Microprocessors and Microsystems 64 (2019) 120–127

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

High performance hardware architecture for singular spectrum analysis of Hankel tensors

Wei-pei Huang [a], Bowen P.Y. Kwan [a], Weiyang Ding [b], Biao Min [a], Ray C.C. Cheung [a,*], Liqun Qi [b], Hong Yan [a]

[a] Department of Electronic Engineering, City University of Hong Kong, Hong Kong
[b] Department of Applied Mathematics, Hong Kong Polytechnic University, Hong Kong

A R T I C L E I N F O

Article history:
Received 21 March 2018
Revised 6 September 2018
Accepted 9 October 2018
Available online 10 October 2018

Keywords:
Hardware architecture
Hankel tensor
Tucker decomposition(TKD)
Higher-order singular value decomposition (HOSVD)

A B S T R A C T

This paper presents a hardware architecture for singular spectrum analysis of Hankel tensors, including computation of tucker decomposition, tensor reconstruction and final Hankelization. In the proposed design, we explore two level of optimization. First, in algorithm level, we optimize the calculation process by exploiting the Hankel property to reduce the computation complexity and on-chip BRAM resource usage. Secondly, in hardware level, parallelism is explored for acceleration. Resource sharing is applied to reduce look-up tables (LUTs) usage. To enable flexibility, the number of processing elements (PEs) can be changed through parameter setting. Our proposed design is implemented on Field-Programmable Gate Arrays (FPGAs) to process third order tensors. Experiment results show that our design achieve a speedup from 172 to 1004 compared with CPU implementation via Intel MKL and 5 to 40 compared with GPU implementation.

© 2018 Published by Elsevier B.V.

1. Background

In mathematics, a tensor is often represented as a multidimensional (or multi-way) data array. In the past, to analyze tensor data, the most common approach is to first flatten or vectorize the data, and then use the well-developed matrix analysis tools, such as principal component analysis (PCA), nonnegative matrix factorization(NMF), and sparse component analysis (SCA) [1,2]. However, the flattening or unfolding operation breaks the tensor structure which can be important as a reflection of information in a multidimensional space, and such method is unable to capture multiple interactions and couplings. Moreover, in matrix analysis, like independent component analysis (ICA) [3,4], strict constraints should be imposed to ensure uniqueness. Such constraints sometimes will affect the accuracy of the model. To overcome such drawbacks, tensor decomposition method became popular in different fields, such as signal processing [5], computer vision [6,7] and data mining. It is proved to be successful in the applications of social network analysis, brain data analysis, web mining, information retrieval and healthcare analytics [8].

Basically, there are two fundamental types of tensor decomposition, i.e. CANDECOMP/ PARAFAC decomposition (CPD) and Tucker decomposition (TKD) [9]. TKD is a form of higher-order principal component analysis, which is an extension of matrix PCA. Such decomposition can be done through higher order singular value decomposition (HOSVD) [9,10], which provides a powerful tool for spectrum analysis. On the other hand, in signal processing, Hankel matrices play an important role [11]. As an extension, Hankel tensor was introduced by Luque and Thibon [12]. It has been successfully applied to exponential data fitting [13–15], and signal separation modeling [16].

However, by now, many researchers only focus on how to apply tensor decomposition method or how to optimize the decomposition algorithm. There is little hardware architecture development that aims at accelerating analysis of Hankel tensors. Tensor decomposition is known to be computationally intensive. For the tensor reconstruction process and Hankelization process, their computational complexity are rather high, which is $(N_1r^3 + N_1N_2r^2 + N_1N_2N_3r)$ and $(N_1N_2N_3)$ respectively, where N_1, N_2 and N_3 denotes the size of each dimension and r is the rank of the tensor, and it increases significantly with the dimension rising. To our knowledge, this is the first attempt to build a hardware accelerator for singular spectrum analysis of Hankel tensors. Our work has three main contributions as follows:

– We propose a hardware architecture to perform Tucker decomposition of a Hankel tensor and its reconstruction. In addition, by simply disabling the Hankelization process module,

* Corresponding author.
E-mail address: r.cheung@cityu.edu.hk (R.C.C. Cheung).

https://doi.org/10.1016/j.micpro.2018.10.004
0141-9331/© 2018 Published by Elsevier B.V.

our technique can be applied to the analysis of arbitrary (non-Hankel) tensors.
- By re-organizing the calculation process and employing the full pipeline technology, we have obtained a speed-up from 286 to 1004 times for different workloads compared with CPU implementation and 16 to 40 compared with GPU implementation.
- We present a BRAM resources friendly architecture to compute HOSVD of Hankel tensors. The on-chip BRAM usage in our method decrease from $O(n^2)$ to $O(n)$.

This paper is organized as follows. In Section 2, a quick introduction of singular spectrum analysis of Hankel tensors is provided. In Section 3, we re-organize the computation process and propose a new architecture for acceleration. In Section 4, we evaluate the performance of the proposed design and compare it with software implementation on CPU. Finally, in Section 5, we conclude our work.

2. Singular spectrum analysis of hankel tensor

For a given input time series $\mathbf{x} \in \mathbb{C}^N$, to derive a Hankel tensor, also known as embedding, first we use an overlapping window to map $\mathbf{x}$ to $\tilde{\mathbf{X}}$ shown in Eq. (1),

$$\tilde{\mathbf{X}} = \begin{pmatrix} x_1 & x_2 & \cdots & x_{l_3} \\ x_{m+1} & x_{m+2} & \cdots & x_{m+l_3} \\ \vdots & \vdots & \ddots & \vdots \\ x_{(N_3-1)m+1} & x_{(N_3-1)m+2} & \cdots & x_{(N_3-1)m+l_3} \end{pmatrix} \tag{1}$$

where l_3 is the window size, m is the interval length between consecutive window positions, and N_3 denotes the number of parts segmented by the window. Then each row of $\tilde{\mathbf{X}}$ is segmented using an overlapping window with interval of one. In other words, the ith row of $\tilde{\mathbf{X}}$ is mapped to a matrix in Eq. (2),

$$\mathcal{X}_{::i} = \begin{pmatrix} \tilde{\mathbf{X}}(i,1) & \tilde{\mathbf{X}}(i,2) & \cdots & \tilde{\mathbf{X}}(i,N_2) \\ \tilde{\mathbf{X}}(i,2) & \tilde{\mathbf{X}}(i,3) & \cdots & \tilde{\mathbf{X}}(i,N_2+1) \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{\mathbf{X}}(i,N_1) & \tilde{\mathbf{X}}(i,N_1+1) & \cdots & \tilde{\mathbf{X}}(i,N_2+N_1) \end{pmatrix} \tag{2}$$

which is the ith frontal slice of constructed Hankel tensor $\mathcal{X}$, denoted as $\mathcal{X}_{::i}$, where N_2 is the size of overlapping window, N_1 is the number of segments obtained by the moving forward window and $N_1 + N_2 = l_3$. More details on how to form the tensor from a time series can be found in [5]. In our method, we only require the time series input, and parameters N_1, N_2, N_3 and m rather than the explicit Hankel tensor. After the embedding process, we can get a third order Hankel tensor $\mathcal{H}(\mathbf{x}) \in \mathbb{C}^{N_1 \times N_2 \times N_3}$, where "third order" refers to the number of dimension of $\mathcal{H}(\mathbf{x})$.

2.1. HOSVD of Hankel tensors

Suppose $\mathcal{H}(\mathbf{x}) \in \mathbb{C}^{N_1 \times N_2 \times N_3}$ is a third order Hankel tensor, which is generated by vector $\mathbf{x}$ with length of $N = N_1 + N_2 + N_3 - 2$. According to the Hankel property, we have $\mathcal{H}(x)_{ijk} = \mathbf{x}(i + j + mk)$. Following the definition of Tucker decomposition in [9], Hankel tensor $\mathcal{H}(\mathbf{x})$ can be decomposed as

$$\hat{\mathcal{H}}(\mathbf{x}) = \mathcal{S} \times_1 \mathbf{U}_1^\top \times_2 \mathbf{U}_2^\top \times_3 \mathbf{U}_3^\top, \tag{3}$$

where $\times_x$ represents tensor-matrix product, $\mathcal{S}$ is called core tensor [9], $\mathbf{U}_1 \in \mathbb{C}^{N_1 \times r_1}$, $\mathbf{U}_2 \in \mathbb{C}^{N_2 \times r_2}$ and $\mathbf{U}_3 \in \mathbb{C}^{N_3 \times r_3}$ are obtained by computing HOSVD of the tensor $\mathcal{H}(\mathbf{x})$, i.e. performing SVD of $\mathcal{H}(\mathbf{x})_{(1)}\mathcal{H}(\mathbf{x})^*_{(1)}$ to get $\mathbf{U}_1$, SVD of $\mathcal{H}(\mathbf{x})_{(2)}\mathcal{H}(\mathbf{x})^*_{(2)}$ to get $\mathbf{U}_2$, and SVD of $\mathcal{H}(\mathbf{x})_{(3)}\mathcal{H}(\mathbf{x})^*_{(3)}$ to get $\mathbf{U}_3$, respectively, where $\mathcal{H}(\mathbf{x})_{(n)}$ denotes the mode-n unfolding of tensor $\mathcal{H}(\mathbf{x})$. For the core tensor $\mathcal{S}$, it can be obtained as follows. In Eq. (4), it includes a series of tensor-matrix product. In fact, to multiply a tensor with a matrix, we just need to reorganize (or unfold) the tensor as a matrix, and then perform matrix multiplication. Afterwards, reorganize the result as a tensor back. The detail definition for how to reorganize the tensor to matrix or matrix to tensor can be found in [9].

$$\tilde{\mathcal{S}} = \mathcal{H}(\tilde{\mathbf{x}}) \times_1 \bar{\mathbf{U}}_1 \times_2 \bar{\mathbf{U}}_2 \times_3 \bar{\mathbf{U}}_3, \tag{4}$$

In order to get $\mathbf{U}_1$, $\mathbf{U}_2$ and $\mathbf{U}_3$, we choose the subspace iteration method for Hermitian Eigenvalue Problems (HEP) [17], which is shown in Algorithm 1 in detail, for computing the dominant eigenvalues and eigenvector of $\mathcal{H}(\mathbf{x})_{(x)}\mathcal{H}(\mathbf{x})^*_{(x)}$, where $x = 1, 2, 3$. By exploiting Hankel property, matrix-matrix multiplication of $\mathcal{H}(\mathbf{x})_{(x)}\mathcal{H}(\mathbf{x})^*_{(x)}$ can be represented as

Algorithm 1 Subspace iteration algorithm for Hermitian eigenvalue problem (HEP).

Input: Random initial guess matrix $\mathbf{Z}$, Hermitian matrix $\mathbf{A} = \mathcal{H}(\mathbf{x})_{(x)}\mathcal{H}(\mathbf{x})^*_{(x)}$;
Output:Orthogonal matrix $\mathbf{V}$;
1: Perform QR factorization of $\mathbf{Z}$: $\mathbf{Z} = \mathbf{VR}$
2: **for** $k = 1, 2, \ldots$ **do**
3: $\mathbf{Y} = \mathbf{AV}$
4: $\mathbf{H} = \mathbf{V}^*\mathbf{Y}$
5: **if** $||\mathbf{Y} - \mathbf{VH}||_2 > \epsilon_M$ **then**
6: Perform QR factorization of $\mathbf{Y}$: $\mathbf{Y} = \mathbf{VR}$
7: **else**
8: Break the for loop and output $\mathbf{V}$
9: **end if**
10: **end for**

$$\begin{aligned}
&\mathbf{A} = \mathcal{H}(\mathbf{x})_{(x)}\mathcal{H}(\mathbf{x})^*_{(x)} = \mathbf{\Lambda_x H_x D_x H_x^* \Lambda_x} \\
&\mathrm{diag}(\mathbf{\Lambda}_x) = \mathrm{ind}\{x\},\ x = 1, 2, 3 \\
&\mathrm{diag}(\mathbf{D}_1) = \mathrm{ind}\{2\} * \mathrm{ind}\{3\},\ \mathrm{diag}(\mathbf{D}_2) = \mathrm{ind}\{1\} * \mathrm{ind}\{3\}, \\
&\mathrm{diag}(\mathbf{D}_3) = \mathrm{ind}\{1\} * \mathrm{ind}\{2\}, \\
&\mathrm{ind}\{1\} \in \{0,1\}^{N_1},\ \mathrm{ind}\{2\} \in \{0,1\}^{N_2}, \\
&\mathrm{ind}\{3\} \in \{0,1\}^{N+2-N_1-N_2}, \\
&\mathrm{ind}\{1\} = \{1,1,\ldots,1\},\ \mathrm{ind}\{2\} = \{1,1,\ldots,1\}, \\
&\mathrm{ind}\{3\} = \{\underbrace{1,0,\ldots,0}_{m}, \underbrace{1,0,\ldots,0}_{m}, \ldots, \underbrace{1,0,\ldots,0}_{m}, 1\},
\end{aligned} \tag{5}$$

where $\mathbf{H}_x$ is a $N_x \times (N + 1 - N_x)$ Hankel matrix generated from input $\mathbf{x}$, and $\mathbf{\Lambda}_x$ and $\mathbf{D}_x$ are diagonal matrices, $x = 1, 2, 3$, ind{1}, ind{2} and ind{3} are indices that denote the generation scheme of the Hankel tensor.

2.2. Core tensor computation and Hankel tensor reconstruction

In signal processing applications, one needs to filter the input signal to obtain a new core tensor [5] and then reconstruct the signal with the original eigen-matrices derived above and new core tensor. Such core tensor $\tilde{\mathcal{S}}$ is obtained from Eq. (4), where $\tilde{\mathbf{x}}$ represents the processed input signal. Our previous developed fast Hankel tensor-vector multiplication algorithm [18] can be applied to accelerate this process. Therefore, we can transform the computation to the element-wise calculation

$$\begin{aligned}\tilde{\mathcal{S}}_{ijk} = ifft\{\tilde{\mathbf{x}}\}^\top \cdot [fft\{\bar{\mathbf{U}}_1(:,i)\} .* fft\{\bar{\mathbf{U}}_2(:,j)\} \\ .* fft\{\bar{\mathbf{U}}_3(:,k)\}],\end{aligned} \tag{6}$$

where .* represent matrix Hadamard product and fft{ · } denotes the fast Fourier transform. We can see that, in the above equation, for calculation of 1 element of core tensor $\tilde{\mathcal{S}}_{ijk}$, we only need to compute 3 FFT, 1 IFFT, 3 point-wise vector multiplication and 1

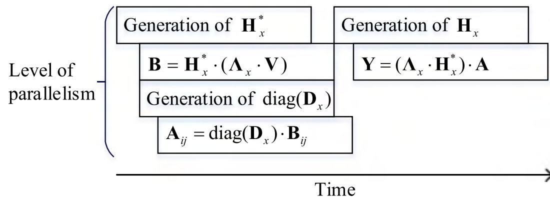

Fig. 1. Modified procedure for computing $\mathbf{Y} = \mathbf{\Lambda}_x\mathbf{H}_x\mathbf{D}_x\mathbf{H}_x^*\mathbf{\Lambda}_x\mathbf{V}$.

vector inner product. Since FFT computation is efficient in hardware, the computation complexity is therefore reduce compared to the original calculation process as Eq. (4). By using the new core tensor, we can reconstruct the tensor $\mathcal{H}$ and Hankelize it as a time series h_l,

$$\tilde{\mathcal{H}} = \tilde{\mathcal{S}} \times_1 \mathbf{U}_1^\top \times_2 \mathbf{U}_2^\top \times_3 \mathbf{U}_3^\top \tag{7}$$

$$\tilde{h}_{l,temp} = \sum_{i+j+mk=l} \tilde{\mathcal{H}}_{i,j,k} \tag{8}$$

$$\tilde{h}_l = \frac{\tilde{h}_{l,temp}}{d_l}, \tag{9}$$

where d_l is determined by $d_l = \mathrm{ind}\{1\} * \mathrm{ind}\{2\} * \mathrm{ind}\{3\}$. As illustrated at the beginning of Section 2.1, for a Hankel tensor, we should ensure $\mathcal{H}(x)_{ijk} = \mathbf{x}(i + j + mk)$. As a result, to make a non-Hankel tensor as a Hankel tensor, i.e. the Hankelization process, we can average entries $\mathcal{H}(x)_{ijk}$ to $\mathbf{h}_l$, where $i + j + mk = l$, which is shown in Eqs. (8) and (9).

3. High performance hardware architecture design

To implement the computation module discussed in the preceding section, we explore parallelism of the process and proposed a high performance and resource friendly hardware architecture.

3.1. HOSVD Module

Based on the subspace iteration Algorithm 1, the main process is to iteratively perform $\mathbf{Y} = \mathbf{\Lambda}_x\mathbf{H}_x\mathbf{D}_x\mathbf{H}_x^*\mathbf{\Lambda}_x\mathbf{V}$, which is a Hankel matrix multiplication procedure, and then update $\mathbf{V}$ by QR factorization $\mathbf{Y} = \mathbf{VR}$ until the process converges. QR factorization is implemented following the Householder algorithm.

For the process of computing $\mathbf{Y}$, the extracted parallelism is shown in Fig. 1. Though some researchers raise a scheme to perform matrix multiplication by Intel AVX [19], it still has some performance lost compared with the FPGA implementation. In the proposed design, we modify the matrix multiplication module proposed in [20]. The differences between our matrix multiplication module, shown as Fig. 2 and [20] is that our module adds an extra multiplier, a multiplexer and a BRAM to cache diag($\mathbf{D}_x$)$_n$. The reason why we add these will be explained later in this section. For $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, $\mathrm{FIFO_1}$ to $\mathrm{FIFO_n}$ are used to store partial column of a same column of matrix $\mathbf{A}$, with a size of 2*x bytes for each. Likely, $\mathrm{BRAM\text{-}B_1}$ to $\mathrm{BRAM\text{-}B_m}$ are used to store partial row of a same row of matrix $\mathbf{B}$, with a size of 2*y bytes for each. $\mathrm{BRAM\text{-}C_{ij}}$, where $i = 1, 2, \ldots n$, $j = 1, 2, \ldots, m$, is used to cache the outer product of partial column in $\mathrm{FIFO_i}$ and partial row in $\mathrm{BRAM_j}$, with a size of 2*xy bytes. The reason for the size of BRAM or FIFO above include a factor of "2" is that we use double buffering strategy for such data storage. $\mathrm{BRAM\text{-}diag(D_x)_1}$ to $\mathrm{BRAM\text{-}diag(D_x)_n}$ are used to cache partial value of *diag*($\mathbf{D}_x$), with a size of x bytes for each.

To start with, as shown in Fig. 1, we first generate the first column of $\mathbf{H}_x^*$ according to Eqs. (1) to (2) and feed it to the FIFOs in Fig. 2. Then the first row of $\mathbf{\Lambda}_x\mathbf{V}$ is also fed to the local $\mathrm{BRAM\text{-}B_m}$ in Fig. 2. According to Eq. (5), entries of diag($\mathbf{\Lambda}_x$) is either 0 or 1. Thus, diag($\mathbf{\Lambda}_x$)$\mathbf{V}$ is equivalent to set specific rows of $\mathbf{V}$ to be zero depending on the value of corresponding entry of diag($\mathbf{\Lambda}_x$). For the process of fetching a row of $\mathbf{\Lambda}_x\mathbf{V}$, we only need to decide whether we should fetch the row of $\mathbf{V}$ or only feed zero row. Then, we can perform matrix multiplication of $\mathbf{H}_x^*(\mathbf{\Lambda}_x\mathbf{V})$. Such procedure is performed in pipeline as Fig. 1. On the other hand, diagonal entries of $\mathbf{D}_x$ is computed while the multiplication takes place following Eq. (5), which is a predefined sequence only related to the dimension of Hankel tensor. Then it will be fed to $\mathrm{BRAM\text{-}diag(D_x)_1}$. Once the result of multiplication between $\mathrm{FIFO_n}$ and $\mathrm{BRAM\text{-}B_m}$ available, control logic will fetch the corresponding entry of diag($\mathbf{D}_x$) to the next multiplier and the multiplication result will be fed to the adder, as shown in Fig. 2.

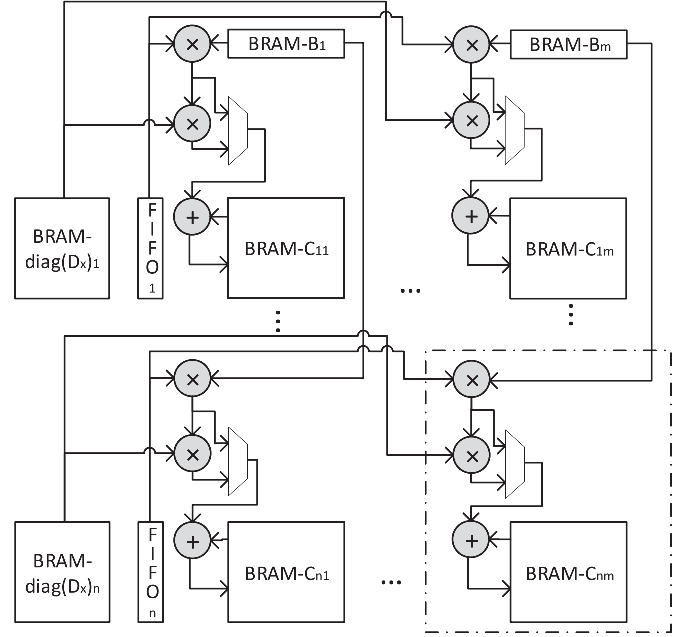

Fig. 2. Matrix multiplication module.

After we get the result of $\mathbf{A} = \mathbf{D}_x\mathbf{H}_x^*\mathbf{\Lambda}_x\mathbf{V}$, which is stored in the group of $\mathrm{BRAM\text{-}C_{nm}}$ separately, we will multiply the remaining matrix to this result to get $\mathbf{Y}$ by reusing the matrix multiplication module. Firstly, one row of $\mathbf{A}$ will be fetch to $\mathrm{BRAM\text{-}B_m}$. At the same time, we need to feed the first column of $\mathbf{\Lambda}_x\mathbf{H}_x$. To do this, we need to generate a column of $\mathbf{H}_x$ by the method we generate $\mathbf{H}_x^*$ before. As discuss before, $\mathbf{\Lambda}_x\mathbf{H}_x$ is equivalent to set the corresponding row of $\mathbf{H}_x$ to be zero. As a result,fetch a column of $\mathbf{\Lambda}_x\mathbf{H}_x$ is equivalent to set corresponding entry of the column of $\mathbf{H}_x$ to be zero. Then, we can perform matrix multiplication of $(\mathbf{\Lambda}_x\mathbf{H}_x)\mathbf{A}$ following the scheme in [20] in pipeline. At this time, since we do not need to multiply diagonal matrix diag($\mathbf{D_x}$), a multiplexer is added to bypass the second multiplier as shown in Fig. 2.

With this implementation, Eq. (5) can be computed with 2 matrix multiplications instead of 5 as suggested by the algorithm. As we can see that, throughout the procedure, we compute the required elements of Hermitian matrix $\mathcal{H}(\mathbf{x})_{(x)}\mathcal{H}(\mathbf{x})_{(x)}^*$ on the fly. Or in other words, Hermitian matrix calculation is merged with the calculation of $\mathbf{Y}$. Thus, we do not need to store $\mathcal{H}(\mathbf{x})_{(x)}\mathcal{H}(\mathbf{x})_{(x)}^*$ explicitly on-chip, which is of size $(N_1^2 + N_2^2 + N_3^2)$, where N_1, N_2, N_3 is each dimension of the Hankel tensor. We only store the vector $\mathbf{x}$ in on-chip BRAM, which is used to form $\mathcal{H}(\mathbf{x})_{(x)}\mathcal{H}(\mathbf{x})_{(x)}^*$. As a result, on-chip BRAM requirement dropped from $(N_1^2 + N_2^2 + N_3^2)$ to $(N_1 + N_2 + N_3 - 2)$.

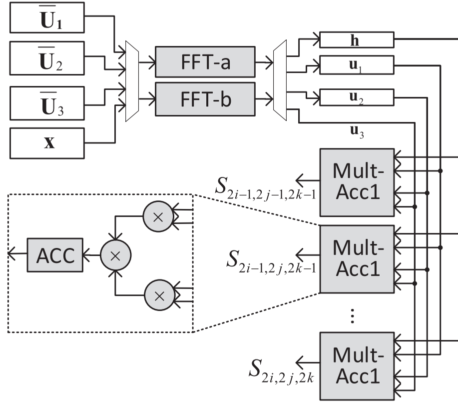

Fig. 3. Architecture of core tensor computation.

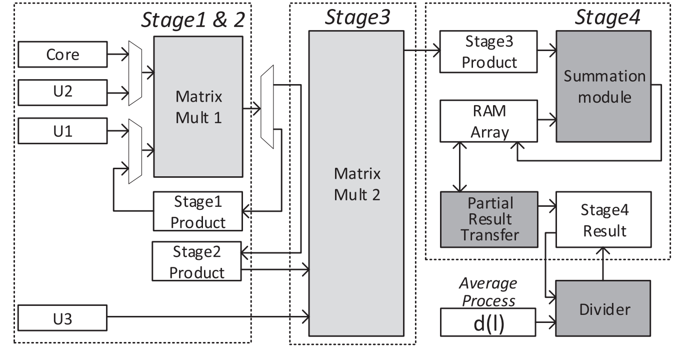

Fig. 4. Pipeline structure of signal reconstruction process.

3.2. Core tensor computation for the Hankel tensor module

First we denote $\mathbf{h} = ifft\{\tilde{\mathbf{x}}\}^T, \mathbf{u}_1^i = fft\{\bar{\mathbf{U}}_1(:,i)\}, \mathbf{u}_2^j = fft\{\bar{\mathbf{U}}_2(:,j)\}$ and $\mathbf{u}_3^k = fft\{\bar{\mathbf{U}}_3(:,k)\}$, where $1 \le i \le R_1$, $1 \le j \le R_2$ and $1 \le k \le R_3$. Then we can rewrite Eq. (6) as

$$\mathcal{S}_{ijk} = \sum_{l=1}^{N} \mathbf{h}(l) \cdot \mathbf{u}_1^i(l) \cdot \mathbf{u}_2^j(l) \cdot \mathbf{u}_3^k(l). \tag{10}$$

Based on this equation, we perform the calculation based on two FFTs, FFT-a and FFT-b and 8 multiplier-accumulator trees (Mult-Acc), as shown in Fig. 3. Basically, we implement a triple nested for-loop in the hardware. The most inner loop sequentially computes a pair of $\mathbf{u}_2$. The second loop calculates a pair of $\mathbf{u}_1$ and then stores the result back to the RAM. The outermost loop calculates a pair of $\mathbf{u}_3$ sequentially. As a trade-off between performance and resources, only 5 vectors, $\mathbf{h}$, $\mathbf{u}_1^{2i-1}$, $\mathbf{u}_1^{2i}$, $\mathbf{u}_2^{2j-1}$ and $\mathbf{u}_2^{2j}$, are cached for reuse and changed iteratively. Once execution flow enters the most inner loop and the FFT pairs of $\mathbf{u}_3$ become available, the previous result in RAM and the $\mathbf{u}_3$ pair are fed to 8 multiplier-accumulator trees. In the proposed design, FFT module is fully pipelined and the Fourier transform is overlapped with the multiply-accumulate process. Assume $R_1 = R_2 = R_3 = r$, the time complexity is reduced from $(2r^3 \cdot N)$ to $(r^3 \cdot N/8)$ after acceleration.

To save resources, the complex number multipliers used in the multiplier-accumulator trees are time-shared with matrix multiplication module which is activated in the signal reconstruction part in our design. The LUT usage is thus reduced by 10% this way.

The core tensor computation is the same as tensor reconstruction process, as shown in Eqs. (4) and (7). Thus, for a tensor without the Hankel property, core tensor can be obtained through the tensor reconstruction module by disabling Hankelization process module which is introduced in the following part.

3.3. Tensor reconstruction and Hankelization module

As illustrated in Section 1, these two processes are computationally intensive, which have complexity $(N_1r^3 + N_1N_2r^2 + N_1N_2N_3r)$ and $(N_1N_2N_3)$ respectively, where N_1, N_2 and N_3 denotes the size of each dimension and r is the rank of the tensor. In this section, we explain how to accelerate the tensor reconstruction and Hankelization processes. By utilizing the property of tensor multiplication, Eq. (7) can be re-written as

$$\mathcal{H} = \{\mathbf{U}_2 \cdot \{\mathbf{U}_1 \cdot \{\mathbf{U}_3 \cdot \mathcal{S}_{(3)}\}_{(1)}\}_{(2)}, \tag{11}$$

where $\{\cdot\}_{(x)}$ refers to mode-x tensor unfolding operation [9]. To accelerate the process, we properly divide the whole task into small partial tensor reconstruction and design a heavy pipeline architecture as shown in Fig. 4. The architecture of each matrix-mult module is shown as Fig. 2. For a PE we mention below, it refers to a single basic processing element as circled in Fig. 2.

For the first 3 stages, which correspond to three tensor-matrix multiplications in Eq. (7). Firstly, the three eigen-matrices, $\mathbf{U}_x$, are divided into k_x row partitions, where $x = 1, 2, 3$, and let $\mathbf{U}_x^{(m)}$ denote the mth row block. We use R_i^j to denote the ith partial result as output of Stage j. For the first stage, we flatten the core tensor $\mathcal{S}$ in mode 3, and multiply it with $\mathbf{U}_3^{(1)}$. The result, R_1^1, will be cached in RAM and flattened into mode 1 and feed to the same matrix-mult module to perform $\mathbf{U}_1^{(1)} \cdot \{R_1^1\}_{(1)}$. Similarly, the result, R_1^2 will be cached and continue to feed to Stage 3 to perform $\mathbf{U}_2^{(1)} \cdot \{R_1^2\}_{(2)}$. At the same time, the matrix-mult1 module will perform $\mathbf{U}_1^{(2)} \cdot \{R_1^1\}_{(1)}$. We continue to multiply R_1^2 with each block of $\mathbf{U}_2$, until the last block have been reached. Then we update Stage 2 result, i.e. computing $\mathbf{U}_1^{(3)} \cdot \{R_1^1\}_{(1)}$. After we get R_2^2, Stage 3 process will execute as before. Until R_1^1 have been multiplied with all the block of $\mathbf{U}_1$, we start to update the Stage 1 result with $\mathbf{U}_3^{(2)} \cdot \{\mathcal{S}_3\}_{(1)}$, i.e. R_2^1, and update Stage 2 result with $\mathbf{U}_1^{(1)} \cdot \{R_2^1\}_{(1)}$. Then the Stage 3 process will continue to be executed. This execution flow will be shown as Table 1. We define one phase as the time taken in deriving a single partial product in Stage 3, i.e. $\frac{N_3N_2N_1r_2}{k_1k_2k_3PE_2}$. To save BRAM resources, the latest partial products of Stage 1 and Stage 2 will be cached, and the older will discard. Calculating partial products in Stage 1 and Stage 2 will consume m and n phases respectively. With proper number of PEs, we make $m + n < k_3$ but not in great difference, which ensure Stage 1 and Stage 2 are overlapped with Stage 3 and use PEs as little as possible. The whole process needs a total number of $((k_3 + k_1k_2k_3 + 1) \cdot \frac{N_3N_2N_1r_2}{k_1k_2k_3PE_2})$ phases.

For Stage 4, i.e. accumulation process as Eq. (8), if a set of entries $\{a_{i_m j_m k_m}\}$ of tensor $\mathcal{A}$ satisfy $i_m + j_m + k_m = C$, they will be accumulated to the Cth entry of the vector $\mathbf{h}$ and finally it is divided by d_l. Since addition in floating point logic always needs multiple pipeline stages, accumulation process using one adder should wait for the previous addition results. In the proposed design, a technique called adder time-sharing is used. Data which will be accumulated to different entries of $\mathbf{h}$ are interleaved to time-share the adders and store intermediate results not used immediately. To enable this interleaving operation, we set row number of Stage 3 partial matrix product to 20 which should be no less than the pipeline stage of a floating point adder, and feed the same adder with elements of partial products through the column major. Let $\mathbf{C}_{i,j}$ denote the partial matrix product caching in the (ij)th result in BRAM of the PE group in Stage 3. To accelerate this process, we use Stage 4 in Fig. 4 which is overlapped with Stage 3 calculation and we also increase adder unit to enhance parallelism. To be specific, we need to constrain the time consumption within one phase. We also further explore adder sharing. Every two matrix products,

Table 1
Execution flow of signal tensor reconstruction.

	Initialization Phase		Phase 1	Phase 2	Phase 3	…	Phase n	…	Phase k_3		
Stage1	$\mathbf{U}_3^1 \cdot \mathcal{S}_{(3)}$	–	–	–	–	–	–	–	–		
Stage2	–	$\mathbf{U}_1^1 \cdot \{R_1^1\}_{(1)}$	$\mathbf{U}_1^2 \cdot \{R_1^1\}_{(1)}$				–	–	–		
Stage3	–	–	$\mathbf{U}_2^1 \cdot \{R_1^2\}_{(2)}$	$\mathbf{U}_2^2 \cdot \{R_1^2\}_{(2)}$	$\mathbf{U}_2^3 \cdot \{R_1^2\}_{(2)}$	…	$\mathbf{U}_2^n \cdot \{R_1^2\}_{(2)}$	…	$\mathbf{U}_2^{k_3} \cdot \{R_1^2\}_{(2)}$		
Stage4	–	–	–	$h(R_1^3)$	$h(R_2^3)$	…	$h(R_{n-1}^3)$	…	$h(R_{k_3-1}^3)$		
	Phase k_3+1	…	…	Phase $(k_2-1)k_3+1$	…	Phase $(k_2-1)k_3+m$	…	…	Phase k_2k_3+1	…	Phase $k_1k_2k_3+1$
Stage1	–	–	–	$\mathbf{U}_3^2 \cdot \{\mathcal{S}_{(3)}\}_{(1)}$		–	–	–	–	…	–
Stage2	$\mathbf{U}_1^3 \cdot \{R_1^1\}_{(1)}$		…	–	–	$\mathbf{U}_1^1 \cdot \{R_2^1\}_{(1)}$		–	$\mathbf{U}_1^2 \cdot \{R_2^1\}_{(1)}$	…	–
Stage3	$\mathbf{U}_2^1 \cdot \{R_2^2\}_{(2)}$	…	…	$\mathbf{U}_2^1 \cdot \{R_{k_2}^2\}_{(2)}$	…	$\mathbf{U}_2^m \cdot \{R_{k_2}^2\}_{(2)}$	…	…	$\mathbf{U}_2^{k_3} \cdot \{R_{k_2+1}^2\}_{(2)}$	…	–
Stage4	$h(R_{k_3}^3)$	…	…	$h(R_{(k_2-1)k_3}^3)$	…	$h(R_{(k_2-1)k_3+m-1}^3)$	…	…	$h(R_{k_2k_3-1}^3)$	…	$h(R_{k_1k_2k_3}^3)$

i.e. $\mathbf{C}_{2i,j}$ and $\mathbf{C}_{2i+1,j}$, will share the same adder and one BRAM, denoted as $h_{mid}^{i,j}$, to store the intermediate accumulated results. Partial products in the odd rows of PE group, i.e. $\{\mathbf{C}_{2i+1,j} | i, j \in \mathbb{N}, i \leq n/2, j \leq m\}$, are first fed to the Stage 4 module, then the process is repeated for the even rows, i.e. $\{\mathbf{C}_{2i,j} | i, j \in \mathbb{N}, i \leq n/2, j \leq m\}$. To enable dynamic routing for the adder array, a RAM selector is implemented to route the data from $\mathbf{C}_{i,j}$ and $h_{mid}^{i,j}$ to the appropriate adder and feed the addition result to the appropriate $h_{mid}^{i,j}$ BRAM. After the whole tensor is reconstructed, i.e. the pipeline of Stages 1 to 4 complete all their work, $\mathbf{h}$ will be divided by a vector $\mathbf{d}$ element-wise in the "Average Process" in Fig. 4, which forms the final Hankelized result, i.e. the vector obtained by Eq. (9).

4. Performance and resource usage analysis

To our knowledge, there are some works that tried to build a distributed system to accelerate Tucker decomposition. In detail, these works are try to optimize the tensor times matrix chain (TTMc) process for sparse tensor [21,22]. However, they just minimize storage usage for sparse tensor and distribute the element-wise computation to different machines. They do not fully explore the fine grain parallelism and none of them target at Hankel tensor nor exploiting the Hankel property. Since there is no existing architecture for Hankel tensor computation acceleration, we implement all the part by using the optimized method provided in Intel MKL library, which is a best choice to exploit the computation ability in CPU. On the other hand, we implement a GPU version for comparison.

In the CPU implementation, in detail, we use cgemm for tensor-matrix multiplication, cdotu for dot product calculation and dfti rountine for FFT and IFFT calculation. While in the GPU implementation, for the HOSVD part, we implement a same algorithm by using cuBLAS and cuSOLVER for a better result. In detail, we use cgemm for complex value matrix multiplication, DnCgeqrf routine for QR decomposition, Scnrm for norm calculation. For the core tensor computation, we use cuFFT to compute fft and ifft and cuBLAS for vector multiplication and dot product. For the tensor reconstruction, we utilize the method raised in [23]. In detail, as the strategy used in our hardware design, for Eq. (11), we also divide $\mathbf{U}_x$ into k_x row partitions, where $x = 1, 2, 3$. Each time we pick a row partition of $\mathbf{U}_1$, $\mathbf{U}_3$ and $\mathbf{U}_3$, following Eq. (11) ,we only reconstruct a small block tensor with a size of $k_1 \cdot k_2 \cdot k_3$. After a small block tensor reconstructed, we perform Hankelization. By this way, the whole big tensor reconstruction and hankelization is broken down to sequentially small block tensors reconstruction and hankelization.

We compare the execution time of our architecture with CPU and GPU implementation. CPU implementation is run on Intel Core i7-6500 CPU at 2.59 GHz with 8-GB memory. GPU implementation is run on NVIDIA GEFORCE GTX 1050i at 1.29 GHz with 4-GB DDR5. Our proposed architecture is implemented on the Xilinx Virtex-7 xc7vx485 FPGA chip by using Verilog HDL on Vivado 2016.2 version, which runs at 100 MHz. We utilize single precision floating point arithmetics to represent complex numbers to enable larger dynamic range. To perform a preliminary evaluation of our hardware architecture, we choose FPGA rather than ASIC to get a prompt result. It can be transfered to ASIC implementation easily.

In our hardware design, the parameter setting is as follows, $PE_1 = 1 \times 4$, $PE_2 = 20 \times 2$, where PE_1 denotes the number of PE in matrix mult1 module, PE_2 denotes that in matrix mult2 module. For both matrix mult1 and mult2 module, the architecture is shown as Fig. 2. For matrix multiplication $\mathbf{A} \cdot \mathbf{B}$ processing in matrix mult1 or matrix mult2 module, each time, we load a partial column of $\mathbf{A}$ and a partial row of $\mathbf{B}$ to each PE. We use X_n to denote the size of such partial row in Stage n ($n = 1, 2, 3$) in tensor recon-

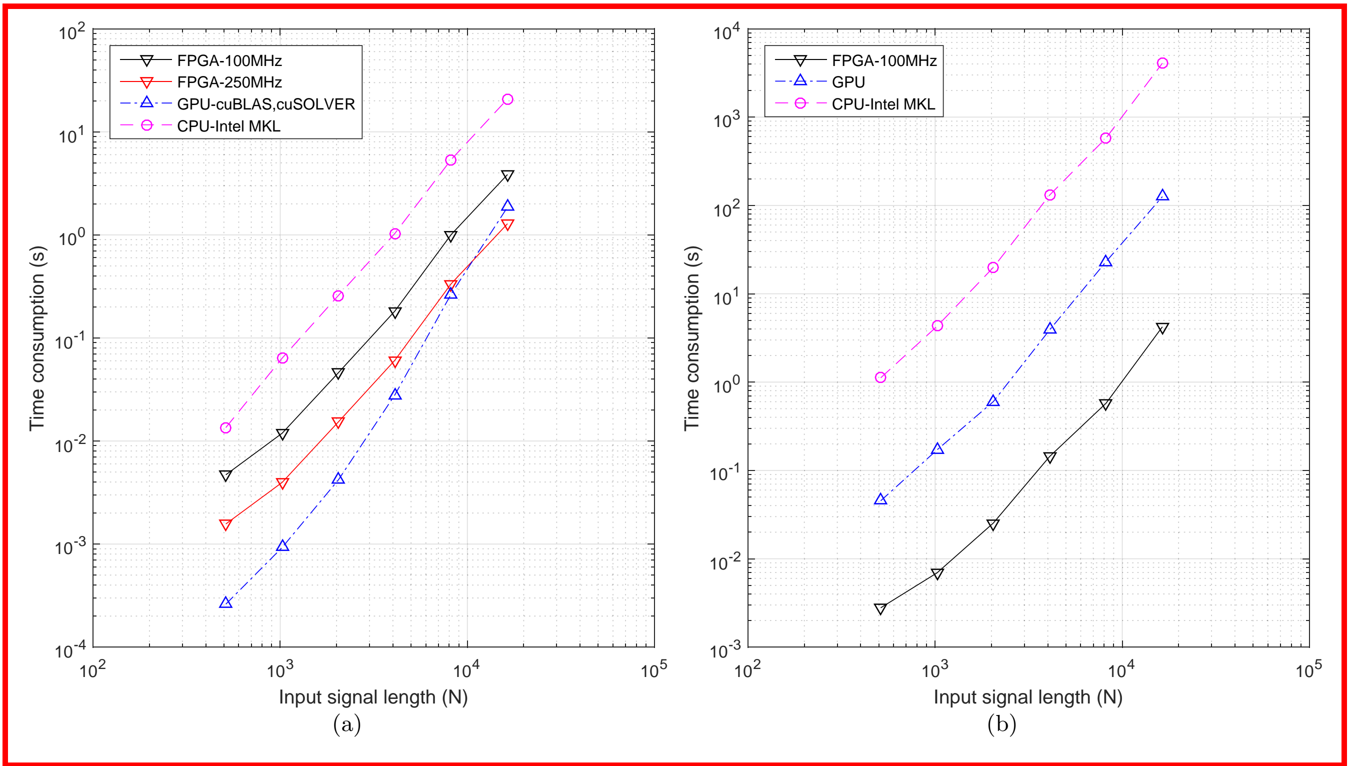

Fig. 5. Performance comparison between software and our hardware design. (a), Time consumption comparison of HOSVD module. (b), Time consumption comparison of core tensor calculation and Hankel tensor reconstruction module.

struction part. Likewise, we use Y_n to denote the size of such partial column. In our implementation, all of these are parameterized, and we set $X_1 = 9,\ X_2 = 15,\ Y_1 = Y_2 = 10$ and $Y_3 = X_3 = 20$. For the input time series $\mathbf{x}$, they are generated six times with lengths of 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13} and 2^{14} , according to the following equation

$$x[n] = s[n] + e[n]; \tag{12}$$

where $s[n]$ is a narrow-band signal, $e[n]$ is normalized non-stationary Gaussian white noise where the noise variance changes with time. When the signal length increases, the size of RAM that stores $\mathbf{U}_x$, $\mathbf{h}$, intermediate FFT result in core tensor reconstruction and the Hankelized result will increase accordingly while other hardware components all remain unchanged. For $N = 2048$, resource usage for the implemented FPGA design is listed in Table 3. For the HOSVD module, since the number of iterations highly depends on the input data and initial guess, we compare the time consumption for one iteration in Fig. 5a. In our proposed design, we use subspace iteration algorithm. As a fair comparison, it consumes less time than the software implementation using the same algorithm. For the software implementation adopting Lanczos algorithm, it is more efficient but more complex and not resource-friendly.

From Fig. 5, we can see that, for FPGA implementation running at 100 MHz, it is better than CPU one but slightly worse than GPU one. However, the performance gap between GPU and FPGA implementation is narrowing with input signal length increasing. For input signal with a length of 163,84, GPU only 2 times faster than FPGA implementation run at 100 MHz. If we barely implement the HOSVD module in another FPGA board, HOSVD module can run at 250 MHz due to more space for routing. In this case, FPGA even outperforms GPU when the input signal is longer than 163,84. Though for FPGA implementation running at low clock frequency, GPU have better performance, it is known that it is not efficient in terms of power. If performance is of most importance, we can build up a heterogeneous system where HOSVD is performed on GPU, while the remaining computation on FPGA.

For core tensor computation and tensor reconstruction module, they are executed sequentially. From Fig. 5b we can see that, our proposed design significantly outperforms the software implementation and the GPU one. In detail, for the 6 evaluation samples, the speed-up compared with CPU implementation range from 172 to 1004. While compared with GPU implementation, the speed-up range from 5 to 40. This mainly benefits from the heavy pipeline design in tensor reconstruction and Hankelization process. In FPGA or ASIC implementation, we can easily determine the execution time of each pipeline stage by alternating the number of PE in each stage. Though GPU is powerful in matrix multiplication, we cannot determine such time beforehand. As a result, we can't build up a efficient pipeline that each stage is overlapped perfectly.

In our proposed design, the block matrix multiplication scheme needs to first divide the two input matrices into sub-blocks. If the dimension is not divisible by the number of PEs, we should append zero rows or columns to make the dimension divisible, which may cause performance degeneracy slightly. In order to observe how computation order infect the performance, we change the tensor-matrix multiplication sequence and decompose different size tensor. As shown in the first two rows in Table 2, in Stage 3, we should divide 799 and 609 by 20 respectively. For the first case, we only append 1 zero row, while for second case, we should append 11 zero rows. As a result, the acceleration rate decrease accordingly. Eq. (7) contains three tensor-matrix multiplications. The computation order of these multiplications does not affect the final result but can affect the computing time when they are implemented in parallel. Since the first two tensor-matrix multiplications will be fed to the smaller matrix multiplication module, then the last one fed to the larger one in terms of the number of PEs needed, the arrangement of the computing order in Eq. (7) will influence the processor speed, as shown in Table 2. In this table, we only include the execution time of Stage 4 in software and hard-

Table 2
Time consumption comparison of Stage 3 among different Hankel tensor construction schemes and computation orders for same input time series with 2048 points, following the signal construction method described at the beginning of Section 2. S1, S2 and S3 represent the row numbers of $\mathbf{U}_x$ allocated in Stages 1, 2 and 3 respectively. Speed-up rate1 refers to the speedup over CPU implementation, while Speed-up rate2 refers to that over GPU implementation. .

S1	S2	S3	CPU Time Consumption(s)	GPU Time Consumption(s)	Our Design Time Consumption(s)	Speed-up Rate1	Speed-up Rate2
800	16	799	19.376	0.47552	0.024418	794	19
800	33	609	30.515	1.0804	0.047737	639	22
800	41	449	28.816	0.81773	0.059396	485	13
1000	65	409	50.452	1.5005	0.103118	489	14
409	65	1000	50.834	1.4482	0.084755	600	17
409	1000	65	51.857	1.5495	0.299867	172	5

Table 3
Proposed design resource usage .

	Used	Available	Utilization
LUT	189,009	303,600	62.26%
Register	245,021	607,200	40.35%
RAMB36E1	61	2060	2.96%
RAMB18E1	381	1030	36.99%
DSP48E	705	2800	25.18%

ware implementation. The first three rows represent different Hankel construction. It can be seen that, $S3$ should be close to multiples of X_3, i.e. 400, while the other two should be close to multiples of X_1, X_2, i.e. 10. The last three rows in Table 2 represent the same tensor construction but with different arrangements of the computing order of Eq. (7). We observe that a large row number of $\mathbf{U}_x^\top$ should be placed to the last in the tensor-matrix multiplication sequence to achieve a fast computation. However, for the CPU and GPU implementation, the execution time is only related to the size of the Hankel tensor, i.e. the value of $N_1 \cdot N_2 \cdot N_3$. The FPGA implementation processing a batch of data in a PE group with fixed number. It is very unlikely that, in the GPU and CPU implementation, the computation is broken down into element-wise computations. Then for the CPU, these element-wise computations are executed in parallel, while for the GPU, such are then assigned to multiple cores. As a result, there is no mismatch between the size of tensor and the size of PE. The execution time of GPU implementation also only relate to the size of the Hankel tensor.

5. Conclusions

In this paper, we have proposed a high performance hardware architecture targeting on singular spectrum analysis of Hankel tensors. Firstly, by exploiting the Hankel property and re-order the computing process, time consumption of Tucker decomposition of Hankel tensors is reduced significantly. Then, through full pipeline design, the computation is further accelerated. Computing resources are shared within different functional blocks, where computation is not executed at the same time. To the best of our knowledge, this is the first FPGA architecture on the speed-up of Tucker decomposition and its reconstruction. The hardware can also process tensors without the Hankel property by using the tensor reconstruction module to obtain the core tensor with the Hankelization process disabled. Our experiment results shows that our design outperforms software implementation significantly.

Acknowledgement

This work is supported by the Hong Kong Research Grants Council (Project C1007-15G).

References

[1] A. Cichocki, R. Zdunek, A. H. Phan, S.-i. Amari, Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation, John Wiley & Sons, Chichester, UK, pp. 8–20.
[2] A.M. Bruckstein, D.L. Donoho, M. Elad, From sparse solutions of systems of equations to sparse modeling of signals and images, SIAM Rev. 51 (1) (2009) 34–81.
[3] A. Hyvärinen, J. Karhunen, E. Oja, Independent component analysis, vol. 46, John Wiley & Sons, New York, USA, pp. 145–164.
[4] J.-F. Cardoso, Chapter 4 - Likelihood, in: P. Comon, C. Jutten (Eds.), Handbook of Blind Source Separation, Academic Press, Oxford, 2010, pp. 107–154.
[5] S. Kouchaki, S. Sanei, E.L. Arbon, D.-J. Dijk, Tensor based singular spectrum analysis for automatic scoring of sleep eeg, IEEE Trans. Neural Syst. Rehabil. Eng. 23 (1) (2015) 1–9.
[6] C. Bauckhage, Robust Tensor Classifiers for Color Object Recognition, in: Image Analysis and Recognition, Springer, 2007, pp. 352–363.
[7] J. Wang, A. Barreto, L. Wang, Y. Chen, N. Rishe, J. Andrian, M. Adjouadi, Multilinear principal component analysis for face recognition with fewer features, Neurocomputing 73 (10) (2010) 1550–1555.
[8] E.E. Papalexakis, C. Faloutsos, N.D. Sidiropoulos, Tensors for data mining and data fusion: models, applications, and scalable algorithms, ACM Trans. Int. Syst. Technol. (TIST) 8 (2) (2016) 16.
[9] T.G. Kolda, B.W. Bader, Tensor decompositions and applications, SIAM Rev. 51 (3) (2009) 455–500.
[10] N.D. Sidiropoulos, L.D. Lathauwer, X. Fu, K. Huang, E.E. Papalexakis, C. Faloutsos, Tensor decomposition for signal processing and machine learning, IEEE Trans. Signal Process. 65 (13) (2017) 3551–3582.
[11] L. Qi, Hankel tensors: associated hankel matrices and vandermonde decomposition, Commun. Math. Sci. 13 (1) (2015) 113–125.
[12] J.-G. Luque, J.-Y. Thibon, Hankel hyperdeterminants and selberg integrals, J. Phys. A Math. Gen. 36 (19) (2003) 5267.
[13] R. Boyer, L. De Lathauwer, K. Abed-Meraim, Higher order tensor-based method for delayed exponential fitting, IEEE Trans. Signal Process. 55 (6) (2007) 2795–2809.
[14] J.-M. Papy, L. De Lathauwer, S. Van Huffel, Exponential data fitting using multilinear algebra: the single-channel and multi-channel case, Numer. Linear Algebra Appl. 12 (8) (2005) 809–826.
[15] J.-M. Papy, L. De Lathauwer, S. Van Huffel, Exponential data fitting using multilinear algebra: the decimative case, J. Chemom. 23 (7–8) (2009) 341–351.
[16] L. De Lathauwer, Blind separation of exponential polynomials and the decomposition of a tensor in rank-(l_r,l_r,1) terms, SIAM J. Matrix Anal. Appl. 32 (4) (2011) 1451–1474.
[17] M. Gu, A. Ruhe, R. Lehoucq, D. Sorensen, R. Freund, G. Sleijpen, H. van der Vorst, Z. Bai, R. Li, Hermitian Eigenvalue Problems, in: Templates for the Solution of Algebraic Eigenvalue Problems, SIAM, 2000, pp. 45–107.
[18] W. Ding, L. Qi, Y. Wei, Fast hankel tensor–vector product and its application to exponential data fitting, Numer. Linear Algebra Appl. 22 (5) (2015) 814–832.
[19] S.A. Hassan, A. Hemeida, M.M. Mahmoud, Performance evaluation of matrix–matrix multiplications using intel's advanced vector extensions (avx), Microprocess. Microsyst. 47 (2016) 369–374.
[20] S.J. Campbell, S.P. Khatri, Resource and delay efficient matrix multiplication using newer fpga devices, in: Proceedings of the 16th ACM Great Lakes Symposium on VLSI, in: GLSVLSI '06, ACM, New York, NY, USA, 2006, pp. 308–311.
[21] O. Kaya, B. Uçar, High performance parallel algorithms for the tucker decomposition of sparse tensors, in: Parallel Processing (ICPP), 2016 45th International Conference on, IEEE, 2016, pp. 103–112.
[22] S. Smith, N. Ravindran, N.D. Sidiropoulos, G. Karypis, Splatt: Efficient and parallel sparse tensor-matrix multiplication, in: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, IEEE, 2015, pp. 61–70.
[23] S.K. Suter, J.A.I. Guitian, F. Marton, M. Agus, A. Elsener, C.P. Zollikofer, M. Gopi, E. Gobbetti, R. Pajarola, Interactive multiscale tensor reconstruction for multiresolution volume visualization, IEEE Trans. Vis. Comput. Graph. 17 (12) (2011) 2135–2143.

Wei-pei HUANG received his B.Eng. and M.Eng. degree from South China University of Technology, Guangzhou, China, in 2013 and 2016, respectively. He is now working towards his Ph.D. degree at the Department of Electronic Engineering, City University of Hong Kong. His research interests include high performance computation hardware architecture design, acceleration of tensor decomposition.

Bowen P.Y. Kwan received the M.Eng degree in Electrical and Electronic Engineering from Imperial College London in 2016. In 2017, he joined the Department of Electronic Engineering, City University of Hong Kong, as a research associate. He is currently a research assistant in the Department of Computing, Imperial College London. His research interests include architectures and application acceleration of reconfigurable hardware (FPGAs), especially on machine-learning algorithms.

Weiyang Ding received the Ph.D. degree from Fudan University, Shanghai, China, in 2016. From 2016 to 2017, he was a Postdoctoral Fellow with Department of Applied Mathematics, the Hong Kong Polytechnic University, Hong Kong. He is currently a Research Assistant Professor with Department of Mathematics, Hong Kong Baptist University, Hong Kong. His research interests include numerical multilinear algebra, tensor analysis, computation, and applications.

Biao Min received the B.E. and M.E. degrees from Xidian University, Xi'an, China, in 2008 and 2011, respectively. In 2018, He received his Ph.D. degree from the Department of Electronic Engineering, City University of Hong Kong, Hong Kong. His current research interests include video coding and reconfigurable computing.

Ray C.C. Cheung received the B.Eng and M.Phil degrees in computer engineering and computer science and engineering at the Chinese University of Hong Kong (CUHK), Hong Kong, in 1999 and 2001, respectively, and the Ph.D. degree and DIC in computing at Imperial College London, London, United Kingdom, in 2007. After completing the Ph.D. work, he received the Hong Kong Croucher Foundation Fellowship for his postdoctoral study in the Electrical Engineering Department, University of California, Los Angeles (UCLA). In 2009, he worked as a visiting research fellow in the Department of Electrical Engineering, Princeton University, Princeton, NJ. Currently, he is an associate professor in the Department of Electronic Engineering, City University of Hong Kong (CityU). He is the author of more than 100 journal and conference papers. His research team, CityU Architecture Lab for Arithmetic and Security (CALAS), focuses on the following research topics: reconfigurable trusted computing, applied cryptography, and high-performance biomedical VLSI designs. He is a member of the IEEE.

Liqun Qi received the B.S. degree at Tsinghua University, Beijing, China, in 1968 and the M.S. and Ph.D. degrees from the University of Wisconsin-Madison in 1981 and 1984, respectively. He has taught at Tsinghua University, the University of Wisconsin-Madison, the University of New South Wales, and the City University of Hong Kong. He is now the Chair Professor of applied mathematics, The Hong Kong Polytechnic University. He has published more than 150 research papers in international journals. His research interests include multilinear algebra, tensor computation, nonlinear numerical optimization and applications.

Hong Yan received his Ph.D. degree from Yale University. He was professor of imaging science at the University of Sydney and currently is professor of computer engineering at City University of Hong Kong. His research interests include image processing, pattern recognition and bioinformatics. He has authored or co-authored over 300 journal papers in these areas. He was elected an IAPR fellow for contributions to document image analysis and an IEEE fellow for contributions to image recognition techniques and applications. He received the 2016 Norbert Wiener Award from IEEE SMC Society for contributions to image and biomolecular pattern recognition techniques.