules

In addition to HBD, the TTD-Engine accelerates the *Sorting* and *Truncation* phases of TTD, both of which can incur heavy data movement if executed on the main core. By introducing dedicated modules, we significantly reduce latency from repeated data retrieval. First, Fig. 4a shows the SORTING module, which implements a bubble-sort-based algorithm for singular values stored in the SPM. For each pair of adjacent singular values (σ_n, σ_{n+1}) , the shared FP-ALU compares them and stores the sorted results $(\sigma'_n, \sigma'_{n+1})$ back in the SPM, updating a *SORTING index vector* to track the new order. Once sorting is complete, the module reorders the vectors composing matrices U and V according to the *SORTING index vector*.

Next, Fig. 4b illustrates the TRUNCATION module, which accelerates the δ -Truncation procedure and employs a lightweight FSM to control its operations. At the start of TTD, the module computes the threshold δ by obtaining the Frobenius norm of the input tensor, $\delta = \frac{\epsilon}{\sqrt{d-1}} \|\mathbf{W}\|_F$, which is simplified to the norm of the singular values σ from the first SVD. The shared FP-ALU sequentially performs *SQRT*, *MUL*, and *DIV* operations to obtain δ . For each truncation request, the module examines the tail of the singular-value vector to form an error vector e and checks $\|e\|_2 > \delta$. If the condition holds, it updates the truncated rank r_k ; otherwise, it decrements r_k and repeats the process until the desired accuracy is achieved.

C. Design of the Shared FP-ALU

Although the GEMM accelerator handles matrix multiplications, TTD still requires a range of additional floating-point operations. To reduce latency and hardware overhead,

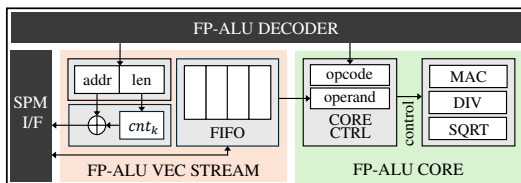


Fig. 5: Architecture of the Shared FP-ALU.

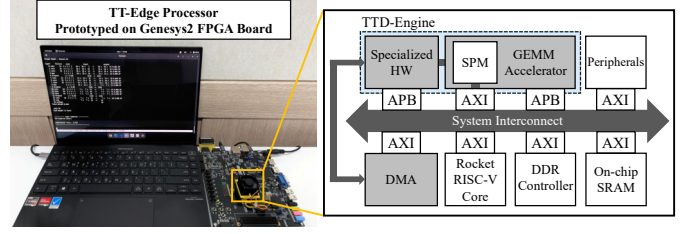


Fig. 6: Block diagram of the TT-Edge processor, prototyped on a Genesys2 FPGA board.

we incorporate a single Shared FP-ALU that is shared across the HBD-ACC, SORTING, and TRUNCATION modules. Fig. 5 shows the Shared FP-ALU, which comprises an FP-ALU Vector Streamer—responsible for loading and storing vector elements from and to the SPM—and an FP-ALU CORE, containing a customized set of floating-point units (MAC, DIV, and SQRT) derived from an open-source FPU [29].

Because TTD frequently computes vector norms, we design the FP-ALU to be able to provide a dedicated *norm* operation. When given the opcode and vector address/length, the streamer sequentially reads elements from the SPM into a FIFO, and the FP-ALU CORE squares and accumulates them via *MAC* operations before applying a final *SQRT*. It also supports single operations (*ADD*, *MUL*, *MAC*, *DIV*, and *SQRT*) by sending operands directly to the FP-ALU CORE.

By centralizing all floating-point operations in this Shared FP-ALU, we avoid replicating multiple floating-point units across different modules, thereby reducing resource overhead and improving the overall energy efficiency of the TTD-Engine.

IV. EXPERIMENTAL WORK

A. Implementation

We first developed two processor designs: the baseline processor and the TT-Edge processor integrating our proposed TTD-Engine. Both designs were implemented at the RTL level in Verilog HDL using RISC-V eXpress (RVX), an EDA tool widely adopted for developing processors on the RISC-V platform [30]–[34]. The baseline processor consists of a Rocket core supporting floating-point instructions, a DDR3 system memory interface, a 64-PE GEMM accelerator, a DMA engine, and additional peripherals for external interfaces and control. In the baseline design, the GEMM accelerator’s control interface employs APB, whereas its data interface is implemented with AXI to ensure low-latency transfers to and

TABLE II: Resource usage of the TT-Edge processor prototyped on Genesys2 FPGA and post-synth. power breakdown at 45 nm.

	IPs	LUTs	FFs	Power (mW) [*]
	Rocket RISC-V Core	15,041	9,890	10.90 / 2.63 ^{**}
	SRAM	166	323	1.87
	DDR Controller	7,961	7,581	89.12
	Peripherals incl. DMA	5,047	10,373	10.60
	System Interconnect	9,748	17,376	17.78
	GEMM Accelerator	84,150	32,939	40.77
TTD-Engine	Specialized HW Modules	7,273	6,517	7.19
	L HBD-ACC	1,346	1,411	1.42
	L TRUNCATION	413	884	0.78
	L SORTING	756	476	0.49
	L FP-ALU	3,314	2,287	2.23
	L Interfaces	1,412	1,167	1.43

^{*}Power analysis obtained from PrimeTime PX. ^{**}No clock gating / With clock gating.

from the on-chip scratchpad memory (SPM). By contrast, the TT-Edge processor replaces the baseline’s GEMM accelerator with our TTD-Engine. Its control signals connect to the system interconnect via APB, while dedicated interfaces link the TTD-Engine with the processor’s DMA, SPM, and GEMM hardware, achieving tight integration. Fig. 6 illustrates the overall architecture of the TT-Edge processor.

Next, the TT-Edge processor was prototyped on a Digilent Genesys2 board [35] featuring a Xilinx Kintex7-325T FPGA device [36], operating at 100 MHz. Table II summarizes the resource utilization of each IP block in our processor prototype. Focusing first on the specialized hardware modules within the TTD-Engine (excluding the existing GEMM accelerator), we observe that the HBD-ACC—which handles the bulk of TTD computations—consumes 18.5% of LUTs and 21.7% of FFs. Meanwhile, the SORTING and TRUNCATION modules occupy 10.4% and 5.7% of LUTs (and 7.3% and 13.6% of FFs), respectively, and the Shared FP-ALU takes up 45.6% of LUTs and 35.1% of FFs. Furthermore, the DMA, SPM, and GEMM interface logic, along with the associated interconnect, account for 19.4% of LUTs and 17.9% of FFs in the TTD-Engine. Because all TTD operations share the GEMM accelerator’s existing SPM, the TTD-Engine requires no additional BRAM resources. From a system-wide perspective, the TTD-Engine introduces minimal overhead by leveraging the existing GEMM accelerator and maintaining a lightweight design for its specialized hardware modules. In fact, the additional resources for the TTD-Engine contributes only 5.6% of LUTs and 7.7% of FFs across the entire processor, confirming that its resource footprint remains modest.

Finally, we implemented a benchmark application based on ResNet-32 model compression to validate our proposed techniques. This application employs TTD with a 3.4× compression ratio for model parameters. We integrated a clock-gating API into the application so that the main core can be clock-gated during HBD, sorting, and truncation—stages handled by the TTD-Engine. Running this application on our FPGA prototype allowed us to evaluate and validate the effectiveness of the proposed TT-Edge design.

B. Verification and Evaluation

We synthesized the proposed TT-Edge processor using Synopsys Design Compiler [37] under the Nangate 45nm process technology [38], and then performed detailed power analysis with Synopsys PrimeTime PX [39]. The results are also reported in Table II. Without clock gating on the main core, TT-Edge consumes a total of 178.23 mW, representing only about a 4% increase relative to the baseline processor’s 171.04 mW. This increment is attributable to the specialized hardware modules in the TTD-Engine, excluding the reused GEMM accelerator. Breaking down the TTD-Engine’s components, the HBD-ACC contributes 1.42 mW (19.7%), while the SORTING and TRUNCATION modules consume 0.49 mW (6.8%) and 0.78 mW (10.8%), respectively. The Shared FP-ALU occupies 2.23 mW (31%), and the remaining DMA/SPM/GEMM interface and interconnect logic draws 1.43 mW (19.9%).

Notably, the main core can enter a sleep state (i.e., be clock-gated) during the primary TTD compression phases—

TABLE III: Execution time T_{exec} and energy E breakdown for TTD-based ResNet-32 compression on baseline and TT-Edge.

TTD procedure	Baseline		TT-Edge	
	T_{exec} (ms)	E (mJ)	T_{exec} (ms)	E (mJ)
HBD	5626.42	962.17	2743.80	466.34*
QR Decomp.	1554.66	265.91	1554.66	277.09
Sort. & Trunc.	312.56	53.46	31.37	5.33*
Update SVD In.	46.65	8.15	46.65	8.49
Reshape & etc	189.24	32.37	189.24	33.73
Total	7729.52	1322.06	4566.71	790.97

*The core is clock gated.

HBD, Sorting, and Truncation. In this scenario, TT-Edge operates at 169.96 mW, which is even less than the baseline’s power consumption. This reduction demonstrates the success of our lightweight TTD-Engine design, wherein each submodule (HBD-ACC, SORTING, TRUNCATION) shares a single FP-ALU, and the existing GEMM accelerator is reused to minimize additional hardware requirements.

The performance improvements achieved by TT-Edge are reported in Table III. More specifically, the table compares the execution times of a TTD-based model compression application on both the baseline and TT-Edge processors, divided into HBD, QR decomposition, Sorting & Truncation, Update SVD Input, and Reshape & etc. The most time-consuming step, HBD, requires 5626.42 ms on the baseline—72.8% of the total TTD runtime—but only 2743.8 ms on TT-Edge, corresponding to a 2.05× speedup. The Sorting & Truncation step also shows dramatic gains, dropping from 312.56 ms on the baseline to 31.37 ms on TT-Edge, yielding a 9.96× acceleration. As a result, the total TTD runtime is reduced from 7729.52 ms to 4566.71 ms, representing an overall 1.7× speedup.

Furthermore, Table III presents the energy savings achieved by TT-Edge. The table reports the energy consumption of the benchmark application’s main procedures on the baseline and TT-Edge processors, highlighting a 51.5% reduction during HBD and a 90% reduction in the Sorting & Truncation step—both of which clock-gate the main core. Overall, TT-Edge reduces total energy consumption by approximately 40.2% compared to the baseline, demonstrating the effectiveness of our solution in delivering substantial improvements in both performance and energy efficiency.

V. CONCLUSION

We presented *TT-Edge*, a HW–SW co-designed framework that accelerates TTD-based model compression on resource-limited edge AI processors. By splitting SVD into Householder-based bidiagonalization and QR-based diagonalization, and offloading the heavy computations to a lightweight dedicated accelerator, TT-Edge achieves efficient compression with modest hardware cost. The TTD-Engine is tightly integrated with the existing GEMM unit, enabling resource reuse and reducing matrix–vector transfers, which substantially lowers both latency and energy consumption. Implemented on a RISC-V-based processor prototype, TT-Edge achieves a 1.7× speedup and a 40.2% reduction in energy consumption compared to the baseline, with only a 4% power overhead. These results demonstrate that advanced tensor decompositions can be made practical even under the constraints of low-power edge devices.

REFERENCES

- [1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017, pp. 1273–1282.
- [2] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE signal processing magazine*, vol. 37, no. 3, pp. 50–60, 2020.
- [3] S. Li, T. Zhou, X. Tian, and D. Tao, "Learning to collaborate in decentralized learning of personalized models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 9766–9775.
- [4] C. Cho, Y. Son, S. Park, and Y. G. Kim, "Clover: Carbon optimization of federated learning over heterogeneous clients," in *Proceedings of the 29th ACM/IEEE International Symposium on Low Power Electronics and Design*, 2024, pp. 1–6.
- [5] L. Yuan, Z. Wang, L. Sun, S. Y. Philip, and C. G. Brinton, "Decentralized federated learning: A survey and perspective," *IEEE Internet of Things Journal*, 2024.
- [6] Y. Zhou, Q. Ye, and J. Lv, "Communication-efficient federated learning with compensated overlap-fedavg," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 192–205, 2021.
- [7] Z. Qin, G. Y. Li, and H. Ye, "Federated learning and wireless communications," *IEEE Wireless Communications*, vol. 28, no. 5, pp. 134–140, 2021.
- [8] W. Liu, L. Chen, and W. Zhang, "Decentralized federated learning: Balancing communication and computing costs," *IEEE Transactions on Signal and Information Processing over Networks*, vol. 8, pp. 131–143, 2022.
- [9] R. Yu and P. Li, "Toward resource-efficient federated learning in mobile edge computing," *IEEE Network*, vol. 35, no. 1, pp. 148–155, 2021.
- [10] R. Song, L. Zhou, L. Lyu, A. Festag, and A. Knoll, "Resfed: Communication efficient federated learning with deep compressed residuals," *IEEE Internet of Things Journal*, 2023.
- [11] F. M. A. Khan, H. Abou-Zeid, and S. A. Hassan, "Deep compression for efficient and accelerated over-the-air federated learning," *IEEE Internet of Things Journal*, 2024.
- [12] W. Dai, J. Fan, Y. Miao, and K. Hwang, "Deep learning model compression with rank reduction in tensor decomposition," *IEEE Transactions on Neural Networks and Learning Systems*, 2023.
- [13] M. Yin, Y. Sui, S. Liao, and B. Yuan, "Towards efficient tensor decomposition-based dnn model compression with optimization framework," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 10 674–10 683.
- [14] M. Yin, H. Phan, X. Zang, S. Liao, and B. Yuan, "Batude: Budget-aware neural network compression based on tucker decomposition," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 8, 2022, pp. 8874–8882.
- [15] N. Li, Y. Pan, Y. Chen, Z. Ding, D. Zhao, and Z. Xu, "Heuristic rank selection with progressively searching tensor ring network," *Complex & Intelligent Systems*, pp. 1–15, 2021.
- [16] H. Ren, Y. Zhou, H. Fu, Y. Huang, R. Xu, and B. Cheng, "Ttpoint: A tensorized point cloud network for lightweight action recognition with event cameras," in *Proceedings of the 31st ACM International Conference on Multimedia*, 2023, pp. 8026–8034.
- [17] C. Yin, B. Acun, C.-J. Wu, and X. Liu, "Tt-rec: Tensor train compression for deep learning recommendation models," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 448–462, 2021.
- [18] C. Yin, D. Zheng, I. Nisa, C. Faloutsos, G. Karypis, and R. Vuduc, "Nimble gnn embedding with tensor-train decomposition," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 2327–2335.
- [19] Y. Ren, B. Wang, L. Shang, X. Jiang, and Q. Liu, "Exploring extreme parameter compression for pre-trained language models," *arXiv preprint arXiv:2205.10036*, 2022.
- [20] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, "TIE: Energy-efficient tensor train-based inference engine for deep neural network," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 264–278.
- [21] Y. Gong, M. Yin, L. Huang, J. Xiao, Y. Sui, C. Deng, and B. Yuan, "ETTE: Efficient tensor-train-based computing engine for deep neural networks," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.
- [22] D. Lee, R. Yin, Y. Kim, A. Moitra, Y. Li, and P. Panda, "TT-SNN: Tensor train decomposition for efficient spiking neural network training," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.
- [23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [24] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 58–70.
- [25] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet *et al.*, *ScaLAPACK users’ guide*. SIAM, 1997.
- [26] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.
- [27] S. Lahabar and P. Narayanan, "Singular value decomposition on GPU using CUDA," in *2009 IEEE international symposium on parallel & distributed processing*. IEEE, 2009, pp. 1–10.
- [28] SIFIVE, <https://github.com/chipsalliance/rocket-chip>, accessed 12 Jan. 2026.
- [29] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 774–787, 2020.
- [30] K. Han, S. Lee, K.-I. Oh, Y. Bae, H. Jang, J.-J. Lee, W. Lee, and M. Pedram, "Developing TEI-aware ultralow-power soc platforms for iot end nodes," *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4642–4656, 2021.
- [31] J. Park, E. Choi, K. Lee, J.-J. Lee, K. Han, and W. Lee, "Developing an ultra-low power RISC-V processor for anomaly detection," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–2.
- [32] J. Park, K. Han, E. Choi, J.-J. Lee, K. Lee, W. Lee, and M. Pedram, "Designing low-power RISC-V multicore processors with a shared lightweight floating point unit for IoT endnodes," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 71, no. 9, pp. 4106–4119, 2024.
- [33] K. Lee, S. Jeon, K. Lee, W. Lee, and M. Pedram, "Radar-PIM: Developing IoT processors utilizing Processing-in-Memory architecture for ultrawideband-radar-based respiration detection," *IEEE Internet of Things Journal*, vol. 12, no. 1, pp. 515–530, 2025.
- [34] J. Choi, E. Choi, S. Choi, and W. Lee, "E-BTS: A low-power event-driven blink tracking system with hardware-software co-optimized design for real-time driver drowsiness detection," *Alexandria Engineering Journal*, vol. 128, pp. 867–877, 2025.
- [35] Genesys2, <https://digilent.com/reference/programmable-logic/genesys-2/reference-manual>, accessed 12 Jan. 2026.
- [36] Kintex-7, <https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/kintex-7.html>, accessed 12 Jan. 2026.
- [37] Synopsys, <https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html>, accessed 12 Jan. 2026.
- [38] NCSU, <https://eda.ncsu.edu/freepdk/freepdk45>, accessed 12 Jan. 2026.
- [39] Synopsys, <https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html>, accessed 12 Jan. 2026.