This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Hardware-Enabled Efficient Data Processing with Tensor-Train Decomposition

Zheng Qu, Lei Deng, *Member*, *IEEE*, Bangyan Wang, Hengnu Chen, Jilan Lin, Ling Liang, Guoqi Li, *Member*, *IEEE*, Zheng Zhang, *Member*, *IEEE*, Yuan Xie, *Fellow*, *IEEE*

***Abstract*—In recent years, tensor computation has become a promising tool for solving big data analysis, machine learning, medical image and EDA problems. To ease the memory and computation intensity of tensor processing, decomposition techniques, especially Tensor-train Decomposition(TTD), are widely adopted to compress the extremely high-dimensional tensor data. Despite TTD's potential to break the curse of dimensionality, researchers have not yet leveraged its full computational potential, mainly because of two reasons:(1) Executing TTD itself is time- and energy-consuming due to the singular value decomposition(SVD) operation inside each of TTD's iteration; (2) Additional software/hardware optimizations are often required to process the obtained TT-format data in certain applications such as deep learning inference.**

In this paper, we address these challenges with two approaches. Firstly, we propose an algorithm-hardware co-design with customized architecture namely TTD Engine to accelerate TTD. We use MRI image compression as a demo application to illustrate the efficacy of the proposed accelerator. Secondly, we present a case study demonstrating the benefit of TT-format data processing and the efficacy of using TTD Engine. In the case study, we use TT approach to realize convolution operation, which is difficult and nontrivial for TT-format data. Experimental results show that, TTD Engine achieves, on average, $14.9\times \sim 36.9\times$ speedup over CPU implementations and $4.1\times \sim 9.9\times$ speedup compared to the GPU baseline. The energy efficiency is also improved by at least $14.4\times$ and $5.4\times$ over CPU and GPU, respectively. Moreover, our hardware-enabled TT-format data processing further leads to more efficient implementations of complicated operations and applications.

***Index Terms*—Tensor-Train Decomposition, Algorithm Hardware Co-design, TT-format Data Processing**

I. Introduction

TENSOR is a high-dimensional generalization of vector and matrix, and is a natural choice for efficiently solving high-dimensional big data analysis problems. Compared with matrix analysis, multiway data processing is more versatile and has the potential to capture multiple interactions and couplings [1]. Previous studies have demonstrated its use in diverse branches of data analysis, such as EDA, signal and image processing, biometrics, quantum computing, and so forth [1]–[6]. Nevertheless, processing big data with tensor-based approaches is challenging due to the high dimensionality and large data size. Therefore, more and more attentions are drawn to tensor decomposition to compress tensors in terms of both dimension and size, which has been playing an important role in data mining, pattern recognition, object detection and classification [7]–[13].

Tensor-train decomposition (TTD) [14] is one of the most popular tensor decomposition methods because of its ability in providing highly compressed tensor data while keeping significant computation accuracy with customizable constraints. More importantly, it also enables efficient data processing on the base of TT-format data. However, there are still challenges existed in TT-format data processing. The reasons are of two folds. First, obtaining the TT representation, the initial step for TT-based data processing, is time consuming because of the iterative decomposition procedure over large-scale tensor data. In each of the TTD iteration, a truncated singular value decomposition (SVD) is used to decompose a large intermediate matrix, which is both memory- and compute-intensive. Second, there is a big gap to adapt a typical algorithm to the TT-based method. On one hand, normal operations like addition and multiplication cause the TT-rank to grow significantly [14], which require us to approximate the TT result afterwards. On the other hand, some simple element-wise operations like ReLU in neural networks, can be very complicated for TT-format data, because each of the original element is now represented as a sequence of matrix multiplication. Therefore, additional efforts are needed if we want to effectively take advantage of the TT-format data analysis. Previous work have mainly focused on directly using TT-format data to perform simple computations like matrix multiplications [3], [13]. However, efficient execution of TTD itself and implementing more complicated operations in TT format are rarely touched.

In this work, we aim at addressing the mentioned problems with the following approaches. (1) To reduce the TTD overhead, we propose TTD Engine, the first customized architecture for efficient execution of the TTD algorithm. Instead of naively implementing the original TTD algorithm, we adapt it by virtue of the special high-order tensor data structure as well as data sparsity and symmetricity. (2) To bridge the gap between TT-format data and application algorithms, we move forward by proposing a decomposed computation pattern for element-wise operations and resolving the rank-growth issue with the help of TTD Engine. We conduct a case study on the base of TTD Engine to implement convolutional operations

This work was partially supported by National Science Foundation (Grant No. 1725447, No. 1817037). Corresponding author: Lei Deng. Zheng Qu, Bangyan Wang, Jilan Lin, Ling Liang, Zheng Zhang, and Yuan Xie are with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA (email: {zhengqu, bangyan, jilan, linglinag, zhengzhang, yuanxie}@ucsb.edu). Lei Deng, Hengnu Chen and Guoqi Li are with the Department of Precision Instrument, Center for Brain Inspired Computing Research, Tsinghua University, Beijing 100084, China (email: leideng@ucsb.edu, chn18@mails.tsinghua.edu.cn, liguoqi@mail.tsinghua.edu.cn).

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

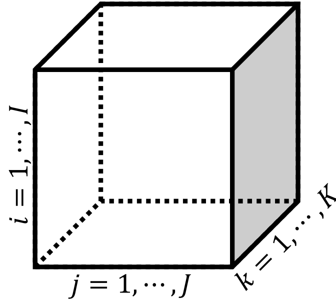


Fig. 1: A 3rd-order tensor.

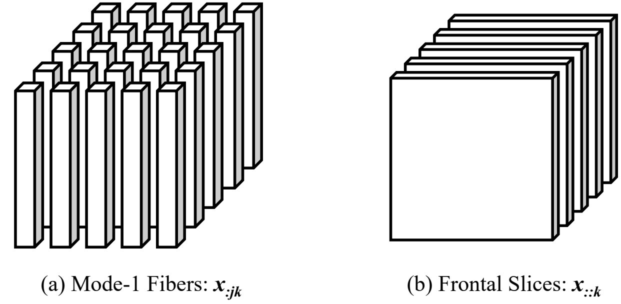


Fig. 2: Fibers and slices of a 3rd-order tensor.

over TT-format data, which are considered to be difficult and inefficient for TT-based data processing. We show that with specialized hardware support and algorithm design, it is possible and beneficial to reformulate the existing operations in various applications using the TT format to achieve better efficiency.

Our contributions in this work are summarized as follows:

- We develop a hardware friendly computing scheme for TTD by adjusting the computation pattern of SVD within each TTD iteration. The modified SVD explores data sparsity and symmetricity during the computation process to reduce the overall compute cost.
- Based on the proposed scheme, we present the first TTD accelerator with decoupled PE array design and optimized dataflow. Experimental results show that TTD Engine achieves up to 36.9× and 9.9× speedup over state-of-the-art CPU and GPU implementations respectively, while providing significant improvements on energy efficiency. We further use a real-world MRI image dataset to perform image compression as a demo application on the proposed TTD accelerator.
- We demonstrate the benefit of hardware-enabled TT-format data analysis by addressing the rank-growth issue with TTD Engine and proposing a decomposed computation pattern for element-wise operations. A case study is presented to use TTD Engine to accelerate data convolution which shows considerable speedup over CPU when dealing with large-scale vectors.

II. Background

A. Tensor Knowledge and Notations

Tensors are multidimensional data arrays, which can be viewed as natural generalizations of vectors and matrices. Each dimension has its own coordinates and length. An N-way tensor, also called an Nth-order tensor, is a tensor with N dimensions or modes. For example, a third-order tensor has three indices, and can be visually described by Figure 1. Under this setting, vectors can be viewed as first-order tensors and denoted as **a**, while matrices are second-order tensors that we denote as A. Finally, high-dimensional tensors are represented with $\mathcal{A}$ in the further content. A real-valued tensor of order N can be denoted as $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and its entry is $a_{i_1,i_2,\ldots,i_N}$.

By using only a subset of the indices in the original tensor and fixing the rest, we can get a subtensor. Particularly, a vector-valued subtensor, also termed as a fiber, is generated by using only one index from the original tensor. A matrix-valued subtensor uses two indices, and is therefore called a slice as shown in Figure 2.

The unfolding matrix of a tensor is generated by reordering the elements of the original N-way tensor into a matrix. In our paper, we focus on the special case of the unfolding matrix, which is called the mode-n unfolding matrix. Its definition is concise. Specifically, for a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, its mode-$n$ unfolding matrix is generated by arranging the mode-n fibers to be the column of the target matrix, and is denoted as $A_{(n)}$. The notation of the unfolding matrix will be further used when we describe the TTD algorithm.

Generalized from matrix multiplication, two tensors can also be multiplied together to form up a new tensor, such process is called tensor contraction. The full definition and procedure of tensor multiplication is much more complicated than those in the matrix case, which are detailed for example in [15]. Here in this work, we only consider the mode-n contraction, i.e., multiplying a tensor by a matrix (or vector). Given tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and matrix $M \in \mathbb{R}^{J \times I_n}$, then the mode-$n$ product $\mathcal{A} \times_n M \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$ is obtained by the contraction over the nth dimension, i.e. each element of $\mathcal{A} \times_n M$ equals $\sum_{i_n=1}^{I_n} x_{i_1} x_{i_2} \cdots i_N \times m_{ji_n}$.

B. Tensor Train Decomposition (TTD)

TTD is originally proposed by Oseledets in [14]. The overall procedure of the naive TTD algorithm is given in Alg. 1. In TTD, we try to approximately represent a given tensor $\mathcal{A}$ with tensor $\mathcal{B}$, which can be described as:

$$\mathcal{B}_{i_1,i_2\cdots,i_d} = G_1(i_1)G_2(i_2)\cdots G_d(i_d). \tag{1}$$

Each $G_k(i_k)$ is an $r_{k-1} \times r_k$ matrix, where r_k is called the TT-rank that can be either predefined before the decomposition or decided during runtime according to the required decomposition accuracy. G_k is an $r_{k-1} \times I_k \times r_k$ tensor core extracted from the original high-order tensor. In each TTD iteration, we need to perform Singular Value Decomposition (SVD) of an auxiliary matrix to get a tensor core. Therefore, it takes d sequential TTD iterations to finish the decomposition of a given tensor. Besides, at the beginning of each iteration, we need to reshape the given matrix into the required size

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

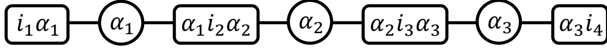


Fig. 3: A tensor-train network.

Algorithm 1 TT-SVD

Require: d-dimensional tensor $\mathcal{A}$, approximation error ϵ.
Ensure: Tensor cores $G_1, ..., G_d$ of the TT-approximation $\mathcal{B}$ in the TT format with TT-ranks r_k equal to the δ-ranks of the unfoldings A_k of $\mathcal{A}$. The approximation error satisfies:

$$||\mathcal{A} - \mathcal{B}||_F \leq \epsilon ||\mathcal{A}||_F$$

1: {Initialization} Compute the truncation parameter: $\delta = \epsilon\sqrt{d-1}||\mathcal{A}||_F$
2: Temporary tensor: $C = \mathcal{A},\ r_0 = 1$.
3: **for** $k = 1$ to $d-1$ **do**
4: C =reshape(C, $[r_{k-1}I_k, numel(C)/(r_{k-1}I_k)]$)
5: Compute δ-truncated SVD: $C = USV^T + E,\ ||E||_F \leq \delta,\ r_k = rank_\delta(C)$
6: New tensor core: G_k =reshape(U, $[r_{k-1}, I_k, r_k]$)
7: $C = SV^T$
8: **end for**
9: $G_d = C$
10: Return tensor $\mathcal{B}$ in the TT format represented by tensor cores $G_1, ..., G_d$.

before we can perform SVD. With the TT-format data, we can simply contracting these tensor cores together to reconstruct the approximated tensor which is close to the original tensor $\mathcal{A}$.

Notice that, the product of these parameter-dependent matrices in Equation (1) is a matrix of size $r_0 \times r_d$, this indicates the boundary condition of $r_0 = r_d = 1$. Moreover, since $r_0 = r_d = 1$, TTD can also be visually represented by a graph called linear tensor network, as shown in Figure 3. There are two different types of nodes in this graphical representation. The rectangles are the tensor cores with the spatial indices (i_k from the original tensor) and auxiliary indices α_k. The circles are indeed links to connect two adjacent tensor cores with same auxiliary index α_k. This means that these two tensor cores are contracted together, and further being contracted with the following tensor cores to form the final d-dimensional tensor.

The most important step of TTD is how to extract these tensor cores from the original high order tensor. In this work, we focus on the classical TT-SVD approach, which computes such TTD using d-sequential SVDs of auxiliary matrices.

III. SVD Algorithm Adaptation

As introduced above, the computation of TTD is dominated by sequential SVD decompositions over the temporary auxiliary matrices. Therefore, reducing the SVD latency is vital for accelerating TTD algorithm. In this section, we present our observations of these auxiliary matrices that motivate us to design an adapted SVD decomposition that directly reduces the overall latency from algorithmic level. The proposed SVD also enables more efficient hardware implementation which will be demonstrated in Section IV&V.

Algorithm 2 Adapted SVD Algorithm

Require: 2-dimensional matrix $A_{m\times n}$ where $n \gg m$, total iteration number N.
Ensure: Approximate decomposition of $A = U \times SV^T$ with an orthonormal matrix U and orthgonal matrix SV^T.
1: {Initialization} $i = 0$, $B = AA^T$, $B \in \mathbb{R}^{m\times m}$, $Q_H = I_n$
2: Compute Arnoldi Iteration: $Q_k^T H Q_k = B$, where H is a symmetric and tridiagonal matrix since B is symmetric.
3: **while** $i \leq N$ **do**
4: $d = H[n-1, n-1]$
5: $H = H - dI_n$
6: $Q_i R_i = qr(H)$
7: $H = R_i Q_i + dI_n$, H stays symmetric and tridiagonal.
8: $Q_H = Q_H Q_i$
9: **end while**
10: $U = Q_k Q_H$
11: $SV^T = U^T A$
12: Return $U,\ SV^T$

First, consider a given matrix $A \in \mathbb{R}^{m\times n}$, the singular value decomposition of A is defined by:

$$A = USV^T \tag{2}$$

where U and V are orthogonal matrices of $m \times r$ and $n \times r$, as $UU^T = I_m$, $VV^T = I_n$ (I_m is the identity matrix of size $m \times m$, same for I_n). S is a diagonal matrix such that $S = diag(\sigma_1, \sigma_2, \cdots, \sigma_r)$, σ_k are the singular values of A. Among the existing numerical methods to compute SVD decomposition, one-sided Jacobi [16] is considered to be the most hardware friendly because of its fast convergence rate and good algorithm parallelism. The basic idea is to zero out off-diagonal elements using a series of orthogonal transformations between each pair of the matrix columns, and repeat this procedure for multiple iterations until convergence. Prior works [17]–[19] have proposed several customized accelerators for SVD using Jacobi method.

However, directly using the Jacobi method for TT-SVD is inefficient. To be more specific, in TTD, the auxiliary matrices to be decomposed are both large and unbalanced (i.e., one dimension is significantly longer than the other). For example, the first matrix to be decomposed is the first-mode unfolding matrix whose size is $I_1 \times I_2 I_3 \cdots I_N$. While Jacobi method requires multiple iterations to converge, within each iteration, we need to load and update the whole matrix. For tensor data, the size of such matrix can easily exceeds the capacity of caches (in CPU and GPU) and on-chip buffers (in customized accelerators). As a consequence, significant latency and energy consumption will be caused by excessive data access from the main memory module (e.g., DRAM). Moreover, designing multiple levels of memory hierarchy is also ineffective since there are no data reuse between different iterations of the Jacobi method.

Therefore, on the base of our observations and analysis, we propose an adapted SVD algorithm targeted for the large-

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

scale unbalanced matrix. As shown in Alg. 2, the modified approach can be divided into three phases. Given an auxiliary matrix A, we first compute a matrix transpose multiplication $B = A \times A^T$. As a result, B is an $m \times m$ symmetric matrix whose size is much smaller compared with A. Then, to obtain $A = USV^T$, we can instead compute the eigenvalue decomposition (EVD) of B. In our approach, we use the Arnoldi method [20] followed by the shifted QR algorithm to compute the EVD result. Applying the Arnoldi method on matrix B gives us an orthonormal basis Q_k of B's Krylov subspace, and a symmetric tridiagonal matrix H where $H = Q_k B Q_k^T$. We then apply the shifted QR algorithm to obtain the eigenvectors of matrix H, which we denote as Q_H. Note that, Q_H is called the Ritz vectors of B that can be used to compute the eigenvectors of B. Finally, after we obtain the eigenvectors, which are indeed the left singular vectors of matrix A, we can compute SV^T with $SV^T = U^T A$.

Mathematically, the proposed SVD provides same results as typical Jacobi-based SVD. However, when dealing with large unbalanced matrix, it has following advantages: (1) We avoid constantly loading and updating (writing) matrix A. Such operations are inefficient as matrix A is often stored in high-cost memory, e.g., off-chip DRAM. (2) Both the Arnoldi method and the QR algorithm can be implemented based on modified Gram-Schmidt orthogonalization (MGS), which can be efficiently mapped onto our proposed architecture in Section IV. (3) The symmetric property of B greatly simplifies the process of Arnoldi method and QR algorithm. For Arnoldi method, when the input matrix is symmetric, the output matrix H will automatically become symmetric and tridiagonal. Therefore, we can directly skip the computations regarding the zero elements in H (output sparsity). For QR algorithm, since the input matrix H is symmetric and tridiagonal, the complexity of each iteration is significantly reduced. More importantly, matrix H will stay symmetric and tridiagonal after each iteration, which means such characteristic will benefit every QR iteration through out the whole process.

A. SVD Algorithm Evaluation

TABLE I: Computation complexity & external memory access

Method	Computation Complexity	Memory Access
Jacobi	$iter1 \times O(m^2 n)$	$iter1 \times O(m^2 n)$
Ours	$O(m^2 n) + iter2 \times O(m^2)$	$O(mn)$

Table I lists the algorithm complexity and memory consumption between the proposed SVD and standard one-sided Jacobi SVD, where m, n are the matrix dimensions and $iter1, iter2$ denote the number of iterations performed in each approach. As we can see from Table I, our approach is more computational efficient when $iter2$ is comparable or smaller than $iter1$. We will demonstrate in Section III.B and Section VII that, when we seek for a low-rank output tensor-train(high compression ratio), which is normally the case of using TTD, then we only need approximate SVD results. Therefore, $iter2$ would be close to the number of iterations in Jacobi method, which makes the above analysis reasonable.

Moreover, as for memory footprint, the Jacobi method updates the whole matrix(A) within each iteration. Since A is of large-scale, it needs to be stored in DRAM rather than on-chip SRAM. Thus, constant access to matrix A will suffer from lower off-chip memory bandwidth and cost higher energy consumption. On the contrary, the proposed approach mainly operates on matrix B, which is much smaller and can be stored on-chip. Although the on-chip data movement will be more frequent, we prove in Section VI with our experiments that this benefits the overall performance while lowering the energy consumption.

The final advantage of using the adapted SVD algorithm is its impacts on the hardware design. By enabling symmetricity and sparsity in matrix B, we open more hardware possibilities to reduce the decomposition latency with a dedicated accelerator. These properties are hard to be adopted by the conventional computing platforms like CPU and GPU.

B. Influence on TTD Accuracy

With the less computation complexity and memory footprints for processing the targeted large-scale unbalanced matrix, we further demonstrate the decomposition accuracy when applying the proposed SVD in TTD decomposition. To do so, we implement a customized TTD based on our adapted SVD algorithm, and compare it with the standard TTD function integrated in *tntorch* [21]. The accuracy of our proposed SVD algorithm can be controlled by the iteration number N, which further reflects on the end to end accuracy of the Tensor-Train decomposition. In our experiments, we set N to be 5, 10 and 15. In contrast, the Jacobi-based-SVD typically requires more rounds of iterations (around 30 or even higher) with longer per iteration latency. We use randomly generated tensor data as the input and decompose it using different TTD implementations. Then, we contract the tensor cores together to reconstruct the tensor data. The error between the reconstructed tensor and the original tensor is defined by the following equation, where $\mathcal{A}'$ is the reconstructed tensor, $\mathcal{A}$ is the original tensor and $norm$ is the Euclidean norm:

$$error = \frac{norm(\mathcal{A}' - \mathcal{A})}{norm(\mathcal{A})}. \tag{3}$$

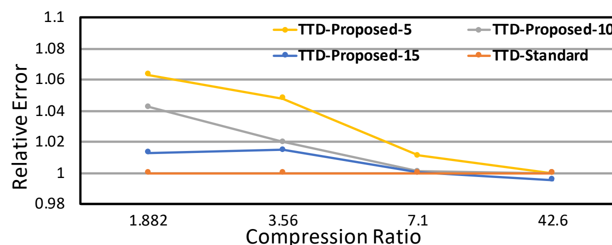


Fig. 4: Accuracy comparison between the proposed TTD and the standard TTD.

We compare the accuracy of the proposed TTD and the standard TTD by dividing their error values. Therefore, the higher the number is, the larger error it has compared with the standard TTD. We show the comparison in Figure 4, where y-axis is the relative error and x-axis is the compression ratio.

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

A higher compression ratio means that the TT ranks are set to be lower to get smaller tensor cores. As we can see from the results, the proposed approach can achieve comparable accuracy with the standard TTD when the TT ranks are low (i.e., high compression ratio). When the compression ratio is up to 42.6, we can achieve nearly the same accuracy compared with the standard TTD under all the three settings of N. For $N = 15$, the relative error is even smaller than 1, indicating that it even has less error than the standard TTD. As the compression ratio decreases, the relative error will increase, which implies that the proposed TTD is less accurate than the standard TTD when the TT ranks are high. Fortunately, such case rarely happens in practice because TTD is designed to be used to compress high order tensor with low TT ranks for good compression ratios. For example, when being used to compress weight matrices in deep neural networks(DNNs), prior work [22], [23] achieve acceptable accuracy loss with the compression ratio to be 82.87× for CNNs on the CIFAR-10 [24] dataset.

C. Discussion

1) Numerical Stability: In order to increase the efficacy of computing the QR factorization, we use the Gram-Schmidt orthogonalization. However, the classical GS method can be numerically unstable due to the rounding error when processing with finite precision. We solve this problem by using the stabilized modified Gram-Schmidt method (MGS). Specifically, traditional GS method computes a new vector by subtracting it with all of its projection vectors based on the existing unit vectors. In the modified GS method, for a new given vector, we start with eliminating the projection vector of the first unit vector to get a new candidate vector. The second projection vector to be eliminated is computed based on the new candidate vector instead of directly using the original vector. It is proved that this approach gives the same result as the original formula in exact arithmetic and introduces significantly smaller errors in finite-precision arithmetic.

2) Novelty: The problem of numerically computing singular value decomposition (SVD) has already been well studied. However, in terms of implementing TTD, prior work have not proposed the idea of transferring the large unbalanced SVD problem to a symmetric small eigenvalue decomposition (EVD) problem. Moreover, using the Arnoldi method and the shifted QR algorithm to approximate the EVD result is normally not suggested, as the number of iterations grows rapidly when requiring a particularly high decomposition accuracy. In our work, we take the advantage of the low-rank property of TTD to explore more efficient implementations while maintaining overall decomposition accuracy. Such low-rank property comes from our observations across different practical applications where TTD is adopted, like CNN/RNN, image compression, and quantum analysis. In these applications, TTD is used to achieve very high compression ratio without influencing much on the overall application accuracy. Thus, this high-compression condition ensures the low-rank settings for our previous analysis. Finally, as we mentioned above, the objective of using our proposed SVD is to eventually benefit the hardware implementation and reduce the execution complexity, which will be further demonstrated in Section IV-VI.

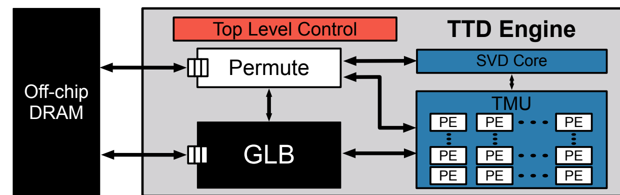


Fig. 5: TTD Engine architecture overview.

IV. TTD Engine Overview

Based on the analysis above, we present the overview of our TTD Engine in this section. We focus on addressing two key challenges during the hardware design. On one hand, the hardware should efficiently implement the proposed TTD algorithm, providing acceptable performance speedup and efficiency improvement. We call this, the **Specialization** of the hardware. On the other hand, it should also have the flexibility to support general matrix/tensor and even tensor-train operations, so that it can be further adopted to accelerate different applications using the tensor-train processing scheme. We call this, the **Generality** of the hardware.

With these two design objectives, we show the top-level architecture of TTD Engine in Figure 5. The off-chip DRAM stores the original tensor data that are unable to be fitted on-chip. Therefore, the accelerator communicates with the external DRAM through a bidirectional data bus and stores intermediate data in the Global Buffer (GLB) for on-chip data reuse. The computing resources are mainly organized into two modules, the Tensor Multiplication Unit (TMU) and the SVD Core. Both of the two modules adopt a spatial 2D processing element (PE) array architecture. TMU efficiently handles regular matrix/tensor operations, including matrix-matrix, matrix-tensor multiplications and so forth. TMU and SVD core can work together to execute the modified TTD algorithm proposed in Section III. The Permute Unit reshapes the auxiliary matrices to be decomposed between each TTD iteration.

As illustrated in Figure 5, TTD Engine applies the Spatial Architecture (SA) design of domain specific accelerators (DSAs) [25]. The SA-style DSAs exploit high compute parallelism by direct communication between the PE array. Besides, the hierarchical memory organization from GLB to the PE's local memory further improves data reuse, achieving higher bandwidth utilization and energy efficiency. Therefore, SAs are widely used to accelerate deep learning algorithms like Convolutional Neural Networks (CNNs) [26], [27], Recurrent Neural Networks (RNNs) [28], [29] and Personalized recommendations [30]. In TTD Engine, while such generality is well preserved, we further add specialized Permute Unit and SVD Core to achieve the efficient execution of Tensor-train Decomposition.

In the rest of this section and Secion V, we focus on illustrating the specialized architecture and dataflow design of TTD Engine when processing Tensor-train Decomposition

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

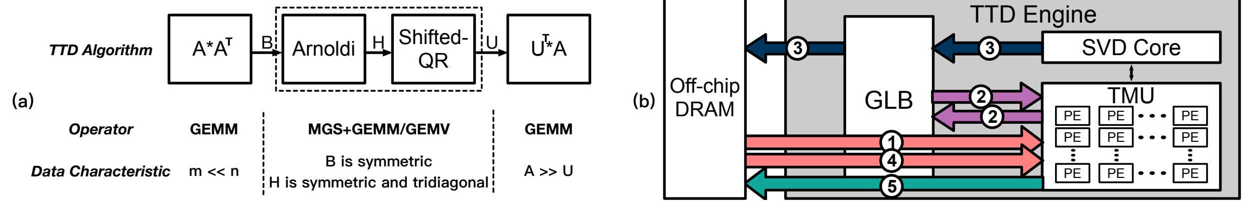


Fig. 6: TTD algorithm abstraction and TTD Engine execution dataflow. (a) shows the key operators and data characteristics; (b) illustrates the data movement with arrow indicating the direction, color indicating specific data, and the number indicating the order.

algorithm. We will dive more into the generality with real world application demos presented in Section VII.

A. Overall Dataflow

We use Figure 6 to illustrate how we implement the proposed TTD algorithm. To do so, we first provide an abstraction of the algorithm to extract the key operators as shown in Figure 6(a). We also mark the special characteristics of these operators' input/output data, which can further simplify the hardware design. Corresponding to the key operators, Figure 6(b) demonstrates the data movement in TTD Engine.

In each decomposition iteration, we first compute $B_i = A_{(i)}A_{(i)}^T$, where ① $A_{(i)}$ is the current auxiliary matrix. This step is essentially matrix-matrix multiplication, but the input matrix has unbalanced size where its width n is much longer than its height m. To execute this step on TTD Engine, we load $A_{(i)}$ patch by patch from off-chip memory to GLB. Tensor Multiplication Unit (TMU) and SVD Core work together as a larger PE array to compute matrix ② B_i.

During the EVD decomposition of matrix B_i, both the Arnoldi iteration and the Shifted QR algorithm can be represented by a two-step process: column orthogonalization & data update. While the former step is realized using modified Gram-Schmidt (MGS), the latter step is nothing but matrix-vector/matrix-matrix multiplication. In TTD Engine, we use SVD core to perform MGS, the resulting matrix will be generated column by column, and will be sent to TMU immediately to perform data update. We use such decoupled design of TMU and SVD core to pipeline the two-step EVD while increasing local data reuse.

After we obtain the left singular matrix of B_i, which we denote as ③ U_i, we can directly permute and output U_i as the extracted tensor core. We then load matrix ④ $A_{(i)}$ again to compute ⑤ $S_iV_i^T = U_i^TA_{(i)}$. The result will be reshaped by the permute unit and sent out as the auxiliary matrix to be decomposed in the next TTD iteration.

As the iteration continues, the matrix to be decomposed would become smaller and more balanced. TTD Engine can also support these cases by storing the matrix completely in GLB and using the Jacobi method for SVD decomposition.

V. TTD Engine Architecture

In this section, we present the detailed architectures of different modules in TTD Engine. We also discuss how we take advantages of the data's special characteristics to efficiently map the algorithm onto hardware.

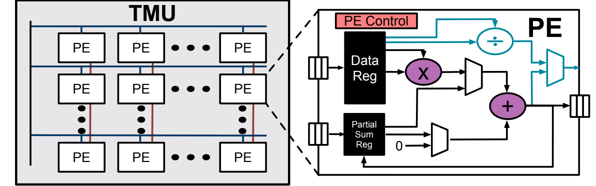


Fig. 7: TMU architecture. The blue-colored logic only exists in PEs inside SVD Core.

A. Tensor Multiplication Unit (TMU)

Figure 7 presents the 2D PE architecture of TMU. Each PE could communicate with its neighbors and also the GLB through an NoC. FIFOs are used at the I/O interface of each PE to balance the data movement between the NoC and the computation. The PE consists of a MAC unit for Multiply-and-Accumulate (MAC) operation, local buffers for matrix and partial sum data, and the PE's local control logic. For normal matrix multiplication, TMU can work as a systolic array to provide high computation throughput with simplified control flow. However, apart from the general matrix multiplication, we still need to consider the following special cases during the computation of TTD.

1) Large-scale matrix × small-scale matrix: Each of the TTD iteration ends up with a matrix-matrix multiplication between the current auxiliary matrix $A_{(i)}$ and the left singular matrix U_i. In most cases, $A_{(i)}$ is much larger than U_i and is stored off-chip. Therefore, in order to reduce the high-cost memory access, we keep a patch of $A_{(i)}$ stationary in TMU and load the corresponded blocks of U_i from the GLB. After finishing all the computations associated with the current patch, we load another patch of $A_{(i)}$ and repeat the process. In this way, although we need to traverse matrix U_i several times in GLB, the large-scale matrix $A_{(i)}$ is loaded only one time from the off-chip DRAM. Therefore, the high-cost off-chip memory access is replaced with low-cost local memory access.

2) Large-scale matrix transpose multiplication: The first step of the proposed SVD algorithm is to perform a matrix transpose multiplication using the unbalanced matrix $A_{(i)}$. Since we already know that the result will be a symmetric matrix, we can save almost half of the redundant computations by only calculating the upper triangular part of the output matrix. For illustrative purpose, we use Figure 8 to demonstrate the matrix transpose multiplication for a matrix

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

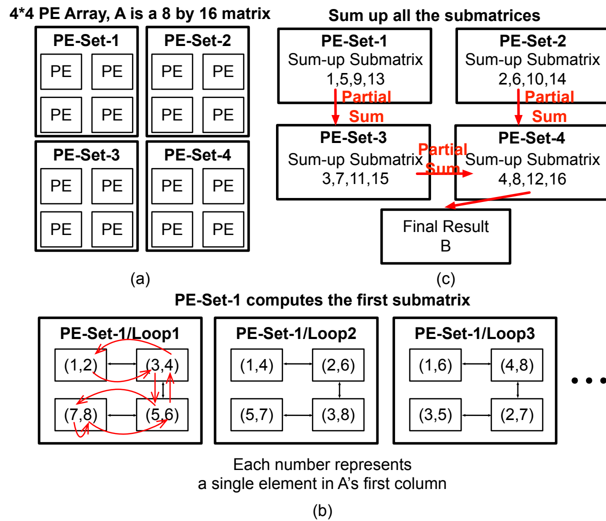


Fig. 8: Using TMU to compute large-scale matrix multiplication: (a) Several PEs are grouped together to compute a specific submatrix; (b) The data movement between different PEs after each computation loop; (c) Summing up all the submatrices across the PE array.

$A = \{a_1, a_2, ...a_{16}\} \in \mathbb{R}^{8\times 16}$ with a 4×4 TMU PE array. The resultant matrix B can be considered as the sum of 16 submatrices where each submatrix $B_i = a_i \times a_i^T$. Therefore, we can group up 4 PEs as a PE set to compute a specific submatrix. The reasons for us to choose outer product to compute B are of two folds. First, using outer product only requires a single traversal through the original matrix to finish the computation. This is especially beneficial as matrix A is stored in high-cost off-chip DRAM. Second, in normal cases, buffering the output submatrices for accumulation can be expensive, but since the columns of matrix A are short, the submatrices computed by these columns become much smaller and easier to buffer on chip.

As shown in Figure 8(b), inside the PE set, each PE is distributed with two elements of a specific column a_i. During each loop, the PE multiplies these two elements together to generate a single element in the submatrix. After each loop, different PEs from the same PE set will exchange data between each other. The data exchanging order is predetermined according to the column length. In this example, the red arrow indicates the data movement direction after each multiplication. Such order avoids all the redundant computations. Finally, all the submatrices are accumulated together to get the result. Note that, if the column is too long, each PE may contain multiple elements of the column. In such case, the PE will generate a small block of the output submatrix after each computation loop.

3) Tensor core contraction: TTD Engine is also designed to be able to perform tensor core contraction to recover the original tensor data. For tensor core contraction, each time we contract the last mode of the current tensor with the first mode of the next tensor core. Therefore, we only need to permute the tensor core and load the existing tensor in its original order. We assume the tensors are always stored by incrementing the mode-1 index, then the second mode index, and so on. To be more specific, suppose we have finished contracting the first m tensor cores which gives us tensor $\hat{G} \in \mathbb{R}^{I_1\times I_2\cdots\times I_m\times r_m}$. The next step is to contract $G_{m+1} \in \mathbb{R}^{r_m\times I_{m+1}\times r_{m+1}}$ with $\hat{G}$. Thus, we first permute G_{m+1} using TTD Engine's permute unit and store it in the GLB as $\hat{G}_{m+1} = r_m \times I_{m+1}r_{m+1}$. Then, we can treat both of them as matrices and perform matrix-matrix multiplication.

B. SVD Core

As introduced previously, the process of the adapted SVD algorithm can be represented by MGS and data update. Data update is essentially matrix-vector/matrix-matrix multiplication that can be efficiently mapped onto TMU. As for MGS, there are two problems need to be addressed. First, the orthogonalization between two columns requires the division operation. Thus, as shown in Figure 7, PEs inside the SVD core are further facilitated with dividers for the operation. Second, the MGS algorithm consists of multiple column orthogonalizations between different pairs of columns that have inter data dependency. Thus, it is important to design a mapping strategy that can maximize the computation resource utilization without breaking the data dependency.

Here we use Figure 9 to demonstrate the data dependency and mapping strategy. As shown in Figure 9, each parenthesis indicates an orthogonalization operation between two columns. For instance, $(2, 1)$ means to orthogonalize column 2 over the reference column 1. Therefore, only column 2 will be updated after this operation. According to MGS, there are two types of data dependency during the computation.

The first type is that, we cannot use a column as a reference column until it is finalized. For example, column 3 needs to be orthogonalized with column 2 and column 1. Therefore, we need to perform $(3, 1)$ and $(3, 2)$ before we can perform operations that use column 3 as the reference column, e.g., $(4, 3), (5, 3)\cdots$. The first type is marked by red arrows in Figure 9. The second type of data dependency is that, we cannot simultaneously orthogonalize the same column with two different reference columns. For example, $(3, 1)$ and $(3, 2)$ cannot be executed at the same time. The second type is represented by blue lines in Figure 9. It also shows that each time we cannot choose more than one operations from the same line.

Based on the analyses above, we propose the mapping of the MGS algorithm as shown in Figure 9. The idea is to choose as many operations as possible from the same vertical line. When we reach the end of one line and have to move across another line, we move to its adjacent line and start from the top. Both two types of data dependencies are avoided to the utmost extent using this mapping. In this example, suppose the SVD core can at most orthogonalize 3 pairs of columns in the same cycle. Then, we first choose $(2, 1), (3, 1), (4, 1)$, and then $(5, 1), (3, 2), (4, 2)$. Due to the second type of data dependency, we can only execute $(5, 2), (4, 3)$ at the third cycle, $(5, 3)$ in the fourth cycle, and $(5, 4)$ in the final cycle. Thus, it takes 5 cycles to finish the MGS process. If we

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

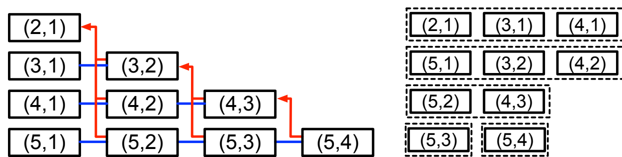


Fig. 9: Mapping the GS orthogonalization onto SVD Core. Operations in the same dotted rectangle are executed simultaneously.

increase the computation resources to support 4 pairs of columns. We can reach a maximum throughput of 4 cycles to finish the MGS process. However, the resource utilization will be lower. Therefore, we design a lightweight SVD core that can achieve near-optimal performance with better resource utilization. Also, choosing most operations from the same vertical line increases the data reuse of the reference column that can further improve energy efficiency.

C. Permute Unit

The tensor permute unit is located between the GLB and external DRAM to reshape the input/output tensor/matrix data. It is used in the below cases: (1) During tensor core contraction, we use the permute unit to reshape the small tensor core; (2) After we compute the left singular matrix U, we permute it into a 3-mode tensor core. (3) After we multiply the current auxiliary matrix A with U^T, we need to permute the result to be the input matrix for the next TTD iteration.

VI. Evaluation

A. Evaluation Methodology

Evaluation Platform. The proposed TTD Engine is implemented in RTL and synthesized in Synopsys Design Compiler with TSMC 45nm standard cell library to obtain the area and power estimation. The timing and energy of on-chip memory are simulated with CACTI [31]. We also develop a cycle-accurate simulator based on RTL implementations to evaluate the performance of TTD Engine.

Baselines. We compare our TTD Engine with the state-of-the-art CPU and GPU. The CPU baseline is an Intel Core i7 8700 processor (14nm), which has 12 SMT cores running at 3.2GHz and 12MB LLC. For GPU comparison, we use NVIDIA Titan V GPU (12nm) that is equipped with 5120 tensor cores and 12GB HBM2. We choose the Tensor Toolbox [32] as the software implementation on CPU and TnTorch [21] on GPU.

Benchmarks. We use synthetic data for the performance evaluation, with tensor sizes of 64KB, 4MB, 256MB, 1GB, and 8GB. The synthetic data are generated with built-in functions in each open-source library. For example, in TnTorch, we use torch.rand()/torch.randn() to generate the tensor data. For the decomposition speed comparison, we don't care about the actual value and distribution of the synthetic data. But for accuracy comparison, we keep the data identical across different implementations. We also set different decomposition parameters to examine how the performance is sensitive to the ranks of tensors. It is worth mentioning that, using synthetic data does not affect the generality of the experiments at all. Instead, it is because of the flexibility of synthetic benchmarks that enables us to evaluate TTD Engine's performance over various input patterns, including the cases that are frequently or rarely encountered in practical applications.

B. TTD Engine Summary

Table II presents the summary of the TTD Engine specifications. We use 16-bit fixed point arithmetics to implement our design. As listed in the table, with 16×16 PEs in TMU and 8×8 PEs in SVD Core running at 400MHz, our accelerator yields a peak performance of 128GMAC/s. Each PE has a 128B register, therefore, TMU and SVD core together have a 40KB of SRAM capacity. The global SRAM buffer is 1MB. Therefore, the total on-chip memory capacity is 1064KB. We show the area and power breakdown in Figure 10, from which we can see that the power is dominated by fixed-point operators as a fraction of 76%, while the total area is dominated by on-chip SRAM with a ratio of 62%.

TABLE II: TTD configuration summary.

Item	Specification
Technology	TSMC 45nm GP standard VT
Total Area	6.94mm^2
Total Power	2.89W
Number of PEs	256 (TMU) + 64 (SVD Core)
Global Buffer	1MB (SRAM)
Register per PE	128B
Arithmetic Precision	16-bit fiexed-point
Frequency	400MHz
Peak Performance	128GMAC/s

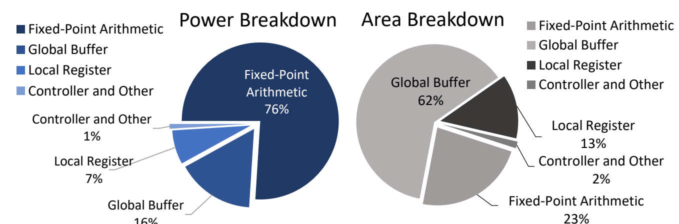


Fig. 10: Power and area breakdown.

C. Overall Performance and Energy Efficiency

We compare the performance of our TTD Engine with CPU and GPU over synthetic data that have different sizes ranging from 64KB to 8GB. For each fixed input tensor size, we manually set the targeting output tensor ranks to 3 different levels to adjust the compression ratio. For example, for a single 4MB tensor, a low-rank decomposition means we generate a low-rank tensor-train from the original tensor data, which indicates a higher compression ratio with larger decomposition error compared with a high-rank result. Usually in real world applications, the rank matches with the low-rank and medium-rank cases in our experiments.

We first evaluate the decomposition accuracy among different implementations. The accuracy is measured with the absolute reconstruction error as expressed by equation (3),

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Section III. Figure 11 shows the reconstruction error under different rank-levels averaging over all sizes of the input tensors. As we can see from the figure, in all three different rank levels referring to different compression ratios, the reconstruction accuracy scores among the implementations are comparable. Also, the proposed approach performs closer to (or even better than) the standard TTD when we are expecting a low-rank output tensor-train.

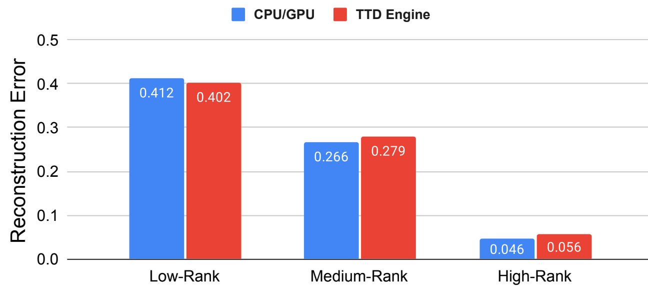


Fig. 11: Average reconstruction error of CPU/GPU and TTD Engine under different rank-levels(compression ratio).

Then, we compare the decomposition speed of TTD Engine with CPU and GPU. As we can see from the results in Figure 12(a), TTD Engine significantly outperforms CPU's performance. On average, it is $14.9\times \sim 36.9\times$ faster than the CPU implementation. Compared with GPU, TTD Engine can achieve speedup on benchmarks that are smaller than 1GB. If the input tensor size exceeds this limit, the speedup over GPU decreases. The reason is because when the tensor size keeps growing, the whole computation process tends to be dominated by the matrix transpose multiplication of the first few TTD iterations. For extremely large-scale matrix multiplication, TTD Engine is limited by computation resource and memory bandwidth, which dilutes the benefit of the proposed algorithm and dataflow optimization. However, in real cases, datasets are usually large for its number of samples rather than the size of each sample. Therefore, typically we do not need to consider a single tensor with a size of 8GB or even larger. Besides, the good scalability of TTD Engine makes it efficient to improve the performance by increasing the on-chip resources.

Also, for a given tensor, the lower the needed TT-ranks are, the higher speedup TTD Engine can achieve. This is because TTD Engine computes the singular vectors in the order of the singular values, and stops the computation as soon as the first r vectors are obtained. Whereas a typical truncated-SVD computes the complete SVD first, and then choose r vectors to output. This makes TTD Engine particularly suited for low-rank decompositions of a tensor.

Finally, we compare the energy efficiency of TTD Engine with CPU and GPU implementations over the same benchmarks. As shown in Figure 13, TTD Engine consumes, on average, 47.2x and 231.6x less energy than CPU and GPU, respectively. Such improvement is mainly gained from two aspects. First, we exploit low-cost data movement through the algorithm-hardware co-design while reducing high-cost external memory access. Second, we exploit the data sparsity and symmetricity during the computation process that helps to reduce both compute and memory consumption. Besides, when dealing with larger tensor, the energy efficiency improvements over GPU tends to be lower and less separable between different bars. Similar with the above analysis, when the decomposition process is more dominated by the matrix-transpose multiplication, the savings that come from adopting symmetricity and sparsity during the computation contributes less to the overall improvements.

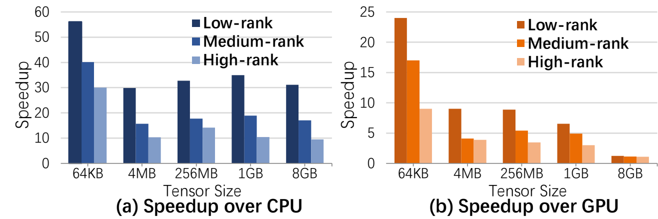


Fig. 12: Speedup of TTD Engine over CPU and GPU.

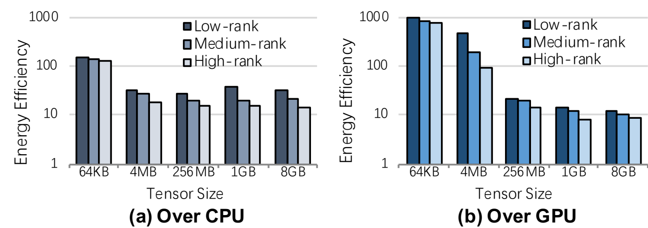


Fig. 13: Energy reduction of TTD Engine over CPU and GPU.

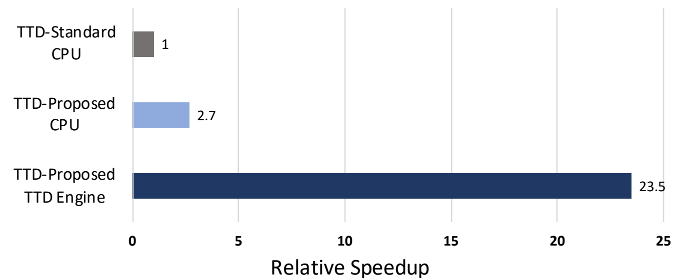


Fig. 14: Performance contribution breakdown

D. Benefits Breakdown

In Section VI.C, we compare the overall performance between TTD Engine and CPU/GPU implementations. Here, we further decouple the contribution of the proposed SVD algorithm and accelerator design to separately demonstrate their benefits. We first run the proposed TTD and standard TTD algorithm on the same CPU to collect the decomposition time. Then, the proposed TTD algorithm is executed on TTD Engine to be compared with the other two cases. As we can see from Figure 14, by using the adapted SVD algorithm alone, we are able to achieve, on average, $2.7\times$ speedup over the original CPU baseline. This speedup mainly comes from the computation reduction brought by the algorithm modification with a small number of iterations($iter2$). However, without dedicated architecture design and specialized dataflow, the sparsity and symmetricity of the matrices are hard to be utilized to benefit the overall performance. This introduces unnecessary computations which dilutes the final speedup. Thus, when TTD Engine is finally used, it further brings another $8.7\times$ times speedup over the TTD-proposed-CPU and

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

provides a final 23.5× speedup over the TTD-standard-CPU baseline.

VII. Application Demo

Through Section III to Section VI, we demonstrated the effectiveness of TTD Engine when processing TTD with the proposed algorithm-hardware co-design approach. In this section, we will further illustrate how TTD Engine can be extended to accelerate different applications.

The first application we choose is medical image compression. TTD is now being used in a wide range of disciplines, including EDA design and simulation, machine learning, medical imaging, etc. One of the direct benefits of TTD is that, it saves considerable amount of memory space for storing the big data required by these applications. Moreover, with the decomposed results, many previous complex computations could be executed much faster and easier. To demonstrate such effectiveness, we first choose medical image compression as an example demo. We use the proposed accelerator to generate TTD results for a real-world magnetic resonance imaging (MRI) image benchmark, which significantly reduces the size of the benchmark while preserving good image quality.

In addition, we further illustrate the benefits of using TT-format data by proposing a TT-based convolution scheme. The proposed TT-convolution algorithm directly uses the TT-format data as input and performs convolution operations based on the convolution kernel. With the decomposed TT-format data, we can greatly reduce the overall computational complexity as well as memory consumption.

A. Medical Image Compression

We take medical imaging application as our first example. MRI is a safe and painless technique and is therefore widely used to generate detailed images of the brain and the brain stem. For general research purpose, numerous brain images are required and the data can easily reach to several gigabytes and even terabytes. Thus, it is very memory consuming to store the dataset. In our experiment, we choose a typical brain image dataset that contains 766 brain images of size 512×512 and is in total about 420MB large. The image dataset is compressed with TTD Engine and other open-source libraries running on CPU and GPU. We compare the decomposition performance between different architectures in terms of compression time and reconstruction error under a specific compression ratio of 7.1×.

The experimental results are shown in Table III. As for decomposition time, TTD Engine achieves 20.4× and 13.9× speedup over CPU and GPU, respectively. The speedup is close to the 4MB bar in Figure 12 even though the dataset is 420MB. This is because we compress each 512×512 image (1MB) separately, which gives us a higher performance speedup compared with compressing the dataset as a large single tensor. As for the reconstruction error presented in Table III, the proposed TTD achieves slightly better decomposition accuracy compared with standard TTD library given the same targeting compression ratio. This matches the conclusion we presented in Section III.A, that the proposed approach is able to generate comparable result when the compression ratio is not very low. Here, we also use equation (3) to measure the absolute reconstruction error.

Finally, to provide an intuitive comparison, four pairs of original images and reconstructed images are randomly selected from the dataset and presented in Figure 15.

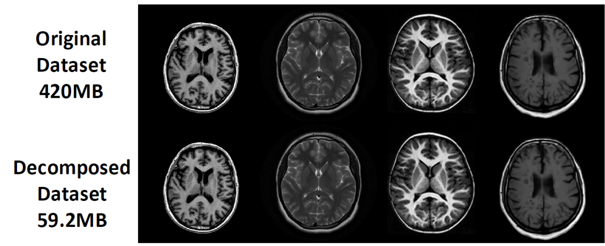


Fig. 15: Comparison between original and decomposed MRI images.

TABLE III: Decomposition Performance&Accuracy

Hardware	Comp. Ratio	Speedup over CPU	Error
CPU	7.1×	1x	0.158
GPU	7.1×	1.47x	0.158
TTD Engine	7.1×	20.40x	0.157

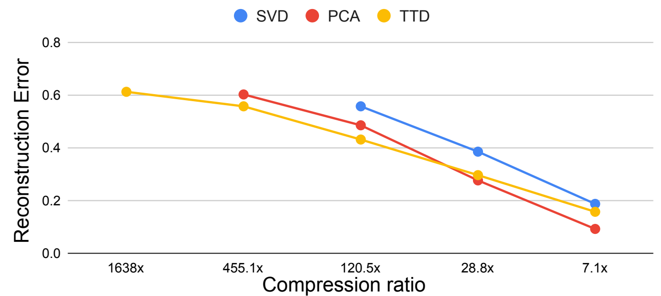


Fig. 16: Comparing TTD with PCA and SVD for MRI image compression.

Finally, TTD is compared with other compression techniques like PCA and SVD under this specific application scenario. We sweep through different compression factors and compare the reconstruction error between different methods. Figure 16 delivers the results and implies several conclusions. Firstly, TTD achieves lower absolute error compared with SVD in all the test cases. Secondly, when being compared with PCA, TTD tends to perform better in the cases with larger compression ratios, while doing worse in the low-compression cases. This indicates its advantage for providing highly compressed data with rather low error, which matches our previous analysis. Finally, PCA and SVD cannot deliver highly compressed images. As shown in the figure, no matter how we reduce the SVD/PCA parameters, SVD cannot deliver the two highest compressed cases and PCA cannot reach the compression ratio as high as 1638×. This is because SVD and PCA are pure 2D data processing techniques and are limited by the dimensions of the original image. On the contrary, TTD is able to first consider the 2D image (matrix) as a

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

high-dimension tensor and then perform decomposition on all of its dimensions to achieve aggressive compression. In real applications, we can flexibly choose the desired compression technique based on different requirements for accuracy and compression ratio.

B. One-Dimensional Convolution

As aforementioned, there is a gap between decomposing a tensor and using the decomposed data to develop TT-based algorithms for practical applications. Previous work [14] has proved that basic TT operations like TT-Addition, TT-Multiplication, TT-GEMV, and scalar product have less complexity than directly operating on original large-scale tensor data. Moving forward, for the first time, we introduce how to use TTD Engine as the base architecture to perform TT-format data convolution on the decomposed data. We believe data convolution is a promising example to demonstrate the potential and benefit of using TTD for more complicated operations and applications. On one hand, element-wise operations are commonly used but rarely studied for TT-format data. On the other hand, multidimensional convolution stands at the core of many important applications including image processing, machine learning, and EDA.

For illustrative purpose, we first consider an 1D data convolution with an 1×3 convolution window sliding over an 8-element vector $\mathbf{v}$. The vector is reshaped into a $2 \times 2 \times 2$ tensor $\mathcal{V}$ and then represented with 3 TT-cores, G_1, G_2, G_3. As shown in Figure 17, if we reverse core G_3's second dimension by exchanging the purple column with the pink column, the order of tensor $\mathcal{V}$'s third dimension is also reversed. This is further equivalent to switching every two consecutive elements in $\mathbf{v}$. Similarly, if we reverse core G_2's second dimension, it is equivalent to switching every two consecutive **pairs** of elements in $\mathbf{v}$.

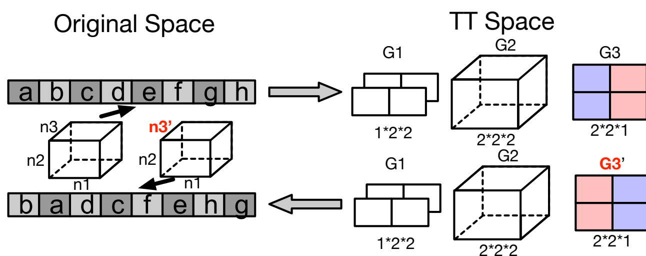


Fig. 17: Demonstration of TT data and its characteristics.

Therefore, we demonstrate the process of TT-based 1D convolution in Figure 18. Without loss of generality, we assume all the weights to be equal to one. We already know that, we can operate on a specific dimension by modifying its corresponding tensor core. Thus, we can represent the final convolution result by the sum of several sub-vectors. The principle is to ensure that the TT-format of each sub-vector can be efficiently obtained from the original tensor-train format data.

As shown in Figure 18, for this specific 1D convolution example, the final result is represented with the sum of 3

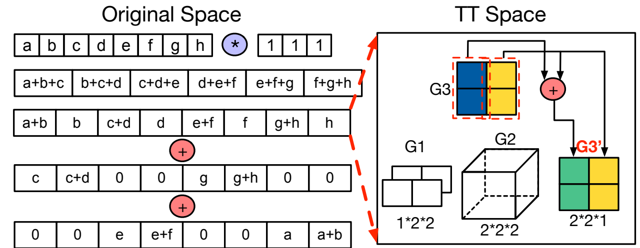


Fig. 18: 1D Convolution in TT-format.

sub-vectors. Taking the first one as an example, in every consecutive pair, the first element is the sum of the original two elements, and we keep the second element unchanged. Therefore, to get the TT-format of this sub-vector, we can simply sum up G_3's second dimension to form up a new column and replace the first one, while leaving the second column the same as before. The other sub-vectors require similar operations. After this, we add the 3 tensor-train format sub-vectors to obtain the final convolution result. Note that, TT-Addition requires no computations but to merge the corresponding tensor cores together.

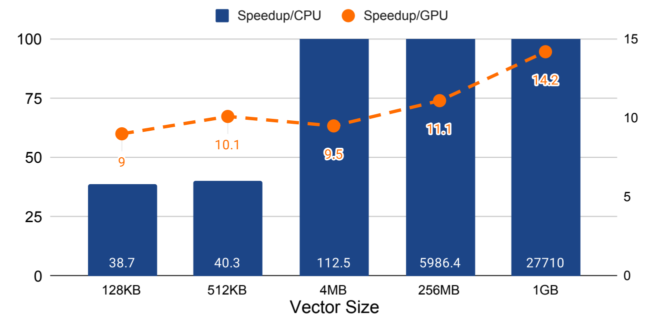


Fig. 19: Speedup over CPU and GPU when performing 1D convolution.

In TTD Engine, we first load or compute the original tensor-train format data. Then, the remaining operations are simple vector/matrix additions and multiplications over the tensor cores. TMU and SVD core can work together as an efficient 2D PE array to handle these operations. This idea can be further generalized to vectors with longer length and with different convolutional kernel sizes. Moreover, even if the vector's length increases dramatically, the tensor cores stay small, which makes the TT-format processing much more efficient. We implement the TT-format 1D convolution based on our TTD Engine, and compare it with baseline 1D convolution kernel running on CPU and GPU. As shown by the results in Figure 19, when comparing with CPU, TTD Engine achieves significant speedup ranging from $38.7\times$ to $21725\times$. Also, the speedup almost scales linearly after the 4MB bar. This is because as soon as the vector's size reaches a certain limit, the processing time of CPU grows proportional to the size of the tensor. However, with TT-format convolution scheme, the total computation is greatly reduced and much less influenced by the size of the vector. Therefore, the speedup of TTD Engine over CPU will grow rapidly for larger tensor. On the contrary, the speedup over GPU is more stable, ranging from $9.0\times$ to

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

$14.2\times$. This is because when the vector's size is small, the GPU execution time is not dominated by the computation, but other non-computation cost like kernel launching time. Only when the size is large enough, like from 256MB to 1GB, the computation time increases and the advantage of TTD Engine will be more obvious. For even larger input vector, we believe TTD Engine can achieve higher speedup over GPU as long as the TT representation of the vector is available.

C. Generalization to high-order convolution

Using the same idea, the TT-based 1D convolution scheme can be further generalized to high-order convolutions so that it can support various applications such as image processing and machine learning. Here we show how to apply the TT-based data processing to the 2D convolution problem. Firstly, the TT-format representation of a matrix $W \in \mathbb{R}^{M\times N}$ is given as follow:

$$W = G_1 * G_2 * \cdots * G_d \tag{4}$$

Where $G_i \in \mathbb{R}^{r_{i-1}\times m_i \times n_i \times r_i}$, $\prod m_i = M, \prod n_i = N$.

Different from TT-format vectors, each tensor core now has four dimensions, including two rank-dimensions, an m-dimension and an n-dimension. Using an example shown in Figure 20, suppose we have an $M \times N$ image where $M = N = 8$, and we want to perform a 2D convolution with a kernel of size 3×3. First, we can decompose this matrix into 3 tensor cores with a shape of $1 \times 2 \times 2 \times r_1$, $r_1 \times 2 \times 2 \times r_2$ and $r_2 \times 2 \times 2 \times 1$, respectively. We know that, a 2D convolution can be considered as two 1D convolutions along each of the dimension. This can be directly applied to TT-based convolution. Thus, we first perform a 1D convolution along the m dimension. In the 1D case, reversing the third core's second dimension is equivalent to switching every two consecutive pairs in the original vector. Here, as illustrated in Figure 20, if we reverse the m-dimension of the third core, every two consecutive rows in the original matrix will be exchanged. In other words, modifying the m-dimension of the tensor core is equivalent to operating on the whole rows of matrix W.

After changing the m-dimension, we indeed get several sub-matrices with modified rows. Similarly, we can further modify the columns of these sub-matrices by operating on the n-dimensions of the tensor cores. As illustrated in the example in Figure 20, if we reverse the n-dimension of core G_1, the left half of the matrix will be exchanged with the right half. Finally, we add these sub-matrices together, to get the 2D Convolution result. Similar to the 1D convolution, the high order convolution can also be efficiently executed on TTD Engine once the original tensor-train is obtained. After this, the proposed TT-convolution ensures the remaining computations to be executed only on certain slides of the few tensor cores. More importantly, if operations like TT-Addition cause the result tensor-train to have high TT-ranks, we can directly re-decompose the tensor-train with TTD Engine to get a new approximation with much lower TT-ranks.

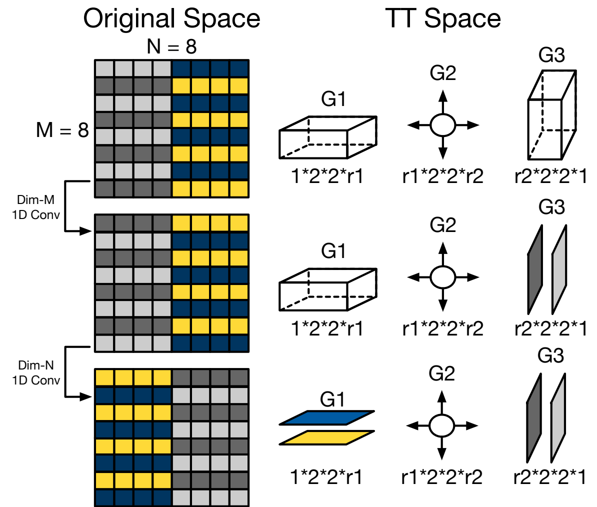


Fig. 20: TT-based 2D Convolution.

In this section, we present three case studies using TTD Engine for different applications. Medical image compression shows the straightforward benefits to decompose high order data using TTD to reduce memory consumption. Furthermore, we propose tensor-train data convolution to show the effectiveness of the TT-based data processing in terms of reducing computational complexity.

VIII. Related Work

Tensor Decomposition Algorithms. Tensor decomposition attempts to compress and represent a high-dimensional tensor with a smaller number of factor tensors. In this work, we aims at accelerating the TTD algorithm. In fact, there are also other efficient tensor decomposition methods apart from TTD. Polyadic Decomposition **(PD)** expresses an n-way tensor as the sum of r rank-1 terms. Particularly, when r is the minimal rank, the decomposition is called Canonical Polyadic Decomposition **(CPD)** . It is also called Canonical Decomposition (CANDECOMP) or Parallel Factor (PARAFAC) in the tensor community [33], [34]. Tucker decomposition [35]–[37] treats a tensor as a multilinear transformation of a core tensor $\mathcal{G}$ by the factor matrix B. It can be considered as an expansion in rank-1 terms that is not necessarily canonical. Among all the decomposition methods, TTD is preferred for high-order tensors since its resulting tensor factors have a low storage requirement linearly dependent on the number of orders and the dimension depth. Moreover, TT has a unique feature, that is it can be implemented with cross approximation [38] without knowing the whole tensor.

SVD Hardware Accelerators. To the best of our knowledge, TTD Engine is the first work to accelerate TTD. In fact, the whole tensor hardware community is still lacking exploration. Previous work have more focused on the acceleration of matrix decomposition. Accelerator design for SVD is a huge fraction [17]–[19]. However, these works have some restrictions that motivate us to conduct the algorithmic adaption together with our TTD hardware design. First, many of previous SVD accelerators target only matrices with a certain

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

shape or size. For instance, some can only support the square-shape matrices, while others cannot work when the input matrix's dimension exceeds the predefined dimension length. Second, directly applying Hestenes-Jacobi method to large-scale unbalanced matrices is extremely memory-inefficient, as it requires constant reads and writes for the original matrix that cannot be stored in local memory.

Tensor Hardware Accelerator. As aforementioned, the study of the tensor hardware, especially tensor-decomposition hardware is still at the early stage. [39] proposes the first FPGA-based accelerator for tucker decomposition. It focuses purely on the acceleration of Tucker decomposition, while the data processing techniques using Tucker-format tensor data are not covered. Other work [40], [41] mainly address the problem of designing general computation kernels for dense/sparse tensor data. While efficient hardware implementations for general Tensor-Tensor multiplications, Tensor-matrix multiplications are proposed, these work still suffer from the curse of dimensionality essentially due to the lack of decomposed tensor data.

IX. Conclusion

This paper presented the first customized architecture to accelerate TTD, a promising tensor technique that is increasingly used in EDA optimization, big data analysis, and machine learning. Experimental results show the proposed TTD Engine is at least 14.9× and 4.1× faster than its CPU and GPU counterparts, respectively. We scale a demo of our TTD Engine on an FPGA board and perform medical imaging compression tasks to demonstrate the application potential. Moreover, we have conducted a case study to use TT-method to implement convolutional operations. The TT-based convolution has shown significant advantages when dealing with large-scale data. With customized algorithm design and specialized hardware support, TTD has the potential to break the curse of dimensionality of big data processing, and this work may stimulate more efforts on this topic.

In the future, we plan to extend TTD Engine following two directions. 1) Since advanced TT Decomposition employs cross-approximation for low-rank matrix factorization, we plan to add the corresponding support in TTD Engine. Therefore, the users can choose the specific matrix factorization method they want to adopt when decomposing the tensor. 2) We plan to further demonstrate the effectiveness of TTD Engine when performing end-to-end applications using the introduced TT-format data processing pattern.

References

[1] A. Cichocki, D. Mandic, L. De Lathauwer, G. Zhou, Q. Zhao, C. Caiafa, and H. A. PHAN. Tensor decompositions for signal processing applications: From two-way to multiway component analysis. *IEEE Signal Processing Magazine*, 32(2):145–163, March 2015.

[2] Z. Zhang, K. Batselier, H. Liu, L. Daniel, and N. Wong. Tensor computation: A new framework for high-dimensional problems in eda. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(4):521–536, April 2017.

[3] Zheng Zhang, Xiu Yang, Ivan V. Oseledets, George E. Karniadakis, and Luca Daniel. Enabling high-dimensional hierarchical uncertainty quantification by ANOVA and tensor-train decomposition. *CoRR*, abs/1407.3023, 2014.

[4] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[5] Andrzej Cichocki. Era of big data processing: A new approach via tensor networks and tensor decompositions. *CoRR*, abs/1403.2048, 2014.

[6] Andrzej Cichocki, Rafal Zdunek, Anh-Huy Phan, and Shun-ichi Amari. *Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-Way Data Analysis and Blind Source Separation*. 10 2009.

[7] Stefan Klus, Patrick Gelß, Sebastian Peitz, and Christof Schütte. Tensor-based dynamic mode decomposition. *Nonlinearity*, 31(7):3359–3380, jun 2018.

[8] Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer. Optimization-based algorithms for tensor decompositions: Canonical polyadic decomposition, decomposition in rank-(lr, lr, 1) terms, and a new generalization. *SIAM Journal on Optimization*, 23:695–720, 2013.

[9] P. Comon and C. Jutten. *Handbook of Blind Source Separation*. 2010.

[10] Lieven De Lathauwer, Pierre Comon, and Nicola Mastronardi. Special issue on tensor decompositions and applications. *SIAM J. Matrix Analysis Applications*, 30, 01 2008.

[11] Andrzej Cichocki. Tensor networks for big data analytics and large-scale optimization problems. *CoRR*, abs/1407.3124, 2014.

[12] I. Oseledets and S. Dolgov. Solution of linear systems and matrix inversion in the tt-format. *SIAM Journal on Scientific Computing*, 34(5):A2718–A2739, 2012.

[13] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry P. Vetrov. Tensorizing neural networks. *CoRR*, abs/1509.06569, 2015.

[14] I. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[15] Brett W. Bader and Tamara G. Kolda. Algorithm 862: Matlab tensor classes for fast algorithm prototyping. *ACM Trans. Math. Softw.*, 32(4):635–653, December 2006.

[16] P. de Rijk. A one-sided jacobi algorithm for computing the singular value decomposition on a vector computer. *SIAM Journal on Scientific and Statistical Computing*, 10(2):359–371, 1989.

[17] X. Wang and J. Zambreno. An fpga implementation of the hestenes-jacobi algorithm for singular value decomposition. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 220–227, May 2014.

[18] Richard Brent, Franklin T. Luk, and CHARLES VAN LOAN. Computation of the singular value decomposition using mesh-connected processors. *Journal of VLSI and Computer Systems*, 1, 01 1985.

[19] L. M. Ledesma-Carrillo, E. Cabal-Yepez, R. d. J. Romero-Troncoso, A. Garcia-Perez, R. A. Osornio-Rios, and T. D. Carozzi. Reconfigurable fpga-based unit for singular value decomposition of large m x n matrices. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 345–350, Nov 2011.

[20] W. E. ARNOLDI. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9(1):17–29, 1951.

[21] R. Ballester. tntorch: Tensor network learning with pytorch. *https://github.com/rballester/tntorch*.

[22] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems 28 (NIPS)*. 2015.

[23] Timur Garipov, Dmitry Podoprikhin, Alexander Novikov, and Dmitry Vetrov. Ultimate tensorization: compressing convolutional and FC layers alike. *arXiv preprint arXiv:1611.03214*, 2016.

[24] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[25] Y. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[26] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2015.

[27] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[28] Tian Zhao, Yaqi Zhang, and Kunle Olukotun. Serving recurrent neural networks efficiently with a spatial accelerator, 09 2019.

[29] Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander Rush, Gu-Yeon Wei, and David Brooks. Masr: A modular accelerator for sparse rnns, 08 2019.

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

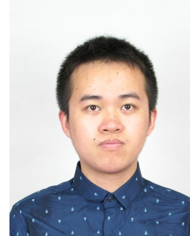
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

[30] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. *ArXiv*, abs/2005.05968, 2020.
[31] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 01 2009.
[32] I. V. Oseledets, S. Dolgov, V. Kazeev, D. Savostyanov, O. Lebedeva, P. Zhlobich, T. Mach, and L. Song. Tt-toolbox. 2011.
[33] J. Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, 35(3):283–319, Sep 1970.
[34] Richard A. Harshman, Peter Ladefoged, Heinrich Graf von Reichenbach, Robert I. Jennrich, Dale Terbeek, Lee Cooper, Andrew L. Comrey, Peter M. Bentler, Jeanne Yamane, and Diane Vaughan. Foundations of the parafac procedure: Models and conditions for an "explanatory" multimodal factor analysis. 1970.
[35] L.R Tucker and C.W Harris. Implications of factor analysis of three way matrices for measurements of change. In *Problems in measuring change*. University of Wisconsin Press, Madison, 1963.
[36] L. R. Tucker. The extension of factor analysis to three-dimensional matrices. In H. Gulliksen and N. Frederiksen, editors, *Contributions to mathematical psychology.*, pages 110–127. Holt, Rinehart and Winston, New York, 1964.
[37] Ledyard R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, Sep 1966.
[38] Ivan Oseledets and Eugene Tyrtyshnikov. Tt-cross approximation for multidimensional arrays. *Linear Algebra and its Applications*, 432(1):70 – 88, 2010.
[39] Kaiqi Zhang, Xiyuan Zhang, and Zheng Zhang. Tucker tensor decomposition on FPGA. *CoRR*, abs/1907.01522, 2019.
[40] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. Herr, C. Hughes, T. Mattson, and P. Dubey. T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–189, 2019.
[41] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 689–702, 2020.

Zheng Qu received the B.S. degree from Tsinghua University, Beijing, China, in 2018. He is currently working toward the Ph.D. degree at the Scalable Energy-efficient Architecture Lab (SEAL), Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA, USA. His current research interests include artificial intelligence (AI) accelerator and architecture, field-programmable gate array (FPGA) design, algorithm and hardware co-design for high-dimensional data processing.

Lei Deng received the B.E. degree from University of Science and Technology of China, Hefei, China in 2012, and the Ph.D. degree from Tsinghua University, Beijing, China in 2017. He is currently a Postdoctoral Fellow at the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA. His research interests span the areas of brain-inspired computing, machine learning, neuromorphic chip, computer architecture, tensor analysis, and complex networks. Dr. Deng has authored or co-authored over 60 refereed publications. He was a PC member for *ISNN* 2019. He currently serves as a Guest Associate Editor for *Frontiers in Neuroscience* and *Frontiers in Computational Neuroscience*, and a reviewer for a number of journals and conferences. He was a recipient of MIT Technology Review Innovators Under 35 China 2019.

Bangyan Wang received his B.E. degree from Tsinghua University, Beijing, China in 2017. He is currently a Ph.D. student at the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA. His current research interests include domain-specific accelerator design and tensor analysis.



Hengnu Chen received his B.E. degree from Tsinghua University, Beijing, China in 2017. He is currently pursuing the Ph.D. degree at the Center for Brain Inspired Computing Research (CBICR), Tsinghua University, Beijing, China. His current research interests include high-order tensor decomposition methods, tensor network based algorithms and architectures, compression and acceleration for neural networks, etc.

Jilan Lin received the B.S. degree from Tsinghua University, Beijing, China, in 2018. He is currently pursuing the Ph.D. degree at the Scalable Energy-efficient Architecture Lab (SEAL), Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA, USA. His current research interests include accelerator design for graph analytics and machine learning.

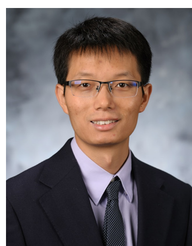
Ling Liang received the B.E. degree from Beijing University of Posts and Telecommunications, Beijing, China, in 2015, and M.S. degree from University of Southern California, CA, USA, in 2017. He is currently pursuing the Ph.D. degree at Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA. His current research interests include machine learning security, tensor computing and computer architecture.

Guoqi Li received the B.E. degree from the Xi'an University of Technology, Xi'an, China, in 2004, the M.E. degree from Xi'an Jiaotong University, Xi'an, China, in 2007, and the Ph.D. degree from Nanyang Technological University, Singapore, in 2011. He was a Scientist with Data Storage Institute and the Institute of High Performance Computing, Agency for Science, Technology and Research (ASTAR), Singapore, from 2011 to 2014. He is currently an Associate Professor with the Center for Brain Inspired Computing Research (CBICR), Tsinghua University, Beijing, China. His current research interests include machine learning, brain-inspired computing, neuromorphic chip, complex systems and system identification.

Dr. Li has authored or co-authored over 80 journal and conference papers. He has been actively involved in professional services such as serving as the International Technical Program Committee Member, the PC Member, the Publication Chair, the Tutorial/Workshop Chair, and the Track Chair for international conferences. He is currently an Editorial-Board Member for *Control and Decision* and *Frontiers in Neuroscience, Neuromorphic Engineering*, and an Associate Editor for *Frontiers in Neuroscience, Neuromorphic Engineering*. He is a reviewer for *Mathematical Reviews* published by the American Mathematical Society and serves as a reviewer for a number of other prestigious journals and conferences. He was the recipient of the 2018 First Class Prize in Science and Technology of the Chinese Institute of Command and Control, Best Paper Awards (*EAIS* 2012 and *NVMTS* 2015), and the 2018 Excellent Young Talent Award of Beijing Natural Science Foundation.

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3058317, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



Zheng Zhang (M'15) has been an Assistant Professor of Electrical and Computer Engineering with the University of California at Santa Barbara (UCSB), since July 2017. He received his Ph.D in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology (MIT), Cambridge, MA, in 2015, M.Phil from the University of Hong Kong in 2010, and B. Eng from Huazhong University of Science and Technology in 2008. His industrial experiences include Coventor Inc., Cambridge, MA, and Maxim-IC, Colorado Springs, CO, USA; academic visiting experiences include the University of California at San Diego, Brown University, and Politechnico di Milano, Milan, Italy; government laboratory experiences include the Argonne National Laboratory, Lemont, IL, USA. His research interests include uncertainty quantification and tensor computation with multi-domain applications including CAD of nano-scale IC/MEMS/photonics, data analytics, machine learning and autonomous systems.

Dr. Zhang received three Best Paper Awards from IEEE Transactions: the Best Paper Award of *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* in 2014, two Best Paper Awards of *IEEE Transactions on Components, Packaging and Manufacturing Technology* in 2018 and 2020, respectively. He also received two Best Paper Awards and three additional Best Paper Nominations at international conferences. His PhD dissertation was recognized by the ACM SIGDA Outstanding Ph.D Dissertation Award in Electronic Design Automation in 2016, and by the Doctoral Dissertation Seminar Award (i.e., Best Thesis Award) from the Microsystems Technology Laboratory of MIT in 2015. He received the NSF CAREER Award in 2019.

Yuan Xie received the B.S. degree in Electronic Engineering from Tsinghua University, Beijing, China in 1997, and M.S. and Ph.D. degrees in Electrical Engineering from Princeton University, NJ, USA in 1999 and 2002, respectively. He was an Advisory Engineer with IBM Microelectronic Division, VT, USA from 2002 to 2003. He was a Full Professor with Pennsylvania State University, PA, USA from 2003 to 2014. He was a Visiting Researcher with Interuniversity Microelectronics Centre (IMEC), Leuven, Belgium from 2005 to 2007 and in 2010. He was a Senior Manager and Principal Researcher with AMD Research China Lab, Beijing, China from 2012 to 2013. He is currently a Professor with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, CA, USA. His interests include VLSI design, Electronics Design Automation (EDA), computer architecture, and embedded systems.

Dr. Xie is an expert in computer architecture who has been inducted to *ISCA*/*MICRO*/*HPCA* Hall of Fame and IEEE/AAAS/ACM Fellow. He was a recipient of Best Paper Awards (*HPCA* 2015, *ICCAD* 2014, *GLSVLSI* 2014, *ISVLSI* 2012, *ISLPED* 2011, *ASPDAC* 2008, *ASICON* 2001) and Best Paper Nominations (*ASPDAC* 2014, *MICRO* 2013, *DATE* 2013, *ASPDAC* 2010-2009, *ICCAD* 2006), the 2016 IEEE Micro Top Picks Award, the 2008 IBM Faculty Award, and the 2006 NSF CAREER Award. He served as the TPC Chair for *ICCAD* 2019, *HPCA* 2018, *ASPDAC* 2013, *ISLPED* 2013, and *MPSOC* 2011, a committee member in IEEE Design Automation Technical Committee (DATC), the Editor-in-Chief for *ACM Journal on Emerging Technologies in Computing Systems*, and an Associate Editor for *ACM Transactions on Design Automations for Electronics Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *IEEE Transactions on VLSI, IEEE Design and Test of Computers*, and *IET Computers and Design Techniques*. Through extensive collaboration with industry partners (e.g. AMD, HP, Honda, IBM, Intel, Google, Samsung, IMEC, Qualcomm, Alibaba, Seagate, Toyota, etc.), he has helped the transition of research ideas to industry.

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on February 10,2021 at 19:12:05 UTC from IEEE Xplore. Restrictions apply.