Tucker Tensor Decomposition on FPGA

Kaiqi Zhang, Xiyuan Zhang and Zheng Zhang
Department of Electrical & Computer Engineering, University of California, Santa Barbara, CA 93106
kzhang70@ucsb.edu; xiyuanzhang@ucsb.edu; zhengzhang@ece.ucsb.edu

***Abstract*—Tensor computation has emerged as a powerful mathematical tool for solving high-dimensional and/or extreme-scale problems in science and engineering. The last decade has witnessed tremendous advancement of tensor computation and its applications in machine learning and big data. However, its hardware optimization on resource-constrained devices remains an (almost) unexplored field. This paper presents an hardware accelerator for a classical tensor computation framework, Tucker decomposition. We study three modules of this architecture: tensor-times-matrix (TTM), matrix singular value decomposition (SVD), and tensor permutation, and implemented them on Xilinx FPGA for prototyping. In order to further reduce the computing time, a warm-start algorithm for the Jacobi iterations in SVD is proposed. A fixed-point simulator is used to evaluate the performance of our design. Some synthetic data sets and a real MRI data set are used to validate the design and evaluate its performance. We compare our work with state-of-the-art software toolboxes running on both CPU and GPU, and our work shows $2.16 - 30.2\times$ speedup on the cardiac MRI data set.**

I. Introduction

As a multi-dimension extension of matrices, tensors are a natural tool to represent and process multi-way data arrays [1]. For instance, an MRI sequence with three spatial dimensions and a time dimension can be naturally represented as a 4-way tensor. The convolution layer in a neural network is also a tensor. Leveraging various tensor decompositions [2]–[4], many high-dimensional data mining [5], machine learning [6]–[11] and EDA [12], [13] problems have been solved efficiently without suffering from the curse of dimensionality.

Due to their superior performance in processing high-volume data, tensors have emerged as a promising tool to enable real-time machine learning and data analysis. Recently, tensor algorithms have achieved great success in training and compressing deep neural networks [6]–[11]. The resulting tensorized models consume tremendously less memory, runtime and power than the original deep neural network models. Tensor algorithms have also been successfully employed to accelerate medical image analysis [14], anomaly detection [15] and speech recognition [16].

Despite the rapid progress of tensor algorithms, the hardware/algorithm co-design of tensor computation on resource-constrained platforms remains a new and (almost) unexplored field. Some tensor libraries [17]–[19] have been developed for high-performance platforms like clusters and super computers. However, little algorithm-architecture co-design targeting on power and cost-limited devices has been done, which has limited the application of tensor-based data analysis and machine learning on IoT and edge devices. Although many hardware accelerators are available for matrix and vector computations [20], [21] and have been applied to machine learning [22]–[24] and signal processing [25], they cannot handle tensor data, because the underlying theory and numerical procedures are fundamentally different. It is inefficient and error-prone to process tensor data by matrix- or vector-computation accelerators. Therefore, resource-constrained hardware accelerators for tensor computation are highly desired.

This paper presents, for the first time, a hardware accelerator for one of the most important tensor algorithms: Tucker decomposition [3]. Tucker decomposition is a high-order generalization of singular value decomposition (SVD) and principal component analysis (PCA), and it often achieves orders-of-magnitude higher data compression ratio than matrix compression algorithms on multi-way data. This method has been widely used in facial recognition [26], signal processing [27], deep learning [28] and data mining [5]. Tucker decomposition is often implemented via the high-order orthogonal iteration (HOOI) [29]. This algorithm involves some computation-intensive operations such as the tensor-times-matrix (TTM) and matrix SVD. Meanwhile, handling the huge amount of tensor data on FPGA or ASIC is a challenging task.

The contributions of our paper are summarized below:

- On the hardware side, we present an hardware architecture for Tucker decomposition. We describe the design and data communication of three units: TTM, SVD via Jacobi iterations, and tensor permutation/reshaping.
- On the algorithm side, we propose a warm-start algorithm to reduce the cost of the Jacobi iterations.
- We analyze the performance of our accelerator, implement it on a Xilinx FPGA, and show the implementation results.
- We compare our FPGA accelerator with some state-of-the-art algorithms on both CPU and GPU, and demonstrate its application on an MRI compression task. Our accelerator shows up to $30.2\times$ speedup on the MRI data set.

II. Algorithm Background

In this section, we introduce the necessary background of tensors and Tucker decomposition.

A. Tensors and Basic Tensor Operations

Notations: We use boldface lower-case letters (e.g., $\mathbf{x}$) to denote vectors, boldface capital letters (e.g. $\mathbf{X}$) to denote matrices, and boldface Euler script letters (e.g. $\boldsymbol{\mathcal{X}}$) to denote tensors.

A tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_d}$ is a multidimensional data array. Here d is called the order or way of $\boldsymbol{\mathcal{X}}$. An integer $k \in [1, d]$ can be used as the index of a specific dimension or mode with size I_k. An entry of a tensor can be specified by an index vector. For instance, the $(i_1, i_2, \ldots i_d)$-th entry in tensor $\boldsymbol{\mathcal{X}}$ is denoted by $x_{i_1, i_2, \ldots i_d}$. Clearly, a vector and a matrix are order-1 and order-2 tensors, respectively.

Tensor Fiber and Slice: A **fiber** is a one-dimensional fragment of a tensor, obtained by fixing all indices but one. Tensor fibers are higher-order extension of matrix rows and columns. A third-order tensor has fibers that can be denoted by $\mathbf{x}_{:jk}$, $\mathbf{x}_{i:k}$ or $\mathbf{x}_{ij:}$ correspondingly. A tensor **slice** is a two-dimensional fragment of a tensor, obtained by fixing all indices but two. For instance, $\mathbf{X}_{i::}$, $\mathbf{X}_{:j:}$ and $\mathbf{X}_{::k}$ denote the horizontal, lateral, and frontal slices of a 3-way tensor, respectively. Fig. 1 shows the slices and fibers of a tensor.

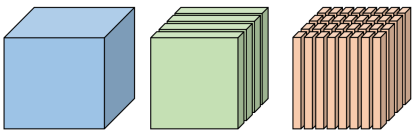


Fig. 1: Left to right: a tensor, slices, and fibers.

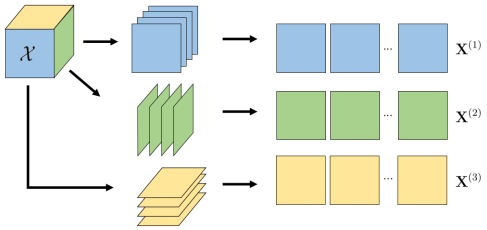


Fig. 2: Tensor unfolding.

Tensor permutation: Tensor permutation changes the mode order of a tensor. It is a high-order extension of the matrix transpose. For instance, given $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{5\times10\times3}$, $\mathbf{permute}(\boldsymbol{\mathcal{X}}, [2, 3, 1])$ generates a new tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{10\times3\times5}$ with $y_{i_2,i_3,i_1} = x_{i_1,i_2,i_3}$.

Unfolding: Unfolding (or matricization) as shown in Fig. 2, is the process of transforming a tensor into a matrix. Unfolding reorders the elements of an d-way data array into a matrix. For instance, the mode-k unfolding of a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times I_2\cdots\times I_d}$ is denoted as $\mathbf{X}^{(k)} \in \mathbb{R}^{I_k\times \prod_{m\neq k} I_m}$. The mapping from the $(i_1, \ldots, i_d)$-th element $\boldsymbol{\mathcal{X}}$ to the (i_k, j)-th element of $\mathbf{X}^{(k)}$ is given as follows

$$j = 1 + \sum_{m=1,m\neq k}^{d} \left((i_m - 1) \prod_{n=1,n\neq k}^{m-1} I_n \right). \tag{1}$$

TTM: Tensor-times-matrix (TTM) in short, is a high-dimensional expansion of matrix multiplication. Given a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times I_2\cdots\times I_d}$ and a matrix $\mathbf{A} \in \mathbb{R}^{J\times I_k}$, their k-mode product is a new tensor

$$\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_k \mathbf{A} \in \mathbb{R}^{I_1\times\cdots I_{k-1}\times J\times I_{k+1}\cdots\times I_d}. \tag{2}$$

It can be expressed as matrix-matrix multiplication

$$\mathbf{Y}^{(k)} = \mathbf{A}\mathbf{X}^{(k)} \tag{3}$$

However, implementing it in this way directly can be inefficient on hardware because tensor permutation will be needed. This needs to move almost all the data in a tensor.

Tensor Norms: The norm of a tensor is a function which maps any tensor to a non-negative scalar. A widely used norm of tensors is the Frobenius norm, defined as

$$||\boldsymbol{\mathcal{X}}||_{\mathrm{F}} = \sqrt{\sum_{i_1,i_2,\ldots i_d} x^2_{i_1,i_2,\ldots i_d}}. \tag{4}$$

B. Tenser Tucker Decomposition

Given an d-way tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_d}$, we may compress it by the Tucker decomposition [30]. As shown in Fig. 3, the Tucker decomposition approximates a large-size

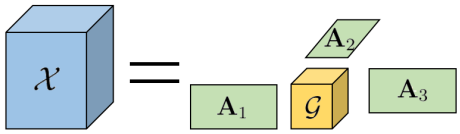


Fig. 3: Tucker decomposition.

tensor with a small-size core tensor $\boldsymbol{\mathcal{G}} \in \mathbb{R}^{R_1\times R_2\cdots\times R_d}$ and d factor matrices $\{\mathbf{A}_k \in \mathbb{R}^{I_k\times R_k}\}_{k=1}^{d}$ as follows:

$$\boldsymbol{\mathcal{X}} \approx \boldsymbol{\mathcal{G}} \times_1 \mathbf{A}_1 \times_2 \mathbf{A}_2 \times \cdots \times_d \mathbf{A}_d, \tag{5}$$

where all columns are orthonormal inside each factor matrix $\mathbf{A}_k$. We call $\mathbf{R} = [R_1, R_2, \cdots, R_d]$ as the **multilinear rank** of $\boldsymbol{\mathcal{X}}$. When $R_k \ll I_k$, $\boldsymbol{\mathcal{X}}$ can be represented with the above Tucker format at a much lower cost. Once all factor matrices are obtained, the core tensor can be computed as

$$\boldsymbol{\mathcal{G}} = \boldsymbol{\mathcal{X}} \times_1 \mathbf{A}_1^T \times_2 \mathbf{A}_2^T \times \cdots \times_d \mathbf{A}_d^T. \tag{6}$$

Two popular methods can be used to compute a Tucker decomposition. The first well-known method is the high-order SVD (HOSVD) [3]. The idea of HOSVD is simple:

- One first unfolds $\boldsymbol{\mathcal{X}}$ along mode k to get $\mathbf{X}^{(k)}$;
- Then, perform a singular value decomposition (SVD)

$$\mathbf{X}^{(k)} = \mathbf{U}_k\mathbf{S}_k\mathbf{V}_k^T. \tag{7}$$

- Finally, pick $\mathbf{A}_k$ as the first R_k columns of $\mathbf{U}_k$.

This method is easy to implement, however it is not optimal in fitting the data. Alternatively, an alternative least-square method called HOOI is widely used to get a better solution.

HOOI: The High Order Orthogonal Iteration (HOOI) [29], [31], [32] method aims to minimize the approximation error

$$\min_{\{\mathbf{A}_k\}_{k=1}^{d}} ||\boldsymbol{\mathcal{X}} - \boldsymbol{\mathcal{G}} \times_1 \mathbf{A}_1 \times_2 \mathbf{A}_2 \times \cdots \times_d \mathbf{A}_d||_F \tag{8}$$

through the iterative process as shown in Alg. 1. Each iteration of the HOOI consists of two steps for every mode index k: (1) obtain a tensor $\boldsymbol{\mathcal{B}}$ via a power iteration (TTM along all modes except k), which can be done from mode d, then $d-1$, ... until mode 1. (2) an SVD of the mode-k unfolded matrix of $\boldsymbol{\mathcal{B}}$ to extract a mode-k factor matrix $\mathbf{A}_k$.

In practice, the initialization process via HOSVD can be very time-consuming, because it needs d SVD operations, and each of it works on a matrix whose size is equal to the original tensor. Therefore, some random orthonormal matrices are often used as the initial factor matrices. In this case, the total number of iterations needed may increase slightly, but the total runtime can decrease significantly. Even though, when the size of the tensor $\boldsymbol{\mathcal{X}}$ is large, the time and energy comsumed to compute the Tucker decomposition can be very high.

III. Proposed Architecture

In this section, we propose a new hardware architecture to perform Tucker decomposition via HOOI.

Fig. 4 shows the system-level diagram of our architecture. In order to load and accommodate the huge amount of tensor data elements, our design stores the tensor in an external DRAM. The HOOI engine consists of three parts: a tensor-times-matrix (TTM) unit, a singular value decomposition (SVD) unit, and a tensor permuting/reshaping unit. The data elements of the three

Algorithm 1: HOOI for Tucker Decomposition

Initialize $\{\mathbf{A}_k\}_{k=1}^d$ via HOSVD
while not converge **do**
  for $k = 1, 2, \ldots, d$ **do**
    $\boldsymbol{\mathcal{B}} \leftarrow \boldsymbol{\mathcal{X}} \times_1 \mathbf{A}_1^T \cdots \times_{k-1} \mathbf{A}_{k-1}^T \times_{k+1} \mathbf{A}_{k+1}^T \cdots \times_d \mathbf{A}_d^T$
    Unfold $\boldsymbol{\mathcal{B}}$ and perform SVD: $\mathbf{B}^{(k)} = \mathbf{U}_k \mathbf{S}_k \mathbf{V}_k^T$
    $\mathbf{A}_k \leftarrow$ the first R_k columns of $\mathbf{U}_k$.
  end for
end while
return $\{\mathbf{A}_k\}_{k=1}^d$.

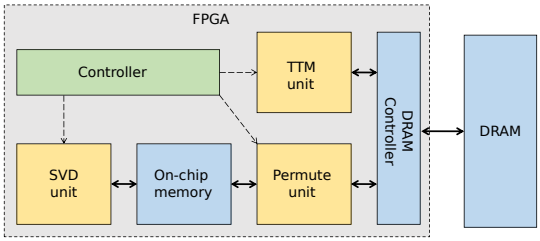


Fig. 4: Overall structure of our Tucker decomposition.

units are stored in different memories. Because the size of a tensor can be very large, all tensors (including the intermediate and final results of TTM) are stored in an external DRAM. The matrix for each SVD is stored in an on-chip memory to reduce latency and to achieve a maximum throughput. Both parts are parallel and pipelined to achieve the maximum performance. The tensor permuting unit moves the data between the on-chip memory used by the SVD unit and the external DRAM used by the TTM unit, and it permutes the tensor accordingly.

A. Tensor-Times-Matrix (TTM) Unit

The TTM unit can be implemented as a matrix-matrix multiplication that is available in some common computational libraries. However, we need to permute tensors before it can be implemented in this way. This is time- and memory-consuming because almost all $\prod_{k=1}^{d} I_k$ data elements of $\boldsymbol{\mathcal{X}}$ have to be moved. In this work, we develop a TTM unit *without tensor permutations*. For simplicity, the tensors are always stored by incrementing the mode-1 index i_1, then the second index i_2, and so forth. Since the size of tensor $\boldsymbol{\mathcal{X}}$ is often beyond the capacity of an on-chip memory, all data elements (including the input and output data, and the intermediate result of a TTM operation) are stored in an external DRAM.

TTM with a 2-D PE Array. Our TTM unit is shown in Fig. 5. In order to maximize the throughput of the TTM module, we design a 2-D array of processing elements (PE) with q columns and r rows. Each PE computes the product of one scalar element of the tensor (black arrow) and one scalar from the matrix (either from the blue line or stored in the PE) at each clock cycle. The PEs in a single column are always connected to the same bus so they handle the same element in the tensor $\boldsymbol{\mathcal{X}}'$ each time. Here $\boldsymbol{\mathcal{X}}'$ can be either the original tensor $\boldsymbol{\mathcal{X}}$ or an intermediate tensor after the TTM of $\boldsymbol{\mathcal{X}}$ with some factor matrices. With q columns, this module can handle up to q neighbouring elements of a tensor fiber in total at the same time. Due to our method of storing $\boldsymbol{\mathcal{X}}'$, the fibers are

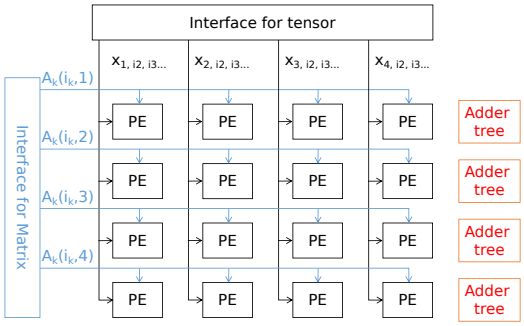


Fig. 5: The TTM unit. The red part is used when computing the mode-1 product and blue part is used when computing mode-j product when $j \neq 1$.

always obtained by only changing the first mode index. Each row of this array handles one column of the factor matrix $\mathbf{A}_j$.

We compute the mode-j TTMs in a decreasing order from $j = d$ to $j = 1$. This PE array works in different ways dependent on the value of j, which is explained below.

- Assume that $j \neq 1$ and that we have done TTMs for all modes $> j$ except mode k. Let us ignore mode k for simplicity, and the size of $\boldsymbol{\mathcal{X}}'$ becomes $I_1 \times I_2 \times \ldots I_j \times R_{j+1} \times \cdots \times R_d$. To simplify the expression, we fold the modes $1, 2, \ldots j-1$ into a single mode and use $\hat{i}$ as its index. The element-wise expression of this operation is

$$y_{\hat{i},r_j,i_{j+1}\ldots i_d} = \sum_{j}^{I_j} x'_{\hat{i},i_j,i_{j+1}\ldots i_d} \mathbf{A}_j(i_j, r_j). \tag{9}$$

  In each clock cycle, each vertical bus can carry some neighbouring elements in a tensor fiber $\mathbf{x}'_{:i_j\ldots i_d}$ and each horizontal bus can carry an element in the factor matrix. Specifically, In the n-th cycle of the i_j-th round, the l-th vertical bus and the r_j-th horizontal bus carry scalars $x'_{\hat{i},i_j,\ldots,i_d}$ and $\mathbf{A}_j(i_j, r_j)$, respectively. Consequently the (r_j, l)-th PE calculates

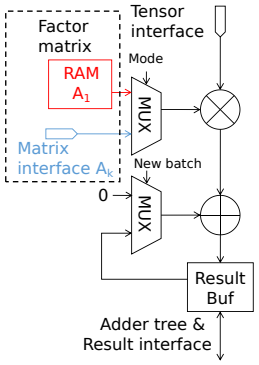
$$x'_{\hat{i},i_j,\ldots,i_d} \times a_{i_j,r_j}, \text{ with } \hat{i} = l + nq \leq \hat{I}' = I_1 I_2 \cdots I_{j-1}.$$

  Finally $y_{\hat{i},r_j,i_{j+1},\ldots,d}$ is obtained by summing the above product terms over i_j (in each round). Note that the index of matrix elements in $\mathbf{A}_j$ used at each PE does not depend on l. Therefore, we can multiply the whole fiber $\mathbf{x}'_{:i_j\ldots i_d}$ with the same matrix element, and all PEs in the same row share the same data elements of $\mathbf{A}_j$ by connecting them to the same horizontal bus. Because the size the new dimension, $\hat{I} = I_1 I_2 \ldots I_{k-1}$, is very large, we divide the partly-folded tensor $\boldsymbol{\mathcal{X}}'$ into some small sub-tensors of size $m \times I_j \times R_{j+1} \cdots \times R_d$, and the resulting tensor $\boldsymbol{\mathcal{Y}}$ into some sub-tensors of size $m \times R_j \times R_{j+1} \cdots \times R_d$. We can compute the sub-tensors of $\boldsymbol{\mathcal{Y}}$ one by one to reduce the buffer size.
- When computing $\boldsymbol{\mathcal{X}}' \times_1 \mathbf{A}_1^T$, the element-wise result is

$$y_{r_1,i_2,\ldots,i_d} = \sum_{i_1=1}^{I_1} x'_{i_1,i_2,\ldots,i_d} \mathbf{A}_1(i_1, r_1). \tag{10}$$

  The r_1-th row of the PE array computes q product terms of (10). Because q is usually smaller than the fiber length, we need to compute all product terms by several cycles. In the n-th cycle, the (r_1, l)-th PE calculates

$$x'_{i_1,i_2,\ldots,i_d} \times a_{i_1,r_1}, \text{ with } i_1 = l + nq \leq I_1.$$



(a)

In-place adder tree:

1: **for** j=1 **to** log(n) **do**
2: **for** i=1 **to** n/2 **do**
3: $a_i \leftarrow a_{2i-1} + a_{2i}$
4: **end for**
5: **for** j=n/2+1 **to** n **do**
6: $a_i \leftarrow 0$
7: **end for**
8: **end for**

(b)

Fig. 6: (a) The details of a PE. The red part is used only for mode-1 TTM, and blue part is only for mode-j TTM with $j \neq 1$. (b) An in-pace adder tree for $\mathbf{a} \in \mathbb{R}^n$.

Note that in mode-1 TTM, the RAM inside the (r_1, l)-th PE stores all a_{i_1,r_1}'s for all $i_1 = l + nq \leq I_1$. The TTM element $y_{r_1,i_2,\ldots i_d}$ is obtained by accumulating all products terms in the r_1-th row during all cycles.

Processing Element (PE). As shown in Fig. 6, each PE consists of a multiplier, a small RAM, and another memory used as an output buffer storing the result before writing it to a DRAM. The RAM (marked in red) is only used to store $\mathbf{A}_1$ when computing the mode-1 TTM. Otherwise, the bus at each row imports data elements of $\mathbf{A}_j$ when $j \neq 1$. Therefore, a mutiplexer (MUX) is used to select the correct data to perform product operations. After computing one batch of the data (a tensor fiber if $j = 1$ and a tensor slice if $j \neq 1$), the result is written to the DRAM and then reset to zero. The buffer temporarily holds the intermediate results, and keeps updating during the multiplication and sum operations. The buffer stops updating when its data is written into an external memory. In order to avoid timing conflicts and to increase throughput, another buffer is used (not shown in the figure) for transferring data to an external memory. These two buffers switch their roles after processing each batch of data.

In-place Adder Tree. As mentioned above, we need an adder tree for the mode-1 TTM. If the adder tree is implemented as a pipeline, a lot of adders and registers will be used. Given that the product terms need to be summed up only once per batch of data instead of per clock cycle, we only need an in-place adder tree. We split a adder tree into multiple stages. After each stage, each two elements are summed up so the total number of data elements is reduced by 50%. The registers and adders are shared among different stages. The data elements are read from and write back to the same group of registers after each clock cycle. This is why we call it an "in-place" adder tree. This treatment can reduce the number of adders and registers by 50%.

B. Singular Value Decomposition (SVD) Unit

Our SVD unit employs the Jacobi iterations [33]–[37]. Both single-side and double-side Jacobi iterations are widely used. We use the single-side version because of its higher accurate, ease to parallelize and less data dependency. The whole framework is summarized in Alg. 2. Given a matrix $\mathbf{B}$, this algorithms computes its left singular vectors by orthogonizing every two rows (i.e., $\mathbf{b}_{i,:}$ and $\mathbf{b}_{j,:}$) of the matrix iteratively.

Algorithm 2: SVD via single-side Jacobi iteration

1: Input: $\mathbf{B} = \mathbf{B}^{(k)}$, initial guess $\mathbf{U} = \mathbf{I}$.
2: **while** Not converge **do**
3: **for** any $(i, j), 1 \leq i < j \leq n$, $i \neq j$ **do**
4: $\alpha = \|\mathbf{b}_{i,:}\|_2^2$, $\beta = \|\mathbf{b}_{j:}\|_2^2$, $\gamma = \langle \mathbf{b}_{i:}, \mathbf{b}_{j:} \rangle$
5: $\theta = \arctan(\frac{2\gamma}{\beta - \alpha})$
6: $\mathbf{b}_{i:} = \mathbf{b}_{i:} \cos\theta - \mathbf{b}_{j:} \sin\theta$,
$\mathbf{b}_{j:} = \mathbf{b}_{i:} \sin\theta + \mathbf{b}_{j:} \cos\theta$
7: $\mathbf{u}_{i:} = \mathbf{u}_{i:} \cos\theta - \mathbf{u}_{j:} \sin\theta$,
$\mathbf{u}_{j:} = \mathbf{u}_{i:} \sin\theta + \mathbf{u}_{j:} \cos\theta$
8: **end for**
9: **end while**
10: **return** $\mathbf{U}$, with its i-th row being $\mathbf{u}_{i:}$.

The iterative process involves the norms and inner products of the row vectors and performing some rotations.

We can select the order of (i, j) in different ways. A natural choice is to increment j in the inner loop and increment i in the outer loop, or vice versa. However, because of the data dependency between two adjacent operations, this choice makes it impossible to implement parallel or pipe-lined design. In order to overcome this issue, we employ the round-robin ordering suggested in [38], which eliminates the data dependency between adjacent iterations. This method starts by dividing all indices into $n/2$ pairs $(1, 2), (3, 4), \ldots (n-1, n)$ where n is the total number of rows. After orthogonalizing all the pair of rows specified above, we generate new index pairs in this way: suppose the pair in the previous round is (p, q), this pair index is updated in the next round as

$$\begin{cases} (p+1, q-1) & \text{if } q - p > 2, \\ (1, p+q) & \text{if } q - p \leq 2 \text{ and } p + q \leq n, \\ (p+q+1-n, n) & \text{if } q - p \leq 2 \text{ and } n < p + q < 2n - 1, \\ (1, 2) & \text{if } q = n \text{ and } p = n - 1. \end{cases}$$

Fig. 7 shows the block diagram of our SVD unit. In each step, two vectors are orthogonalized. The on-chip memory provides two ports to operate independently. In this part, one port takes the two vectors from the memory, and another port writes the orthogonalized vectors back into the memory. Because we use only one port to input data and the other for output, the two vectors have to be fetched through the same port alternatively. Three sum-of-products are needed to calculate the rotation angle θ. Given that the two vectors are fetched alliteratively, the multiplier and adder to get $\|\mathbf{b}_{i:}\|_2$ and $\|\mathbf{b}_{j:}\|_2$ can be shared. Therefore, only two sets of multipliers and adders are used. In order to get θ, $\sin\theta$ and $\cos\theta$, we employ the CORDIC algorithm [39], which uses the rotations of some fixed angles to approximate the rotation of any angle. This algorithm is efficient to calculate the trigonometric functions on hardware, and an FPGA IP core is available. After these two vectors are fetched from the memory, they are stored in a local buffer until $\sin\theta$ and $\cos\theta$ are calculated, then they are rotated accordingly. In this way, it is guaranteed each vector is read from the memory only once when orthogonalizing each pair of rows.

Besides the input matrix $\mathbf{B}$, the orthogonal matrix $\mathbf{U}$ also needs to be rotated in the same way. We store $\mathbf{U}$ and $\mathbf{B}$ in the same memory and handle them in the same way, except that $\mathbf{U}$ is not used for calculating α, β and γ. Since $\mathbf{U}$ has a much smaller size than $\mathbf{B}$, such a design only causes negligible run-time overhead but saves half of the area and power.

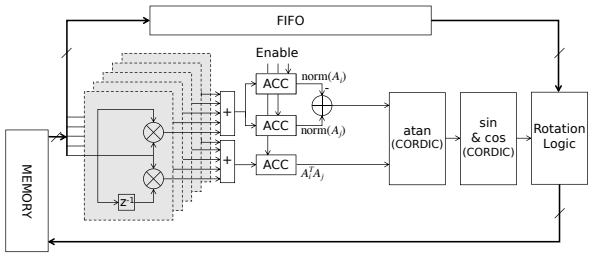


Fig. 7: Block diagram of SVD unit. ACC: accumulator. FIFO: first-in, first-out queue.

C. Tensor Permutation and Reshaping

Once $\boldsymbol{\mathcal{B}} = \boldsymbol{\mathcal{X}} \times_1 \mathbf{A}_1^T \cdots \times_{k-1} \mathbf{A}_{k-1}^T \times_{k+1} \mathbf{A}_{k+1}^T \cdots \times_d \mathbf{A}_d^T$ is computed, we need to permute and reshape the tensor $\boldsymbol{\mathcal{B}}$ into $\mathbf{B}^{(k)}$ before performing an SVD. Since $\boldsymbol{\mathcal{B}}$ is stored in an external DRAM and the matrix $\mathbf{B}^{(k)}$ is stored in an on-chip memory, we need an extra module to move the data between the DRAM and the on-chip memory and re-organize the data elements to match $\mathbf{B}^{(k)}$. After SVD, the factor matrix need to be transposed and moved from the on-chip memory to DRAM, which is done by this module as well.

When moving the data from on-chip memory and external DRAM, the data is first read from the its original memory, written to a local buffer with size $p' \times q'$, then written to the destination memory. The size of the buffer determines the size of data set that can be moved in every batch.

IV. Implementation Details

A. Fixed Point Number

Floating-point numbers usually cause higher hardware cost and longer latency. Therefore, we use a fixed-point number system based on the trade-off between accuracy and hardware complexity.

We decide the fixed-point precision based on some hardware constrains. Because the data width at the interface of a DRAM controller is fixed as 512 bits, the memory is most efficient if the data width is a factor of 512 (i.e., 2^n with integer $n \leq 9$). Meanwhile, each DSP our FPGA can perform an multiplication with data sizes up to 27 bits $\times$ 18 bits. Considering these constraints, we use 16-bit numbers to represent all tensor data elements, and store them in an external DRAM. On the other hand, we use 27-bit numbers to represent the matrix data in both TTM and SVD in order to achieve higher accuracy and to avoid excessive use of multipliers. Note that we use a smaller data width for tensor data in order to save some memory space when processing the huge amount of data in a tensor. In this case, each multiplier in the TTM unit requires one DSP block, and each multiplier in the SVD unit requires two DSP blocks.

There are many sum-of-product operations in both TTM and SVD. The small error in the product terms will accumulate when calculating the sum. In order to address this issue, we use 48-bit numbers to represent the product terms. We use 27-bit numbers to represent most of the intermediate results, except for the 32-bit $\alpha, \beta, \gamma, \theta$ in the SVD unit.

B. HOOI with A Warm-start Algorithm

We observe that the SVD $\mathbf{B}^{(k)} = \mathbf{U}_k \mathbf{S}_k \mathbf{V}_k$ via the Jacoboi iterations is the most time-consuming part of HOOI. Therefore, we employ a warm-start strategy to reduce the number of Jacobi iterations. In the standard Jacobi iterations, the initial guess is chosen as an identity matrix. In our implementation, we use the orthonormal matrix $\mathbf{U}_k$ obtained from the previous iteration of HOOI as the initial guess of the Jacobi iteration. Thanks to the good initial guess, we only need to perform one or two rounds of Jacobi iterations inside each SVD, and the whole HOOI still converges after an enough number of power iterations. The optimized algorithm is shown in Alg. 3.

Algorithm 3: HOOI with warm-start Jacobi iterations

1: Initialize $\mathbf{A}_k$ as any orthonormal matrix.
2: **while** Not converge **do**
3: **for** $k = 1, 2, \ldots, d$ **do**
4: $\boldsymbol{\mathcal{B}} \leftarrow \boldsymbol{\mathcal{X}} \times_1 \mathbf{A}_1^T \cdots \times_{k-1} \mathbf{A}_{k-1}^T \times_{k+1} \mathbf{A}_{k+1}^T \cdots \times_d \mathbf{A}_d^T$
5: Unfold $\boldsymbol{\mathcal{B}}$ into $\mathbf{B}^{(k)}$
6: SVD: run Jacobi iterations (i.e., Alg. 2) with input $\mathbf{B} = \mathbf{U}_k^T \mathbf{B}^{(k)}$ and initial guess $\mathbf{U} = \mathbf{U}_k$.
7: $\mathbf{A}_k \leftarrow$ the first R_k columns of $\mathbf{U}$.
8: **end for**
9: **end while**

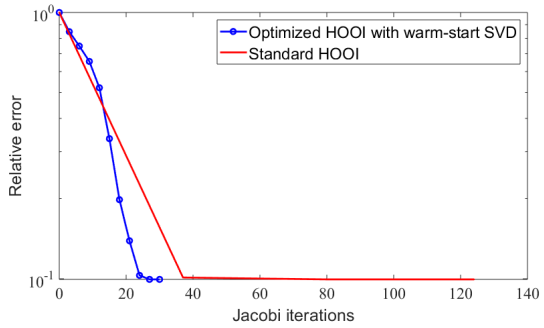


Fig. 8: Convergence speed (measured as the total number of Jacobi iterations) of a standard HOOI and the optimized HOOI with the warm-start inside SVD.

Fig. 8 shows the convergence curves of the standard HOOI and our proposed warm-start HOOI, respectively. We consider a tensor of size $128 \times 128 \times 128$ and with a multi-linear rank $\mathbf{R} = [32, 32, 32]$, which is generated by a Gaussian distribution and corrupted with some Gaussian noise. The noise variance is half of that of the tensor element. The standard method converges after only two HOOI iterations, but the SVD of each mode requires about 10 rounds of Jacobi iterations. Our optimized HOOI converges after 7 iterations, but each SVD requires only one round of Jacobi iterations, leading to a significant reduction of the total cost.

V. Performance Analysis

This section analyzes the hardware performance of our FPGA-accelerated Tucker decomposition.

A. Run-time Analysis

The total runtime is the sum of each part: TTM, SVD, and tensor permuting. For a d-way tensor, each HOOI iteration requires $d(d-1)$ TTMs, d SVDs and $2d$ tensor permuting/reshaping operations. Some intermediate results can be reused. For instance, after computing $\boldsymbol{\mathcal{X}} \times_3 \mathbf{A}_3 \times_2 \mathbf{A}_2$ for a 3-way $\boldsymbol{\mathcal{X}}$, the result of $\boldsymbol{\mathcal{X}} \times_3 \mathbf{A}_3$ can be reused when computing $\boldsymbol{\mathcal{X}} \times_3 \mathbf{A}_3 \times_1 \mathbf{A}_1$. By considering the TTM data reuse, the actual total number of TTMs is reduced to $(d-1)(1+d/2)$.

When applying the warm-start algorithm for Jacobi iteration, the unfolded matrix $\mathbf{B}^{(k)}$ need to be multiplied with $\mathbf{U}_k$ first, and this is done by TTM as well, causing additional d TTM operations. The total run-time is given by

$$T_{\text{total}} = \sum_{k=1}^{d}(\sum_{j=1}^{k} T_{\text{TTM},k,j} + T_{\text{SVD},k} + \sum_{i=1}^{2} T_{\text{permute},i,k}). \quad (11)$$

The details of each term are provided below.

TTM Part: The run-time of TTM depends on the size of its multiplier matrix. Suppose that we have a multiplier matrix of size $q \times r$. In this case, the multiplier takes in q elements of $\boldsymbol{\mathcal{X}}'$ per clock cycle, and each element is multiplied with r elements of the factor matrix $\mathbf{A}_j$. At most qr product and sum operations can be done per clock cycle. Therefore, the number of clock cycles is

$$T_{\text{TTM},k,j} = \begin{cases} I_j \prod_{j'=j+1}^{d} R_{j'} \times \lceil \frac{\prod_{j'=1}^{j-1} I_{j'}}{q} \rceil \times \lceil \frac{R_j}{r} \rceil & j > k \\ I_j I_k \prod_{j'=j+1, j' \neq k}^{d} R_{j'} \times \lceil \frac{\prod_{j'=1}^{j-1} I_{j'}}{q} \rceil \times \lceil \frac{R_j}{r} \rceil & 1 < j < k \\ I_k \prod_{j'=2, j' \neq k}^{d} R_{j'} \times \lceil \frac{I_1}{q} \rceil \times \lceil \frac{R_1}{r} \rceil & j = 1. \end{cases}$$

Similarly, the clock cycles required for $\mathbf{U}^{(k)}\mathbf{B}_k$ is

$$T_{\text{TTM},k,k} = \begin{cases} I_k \prod_{j'=k+1}^{d} R_{j'} \times \lceil \frac{\prod_{j'=1}^{k-1} R_{j'}}{q} \rceil \times \lceil \frac{I_k}{r} \rceil & k \neq 1 \\ \prod_{j'=2}^{d} R_{j'} \times \lceil \frac{I_1}{q} \rceil \times \lceil \frac{I_1}{r} \rceil & k = 1. \end{cases}$$

Some extra clock cycles are caused by the latency of the pipeline and ping-pong buffer, but they are often less than 1% of the total run-time and thus negligible. When the TTM is applied over the first mode, the data need to be copied to the local memory of each PE in advance. This causes extra $O(I_k \lceil \frac{J_k}{r} \rceil)$ clock cycles, which is negligible again and can be done in parallel with other operations.

SVD Part: When updating the k-th factor matrix, we need to do SVD to a matrix with size $I_k \times R_{/k}$, with $R_{/k} = \prod_{i \neq k} R_i$. In each Jacobi iteration, $I_k(I_k - 1)/2$ pairs of matrices will be orthogonalized. Each orthogonalization handles $2(I_k + R_{/k})$ numbers, therefore it takes $2\lceil \frac{I_k + R_{/k}}{p} \rceil$ clock cycles, where p is the degree of parallelism. As a result, each Jacobi iteration takes approximately

$$T_{\text{SVD},k} = I_k(I_k - 1)\lceil \frac{I_k + R_{/k}}{p} \rceil \quad (12)$$

clock cycles. Similar to the case in TTM, the extra time cased by the latency of pipeline is negligible.

Tensor Permutation: Suppose that the size of the buffer is $p' \times q'$. In each clock cycle, this buffer can either exchange (read or write) p' numbers with the internal memory or q' numbers with the DRAM. Note that p' and q' are not necessarily equal to p and q, respectively, as long as the memory supports writing p' or q' elements each time. However, setting $p' = p, q' = q$ can simplify our design and maximize the hardware efficiency. Each tensor permutation requires $O\left(I_k \lceil \frac{R_{/k}}{q'} \rceil + R_{/k} \lceil \frac{I_k}{p'} \rceil\right)$ clock cycles, with $R_{/k} = \prod_{i \neq k} R_i$. Our simulation shows that the practical run-time is

$$T_{\text{permute},1,k} \approx 5(I_k \lceil \frac{R_{/k}}{q'} \rceil + R_{/k} \lceil \frac{I_k}{p'} \rceil) \quad (13)$$

q	r	LUTs	Registers	DSPs	Clock rate	Power
16	16	46,056	24,556	256	212MHz	2.008W
16	32	99,384	48,357	512	200MHz	2.395W
32	16	99,505	48,189	512	203MHz	2.503W
32	32	198,269	95,743	1,024	187MHz	3.141W

TABLE I: Performance of the TTM unit.

p	LUTs	Registers	DSPs	Clock rate	Power
16	8,711	12,284	128	209MHz	0.477W
32	11,134	13,965	256	207MHz	0.683W
64	16,127	17,532	512	208MHz	1.095W
128	25,360	24,631	1,024	203MHz	1.871W

TABLE II: Performance of the SVD unit (fixed point).

q	p	LUTs	Registers	Clock rate	Power
16	64	29,929	27,718	241MHz	1.342W
32	64	59,308	55,366	209MHz	1.961W
32	128	115,749	110,662	205MHz	2.981W

TABLE III: Performance of the tensor permute/reshape unit.

clock cycles when we move $\boldsymbol{\mathcal{B}}$ from a DRAM to an on-chip memory and permute it. We need

$$T_{\text{permute},2,k} \approx 5I_k(\lceil \frac{I_k}{q'} \rceil + \lceil \frac{I_k}{p'} \rceil) \quad (14)$$

clock cycles to move $\mathbf{U}_k$ to DRAM and transpose it.

B. Area and Power

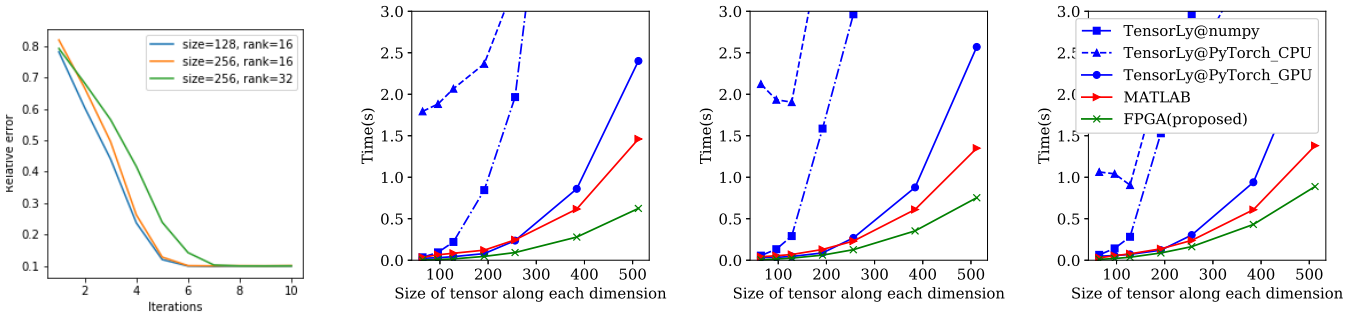
The area and power depends on the design parameters p, q and r. The higher the degree of parallelism is, the more PEs, hardware area and power will be required. In the TTM unit, the total number of multipliers, adders, accumulators and buffers are proportional to the size of multiplier matrix, $q \times r$. Therefore, the area of TTM is approximately $O(q \times r)$. Similarly, the area of the tensor permutation unit is proportional to the buffer size $p \times q$. The power also increases when the area increases.

The area of the SVD unit is independent of its input matrix size, but depends on the degree of parallelism p. Additionally, one **arctan** module and one **sin/cos** module are required. Therefore, the area of the SVD unit is estimated as $c_1 p + c_2$, where c_1 represents the area (multipliers, adders, accumulators, etc.) required for each matrix element, and c_2 represents the area of **arctan** and **sin/cos** blocks.

VI. Results

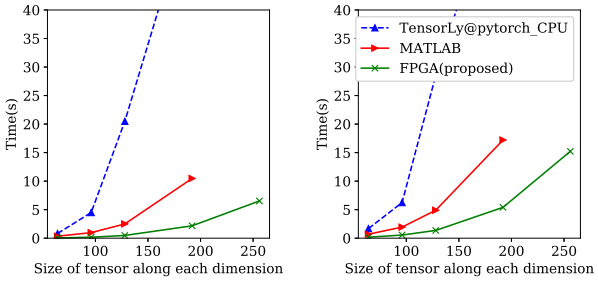
A. FPGA Performance Validation

We implement all parts of the optimized HOOI with different design parameters on FPGA. All the results below are based on Xilinx XCVU9P-L2FSGD2104E FPGA, which is available on a Xilinx VCU1525 acceleration board. The power is estimated with a 200-MHz clock rate. The results for different units are shown in Tables I-III. The hardware complexity, including the number of lookup tables (LUTs), registers and DSP blocks, is determined by the design parameters. The area of the TTM unit is approximately proportional to $q \times r$. The area of the SVD unit is approximately proportional to p. In tensor permute unit, its area is approximately proportional to $p' \times q'$. The power consumption increases as we increase the design parameters but the relationship is not strictly linear, since the power consumption of some parts (e.g., the clock generator) is independent of our design parameters.



(a) (b) (c) (d)

Fig. 9: Runtime and convergence of HOOI on some randomly generated 3-way tensors with size $I_1 = I_2 = I_3$ and $R_1 = 16, R_2 = 24, R_3 = 32$. (a) Convergence of proposed FPGA-based HOOI on a 3-way tensor, in the number of HOOI iterations. (b)-(d) Runtime of HOOI for various sizes of tensors. TensorLy uses the standard HOOI which uses SVD for initialization and for updating $\mathbf{A}_k$; the MATLAB implementation uses random initialization and eigenvalue decomposition to update $\mathbf{A}_k$; our proposed FPGA implementation uses random initialization and SVD Jacobi method with warmstart to update $\mathbf{A}_k$.



(a) (b)

Fig. 10: Runtime of HOOI on some 4-way tensors with the same size along each dimension, and rank $\mathbf{R} = [16, 16, 16, 16]$ and $\mathbf{R} = [32, 32, 32, 32]$, respectively.

B. Performance Comparisons

We compare our FPGA accelerator with other two commonly used toolboxes on different platforms: the Tensor toolbox [40], [41] in MATLAB on CPU, and the TensorLy toolbox [42] in Python on both CPU and GPU. The TensorLy can select NumPy, PyTorch or MXNet as its backend, and the latter two allow high-performance GPU computation. In our experiment, we use NumPy and PyTorch for comparison. The runtime on CPU is measured on a Linux desktop with Intel i5-8400 6-core CPU and 32 GB memory. The GPU experiments are conducted on a Titan V GPU. Since the accuracy and convergence of our Tucker decomposition depends on the fixed-point number system, we develop a fixed-point simulator with the Xilinx fixed-point number library to simulate the truncation error and overflow in our FPGA accelerator.

We perform the comparison by using some randomly generated low-rank tensors. For each tensor, both the core tensor and the factor matrices are generated by Gaussian distributions, and the tensor is then corrupted by some Gaussian random noise. The relative error is evaluated as

$$\frac{\|\boldsymbol{\mathcal{X}} - \boldsymbol{\mathcal{G}} \times_1 \mathbf{A}_1 \times_2 \mathbf{A}_2 \times \cdots \times_d \mathbf{A}_d\|_{\mathrm{F}}}{\|\boldsymbol{\mathcal{A}}\|_{\mathrm{F}}} \times 100\%. \qquad (15)$$

Fig. 9 shows the results on a set of 3-way tensors with size $I_1 = I_2 = I_3$. Our architecture can get $1.41 - 9.90\times$ speedup compared with MATLAB tensor toolbox on CPU, and even more compared with the TensorLy toolbox on both CPU and GPU.The convergence behavior of our FPGA-based Tucker decomposition is shown in Fig. 9(a). It is shown that our method always converges after 6-8 HOOI iterations. Therefore, we assume 8 HOOI iterations to estimate the runtime of our FPGA architecture.

We further perform comparisons using some 4-way tensors and show the results in Fig. 10. Our PC with 32GB RAM can no longer accommodate such 4-way tensor data, therefore the results on CPU are obtained by running our experiments on Amazon AWS r4.4x large instance with 16 virtual CPUs (vCPU) and 122GB memory. Since large 4-way tensors exceed the memory of GPU, and TensorLy with the NumPy backend consumes extremely long time, their runtimes are not shown in Fig. 10. When the size along each dimension is 256, MATLAB run out of memory when computing. On these 4-way tensor data, our FPGA design can get $3.18-8.22\times$ speedup compared with the MATLAB tensor toolbox on the AWS CPU.

Remark: Our FPGA accelerator uses the Jacobi iteration to perform SVD, whereas the MATLAB tensor toolbox and TensorLy use more powerful advanced SVD algorithms. If the same SVD algorithms are used in all implementations, our FPGA design may achieve more significant speedup.

C. Application: MRI Compression

The accelerated Tucker decomposition can be applied to multiple application domains. Here we demonstrate its application in compressing multi-way MRI data. This dataset is a single-channel, first-pass myocardial perfusion real-time MRI sequence ($n_x = 190, n_y = 90, n_t = 70$). We use the pre-processed data available in [43], [44], and set rank to $R_x = 40, R_y = 32, R_t = 28$ in HOOI. The estimated runtime with our architecture is 0.0447s at a clock rate of 185MHz. In comparison, on a Linux PC with Intel i5-8500 CPU 6 core CPU, the same operation takes 0.0964s with the MATLAB tensor toolbox, 0.335s with TensorLy toolbox and NumPy backend, 1.352s with TensorLy toolbox and Pytorch backend. on a PC with NVIDIA TITAN V GPU, this operation takes 0.217s. Therefore, our method provides 2.16-30.2× speedup compared with existing frameworks on CPU, and 4.85× speedup compared with GPU. Fig. 11 shows the original MRI data and the data approximated by our low-rank Tucker decomposition on FPGA.

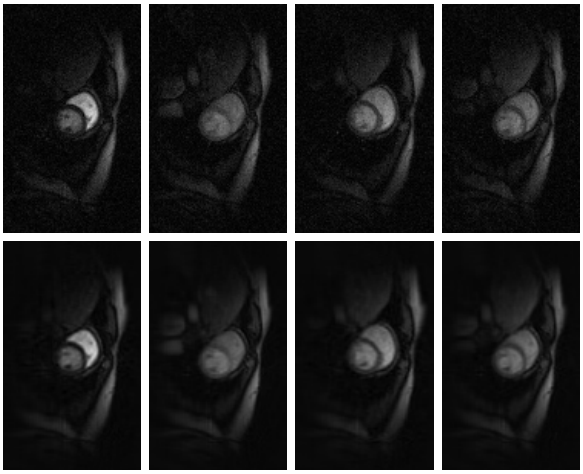


Fig. 11: Decomposition result of MRI dataset. Top: original $190 \times 90 \times 70$ data at $t = 15, 30, 45, 60$. Bottom: approximated data by Tucker decomposition on FPGA, with rank $\mathbf{R} = [40, 32, 28]$. The data compression ratio is $24.8\times$.

VII. Conclusion

This paper has presented an algorithm-architecture co-design to perform tensor Tucker decomposition. We have implemented Tensor-Times-Matrix, matrix SVD with single side Jacobi iteration, and tensor permuting on FPGA. We have also proposed a warm-start algorithm for the Jacobi iterations to reduce its computation time. We have done simulations on both synthetic data sets and an MRI data set. Our method is significantly faster than existing computation frameworks on both CPU and GPU. This accelerator can be employed in a broad range of applications including data mining, scientific computing, computer vision, and deep learning.

Acknowledgment

This work was supported by NSF CCF 1817037. We would like to thank Sophia Shao (NVIDIA), Lei Deng, Zheng Qu and Bangyan Wang (UCSB) for their technical discussions.

References

[1] T. Kolda and B. Bader, "Tensor decompositions and applications," *SIAM Rev*, vol. 51, no. 3, pp. 455–500, 2009.

[2] F. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *J. Math. Phys.*, vol. 6, no. 1-4, pp. 164–189, 1927.

[3] L. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[4] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal Sci. Comp.*, vol. 33, no. 5, pp. 2295–2317, 2011.

[5] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *Proc. IEEE Int. Conf. Data Mining*, 2008, pp. 363–372.

[6] C. Hawkins and Z. Zhang, "Bayesian tensorized neural networks with automatic rank selection," *arXiv preprint arXiv:1905.10478*, 2019.

[7] H. Ding, K. Chen, Y. Yuan, M. Cai, L. Sun, S. Liang, and Q. Huo, "A compact CNN-DBLSTM based character model for offline handwriting recognition with tucker decomposition," in *Proc. IEEE Int. Conf. Document Analysis and Recognition*, vol. 1, 2017, pp. 507–512.

[8] J.-T. Chien and Y.-T. Bao, "Tensor-factorized neural networks," *IEEE Trans. Neur. Networks Learn. Syst.*, vol. 29, no. 5, pp. 1998–2011, 2018.

[9] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *NIPS*, 2015, pp. 442–450.

[10] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned CP-decomposition," *arXiv preprint arXiv:1412.6553*, 2014.

[11] Y. Yang, D. Krompass, and V. Tresp, "Tensor-train recurrent neural networks for video classification," in *Proc. Int. Conf. Machine Learning*, 2017, pp. 3891–3900.

[12] Z. Zhang, X. Yang, I. V. Oseledets, G. E. Karniadakis, and L. Daniel, "Enabling high-dimensional hierarchical uncertainty quantification by anova and tensor-train decomposition," *IEEE Trans. CAD of Integrated Circuits and Systems*, vol. 34, no. 1, pp. 63–76, 2015.

[13] Z. Zhang, T.-W. Weng, and L. Daniel, "Big-data tensor recovery for high-dimensional uncertainty quantification of process variations," *IEEE Trans. Comp., Pack. Manuf. Tech.*, vol. 7, no. 5, pp. 687–697, 2017.

[14] S. F. Roohi, D. Zonoobi, A. A. Kassim, and J. L. Jaremko, "Dynamic mri reconstruction using low rank plus sparse tensor decomposition," in *Proc. Int. Conf. Image Process.*, 2016, pp. 1769–1773.

[15] J. Li, G. Han, J. Wen, and X. Gao, "Robust tensor subspace learning for anomaly detection," *Int. J. Machine Learning and Cybernetics*, vol. 2, no. 2, pp. 89–98, 2011.

[16] D. Saito, K. Yamamoto, N. Minematsu, and K. Hirose, "One-to-many voice conversion based on tensor representation of speaker space," in *Proc. Int. Conf. Speech Comm. Assoc.*, 2011.

[17] O. Kaya and B. Uçar, "High performance parallel algorithms for the Tucker decomposition of sparse tensors," in *Proc. IEEE Int. Conf. Parall. Proc.*, 2016, pp. 103–112.

[18] S. Smith, J. Park, and G. Karypis, "Sparse tensor factorization on many-core processors with high-bandwidth memory," in *Proc. IEEE Int. Parallel and Distributed Processing Symp*, 2017, pp. 1058–1067.

[19] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc, "An input-adaptive and in-place approach to dense tensor-times-matrix multiply," in *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[20] A. Amira, A. Bouridane, and P. Milligan, "Accelerating matrix product on reconfigurable hardware for signal processing," in *Proc. Int. Conf. Field Programmable Logic and Applications*, 2001, pp. 101–111.

[21] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proc. Int. Symp. Field-programmable Gate Arrays*, 2005, pp. 86–95.

[22] J. Zhu and P. Sutton, "FPGA implementations of neural networks–a survey of a decade of progress," in *Proc. FPLA*, 2003, pp. 1062–1066.

[23] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Trans. CAD of Integr. Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2017.

[24] K. Irick, M. DeBole, V. Narayanan, and A. Gayasen, "A hardware efficient support vector machine architecture for FPGA," in *Proc. Int. Symp. FPCCM*, 2008, pp. 304–305.

[25] J. L. Stanislaus and T. Mohsenin, "Low-complexity FPGA implementation of compressive sensing reconstruction," in *Int. Conf. Comput., Netw. Comm.*, 2013, pp. 671–675.

[26] M. A. O. Vasilescu and D. Terzopoulos, "Multilinear image analysis for facial recognition," in *Object recognition supported by user interaction for service robots*, vol. 2. IEEE, 2002, pp. 511–514.

[27] N. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Trans. Sign. Proc.*, vol. 65, no. 13, pp. 3551–3582, 2017.

[28] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint arXiv:1511.06530*, 2015.

[29] L. De Lathauwer, B. De Moor, and J. Vandewalle, "On the best rank-1 and rank-(r_1, r_2,..., r_n) approximation of higher-order tensors," *SIAM J. Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1324–1342, 2000.

[30] L. R. Tucker, "Implications of factor analysis of three-way matrices for measurement of change," *Problems in Measuring Change*, vol. 15, pp. 122–137, 1963.

[31] P. M. Kroonenberg and J. De Leeuw, "Principal component analysis of three-mode data by means of alternating least squares algorithms," *Psychometrika*, vol. 45, no. 1, pp. 69–97, 1980.

[32] A. Kapteyn, H. Neudecker, and T. Wansbeek, "An approach ton-mode components analysis," *Psychometrika*, vol. 51, no. 2, pp. 269–275, 1986.

[33] E. R. Hansen, "On cyclic Jacobi methods," *J. Soc. Indust. and Appl. Math.*, vol. 11, no. 2, pp. 448–459, 1963.

[34] J. Demmel and K. Veselić, "Jacobi's method is more accurate than QR," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 4, pp. 1204–1245, 1992.

[35] N. Hemkumar and J. Cavallaro, "A systolic VLSI architecture for complex SVD," in *Proc. IEEE ISCAS*, vol. 3, 1992, pp. 1061–1064.

[36] A. Ahmedsaid, A. Amira, and A. Bouridane, "Improved SVD systolic array and implementation on FPGA," in *Proc. FPL*, 2003, pp. 35–42.

[37] M. Rahmati, M. S. Sadri, and M. A. Naeini, "FPGA based singular value decomposition for image processing applications," in *IEEE Intl. Conf. ASSAP*, 2008, pp. 185–190.

[38] R. Brent and F. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays," *SIAM J. Sci. Stat. Comp.*, vol. 6, no. 1, pp. 69–84, 1985.

[39] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Computers*, no. 3, pp. 330–334, 1959.

[40] B. Bader, T. Kolda *et al.*, "Matlab tensor toolbox version 2.6," Available online, February 2015. [Online]. Available: http://www.sandia.gov/~tgkolda/TensorToolbox/

[41] B. W. Bader and T. G. Kolda, "Algorithm 862: MATLAB tensor classes for fast algorithm prototyping," *ACM Trans. Math. Software*, vol. 32, no. 4, pp. 635–653, Dec 2006.

[42] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, "TensorLy: Tensor learning in python," *CoRR*, vol. abs/1610.09555, 2018.

[43] "First-pass myocardial perfusion real-time MRI dataset," https://statweb.stanford.edu/~candes/SURE/matlab/JDT/DATA/invivo_perfusion4.mat, accessed: 2019-03-19.

[44] S. Lingala, Y. Hu, E. DiBella, and M. Jacob, "Accelerated dynamic MRI exploiting sparsity and low-rank structure: k-t SLR," *IEEE Trans. Medical Imaging*, vol. 30, no. 5, pp. 1042–1054, 2011.