Microelectronics Journal 85 (2019) 25–33

Contents lists available at ScienceDirect

Microelectronics Journal

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

A high performance hardware architecture for non-negative tensor factorization

Biao Min, Wei-pei Huang, Ray C.C. Cheung *, Hong Yan

Department of Electronic Engineering, City University of Hong Kong, Hong Kong

ARTICLE INFO

Keywords:
Non-negative tensor factorization
High-dimensional data
Hardware architecture
Field-programmable gate array (FPGA)
Parallel architecture

ABSTRACT

Non-negative tensor factorization (NTF) algorithm is an emerging method for high-dimensional data analysis, which is applied in many fields such as computer vision, and bioinformatics. This paper presents an effective method to accelerate NTF computations and proposes a corresponding hardware architecture, which consists of multiple processing units. The decomposed factors are calculated by using shared intermediate results. By using the proposed method, NTF can be implemented in parallel and hardware resources can be saved by sharing. In this paper, we evaluate the proposed architecture on Xilinx Virtex-6 FPGA XC6VLX760T and apply it into 2 applications, i.e. video background estimation and facial images processing. The experimental results show that the proposed hardware architecture achieves over 80 times faster than CPU implementation of NTF. Compared with the implementations on GPGPU, the proposed architecture achieves nearly the same speedup. While the strategy in this paper is applied to GPU platforms, the execution time can be reduced by a half, due to the computational sharing in this paper.

1. Introduction

Higher dimensional array, which is known as a tensor, is prevalent across science and engineering [1]. It is widely applied in the fields of data mining [2], computer vision [3] and image processing [4]. The order of a tensor is defined as the number of dimensions. Specially, a vector and a matrix are the first-order and second-order tensor, respectively [5]. We consider a third-order tensor, also known as three way tensor, is addressed by three indices, as shown in Fig. 1. For a third-order tensor of rank-1, i.e. $G \in \mathbb{R}^{R\times S\times T}$, it can be written as the outer-product of 3 vectors, $G = u^{(1)} \otimes v^{(1)} \otimes w^{(1)}$. Similarly, a rank-$K$ tensor is the sum of K outer-products of 3 vectors, $G = \sum_{j=1}^{K} u^{(j)} \otimes v^{(j)} \otimes w^{(j)}$.

For tensor factorization, the main goal is to decompose the tensor as factorized matrices or tensors and by using such factorized components one can reconstruct the tensor approximately. It is a high-dimensional technique to generate basis elements, which is widely used in feature extraction and dimensionality reduction. Tensor decomposition can be traced back to 1920s, Hitchcock first introduced how to express a tensor with a sum of products. Later in 1944, Cattell raised multi-dimension model applying on psychological theory. During 1960s–1970s, Tucker, Carroll, Chang, Harshman used tensor decomposition method on psychometrics. In 1980s, this method start to be applied and soon prevailed in the field of chemometrics [6]. Over the past decade, this method have been applied to a wide range of fields. Basically, there are two types of tensor factorizations: CANDECOMP/PARAFAC (CP) decomposes a tensor as a sum of rank-1 tensors, which is visualized as Fig. 2, and the TUCKER factorization (Higher-Order SVD, HOSVD) is a higher-order form of principal component analysis (PCA) [7].

In this paper, we mainly focus on CP factorization, which is unique under mild constraints and easy to interpret the decomposed components. Suppose $G \in \mathbb{R}^{R\times S\times T}$ is a three way tensor. To perform CP decomposition means to find $(\min_{\widehat{G}} \parallel G - \widehat{G} \parallel)$ with $\widehat{G} = \sum_{j=1}^{K} u^{(j)} \otimes v^{(j)} \otimes w^{(j)}$, which is illustrated as Fig. 2.

Because the non-negative property is used in a wide variety of applications ranging from document analysis to image processing to bioinformatics, it is natural to impose non-negative constraints on the decomposed terms, which results in non-negative tensor factorization (NTF). Researchers have proposed various methods to solve the NTF computation. Lathauwer [8] presented a higher-order power method to calculate rank-1 tensor factorization. Kolda [9] used shifted power method to calculate eigenvectors of tensors. Shashua [3] proposed a non-negative tensor factorization algorithm and applied it to statistics and computer vision. It treated a set of images as a third-order tensor that is better in handling the spatial redundancy in images.

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

https://doi.org/10.1016/j.mejo.2018.11.006
Received 15 October 2017; Received in revised form 14 October 2018; Accepted 13 November 2018
Available online 16 November 2018

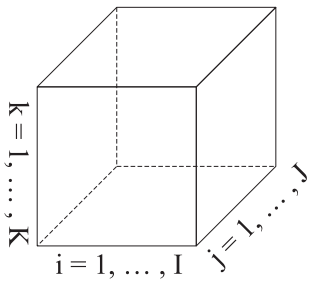


Fig. 1. A third-order tensor.

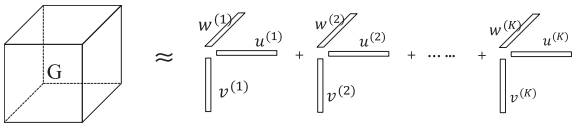


Fig. 2. The illustration of 3-D tensor factorization.

The acceleration of non-negative matrix factorization (NMF) has been implemented on GPU [10,11]. As an extension of NMF, however, NTF is of much higher dimensionality that cause its computations take hours [12]. Fast Algorithms have been proposed to accelerate NTF computations [13]. Some existing works have been proposed to implement NTF on GPU platforms. In Ref. [14], a multi-processor strategy is presented to evaluate the speed of NTF computations. The peak speedup is only 6.8 times with even 10 processor used. In Ref. [12], it proposed to divide the computation into different thread blocks. It achieved a higher speedup than that of CPU platforms. Although it stated that the optimized method is presented, we consider that more improvements can be obtained. In this paper, we propose an improved computational scheme to execute NTF algorithm on hardware. Some of our proposed ideas can also be applied to GPU platforms. In this paper, the GPU-based parallel implementation using the motivated ideas is also given. The contributions of this paper can be summarized as following:

- The proposed hardware architecture can achieve a speedup of 80× compared with NTF implemented on traditional CPU platforms.
- In the proposed architecture, we adopt our improved scheme that produces the components in parallel, while they can only be computed one-by-one in the previous works because of the data dependency.
- By using the proposed methods, the computational complexity decreases. Meanwhile, hardware resources can be also be reduced by hardware sharing in the proposed architecture.

The paper is organized as follows. In Section 2, the overview of an NTF algorithm is presented and analyzed. The challenges of computing the NTF algorithm and corresponding improved methods are provided in Section 2.2. The proposed hardware architecture is described in Section 3 and the improved GPU implementation is discussed in Section 4. In Section 5, we analyze the performance of the proposed architecture. The application results and comparisons are also given. In Section 6 we briefly conclude the work.

2. Algorithm for nonnegative tensor factorization (NTF)

2.1. Original algorithm

Taking a 3-way tensor G as an example, non-negative tensor factorization (NTF) is to obtain an approximation of G by a series of outer-product (Kronecker product) of vectors. As shown in Fig. 2, the outer-product of one set of vector, $u \otimes v \otimes w$, can be used to approximate the original tensor G. It could be more precise by using K sets of vectors. The formulation of tensor factorization of a 3-way tensor G is given as Eq. (1), where $u^{(j)}$ denotes one of the decomposed vector in the $j-$th set of component, as shown in Fig. 2.

$$\widetilde{G} = \sum_{j=1}^{K} u^{(j)} \otimes v^{(j)} \otimes w^{(j)} \tag{1}$$

$\widetilde{G}$ is an approximation to original tensor G. It is obtained by solving a least-square problem in Eq. (2) with non-negative constraints, where $\| \cdot \|_F$ represents F-norm. When K is set to 1, the procedure is similar and is also widely adopted in various applications.

$$\begin{cases} \min\limits_{(u,v,w)} \dfrac{1}{2} \left\| G - \sum\limits_{j=1}^{K} u^{(j)} \otimes v^{(j)} \otimes w^{(j)} \right\|_F^2 \\ \text{subject to}: u^{(j)}, v^{(j)}, w^{(j)} \geq 0 \end{cases} \tag{2}$$

There are various methods to solve the optimization problem in Eq. (2). In Ref. [3], the solution is obtained by gradient descent method iteratively. The convergent property has been proved in Refs. [3,15]. The components u, v, and w are updated iteratively, as shown in Eq. (3).

$$u_r^{(j)} \leftarrow \frac{u_r^{(j)} \sum_{s,t} G_{r,s,t} v_s^{(j)} w_t^{(j)}}{\sum_{m=1}^{K} u_r^{(m)} < v^{(m)}, v^{(j)} >< w^{(m)}, w^{(j)} >} \tag{3a}$$

$$v_s^{(j)} \leftarrow \frac{v_s^{(j)} \sum_{r,t} G_{r,s,t} u_r^{(j)} w_t^{(j)}}{\sum_{m=1}^{K} v_s^{(m)} < u^{(m)}, u^{(j)} >< w^{(m)}, w^{(j)} >} \tag{3b}$$

$$w_t^{(j)} \leftarrow \frac{w_t^{(j)} \sum_{r,s} G_{r,s,t} u_r^{(j)} v_s^{(j)}}{\sum_{m=1}^{K} w_t^{(m)} < u^{(m)}, u^{(j)} >< v^{(m)}, v^{(j)} >} \tag{3c}$$

In Eq. (3), $u_r^{(j)}$ represents the $r-$th element of $u^{(j)}$. $\sum_{s,t} G_{r,s,t} v_s^{(j)} w_t^{(j)}$ represents a tensor-vector product. To clearly express the algorithm, all the inner products in the denominator of Eq. (3) can be calculated and stored in three matrices with a size of $K \times K$ respectively, which is named as M_u, M_v, and M_w. For example, $< u^{(m)}, u^{(j)} >$ is stored in $M_u(m, j)$.

$$M_u = \begin{bmatrix} < u^{(1)}, u^{(1)} > & \cdots & < u^{(1)}, u^{(k)} > \\ < u^{(2)}, u^{(1)} > & \cdots & < u^{(2)}, u^{(k)} > \\ \vdots & \ddots & \vdots \\ < u^{(k)}, u^{(1)} > & \cdots & < u^{(k)}, u^{(k)} > \end{bmatrix}$$

Therefore, the procedure of computing the component u and v is shown in Algorithm 1. To simplify the representation of the expression, in Algorithm 1, the $u_r^{(j)}$ is given by u_r, with the elimination of the symbol j. Since we optimize the computation scheme for every two consecutive components and all the computations for every two consecutive components follow the same pattern, only the first two components computation, i.e u, and v, are given in Algorithm 1 and Algorithm 2. Through this simplification, the hardware resources and computation sharing strategy we adopt can be observed more clearly.

Algorithm 1 Original algorithm for nonnegative tensor factorization.

Input: G, u, v, w
Output: u, v, w

for *the k-th iteration* **do**
  Load original tensor data G
  for $r \in \{1, 2, \cdots, R\}$ **do**
    for $s \in \{1, 2, \cdots, S\}$ **do**
      $\alpha_{rs} = \sum_t G_{r,s,t} w_t$
    end
    $u_r \leftarrow \frac{u_r \sum_s \alpha_{rs} v_s}{\sum_{m=1}^{K} u_r M_v(m,j) M_w(m,j)}$
  end
  Load original tensor data G
  for $s \in \{1, 2, \cdots, S\}$ **do**
    for $r \in \{1, 2, \cdots, R\}$ **do**
      $\beta_{rs} = \sum_t G_{r,s,t} w_t$
    end
    $v_s \leftarrow \frac{v_s \sum_r \beta_{rs} u_r}{\sum_{m=1}^{K} v_s M_u(m,j) M_w(m,j)}$
  end
end

2.2. Computation-shared in NTF

Our Goal is to propose a flexible hardware architecture that can execute the NTF algorithm more efficiently. The execution time on hardware is expected to outperform software implementations. To demonstrate our ideas more explicitly, we first analyze the bottlenecks of the NTF computations and then propose the strategies for building up an efficient NTF hardware architecture.

By analyzing Eqs. (3a)–(3c), the main challenges are summarized as follows:

1) Considering the original tensor and the decomposed components are possible to be in large size, it requires a large amount of memories and communication bandwidth. Thus, the computation must be divided into blocks. In Ref. [12], a method based on block dividing on GPU platforms is proposed. It can also be applied to CPU, FPGA or other platforms. Taking 3-way NTF as an example, the block-based computation is shown as Eq. (4), where t_1, t_2 are two specified block size value for component w.
2) The computation of each component u, v, and w are dependent on each other. To calculate v, the component u in Eq. (3b) is derived from the new updated results in Eq. (3a). In the original algorithm, the computation of v would begin after the computation of u has been completed. Thus, the components are produced one-by-one in previous works [12].
3) The computational complexity of the algorithm is large. Meanwhile, the tensor data have to be loaded each time when computing a new update of each component. For example, the tensor data are used three times in 3-way case. It will increase the requirements of memory bandwidth during the computation.

In order to overcome the challenges in the original algorithm, we propose an efficient method, as shown in Algorithm 2. In Algorithm 1 and 2, only two components are shown. In the improved algorithm, the numerators of the first component are calculated, while some intermediates are stored for the second component computation. By using the intermediates in the first component, the computations for the second component are much easier than that in the original algorithm.

Algorithm 2 Improved algorithm for sharing data and reducing memory accessing.

Input: G, u, v, w
Output: u, v, w

for *the k-th iteration* **do**
  Load original tensor data G
  for $r \in \{1, 2, \cdots, R\}$ **do**
    for $s \in \{1, 2, \cdots, S\}$ **do**
      $\alpha_{rs} = \sum_t G_{i,s,t} w_t$
    end
    $u_r \leftarrow \frac{u_r \sum_s \alpha_s v_s}{\sum_{m=1}^{K} u_r M_v(m,j) M_w(m,j)}$
  end
  for $s \in \{1, 2, \cdots, S\}$ **do**
    $\beta_s = \sum_r \alpha_{rs} \cdot u_r$
    $v_s \leftarrow \frac{v_s \beta_s}{\sum_{m=1}^{K} v_s M_u(m,j) M_w(m,j)}$
  end
end

$$\sum_t [\sum_s [G_{s,t} v_s^{(j)}] w_t^{(j)}] = \sum_{t_1} [\sum_{s_1} [G_{s_1,t_1} v_{s_1}^{(j)}] w_{t_1}^{(j)}] + \sum_{t_1} [\sum_{s_2} [G_{s_2,t_1} v_{s_2}^{(j)}] w_{t_1}^{(j)}] + \cdots \quad (4)$$

As shown in Fig. 3, a 3-D tensor is sliced in different ways when computing the components. A slice is used to calculate one element of the first component u. The intermediates can be used to compute a partial result of the second component v.

Compared with the original algorithm, the results are the same as that in the original algorithm. The differences are that time complexity is smaller and hardware resources can be saved by using the cached intermediates. Especially based on the hardware architecture strategies, such as parallelism and pipeline flow, more advantages in the improved method can be achieved to address the challenges.

- The computational complexity is less than the original algorithm. The complexity for the first component u is the same with only one additional operation added into the loop, but the complexity for the second component v is much less than the original algorithm. In the improved algorithm, there is only one loop for the computation of second component, while there are nested loops in the original algorithm. This improvement is then applied to the third component w and the first component u in the following iteration. Meanwhile, for the hardware architecture, some specified strategies can be applied to improved the speed, such as parallelism and pipeline flow.
- From algorithm 2, the tensor data are loaded only once for two components computation and the cached intermediates generated in the first component computation are reusable, which reduces the mem-

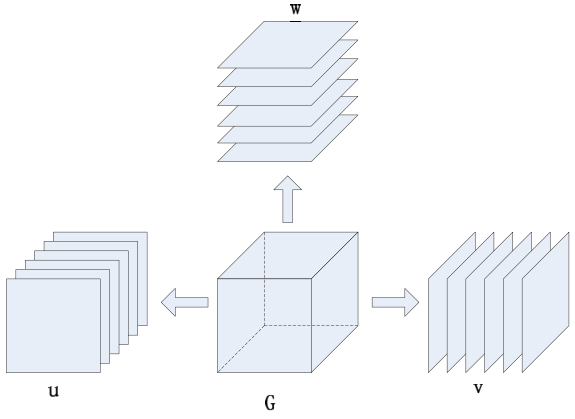


Fig. 3. The slicing of 3-way tensor for 3 components computation.

Table 1
The output sequence of the two parts.

Round	1	2	3	4	5	6
Output	$u^{(1)}$	$w^{(1)}$	$v^{(2)}$	$u^{(3)}$	$w^{(3)}$	$v^{(4)}$
	$v^{(1)}$	$u^{(2)}$	$w^{(2)}$	$v^{(3)}$	$u^{(4)}$	$w^{(4)}$

ory bandwidth requirement. Meanwhile, the hardware resources for the second component computation will be obviously much less because of the compacted computations.

- Two components computations can work in parallel. During the computations of the first component, some intermediates are reorganized and used for the computation of the second component. The second component can be produced once the first component.

Throughout the paper, we call the parallel computations of every two consecutive components as a round. As shown in Table 1, in each round, i.e. in each column, the upper output is called the first component and the lower called the second component. In the first round, the component u and v are produced, while in the second round, the component w and component u of the next iteration are calculated.

Pipeline and scalability are very important aspects in a hardware architecture. Based on the improved algorithm, it is also necessary to consider how to design the details to execute the improved algorithm. The corresponding architecture is shown in the next section.

3. Hardware architecture

To illustrate the proposed architecture more explicitly, we consider a video background estimation problem [16], where the video is considered as a third-order tensor. We analyze an example of video background estimation in which 256 frames are processed as a group and a 32 × 32 block is tiled in a frame. Thus, the original tensor data G is of size 32 × 32 × 256. Performing a NTF on such tensor returns three vectors, the last of which represents the variation of time domain. The results and analysis would be shown in the next section. Here, we first present the architecture for NTF.

Compacted from Eq. (3), the formulation of rank-1 NTF is rewritten as Eq. (5), which is adopted in various applications, such as video background estimation.

$$u_r \leftarrow \frac{\sum_{s,t} G_{r,s,t} v_s w_t}{< v, v >< w, w >} \tag{5a}$$

$$v_s \leftarrow \frac{\sum_{r,t} G_{r,s,t} u_r w_t}{< u, u >< w, w >} \tag{5b}$$

$$w_t \leftarrow \frac{\sum_{r,s} G_{r,s,t} u_r v_s}{< u, u >< v, v >} \tag{5c}$$

The challenges to compute the rank-1 NTF in (5), have been described in the previous sections. The proposed architecture discussed in the following part could address them. It can handle any size of tensor data.

3.1. Top-level architecture

The proposed top-level architecture for the improved algorithm is shown in Fig. 4. The whole architecture consists of four parts: '*U*_1', '*U*_2', '*U*_3', and Control Unit (CU). '*U*_1' computes the numerators of the first component *u*. In '*U*_1', the computations for eight elements $\{u_1, u_2, \ldots, u_8\}$ are parallel. The results of '*U*_1' are sent to '*U*_2', which calculates the denominator and produces final results of the first component u in the first round.

Meanwhile, the intermediate results in the '*U*_1' are sent to '*U*_3'. Along with the updated *u*, it can produce the second component *v*. '*U*_3' acts the same roles as '*U*_1' and '*U*_2', but the internal structures are much simpler. In this proposed architecture, the two data-dependent computations can work simultaneously, which can reduce the computational complexity of the whole process.

Based on the improved algorithm and corresponding architecture, two components can be produced in one round. Since there are three components in our case and the calculated components are fed back to '*U*_1', '*U*_2', and '*U*_3' to update themselves in the next iteration, the produced components are needed to be stored in corresponding memories. Meanwhile, the arrangements of feedback are also essential. All these works are executed by CU (Control Unit).

3.2. Architecture of 'U_1'

The internal structure of '*U*_1' is shown in Fig. 5. The numerators in Eq. (5) are computed by '*U*_1'. Considering the hardware resources in a single FPGA chip, we design eight pipelines to work simultaneously,

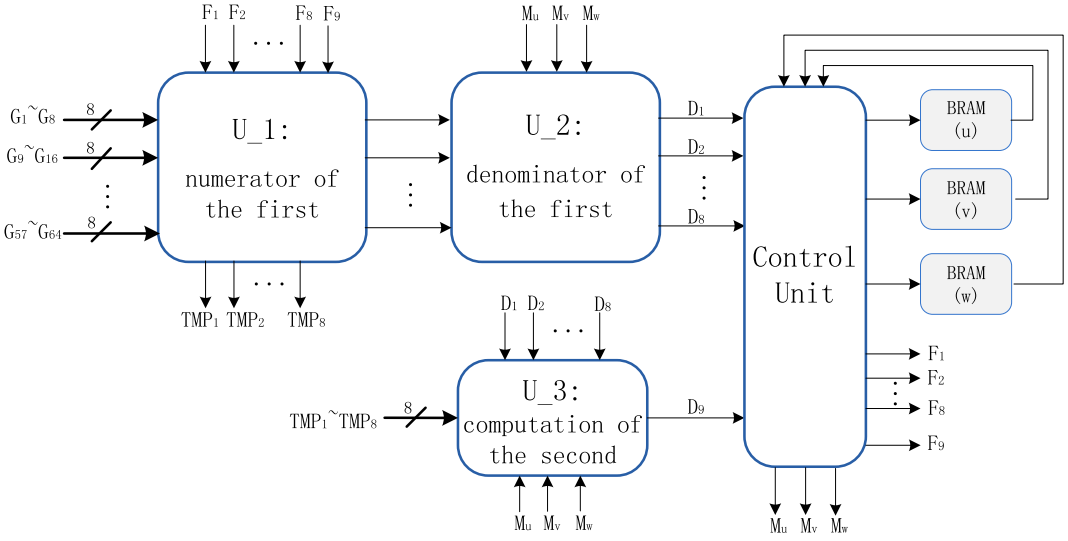


Fig. 4. The slicing of 3-way tensor for 3 components computation.

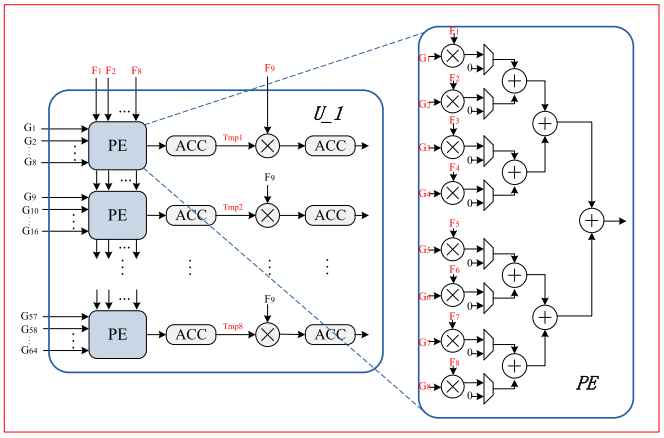


Fig. 5. The internal structure of *U_1* in top architecture.

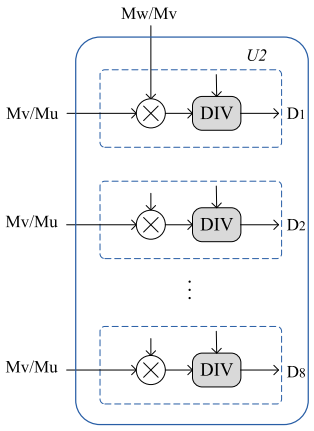


Fig. 6. The internal structure of *U_2* in top architecture.

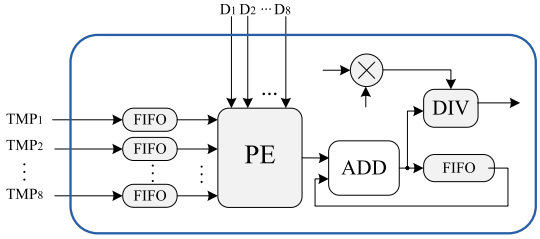


Fig. 7. The internal structure of U_3 in top architecture.

each of which calculates a part of numerator of specific component in Eq. (5). Taking the first component u as an example, in the first pipeline, PEs (Processing Elements) execute multiply-accumulate operations with eight tensor data $\{G_{1,1,i+1}, G_{1,1,i+2}, \ldots, G_{1,1,i+8}\}$ and component w elements $\{w_{i+1}, w_{i+2}, \ldots, w_{i+8}\}$. The results of PEs are then multiplied with another components $\{v_{i+1}, v_{i+2}, \ldots, v_{i+8}\}$. Since there are eight pipelines, '*U_1*' can process 64 tensor data each time. In the end, it generates the numerator of $\{u_{i+1}, u_{i+2}, \ldots, u_{i+8}\}$ in the first round.

In Fig. 5, eight PEs are employed, each of which processes eight elements of a tensor. Therefore, "U_1" produces the intermediate results for a sub-block of tensor. To get the summation result in Eq. (5), two accumulators (ACCs) are utilized to produce the summation of intermediate results of sub-block tensor. Two accumulators are used in tensor-vector-multiplication for two vectors respectively.

The internal structure of a PE is shown in the right block of Fig. 5. Besides the multiplier-accumulator unit, there are also multiplexers in it. Since the computation is divided into blocks of size eight, the multiplexers are used to select corresponding multiplier-accumulator units for the last block if its size cannot be divided eight exactly.

The contents of feedback components $\{F_1, F_2, \ldots, F_8\}$ and F_9 to PEs varies in different round. $\{F_1, F_2, \ldots, F_8\}$ and F_9 are w and v in the first round. In the second round, they are v and u. $\{F_1, F_2, \ldots, F_8\}$ and F_9 are w and v again in the third round. This is realized by CU (Control Unit), which will be described in the following.

3.3. Architecture of 'U_2'

The internal structure of computing unit '*U_2*' is shown in Fig. 6. It generates the denominators, which are used to produce the first updated component. In this paper, we use rank-1 NTF as a demo, so the denominator in the formulation is the product of the two values. Taking the first component as an example, the CU (Control Unit) is responsible for computing $< v, v >$ and $< w, w >$. Corresponding to the eight pipelines in the '*U_1*', there are also eight dividers in the '*U_2*' to update eight elements of the first component in each clock cycle.

3.4. Architecture of 'U_3'

The internal structure of computing unit '*U_3*' is shown in Fig. 7. It computes the second component by using the intermediates from '*U_1*'. The intermediates are the data after one dimension reduction. Since the slicing of the original 3-D tensor data for the second component is different from the first one, the intermediates cannot be used for computing the second component in the same order as the first one. Thus, the intermediates are first stored in FIFO. After the computation of the first component is completed, they are feedback to the '*U_3*' to calculate a part of each element in the second component. By applying this strategy, the hardware resources for the second component computations are much less than the first component. Meanwhile, the second component can be produced sequentially once the computations of the first component are completed, while in the original algorithm the two components are calculated one by one.

In the computations for the third component w, both u and v should be newly updated previously. The fact means that the computations for w cannot be executed in parallel with u and v in the same iteration, but the proposed architecture can still work for the w in the current iteration and u in the next iteration, as shown in Table 1.

3.5. Control unit

In the proposed architecture, the CU (Control Unit) is responsible for generating control signals and managing memory. Since there are three memories, each of which is used to stored one component, the two parallel results should be allocated into the correct memories by CU. The memory management scheme is shown in Fig. 8. In the 'Round 1' (described in Table 1), the updated results are buffered in the memories for u and v, while the v and w are fed back to the computation units. As Fig. 8 shows, it works in the same way for the other rounds.

In "*U_1*", 8 PEs work simultaneously, each of which processes 8 tensor data at a time. Since all tensor data are loaded once in one round, it takes $(R^*S^*T/64)$ clock cycles for the proposed architecture to produce the first component. Once an element of the first component is produced, it is fed to "*U_3*" to compute all the elements of the second component. Given u and v are the first and second component, it takes $(R^*S/8)$ clock cycles to produce the second component, where R and S are the size of two components. As shown in Table 1, 2 iterations are completed in 3 rounds, and it takes $(R^*S^*T/64) + (R^*S/8)$, $(R^*S^*T/64) + (R^*T/8)$, and $(R^*S^*T/64) + (S^*T/8)$ clock cycles for the round 1, 2, and 3. The time complexity in the following rounds is in the same way.

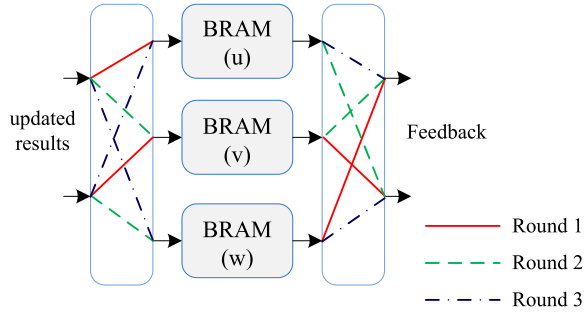


Fig. 8. The memory management in the proposed architecture.

4. GPU implementations

4.1. Parallelizing NTF algorithm with CUDA

Algorithm 3 Proposed GPU-based NTF Algorithm (one round).
Input: G, u, v, w
Output: u, v, w
Intermediate: α, M_w
1: //Copy G, u, v, w from CPU memory to GPU global memory d_G, d_u, d_v, d_w
2: **cudaMemcpy** ({d_G, d_u, d_v, d_w}, {G, u, v, w}, size of the data, cudaMemcpyHostToDevice)
3: **for** the $k-th$ iteration **do**
4: //Launch *GPU* kernel 1
5: **Kernel_1** <<< $dimGrid, dimBlock\_1$ >>> (d_G, d_u, d_v, d_w, d_α, d_M_w)
6: **cudaThreadSynchronize**()
7: //Launch *GPU* kernel 2
8: **Kernel_2** <<< $dimGrid, dimBlock\_2$ >>> (d_u, d_v, d_α, d_M_w)
9: **cudaThreadSynchronize**()
10: **end for**

Algorithm 4 GPU Kernel 1.
Input: G, v, w
Output: d_u, d_α, d_M_w
1: calculate M_v and M_w
2: $d\_M_w \leftarrow M_w$
3: **_syncthreads**()
4: $\alpha_{r,s} = \sum_t G_{r,s,t} w_t$
5: $d\_\alpha_{r,s} \leftarrow \alpha_{r,s}$
6: **_syncthreads**()
7: $u_r = \frac{\sum_s \alpha_{r,s} v_s}{M_v M_w}$
8: $d\_u_r \leftarrow u_r$

Algorithm 5 GPU Kernel 2.
Input: d_u, d_α, d_M_w
Output: d_v
1: $u_r \leftarrow d\_u_r$
2: calculate M_u
3: $M_w \leftarrow d\_M_w$
4: $\alpha_{r,s} \leftarrow d\_\alpha_{r,s}$
5: **_syncthreads**()
6: $\beta_s = \sum_r \alpha_{r,s} u_r$
7: $v_s = \frac{\beta_s}{M_u M_w}$
8: $d\_v_s \leftarrow v_s$

The proposed GPU-based NTF algorithm is shown in Algorithm 3. We take the first round as an example where the vectors u and v need to be updated. Because the new u is required in order to calculate v, two GPU kernels (Algorithm 4 and Algorithm 5) are designed for updating u and v respectively and also implementing global synchronization. There is only one-round data transmission for the tensor data and initial u, v, w data between CPU memory and GPU global memory.

4.2. Thread allocation and memory access

For kernel 1, we allocate N × N thread blocks where N is determined by GPU Compute Capability, and one slice of G is divided into N × N tiles that are computed by one thread block. Different blocks deal with corresponding slices in tensor value G. For kernel 2, the number of threads equals to the length of vector u.

In order to avoid bank conflict by accessing values on the same column of G, we allocate N × (N+1) shared memory for G and intermediate results where N is block dimension. The tensor value G is *unsigned char* type. To maximize data transfer rate when accessing G in global memory, each thread deals with one byte (four G elements) in a coalesced access manner.

5. Performances evaluation and results

5.1. Case study: background estimation

Tensor Factorization can be also applied to background estimation of a video with moving object in the scene. Taking a gray video as an example, the data are represented by a 3-way tensor with the size of $R \times S \times T$, where R and S are video frame size, and T is the number of frames to be analyzed. By applying tensor factorization to the video, the three components, u, v, and w represents the scene changing in different dimensions, where w indicates moving object or still background.

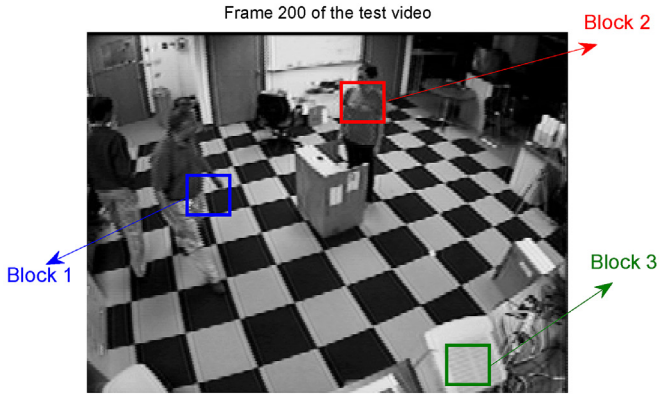
Fig. 9 presents an example of video background estimation using tensor factorization. The 200-th frame of the test video is shown in Fig. 9(a). Three typical blocks are selected to be analyzed. 'Block 1' is the region of fast moving object, while the objects stay still at region 'Block 2' after it enters the scene. In region 'Block 3', it is the still background in the video.

In Fig. 9(c)–(e), the last decomposed vector of NTF results, w, in different region is shown. The locations of moving objects are represented by a vector, instead of a video. The elements in the vector indicate when the object enters or exits the regions. The value keeps almost constant if there is no moving objects, or it goes down while the objects appear or disappear in the regions. It is easier to determine the locations of objects by analyzing the vector. After identifying the location of objects, the entire background can then be reconstructed. In this example, the fast moving object moves into 'Block 1' in the 200-th frame, while another object enter in 'Block 2' in the 150-th frame and staying still. In the 'Block 3', the value of w and the corresponding curve in Fig. 9(e) almost keeps constant, and thus it is considered as background all the time.

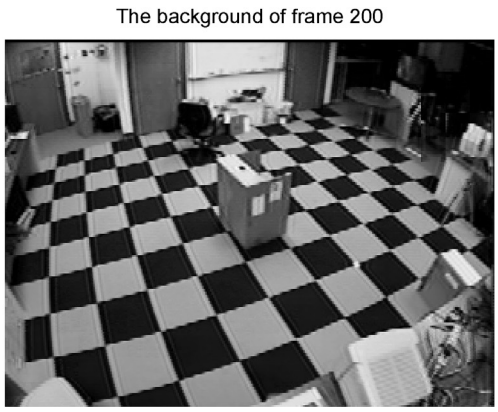
The estimated background is shown in Fig. 9(b). The static regions are preserved in the estimated background, and different types of moving objects are removed from the video.

5.2. Case study: facial images

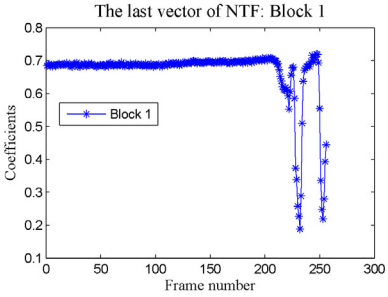
For gray facial images, if we stack the training images as a cube, a 3-way tensor can be constructed. After NTF, we can get u, v and w. The outer product of u and v are considered as basis facial image. In the inference stage, when new images come, they are calculated with the basis vector u and v by tensor-vector-multiplication, and thus new images are projected on the basis facial images, which can be called the coefficients. Giving that K is 50, the obtained 50 coefficients for each images are called the representation of image features, which will be fed to an SVM classifier for classification.



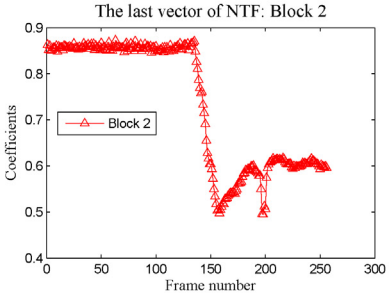
(a) The 200-*th* frame of the test video.



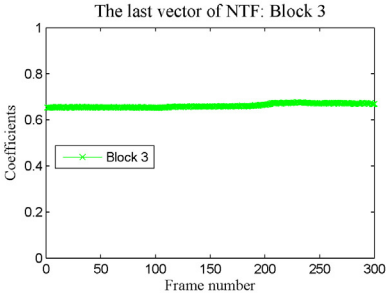
(b) The estimated background of frame 200 in the test video.



(c) The coefficients of the last vector of NTF in Block 1



(d) The coefficients of the last vector of NTF in Block 2



(e) The coefficients of the last vector of NTF in Block 3

Fig. 9. An example of NTF application to video background. (a). The original 200-th frame; (b). The estimated video background. (c)–(e).the tensor factorization results, *w*, representing the scene changing in three block region in (a).

We used the MIT CBCL face set to recover the factors. The measurement vector is calculated by the inner-product between the factors and the facial or non-facial image. SVM classifier is used to categorize positive (faces) and negative (non-faces) examples. We varied the kernel of the SVM from linear to polynomial of degree five to RBF and recorded the percentage of correctness over a test set. The training and testing was conducted on a leave one out paradigm where the facial and non-facial images are selected as training set and test set.

In each trial, different training and testing subsets were used and the results were averaged over the trials. We used 50 NTF factors and 50, 20 and 6 NMF factors in three separate trials and also used PCA factors for comparison. The measurements from the NTF factors generated the highest classification accuracy compared to 50 NMF factors which contain a 20-fold space increase.

Table 2 presents the classification results of the measurements by SVM classifier. It demonstrates that the classification result obtained by using NTF is more accurate than using matrix factorization and PCA method. To compare these methods, for NMF method, different number of factors, such as 6, 20, and 50, are taken as example, which is corresponding to K in Eq. (2) and Fig. 2. The results in Table 2 shows that NTF achieves the best accuracy under the same number of factors. In this example, only the facial images are used to generate the 50 factors. Fig. 11 presents the comparisons between the original facial images and the reconstructed faces using 20 and 50 factors.

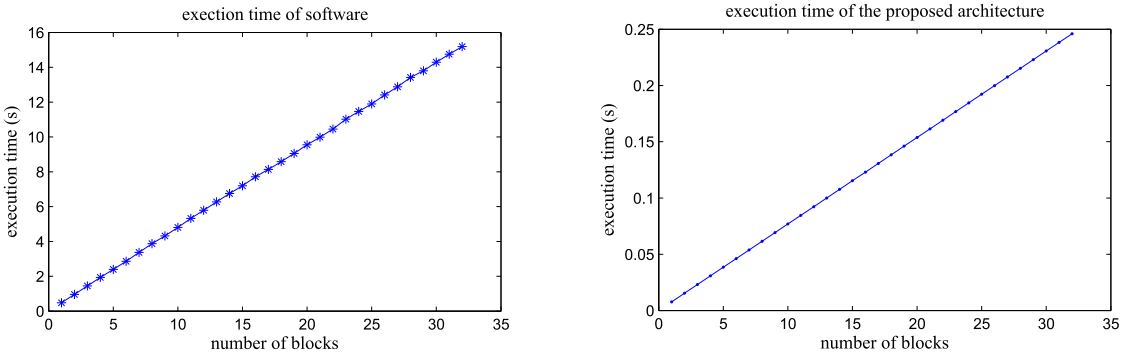
Table 2
The correct percentages of the test facial and non-facial images by the classifier with linear, polynomial and RBF kernels. NTF is only applied to facial images.

	linear	poly d = 5	RBF
NTF(50)	93.8%	95.8%	97.6%
NMF(50)	91.6%	94%	95%
NMF(20)	87.5%	90.1%	89%
NMF(6)	83.2%	84.3%	86%
PCA	90.8%	94%	91.7%

5.3. Speed

Given that the size of 3-way tensor to be decomposed is $R \times S \times T$, the time complexity of the proposed architecture is $O(R \times S \times T/N)$, where $R \times S \times T$ is the size of a tensor, and N is parallelism factor of our proposed architecture. In our design, N is (64 × 2), where ”64” is derived from the number of PEs we used in the architecture, and ”2” originate from the computation sharing.

Here, the comparisons of execution time between the proposed architecture and software implementation are presented. For the software implementation, we program in C language following the flow of Algorithm 1. It is run on the Intel Xeon CPU X5675 @3.07 GHz with installed memory 12 GB. We measure the execution time of software on CygWin Linux. The hardware architecture is simulated on Xilinx Virtex-6 xc6vlx760 FPGA.



(a) Execution time of the software

(b) Execution time of the proposed hardware architecture

Fig. 10. Execution time comparison between software and the proposed architecture.

In the comparison, we run 100 iterations in both hardware architecture and software implementation. Since 32 × 32 × 256 is selected as block size in our design, we evaluate the speed on different frame size of video, from 32 × 32 to 512 × 512.

As shown in Fig. 10, under different numbers of blocks, the execution time of the proposed architecture in this paper is about 80 times less than software implementations. As the hardware architecture processes a tensor block-by-block, the number of blocks actually represents the size of tensor to be calculated. We also implement 2 GPU versions to illustrate the improvement by using our method. The GPU implementation was run on NVIDIA Tesla C2075 at 575 MHz. For the first one, we follow the flow of algorithm 1 and for the second, we implement it as Section 4 mentioned. From Fig. 12, the execution time of our improved GPU version is about 1.6× faster than the original non-optimized algorithm.

5.4. Area

Table 3 shows the hardware resource usage of the proposed architecture based on Xilinx Virtex-6 xc6vlx760 FPGA. The architecture consumes about 10% of available resources of the chip. As this proposed hardware architecture is the first one for non-negative tensor factorization, it is compared with non-negative matrix factorization (NMF) in terms of hardware resource usage, as shown in Table 3. Through this table, we can observe that, though NTF is an 3D extension of NMF and thus more computational complex, the resource usage will not increase linearly with the computational complexity increasing and still comparable.

Fig. 11. The facial reconstruction using NTF. Top to Bottom: reconstruction using 20 NTF factors, reconstruction using 50 NTF factors, and the original facial images.

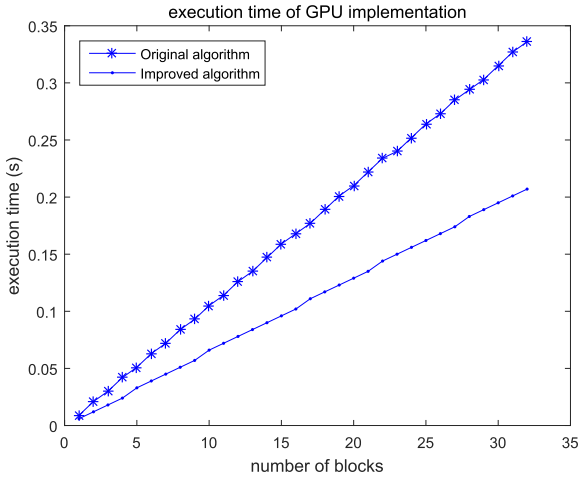


Fig. 12. Execution time comparison between original and improved algorithm.

5.5. Comparison with previous works

Table 4 presents the time speedup of the proposed tensor architecture, which is about 80×. Though NMF is far less computational intensive than the NTF algorithm, in Ref. [17], the speedup was claimed to be 1.53 compared to its CPU implementation and 23.9 to its ARM implementation, which is far less than our improvement on NTF. When NTF is implemented on GPGPU, it accelerates the computations by 60–100 times. From Table 3, although the hardware resource of [17] is about one-third of our proposed architecture, the speedup is over three times higher than [17], which demonstrates the efficiency the proposed architecture.

In Ref. [14], a multi-processor implementation is proposed. It can complete the calculation with 100 iterations in 290 s for a 600 × 400 × 200 data set, while it takes about 2.5 s to compute a 512 × 512 × 256 data set on the proposed architecture in this paper.

Table 3
Resource Usage of the proposed architecture.

	proposed tensor architecture			[17]
	Used	Available	Utilization	Used
Occupied Slices	12,057	118,560	10%	–
LUTs	39,506	474,240	8%	13,710
Registers	28,705	948,480	3%	16,871
RAMB36E	9	720	1%	71
RAMB18E	41	1440	2%	
DSP48E	117	864	13%	96
Max Frequency	82.7 MHz			100 MHz

Table 4
The comparison of time speedup with existing matrix hardware architecture and tensor GPU accelerator.

	proposed hardware FPGA	NMF hardware [17]	NTF GPU [12]	Our NTF GPU
speedup	80	1.53–23.9	60–100	80

Thus, compared with [14], our architecture achieves more than 100 times speedup.

In Ref. [12], an GPU-based method is proposed to accelerate the NTF computation. Though it claimed the speedup over single-core CPU are around 60–100 for small data set. When it processes larger data set, i.e. 600 × 400 × 200, it costs 2.36 s. Compared with [12], the proposed hardware architecture implemented on FPGA can achieve about 80 times speedup and it takes about 2.5 s to compute a larger data set, i.e. 512 × 512 × 256. It is not possible to compare our GPU implementation with [12] in terms of absolute time consumption, since the processors are different and it is impossible to imitate all circumstances of the measurement. Moreover, different CPU and different software implementation will also affect the speedup value slightly. However, from the trend shown in Fig. 12, It is reasonably expected that the implementation in Ref. [12] can run faster if our ideas is adopted.

6. Conclusion

This paper has presented an improved method and corresponding hardware architecture for non-negative tensor factorization algorithm. By using the improved method on hardware platforms, the computations can work in parallel and the hardware resources usage can be saved. Compared with traditional CPU platforms, the proposed architecture has achieved a significant speedup.

The proposed method can also achieve a better performance on GPU platform. The ideas in this paper can be applied in various computation platforms, not only based on hardware architectures, but also multi-core CPU or GPU platforms.

In this paper, the rank of tensor is set as 1 to simplify the demonstration of the proposed architecture. For a more general situation where the rank of tensor is K, the computations follow the same pattern and are dependent. Therefore, the proposed method in this paper can be adopted in general applications. As the major contribution of this paper is to utilize the shared computations by increasing the cost of acceptable number of memories, the methods can also be used in other platforms, such as multi-core CPUs.

Acknowledgement

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

References

[1] T.G. Kolda, B.W. Bader, Tensor decompositions and applications, SIAM Rev. 51 (3) (2009) 455–500.
[2] E. Acar, S.A. Camtepe, B. Yener, Collective sampling and analysis of high order tensors for chatroom communications, in: S. Mehrotra, D. Zeng, H. Chen, B. Thuraisingham, F.-Y. Wang (Eds.), Intelligence and Security Informatics, Vol. 3975 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 213–224.
[3] A. Shashua, T. Hazan, Non-negative tensor factorization with applications to statistics and computer vision, in: Proceedings of the 22nd International Conference on Machine Learning, ICML '05, ACM, New York, NY, USA, 2005, pp. 792–799.
[4] A. Shashua, A. Levin, Linear image coding for regression and classification using the tensor-rank principle, in: Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on, vol. 1, 2001, I–42 – I–49 vol. 1.
[5] T.G. Kolda, B.W. Bader, Tensor decompositions and applications, SIAM Rev. 51 (3) (2009) 455–500.
[6] T.G. Kolda, B.W. Bader, Tensor decompositions and applications, SIAM Rev. 51 (3) (2009) 455–500.
[7] B.N. Sheehan, Y. Saad, Higher order orthogonal iteration of tensors (hooi) and its relation to pca and glram, in: Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA, SIAM, 2007.
[8] L.D. Lathauwer, B.D. Moor, J. Vandewalle, On the best rank-1 and rank-(r1,r2,…,rn) approximation of higher-order tensors, SIAM J. Matrix Anal. Appl. 21 (4) (2000) 1324–1342.
[9] T.G. Kolda, J.R. Mayo, Shifted power method for computing tensor eigenpairs, SIAM J. Matrix Anal. Appl. (2011) 1095–1124.
[10] V. Kysenko, K. Rupp, O. Marchenko, S. Selberherr, A. Anisimov, Gpu-accelerated non-negative matrix factorization for text mining, in: International Conference on Applications of Natural Language Processing and Information Systems, 2012, pp. 158–163.
[11] E. Mejía-Roa, D. Tabas-Madrid, J. Setoain, C. García, F. Tirado, A. Pascual-Montano, Nmf-mgpu: non-negative matrix factorization on multi-gpu systems, BMC Bioinf. 16 (1) (2015) 43.
[12] J. Antikainen, J. Havel, R. Josth, A. Herout, P. Zemcik, M. Hauta-Kasari, Nonnegative tensor factorization accelerated using gpgpu, IEEE Trans. Parallel Distr. Syst. 22 (7) (2011) 1135–1141.
[13] G. Zhou, A. Cichocki, S. Xie, Fast nonnegative matrix/tensor factorization based on low-rank approximation, IEEE Trans. Signal Process. 60 (6) (2012) 2928–2940.
[14] Q. Zhang, M. Berry, B. Lamb, T. Samuel, A parallel nonnegative tensor factorization algorithm for mining global climate data, in: Computational Science-ICCS 2009, Vol. 5545 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 405–415.
[15] T. Hazan, S. Polak, A. Shashua, Sparse image coding using a 3d non-negative tensor factorization, in: International Conference of Computer Vision, ICCV, 2005, pp. 50–57.
[16] A. Cichock, R. Zdunek, A.H. Phan, S. ichi Amari, Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-way Data Analysis and Blind Source Separation, Wesley, 2009.
[17] L. Cerina, P. Cancian, G. Franco, M.D. Santambrogio, A hardware acceleration for surface emg non-negative matrix factorization, in: Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International, IEEE, 2017, pp. 168–174.