Sparse Tucker Tensor Decomposition on a Hybrid FPGA–CPU Platform

Weiyun Jiang, Kaiqi Zhang, Colin Yu Lin, Feng Xing, and Zheng Zhang, *Member, IEEE*

***Abstract*—Recommendation systems, social network analysis, medical imaging, and data mining often involve processing sparse high-dimensional data. Such high-dimensional data are naturally represented as tensors, and they cannot be efficiently processed by conventional matrix or vector computations. Sparse Tucker decomposition is an important algorithm for compressing and analyzing these sparse high-dimensional datasets. When energy efficiency and data privacy are major concerns, hardware accelerators on resource-constraint platforms become crucial for the deployment of tensor algorithms. In this work, we propose a hybrid computing framework containing CPU and FPGA to accelerate sparse Tucker factorization. This algorithm has three main modules: 1) tensor-times-matrix (TTM); 2) Kronecker products; and 3) QR decomposition with column pivoting (QRP). In addition, we accelerate the former two modules on a Xilinx FPGA and the latter one on a CPU. Our hybrid platform achieves 23.6× ∼ 1091× speedup and over 93.519% ∼ 99.514% energy savings compared with CPU on the synthetic and real-world datasets.**

***Index Terms*—Field programmable gate arrays, high level synthesis, high performance computing, neural network hardware.**

I. Introduction

As MASSIVE data is collected from social media, wearable devices, and Internet of Things, novel algorithms and platforms are highly desired to handle data-intensive computing tasks. Vector- and matrix-based methods can efficiently process 1-way data (e.g., a sequence of voice data) or 2-way data (e.g., a gray-scale image), but they are often inefficient to handle multiway data. Representative examples includes 3-way (or order-3) E-commerce data (which records customers' preference on massive products over a few months), 4-way (or order-4) cardiac image data (which records the spatial data of 3-D at multiple time points). Processing such multiway data often suffers from the curse of dimensionality.

Tensors are a high-order generalization of matrices and vectors, and they are a natural tool to represent and process multiway data [1]. Leveraging various tensor decomposition or factorization methods [1]–[4], the curse of dimensionality of storing and computing multiway data can be avoided or significantly mitigated in many applications. For instance, the canonical polyadic (CP) [5], [6] and tensor-train [2] factorizations can reduce the storage cost and unknown variables from an exponential function to a linear one. Tucker factorization [3] can be used for high-order principle component analysis or facial recognition [7]–[9]. Tensor computation has achieved tremendous success in data mining [10], computer vision [7]–[9], medical imaging [11], electronic design automation [12]–[15], and deep learning [16]–[18].

The emerging tensor computation concept brings in massive research opportunities and challenges on the hardware level. Due to the fundamental difference between tensor and matrix computations, we may need to rethink many aspects of tensor computation (e.g., storage, computing, and data movement) on specific platforms. Increasing research results have been reported to improve the tensor data storage and computing on the cloud and high-performance clusters [19]–[21]. However, little work has been done on resource-constrained platforms. This becomes increasingly important as the need of energy-efficient machine learning and data privacy surges. In order to address these issues, some efforts have been made toward tensor-compressed neural networks on mobile devices [22] and dense tensor operations on FPGA. For instance, some dense tensor operations, including MTTKRP, tensor-times-matrix (TTM), and TTMc, were accelerated in [23]; a spectral analysis of Hankel tensors was reported in [24]. To perform dense Tucker decomposition on FPGA, Zhang *et al.* [25] divided the hardware architectures into three modules: 1) TTM; 2) singular value decomposition (SVD) via Jacobi iterations; and 3) tensor permutation/reshaping. In addition, a warm-start algorithm was used to reduce the cost of Jacobi iterations. The resulting FPGA accelerator demonstrated significant speedup compared with both CPU and GPU. However, the FPGA accelerator [25] cannot exploit data sparsity, and it becomes energy- and time-inefficient when dealing with sparse tensors. Srivastava *et al.* [26] reported some sparse tensor computation kernels. For instance, it demonstrated how to implement both dense and sparse tensor operations, such as sparse TTMc via sparse compute pattern SF^3. To the best of our knowledge, there is no FPGA accelerator available for sparse Tucker decomposition.

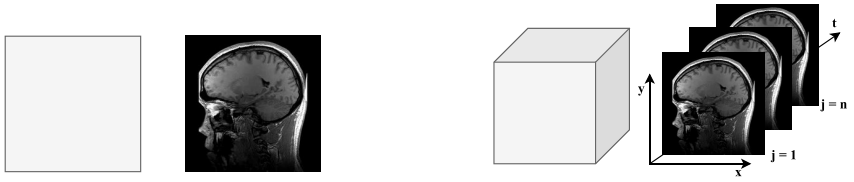
In this article, we investigate the hardware acceleration of Tucker factorization for *sparse* tensor data. Sparse tensors widely appear in practice due to the missing information in recommendation systems, medical image, or E-commerce data. For instance, in magnetic resonance imaging (MRI), one

Manuscript received May 14, 2020; revised August 20, 2020; accepted September 28, 2020. Date of publication October 20, 2020; date of current version August 20, 2021. This work was supported by NSF under Award 1817037. This article was recommended by Associate Editor W. Zhang. *(Corresponding author: Weiyun Jiang.)*

Weiyun Jiang, Kaiqi Zhang, and Zheng Zhang are with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 USA (e-mail: weiyunjiang@ucsb.edu; kzhang70@ucsb.edu; and zhengzhang@ece.ucsb.edu).

Colin Yu Lin and Feng Xing are with the Data Center Group, Xilinx, Beijing 100101, China (e-mail: yulin1@xilinx.com; fengx@xilinx.com).

Digital Object Identifier 10.1109/TCAD.2020.3032626



(a) (b)

Fig. 1. (a) Matrix is a 2-D data array (e.g., one slice of MRI data). (b) 3-way tensor is a 3-D data array (e.g., multiple slices of images).

can generate a sparse tensor by partial MRI scanning, then reconstruct the whole image with a low cost [27]. In neuroscience, researchers use sparse tensors to monitor the brain variability [28]. In EDA, it is often too expensive to obtain all simulation or measurement data, thus one uses a partially sampled sparse tensor for process variation or performance uncertainty prediction [12], [14], [15]. Although extensive algorithms have been developed to process sparse tensors, their hardware/algorithm co-optimization remains a rarely explored field [25]. This task has become increasingly important as energy efficiency and privacy cause lots of concerns in the data science and machine learning community.

A. Article Contributions and Organization

This article proposes to design an energy- and memory-efficient hybrid FPGA–CPU accelerator for sparse Tucker decomposition [19]. This algorithm consists of three major components: 1) TTM [1]; 2) Kronecker product [29]; and 3) QR decomposition with column pivoting (QRP) [30]. Our specific contributions include the following.

1) On the hardware side, we present a high-level synthesis (HLS) FPGA implementation for sparse Tucker decomposition. We describe the design of two modules, TTM and Kronecker product, by exploiting the data sparsity.
2) On the algorithm side, we replace the conventional SVD [31] with the QRP [30] to reduce the data storage cost and to speedup the computation.
3) We implement our FPGA accelerator in a Xilinx FPGA on Amazon Web service (AWS). Then we compare our hybrid FPGA–CPU accelerator with CPU and with the recently developed dense FPGA accelerator [25] on synthetic and real-world sparse tensor benchmarks. Our hybrid FPGA–CPU accelerator achieves $1.15\times\sim1091\times$ speedup and consumes $93.519\% \sim 99.514\%$ less energy. In addition, our proposed accelerator achieves significant speedup ($23.6\times\sim 1091\times$) when the tensor is very large and sparse

This article is organized as follows. Section II introduces some background information about tensor operations. Section III presents the algorithm and our Vivado HLS FPGA design of a sparse Tucker decomposition. We compare our FPGA/CPU hybrid platform with CPU and the dense Tucker FPGA accelerator [25] in terms of runtime and energy efficiency in Section IV. Finally, Section V concludes this article.

II. Preliminaries of Tensors

This section presents some background about tensors, which is necessary for understanding the ideas of this article.

Definition 1: A tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ is a high-dimensional array of order N. Here, the order N (also known as "way") is the total number of dimensions. A matrix $\mathbf{X} \in \mathbb{R}^{n_1\times n_2}$ is a second-order (or 2-D) tensor, and its element indexed by (i_1, i_2) can be denoted as $x_{i_1 i_2}$. For a general Nth-order (or N-way) tensor $\boldsymbol{\mathcal{X}}$, its element indexed by $(i_1, i_2, \ldots, i_N)$ is denoted as $x_{i_1 i_2\cdots i_N}$.

Fig. 1 shows a matrix (e.g., one slice of MRI data) and a 3-way tensor, respectively. In this article, we use boldface lower-case letters (e.g., $\mathbf{x}$) to denote vectors, boldface upper-case letters (e.g., $\mathbf{X}$) to denote matrices, and boldface Euler script letters (e.g., $\boldsymbol{\mathcal{X}}$) to denote tensors. A scalar is denoted by a lower-case letter, e.g., x.

Definition 2: The inner product of two tensors with the same size is defined as

$$\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}} \rangle = \sum_{i_1 i_2\cdots i_N} x_{i_1 i_2\cdots i_N} y_{i_1 i_2\cdots i_N}. \tag{1}$$

Furthermore, the Frobenius norm (also known as F-norm) of a tensor $\boldsymbol{\mathcal{X}}$ is defined as $||\boldsymbol{\mathcal{X}}||_{\mathrm{F}} = \sqrt{\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}} \rangle}$.

Definition 3: A matricization operation (also known as unfolding or flattening) reshapes a tensor into a matrix. The mode-n matricization of a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ is denoted as $\mathbf{X}_{(n)}$ which has I_n rows and $\prod_{k\neq n} I_k$ columns. Elementwise, we have each entry of $\mathbf{X}_{(n)}$ as

$$\mathbf{X}_{(n)}(i_n, j) = x_{i_1 i_2\cdots i_N} \quad \text{with } j = 1 + \sum_{k=1, k\neq n}^{N} (i_k - 1) \prod_{m=1, m\neq n}^{k-1} I_m. \tag{2}$$

Definition 4: The mode-$\mathbf{n}$ tensor matrix product (or TTM), between a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ and a matrix $\mathbf{U} \in \mathbb{R}^{J\times I_n}$ is denoted as

$$\boldsymbol{\mathcal{G}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{U}, \text{ where } \boldsymbol{\mathcal{G}} \in \mathbb{R}^{I_1\times\cdots\times I_{n-1}\times J\times I_{n+1}\times\cdots\times I_N}. \tag{3}$$

Elementwise, we can write this operation as

$$g_{i_1\cdots_{n-1} j i_{n+1}\ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2\ldots i_N} u_{j i_n}. \tag{4}$$

We may also obtain a TTM product by using the unfolded tensors

$$\boldsymbol{\mathcal{G}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{U} \Leftrightarrow \mathbf{G}_{(n)} = \mathbf{U}\mathbf{X}_{(n)}. \tag{5}$$

We further introduce a matrix operation that will be used in our subsequent tensor computation.

Algorithm 1 Standard HOOI for Tucker Decomposition

1: Initialize $\{\mathbf{U}_n\}_{k=1}^N$ via HOSVD
2: **while** not converge **do**
3: **for** $n = 1, 2, \ldots, N$ **do**
4: $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}_1^T \cdots \times_{n-1} \mathbf{U}_{n-1}^T \times_{n+1} \mathbf{U}_{n+1}^T \cdots \times_N \mathbf{U}_N^T$
5: Unfold $\mathcal{Y}$ and perform SVD: $\mathbf{Y}_{(n)} = \mathbf{USV}^T$
6: $\mathbf{U}_n \leftarrow$ the first R_n columns of $\mathbf{U}$.
7: **end for**
8: **end while**
9: **return** $\{\mathbf{U}_n\}_{n=1}^N$.

TABLE I
COORDINATE (COO) FORMAT OF A $5 \times 5 \times 5 \times 5$ SPARSE TENSOR. HERE, (i, j, k, l) DENOTES AN INDEX, AND nnz IS THE VALUE OF AN ASSOCIATED NONZERO DATA ELEMENT

i	j	k	l	nnz
1	1	1	1	2
1	1	1	5	7.5
1	1	3	5	4
2	2	2	4	5

Definition 5: Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and another matrix $\mathbf{B} \in \mathbb{R}^{p \times q}$, their Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is the following matrix $\mathbf{C} \in \mathbb{R}^{mp \times nq}$

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}. \tag{6}$$

III. ACCELERATOR FOR SPARSE TUCKER DECOMPOSITION

Given a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, the Tucker decomposition [4] approximates it with a small low-rank core tensor $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}$ and N factor matrices $\{\mathbf{U}_n \in \mathbb{R}^{I_n \times R_n}\}_{n=1}^N$

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{U}_1 \times_2 \mathbf{U}_2 \cdots \times_N \mathbf{U}_N. \tag{7}$$

Here $(R_1, R_2, \ldots, R_N)$ is a multilinear tensor rank.

The Tucker decomposition can be regarded as a high-order generalization of SVD, and it is often implemented with the power iteration method called high-order orthogonal iteration (HOOI) in [4]. As shown in Algorithm 1, it aims to find the orthogonal matrices $\{\mathbf{U}_n \in \mathbb{R}^{I_n \times R_n}\}_{n=1}^N$ to maximize the F-norm of

$$\mathcal{G} = \mathcal{X} \times_1 \mathbf{U}_1^T \times_2 \mathbf{U}_2^T \cdots \times_N \mathbf{U}_N^T. \tag{8}$$

In every iteration, we need to compute the R_n dominant left singular vectors of unfolded matrix $\mathbf{Y}_{(n)}$, where

$$\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}_1^T \cdots \times_{n-1} \mathbf{U}_{n-1}^T \times_{n+1} \mathbf{U}_{n+1}^T \cdots \times_N \mathbf{U}_N^T. \tag{9}$$

The orthogonal matrix is obtained by an SVD of the unfolded matrix $\mathbf{Y}_{(n)}$.

The standard HOOI becomes very inefficient for sparse tensors because line 4 of Algorithm 1 does not exploit any data sparsity and always performs $N-1$ times of TTM operations.

Algorithm 2 Sparse Tucker Decomposition

Input: A sparse tensor $\mathcal{X}$
$R_1, \ldots, R_N$: rank of approximation
initialize $\mathbf{U_1}, \ldots, \mathbf{U_N}$ randomly.
repeat
for $n = 1, 2, \ldots, N$ **do**
for $x_{i_1,\ldots,i_N} \neq 0$ **do**
$\mathbf{Y}_{(n)}(i_n, :) \mathrel{+}= x_{i_1,\ldots,i_N}[\otimes_{t \neq n} \mathbf{U}_t(i_t, :)]$
end for
$\mathbf{U}_n \leftarrow \mathbf{QRP}(\mathbf{Y}_{(n)}, R_n)$
end for
$\mathcal{G} \leftarrow \mathcal{Y} \times_N \mathbf{U}_N^T$
until convergence or maximum number of iterations reached
Output:
$\mathcal{G}$: a $R_1 \times \ldots \times R_N$ core tensor
$\mathbf{U_1}, \ldots, \mathbf{U_N}$: $\mathbf{U}_n$ is a $R_n \times I_n$ factor matrix

A. Overall Algorithm Flow

In this article, we design an FPGA–CPU hybrid accelerator based on [19] to perform Tucker factorization for sparse tensors. Two formats can be used to represent sparse tensors.

1) The coordinate format (COO) stores a sparse tensor with all nonzero elements and their associated coordinate vectors, shown in Table I. The first four columns represent the coordinate (i, j, k, l) of four nonzero elements, and the last column represents the corresponding value. The COO format usually requires storage of $O(nnz*N)$ index values and $O(nnz)$ nonzero data values, where nnz is the number of nonzero elements and N is the mode of the tensor.
2) Compressed sparse fiber format (CSF) stores a sparse tensor by compressing the indices of nonzero elements that share the same coordinates. It is regarded as high-dimensional version of the compressed sparse row (CSR) or compressed sparse column (CSC) formats used for matrices in [32]. The CSF format requires $O(2 * (nnz + s + f) + 2)$ to store an order-3 tensor with s slices, f fibers, and nnz nonzero values.

In this article, we use the COO format because of its flexibility and simplicity. Furthermore, the COO format provides better performance on merging-related TTM [33]. If we do not assume any special structure of the tensor and the nonzero elements are uniformly distributed, there will be rarely multiple nonzero elements in a given fiber. In such a general case, the CSF format barely has any advantages in storage compression.

The algorithm flow is summarized in Algorithm 2. Compared with the standard dense Tucker factorization, the following techniques are used to exploit the data sparsity.

1) Instead of storing the whole tensor, we only store the nonzero entries by specifying their values and indices.
2) When performing the TTM in (9), we do not perform $N-1$ levels of iterations over all modes except mode n. Instead, we only consider the nonzero elements of $\mathcal{X}$ and have a one-level iteration over the indices of all nonzero elements in $\mathcal{X}$.

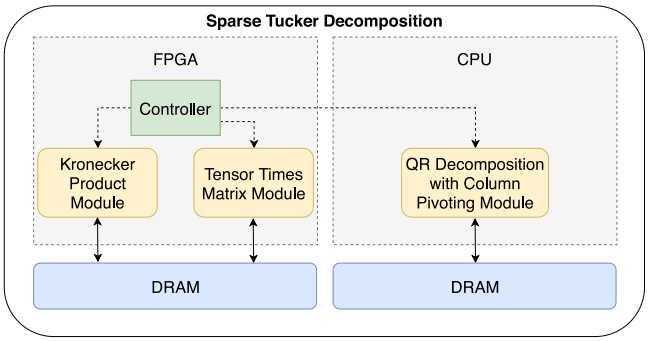


Fig. 2. Hybrid FPGA–CPU platform for sparse Tucker factorization.

3) In order to reduce the computational and memory cost of extracting orthogonal matrix factor $\mathbf{U}_n$, we replace the SVD of $\mathbf{Y}_{(n)}$ with a QRP.

The proposed accelerator architecture is shown in Fig. 2. Because it is difficult to parallelize the QRP operation, we implement it on CPU. Both (8) and (9) require TTM operations, but they are handled in different ways. For (8), we only need to compute

$$\mathcal{G} = \mathcal{Y} \times_N \mathbf{U}_N \tag{10}$$

once for each iteration after obtaining $\mathcal{Y}$ (which is often dense) by (9). Therefore, we design a specialized TTM module on FPGA in Section III-B. For the power iteration in (9), we design a Kronecker product module on FPGA to accelerate the sparse operation, which is detailed in Section III-C.

B. Tensor-Times-Matrix on FPGA

The computation of $\mathcal{G}$ in (8) requires N tensor-matrix products on the original huge-size tensor $\mathcal{X}$. This expensive computation actually can be simplified.

Assuming that we have already done the power iteration (9) for $n = N$ and obtained a small-size tensor $\mathcal{Y} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times I_N}$ and an orthogonal factor matrix $\mathbf{U}_N \in \mathbb{R}^{I_N \times R_N}$. We only need to compute the mode-N tensor-matrix product (10) to obtain the core tensor $\mathcal{G}$ (line 9, Algorithm 2). This TTM can be written in an elementwise manner:

$$\left(\mathcal{Y} \times_N \mathbf{U}_N^T\right)_{r_1 r_2 \ldots r_N} = \sum_{i_N=1}^{I_N} y_{r_1 r_2 \ldots i_N} \mathbf{U}_N(i_N, r_N). \tag{11}$$

Equivalently, we can express this particular TTM with unfolded tensors as follows:

$$\mathcal{G} = \mathcal{Y} \times_N \mathbf{U}_N^T \Leftrightarrow \mathbf{G}_{(N)} = \mathbf{U}_N^T \mathbf{Y}_{(N)}. \tag{12}$$

Here $\mathcal{G}_{(N)}$ and $\mathbf{Y}_{(N)}$ are the mode-N unfolding of the tensors $\mathcal{G}$ and $\mathcal{Y}$, respectively.

In FPGA design, the 3-D sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ is stored with a cost $O(nnz)$, where nnz denotes the number of nonzero elements. However, the tensor $\mathcal{Y} \in \mathbb{R}^{R_1 \times R_2 \times I_3}$ in (10) is dense, and we need to store all of its elements. Although $\mathcal{Y}$ is multidimensional, it is unnecessary to create a new copy of this tensor. We can just reshape it into a 2-D matrix of size $R_1 R_2 \times I_3$. Meanwhile, it is critical to process the entries of $\mathcal{Y}$ in several batches. The batch size, b, controls the number of entries in $\mathcal{Y}$, being processed in each iteration. If we set the batch size as $b = R_1 R_2$, we will end up with three nested for-loops because the outermost for-loop is redundant. As a result, all the entries of $\mathcal{Y}$ have to be processed at the same time, resulting in an extremely large amount of loop unrolling, which is not practical when $R_1 R_2$ is larger. To overcome this issue, we decrease our batch size to 32 and separate this loop into two parts, resulting in four nested for-loops to compute the resultant tensor of the TTM. In this way, we could achieve optimal loop unrolling on memory-constrained FPGAs.

Algorithm 3 Vivado HLS Implementation of TTM on 3-Way Tensors

Require: $\mathbf{Y} \in \mathbb{R}^{R_1 R_2 \times I_3}$, $\mathbf{U} \in \mathbb{R}^{R_3 \times I_3}$
  $\ell = R_1 R_2$, $b = 32$
  for $(i_b = 0; i_b < \ell; i_\delta \mathrel{+}= b)$ **do**
    initialize tmp as zero
    for $(k = 0; k < R_3; k{+}{+})$ **do**
      for $(i_o = 0; i_o < b; i_o{+}{+})$ **do**
        for $(t = 0; t < I_3; t{+}{+})$ **do**
          $\mathbf{tmp}[i_o, k] \mathrel{+}= \mathbf{Y}[i_o + i_b, t] * \mathbf{U}[k, t]$
        end for
      end for
    end for
    for $(k = 0; k < R_3; k{+}{+})$ **do**
      for $(i_o = 0; i_o < b; i_o{+}{+})$ **do**
        $\mathbf{G}[i_o + i_b, k] = \mathbf{tmp}[i_o, k]$
      end for
    end for
  end for
Output: $\mathbf{G} \in \mathbb{R}^{R_1 R_2 \times R_3}$

We provide the Vivado HLS implementation pseudocode of the TTM for a 3-way tensor $\mathcal{X}$ in Algorithm 3. Given a 3-way tensor, $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, (10) is a mode-3 TTM between $\mathcal{Y} \in \mathbb{R}^{R_1 \times R_2 \times I_3}$ and $\mathbf{U} \in \mathbb{R}^{I_3 \times R_3}$, where $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ is the result. In the pseudocode, we reshape our tensors $\mathcal{Y} \in \mathbb{R}^{R_1 \times R_2 \times I_3}$ and $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ into matrices $\mathbf{Y} \in \mathbb{R}^{R_1 R_2 \times I_3}$ and $\mathbf{G} \in \mathbb{R}^{R_1 R_2 \times R_3}$. Basically, we divide our result, $\mathbf{G}$, into several portions such that we can update one portion of $\mathbf{G}$ in each batch.

1) We initialize the temporary matrix, **tmp** as zero matrix of size $b \times R_3$, where b is the batch size. This temporary matrix stores one portion of our result $\mathbf{G}$.
2) We compute TTM by multiplying unfolded tensor $\mathbf{Y}$ and $\mathbf{U}$ based on (12) and store the results in **tmp**.
3) We just update one portion of $\mathbf{G}$ with **tmp**.

In order to optimize the Vivado HLS implementation, we reshape $\mathbf{U}$ in cyclic forms by a factor of 8, and we reshape $\mathbf{Y}$ and **tmp** in cyclic forms by a factor of 16. Furthermore, in order to save RAM usage, we assign only one port of RAM to the variables, $\mathbf{Y}$, $\mathbf{U}$, and **tmp**. We also assign the intermediate variable **tmp** to registers instead of memory to minimize memory usage.

Fig. 3 shows the dataflow in the TTM computation module on FPGA. According to the elementwise formula (11), each entry of the resultant tensor can be recognized as the sum of

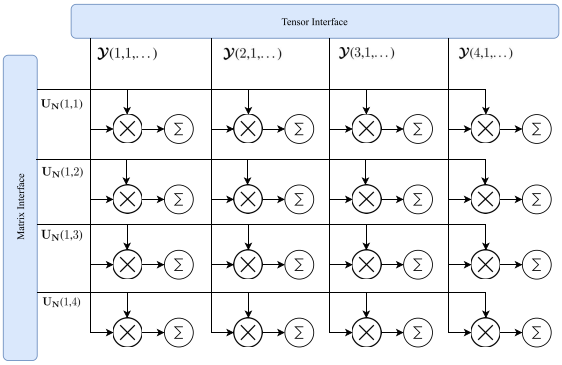


Fig. 3. TTM dataflow.

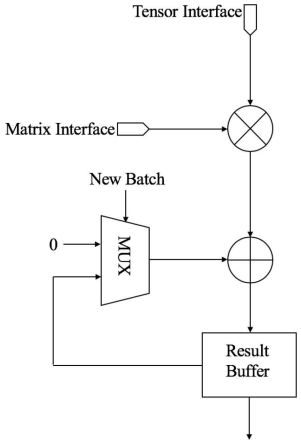


Fig. 4. TTM PE [25].

product between the entries from the original tensor $\mathcal{Y}$ and the entries from the matrix $\mathbf{U_N}$. In Fig. 3, it shows that data from the tensor interface, $y_{r_1 r_2 \ldots i_N}$ multiplies with the data from the matrix interface, $\mathbf{U_N}(i_N, r_N)$. After the multiplication, the results are summed up to obtain the entries in the resultant tensor, $(\mathcal{Y} \times_N \mathbf{U}_N^T)_{r_1 r_2 \ldots r_N}$.

A detailed dataflow of the processing element (PE) for TTM is shown in Fig. 4, which was proposed in [25]. A buffer temporarily stores the intermediate result after multiplying the tensor and the matrix. For each batch, the multiplexer selects and adds the intermediate result to the new product. Once all batches are processed, the final result is stored the DRAM.

C. Kronecker Products on FPGA

The power iteration (9) requires $O(R^d \times n)$ operations, and it consumes most of the computational power and runtime in the sparse Tucker decomposition. Although an FPGA design was presented in [25] to accelerate power iterations, existing design cannot handle sparse tensor data efficiently. Therefore, leveraging [19], [29], we design an FPGA module to compute the power iteration via Kronecker products.

We consider a sparse 3-way tensor $\mathcal{X}$ as an example. We investigate the power iteration of mode 1, which is written as $\mathcal{Y} = \mathcal{X} \times_2 \mathbf{U_2}^T \times_3 \mathbf{U_3}^T$. To exploit the sparsity, we may choose to compute the Kronecker products and consider only nonzero elements $x_{ijk} \neq 0$ [19]

$$\mathbf{Y_{(1)}}(i,:) = \mathbf{Y_{(1)}}(i,:) + x_{ijk}\big[\mathbf{U_2}(j,:) \otimes \mathbf{U_3}(k,:)\big]. \quad (13)$$

The number of Kronecker products depends on the number of nonzero elements in $\mathcal{X}$, which is often very small for sparse tensors. Furthermore, a Kronecker product can be reused for all nonzero elements that share the same indices (j, k) for the second and third modes. Therefore, replacing TTM of (9) with some Kronecker products can largely reduce the computational complexity. Additionally, directly computing TTM is memory-inefficient when the size and order of $\mathcal{X}$ are large, causing a high cost of RAM and registers on FPGA.

In the Vivado HLS implementation, we utilize nested for-loops to implement the Kronecker product (Algorithm 4).

1) In order to parallelize the Kronecker product on FPGA, we pipeline the first for-loop and unroll the second for-loop. The rank of approximation, R_1, R_2, and R_3, are usually very small compared with the mode sizes. Therefore, the available memory, lookup tables (LUTs) and registers are often sufficient for parallelization.
2) To update the corresponding rows of unfolded data $\mathbf{Y}_{(n)}$ in the power iteration, we simply multiply the Kronecker product result in the LUTs with the corresponding nonzero element $y_{r_1 r_2 \ldots i_N}$.
3) In addition, different nonzero elements may share the same index of some modes. In this case, we accumulate the multiplications between these nonzero elements and their corresponding Kronecker product results.

Algorithm 4 Vivado HLS Implementation of a Kronecker Product

```
Input: a ∈ R^{1×R2}, b ∈ R^{1×R3}
for (i = 0; i < R2; i++) do
    for (j = 0; j < R3; j++) do
        c[R3 × i + j] = a[i] × b[j]
    end for
end for
Output: c ∈ R^{1×R2R3}
```

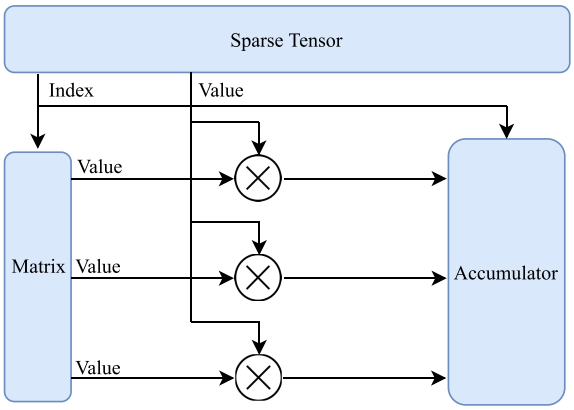


Fig. 5. Dataflow of a Kronecker product.

Fig. 5 shows the dataflow inside our Kronecker product module on FPGA. To begin with, the indices of the nonzero elements in the original tensor are extracted. Then, based on the indices of the nonzero entries, the corresponding rows of

TABLE II
ACCURACY COMPARISON OF TUCKER DECOMPOSITION WITH SVD AND WITH QRP

Tensor Size	Tucker Decomposition with SVD	Tucker Decomposition with QRP
$50 \times 50 \times 50$	1.9222×10^{-09}	1.9228×10^{-09}
$100 \times 100 \times 100$	1.3846×10^{-09}	1.3820×10^{-09}
$200 \times 200 \times 200$	1.1588×10^{-09}	1.1786×10^{-09}
$400 \times 400 \times 400$	1.2114×10^{-09}	1.2115×10^{-09}
$800 \times 800 \times 800$	3.8450×10^{-10}	3.8531×10^{-10}

the orthogonal matrix factor, $\mathbf{U}_t(i_t, :)$, are selected. Assuming there are two row vectors, every entry in one row vector multiply with every entry in the other row vector to generate the Kronecker product. Since we only compute the Kronecker product between two row vectors (not two matrices), the module only requires multiplication units (no addition units).

D. QR Decomposition With Column Pivoting

In the existing dense and sparse Tucker factorization [4], [19], the orthogonal matrix $\mathbf{U}_n$ is obtained with an SVD [31] of $\mathbf{Y}_{(n)}$. The SVD is accurate but extremely slow at computing the orthogonal matrices. In order to speedup the computation and minimize the memory usage, we propose to use QRP [30] to obtain $\mathbf{U}_n$. The QRP implementation does not lose any accuracy compared with the SVD implementation. This is clearly shown in Table II, which reports the errors of several low-rank Tucker decomposition with both SVD and QRP implementations, respectively.

Given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, the QRP get an orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{m\times n}$ and an upper-triangular matrix $\mathbf{R} \in \mathbb{R}^{n\times n}$

$$\mathbf{AP} = \mathbf{QR} \tag{14}$$

with $\mathbf{P}$ being a permutation matrix. The $\mathbf{P}$ is chosen so that the diagonal elements of $\mathbf{R}$ is nonincreasing

$$|r_{11}| \geq |r_{22}| \geq \cdots \geq |r_{nn}|. \tag{15}$$

A QRP costs about $2mn^2 - 2n^3/3$ flops, and an SVD costs about $2mn^2 + 11n^3$ flops, where $m \geq n$. In the sparse Tucker factorization of a tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$, $\mathbf{A}$ is $\mathbf{Y}_{(n)}$, the mode-n unfolding of the tensor $\mathcal{Y}$ obtained in (9). Consequently, $m = I_n$, $n = \prod_{k\neq n} R_n$, and the computational saving is huge when the tensor order N or multilinear rank parameters $(R_1, R_2, \ldots, R_N)$ are large. In some particular cases, we may end up with a fat rectangular matrix, $\mathbf{Y}_{(n)}$ $(n > m)$. In this case, we can perform QRP on a square matrix, $\mathbf{Y}_{(n)}\mathbf{Y}_{(n)}^T$.

QRP Implementation: The QRP in our implementation is based on the Householder reflection. This method computes the orthogonal matrix $\mathbf{Q}$ as the product of multiple Householder reflection matrices

$$\mathbf{Q} = \mathbf{H_1 H_2} \ldots \mathbf{H_{m-2} H_{m-1}}. \tag{16}$$

The jth reflection matrix, $\mathbf{H}_j$, is defined as

$$\mathbf{H}_j = \mathbf{I} - 2\mathbf{v}_j\mathbf{v}_j^T = \mathbf{I} - 2\frac{\mathbf{u}_j\mathbf{u}_j^T}{\mathbf{u}_j^T\mathbf{u}_j} \tag{17}$$

where $\mathbf{u}_j$ is an unit vector and $\mathbf{u}_j = [\mathbf{v}_j/(\|\mathbf{v}_j\|)]$. Vector $\mathbf{v}_j$ can be chosen based on the jth column of $\mathbf{A}$, $\mathbf{a}_j$:

$$\mathbf{v}_j = \mathbf{a}_j + \text{sign}(a_{jj})\|\mathbf{a}_n\|\mathbf{e}_1. \tag{18}$$

During every iteration of QRP, we need to update $\mathbf{A}$ by multiplying it with the Householder matrix $\mathbf{H}$. In order to generate the permutation matrix, $\mathbf{P}$, we need to compare the norms of the columns of the updated matrix $\mathbf{A}$ at every iteration, arranging the columns so that the norms of the columns are in descending order. In this way, we can place the most weighted entries in the upper left corner of $\mathbf{Q}$, achieving the similar accuracy to SVD. Since we need to compare the norms of the columns at each iteration, the QRP operation is sequential. In other words, the comparison of the column norms made it very difficult to parallelize the algorithm on FPGA. Thus, we implement the Householder QR decomposition [30] with column pivoting on CPU.

IV. RESULTS

This section shows the performance of our hybrid FPGA–CPU accelerator on both synthetic and real-world datasets. We first verify the performance of individual FPGA modules for the TTM and Kronecker product. Afterward, we verify the performance of the whole FPGA–CPU sparse Tucker accelerator and compare it with the CPU. We use the FPGA model XCVU9P-FLGA2577-3-e in our experiment. The maximum frequency of the FPGA implementation is 890 MHz. The CPU model used is Intel Core i7-6820HK CPU@2.70 GHz. The size of the RAM is 16 GB. The CPU has a maximum memory bandwidth of 34.1 GB/s and a thermal design power (TDP) of 45 W. In the experiments, we prioritize the computations on CPU to achieve the maximum performance, therefore, the energy consumption on CPU can be estimated as the product of runtime and TDP. We estimate the energy cost of sparse Tucker decomposition on FPGA on Xilinx Vivado via AWS. The communication protocol between FPGA and CPU is PCI express, which has a maximum bandwidth of 10 GB/s. Our design can also be implemented on a low-end FPGA such as Zynq-7100 as well. On a low-end FPGA, we may decrease the LUT utilization by adjusting the unroll factor in our TTM module implementation.

A. Performance of Individual FPGA Modules

First, we verify the performance of the TTM and Kronecker product modules on some synthetic tensor data and summarize their performance below.

1) *TTM Module:* We verify the performance by considering a set of 3-way tensors $\mathcal{Y} \in \mathbb{R}^{R_1\times R_2\times I_3}$ and factor matrices $\mathbf{U} \in \mathbb{R}^{R_3\times I_3}$. The rank of approximate, R_1, R_2, and R_3, are always very small compared with the original tensor size for data compression. Thus, we set $R_1 = R_2 = R_3 = 32$. The original tensor size, I_3, is set to increase from 32 to 256 as shown in Table III. In the real-life examples, the original tensor size I_3 can definitely be larger than 256. And the performance of the TTM module will not perform significantly worse when the original tensor size becomes extremely large. Here,

TABLE III
PERFORMANCE COMPARISON OF FPGA AND CPU ON THE TTM TASK

Tensor Size	Matrix Size	CPU		FPGA	
		Run-Time	Energy	Run-Time	Energy
$32 \times 32 \times 32$	32×32	0.493 ms	22.19 mJ	0.148 ms	0.4212 mJ
$32 \times 32 \times 64$	32×64	0.596 ms	26.82 mJ	0.281 ms	0.8000 mJ
$32 \times 32 \times 128$	32×128	1.165 ms	52.43 mJ	0.546 ms	1.556 mJ
$32 \times 32 \times 256$	32×256	2.021 ms	90.95 mJ	1.077 ms	3.067 mJ

TABLE IV
PERFORMANCE COMPARISON OF FPGA AND CPU ON THE KRONECKER PRODUCT TASK

Size of $\mathbf{x}_j$	Size of $\mathbf{x}_k$	CPU		FPGA	
		Run-Time	Energy	Run-Time	Energy
1×32	1×32	9.655 μs	0.4345 mJ	0.578 μs	2.111 μJ
1×64	1×64	14.72 μs	0.6624 mJ	2.301 μs	8.403 μJ
1×128	1×128	24.87 μs	1.119 mJ	9.195 μs	33.58 μJ
1×256	1×256	48.24 μs	2.171 mJ	38.55 μs	140.7 μJ

we set the maximum of our tensor size to be 256 for experimental purpose only. The FPGA achieves 1.560× to 3.331× speedup than CPU on these tensor-matrix products. We also compare the energy consumption between FPGA and CPU on the TTM task. As shown in Table III, the FPGA saves 95.6%–98.1% of energy compared with CPU.

2) *Kronecker Product Module:* As shown in Section IV-C, the Kronecker product used in the sparse Tucker decomposition deals with two row vectors, $\mathbf{x}_j \in \mathbb{R}^{1\times R_j}$ and $\mathbf{x}_k \in \mathbb{R}^{1\times R_k}$. Therefore, we compare the performance of Kronecker products on FPGA and CPU by changing the rank parameters R_1 and R_2 from 32 to 256. The rank of approximation R_1 and R_2 does not necessarily need to be equal to each other. We set R_1 and R_2 to be equal for experimental purpose only. In addition, the rank of approximation R_1, R_2, and R_3 are usually very small compared with the original tensor size for data compression. We increase the rank from 32 to 256 to demonstrate the performance of the Kronecker product module. We estimated the power of the CPU to be 45 W. The energy consumption of CPU is estimated by multiplying the power with the CPU time. The results are shown in Table IV. The speedup of FPGA over CPU ranges from 1.251× to 16.704×. As shown in Table IV, FPGA consumes 93.519% to 99.514% less energy than CPU on the Kronecker-product tasks.

B. Accelerator's Performance: Synthetic Datasets

Now we evaluate the whole hybrid FPGA–CPU accelerator on some randomly generated synthetic sparse tensor datasets. Specifically, we consider a set of 3-way tensors $\mathcal{X} \in \mathbb{R}^{200\times 200\times 200}$ with different sparsity. We fix the rank parameters $R_1 = R_2 = R_3 = 16$.

Fig. 6 compares the runtime of our hybrid FPGA–CPU platform with CPU and densor FPGA accelerator [25]. The speedup of the hybrid FPGA–CPU accelerator is $27\times \sim 853\times$ compared with CPU. The speedup of our sparse Tucker accelerator is $1.167\times \sim 126\times$ faster than the FPGA accelerator designed for dense Tucker decomposition [25]. In the whole sparse Tucker decomposition algorithm, the Kronecker product module takes the most amount of time. However, this module is parallelized in our design, and it is significantly speedup on FPGA as shown in Section IV-A. When the tensor has more nonzero elements, more Kronecker-product operations are required, leading to a more significant speedup on FPGA.

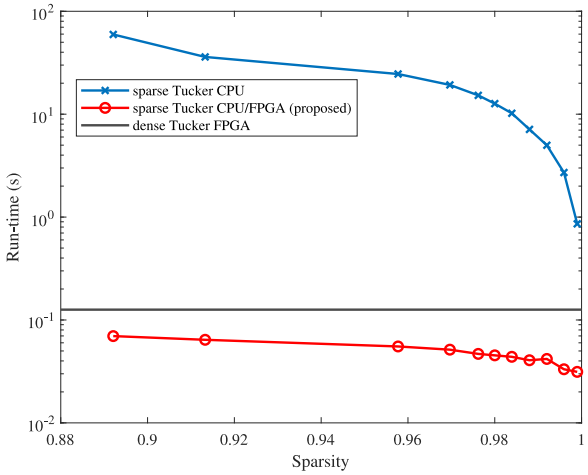


Fig. 6. Runtime comparison between the proposed hybrid platform, dense FPGA accelerator, and CPU on a set of $200 \times 200 \times 200$ synthetic random tensors with different sparsity.

C. Real-World Datasets

Finally, we verify our accelerator on four real-world sparse tensor datasets [34]–[37]. In addition, we compare the performance of our accelerator with sparse Tucker decomposition on CPU and with the dense FPGA accelerator in [25]. Table V shows the detailed runtime and energy consumption of different methods on these datasets. Table VI further shows the overall hardware resource utilization of our method on FPGA. The FPGA design is compiled for each dataset in order to achieve the maximum efficiency. We use BRAM_18K, BDSP48E, FF, and LUT to denote block random access memory, digital signal PEs, flip-flops, and LUTs, respectively.

The detailed experiments and results are summarized below.

1) *Amazon Reviews Datasets [34]:* The modes of this three-way tensor represent users, products, and words, respectively. Each nonzero element in this tensor is the number of times a word appears in a given review.

TABLE V
PERFORMANCE OF SPARSE TUCKER DECOMPOSITION ON REAL-WORLD BENCHMARKS

Benchmarks		Amazon	Nell-2	Parallel Matrix Multiplication	Retinal Angiogram
Tensor Size		$20K \times 20K \times 20K$	$1K \times 1K \times 1K$	$25 \times 25 \times 25$	130×150
Sparsity		1.128×10^{-10}	2.40×10^{-5}	8×10^{-3}	0.18
CPU	Run-Time	100.045 s	7.355 s	8.175×10^{-2} s	0.1838 s
	Energy	4502.03 J	330.98 J	3.68 J	8.27 J
Hybrid FPGA/CPU (proposed)	Run-Time	86.785 s	0.403 s	2.179×10^{-3} s	9.898×10^{-3} s
	Energy	3896.08 J	17.10 J	0.1057 J	0.4667 J
Dense FPGA Tucker [25]	Run-Time	9.47×10^4 s	9.5 s	9.9×10^{-3} s	1.18×10^{-2} s

TABLE VI
UTILIZATION OF FPGA ON REAL-WORLD BENCHMARKS. IN THE COLUMN OF "MEMORY," WE LIST THE NUMBER OF BRAM, WHERE EACH BRAM HAS 18×10^3 BITS

Name		Expression	Instance	Memory	Multiplexer	Register	Total	Available	Utilization (%)
Amazon	BRAM_18K	-	-	542	-	-	542	4320	13
	DSP48E	-	282	-	-	-	282	6840	4
	FF	0	17257	-	-	107670	124927	2364480	5
	LUT	406251	17649	-	20587	-	443268	1182240	37
Nell-2	BRAM_18K	-	-	63	-	-	63	4320	1
	DSP48E	-	470	-	-	-	470	6840	7
	FF	0	29495	-	-	54691	84186	2364480	4
	LUT	405656	30863	-	13972	-	450491	1182240	38
Parallel Matrix Multiplication	BRAM_18K	-	-	2	-	-	2	4320	~ 0
	DSP48E	-	16	-	-	-	16	6840	~ 0
	FF	0	759	-	-	107	866	2364480	~ 0
	LUT	49799	778	-	707	-	51284	1182240	4
Retinal Angiogram	BRAM_18K	-	-	5	-	-	5	4320	~ 0
	DSP48E	-	21	-	-	-	21	6840	~ 0
	FF	0	1171	-	-	9438	10609	2364480	~ 0
	LUT	121303	1089	-	2256	-	124648	1182240	11

Additionally, we extract one portion of the Amazon reviews tensor of size $20000 \times 20000 \times 20000$ and choose the rank of approximation as $R_1 = R_2 = R_3 = 32$. We perform two power iterations on all modes. The sizes of the tensors and matrices in TTM (12) are $32 \times 32 \times 20000$ and 32×20000, respectively. This sparse Tucker factorization involves nine calls of QR decomposition on a set of 20000×32 matrices in total to compute the orthogonal factor matrices. Finally, there are totally 8820 calls of Kronecker products, which depends on the number of nonzero tensor entries. On this dataset, our hybrid FPGA/CPU platform achieves 1.15× speedup than CPU with only 13.5% energy consumption. Our method also achieves 1091× speedup than the dense Tucker FPGA accelerator [25].

2) *NELL-2 Datasets [37]:* This dataset is extracted from the never ending language learner knowledge base. The nonzero entries represent some entity-relation-entity tuples. We extract one portion of the NELL-2 dataset and obtain a sparse tensor of size $1000 \times 1000 \times 1000$. In addition, we choose our rank of approximation as $R_1 = R_2 = R_3 = 16$. We perform five power iterations on all modes. The sizes of the tensors and matrices in TTM (12) are $16 \times 16 \times 1000$ and 16×1000, respectively. This sparse Tucker factorization involves 15 calls of QR decomposition on a set of 1000×256 matrices in total to compute the orthogonal factor matrices. Finally, there are totally 432 555 calls of Kronecker products, which depends on the number of nonzero tensor entries. Our hybrid FPGA/CPU platform achieves 18× speedup and 94.8% energy saving compared with CPU. Our method is also 23.6× faster than the dense FPGA accelerator [25].

3) *Binary 3-Way Tensor for Parallel Matrix Multiplication [35], [36]:* This binary tensor describes the parallel computation process of matrix multiplications. Given two matrices $\mathbf{A} \in \mathbb{R}^{M \times K}$ and $\mathbf{B} \in \mathbb{R}^{K \times N}$, their product results in a matrix $\mathbf{C} \in \mathbb{R}^{M \times N}$. Let $I_1 = MK$, $I_2 = KN$ and $I_3 = MN$, then a binary 3-way tensor $\mathcal{X}$ can represent the parallel matrix multiplication. The first mode corresponds to the first input matrix $\mathbf{A}$ with entries in row-major order; the second mode corresponds to the input matrix $\mathbf{B}$ with entries in row-major order; and the third mode corresponds to the output matrix $\mathbf{C}$ with entries in column-major order. A nonzero entry $x_{i_1 i_2 i_3} = 1$ corresponds to a scalar multiplication within the classical matrix multiplication algorithm: the i_1th entry of $\mathbf{A}$ is multiplied with the i_2th entry of $\mathbf{B}$, and the result is accumulated into the i_3th entry of $\mathbf{C}$. The number of nonzero elements in $\mathcal{X}$ is $nnz = MKN$. We consider the case $M = N = K = 5$, which results in a binary tensor $\mathcal{X}$ with size $25 \times 25 \times 25$ and a sparsity of 8×10^{-3}. To perform sparse Tucker decomposition on this 3-way binary tensor, we choose an approximation rank of $R_1 = R_2 = R_3 = 5$. We perform three steps of high-order power iterations on all modes, leading to 3 TTM in (12) and totally 6 calls for QRP. Finally, the number of Kronecker products used in this dataset is 1125. Our method achieves 37× and 1.52× speedup than CPU and than the dense FPGA accelerator [25], respectively. Compared with the sparse Tucker decomposition on CPU, our accelerator saves 97.1% energy.

4) *Retinal Angiogram:* Angiogramy is a medical diagnostic test that uses X-ray to take picture of the blood vessels. The images, angiogram, are always very sparse. Fig. 6

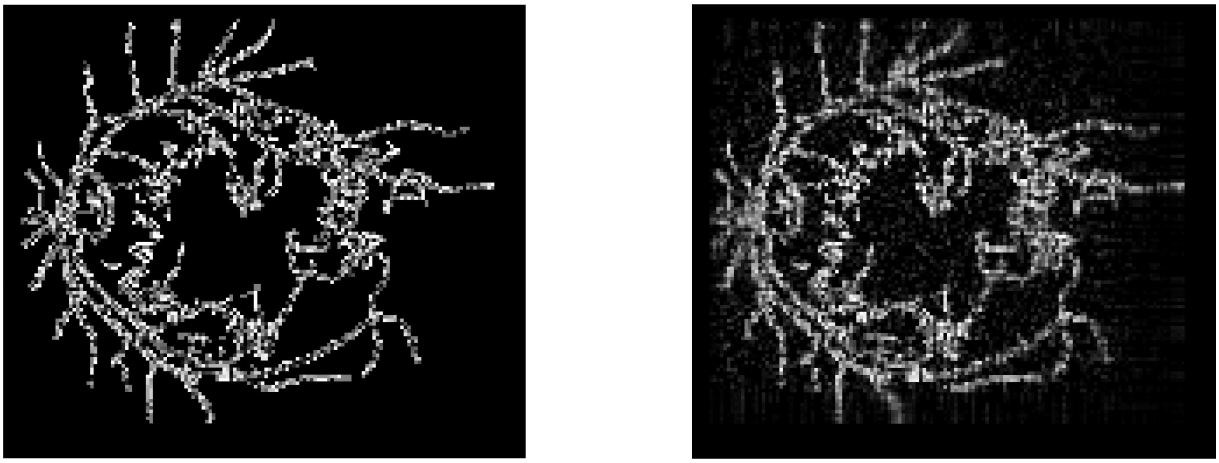


Fig. 7. Left: original retinal angiogram. Right: approximated image by our sparse Tucker decomposition on the FPGA/CPU hybrid platform.

shows the retinal angiogram of a patient on the left. The size of the original retinal angiogram is 130×150 [38]. Tucker factorization can also be employed to compress 2-D data, because a matrix is the special case of a tensor. Different from SVD compression of a matrix where the rank is a scalar, a Tucker decomposition allows one to set two rank parameters. We perform a sparse Tucker decomposition with rank $R = [30, 35]$ on this image. We performed 12 steps of high-order power iterations on all modes, leading to 12 TTM in (12) and totally 24 calls for QRP. We do not need any Kronecker products since the order of the tensor is 2. Our proposed method achieves 19× speedup than CPU and 1.91× speedup than dense FPGA accelerator [25], and it saves 94.4% energy compared with the sparse Tucker factorization on CPU. Fig. 7 compares the original retinal angiogram and the resulting compressed image from our FPGA/CPU hybrid accelerator. The compression ratio is 18.57×. While the image is highly compressed, the essential features, such as blood vessels, are still clearly preserved.

V. Conclusion

This article has proposed a hybrid FPGA–CPU accelerator for sparse Tucker decomposition. On the algorithm level, the Kronecker products have exploited the data sparsity and has significantly reduced the computational complexity. The QR with pivoting method have dramatically reduced the complexity of obtaining the orthogonal mode-n matrix factors. The FPGA modules for the TTM and for the Kronecker products have achieved 93.519%–99.514% energy saving compared with CPU on synthetic benchmarks. The proposed hybrid FPGA–CPU accelerator has been evaluated with both synthetic and realistic sparse tensor datasets. It has achieved 27× ∼853× speedup over CPU and 1.167×∼ 126× speedup over the recently developed dense Tucker FPGA accelerator [25] on the synthetic datasets. Our proposed methods have also achieved 1.15× ∼1091× speedup and over 95% energy savings on the tested real-world tensor datasets. Our proposed accelerator have significantly outperformed CPU and dense Tucker FPGA accelerator [25] when the tensor is very large and sparse.

References

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

[2] I. Oseledets and E. Tyrtyshnikov, "TT-cross approximation for multidimensional arrays," *Linear Algebra Appl.*, vol. 432, no. 1, pp. 70–88, 2010.

[3] L. De Lathauwer, B. De Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 4, pp. 1253–1278, 2000.

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

[5] J. D. Carroll and J. J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of 'Eckart–Young' decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[6] R. A. Harshman, "Foundations of the PARAFAC procedure: Models and conditions for an 'explanatory' multi-modal factor analysis," UCLA, Los Angeles, CA, USA, Working Papers in Phonetics, 1970.

[7] M. A. O. Vasilescu and D. Terzopoulos, "Multilinear analysis of image ensembles: TensorFaces," in *Proc. Eur. Conf. Comput. Vis.*, 2002, pp. 447–460.

[8] M. A. O. Vasilescu and D. Terzopoulos, "Multilinear subspace analysis of image ensembles," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2. Madison, WI, USA, 2003, pp. 93–99.

[9] M. A. O. Vasilescu and D. Terzopoulos, "Multilinear independent components analysis," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, vol. 1. San Diego, CA, USA, 2005, pp. 547–553.

[10] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *Proc. 8th IEEE Int. Conf. Data Min.*, Pisa, Italy, 2008, pp. 363–372.

[11] N. Batmanghelich, A. Dong, B. Taskar, and C. Davatzikos, "Regularized tensor factorization for multi-modality medical image classification," in *Proc. Int. Conf. Med. Image Comput. Comput. Assist. Intervention*, 2011, pp. 17–24.

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

[13] 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. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 1, pp. 63–76, Jan. 2015.

[14] Z. Zhang, K. Batselier, H. Liu, L. Daniel, and N. Wong, "Tensor computation: A new framework for high-dimensional problems in EDA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 4, pp. 521–536, Apr. 2017.

[15] J. Luan and Z. Zhang, "Prediction of multidimensional spatial variation data via bayesian tensor completion," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 2, pp. 547–551, Feb. 2020.
[16] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Proc. Adv. Neural Inf. Process. Syst. Conf.*, Montreal, QC, Canada, 2015, pp. 442–450.
[17] Y. Yang, D. Krompass, and V. Tresp, "Tensor-train recurrent neural networks for video classification," in *Proc. Int. Conf. Mach. Learn.*, vol. 70, Aug. 2017, pp. 3891–3900.
[18] C. Hawkins and Z. Zhang, "Bayesian tensorized neural networks with automatic rank selection," 2019. [Online]. Available: arXiv:1905.10478.
[19] O. Kaya and B. Uçar, "High performance parallel algorithms for the tucker decomposition of sparse tensors," in *Proc. IEEE 45th Int. Conf. Parallel Process. (ICPP)*, Philadelphia, PA, USA, 2016, pp. 103–112.
[20] S. Smith, J. Park, and G. Karypis, "Sparse tensor factorization on many-core processors with high-bandwidth memory," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Orlando, FL, USA, 2017, pp. 1058–1067.
[21] 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 Perform. Comput. Netw. Storage Anal.*, Austin, TX, USA, 2015, pp. 1–12.
[22] 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," 2015. [Online]. Available: arXiv:1511.06530.
[23] N. Srivastava *et al.*, "T2S-tensor: Productively generating high-performance spatial hardware for dense tensor computations," in *Proc. 27th Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, San Diego, CA, USA, 2019, pp. 181–189.
[24] W.-P. Huang *et al.*, "High performance hardware architecture for singular spectrum analysis of hankel tensors," *Microprocess. Microsyst.*, vol. 64, pp. 120–127, Feb. 2019.
[25] K. Zhang, X. Zhang, and Z. Zhang, "Tucker tensor decomposition on FPGA," in *Proc. Int. Conf. Comput.-Aided Design*, 2019, pp. 1–8.
[26] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, San Diego, CA, USA, 2020, pp. 689–702.
[27] S. F. Roohi, D. Zonoobi, A. A. Kassim, and J. L. Jaremko, "Dynamic MRI reconstruction using low rank plus sparse tensor decomposition," in *Proc. IEEE Int. Conf. Image Process.*, Phoenix, AZ, USA, 2016, pp. 1769–1773.
[28] P. Fillard, V. Arsigny, X. Pennec, P. M. Thompson, and N. Ayache, "Extrapolation of sparse tensor fields: Application to the modeling of brain variability," in *Proc. Biennial Int. Conf. Inf. Process. Med. Imag.*, 2005, pp. 27–38.
[29] C. F. Van Loan, "The ubiquitous Kronecker product," *J. Comput. Appl. Math.*, vol. 123, nos. 1–2, pp. 85–100, 2000.
[30] G. H. Golub and C. F. V. V. Loan, *Matrix Computations*, 3rd ed. Baltimore, MD, USA: Johns Hopkins Univ. Press, 1996.
[31] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," in *Linear Algebra*. Berlin, Germany: Springer, 1971, pp. 134–151.
[32] F. G. Gustavson, "Some basic techniques for solving sparse systems of linear equations," in *Sparse Matrices and Their Applications*. Boston, MA, USA: Springer, 1972, pp. 41–52.
[33] P. A. Tew, "An investigation of sparse tensor formats for tensor libraries," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2016.
[34] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: Understanding rating dimensions with review text," in *Proc. 7th ACM Conf. Recommender Syst.*, 2013, pp. 165–172.
[35] A. R. Benson and G. Ballard, "A framework for practical parallel fast matrix multiplication," in *Proc. ACM SIGPLAN Symp. Principles Pract. Parallel Program.*, 2015, pp. 42–53.
[36] R. P. Brent, "Algorithms for matrix multiplication," Dept. Comput. Sci., Stanford University, Stanford, CA, USA, Rep. TR-CS-70-157, DCS, 1970.
[37] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka, Jr., and T. M. Mitchell, "Toward an architecture for never-ending language learning," in *Proc. 24th AAAI Conf. Artif. Intell.*, vol. 5, 2010, p. 3.
[38] A. Hoover, V. Kouznetsova, and M. Goldbaum, "Locating blood vessels in retinal images by piecewise threshold probing of a matched filter response," *IEEE Trans. Med. Imag.*, vol. 19, no. 3, pp. 203–210, Mar. 2000.

Weiyun Jiang received the B.Sc. degree in electrical engineering from the University of California at Santa Barbara, Santa Barbara, CA, USA, in 2020. He is currently pursuing the graduation degree in electrical engineering with Stanford University, Stanford, CA, USA.

His research interests include algorithm/hardware co-design for tensor data analysis and machine learning.

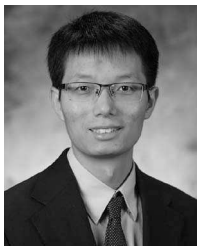
Kaiqi Zhang received the B.Sc. degree in electronic engineering from Tsinghua University, Beijing, China, in 2016, and the M.S. degree in electrical and computer engineering from the University of California at Davis, Davis, CA, USA, in 2018. He is currently pursuing the Ph.D. degree in electrical and computer engineering with the University of California at Santa Barbara, Santa Barbara, CA, USA.

Colin Yu Lin received the B.Sc. degree in electronic engineering from Sun Yat-sen University, Guangzhou, China, in 2005, the M.E. degree in computer engineering from the University of Chinese Academy of Sciences, Beijing, China, in 2008, and the Ph.D. degree in electrical and electronic engineering from the University of Hong Kong, Hong Kong, in 2012.

From 2011 to 2012, he was a Visiting Student Researcher with the Department of Electrical Engineering and Computer Sciences and the Berkeley Wireless Research Center, University of California at Berkeley, Berkeley, CA, USA. He was an Assistant Professor with the System on Programmable Chip Research Department, Institute of Electronics, Chinese Academy of Sciences, Beijing, from 2012 to 2016. He is currently a Software Development Senior Manager with Data Center Group, Xilinx, Inc., Beijing. His current research interests include field programmable gate array (FPGA) architecture, CAD for FPGAs, high-level synthesis, and FPGA for high performance computing.

Feng Xing received the B.Sc. degree from Wuhan University, Wuhan, China, the M.Sc. degree in pure mathematics and applied mathematics from the University of Lille, Lille, France, and the Ph.D. degree in high performance computing from "Maison de la Simulation," CEA Saclay, Paris, France, in 2014.

He worked as a Post-Doctoral Researcher with INRIA French, Rocquencourt, France, and BRGM, Orléans, France, for two years on high performance geothermal simulation. He is currently a Software Development Manager with Xilinx, Inc., Beijing, China.



Zheng Zhang (Member, IEEE) received the Ph.D. degree in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 2015.

He has been an Assistant Professor of Electrical and Computer Engineering with the University of California at Santa Barbara, Santa Barbara, CA, USA, since July 2017. His research interests include uncertainty quantification and tensor computation. The applications of his research include design automation of nanoscale electronics and photonics, algorithm/hardware co-design of high-dimensional, robust, and safe machine learning systems.

Dr. Zhang received three best paper awards from IEEE TRANSACTIONS: the Best Paper Award of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS in 2014, two best paper awards of the IEEE TRANSACTIONS ON COMPONENTS, PACKAGING AND MANUFACTURING TECHNOLOGY in 2018 and 2020, respectively, and three best paper awards at international conferences. His Ph.D. dissertation won the ACM SIGDA Outstanding Ph.D. Dissertation Award in Electronic Design Automation in 2016, and the Best Thesis Award from the Microsystems Technology Laboratory of MIT in 2015. He received the NSF CAREER Award in 2019 and a Facebook Research Award in 2020.